Broadview®
www.broadview.com.cn

iOS开发进阶

唐巧 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

iOS开发进阶

唐巧 著

電子工業出版社
Publishing House of Electronics Industry

内容简介

本书分工具、实践、理论三大部分。第一部分介绍 iOS 开发的常用工具，第二部分介绍 iOS 开发中的一些常见的实践经验，第三部分介绍 iOS 开发中涉及的原理。

如果把成为 iOS 开发高手的过程比作武侠小说中的修炼过程的话，工具、实践和理论的学习就分别对应兵器、招式和内功的修炼。本书希望通过这三方面的综合讲解，全面提高开发者的技能水平。

图书在版编目（CIP）数据

iOS 开发进阶 / 唐巧著. — 北京：电子工业出版社，2015.1
ISBN 978-7-121-24745-3
I. ① i…II. ① 唐…III. ① 移动终端－应用程序－程序设计 IV. ① TN929.53

中国版本图书馆 CIP 数据核字（2014）第 268564 号

责任编辑：徐津平
印　　刷：北京丰源印刷厂
装　　订：三河市鹏成印业有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编 100036
开　　本：787×980　1/16　印张：16　字数：328 千字
版　　次：2015 年 1 月第 1 版
印　　次：2015 年 1 月第 1 次印刷
定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推荐序

人生如长跑，成长如进阶

曾经有很多次想提笔写一本纯粹的技术图书，有几次甚至已经把目录做完了，但是看到“层峦叠嶂”的目录结构，我总是心生恐惧。一想到要耗费一年多的业余时间去写那些枯燥无味的技术文字，并逐一核对各种技术细节，我仿佛置身幽暗的森林而无法自拔。计划还未开始，就已经结束了。

总有人会去做这些费时费力并能够惠及大众的事情，他们像一个个沉默的跑者，在奔跑的同时，不停地标出路线，记录经验，传播技巧，并吸引更多的人加入跑步的队伍一起成长。唐巧就是其中之一。

我很早就认识了唐巧，那时他还是一个初入江湖的“小球”，高高瘦瘦，正在网易有道做云笔记的开发。短短几年之内，唐巧获得了飞速的成长，他不仅是一个优秀的 iOS 开发者，而且成为 iOS 领域的布道者。他从 2011 年开始构建独立博客“唐巧的技术博客”，撰写了大量的 iOS 开发相关的文章，成为知名 iOS 博主。2012 年唐巧离开网易有道，加入猿题库，成为一个创业者。期间他组建了由 iOS 开发领域顶尖开发者构成的“真 · iOS 开发”QQ 群，大家一边在群里讨论技术问题，一边通过各自的博客把成果分享出去。我有幸参与其中，可以说，这是 iOS 领域最专业的技术群体之一。2013 年春节，唐巧开通了微信公众平台“iOS 开发”（iOSDevTips），开始基于微信公众号传播自己的开发理念和相关技术，迄今为止已经积累了一万多名专业订阅者。

在承担日常研发任务的同时，还能做这么多的技术写作和内容分享，他让我这个老程序员亦十分佩服。

2013 年的一天，我在一次技术会议上碰到了唐巧，谈起了技术写作的事情。现在国内 iOS 开发领域的技术图书良莠不齐，很多是“编著”，摘抄或翻译自官方文档，而且大家很喜欢去写入门类的图书，从编程语言、MVC、协议、策略到 Xcode 的使用，到第一个 App 项目的

构建，等等，高级一点的进阶内容就少有涉及，很多 iOS 开发者让我推荐一本面向中高级开发者的中文书时，我总是遍寻不着。那时候我正准备出版自己的《MacTalk · 人生元编程》，于是也希望唐巧写一本 iOS 进阶方面的书。我对唐巧说，你的博客上已经产出了那么多高质量的 iOS 开发进阶内容，为什么不再增加一些内容，集结成册梳理成书呢？他若有所思地回答，我正准备这么干！

经年以后，唐巧再也没和我提起这件事，我也在忙忙碌碌中把他的书忘得一干二净。上周突然收到唐巧的一份书稿，打开一看，《iOS 开发进阶》静静地躺在邮箱里，已然成书。唐巧用了一年的时间，在博客文章的基础上进行完善、修改、增补，用清晰的“丝线”把散落各处的“珍珠”串起，形成脉络完整的知识体系，然后又新增了超过成书一半的内容，最终完成了这本关于 iOS 开发的进阶图书。

书中从 iOS 开发实战最常用的工具讲起，一直写到底层开发原理。内容分成三块：工具、实践和原理，每个章节都独立成文，读者可以从头细细品读，也可以随时翻阅其中一节，利用碎片时间阅读学习。就我个人而言，非常喜欢实践和原理两个部分，比如处理多核编程的 GCD 技术、应用程序安全技术、CoreText 的排版引擎、Objective-C 对象模型、block 对象模型等，这些内容要么能帮助我解决实际问题，要么能够答疑解惑，深入细节。作为一个仍然在一线开发的技术作者，唐巧采用了图文并茂的方式去阐述问题，每个技术专题都配备了恰当的代码，力求将复杂的技术以最为简洁的方式呈现出来，让读者一目了然，读来深入浅出，并不枯燥。

写书永远有遗憾。由于出版时间的要求，这本书的原理部分略显单薄，另外，全书并未涉及苹果的最新语言 Swift 相关的内容，相关的代码示例都是基于 Objective-C 的。希望唐巧的下一本书能够涉及更多这部分内容，这也是我的一个心愿。

在这样一个原创匮乏的年代，我能做到的就是向更多的人去推荐这样一本书。人生如长跑，成长如进阶，如果你是一个 iOS 开发者，这本书一定不要错过！

池建强

《MacTalk · 人生元编程》作者

微信平台 MacTalk 出品人

2014 年，冬

目录

第1章 引言

1.1 我为什么要写本书

1.1.1 我的构思

还记得几年前，我在学习 iOS 开发入门后，发现同行间的交流环境比较差，自己提高的过程很慢。在出版物方面，我也一直苦于没有找到一本 iOS 开发进阶方面的图书。

随着移动互联网的快速发展，移动开发的人才也慢慢增多，现在同行间的交流环境相比以前好了很多。但是在国内，仍然没有一本原创的 iOS 进阶方面的图书出现。因此，我萌生了写作的想法。

写作最开始是构思的过程，我回顾自己维护多年的 iOS 开发博客 (http://www.devtang.com)，从里面 100 多篇关于 iOS 开发的原创文章中，整理出涉及 iOS 开发进阶提高的三个主要的方向：工具、实践、理论。于是，我围绕着上述的三个方向，把以前的文章作为基础，再进一步完善相关内容，让零散的知识点能够衔接起来，成为一个完整的体系。整个写作的过程是痛并快乐着的，为了保证知识的完整，最终书稿中有超过 50% 是新增加的内容。

从知识的难度上，工具、实践、理论这三大部分的难度逐步提高，以保证读者能慢慢适应。而三大部分内容其实各自独立，有经验的读者也可以按兴趣点或具体需求，跳跃性地阅读本书。

1.1.2 内容导读

本书分工具、实践、理论三大部分。第一部分介绍 iOS 开发的常用工具，第二部分介绍 iOS 开发中的一些常见的实践经验，第三部分介绍 iOS 开发中涉及的原理。

如果把成为 iOS 开发高手的过程比作武侠小说中的修炼过程的话，工具、实践和理论的学习

就分别对应兵器、招式和内功的修炼。本书希望通过这三方面的综合讲解，全面提高开发者的技能水平。

好的工具可以使得开发效率成倍增长。本书第一部分介绍了 iOS 开发中的各种工具或服务，使用它们可以极大地方便我们的日常开发和维护，提高开发效率。

一个入门之后的 iOS 开发者，需要的是更深入的实际开发经验。本书第二部分选择了 iOS 开发中几个常见的实际场景，进一步介绍各种 iOS 开发进阶的实战技巧。对于一些比较零散的知识技巧，我也在实战小技巧集锦中进行了介绍。

如果把前两部分比作工具和招式的修炼的话，本书第三部分则是内功的修炼。在第三部分中，本书详细分析了 iOS 开发涉及的语言对象模型，从而能够帮助读者深入理解语言的各种特性和限制。

1.1.3 目标读者

本书定位于帮助那些 iOS 开发刚刚入门的同行快速提高自己的水平，适用于有三个月以上 iOS 开发经验的读者，不适合没有任何 iOS 开发基础的读者阅读。

对于 iOS 熟练开发者，本书也能补充其知识点的可能的盲区，使其相关知识体系更为完善。

1.1.4 随书示例代码和勘误

为了节省篇幅，本书中的示例工程代码只选取了关键的部分，完整的工程代码在与本书对应的 github 项目里，项目地址是：https://github.com/tangqiaoboy/iOS-Pro。

本书花费了我大量精力，但就像几乎没有毫无漏洞的代码一样，本书中难免会有一些错误，所以本书的勘误也会一同更新在该 github 项目中。

1.1.5 致谢

首先，感谢我的老婆和岳母，在我集中写作的这一年内，她们承担了大部分的家务，给了我足够安静的写作环境。然后，感谢我建的“真 · iOS 开发”群里的每一个朋友，在我眼中这是全中国最牛的 iOS 开发者聚集地，我每天从大家的讨论中收获很多。最后，感谢电子工业出版社的张春雨老师、高丽阳老师，是他们的鼓励和协助，让本书能够快速出版。

在本书的写作过程中，我的宝宝也顺利降生，也把本书献给她，愿她能够健康快乐地成长。

1.2 如何提高 iOS 开发技能

许多人在博客上咨询我 iOS 开发技能如何提高，本书就是一本旨在帮助读者提高 iOS 开发技能的书，除了书中专门介绍的方法外，我也介绍其他一些方法。

1.2.1 阅读博客

在现在这个碎片化阅读流行的年代，博客的风头早已被微博盖过。而我却坚持写博客，并且大量地阅读同行的 iOS 开发博客。博客的文章长度通常在 3000 字左右，许多 iOS 开发知识都至少需要这样的篇幅才能完整地讲解清楚。博客相对于书籍来说，并没有较长的出版发行时间，所以阅读博客对于获取最新的 iOS 开发知识有着非常良好的效果。

我自己精心整理了国内 40 多位 iOS 开发博主的博客地址列表：https://github.com/tangqiaoboy/iOSBlogCN，希望大家都能培养起阅读博客的习惯。

国外也有很多优秀的 iOS 开发博客，它们整体质量比中文的博客更高，以下是一些值得推荐的博客地址。

博客名	博客地址
objc.io	http://www.objc.io/
Ray Wenderlich	http://www.raywenderlich.com
iOS Developer Tips	http://iosdevelopertips.com/
iOS Dev Weekly	http://iosdevweekly.com/
NSHipster	http://nshipster.com/
Bartosz Ciechanowski	http://ciechanowski.me
Big Nerd Ranch Blog	http://blog.bignerdranch.com
Nils Hayat	http://nilsou.com/

另外，使用博客 RSS 聚合工具（例如 Feedly：http://www.feedly.com/）可以获得更好的博客阅读体验。手机上也有很多优秀的博客阅读工具（我使用的是 Newsfy）。合理地使用这些工具也可以将你在地铁上、睡觉前等碎片时间充分利用起来。

1.2.2 读书

博客的内容通常只能详细讲解一个知识点，而图书则能成体系地介绍整个知识树。与国外相比，中国的图书售价相当便宜，所以读书其实是一个非常划算的提高方式。建议大家每年至少坚持读完一本高质量的 iOS 开发图书。

1.2.3 看 WWDC 视频

由于 iOS 开发在快速发展，每年苹果都会给我们带来很多新的知识。而对于这些知识，第一手的资料就是 WWDC 的视频。

通常情况下，iOS 开发的新知识首先会在 WWDC 上被苹果公开，三个月左右之后，会有国内外的博客介绍这些知识，再过半年左右，会有国外的图书介绍这些知识。所以如果想尽早地了解这些知识，只有通过 WWDC 的视频。

现在每年的 WWDC 视频都会在会议过程中逐步放出，重要的视频会带有英文字幕。坚持阅读这些视频不但可以获得最新的 iOS 开发知识，还可以提高英文听力水平。

1.2.4 看苹果的官方文档

苹果的官方文档相当详尽，对于不熟悉的 API，阅读官方文档也是最直接有效的解决方式。

1.2.5 看开源项目的代码

大家一定有这样的感受，很多时候用文字讲解半天，还不如写几行代码来得直观。阅读优秀的开源项目代码，不但可以学习到 iOS 开发本身的基本知识，还能学习到设计模式等软件架构上的知识。

如果读者能够参与到开源项目的开发中，则能进一步提高自己的能力。

1.2.6 多写代码，多思考

知识的积累离不开实践和总结，我认为写过的 iOS 代码如果没有超过 10 万行，是不能称得上熟悉 iOS 开发的。某些在校的学生，仅仅做了几个 C++ 的大作业，就在求职简历里面写上“精通 C++”，真是让人哭笑不得。

在多写代码的同时，我们也要注意不要“重复造轮子”，尽量保证每次写的代码都能具有复用性。在代码结构因为业务需求需要变更时，及时重构，在不要留下技术债的同时，我们也要多思考如何设计应用架构，才能够保证满足灵活多变的产品需求。

在多次重构和思考的过程中，我们会慢慢积累出一类问题的最佳解决方式，成为自己宝贵的经验。

1.2.7 多和同行交流

有些时候遇到一些难解的技术问题，和同行的几句交流就可能让你茅塞顿开。另外，常见的技术问题通常都有人以前遇到过，简单指导几句就能让你一下子找到正确的解决方向。

国内开发者之间的交流，可以通过论坛、微博、QQ 群等方式来进行。另外各大公司有时候会办技术沙龙，这也是一个认识同行的好机会。

需要特别提醒的是，和国内开发者交流要注意讨论质量，有一些论坛和 QQ 群讨论质量相当低下，提的问题都是能通过简单搜索解决的，这种社区一定要远离，以提高自己的沟通效率。

除了在国内的技术社区交流，建议读者去国外的 Stack Overflow（http://www.stackoverflow.com）上提问或回答问题。

1.2.8 分享

值得尝试的分享方式有：发起一个开源项目、写技术博客、在技术会议上做报告。这几种方式都比较有挑战性，但是如果能大胆尝试，肯定会有巨大的收获。

第一部分：iOS 开发工具

iOS 开发工具部分详细介绍了 iOS 开发必备的命令行工具、图形工具、插件工具及第三方网站提供的相关服务。

命令行工具，我们将学习 CocoaPods。

图形工具，我们将学习 Charles 和 Reveal。

插件工具，我们将学习 Alcatraz 及一系列 Xcode 增强插件。

第三方的服务，我们将学习统计服务 Flurry、崩溃日志记录服务 Crashlytics、App Store 统计服务 App Annie。

熟练掌握这些 iOS 开发工具和服务，可以使我们的开发效率得到成倍地提高。

第一部分 iOS 开发工具

第2章 使用 CocoaPods 做依赖管理

每种语言发展到一定阶段，都会出现相应的依赖管理工具，例如 Java 语言的 Maven、Node.js 的 npm 等。本章讲解 iOS 项目的依赖管理工具 CocoaPods。

2.1 CocoaPods 简介

随着 iOS 开发者的增多，业界也出现了为 iOS 程序提供依赖管理的工具，它的名字叫作 CocoaPods（http://cocoapods.org/）。

CocoaPods 项目的源码（https://github.com/CocoaPods/CocoaPods）在 Github 上管理。该项目开始于 2011 年 8 月，经过多年发展，现在已经成为 iOS 开发事实上的依赖管理标准工具。开发 iOS 项目不可避免地要使用第三方开源库，CocoaPods 的出现使得我们可以节省设置和更新第三方开源库的时间。

我在开发猿题库客户端时，使用了 24 个第三方开源库。在使用 CocoaPods 以前，我需要：

1. 把这些第三方开源库的源代码文件复制到项目中，或者设置成 git 的 submodule。
2. 这些开源库通常需要依赖系统的一些 framework，我需要手工地将这些 framework 一一增加到项目依赖中。比如通常情况下，一个网络库就需要增加以下 framework：CFNetwork、SystemConfiguration、MobileCoreServices、CoreGraphics、zlib。
3. 对于某些开源库，我还需要设置 `-licucore` 或者 `-fno-objc-arc` 等编译参数。
4. 管理这些依赖包的更新。

这些“体力活”虽然简单，但毫无技术含量并且浪费时间。在使用 CocoaPods 之后，我只需要将用到的第三方开源库放到一个名为 Podfile 的文件中，然后执行 `pod install`。CocoaPods 就会自动将这些第三方开源库的源码下载下来，并且为我的工程设置好相应的系统依赖和编译参数。

2.2 CocoaPods 的安装和使用

2.2.1 CocoaPods 的安装

CocoaPods 的安装方式异常简单，Mac 下都自带 ruby，使用 ruby 的 gem 命令即可下载安装：

```
$ sudo gem install cocoapods
$ pod setup
```

如果你的 gem 太老，可以尝试用如下命令升级 gem：

```
sudo gem update --system
```

另外，ruby 的软件源rubygems.org因为使用亚马逊的云服务，所以被屏蔽了，需要更新一下 ruby 的源，下面的代码将官方的 ruby 源替换成国内淘宝的源：

```
gem sources --remove https://rubygems.org/
gem sources -a http://ruby.taobao.org/
gem sources -l
```

还有一点需要注意，`pod setup` 在执行时，会输出 `Setting up CocoaPods master repo`，但是会等待比较久的时间。这步其实是 CocoaPods 在将它的信息下载到 `~/.cocoapods` 目录下，如果你等太久，可以试着 cd 到那个目录，用 `du -sh *` 来查看下载进度。你也可以参考本文接下来的“使用 CocoaPods 的镜像索引”一节的内容来提高下载速度。

2.2.2 使用 CocoaPods 的镜像索引

所有项目的 Podspec 文件都托管在https://github.com/CocoaPods/Specs。第一次执行 `pod setup` 时，CocoaPods 会将这些 Podspec 索引文件更新到本地的 `~/.cocoapods/`目录下，这个索引文件比较大，有 80MB 左右。所以第一次更新时非常慢，我当初就更新了将近一个小时才完成。

一个叫 akinliu（http://akinliu.github.io/2014/05/03/cocoapods-specs-/）的朋友在 gitcafe（http://gitcafe.com/）和 occhina（http://www.oschina.net/）上建立了 CocoaPods 索引库的镜像，因为 gitcafe 和 occhina 都是国内的服务器，所以在执行索引更新操作时，会快很多。如下操作可以将 CocoaPods 设置成使用 gitcafe 镜像：

```
pod repo remove master
pod repo add master https://gitcafe.com/akuandev/Specs.git
pod repo update
```

将以上代码中的 `https://gitcafe.com/akuandev/Specs.git` 替换成 `http://git.oschina.net/akuandev/Specs.git` 即可使用 occhina 上的镜像。

2.2.3 使用 CocoaPods

使用时需要新建一个名为 Podfile 的文件，以如下格式，将依赖的库名字依次列在文件中即可：

```
platform :ios
pod 'JSONKit',       '~> 1.4'
pod 'Reachability',  '~> 3.0.0'
pod 'ASIHTTPRequest'
pod 'RegexKitLite'
```

然后将编辑好的 Podfile 文件放到项目根目录中，执行如下命令即可：

```
cd "your project home"
pod install
```

现在，所有第三方库都已经下载完成并且设置好了编译参数和依赖，你只需要记住如下两点即可：

1. 使用 CocoaPods 生成的 *.xcworkspace 文件来打开工程，而不是以前的 *.xcodeproj 文件。
2. 每次更改了 Podfile 文件，都需要重新执行一次 pod update 命令。

2.2.4 查找第三方库

如果不知道 CocoaPods 管理的库中是否有你想要的库，那么你可以通过 pod search 命令进行查找，以下是我用 pod search json 查找到的所有可用的库：

```
$ pod search json

-> AnyJSON (0.0.1)
   Encode / Decode JSON by any means possible.
   - Homepage: https://github.com/mattt/AnyJSON
   - Source:   https://github.com/mattt/AnyJSON.git
   - Versions: 0.0.1 [master repo]


-> JSONKit (1.5pre)
   A Very High Performance Objective-C JSON Library.
   - Homepage: https://github.com/johnezang/JSONKit
   - Source: git://github.com/johnezang/JSONKit.git
   - Versions: 1.5pre, 1.4 [master repo]

// ...以下省略若干行
```

2.3 注意事项

2.3.1 关于.gitignore

当你执行 pod install 之后，除了 Podfile 外，CocoaPods 还会生成一个名为 Podfile.lock 的文件，你不应该把这个文件加入到.gitignore 中。因为 Podfile.lock 会锁定当前各依赖库的版本，之后即使多次执行 pod install 也不会更改版本，只有执行 pod update 才会改变 Podfile.lock。在多人协作的时候，这样可以防止第三方库升级时造成大家各自的第三方库版本不一致。

CocoaPods 的一篇官方文档（http://guides.cocoapods.org/using/using-cocoapods.html#should-i-ignore-the-pods-directory-in-source-control）也在 What is a Podfile.lock 一节中介绍了 Podfile.lock 的作用，并且指出：

> This file should always be kept under version control.

2.3.2 为自己的项目创建 podspec 文件

我们可以为自己的开源项目创建 podspec 文件，首先通过如下命令初始化一个 podspec 文件：

```
pod spec create your_pod_spec_name
```

该命令执行之后，CocoaPods 会生成一个名为 your_pod_spec_name.podspec 的文件，然后我们修改其中的相关内容即可。

具体步骤可以参考这两篇博文中的相关内容：《如何编写一个 CocoaPods 的 spec 文件》（http://ishalou.com/blog/2012/10/16/how-to-create-a-cocoapods-spec-file/）和《CocoaPods 入门》（http://studentdeng.github.io/blog/2013/09/13/cocoapods-tutorial/）。

2.3.3 使用私有的 pods

我们可以直接指定某一个依赖的 podspec，这样就可以使用企业内部的私有库。该方案有利于使企业内部的公共项目支持 CocoaPods。如下是一个示例：

```
pod 'MyCommon', :podspec => 'https://yuantiku.com/common/myCommon.podspec'
```

2.3.4 不更新 podspec

CocoaPods 在执行 pod install 和 pod update 时，会默认先更新一次 podspec 索引。使用--no-repo-update 参数可以禁止其做索引更新操作。代码如下所示：

```
pod install --no-repo-update
pod update --no-repo-update
```

2.3.5 生成第三方库的帮助文档

如果你想让 CococaPods 帮你生成第三方库的帮助文档，并集成到 Xcode 中，那么用 brew 安装 appledoc 即可：

```
brew install appledoc
```

关于 appledoc，在本书的第 9 章“其他工具介绍”一章有专门介绍。它最大的优点是可以将帮助文档集成到 Xcode 中，这样你在敲代码的时候，按住 Opt 键单击类名或方法名，就可以显示出相应的帮助文档。

2.3.6 原理

CocoaPods 的原理是将所有的依赖库都放到另一个名为 Pods 的项目中，然后让主项目依赖 Pods 项目，这样，源码管理工作都从主项目移到了 Pods 项目中。下面是一些技术细节：

1. Pods 项目最终会编译成一个名为 libPods.a 的文件，主项目只需要依赖这个.a 文件即可。
2. 对于资源文件，CocoaPods 提供了一个名为 Pods-resources.sh 的 bash 脚本，该脚本在每次项目编译的时候都会执行，将第三方库的各种资源文件复制到目标目录中。
3. CocoaPods 通过一个名为 Pods.xcconfig 的文件在编译时设置所有的依赖和参数。

第3章 网络封包分析工具 Charles

Charles（http://www.charlesproxy.com/）是在 Mac 下常用的截取网络封包的工具。Charles 通过将自己设置成系统的网络访问代理服务器，使得所有的网络访问请求都通过它来完成，从而实现了网络封包的截取和分析。

3.1 Charles 简介

Charles 是收费软件，可以免费试用 30 天。试用期过后，未付费的用户仍然可以继续使用，但是每次使用时间不能超过 30 分钟，并且启动时将会有 10 秒钟的延时。

因此，该付费方案对广大用户还是相当友好的，即使你长期不付费，也能使用完整的软件功能。只是当你需要长时间进行封包调试时，会因为 Charles 强制关闭而受到影响。

Charles 主要的功能包括：

1. 支持 SSL 代理。可以截取分析 SSL（http://zh.wikipedia.org/wiki/%E5%AE%89%E5%85%A8%E5%A5%97%E6%8E%A5%E5%B1%82）的请求。
2. 支持流量控制。可以模拟慢速网络，以及等待时间（latency）较长的请求。

3. 支持 AJAX 调试。可以自动将 JSON 或 XML 数据格式化，方便查看。
4. 支持 AMF 调试。可以将 Flash Remoting 或 Flex Remoting 信息格式化，方便查看。
5. 支持重发网络请求，方便后端调试。
6. 支持修改网络请求参数。
7. 支持网络请求的截获和动态修改。
8. 检查 HTML、CSS 和 RSS 内容是否符合 W3C 标准（http://validator.w3.org/）。

3.2 Charles 的安装和使用

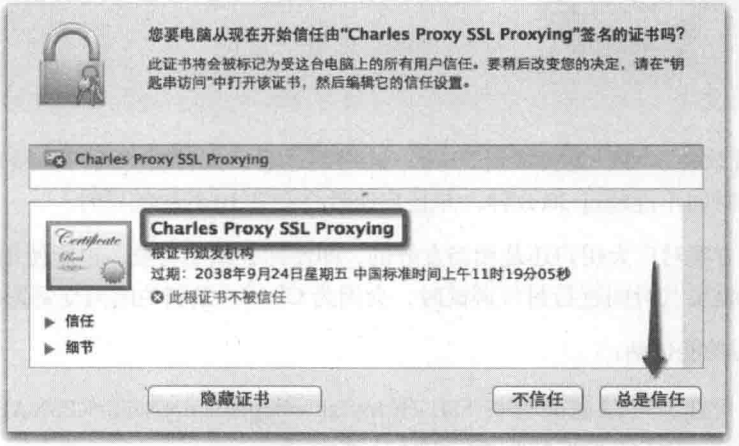
3.2.1 安装 Charles

去 Charles 的官方网站（http://www.charlesproxy.com）下载最新版的 Charles 安装包，它是一个 dmg 后缀的文件。打开后将 Charles 拖到 Application 目录下即完成安装。

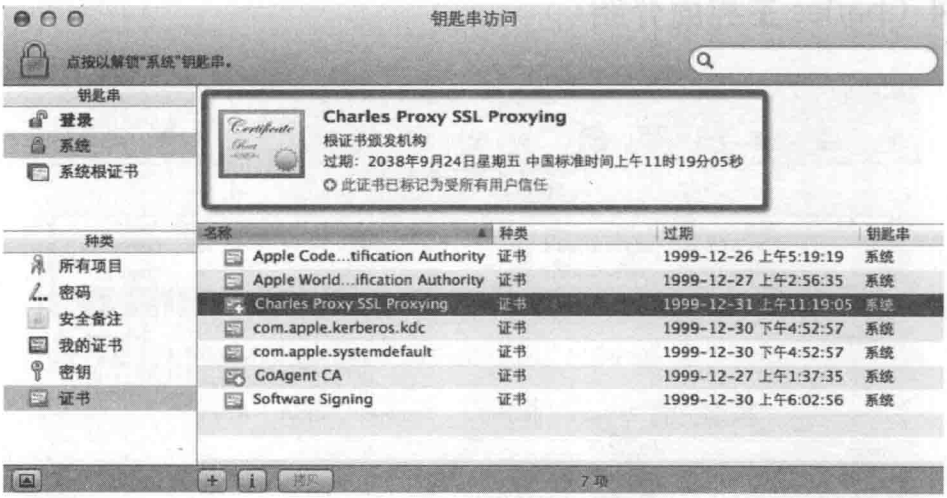
3.2.2 安装 SSL 证书

如果你需要截取分析 SSL 协议相关的内容，那么需要安装 Charles 的 CA 证书。具体步骤如下：

1. 去http://www.charlesproxy.com/ssl.zip下载 CA 证书文件。
2. 解压该 zip 文件后，双击其中的.crt 文件，这时候在弹出的菜单中选择“总是信任”按钮，如下图所示。



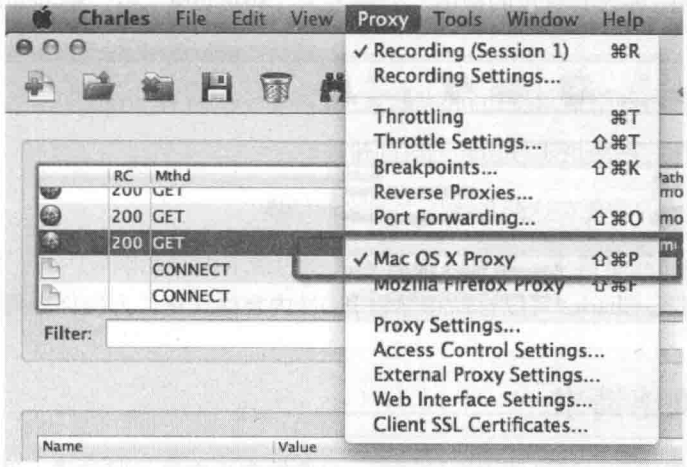
3. 从钥匙串访问中即可看到添加成功的证书，如下图所示。



3.2.3 将 Charles 设置成系统代理

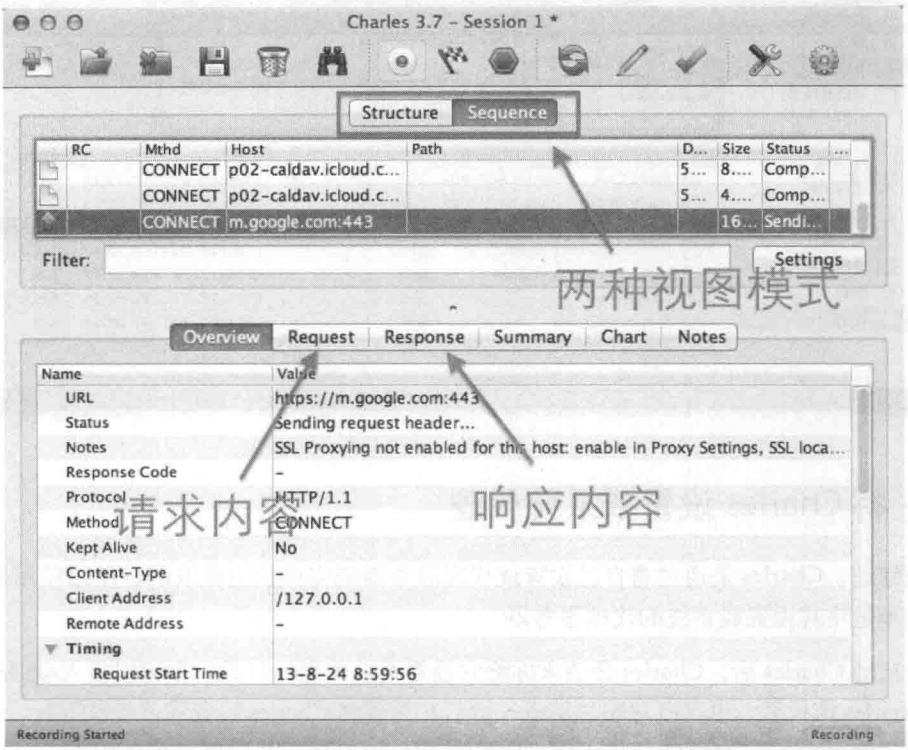
前面提到过，Charles 是通过将自己设置成代理服务器来完成封包截取的，所以使用 Charles 的第一步是将其设置成系统的代理服务器。

第一次启动 Charles 后，Charles 会请求你给它设置系统代理的权限。你可以输入登录密码以授予 Charles 该权限，也可以忽略该请求，然后在需要将 Charles 设置成系统代理时，选择菜单中的“Proxy”→“Mac OS X Proxy”来将 Charles 设置成系统代理，如下图所示。



之后，你就可以看到源源不断的网络请求出现在 Charles 的界面中。

3.2.4 Charles 主界面介绍



Charles 主要提供两种查看封包的视图，分别名为“Structure”和“Sequence”，它们的功能分别为：

1. Structure 视图将网络请求按访问的域名分类。
2. Sequence 视图将网络请求按访问的时间排序。

大家可以根据具体的需要在这两种视图之间来回切换。

对于某一个具体的网络请求，你可以查看其详细的请求内容和响应内容。如果响应内容是 JSON 格式的，那么 Charles 可以自动帮你将 JSON 内容格式化，方便你查看。

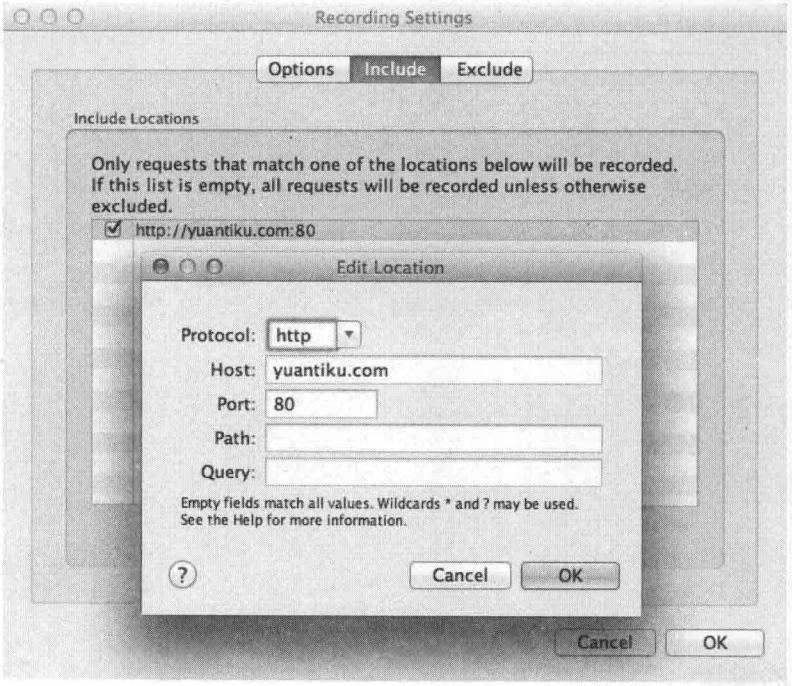
3.2.5 过滤网络请求

通常情况下，我们需要对网络请求进行过滤，只监控向指定目录服务器上发送的请求。对于这种需求，我们有两种办法来设置：

1. 在主界面的中部的 Filter 栏中填入需要过滤出来的关键字。例如我们的服务器的地址

是http://yuantiku.com，那么只需要在 Filter 栏中填入 yuantiku 即可。

2. 在 Charles 的菜单栏选择“Proxy”→“Recording Settings”，然后选择 Include 栏，选择添加一个项目，然后填入需要监控的协议、主机地址、端口号。这样就可以只截取目标网站的封包了，如下图所示。



通常情况下，我们使用方法 1 做一些临时性的封包过滤，使用方法 2 做一些经常性的封包过滤。

3.3 使用 Charles 协助 iOS 开发

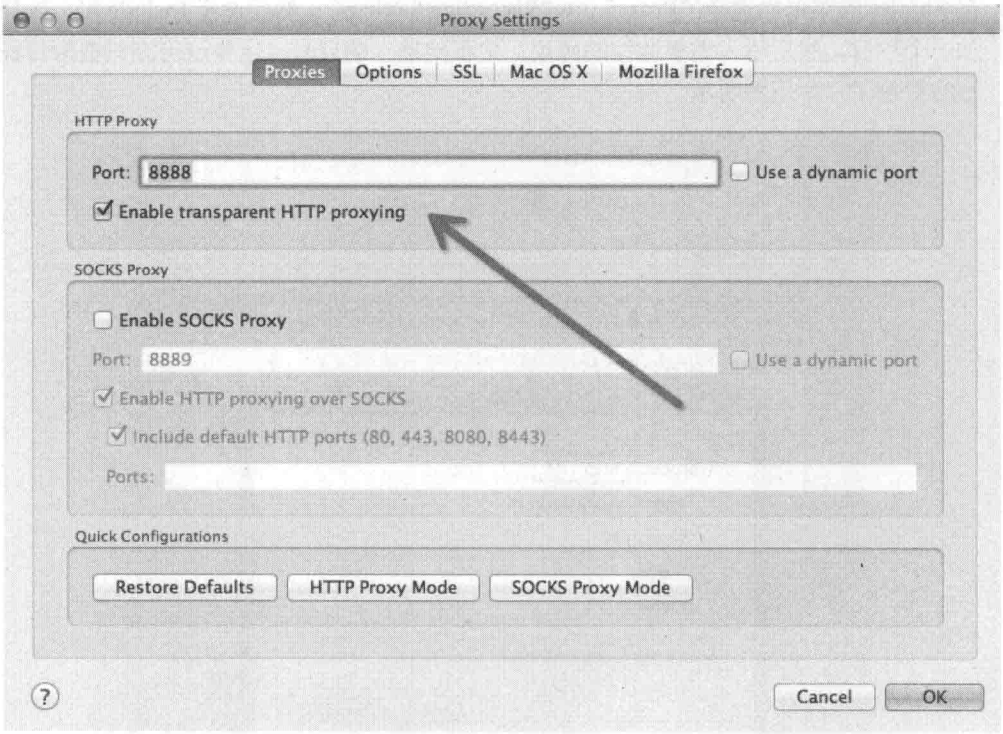
3.3.1 截取 iPhone 上的网络封包

Charles 通常用来截取本地的网络封包，但是当需要时，我们也可以用它来截取其他设备上的网络请求。下面我就以 iPhone 为例，讲解如何进行相应操作。

Charles 上的设置

要截取 iPhone 上的网络请求，我们首先需要将 Charles 的代理功能打开。在 Charles 的菜单栏上选择“Proxy”→“Proxy Settings”，填入代理端口 8888，并且勾选“Enable transparent

HTTP proxying”，就完成了在 Charles 上的设置，如下图所示。



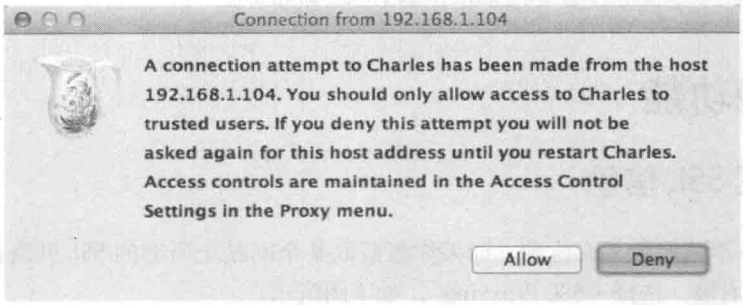
iPhone 上的设置

首先我们需要获取 Charles 运行所在的电脑的 IP 地址，打开 Terminal，输入 `ifconfig en0`，即可获得该电脑的 IP 地址，如下图所示。

```
➜ ~ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether 84:38:35:5c:2a:88
        inet6 fe80::8638:35ff:fe5c:2a88%en0 prefixlen 64 scopeid 0x4
        inet 192.168.1.105 netmask 0xffffff00 broadcast 192.168.1.255
        media: autoselect
        status: active
```

在 iPhone 的“设置”→“无线局域网”中，可以看到当前连接的 WiFi 名，通过单击右边的详情按钮，可以看到当前连接上的 WiFi 的详细信息，包括 IP 地址、子网掩码等信息。在其最底部有“HTTP 代理”一项，我们将其切换成手动，然后填上 Charles 运行所在的电脑的 IP 地址，以及端口号 8888，如下图所示。

设置好之后，我们打开 iPhone 上的任意需要网络通讯的程序，就可以看到 Charles 弹出请求连接的确认菜单，单击“Allow”按钮即可完成设置，如下图所示。

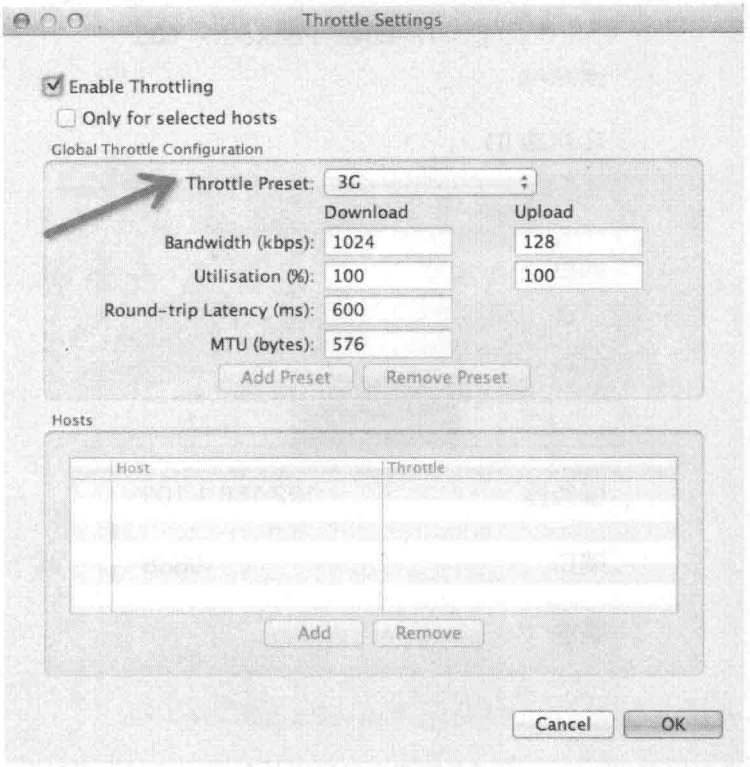


3.3.2 模拟慢速网络

在做 iPhone 开发的时候，我们常常需要模拟慢速网络或者高延迟的网络，以测试在移动网络下应用的表现是否正常。Charles 对此需求提供了很好的支持。

在 Charles 的菜单上，选择“Proxy”→“Throttle Setting”项，在弹出的对话框中，我们可以

勾选上“Enable Throttling”，并且可以设置 Throttle Preset 的类型，如下图所示。

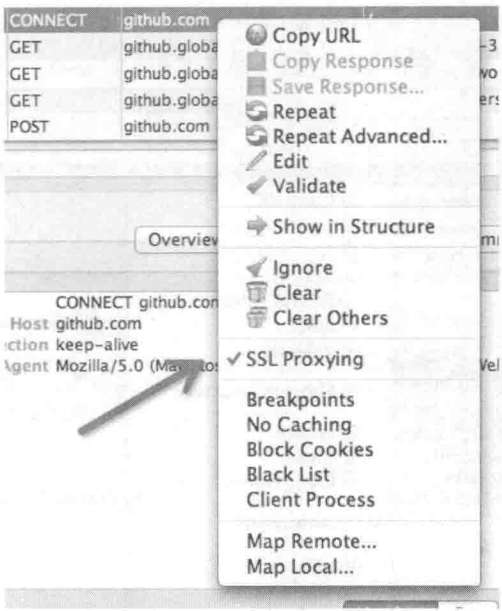


如果我们只想模拟指定网站的慢速网络，可以再勾选上图中的“Only for selected hosts”项，然后在对话框的下半部分设置中增加指定的 Hosts 项即可。

3.4 高级功能

3.4.1 截取 SSL 信息

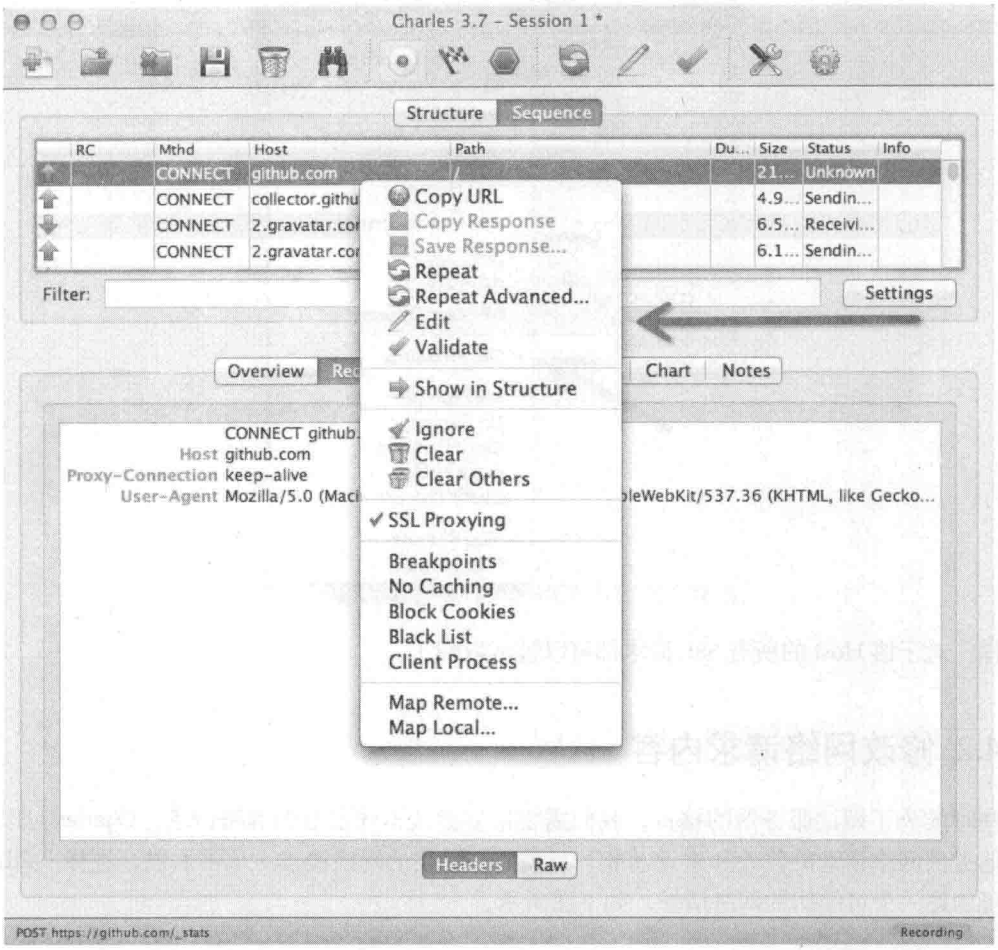
Charles 默认并不截取 SSL 的信息，如果你想截取某个网站上所有的 SSL 网络请求，可以在该请求上单击右键，选择“SSL Proxying”，如下图所示。



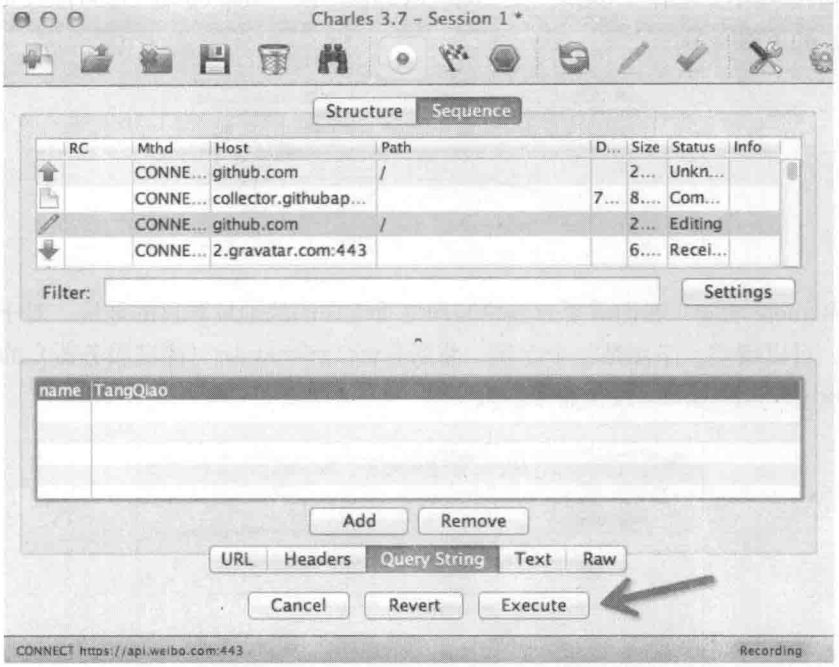
这样，对于该 Host 的所有 SSL 请求都可以被截取到了。

3.4.2 修改网络请求内容

有些时候为了调试服务器的接口，我们需要反复尝试不同参数的网络请求。Charles 可以方便地提供网络请求的修改和重发功能。只需要在以往的网络请求上单击右键，选择“Edit”，即可创建一个可编辑的网络请求，如下图所示。



我们可以修改该请求的任何信息，包括 URL 地址、端口、参数等，之后单击“Execute”按钮即可发送该修改后的网络请求，如下图所示。Charles 支持我们多次修改和发送该请求，这对于我们调试与服务器端间的接口非常方便。



3.4.3 修改服务器返回内容

有些时候我们想让服务器返回一些指定的内容，方便我们在一些特殊情况下的调试。例如，列表页面为空的情况，数据异常的情况，或部分耗时的网络请求超时的情况等。如果没有Charles，要服务器配合构造相应的数据显得比较麻烦。这个时候，使用 Charles 相关的功能就可以满足我们的需求。

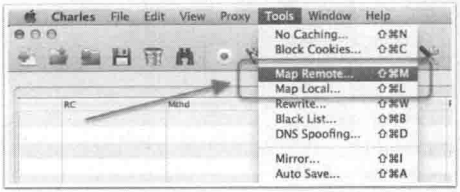
根据具体的需求，Charles 提供了 Map 功能、Rewrite 功能及 Breakpoints 功能，它们都可以达到修改服务器返回内容的目的。这三者在功能上的差异是：

1. Map 功能适合长期地将某一些请求重定向到另一个网络地址或本地文件。
2. Rewrite 功能适合对网络请求进行一些正则替换。
3. Breakpoints 功能适合做一些临时性的修改。

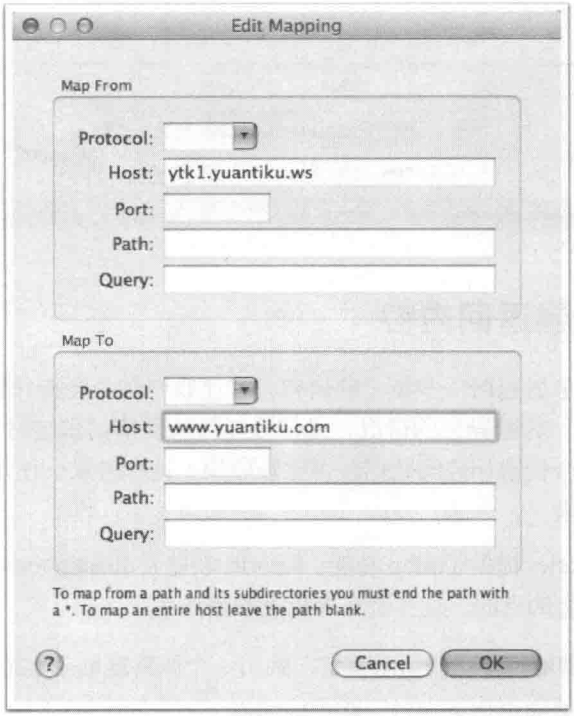
3.5 Map 功能

Charles 的 Map 功能分 Map Remote 和 Map Local 两种，顾名思义，Map Remote 是将指定的网络请求重定向到另一个网址，Map Local 是将指定的网络请求重定向到本地文件。

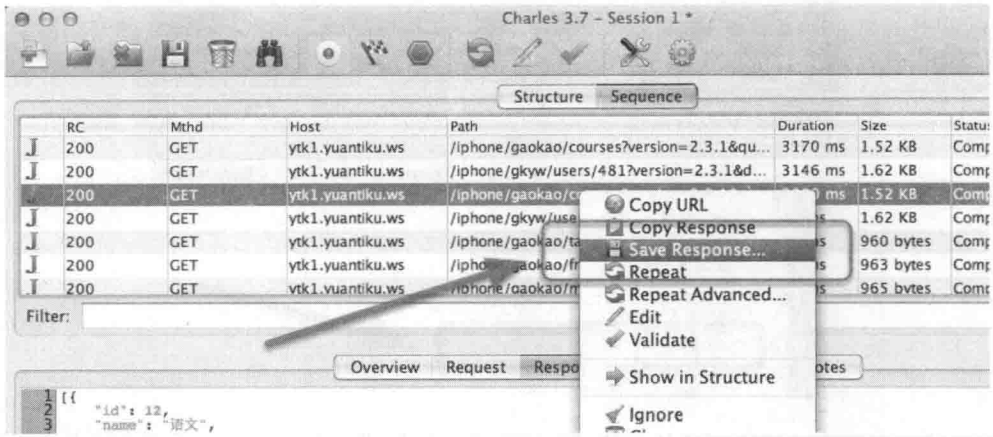
在 Charles 的菜单中，选择“Tools”→“Map Remote”或“Map Local”即可进入相应功能的设置页面，如下图所示。



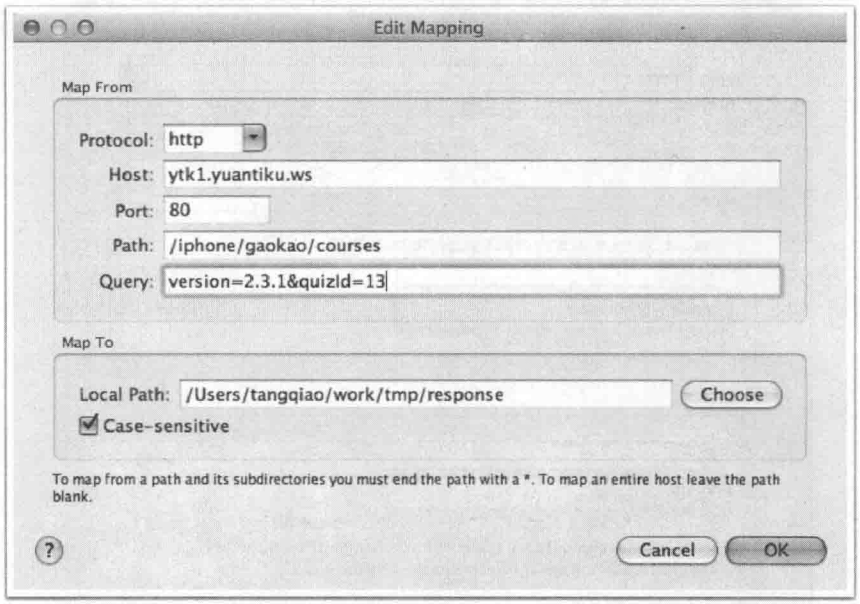
对于 Map Remote 功能，我们需要分别填写网络重定向的源地址和目的地址，对于不需要限制的条件，可以留空。下图是一个示例，将所有ytk1.yuanku.ws（测试服务器）的请求重定向到了www.yuantiku.com（线上服务器）。



对于 Map Local 功能，我们需要填写重定向的源地址和本地的目标文件。对于一些复杂的网络请求结果，我们可以先使用 Charles 提供的“Save Response…”功能，将请求结果保存到本地，如下图所示。然后稍加修改，使其成为我们的目标映射文件。



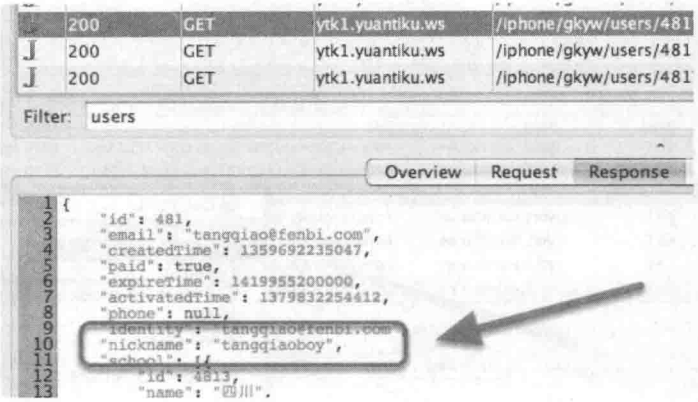
下图是一个示例，将一个指定的网络请求通过 Map Local 功能映射到了本地的一个经过修改的文件中。



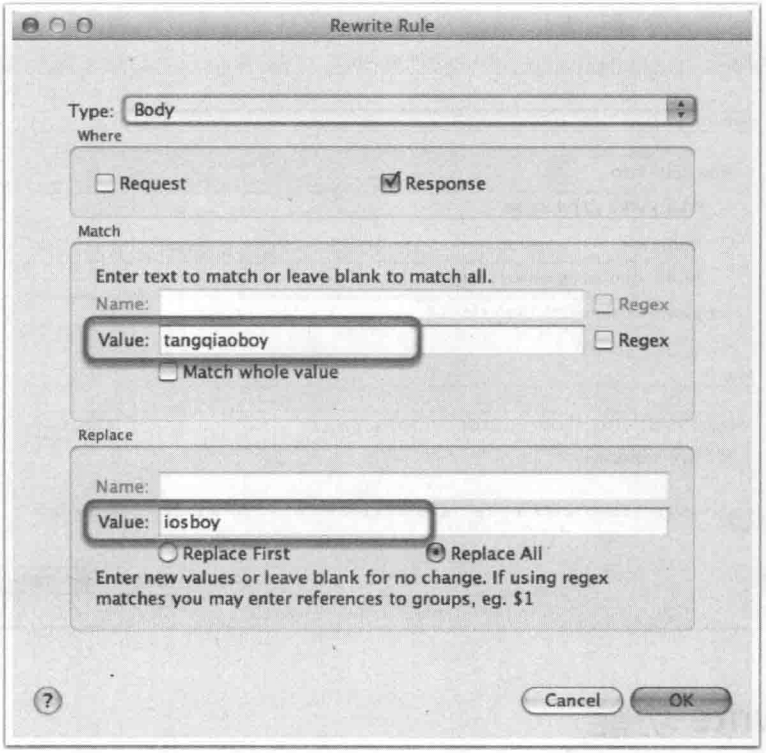
3.6 Rewrite 功能

Rewrite 功能适合对某一类网络请求进行一些正则替换，以达到修改结果的目的。

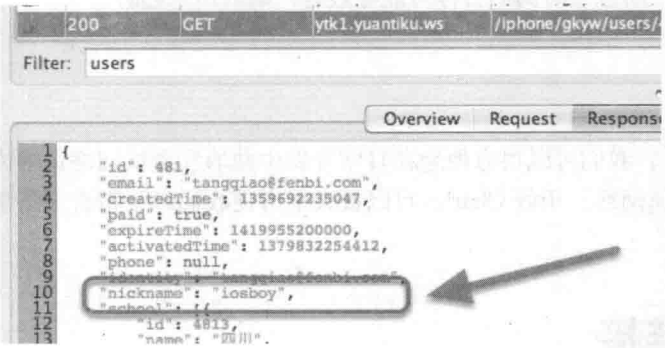
例如，我们的客户端有一个 API 请求是获得用户昵称，而我当前的昵称是“tangqiaoboy”，如下图所示。



我们想试着直接修改网络返回值，将“tangqiaoboy”换成“iosboy”。于是我们启用 Rewrite 功能，然后按下图进行设置。



完成设置之后，我们就可以从 Charles 中看到，之后的 API 获得的昵称被自动 Rewrite 成了“iosboy”，如下图所示。

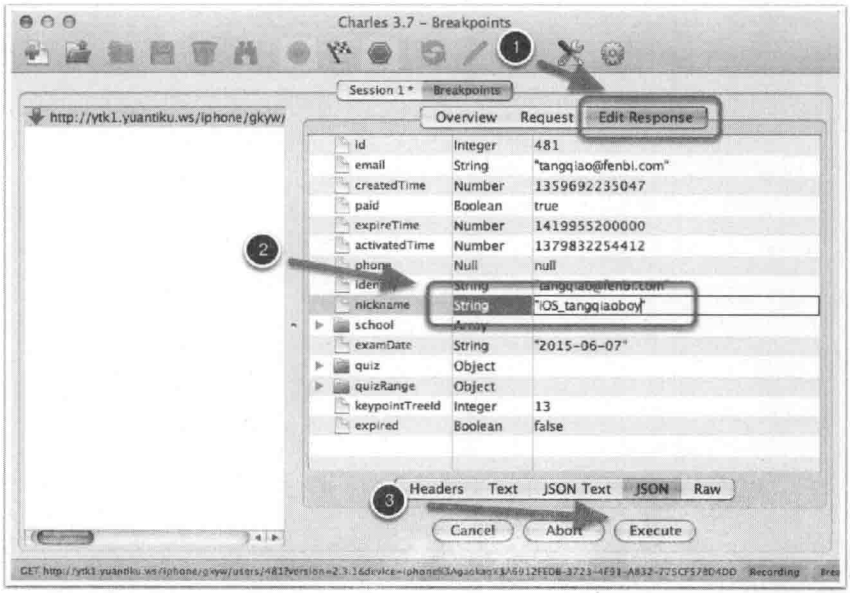


3.7 Breakpoints 功能

上面提供的 Rewrite 功能最适合做批量和长期的替换，但是很多时候，我们只是想临时修改一次网络请求结果，使用 Rewrite 功能虽然也可以达到目的，但是过于麻烦，对于临时性的修改，我们最好使用 Breakpoints 功能。

Breakpoints 功能类似我们在 Xcode 中设置的断点，当指定的网络请求发生时，Charles 会截获该请求，这个时候，我们可以在 Charles 中临时修改网络请求的返回内容。

下图是我们临时修改获取用户信息的 API，将用户的昵称进行了更改，修改完成后单击“Execute”按钮就可以让网络请求继续进行。



需要注意的是，在使用 Breakpoints 功能将网络请求截获并修改的过程中，整个网络请求的

计时并不会暂停，所以长时间的暂停可能导致客户端的请求超时。

3.8 总结

通过 Charles 软件，我们可以很方便地在日常开发中截取和调试网络请求内容，分析封包协议，以及模拟慢速网络。用好 Charles 可以极大地方便我们对于带有网络请求的应用的开发和调试。

3.9 参考链接

1. Charles 主要的功能列表：http://www.charlesproxy.com/overview/about-charles/
2. Charles 官方网站：http://www.charlesproxy.com/

第4章 界面调试工具 Reveal

Reveal（http://revealapp.com/）是一个 iOS 程序界面调试工具。使用 Reveal，我们可以在 iOS 开发时动态地查看和修改应用程序的界面。

对于动态或复杂的交互界面，手写 UI 是不可避免的。通过 Reveal，我们可以方便地调试和修改应用界面，免去了每次修改代码后重新启动的痛苦。

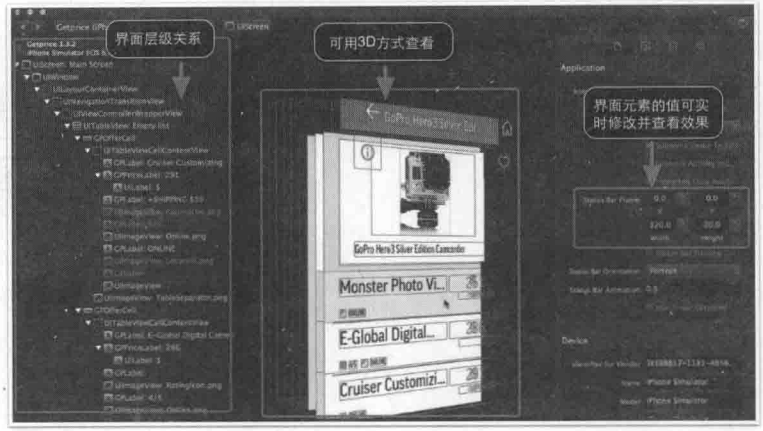
4.1 Reveal 简介

Reveal 类似 Chrome 的“审查元素”功能，使我们不但可以在运行时看到 iOS 程序的界面层级关系，还可以实时地修改程序界面，不用重新运行程序就可以看到修改之后的效果。

在使用时，我们将 Reveal 连接上模拟器或真机上正在运行的 iOS 程序，然后就可以查看调试 iOS 程序的界面。

下图是 Reveal 的运行界面，其界面主要分成三部分：

1. 左边部分是整个界面的层级关系，在这里可以以树形层级的方式来查看全部界面元素。
2. 中间部分是一个可视化的查看区域，用户可以在这里切换 2D 和 3D 的查看方式，这里看到的也是程序运行的实时界面。
3. 右边部边是控件的详细参数查看区域，当我们选中某一个具体的控件时，右边就可以显示出该控件的具体的参数列表。我们除了可以查看这些参数值是否正确外，还可以尝试修改这些值。所有的修改都可以实时反应到中间的实时预览区域。



Reveal 工具适合调试非 Interface Builder 创建的界面，Interface Builder 中创建的 xib 和 storyboard 在企业开发中并不是总能胜任。

一方面原因是企业开发常常是多人协作，xib 和 storyboard 对于版本管理工具并不友好，例如 xib 和 storyboard 的内容中，包含有编辑者使用的操作系统和 Xcode 的版本号信息，这样一旦协同开发者的操作系统版本或 Xcode 版本不一致，则每次打开 xib 和 storyboard，即使不做任何操作，也会修改其内容。

另一方面是由于商业应用界面的复杂性，xib 和 storyboard 并不能方便地提供更细粒度的模块复用。例如在开发猿题库时，为了复用，我们将很多按钮风格都进行了定制，如果使用 xib 或 storyboard，则不能很好地管理这些按钮。

使用 Reveal 可以很好地调试使用代码编写的 UI 界面，下面我们来看具体如何使用 Reveal 来进行界面调试。

4.2 Reveal 的使用

4.2.1 用 Reveal 连接模拟器调试

Reveal 官方介绍了好几种办法使 Reveal 连接模拟器，它们都需要修改工程文件。但如果修改了工程文件，就需要参与项目开发的所有人都装有 Reveal，这其实是相当不友好的。本节要介绍一种不修改任何工程文件的办法，在实际使用中，这种办法最简单方便。该方法的步骤如下：

首先打开 Terminal，输入 `vim ~/.lldbinit` 创建一个名为.lldbinit 的文件，然后将如下内容输入该文件中：

```
command alias reveal_load_sim expr (void*)dlopen("/Applications/Reveal.app/Contents/
```

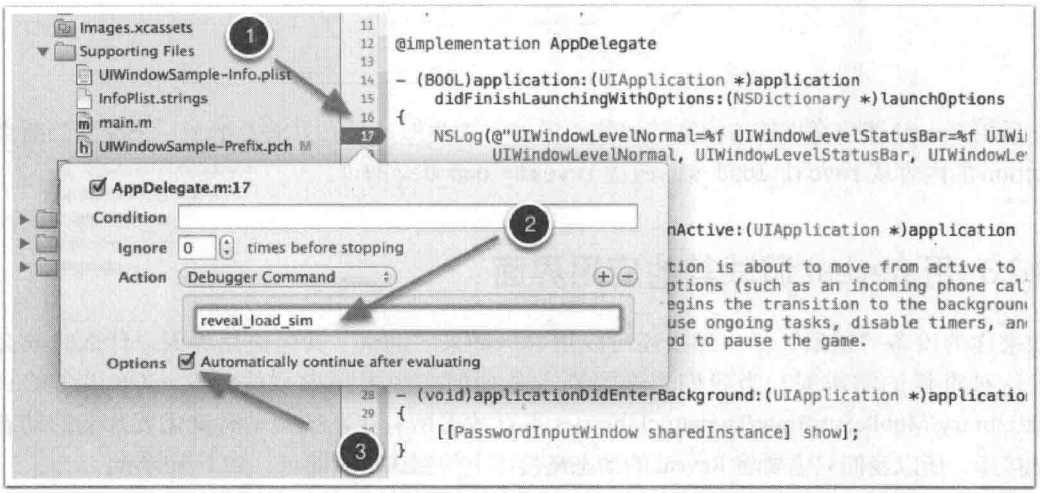
```
    SharedSupport/iOS-       Libraries/libReveal.dylib", 0x2);
command alias reveal_load_dev expr (void*)dlopen([(NSString*)[(NSBundle*)[NSBundle
    mainBundle]              pathForResource:@"libReveal" ofType:@"dylib"]
    cStringUsingEncoding:0x4], 0x2);
command alias reveal_start expr (void)[(NSNotificationCenter*)[NSNotificationCenter
    defaultCenter]           postNotificationName:@"IBARevealRequestStart" object:
    nil];
command alias reveal_stop expr (void)[(NSNotificationCenter*)[NSNotificationCenter
    defaultCenter]           postNotificationName:@"IBARevealRequestStop" object:
    nil];
```

该步骤其实是为 lldb 设置了 4 个别名，这样可以方便后续操作。这 4 个别名意义如下：

1. `reveal_load_sim` 为模拟器加载 reveal 调试用的动态链接库。
2. `reveal_load_dev` 为真机加载 reveal 调试用的动态链接库。
3. `reveal_start` 启动 reveal 调试功能。
4. `reveal_stop` 结束 reveal 调试功能。

接下来，我们在 AppDelegate 类的 `application: didFinishLaunchingWithOptions:` 方法中，进行下面三步操作，如下图所示。

1. 单击该方法左边的行号区域，增加一个断点，之后右击该断点，选择“Edit Breakpoint”。
2. 单击“Action”项右边的“+”按钮，然后输入“reveal_load_sim”。
3. 勾选上“Options”上的”Automatically continue after evaluating”选项。

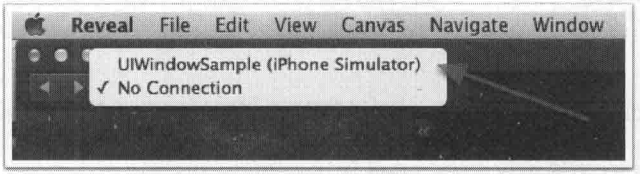


之后我们运行模拟器，打开 Reveal，就可以在 Reveal 界面的左上角看到有模拟器可以连接调试，选择它，则可以在 Reveal 中查看调试该 iOS 程序的界面了。

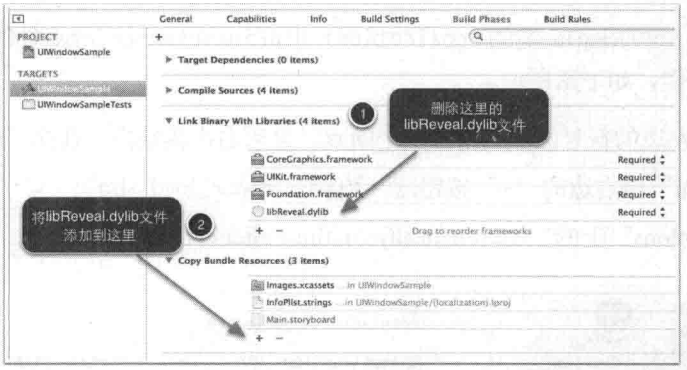
4.2.2 用 Reveal 连接真机调试

要用 Reveal 连接真机调试，我们需要先把 Reveal 的动态链接库上传到真机上。由于 iOS 设备有沙盒存在，所以我们只能将 Reveal 的动态链接库添加到工程中。

选择 Reveal 菜单栏的“Help”→“Show Reveal Library in Finder”选项，可以在“Finder”中显示出 Reveal 的动态链接库 libReveal.dylib，如下图所示。



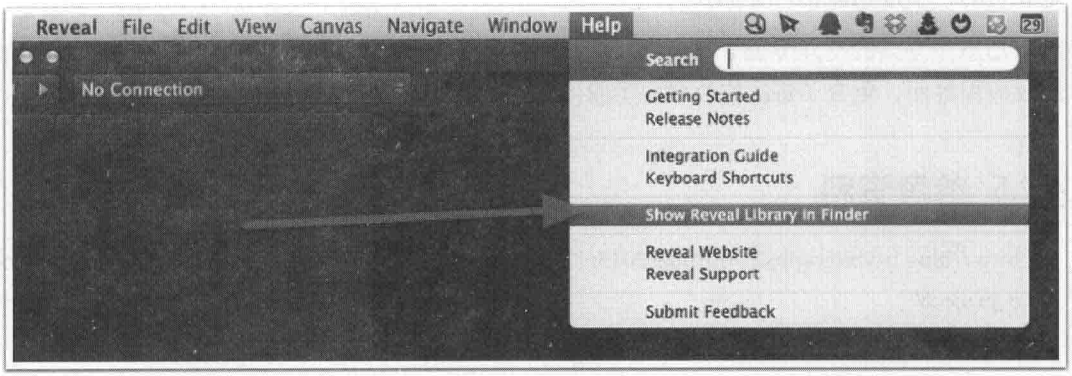
我们将 libReveal.dylib 文件拖动到目标 Xcode 工程中，Xcode 默认将 libReveal.dylib 错误地设置到了“Link Binary With Libraries”下，我们需要进行调整，将其中的“Link Binary With Libraries”删除，然后将其添加到“Copy Bundle Resources”下面，如下图所示。



之后用 Reveal 连接真机的方式和连接模拟器的方式类似，我们只需要把上一节提到的断点 Action 的内容从 `reveal_load_sim` 改成 `reveal_load_dev` 即可。

4.2.3 用 Reveal 调试其他应用界面

如果你的设备“越狱”了，那么还可以用 Reveal 来“调试”其他应用界面，什么时候会有这种奇怪的需求呢？当我们想学习别人是如何实现界面效果的时候。iOS 设备的目录/Library/MobileSubstrate/DynamicLibraries 下存放着所有在系统启动时就需要加载的动态链接库，所以我们只需要将 Reveal 的动态链接库上传到该目录即可，如下图所示。



对于“越狱”的设备，我们可以在安装 OpenSSH 之后，用 scp 来上传该文件。具体步骤如下：

1. 将 libReveal.dylib 上传到/Library/MobileSubstrate/DynamicLibraries。
2. 如果 libReveal.dylib 没有执行权限，用 `chmod+x libReveal.dylib` 命令，给其增加执行权限。
3. 执行 `killall SpringBoard` 重启桌面。

下图是我学习网易新闻客户端 iPad 版时的截图。

4.2.4 总结

除了 Reveal(http://revealapp.com/)外，PonyDebugger(https://github.com/square/PonyDebugger)和 Spark Inspector（http://sparkinspector.com/）也是同类型的界面调试工具，可以在程序运行时动态调试 iOS 应用界面。其中 PonyDebugger 是免费并且开源的，Reveal 和 Spark Inspector

是收费的，不过功能相对更加强大。

对于动态或复杂的交互界面，手写 UI 是不可避免的。通过 Reveal，我们可以方便地调试和修改应用界面，免去了每次修改代码后重新启动的痛苦。

4.2.5 参考资料

- http://blog.ittybittyapps.com/blog/2013/11/07/integrating-reveal-without-modifying-your-xcode-project/
- http://c.blog.sina.com.cn/profile.php?blogid=cb8a22ea89000gtw/
- http://wufawei.com/2013/12/use-reveal-to-inspect-ios-apps/

第5章 移动统计工具 Flurry

Flurry（http://www.flurry.com/）是一家专门为移动应用提供数据统计和分析的公司，其全球的市场份额在移动统计工具中处于领先地位。合理地使用 Flurry 能够帮助读者更加方便地做应用的统计和分析工作。

本章会介绍 Flurry 的基本功能，以及如何做自定义的统计，最后会将其与业界其他同类工具做对比。

5.1 Flurry 简介

Flurry 的数据统计分析 SDK 支持的平台包括 iPhone、iPad、Android、Windows Phone、Java ME 和 BlackBerry。使用 Flurry 服务的公司包括 eBay、Yahoo、Hulu 和 Skype 等超过 11 万家公司，涉及的应用超过 36 万个。

利用 Flurry 提供的分析平台，可以很容易地统计出应用的使用情况，例如：

1. 每天（每周或每月）登录用户数，应用使用次数。
2. 每天（每周或每月）新用户数，活跃用户数。
3. 用户的所在地、年龄、性别的分布情况。

Flurry 也可以自动统计出移动设备的分类情况，例如：

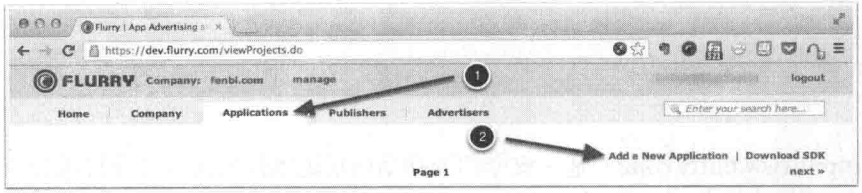
1. 使用 3G、WiFi 会话的比例。
2. 使用 iOS 系统各版本（例如 iOS6.0、iOS7.0 等）的比例。
3. 使用 iOS 各种设备（例如 iPhone4、iPhone5 等）的比例。

除了上面介绍的自动统计项目，Flurry SDK 也提供了统计用的相关 API，便于我们针对自己产品的特点，做有针对性的统计。例如统计应用中某个按钮的按下次数，或者网络请求的平均响应时间等。

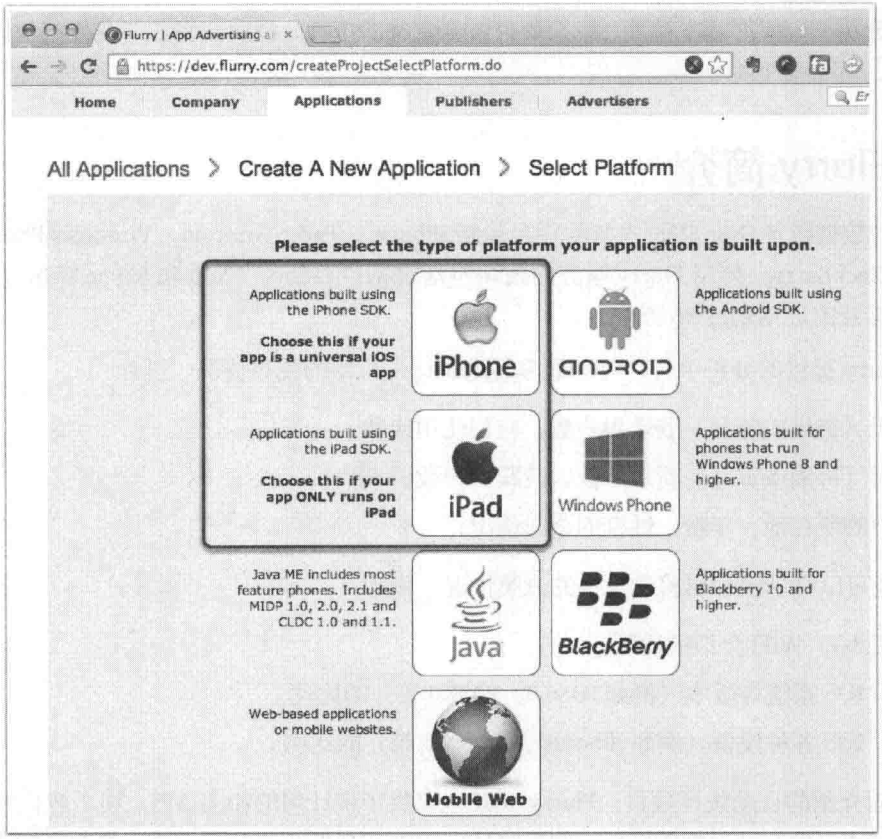
5.2 Flurry 的基本使用

5.2.1 注册和下载对应 SDK

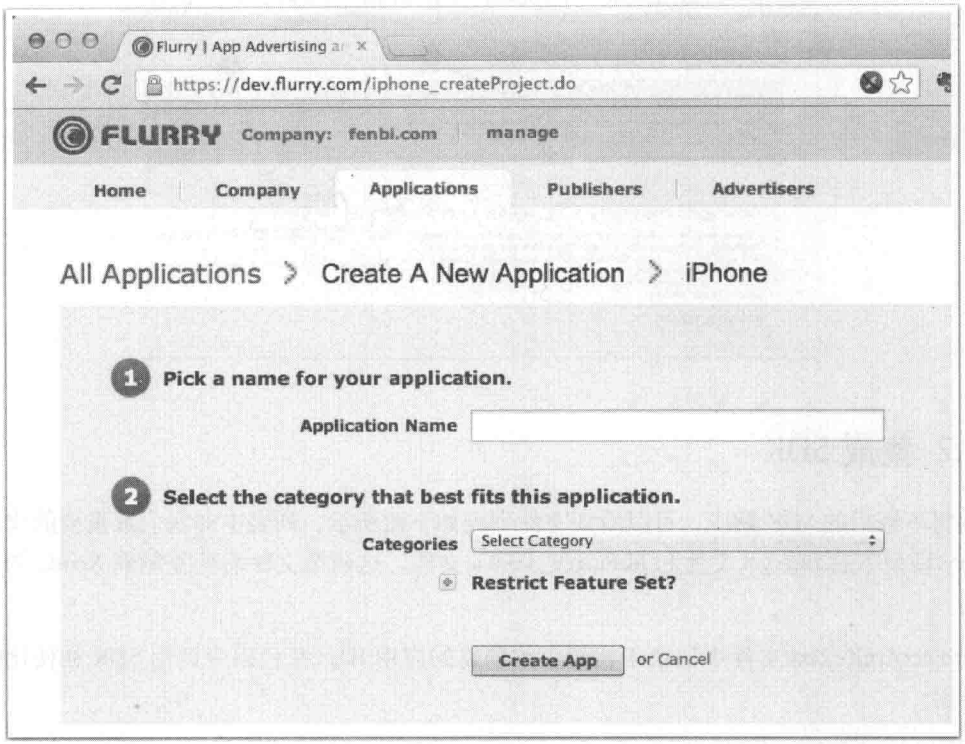
使用 Flurry 前，需要先到其官方网站（http://www.flurry.com/）注册账号。然后登录到 Flurry 后台，依次选择“Applications”→“Add a New Application”，增加一个需要统计分析的应用，如下图所示。



然后，在接下来的界面中根据你的应用类型，选择 iPhone 或 iPad 应用，如下图所示。

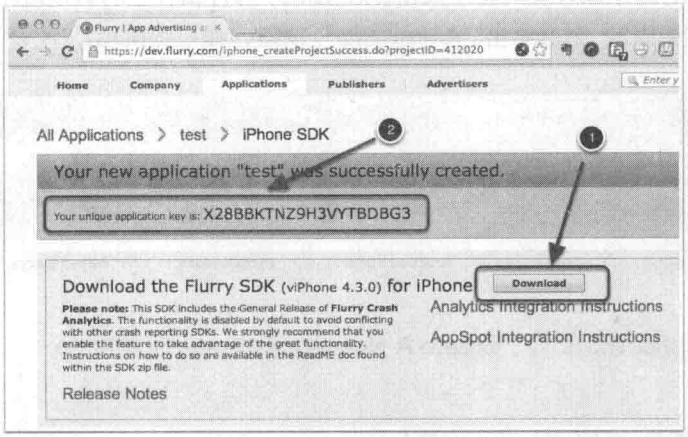


接着，填入应用的名字和分类（名字仅用作在 Flurry 后台和自己的其他应用区分，不需要和应用的真实名字相同），之后单击“Create App”按钮，如下图所示。



到此，我们就成功在后台创建了一个新的应用统计和分析项目。

单击下图中的“Download”按钮，可以下载需要集成在应用中的 SDK。而下图中提示 2 中的 Key：`X28BBKTNZ9H3VYTBDBG3` 则是我们在集成时用于标记自己应用的 ID。

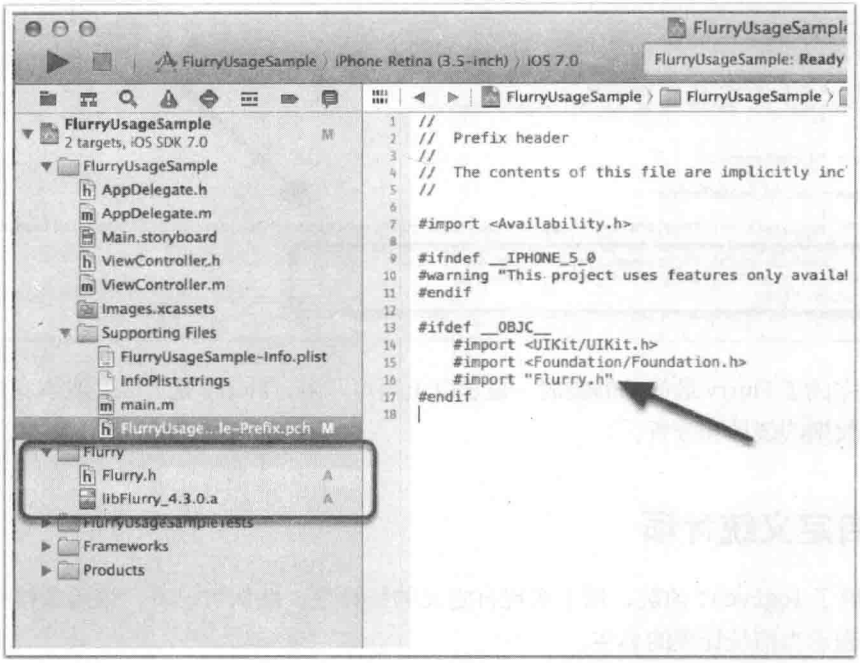


5.2.2 集成 SDK

我们将下载后的 SDK 解压，可以看到文件列表如下图所示。列表中对我们最重要的文件是 Flurry 目录下的 flurry.h 文件和 libFlurry_4.3.0.a 文件。这两个文件需要复制到 Xcode 的工程中去。

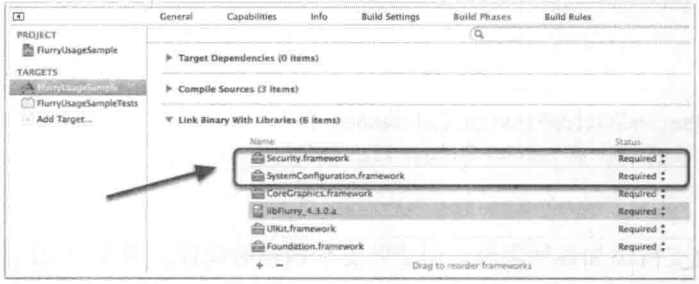
而 ProjectApiKey.txt 文件中记录了我们之前创建的应用 ID，在代码中调用 SDK 初使化时需要使用。

接着我们打开 Xcode 工程，将之前下载解压的 Flurry 目录拖动添加到工程中，同时在工程的.pch 文件中加上 `#import "Flurry.h"`，如下图所示。

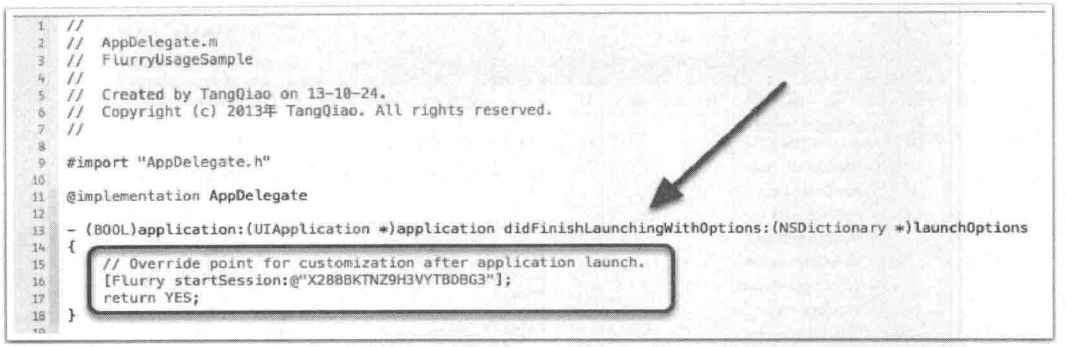


接着我们在 Link Binary With Libraries 中加入两个 framework，如下图所示。

- Security.framework
- SystemConfiguration.framework



接着我们打开 AppDelegate.m，在`- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions` 方法中，加入代码`[Flurry startSession:@"X28BBKTNZ9H3VYTBDBG3"];`，如下图所示。



```
//
//  AppDelegate.m
//  FlurryUsageSample
//
//  Created by TangQiao on 13-10-24.
//  Copyright (c) 2013年 TangQiao. All rights reserved.
//

#import "AppDelegate.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    [Flurry startSession:@"X28BBKTNZ9H3VYTBDBG3"];
    return YES;
}
```

这样，就完成了 Flurry 最简单的集成。通过以上短短几步，Flurry 就可以帮我们完成应用的基本使用数据的统计和分析。

5.2.3 自定义统计项

Flurry 提供了 logEvent 函数，用于实现自定义的统计项。默认情况下，该函数接受一个参数，用于表示当前统计项的名字。

例如我们的界面中有两个按钮，我们想统计它们各自被用户点击的次数，则可以用如下代码实现。在该代码中，我们定义了两个自定义的统计项，名字分别为 First Button Pressed 和 Second Button Pressed。

```
- (IBAction)firstButtonPressed:(id)sender {
    [Flurry logEvent:@"First Button Pressed"];
}


- (IBAction)secondButtonPressed:(id)sender {
    [Flurry logEvent:@"Second Button Pressed"];
}
```

logEvent 函数也支持添加各种参数，用于做更加精细的统计，例如，我们想统计用户在同一个页面点击不同按钮的次数分布，看哪些按钮更加常用，则统计代码可以这样实现：

```
- (IBAction)firstButtonPressed:(id)sender {
    [Flurry logEvent:@"Button Pressed"
      withParameters:@{@"target": @"first"}];
}


- (IBAction)secondButtonPressed:(id)sender {
    [Flurry logEvent:@"Button Pressed"
      withParameters:@{@"target": @"second"}];
```

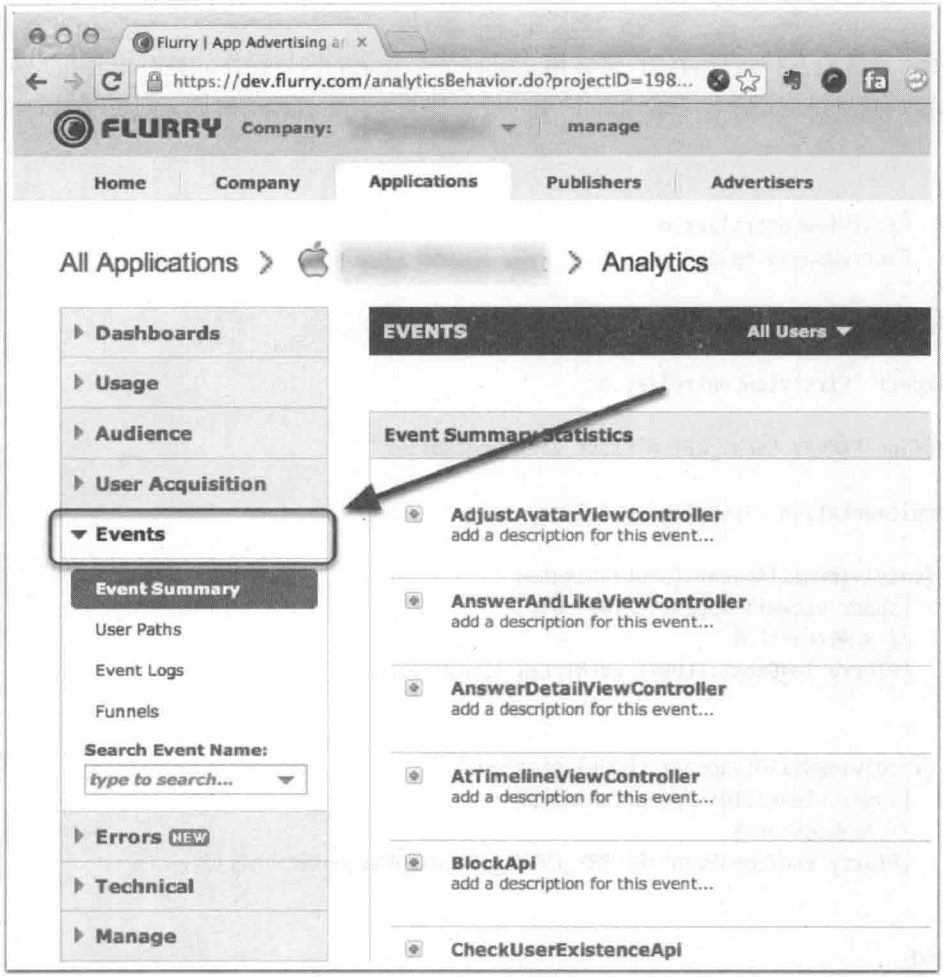
```
}
```

logEvent 函数也支持统计时间，常常用来统计某个复杂的网络操作的耗时或者用户对于某些界面的响应时间。例如，我们想统计用户停留在某个提示界面的时间，则可以用如下代码完成：

```
//
//  FirstViewController.m
//  FlurryUsageSample
//
//  Created by TangQiao

#import "FirstViewController.h"

#define FLURRY_EVENT_KEY @"First View Controller"

@implementation FirstViewController

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    // 开始统计时间
    [Flurry logEvent:FLURRY_EVENT_KEY timed:YES];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    // 结束统计时间
    [Flurry endTimedEvent:FLURRY_EVENT_KEY withParameters:nil];
}

@end
```

5.2.4 查看统计结果

完成上面的自定义统计的代码，待应用发布后，我们就可以从后台的 Events 栏中看到相应的统计结果了，如下图所示。



5.2.5 统计 Crashlog

Flurry 从 4.2.3 版开始，支持应用的 Crashlog 统计。只需要在 AppDelegate.m 文件中，在调用 `startSession` 方法之前，调用 `setCrashReportingEnabled:YES` 即可：

```
[Flurry setCrashReportingEnabled:YES];
[Flurry startSession:@"YOUR\_API\_KEY"];
```

注意，一定要在 `startSession` 之前调用 `setCrashReportingEnabled`，否则将无法记录 Crashlog 信息！切记!!

之后你就可以从后台管理界面的 Errors 项中，获得应用的 Crashlog 信息。

5.3 对比和总结

5.3.1 和其他统计分析平台的对比

和著名的统计工具 Google Analytics（http://www.google.com/analytics/）相比，Flurry 的优点是：

1. Flurry 专门针对移动端做了许多优化，例如统计流量就小很多。
2. Flurry 没有被“墙”的问题。

Flurry 的缺点是：

1. Google Analytics 的统计功能相对更强大一些。
2. Google Analytics 可以和网页版的统计数据做整合。

和国内的分析平台友盟（http://www.umeng.com/）相比，Flurry 的优点是：

1. 使用 Flurry 的应用相对更多。根据 Flurry 和友盟的官方数据，有超过 36 万个应用使用 Flurry（http://www.flurry.com/big-data.html），有超过 18 万个应用使用友盟（http://www.umeng.com/analytics/）。
2. Flurry 是国外的公司，保持独立和专注，数据安全性更高；友盟现在已经被阿里收购（http://tech.163.com/13/0426/16/8TDB6H1N00094MOK.html），当用户的应用涉及的业务和阿里有类似或重合的时候，那么该统计数据有潜在的安全性问题。这也是京东不支持用支付宝付款的原因。

Flurry 的缺点是：

1. 友盟因为是中国公司，所以对国内开发者非常友善，相关的文档或界面都是中文的。而 Flurry 并不提供中文的后台管理界面或相关文档。
2. Flurry 的服务器在国外，在响应速度上应该比友盟慢一些。但在测试中，Flurry 服务器都保证了 500ms 左右的响应时间，还是比较好的。

5.3.2 总结

本章介绍了 Flurry 的基本功能及如何做自定义的统计，最后与业界同类工具做了对比。

我将相关示例代码整理到 Github 上，地址是：https://github.com/tangqiaoboy/FlurryUsageSample，合理地使用 Flurry 能够帮助读者更加方便地做应用的统计和分析工作。

第6章 崩溃日志记录工具 Crashlytics

Crashlytics（http://try.crashlytics.com/）是专门为移动应用开发者提供的保存和分析应用崩溃信息的专业工具。Crashlytics 有专业的崩溃信息分析，可以和现有的项目管理系统（如 Redmine、Jira 等）集成，有助于我们追踪和管理应用缺陷，持续改进产品质量。

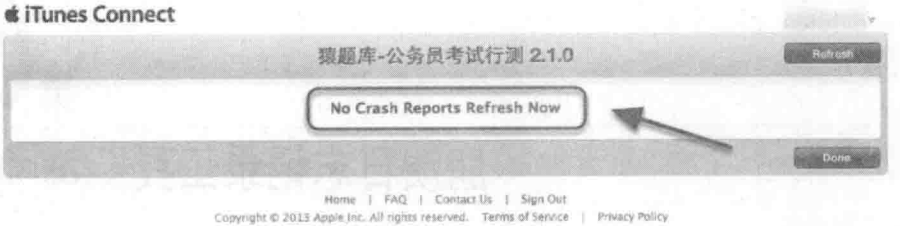
6.1 Crashlytics 简介

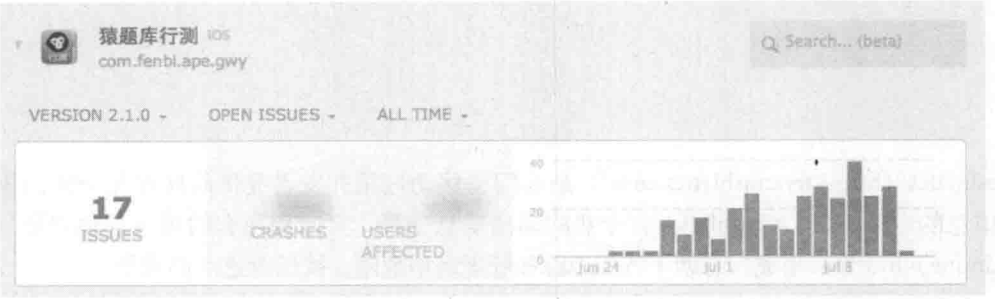
Crashlytics 成立于 2011 年，它的使用者包括支付工具 Paypal、点评工具 Yelp、照片分享工具 Path 和团购工具 Groupon 等移动应用。

2013 年 1 月，Crashlytics 被 Twitter 收购（http://www.crashlytics.com/blog/crashlytics-is-joining-forces-with-twitter/），成为又一个成功的创业产品。被收购之后，由于没有了创业公司的不稳定因素，我们更有理由使用它来分析应用崩溃信息。

使用 Crashlytics 的好处有：

1. Crashlytics 不会漏掉任何应用崩溃信息。拿我的应用举例来说，在 iTunes Connect 的后台查看不到任何崩溃信息。但是用户确实会通过微博或者客服电话反馈应用崩溃的情况。而这些在 Crashlytics 中都可以统计到。下面的截图分别显示了我的某应用在苹果 iTunes Connect 后台和 Crashlytics 中的差别。





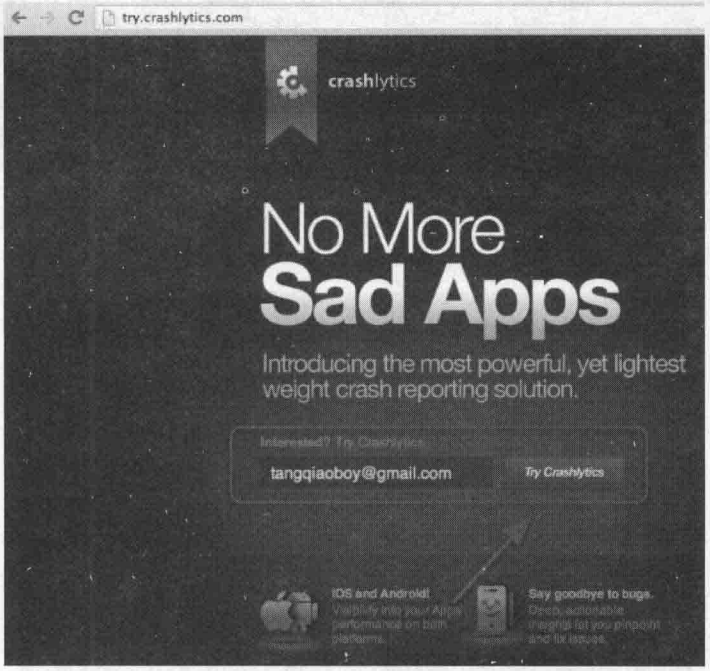
2. Crashlytics 可以象 Bug 管理工具那样，管理这些崩溃日志。例如：Crashlytics 会根据每种类型的 Crash 的出现频率，以及影响的用户量来自动设置优先级。对于每种类型的 Crash，Crashlytics 除了会像一般的工具提供 Call Stack 外，还会显示更多相关的有助于诊断的信息，例如设备是否“越狱”，当时的内存量，当时的 iOS 版本等。对于修复掉的 Crash 日志，可以在 Crashlytics 的后台将其关掉。下图中显示的是一个我的早期应用的崩溃记录，在修复后，我将其更新为已修复状态。

3. Crashlytics 可以每天和每周将崩溃信息汇总发到你的邮箱，所有信息一目了然。

6.2 Crashlytics 的使用

申请账号

Crashlytics的服务是免费提供的，但是用户并不能直接注册使用，需要先申请。打开 Crashlytic 的官方网站（http://try.crashlytics.com/），输入自己的邮箱申请使用，如下图所示。



提交完邮箱之后，你的申请会放在 Crashlytics 的申请队列中，网页跳转到如下图所示的界面。在这个界面的右侧，你可以提供更多有效信息来让 Crashlytics 优先处理你的申请，建议大家都填上自己更多的信息。

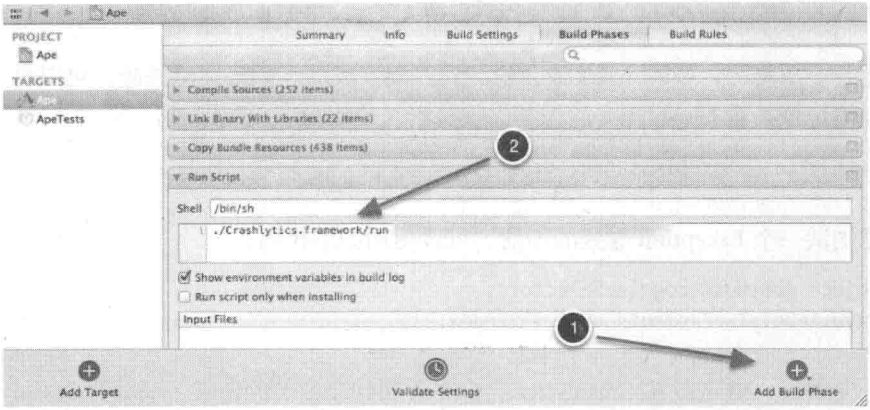
如果顺利，通常一两天左右，你就会收到 Crashlytics 发来的邮件，提示你申请通过，如下图所示，通过邮件链接跳转到注册界面，填写密码即可完成注册。

设置工程

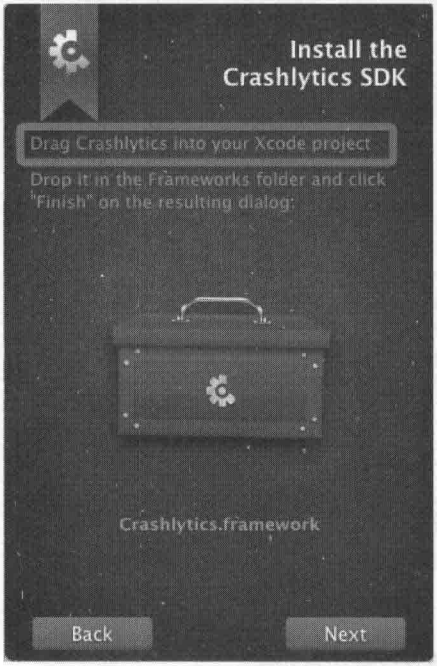
在使用 Crashlytics 前需要对原有的 Xcode 工程进行配置，在这一点上，Crashlytics 做得比其他任何我见过的 SDK 提供商都体贴。因为 Crashlytics 专门做了一个 Mac 端的应用来帮助你进行配置，所以，在配置前你先需要去https://www.crashlytics.com/downloads/xcode下载该应用。

应用下载后，运行该应用并登录账号。选择应用中的“New App”按钮，然后选择自己

要增加 Crashlytics 的工程，Crashlytics 的应用会提示你为工程增加一个 Run Script，如果你不知道如何添加，这里有一个帮助文档（http://www.runscriptbuildphase.com/?utm_source=desktopapp&utm_medium=setup&utm_campaign=mac）。添加好之后的工程截图如下所示。



接着，Crashlytics 的本地应用会提示你将 Crashlytics 相关的 framework 拖到工程中，如下图所示。



按照提示做完之后，就到了最后一步了，在 AppDelegate 的 `didFinishLaunchingWithOptions` 方法中加入如下代码：

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

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
   [Crashlytics startWithAPIKey:@"your identify code"];
}
```

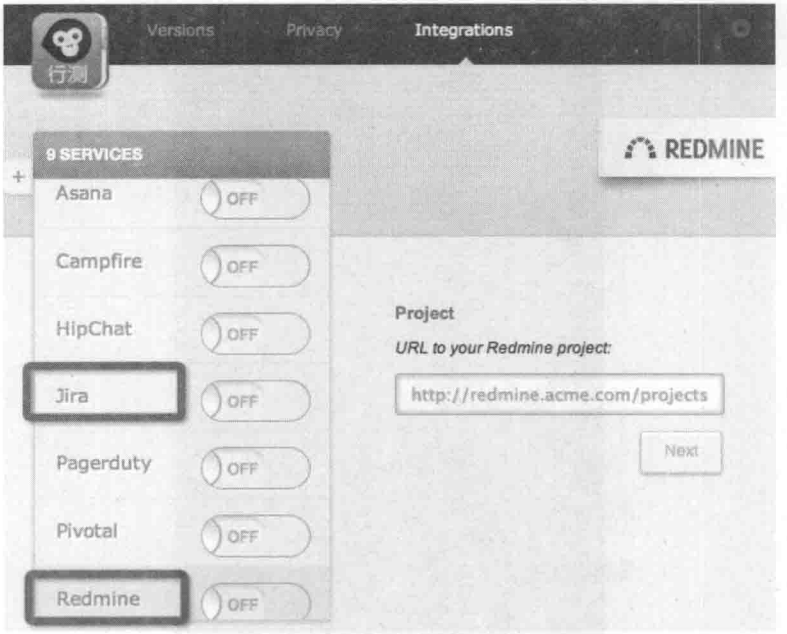
之后，运行一下程序，Crashlytics 就会检测到你设置成功。如果你感兴趣，可以自己手工触发一个崩溃记录，看 Crashlytics 能否帮你收集到。使用如下代码即可：

```
[[Crashlytics sharedInstance] crash];
```

如果你想测试一个 Exception 导致的崩溃，可以使用如下代码：

```
[NSObject doesNotRecognizeSelector];
[arrayWithOnlyTwoElements objectAtIndex:3];
```

应用对外发布后，就可以在 Crashlytics 后台查看和修改相关的记录。另外，Crashlytics 还支持将数据导入到其他项目管理工具（例如 Redmine 或 Jira），如下图所示，配置都非常简单。



6.3 实现原理和使用体会

实现原理

在原理上，Crashlytics 通过以下两步完成崩溃日志的上传和分析：

1. 提供应用的 SDK，你需要在应用启动时调用其 SDK 来设置你的应用。SDK 会集成到你的应用中，完成 Crash 信息的收集和上传。
2. 修改工程的编译配置，加入一段代码，在你每次工程编译完成后，上传该工程对应的 dSYM 文件。研究过手工分析 Crash 日志的读者应该知道，只有通过该文件，才能将 Crash 日志还原成可读的 Call Stack 信息。

使用体会

为了进一步方便开发者设置相应的工程，Crashlytics 提供了 Mac 端的应用程序，帮助你检测相关工程是否正确设置，并且提供相应的帮助信息。后来我还发现，该程序还会自动帮你升级 Crashlytics 的 SDK 文件。虽然这一点很体贴，但是我个人觉得还是不太友好。因为毕竟修改 SDK 会影响应用编译后的内部逻辑，在没有任何通知的情况下升级，我都无法确定 Crashlytics 有没有“干坏事”。不过国外的服务，特别是象 Twitter 这种知名度相对较大的公司提供的服务要有“节操”得多，所以在这一点上我还是比较放心的。

使用 Crashlytics 可以让你摆脱管理应用崩溃记录的烦恼。并且帮助你找出应用的一些重大隐藏性 Bug。例如我之前写的一个应用就有过一个缓存过期的问题，只有当缓存过期时才会触发这个 Bug，这样的问题在测试人员那里很难触发，因为他们不可能了解你的应用内部实现的细节。通过 Crashlytics，我清楚地了解到应用 Crash 的数量和位置，结合自己的开发经验，就很容易找到问题所在了。

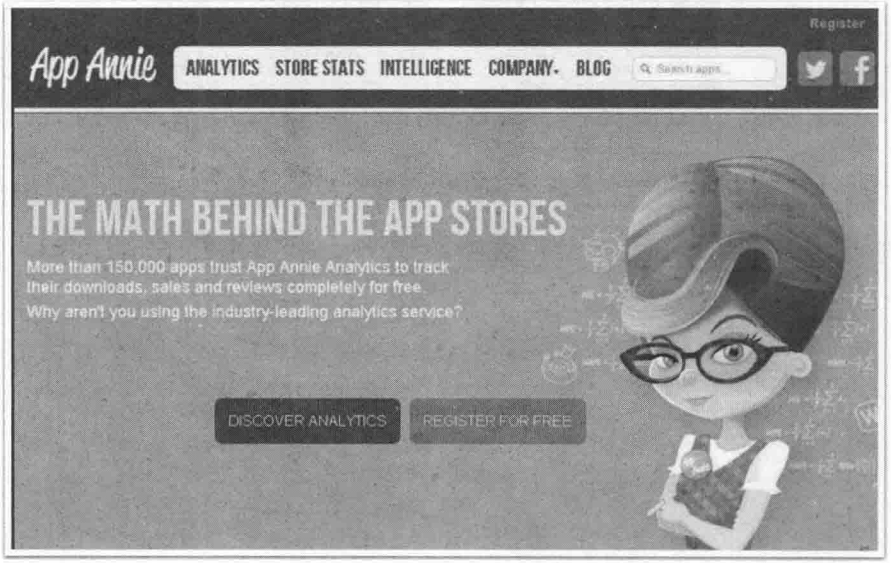
Crashlytics 本身的官方文档（http://support.crashlytics.com/knowledgebase/topics/14721-crashlytics-sdk-for-ios ）也非常健全，如果你在使用中遇到任何问题，也可以去查看详细的文档。

第 7 章

App Store 统计工具 App Annie

App Annie（http://www.appannie.com/）是一个 App Store 数据的统计分析工具。该工具可以统计 App 在 App Store 的下载量、排名变化、销售收入情况及用户评价等信息。

7.1 App Annie 简介



苹果官方的 iTunes Connect 提供的销售数据统计功能比较差，例如只能保存最近 30 天的详细销售数据、界面丑陋、无法查看应用的排名历史变化情况等。

App Annie 是一个专门为开发者提供的，针对 App Store 的统计分析工具，用它来统计应用在 App Store 的下载量、排名变化、销售收入情况及用户评价等信息，会方便得多。

App Annie 实现的原理是：通过你配置的管理账号，向 iTunes Connect 请求获得你的应用的相关数据，包括每日下载量、用户的评分数据，以及销售数据等。

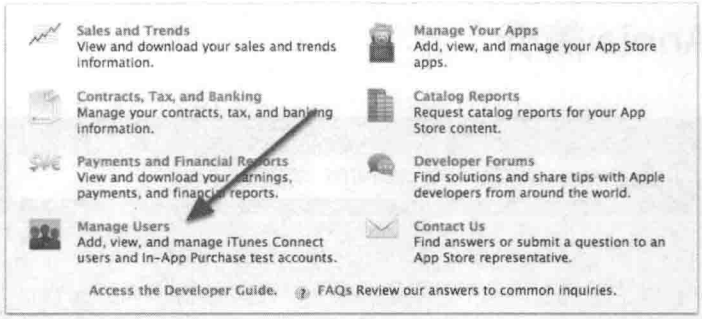
7.2 App Annie 的使用

注册 Sales 类型的账号

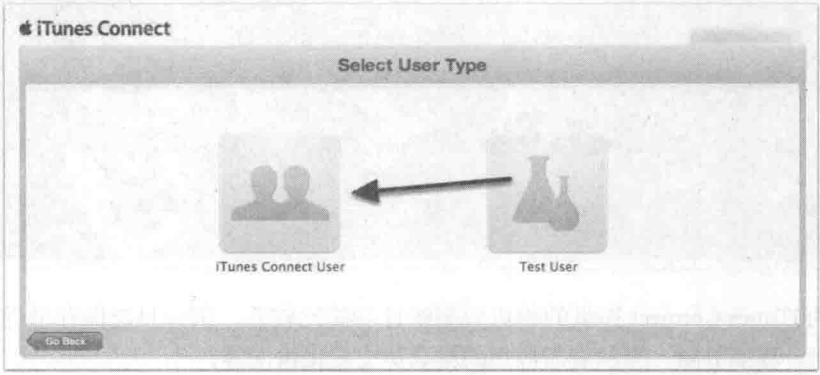
使用 App Annie，首先需要在苹果官方的 iTunes Connect 中配置一个 Sales 类型的账号。因为默认的开发者账号是 Admin 级的权限，该权限是非常高的，可以修改应用的价格或者直接下架商品。如果将这个账号直接配置在 App Annie 中，虽然不影响其获得相关数据，但是有一定的账号安全风险。

配置该账号的详细步骤如下：

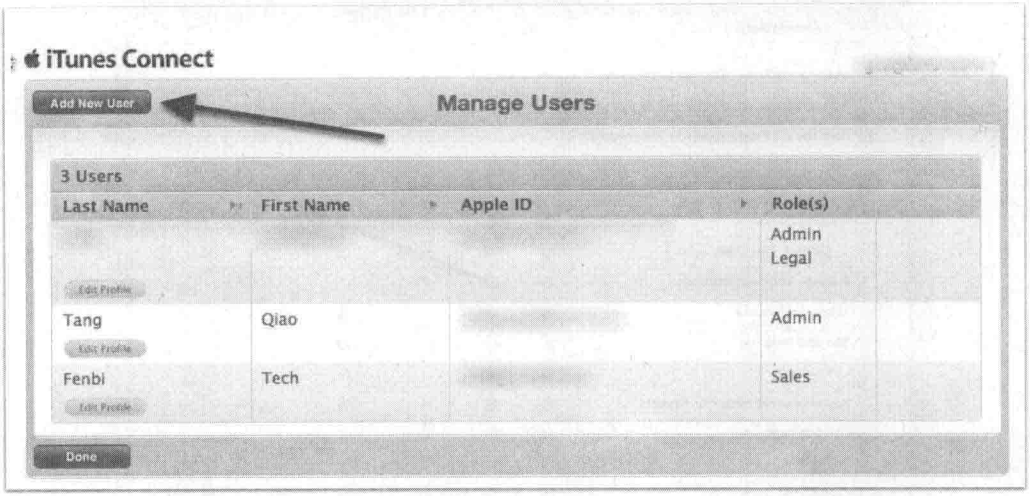
1. 登录 iTunes Connect，选择“Manager Users”，如下图所示。



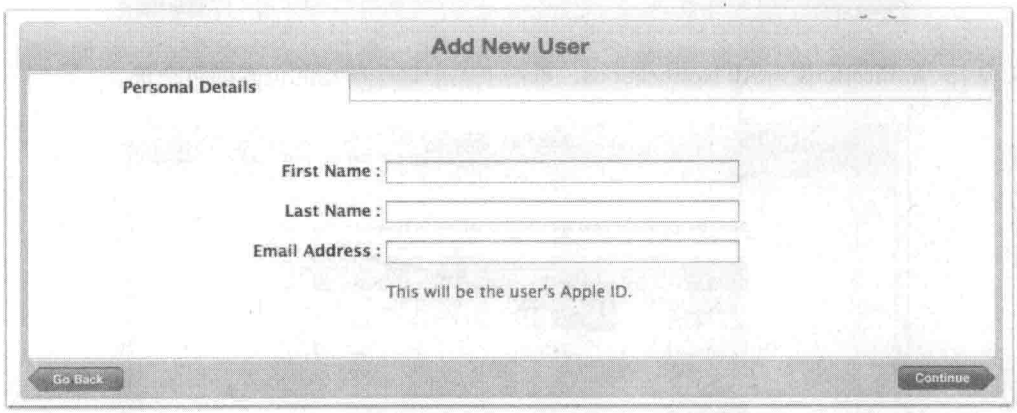
2. 选择“iTunes Connect User”，如下图所示。



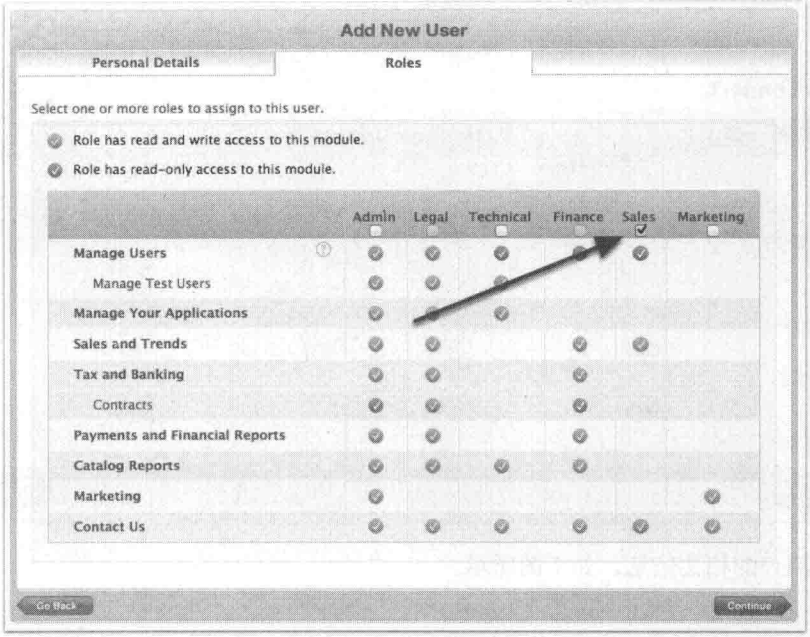
3. 单击“Add New User”按钮，如下图所示。



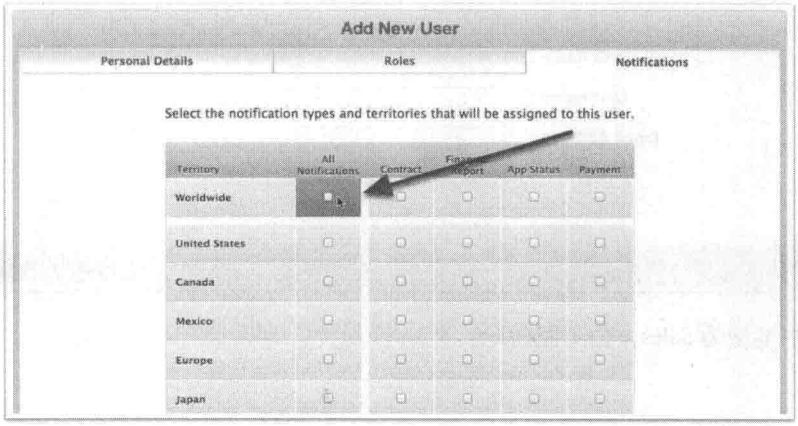
4. 填写新用户的相关信息，如下图所示。



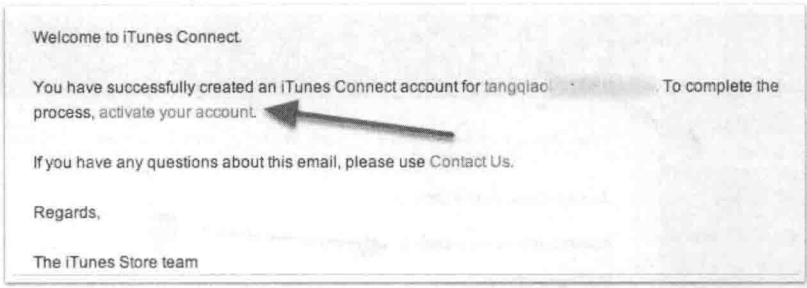
5. 勾选用户类型为 Sales，如下图所示。



6. 选择 Notifications 为 All Notifications。单击下图中箭头所指的位置即可全选。



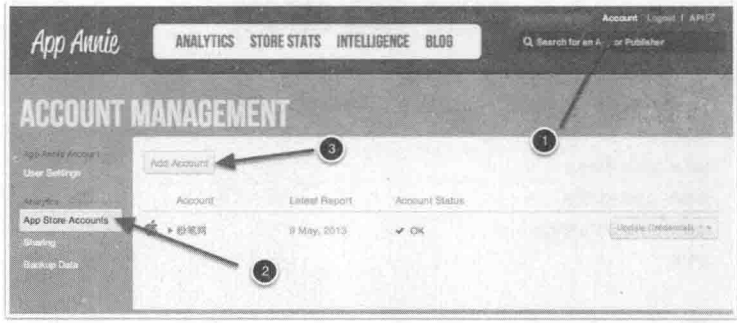
7. 然后，邮箱中会收到 iTunes Connect 发来的激活邮件。单击邮件中的激活链接，即可进入账号注册界面，之后注册账号即可激活，如下图所示。如果该邮箱已经注册过 Apple Id，则会进入到登录界面，登录后即可激活。



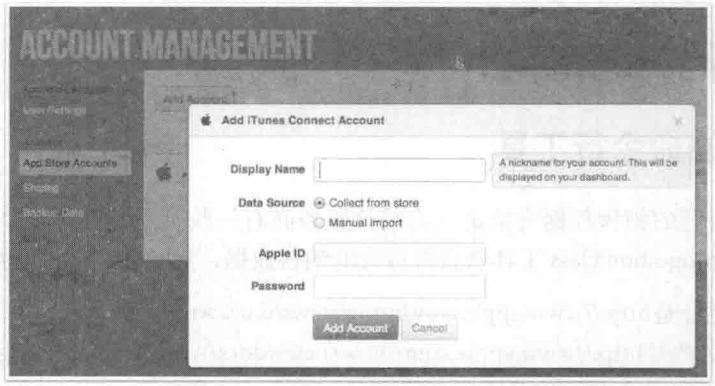
7.3 App Annie 账号的注册及配置

打开 App Annie 的官方网站http://www.appannie.com/，注册步骤和一般网站的步骤一样。注册完成之后的配置步骤如下：

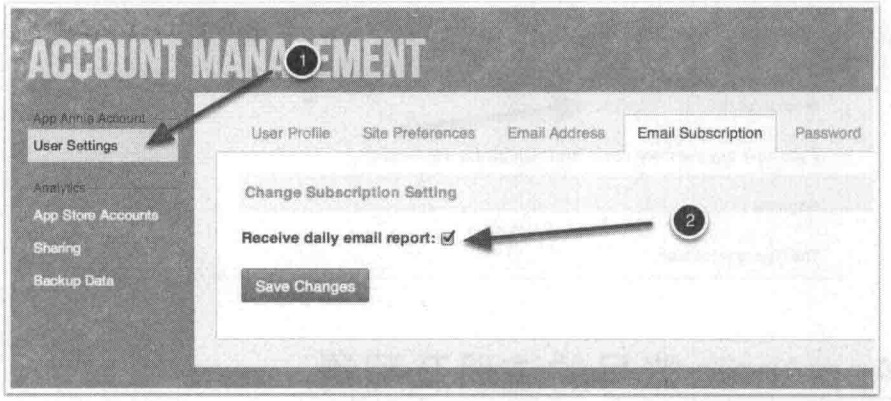
1. 在设置页面中增加 iTunes Connect 账号，如下图所示。



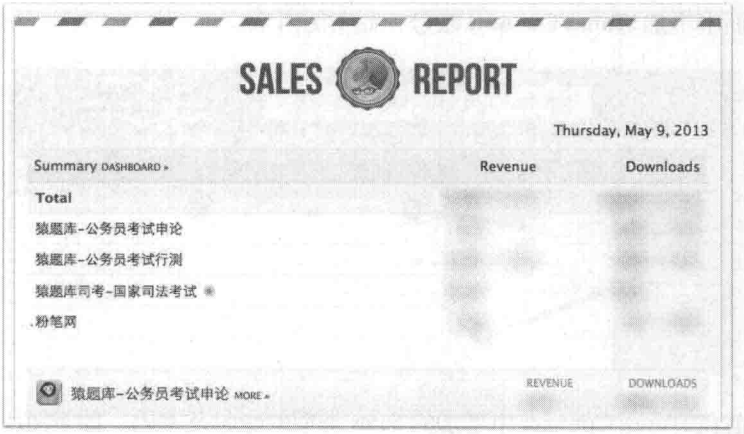
2. 填写你之前在 iTunes Connect 中增加的 Sales 类型的账号及密码，如下图所示。



3. 在 User Setting 中勾选接收每日 Report 选项，如下图所示。



4. 这样，每天就可以收到 AppAnnie 发来的相关统计邮件了。下面是一封粉笔网的销售报告邮件截图。



7.4 和其他工具的对比

7.4.1 官方的命令行工具

如果你觉得将自己的销售数据交给第三方统计服务商有一些不太安全，可以考虑使用苹果官方提供的 Autoingestion.class 工具来获得每天的销售数据，然后存到本地的数据库中。

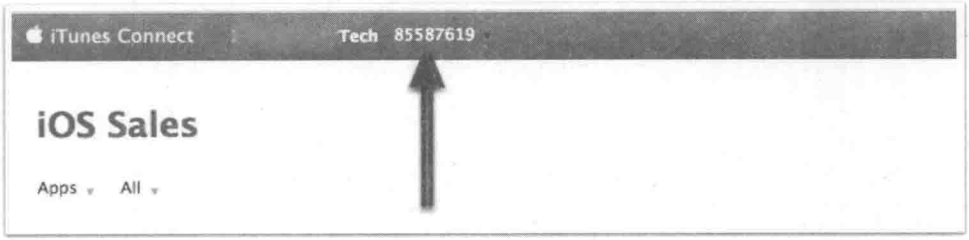
该工具的下载地址是http://www.apple.com/itunesnews/docs/Autoingestion.class.zip，苹果对于该工具的帮助文档是http://www.apple.com/itunesnews/docs/AppStoreReportingInstructions.pdf。

下面介绍一下这个工具的使用方法。下载 Autoingestion.class 之后，切换到 class 文件所在目

录，执行如下命令，即可获得对应的统计数据：

```
java Autoingestion <账号名> <密码> <vendorId> <报告类型> <时间类型> <报告子类型> <时间>
```

其中 vendor Id 在 iTunes Connect 的如下图所示的位置获得，是一个数字 8 开头的序列。



报告类型可选的值是：Sales 或 Newsstand。

时间类型可选的值是：Daily、Weekly、Monthly 或 Yearly。

报告子类型可选的值是：Summary、Detailed 或 Opt-In。

时间以如下的格式给出：YYYYMMDD。

下面是一个示例，它将获得 2013 年 5 月 8 日的日销售摘要数据：

```
java Autoingestion username@fenbi.com password 85587619 Sales Daily Summary 20130508
```

我试用了一下该工具，觉得还是太简陋了一些，它仅仅能够将销售数据备份下来，如果要做 App Annie 那样的统计报表，还需要写不少代码。而且，该工具并不像 App Annie 那样，能提供应用在 App Store 的排名变化情况。虽然可以自己再做抓取，但也是需要工作量的。

7.4.2 其他类似服务

类似 App Annie 这样的服务还有 appFigures（http://appfigures.com）。我试用过之后，发现它不如 App Annie 功能强大。不过作为一个替代方案，也一并介绍给大家。

在 Github 上也有一些开源的统计工具（https://github.com/alexvollmer/itunes-connect），感兴趣的朋友也可以尝试一下。这些工具基本上也就是对苹果的命令行工具的增强，例如增加了将数据导入到数据库中等功能。

7.4.3 功能对比

App Annie 和苹果本身提供的命令行工具虽然都能统计 App Store 的数据，但是二者功能相差悬殊。苹果的命令行工具仅仅能提供销售数据的按日、周、月、年等方式的统计和备份，

而 App Annie 除了以更加良好的界面和交互提供这些功能外，还能跟踪应用的排名变化，以及应用在苹果的各种榜单中所处位置等情况。

建议大家尝试使用 App Annie 或 App Figures 这类统计工具，帮助你方便地查看应用的销售和排名情况。

第8章 Xcode 插件

本章介绍 Xcode 插件管理工具及常见的 Xcode 插件。在编程中合理地使用 Xcode 插件可以提高我们的开发效率。

8.1 Xcode 插件管理工具 Alcatraz

8.1.1 简介

Alcatraz 是一个能帮你管理 Xcode 插件、模板及颜色配置的工具。它可以直接集成到 Xcode 的图形界面中，让你感觉就像在使用 Xcode 自带的功能一样。

8.1.2 安装和删除

使用如下的命令行来安装 Alcatraz：

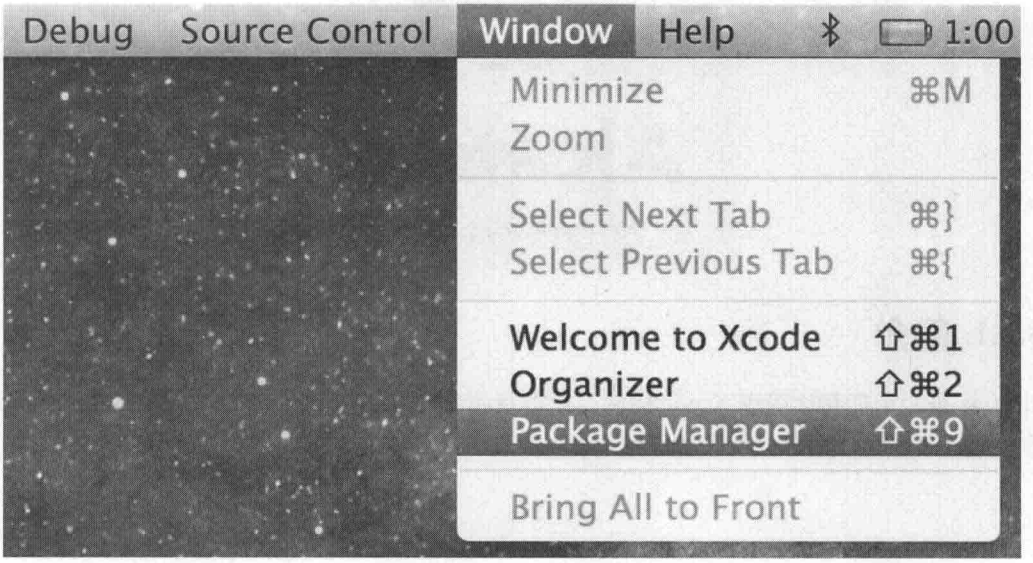
```
mkdir -p ~/Library/Application\ Support/Developer/Shared/Xcode/Plug-ins;
curl -L http://git.io/lOQWeA | tar xvz -C ~/Library/Application\ Support/Developer/
    Shared/Xcode/Plug-ins
```

如果你不想使用 Alcatraz 了，可以使用如下命令来删除：

```
rm -rf ~/Library/Application\ Support/Developer/Shared/Xcode/Plug-ins/Alcatraz.
    xcplugin
rm -rf ~/Library/Application\ Support/Alcatraz
```

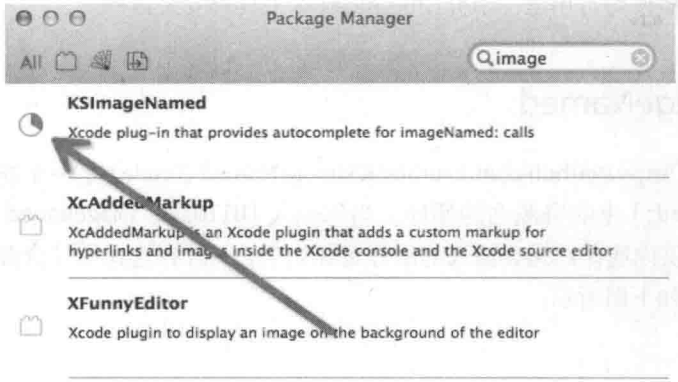
8.1.3 使用

安装成功后重启 Xcode，就可以在 Xcode 的顶部菜单中找到 Alcatraz，如下图所示。



单击“Package Manager”，即可启动插件列表页面，如下图所示。

之后你可以在右上角搜索插件，对于想安装的插件，单击其左边的图标，即可下载安装。如下图所示，正在安装 KImageNamed 插件。



安装完成后，再次单击插件左边的图标，可以将该插件删除。

8.1.4 插件路径

Xcode所有的插件都安装在目录~/Library/Application Support/Developer/Shared/Xcode/Plug-ins/下，你也可以手工切换到这个目录来删除插件。

8.2 关于 Xcode 的插件机制

Alcatraz 当前只支持最新版本的 Xcode 。这其实主要是因为苹果并没有开放插件机制，每次升级 Alcatraz 都要重新适配。如果你看到 Alcatraz 的 Commit Log，你就会发现，Alcatraz 花了几个月时间，才适配到 Xcode 5，这会让插件开发者感到比较难受。

所以作为一款开源并且免费的插件，只支持最新版的 Xcode 可以让开源作者节省大量精力。我们也希望苹果能早日开放 Xcode 的插件机制标准，方便广大的插件开发者构建强大的第三方增强工具。

8.3 常用 Xcode 插件

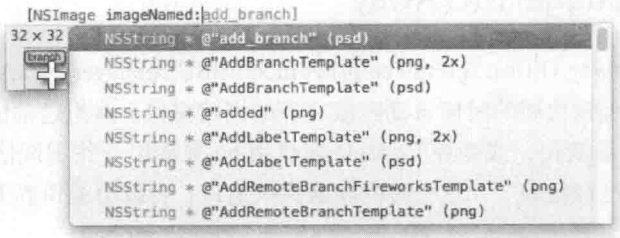
Xcode 是 iOS 的集成开发环境，虽然苹果一直在不断改进 Xcode，但程序员总是有各种新奇的想法和需求，当 Xcode 无法满足他们时，他们就会通过使用插件的方式来为 Xcode 增加新的功能。本节将会给大家介绍一些常用的 Xcode 增强插件。

Xcode 所有的插件都安装在目录 ~/Library/Application Support/Developer/Shared/Xcode/Plug-ins/下，每个插件为一个子目录，你也可以手工切换到这个目录来增加或删除插件。

以下介绍的所有插件均可用上一节介绍的 Alcatraz 工具来安装或删除。

8.3.1 KSImageNamed

KSImageNamed (https://github.com/ksuther/KSImageNamed-Xcode) 是一个能帮助你输入 [UI-Image imageNamed:] 中的资源名的插件。当你输入 [UIImage imageNamed:] 时，会自动弹出上下文菜单，供你选择你需要输入的图片资源名字，另外在选择图片资源时，还可以在左侧预览该资源，如下图所示。



8.3.2 XVim

XVim（https://github.com/JugglerShu/XVim）是一个 Xcode 的 vim 插件，可以在 Xcode 的编辑窗口中开启 vim 模式。

vim 模式最大的好处是可以全键盘操作，可以方便地移动光标，以及复制、粘贴代码。XVim 对于 Xcode 的分栏模式也有很好的支持，与 vim 自带的分栏模式一样，可以用快捷键 `Ctrl + W` 来切换当前编辑的分栏。

8.3.3 FuzzyAutocompletePlugin

FuzzyAutocompletePlugin（https://github.com/FuzzyAutocomplete/FuzzyAutocompletePlugin）允许使用模糊的方式来进行代码自动补全。

举个例子，如果我们要重载 `viewDidAppear:` 方法，那么我们必须依次建入 view、did、appear 才能得到相应的补全信息，使用 FuzzyAutocompletePlugin 之后，我们可以建入 vda（view、did、appear 三个单词的首字母），或任意符合 viewDidAppear 整个单词出现顺序的子串（例如 vdapp, idear 等），即可匹配到该方法。

8.3.4 XToDo

XToDo（https://github.com/trawor/XToDo）是一个查找项目中所有的带有 `TODO`、`FIXME`、`???`、`!!!` 标记的注释。

通常我们在项目开发中，由于种种原因，一些事情需要以后处理，这个时候为了防止遗忘，加上 `TODO` 或 `FIXME` 注释是非常有必要的，但是上线或提交代码前要寻找这些未解决的事项却稍显麻烦。XToDo 可以提供一个汇总的界面，集中显示所有的未完成的 `TODO` 和 `FIXME` 标记。

8.3.5 BBUDebuggerTuckAway

BBUDebuggerTuckAway（https://github.com/neonichu/BBUDebuggerTuckAway）是一个非常小的工具，可以在你编辑代码的时候自动隐藏底部的调试窗口。因为通常情况下，调试的时候是加断点或监控变量变化，或者在 Console 窗口用 po 来输出一些调试信息。如果开始编辑代码了，说明调试已经结束了，这个时候隐藏调试窗口，可以给编辑界面更多空间，方便我们修改代码。

8.3.6 SCXcodeSwitchExpander

SCXcodeSwitchExpander（https://github.com/stefanceriu/SCXcodeSwitchExpander）能帮助你迅速地在 switch 语句中填充枚举类型的每种可能的取值。

例如，当你输入 switch，然后键入 NSTableViewAnimationOptions 类时，该插件会将其可能的取值补全在每一个 case 之后，如下图所示。

```
- (void)test
{
    switch (self.animationOption) {
        case NSTableViewAnimationEffectNone:
            statement
            break;
        case NSTableViewAnimationEffectFade:
            statement
            break;
        case NSTableViewAnimationEffectGap:
            statement
            break;
        case NSTableViewAnimationSlideUp:
            statement
            break;
        case NSTableViewAnimationSlideDown:
            statement
            break;
        case NSTableViewAnimationSlideLeft:
            statement
            break;
        case NSTableViewAnimationSlideRight:
            statement
            break;
            |
        default:
            break;
    }
}
```

8.3.7 deriveddata-exterminator

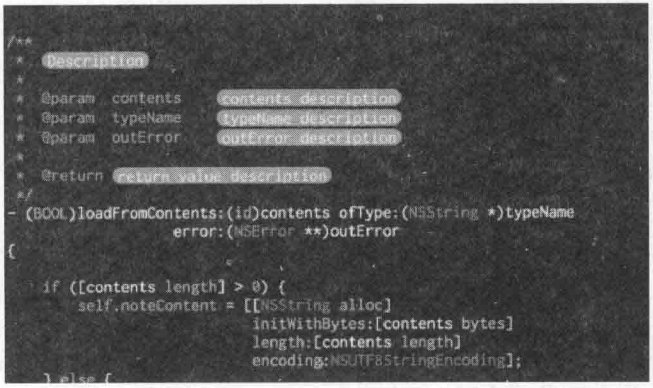
deriveddata-exterminator(https://github.com/kattrali/deriveddata-exterminator)是一个清除 Xcode 缓存目录的插件。

有些时候 Xcode 会出现各种奇怪的问题，最常见的是在某些复杂操作下（例如，在同一个项目中，来回切换到各种分支版本），会造成 Xcode 显示一些编译的错误或警告，但是最终却

又可以编译通过。新手遇到这种问题常常束手无策，而熟悉 Xcode 的人就知道，通常清除 Xcode 缓存就可以解决这类问题。该插件在 Xcode 菜单上增加了一个清除缓存按钮，可以方便地一键清除缓存内容。

8.3.8 VVDocumenter

VVDocumenter（https://github.com/onevcat/VVDocumenter-Xcode）是一个自动生成代码注释的工具，可以方便地将函数的参数名和返回值提取出来，这样结合上一节介绍的 appledoc 命令，就可以方便地将帮助文档输出，如下图所示。

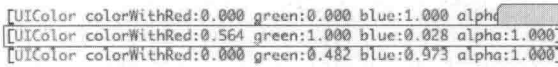


8.3.9 ClangFormat

ClangFormat（https://github.com/travisjeffery/ClangFormat-Xcode）是一个自动调整代码风格（Code Style）的工具。Xcode 本身的代码缩进自动调整功能比较弱，特别是对于 JSON 格式，常常产生非常丑陋的默认缩进效果。ClangFormat-Xcode 可以更好地对代码进行重新排版，并且内置了各种排版风格，也支持自定义风格。

8.3.10 ColorSense

ColorSense（https://github.com/omz/ColorSense-for-Xcode）是一个 UIColor 颜色输入辅助工具，可以帮助你在编写 UIColor 代码时，实时预览相应的颜色，如下图所示。



8.3.11 XcodeBoost

XcodeBoost（https://github.com/fortinmike/XcodeBoost）包含多个辅助修改代码的小功能，比如：

- 可以方便地将.m 文件中方法的定义暴露到对应的.h 文件中。
- 可以在某一个源文件中直接输入正则表达式查找。
- 可以复制粘贴代码时不启用 Xcode 的自动缩进功能（Xcode 的自动缩进经常出问题，造成已经调整好的代码缩进，在粘贴时被 Xcode 调整坏了）。

第9章 其他工具介绍

本章作为之前内容的补充，再简略介绍一些 iOS 开发的效率工具，合理地使用这些工具可以提高我们的工作效率。

9.1 取色工具：数码测色计（DigitalColor Meter）

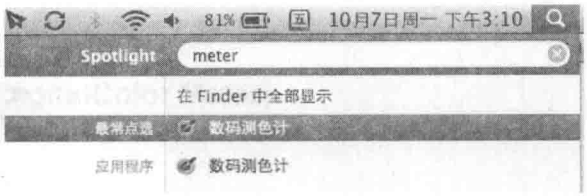
9.1.1 前言

在做 iOS 界面时，由于美术人员提供的多是 PDF 或者是其他图像格式，开发人员常常需要获取图像的具体 RGB 值。而系统自带的数码测色计（DigitalColor Meter）就能完成该任务。本文主要介绍它的使用及其他类似工作。

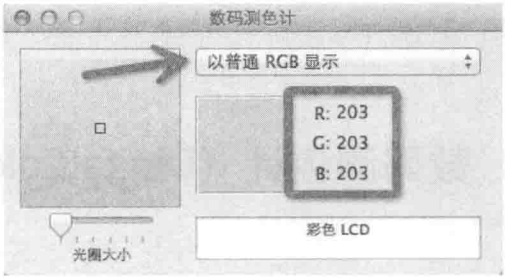
9.1.2 使用介绍

在“应用程序”→“实用工具”界面，我们能找到名为“数码测色计”的程序，如下图所示。

我们也可以在“spotlight”或“alfred”中输入“meter”，这样也可以找到该程序，如下图所示。



数码测色计的应用界面如下，我们将显示方式切换到“以普通 RGB 显示”，即可在界面中看到以 RGB 方式呈现的色块的色值。通过移动鼠标，数码测色计将会实时地显示当前鼠标位置的色值，如下图所示。

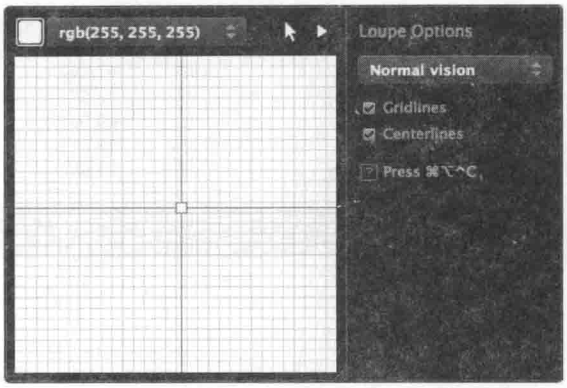


对于目标色值，我们可以通过快捷键 Shift + Cmd + C 将该 RGB 值复制到剪贴板中，随后用 Cmd + V 贴粘到代码中，完成取色值的操作。

9.1.3 其他类似工具：xScope

Mac 系统自带的“数码测色计”程序还是比较简单，业界也有一些做得更加精良的商业取色工具，xScope（http://xscopeapp.com/）就算其中一个。

下图是 xScope 的应用界面。



xScope 在截图上比较方便的地方主要是，可以用快捷键 `Cmd + Alt + Ctrl + C`，将多个取好的色值记录下来。之后将取好的色值直接拖动到代码中，即可生成相应的代码。

xScope 是收费软件，但是，你也可以一直免费使用它的试用版本。使用试用版本，你在粘贴色值时，会多一些未付费的信息附加在色值后面，手工将其删掉即可。

如果你觉得好用，也可以考虑付费购买它。它除了提供取色值的功能外，还提供另外 7 项与 UI 相关的 iOS 开发辅助功能，其中的标尺功能也非常好用。感兴趣的可以访问它的官方网站（http://xscopeapp.com/）了解更多，在此不再赘述。

9.2 其他图形工具

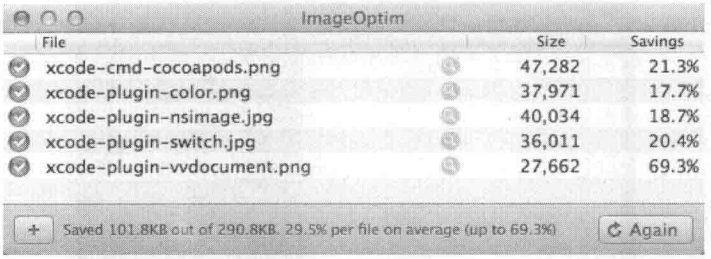
9.2.1 ImageOptim

ImageOptim（http://imageoptim.com/）是一个免费的图像压缩工具。iOS 工程默认使用 `pngcrush` 命令来压缩图片，不过其压缩比率其实不高。对于应用中图片资源比较多的读者，可以尝试使用 ImageOptim 来达到最大的图片压缩效果。

如果你从未尝试过 ImageOptim 一类的图片压缩工具，那么第一次给 IPA 文件瘦身的效果应该是比较惊人的。我个人的经验是，初次使用时，ImageOptim 能减少至少 10% 的应用图片资源占用。

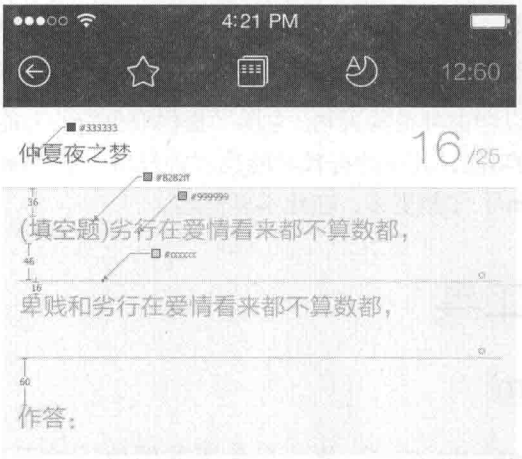
ImageOptim 的实现原理是使用各种开源的图像压缩工具，然后取效果最好的那一个。它尝试的工具包括：PNGOUT、Zopfli、Pngcrush、AdvPNG、extended OptiPNG、JpegOptim、jpegrescan、jpegtran 和 Gifsicle。

安装方式是从其官方网站（http://imageoptim.com/）上下载程序文件，拖动到“应用程序”目录即可。下图是本书的部分配图列表，可以看到使用 ImageOptim 压缩了 29.5% 的体积。



9.2.2 马克鳗

马克鳗（http://www.getmarkman.com/）是国人开发的一款免费标注工具，使用它可以方便地在输出的美术设计稿上标注相应界面元素的大小、颜色、边距、说明等，如下图所示。



9.2.3 Dash

Dash（http://kapeli.com/dash）是一款 API 文档查询及代码片段管理工具。与 Xcode 相比，它的代码查询速度更快，而且支持各种编程语言。它的代码片段支持通过 Dropbox 等网盘进

行同步，使用起来也非常方便。

Dash 可以在 Mac App Store 上免费下载，但是如果要频繁使用还是需要付费，否则每次会有一段时间的等待才可继续使用。

9.2.4 蒲公英

蒲公英（http://www.pgyer.com）是一个应用的内测分发工具，类似苹果的 TestFlight。很多 iOS 的开发者可能使用过 TestFlight，不过因为 TestFlight 的服务器部署在美国，所以在国内使用时，速度其实并不快。对于速度要求比较高的读者，可以尝试使用蒲公英，速度会快一些，使用体验也比较适合中国人。

蒲公英的本质原理，其实是一个应用托管平台，有点类似于 App 网络存储空间。首次使用时，可以把 App 的安装包文件上传到蒲公英，会得到一个二维码，用户使用手机扫描这个二维码就可以安装应用，后续上传的同一个应用，都会被归属到同一个应用页面中进行显示，也可以管理。

蒲公英也提供了 Mac 和 Windows 版本的客户端，对于上传应用频度比较高的读者，使用客户端上传会稍微方便一点。

除了蒲公英外，国内也有 FIR（http://fir.im）提供类似的服务。

9.3 命令行工具

9.3.1 nomad

nomad（http://nomad-cli.com/）是一个方便你操作苹果开发者中心（Apple Developer Center）的命令行工具，利用它可以做的事情包括方便地添加测试设备、更新证书文件、增加 App ID、验证 IAP 的凭证等。

安装方式：

```
gem install nomad-cli
```

安装完后，首先执行 `ios login`，你的 Developer 账号和密码会被它存储到 Keychain 中，之后就可以用命令行来完成各种后台操作了，下面是几个例子。

添加测试设备：

```
ios devices:add "TangQiaos iPhone"=<Device Identifier>
```

更新证书文件：

```
ios profiles:devices:add TangQiao_Profile "TangQiaos iPhone"=<Device Identifier>
```

nomad 还有很多功能，建议大家阅读其官方网站的文档进一步学习。

9.3.2 xctool

xctool（https://github.com/facebook/xctool）是 Facebook 开源的一个 iOS 编译和测试的工具。使用它而不是用 Xcode 的 UI 界面是因为它是一个纯命令行工具。比如：我们可以使用 xctool 在命令行下进行编译和单元测试，然后将测试结果集成到 Jenkins 中，这样就实现了自动化的持续集成。虽然苹果也在 OSX Server 上推出了自己的自动化集成工具 BOT，但其配置和使用现在仍然不太方便。

安装 xctool 可以使用 brew 命令：

```
brew install xctool
```

使用 xctool 编译项目，可以用如下命令：

```
path/to/xctool.sh \
  -project YourProject.xcodeproj \
  -scheme YourScheme \
  build
```

使用 xctool 执行单元测试，可以用如下命令：

```
path/to/xctool.sh \
  -workspace YourWorkspace.xcworkspace \
  -scheme YourScheme \
  test
```

xctool 还有很多功能，建议大家阅读 xctool 官方网站的文档进一步了解更多的功能。

9.3.3 appledoc

appledoc（https://github.com/tomaz/appledoc）是一个从源代码中抽取文档的工具。

对于开发者来说，文档最好和源代码在一起，这样更新起来更加方便和顺手。象 Java 一类的语言本身就自带 javadoc 命令，可以从源代码中抽取文档。而 appledoc 就是一个类似 javadoc 的命令行程序，可以从 iOS 工程的源代码中抽取相应的注释，生成帮助文档。

相对于其他的文档生成工具，appledoc 的优点是：

- 它默认生成的文档风格和苹果的官方文档是一致的。
- appledoc 就是用 objective-c 写的，必要的时候调试和改动也比较方便。

- 它可以生成 docset，并且集成到 xcode 中。集成之后，在相应的 API 调用处，按住 Option 键再单击就可以调出相关的帮助文档。
- 它没有特殊的注释要求，兼容性高。

安装 appledoc 可以直接使用 `brew` 命令：

```
brew install appledoc
```

使用时切换到 iOS 工程目录下，执行以下操作即可，appledoc 会扫描当前路径下的所有文件，然后生成文档放到 doc 目录下。你也可以用 `appledoc -help` 查看所有可用的参数。

```
appledoc -o <output_path> \
--project-name <project_name> \
--project-company <project_company> .
```

II

第二部分：iOS 开发实践

第二部分我们进入实践阶段，讨论在实际深入 iOS 开发后，将会遇到的各种进阶类的问题。

考虑到某些读者一开始学习 iOS 开发时就在 ARC 环境下，所以内存管理知识会比较欠缺，不管是 Objective-C 语言还是 Swift 语言，其内存管理方式都是基于引用计数的，所以我们首先会讨论 iOS 开发中的内存管理。

接着，我们在这个部分将讨论 GCD 的使用、UIWindow 的使用、动态下载系统提供的多种中文字体、应用内支付、基于 UIWebView 的混合编程、iOS 开发的安全性问题、如何基于 CoreText 实现一个复杂排版引擎，以及一些实践小技巧。

第10章 理解内存管理

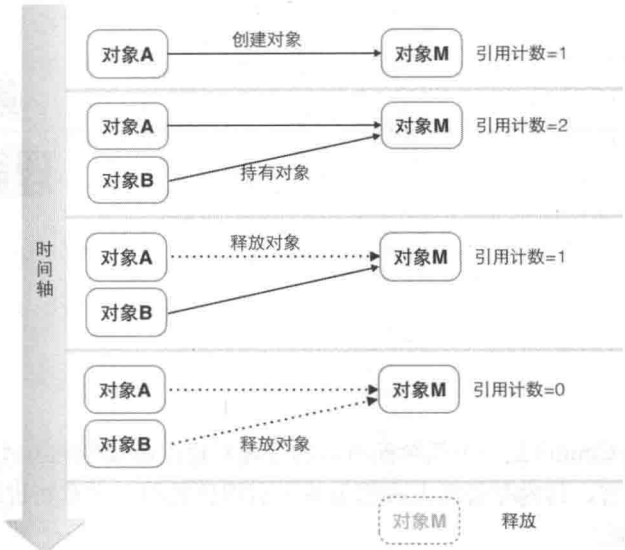
引用计数（Reference Count）是一个简单而有效的管理对象生命周期的方式。不管是 Objective-C 语言还是 Swift 语言，其内存管理方式都是基于引用计数的。本章将讲解这种内存管理方式的特点及注意事项。

10.1 引用计数

10.1.1 什么是引用计数，原理是什么

引用计数可以有效地管理对象生命周期。当我们创建一个新对象的时候，它的引用计数为 1，当有一个新的指针指向这个对象时，我们将其引用计数加 1，当某个指针不再指向这个对象时，我们将其引用计数减 1，当对象的引用计数变为 0 时，说明这个对象不再被任何指针指向了，这个时候我们就可以将对象销毁，回收内存。

由于引用计数简单有效，除了 Objective-C 语言外，微软的 COM（Component Object Model）、C++11（C++11 提供了基于引用计数的智能指针 share_prt）等语言也提供了基于引用计数的内存管理方式，如下图所示。



为了更形象一些，我们再来看一段 Objective-C 的代码。新建一个工程，因为现在默认的工程都开启了自动的引用计数 ARC（Automatic Reference Count），我们先修改工程设置，给 AppDelegate.m 加上-fno-objc-arc 的编译参数，如下图所示。这个参数可以启用手工管理引用计数的模式。

然后，我们输入如下代码，可以通过 Log 看到相应的引用计数的变化。

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(
    NSDictionary *)launchOptions
{
    NSObject *object = [[NSObject alloc] init];
    NSLog(@"Reference Count = %u", [object retainCount]);
    NSObject *another = [object retain];
    NSLog(@"Reference Count = %u", [object retainCount]);
    [another release];
    NSLog(@"Reference Count = %u", [object retainCount]);
    [object release];
    // 到这里时，object的内存被释放了
```

```
    return YES;
}
```

运行结果如下：

```
Reference Count = 1
Reference Count = 2
Reference Count = 1
```

对 Linux 文件系统比较了解的读者可能会发现，引用计数的这种管理方式类似于文件系统里面的硬链接。在 Linux 文件系统中，我们用 ln 命令可以创建一个硬链接（相当于我们这里的 retain），当删除一个文件时（相当于我们这里的 release），系统调用会检查文件的 link count 值，如果这个值大于 1，则不会回收文件所占用的磁盘区域。直到最后一次删除前，系统发现 link count 值为 1，系统才会执行真正的删除操作，把文件所占用的磁盘区域标记成未使用。

10.1.2 我们为什么需要引用计数

从上面那个简单的例子中，我们还看不出引用计数真正的用处。因为该对象的生命期只是在一个函数内，所以在真实的应用场景中，我们在函数内使用一个临时的对象，通常是不需要修改它的引用计数的，只需要在函数返回前将该对象销毁即可。

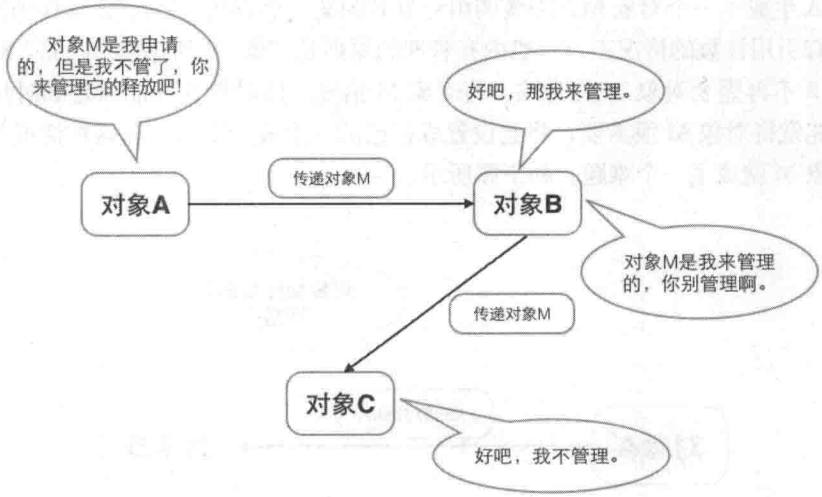
引用计数真正派上用场的场景是在面向对象的程序设计架构中，用于对象之间传递和共享数据。我们举一个具体的例子：

假如对象 A 生成了一个对象 M，需要调用对象 B 的某一个方法，将对象 M 作为参数传递过去。在没有引用计数的情况下，一般内存管理的原则是“谁申请谁释放”，那么对象 A 就需要在对象 B 不再需要对象 M 的时候，将对象 M 销毁。但对象 B 可能只是临时用一下对象 M，也可能觉得对象 M 很重要，将它设置成自己的一个成员变量，在这种情况下，什么时候销毁对象 M 就成了一个难题，如下图所示。

对于这种情况，有一个暴力的做法，就是对象 A 在调用完对象 B 之后，马上就销毁参数对

象 M，然后对象 B 需要将参数另外复制一份，生成另一个对象 M2，然后自己管理对象 M2 的生命期。但是这种做法有一个很大的问题，就是它带来了更多的内存申请、复制、释放的工作。本来一个可以复用的对象，因为不方便管理它的生命期，就简单地把它销毁，又重新构造一份一样的，实在太影响性能，如下图所示。

如下图所示，我们还有另外一种办法，就是对象 A 在构造完对象 M 之后，始终不销毁对象 M，由对象 B 来完成对象 M 的销毁工作。如果对象 B 需要长时间使用对象 M，就不销毁它，如果只是临时用一下，则可以用完后马上销毁。这种做法看似很好地解决了对象复制的问题，但是它强烈依赖于 A、B 两个对象的配合，代码维护者需要明确地记住这种编程约定。而且，由于对象 M 的申请是在对象 A 中，释放在对象 B 中，使得它的内存管理代码分散在不同对象中，管理起来也非常费劲。如果这个时候情况再复杂一些，例如对象 B 需要再向对象 C 传递对象 M，那么这个对象在对象 C 中又不能让对象 C 管理。所以这种方式带来的复杂性更大，更不可取。



所以引用计数很好地解决了这个问题，在参数 M 的传递过程中，哪些对象需要长时间使用

这个对象，就把它的引用计数加 1，使用完了之后再把引用计数减 1。所有对象都遵守这个规则的话，对象的生命期管理就可以完全交给引用计数了。我们也可以很方便地享受到共享对象带来的好处。

10.1.3 不要向已经释放的对象发送消息

有些读者想测试当对象释放时，其 retainCount 是否变成了 0，他们的试验代码如下：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(
    NSDictionary *)launchOptions
{
    NSObject *object = [[NSObject alloc] init];
    NSLog(@"Reference Count = %u", [object retainCount]);
    [object release];
    NSLog(@"Reference Count = %u", [object retainCount]);
    return YES;
}
```

但是，如果你真的这么做，你得到的输出结果可能是以下这样：

```
Reference Count = 1
Reference Count = 1
```

我们注意到，最后一次输出，引用计数并没有变成 0。这是为什么呢？因为该对象的内存已经被回收，而我们向一个已经被回收的对象发了一个 retainCount 消息，所以它的输出结果应该是不确定的，如果该对象所占的内存被复用了，那么就有可能造成程序异常崩溃。

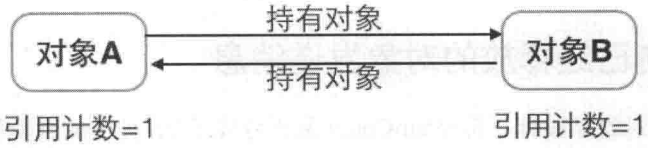
那为什么在这个对象被回收之后，这个不确定的值是 1 而不是 0 呢？这是因为当最后一次执行 release 时，系统知道马上就要回收内存了，就没有必要再将 retainCount 减 1 了，因为不管减不减 1，该对象都肯定会被回收，而对象被回收后，它的所有的内存区域，包括 retainCount 值也变得没有意义。不将这个值从 1 变成 0，可以减少一次内存的操作，加速对象的回收。

拿我们之前提到的 Linux 文件系统举列，Linux 文件系统下删除一个文件，也不是真正地在文件的磁盘区域进行抹除操作，而只是删除该文件的索引节点号。这也和引用计数的内存回收方式类似，即回收时只做标记，并不抹除相关的数据。

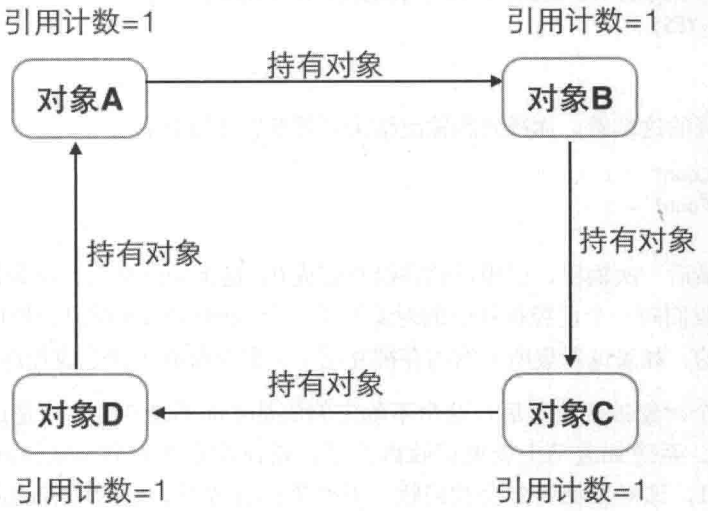
10.1.4 循环引用（reference cycles）问题

引用计数这种管理内存的方式虽然很简单，但是有一个比较大的瑕疵，即它不能很好地解决循环引用问题。如下图所示，对象 A 和对象 B，相互引用了对方作为自己的成员变量，只有当自己销毁时，才会将成员变量的引用计数减 1。因为对象 A 的销毁依赖于对象 B 的销毁，

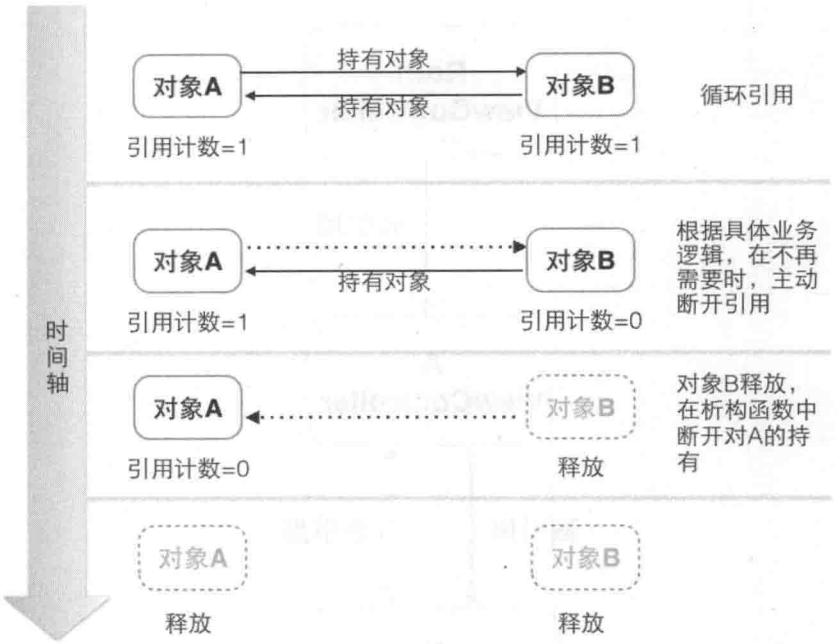
而对象 B 的销毁与依赖于对象 A 的销毁，这样就造成了我们称为循环引用（reference cycles）的问题，即使在外界已经没有任何指针能够访问到它们了，它们也无法被释放。



不止两个对象时存在循环引用问题，多个对象间依次持有，形式一个环状，也可以造成循环引用问题，而且在真实编程环境中，环越大就越难被发现。下图是 4 个对象形成的循环引用问题。

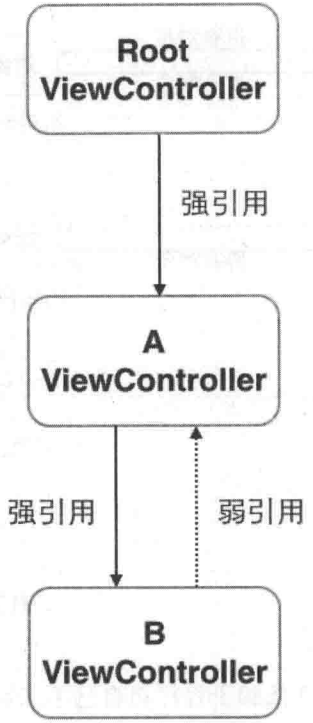


解决循环引用问题主要有两个办法，第一个办法是我明确知道这里会存在循环引用，在合理的位置主动断开环中的一个引用，使得对象得以回收，如下图所示。



不过，主动断开循环引用这种操作依赖于程序员自己手工显式地控制，相当于回到了以前“谁申请谁释放”的内存管理年代，它需要程序员自己有能力发现循环引用，并且知道在什么时机断开循环引用回收内存（这通常与具体的业务逻辑相关），所以这种解决方法并不常用，更常见的是使用弱引用（weak reference）的办法。

弱引用虽然持有对象，但是并不增加引用计数，这样就避免了循环引用的产生。在 iOS 开发中，弱引用通常在 delegate 模式中使用。举个例子来说，两个 ViewController A 和 B，ViewController A 需要弹出 ViewController B，让用户输入一些内容，当用户输入完成后，ViewController B 需要将内容返回给 ViewController A。这个时候，View Controller 的 delegate 成员变量通常是一个弱引用，以避免两个 ViewController 相互引用对方造成循环引用问题，如下图所示。

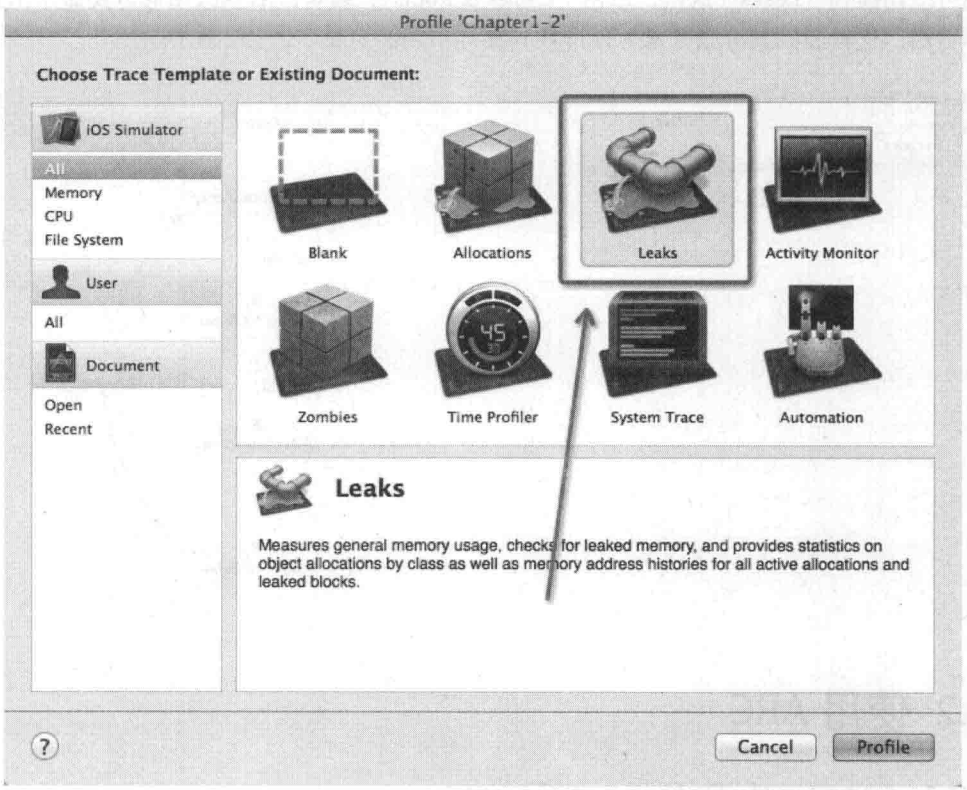


10.1.5 使用 Xcode 检测循环引用

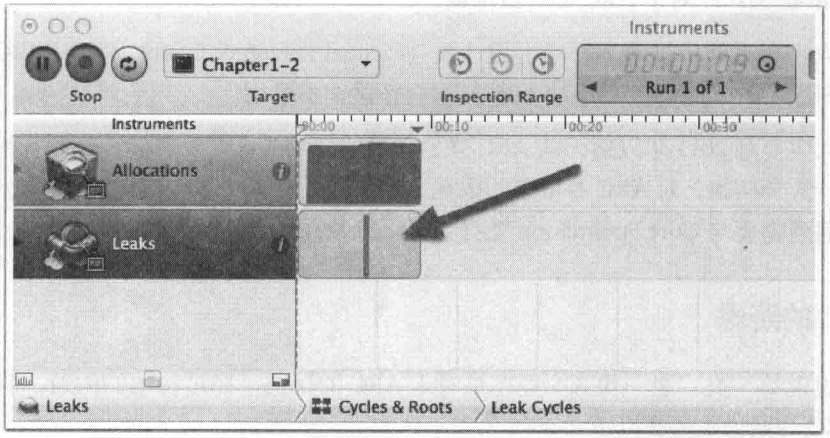
Xcode 的 Instruments 工具集可以很方便地检测循环引用。为了测试效果，我们在一个测试用的 ViewController 中填入以下代码，该代码中的 firstArray 和 secondArray 相互引用了对方，构成了循环引用。

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    NSMutableArray *firstArray = [NSMutableArray array];
    NSMutableArray *secondArray = [NSMutableArray array];
    [firstArray addObject:secondArray];
    [secondArray addObject:firstArray];
}
```

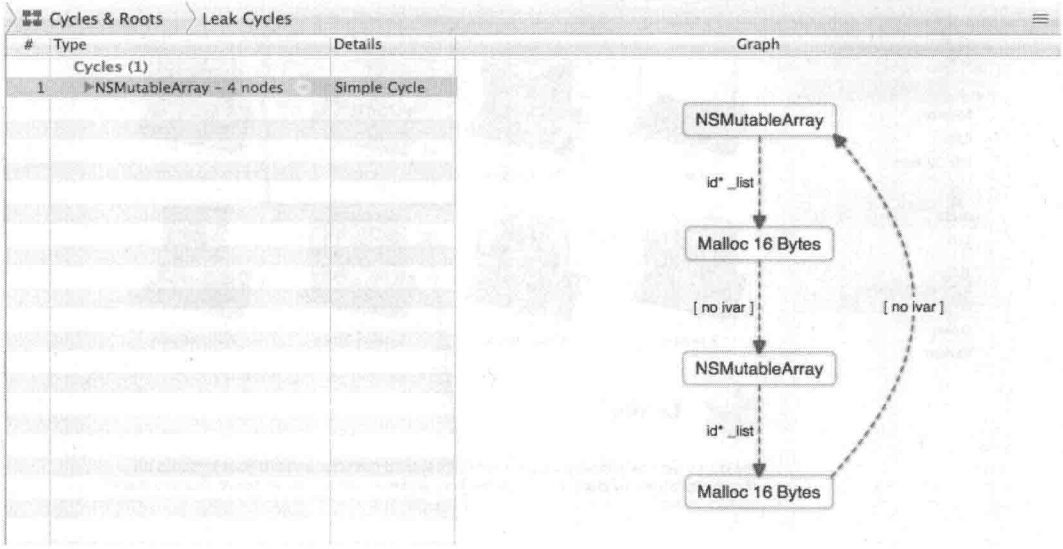
在 Xcode 的菜单栏选择“Product”→“Profile”，然后选择“Leaks”，再单击右下角的“Profile”按钮开始检测，如下图所示。



这个时候 iOS 模拟器会运行起来，我们在模拟器里进行一些界面的切换操作。稍等几秒钟，就可以看到 Instruments 检测到了我们的这次循环引用。Instruments 中会用一条红色的线条来表示一次内存泄漏的产生，如下图所示。



我们可以切换到“Leaks”这栏，单击“Cycles & Roots”，就可以看到以图形方式显示出来的循环引用。这样我们就可以非常方便地找到循环引用的对象了，如下图所示。



10.2 使用 ARC

10.2.1 Automatic Reference Count

自动引用计数（Automatic Reference Count，简称 ARC），是苹果在 WWDC 2011 年大会上提出的用于内存管理的技术。ARC 技术直到今天，仍然被不少人误解。误解主要分为两类：1. 对于 ARC 的疑虑；2. 对于 ARC 的盲目依赖。

第一类的人主要是经历过手工管理引用计数（Manual Reference Count，简称 MRC）时代的老一代 iOS 程序员，他们主要是对 ARC 技术有怀疑，不敢用。

第二类的人主要是 2011 年以后，从 ARC 开始学习的新的 iOS 开发者们，他们有些人完全不知道引用计数为何物，对 ARC 有很强的依赖，但是不知道 ARC 内部的原理，过于依赖 ARC 使得他们遇到需要与 Core Foundation 类打交道，以及循环引用的问题时，显得一筹莫展。

对于 ARC 的疑虑

我们来先说说第一类：老一代 iOS 程序员对于 ARC 的疑虑，经过和他们沟通，我了解到他们对于 ARC 的疑虑主要在以下几点：

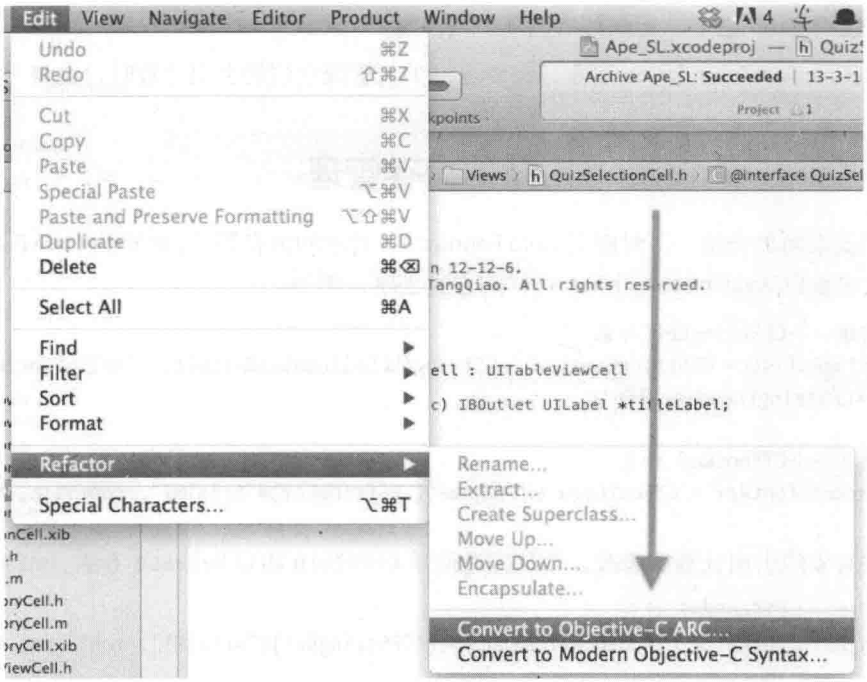
1. 担心这个技术方案不靠谱。苹果大多数时候的技术方案都是比较靠谱的，但也有一些技

术方案有很多“坑”，例如 storyboard。

2. 原有的项目在非 ARC 环境下运行良好，担心迁移成本或引入新的问题。原有的第三方库可能也没有兼容 ARC，担心有迁移成本。
3. 苹果以前手工管理内存需要非常小心，稍微不注意应用程序就崩溃了。有过这段经历的 iOS 开发老手，心理上还是觉得自己手工管理内存更踏实一些。
4. 使用 ARC 需要了解 ARC 的一些细节，还需要引入 _bridge 等新的关键字，学习成本还是有的。
5. 以为 ARC 只能支持 iOS5.0 以上版本，这是非常大的误解。

对于上面提到 5 点问题，我认为相应的回答如下：

1. ARC 是 WWDC 2011 大会时提出的技术，离现在已经很多年了，而且苹果现在将 MacOS 上的垃圾回收机制废弃（Deprecated），采用 ARC 替代，无疑证明了 ARC 已经成熟。
2. 确实有一些迁移成本，但苹果在 Xcode 中专门集成了迁移工具，成本已经非常小了。如下图所示就是 Xcode 集成的将非 ARC 工程转换成 ARC 工程的工具。另外，为了兼容第三方的非 ARC 开源库，你也可以在工程中随意使用编译参数-fno-objc-arc，这个参数允许对部分文件关闭 ARC。



3. 手工管理内存虽然踏实，但是很容易发生泄露。常常在开发完成后，需要使用 Instruments 来检测泄露。但用了 ARC 后，基本不会出现泄露了，我在开发粉笔网 iPhone 客户端时，

由于使用了 ARC，花三个月开发完的应用，用 Instruments 检测后，没有发现任何内存泄漏问题。这在没有使用 ARC 的工程中是不可想象的。

4. ARC 确实有学习成本，但是非常值得学习，学成后能省不少开发精力。
5. 虽然 ARC 是与 iOS5 一同推出，但是由于 ARC 的实现机制是在编译期完成，所以使用 ARC 之后应用仍然可以支持 iOS4.3。稍微需要注意的是，如果要在 ARC 开启的情况下支持 iOS4.3，需要将 weak 关键字换成 __unsafe_unretained。

所以我希望大家都能在项目中使用 ARC，一旦你感受到它带来的好处，你就离不开它了。它也能让你从繁琐的内存管理代码中解放出来，用更多精力关注代码结构、设计模式而不是底层的内存管理。

对于 ARC 的盲目依赖

ARC 能够解决 iOS 开发中 90% 的内存管理问题，但是另外还有 10% 的内存管理，是需要开发者自己处理的，这主要就是与底层 Core Foundation 对象交互的那部分，底层的 Core Foundation 对象由于不在 ARC 的管理下，所以需要自己维护这些对象的引用计数。

对于 ARC 盲目依赖的 iOS 新人们，由于不知道引用计数，他们的问题主要体现在：

1. 过度使用 block 之后，无法解决循环引用问题。
2. 遇到底层 Core Foundation 对象，需要自己手工管理它们的引用计数时，显得一筹莫展。

10.2.2 Core Foundation 对象的内存管理

下面我们就来简单介绍一下对底层 Core Foundation 对象的内存管理。底层的 Core Foundation 对象，大多数以 XxxCreateWithXxx 这样的方式创建，例如：

```
// 创建一个CFStringRef 对象
CFStringRef str= CFStringCreateWithCString(kCFAllocatorDefault, “hello world",
    kCFStringEncodingUTF8);

// 创建一个CTFontRef 对象
CTFontRef fontRef = CTFontCreateWithName((CFStringRef)@"ArialMT", fontSize, NULL);
```

对于这些对象的引用计数的修改，要相应地使用 CFRetain 和 CFRelease 方法。如下所示：

```
// 创建一个CTFontRef 对象
CTFontRef fontRef = CTFontCreateWithName((CFStringRef)@"ArialMT", fontSize, NULL);

// 引用计数加1
CFRetain(fontRef);
// 引用计数减1
```

```
CFRelease(fontRef);
```

对于 CFRetain 和 CFRelease 两种方法，读者可以直观地认为，它们与 Objective-C 对象的 retain 和 release 方法等价。[1]

所以对于底层 Core Foundation 对象，我们只需要延续以前手工管理引用计数的办法即可。

除此之外，还有另外一个问题需要解决。在 ARC 下，我们有时需要将一个 Core Foundation 对象转换成一个 Objective-C 对象，这个时候我们需要告诉编译器，转换过程中的引用计数需要如何调整。这就引入了与 bridge 相关的关键字，以下是这些关键字的说明：

- __bridge：只做类型转换，不修改相关对象的引用计数，原来的 Core Foundation 对象在不用时，需要调用 CFRelease 方法。
- __bridge_retained：类型转换后，将相关对象的引用计数加 1，原来的 Core Foundation 对象在不用时，需要调用 CFRelease 方法。
- __bridge_transfer：类型转换后，将该对象的引用计数交给 ARC 管理，Core Foundation 对象在不用时，不再需要调用 CFRelease 方法。

我们根据具体的业务逻辑，合理使用上面的三种转换关键字，就可以解决 Core Foundation 对象与 Objective-C 对象相对转换的问题了。

[1] 在本书的“基于 CoreText 的排版引擎”相关章节，由于大量使用了底层的 CoreText 技术，读者也能看到大量相关的手工管理 Core Foundation 对象内存的代码。

第11章 掌握GCD

GCD是苹果开发的一个多核编程的解决方法，GCD和其他的多线程技术方案相比，使用起来更加简单和方便。

11.1 GCD简介

Grand Central Dispatch（GCD）在MacOS X10.6（雪豹）中首次推出，并随后被引入到了iOS4.0中。GCD和其他的多线程技术方案，如NSThread、NSOperationQueue、NSInvocationOperation等技术相比，使用起来更加方便。

让我们来看一个编程场景。我们要在iPhone上做一个下载网页的功能，该功能非常简单，就是在iPhone上放置一个按钮，单击该按钮时，显示一个转动的圆圈，表示正在进行下载，下载完成之后，将内容加载到界面上的一个文本控件中。

使用GCD前

虽然功能简单，但是我们必须把下载过程放到后台线程中，否则会阻塞UI线程显示。所以，如果不用GCD，我们需要写如下三个方法：

- someClick方法是单击按钮后的代码，可以看到我们用`NSInvocationOperation`建了一个后台线程，并且放到`NSOperationQueue`中。后台线程执行download方法。
- download方法处理下载网页的逻辑。下载完成后用`performSelectorOnMainThread`执行download_completed方法。
- download_completed进行clear up的工作，并把下载的内容显示到文本控件中。

这三个方法的代码如下。可以看到，虽然“开始下载”→“下载中”→“下载完成”这三个步骤是整个功能的三步。但是它们被切分成了三块。因为它们是三个方法，所以还需要传递数据参数。如果是复杂的应用，数据参数很可能就不像本例子中的 NSString 那么简单了，另外，下载放到 Model 的类中来做，而界面的控制放到 View Controller 层来做，这使得本来就分开的代码变得更加零散，代码的可读性大大减弱。

```
static NSOperationQueue * queue;

- (IBAction)someClick:(id)sender {
    self.indicator.hidden = NO;
    [self.indicator startAnimating];
    queue = [[NSOperationQueue alloc] init];
    NSInvocationOperation * op = [[[NSInvocationOperation alloc] initWithTarget:self
         selector:@selector(download) object:nil] autorelease];
    [queue addOperation:op];
}

- (void)download {
    NSURL * url = [NSURL URLWithString:@"http://www.youdao.com"];
    NSError * error;
    NSString * data = [NSString stringWithContentsOfURL:url encoding:
        NSUTF8StringEncoding error:&error];
    if (data != nil) {
        [self performSelectorOnMainThread:@selector(download_completed:) withObject:
            data waitUntilDone:NO];
    } else {
        NSLog(@"error when download:%@", error);
        [queue release];
    }
}

- (void) download_completed:(NSString *) data {
    NSLog(@"call back");
    [self.indicator stopAnimating];
    self.indicator.hidden = YES;
    self.content.text = data;
    [queue release];
}
```

使用 GCD 后

如果使用 GCD，以上三个方法可以放到一起：

```
// 原代码块一
self.indicator.hidden = NO;
```

```
[self.indicator startAnimating];
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // 原代码块二
    NSURL * url = [NSURL URLWithString:@"http://www.youdao.com"];
    NSError * error;
    NSString * data = [NSString stringWithContentsOfURL:url encoding:
        NSUTF8StringEncoding error:&error];
    if (data != nil) {
        // 原代码块三
        dispatch_async(dispatch_get_main_queue(), ^{
            [self.indicator stopAnimating];
            self.indicator.hidden = YES;
            self.content.text = data;
        });
    } else {
        NSLog(@"error when download:%@", error);
    }
});
```

首先我们可以看到，代码变短了。因为少了原来三个方法的定义，也少了相互之间需要传递的变量的封装。

另外，代码变清楚了，虽然是异步的代码，但是它们被 GCD 合理地整合在一起，逻辑非常清晰。如果应用 MVC 模式，我们也可以将 View Controller 层的回调函数用 GCD 的方式传递给 Modal 层，这相比以前用 @selector 的方式，代码的逻辑关系更加清楚。

11.2 使用 GCD

11.2.1 block 的定义

简单 block 的定义有点像函数指针，差别是用“^”替代了函数指针的“*”符号：

```
// 申明变量
(void) (^loggerBlock)(void);

// 定义
loggerBlock = ^{
    NSLog(@"Hello world");
};

// 调用
loggerBlock();
```

但是大多数情况下，我们使用内联的方式来定义它，即将它的程序块写在调用的函数里面，例如这样：

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
  // something
});
```

从上面大家可以看出，block 有如下特点：

1. 程序块可以在代码中以内联的方式来定义。
2. 程序块可以访问在创建它的范围内的可用的变量。

11.2.2 系统提供的 dispatch 方法

为了方便地使用 GCD，苹果提供了一些方法方便我们将 block 放在主线程或后台线程执行，或者延后执行。使用的例子如下所示：

```
// 后台执行：
dispatch_async(dispatch_get_global_queue(0, 0), ^{
  // something
});
// 主线程执行：
dispatch_async(dispatch_get_main_queue(), ^{
  // something
});
// 一次性执行：
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
 // code to be executed once
});
// 延迟2秒执行：
double delayInSeconds = 2.0;
dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, delayInSeconds *
    NSEC_PER_SEC);
dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
  // code to be executed on the main queue after delay
});
```

dispatch_queue_t 也可以自己定义，如要自定义 queue，可以用 dispatch_queue_create 方法，示例如下：

```
dispatch_queue_t urls_queue = dispatch_queue_create("blog.devtang.com", NULL);
dispatch_async(urls_queue, ^{
    // your code
});
dispatch_release(urls_queue);
```

另外，GCD 还有一些高级用法，例如让后台两个线程并行执行，然后等两个线程都结束后，再汇总执行结果。这个可以用 dispatch_group、dispatch_group_async 和 dispatch_group_notify 来实现，示例如下：

```
dispatch_group_t group = dispatch_group_create();
dispatch_group_async(group, dispatch_get_global_queue(0,0), ^{
    // 并行执行的线程一
});
dispatch_group_async(group, dispatch_get_global_queue(0,0), ^{
    // 并行执行的线程二
});
dispatch_group_notify(group, dispatch_get_global_queue(0,0), ^{
    // 汇总结果
});
```

11.2.3 修改 block 之外的变量

默认情况下，在程序块中访问的外部变量是复制过去的，即写操作不对原变量生效。但是你可以加上 __block 来让其写操作生效，示例代码如下：

```
__block int a = 0;
void  (^foo)(void) = ^{
    a = 1;
}
foo();
// 这里，a的值被修改为1
```

11.2.4 后台运行

使用 block 的另一个用处是可以让程序在后台较长久地运行。在以前，当应用被按 Home 键退出后，应用仅有最多 5 秒钟的时间做一些保存或清理资源的工作。但是应用可以调用 UIApplication 的 beginBackgroundTaskWithExpirationHandler 方法，让应用最多有 10 分钟的时间在后台长久运行。这个时间可以用来做清理本地缓存、发送统计数据等工作。

让程序在后台长久运行的示例代码如下：

```
// AppDelegate.h文件
@property (assign, nonatomic) UIBackgroundTaskIdentifier backgroundUpdateTask;

// AppDelegate.m文件
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [self beingBackgroundUpdateTask];
```

```
    // 在这里加上你需要长久运行的代码
    [self endBackgroundUpdateTask];
}

- (void)beingBackgroundUpdateTask
{
    self.backgroundUpdateTask = [[UIApplication sharedApplication]
        beginBackgroundTaskWithExpirationHandler:^{
        [self endBackgroundUpdateTask];
    }];
}

- (void)endBackgroundUpdateTask
{
    [[UIApplication sharedApplication] endBackgroundTask: self.backgroundUpdateTask
        ];
    self.backgroundUpdateTask = UIBackgroundTaskInvalid;
}
```

11.2.5 总结

总体来说，GCD 能够极大地方便开发者进行多线程编程，大家应该尽量使用 GCD 来处理后台线程和 UI 线程的交互。

第12章 使用 UIWindow

UIWindow 是最顶层的界面容器，本章将讲解它的特点及一些使用技巧。

12.1 UIWindow 简介

在 iOS 应用中，我们使用 UIWindow 和 UIView 来呈现界面。UIWindow 并不包含任何默认的内容，但是它被当作 UIView 的容器，用于放置应用中所有的 UIView。而每一个 UIView 通常都是用来表示具体的某一部分界面，例如一段文字、一张图片等，当然，你也可以用 UIView 来当作其他 UIView 的容器。所以 UIWindow 更多的时候只作为 UIView 的顶层容器存在。

从继承关系上，我们能看到 UIWindow 继承自 UIView，所以 UIWindow 除了具有 UIView 的所有功能外，还增加了一些特有的属性和方法。

```
NS_CLASS_AVAILABLE_IOS(2_0) @interface UIWindow : UIView
```

而我们最常用的方法，就是在程序刚启动时，调用 UIWindow 的 makeKeyAndVisible 方法，使整个程序界面可见，代码如下所示：

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.window.rootViewController = [self rootViewController];
    [self.window makeKeyAndVisible];
    return YES;
}
```

所以 UIWindow 的主要作用有：

1. 作为 UIView 的最顶层容器，包含应用显示所需要的所有的 UIView。

2. 传递触摸消息和键盘事件给 UIView。

12.2 为 UIWindow 增加 UIView

通常我们有两种办法给 UIWindow 增加子 UIView：

1. 通过调用 addSubView 方法。因为 UIWindow 是 UIView 的子类，所以它可以使用 UIView 的 addSubView 方法给自己增加子 UIView，从而承担容器的作用。
2. 通过设置其特有的 rootViewController 属性。通过设置该属性为要添加 view 对应的 UIViewController，UIWindow 将会自动将其 view 添加到当前 window 中，同时负责维护 ViewController 和 view 的生命周期。我们在之前的 application: didFinishLaunching-WithOptions: 示例代码中看到的就是这种办法。

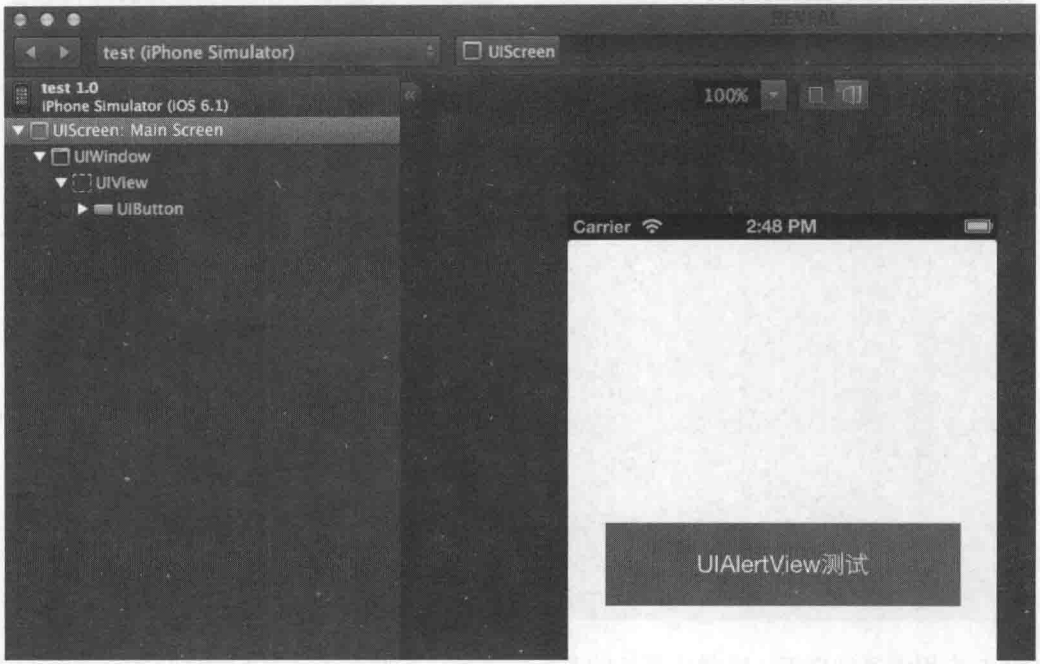
12.3 系统对 UIWindow 的使用

通常在一个程序中只会有一个 UIWindow，但有些时候我们调用系统的控件（例如 UIAlertView）时，iOS 系统为了保证 UIAlertView 在所有的界面之上，它会临时创建一个新的 UIWindow，通过将其 UIWindow 的 UIWindowLevel 设置得更高，让 UIAlertView 盖在所有的应用界面之上。

为了验证这个说法，我们创建一个新的测试 Xcode 工程，在其主 ViewController 上添加一个 UIButton，然后为该 UIButton 增加一个按下逻辑，按下后的行为是弹出一个系统的 UIAlertView:

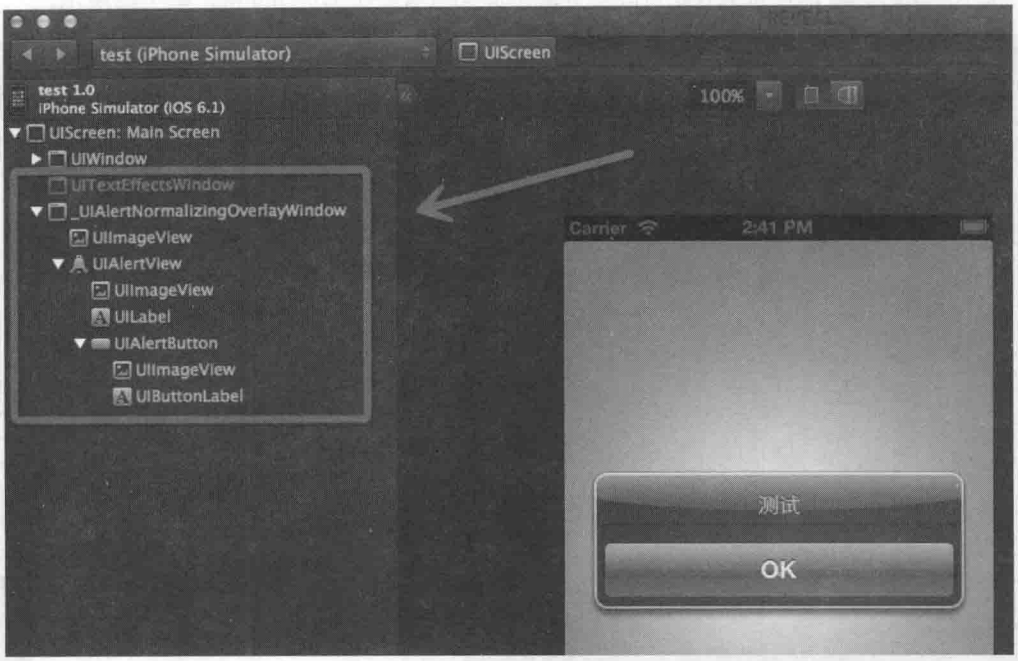
```
- (IBAction)testButtonPressed:(id)sender {
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:nil
                                                        message:@"测试"
                                                       delegate:nil
                                              cancelButtonTitle:@"OK"
                                              otherButtonTitles:nil, nil];
    [alertView show];
}
```

在程序启动时，我们通过 reveal 可以看到整个界面只有一个 UIWindow，如下图所示。



但是当我们单击按钮，触发 UIAlertView 弹出后，我们就可以在 reveal 的界面中看到，系统另外创建了 `UITextEffectsWindow` 和 `UIAlertNormailizingOverlayWindow`，来为 UIAlertView 显示界面。

当我们单击 UIAlertView 上的“OK”按钮使 UIAlertView 消失后，我们又能发现新创建的 UIWindow 消失了，如下图所示。



除了在应用中通过调用 API 弹出系统的各种控件外，有些时候 iOS 系统也会根据当时的各种情况（例如电池电量不足，收到来电或短信等）通过创建新的 UIWindow 来作为系统控件的容器。

12.3.1 WindowLevel

那么是不是新创建的 UIWindow 一定会覆盖在界面的最上面呢？其实并不是这样。UIWindow 有一个类型为“UIWindowLevel”的属性，该属性定义了 UIWindow 的层级，系统定义的 WindowLevel 一共有 3 种取值，如下所示：

```
UIKIT_EXTERN const UIWindowLevel UIWindowLevelNormal;
UIKIT_EXTERN const UIWindowLevel UIWindowLevelAlert;
UIKIT_EXTERN const UIWindowLevel UIWindowLevelStatusBar;
```

我们通过如下代码将这些值输出：

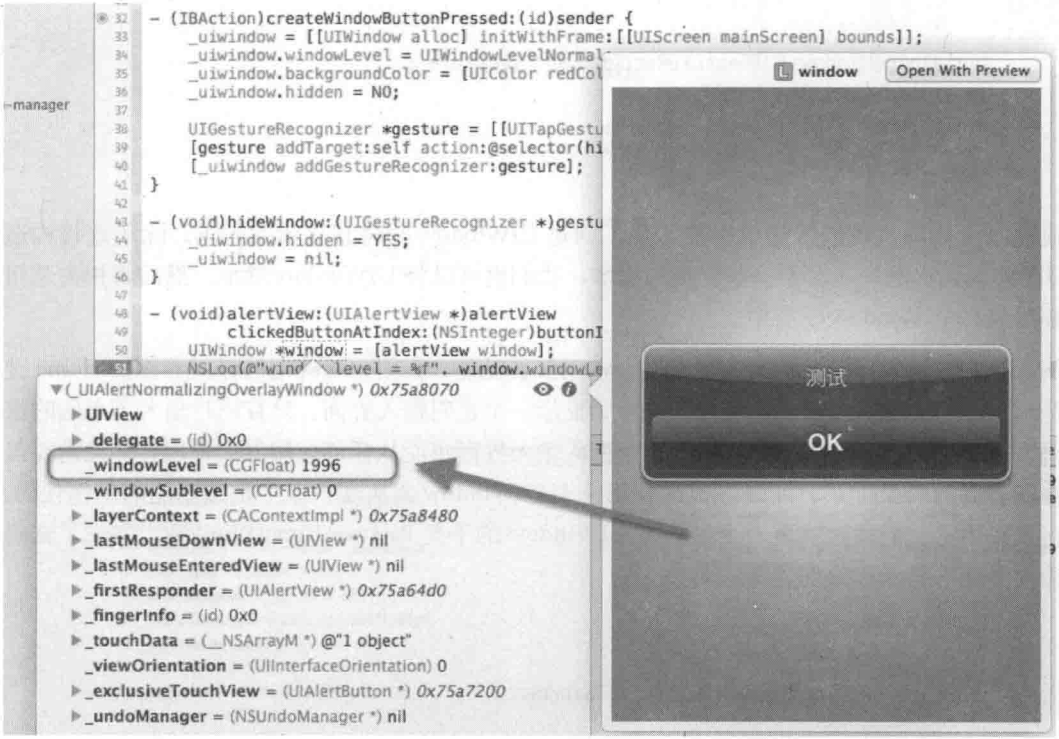
```
NSLog(@"UIWindowLevelNormal=%f UIWindowLevelStatusBar=%f UIWindowLevelAlert=%f",
          UIWindowLevelNormal, UIWindowLevelStatusBar, UIWindowLevelAlert);
```

最终得到的结果是：

```
UIWindowLevelNormal=0.000000 UIWindowLevelStatusBar=1000.000000 UIWindowLevelAlert
    =2000.000000
```

从中我们就能够看出来了，默认程序的 UIWindow 的层级是 UIWindowLevelNormal，当系统需要在其上面覆盖 UIAlertView 时，就会创建一个层级是 UIWindowLevelAlert 的 UIWindow，因为其 WindowLevel 值更高，所以就覆盖在上面了。

在实际应用中，WindowLevel 的取值并不限于上面提到的 3 个值。例如在上面的例子中，我们可以在 UIAlertView 的点击回调中，查看该 view 对应的 UIWindow 的 WindowLevel 值，如下图所示，其 WindowLevel 值为 1996。



12.3.2 手工创建 UIWindow

有些时候，我们也希望在应用开发中，将某些界面覆盖在所有界面的最上层。这个时候，我们就可以手工创建一个新的 UIWindow。需要注意的是，和创建 UIView 不同，UIWindow 一旦被创建，它就自动被添加到整个界面上了。下面是一个示例代码，在代码中，我们创建一个新的 UIWindow，并且把它添加到界面中：

```
@implementation ViewController {
    UIWindow *_uiwindow;
}

- (IBAction)createWindowButtonPressed:(id)sender {
```

```
    _uiwindow = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    _uiwindow.windowLevel = UIWindowLevelNormal;
    _uiwindow.backgroundColor = [UIColor redColor];
    _uiwindow.hidden = NO;

    UIGestureRecognizer *gesture = [[UITapGestureRecognizer alloc] init];
    [gesture addTarget:self action:@selector(hideWindow:)];
    [_uiwindow addGestureRecognizer:gesture];
}

- (void)hideWindow:(UIGestureRecognizer *)gesture {
    _uiwindow.hidden = YES;
    _uiwindow = nil;
}
```

通过以上代码，我们就简单实现了手工创建 UIWindow 并将其显示的需求。除了直接构造 UIWindow 实例外，对于一些复杂的需求，我们也可以将 UIWindow 继承，然后将相关逻辑都封装在 UIWindow 子类中。

例如我们在做有道云笔记时，想做一个密码保护功能，在用户从应用的任何界面按 Home 键退出，过一段时间再从后台切换回来时，显示一个密码输入界面。只有用户输入了正确的密码，才能进入退出前的界面。因为这个密码输入界面可能从任何应用界面弹出，并且需要盖在所有界面的最上层，所以它很适合用一个 UIWindow 来实现，以下是该功能的示例代码。在代码中，我们通过实现一个继承自 UIWindow 的子类 PasswordInputWindow，完成了密码输入界面的显示和处理逻辑：

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

@interface PasswordInputWindow : UIWindow

+ (PasswordInputWindow *)sharedInstance;

- (void)show;

@end

#import "PasswordInputWindow.h"

@implementation PasswordInputWindow {
    UITextField *_textField;
}

+ (PasswordInputWindow *)sharedInstance
{
    static id sharedInstance = nil;
```

```
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] initWithFrame:[[UIScreen mainScreen] bounds
            ]];
    });
    return sharedInstance;
}

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(10, 50, 200, 20)
            ];
        label.text = @"请输入密码";
        [self addSubview:label];

        UITextField *textField = [[UITextField alloc] initWithFrame:CGRectMake(10,
            80, 200, 20)];
        textField.backgroundColor = [UIColor whiteColor];
        textField.secureTextEntry = YES;
        [self addSubview:textField];

        UIButton *button = [[UIButton alloc] initWithFrame:CGRectMake(10, 110, 200,
            44)];
        [button setBackgroundColor:[UIColor blueColor]];
        button.titleLabel.textColor = [UIColor blackColor];
        [button setTitle:@"确定" forState:UIControlStateNormal];
        [button addTarget:self
                  action:@selector(completeButtonPressed:)
         forControlEvents:UIControlEventTouchUpInside];
        [self addSubview:button];

        self.backgroundColor = [UIColor yellowColor];
        _textField = textField;
    }
    return self;
}

- (void)show {
    [self makeKeyWindow];
    self.hidden = NO;
}

- (void)completeButtonPressed:(id)sender {
    if ([_textField.text isEqualToString:@"abcd"]) {
```

```
        [_textField resignFirstResponder];
        [self resignKeyWindow];
        self.hidden = YES;
    } else {
        [self showErrorAlertView];
    }
}

- (void)showErrorAlertView {
    UIAlertView *alertView =
        [[UIAlertView alloc] initWithTitle:nil
                                   message:@"密码错误，正确密码是abcd"
                                  delegate:nil
                         cancelButtonTitle:@"OK"
                         otherButtonTitles:nil, nil];
    [alertView show];
}

@end
```

我们只需要在应用进入后台的回调函数中，将该 UIWindow 创建显示出来即可，代码如下所示：

```
- (void)applicationDidEnterBackground:(UIApplication *)application {
    [[PasswordInputWindow sharedInstance] show];
}
```

最后还有一点需要注意的是，如果我们创建的 UIWindow 需要处理键盘事件，那就需要合理地将其设置为 keyWindow。keyWindow 是被系统设计用来接收键盘和其他非触摸事件的 UIWindow。我们可以通过 `makeKeyWindow` 和 `resignKeyWindow` 方法来将自己创建的 UIWindow 实例设置成 keyWindow。

支付宝客户端的手势解锁功能，也是 UIWindow 的一个极好的应用。除了密码输入界面外，其他适合用 UIWindow 来实现的功能还包括：应用的启动介绍页、应用内的通知提醒显示、应用内的弹框广告等。

12.3.3 不要滥用 UIWindow

通过创建 UIWindow，我们很容易地实现了将某个特定界面置于最上层的效果，但是这种特性不应该被滥用。很多时候，如果弹出界面明显属于某一个 ViewController，那么更适合把弹出的界面当作这个 ViewController 的 view 的 subView 来实现。

常见的滥用方式是把需要的弹出界面都设置成单例，需要的时候就调用显示。这种做法会使得新创建的 UIWindow 一直得不到释放。并且当出现多个 UIWindow 需要相互有层级覆

盖关系时，实现起来会比较复杂。

12.3.4 参考资料

- UIScreen Class Reference: http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIScreen_Class/Reference/UIScreen.html
- UIView Class Reference: http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIView_Class/UIView/UIView.html
- UIWindow Class Reference：http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIWindow_Class/UIWindowClassReference/UIWindowClassReference.html
- UIViewController Class Reference：http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIViewController_Class/Reference/Reference.html

第13章 动态下载系统提供的多种中文字体

从 iOS6 开始，苹果开始支持动态下载官方提供的中文字体到系统中。使用苹果官方提供的中文字体，既可以避免版权问题，又可以节省应用体积。该方案适合对字体有较多需求的应用。本章详细介绍了该功能的使用方法。

13.1 功能简介

13.1.1 前言

为了实现更好的字体效果，有些应用在自己的应用资源包中加入了字体文件。但自己打包字体文件比较麻烦，原因在于：

1. 字体文件通常比较大，10~20MB 是常见的字体库的大小。大部分的非游戏的应用体积都集中在 10MB 以内，因为字体文件的加入而造成应用体积翻倍让人感觉有些不值。如果只是很少量的按钮字体需要设置，可以用一些工具把使用到的汉字字体编码从字体库中抽取出来，以节省体积。但如果是一些变化的内容需要自定义的字体，那就只能打包整个字体库了。
2. 中文字体通常都是有版权的，在应用中加入特殊中文字体还需要处理相应的版权问题。对于一些小公司或个人开发者来说，这是一笔不小的开销。

以上两点造成 App Store 里面使用特殊中文字体库的 iOS 应用较少。现在通常只有阅读类的应用才会使用特殊中文字体库。

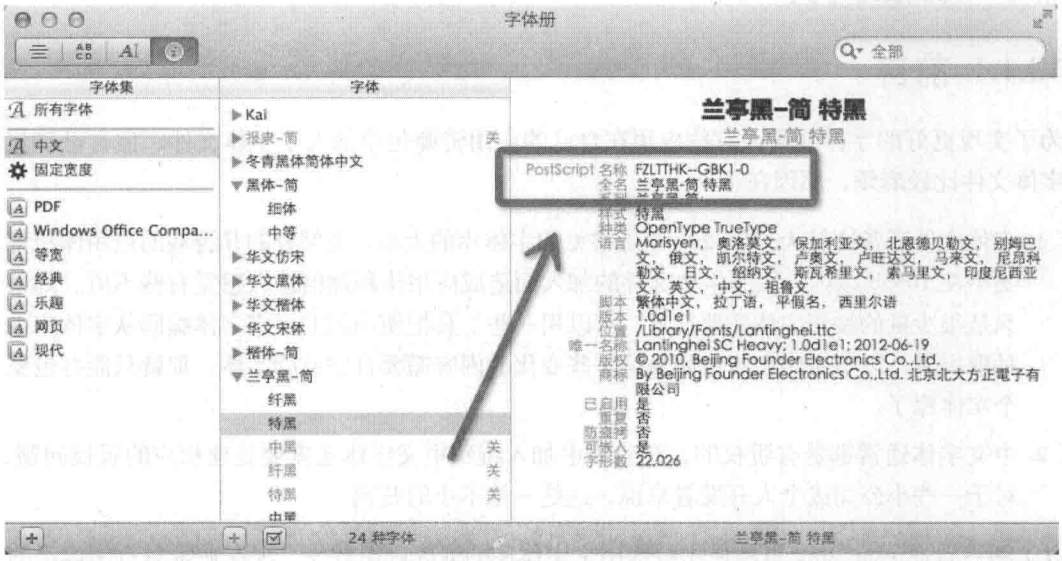
从 iOS6 开始，苹果开始支持动态下载中文字体到系统中，接下来就让我们来一起了解这个功能。

13.1.2 功能介绍

使用动态下载中文字体的 API 可以动态地向 iOS 系统中添加字体文件，这些字体文件都是下载到系统的目录中（目录是/private/var/mobile/Library/Assets/com_apple_MobileAsset_Font/），所以并不会造成应用体积的增加。并且，由于字体文件是 iOS 系统提供的，也免去了字体使用版权的问题。虽然第一次下载相关的中文字体需要一些网络开销和下载时间，但是这些字体文件下载后可以在所有应用间共享，所以可以预见，随着该 API 使用的普及，大部分应用都不需要提示用户下载字体，因为很可能这些字体在之前就被其他应用下载下来了。

13.1.3 字体列表

在官方文档（http://support.apple.com/kb/HT5484?viewlocale=zh_CN）中，苹果列出了提供动态下载和使用的中文字体文件列表。不过，由于下载的时候需要使用的名字是 PostScript 名称，所以如果你真正要动态下载相应的字体的话，还需要使用 Mac 内自带的应用“字体册”（Font Book）来获得相应字体的 PostScript 名称。下面是从字体册中获取“兰亭黑-简 特黑”字体的 PostScript 名称的截图。



13.2 使用教程

13.2.1 相关 API 介绍

苹果提供了动态下载代码的 Demo 工程（http://developer.apple.com/library/ios/#samplecode/DownloadFont/Listings/DownloadFont_ViewController_m.html）。将此 Demo 工程下载下来，可以学习相关 API 的使用。下面对该工程中的相关 API 做简单的介绍。

假如我们现在要下载“娃娃体”，它的 PostScript 名称为“DFWaWaSC-W5”。具体操作步骤如下。

1. 我们先判断该字体是否已经被下载下来，代码如下所示：

```
- (BOOL)isFontDownloaded:(NSString *)fontName {
    UIFont* aFont = [UIFont fontWithName:fontName size:12.0];
    if (aFont && ([aFont.fontName compare:fontName] == NSOrderedSame
              || [aFont.familyName compare:fontName] == NSOrderedSame)) {
        return YES;
    } else {
        return NO;
    }
}
```

2. 如果该字体已经下载过了，则可以直接使用。否则我们需要先准备下载字体 API 需要的一些参数，代码如下所示：

```
// 用字体的PostScript名字创建一个Dictionary
NSMutableDictionary *attrs = [NSMutableDictionary dictionaryWithObjectsAndKeys:
    fontName, kCTFontNameAttribute, nil];

// 创建一个字体描述对象CTFontDescriptorRef
CTFontDescriptorRef desc = CTFontDescriptorCreateWithAttributes((__bridge
    CFDictionaryRef)attrs);

// 将字体描述对象放到一个NSMutableArray中
NSMutableArray *descs = [NSMutableArray arrayWithCapacity:0];
[descs addObject:(__bridge id)desc];
CFRelease(desc);
```

3. 准备好上面的 descs 变量后，就可以进行字体的下载了，代码如下所示：

```
__block BOOL errorDuringDownload = NO;

CTFontDescriptorMatchFontDescriptorsWithProgressHandler( (__bridge CFArrayRef)
    descs, NULL,  ^(CTFontDescriptorMatchingState state, CFDictionaryRef
    progressParameter) {
```

```
        double progressValue = [[(__bridge NSDictionary *)progressParameter
            objectForKey:(id)kCTFontDescriptorMatchingPercentage] doubleValue];

        if (state == kCTFontDescriptorMatchingDidBegin) {
            NSLog(@"字体已经匹配");
        } else if (state == kCTFontDescriptorMatchingDidFinish) {
            if (!errorDuringDownload) {
                NSLog(@"字体%@ 下载完成", fontName);
            }
        } else if (state == kCTFontDescriptorMatchingWillBeginDownloading) {
            NSLog(@"字体开始下载");
        } else if (state == kCTFontDescriptorMatchingDidFinishDownloading) {
            NSLog(@"字体下载完成");
            dispatch_async( dispatch_get_main_queue(), ^ {
                // 可以在这里修改UI控件的字体
            });
        } else if (state == kCTFontDescriptorMatchingDownloading) {
            NSLog(@"下载进度 %.0f%% ", progressValue);
        } else if (state == kCTFontDescriptorMatchingDidFailWithError) {
            NSError *error = [(__bridge NSDictionary *)progressParameter
                objectForKey:(id)kCTFontDescriptorMatchingError];
            if (error != nil) {
                _errorMessage = [error description];
            } else {
                _errorMessage = @"ERROR MESSAGE IS NOT AVAILABLE!";
            }
            // 设置标志
            errorDuringDownload = YES;
            NSLog(@"下载错误: %@", _errorMessage);
        }

        return (BOOL)YES;
    });
```

通常需要在下载完字体后开始使用字体，一般是将相应代码放到 kCTFontDescriptorMatchingDidFinish 那个条件中，可以像苹果官方网站的示例代码那样，用 GCD 来修改 UI 的逻辑，也可以发 Notification 来通知相应的 Controller。

下面是通过以上示例代码下载的娃娃体字体截图。

13.2.2 总结

使用系统提供的中文字体，既可以避免版权问题，又可以减小应用体积。该方案适合对字体有较多需求的应用。

第14章 使用应用内支付

本章讲解开发应用内支付的基本步骤及注意事项。

14.1 后台设置

配置 Developer.apple.com

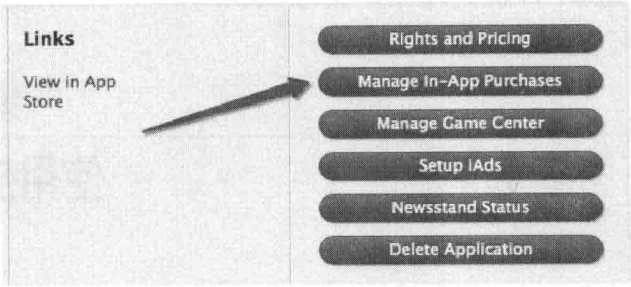
登录 Developer.apple.com（https://developer.apple.com/），然后进行以下操作：

1. 为应用建立一个不带通配符的 App ID。
2. 用该 App ID 生成和安装相应的 Provisioning Profile 文件。

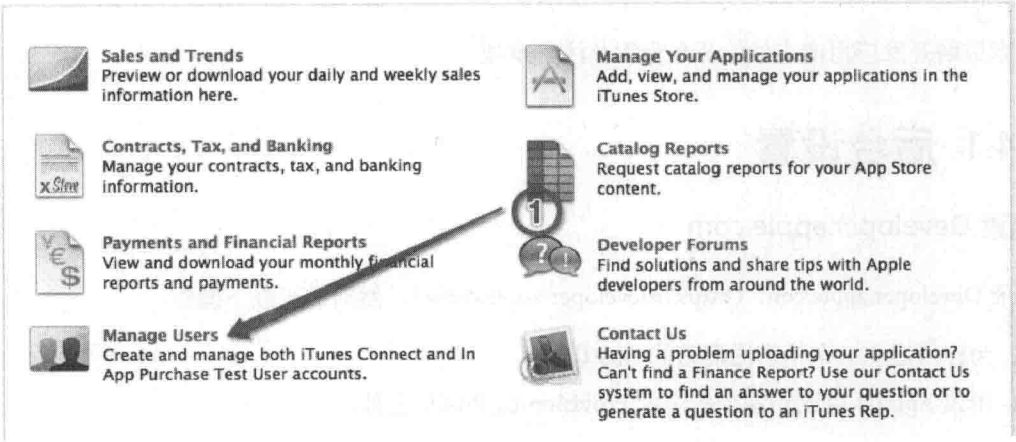
配置 iTunes Connect

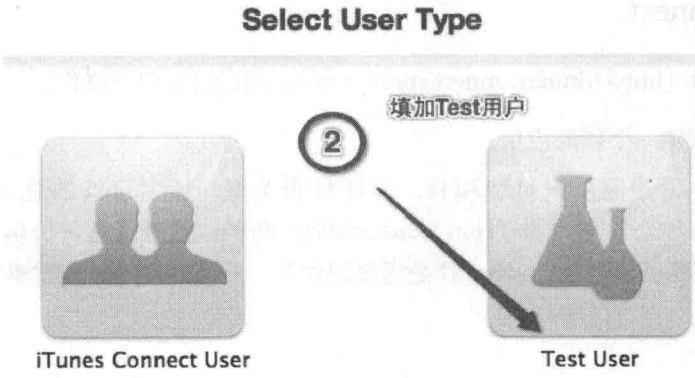
登录 iTunes Connet（https://itunesconnect.apple.com/），然后进行以下操作：

1. 用该 App ID 创建一个新的应用。
2. 在该应用中，创建应用内付费项目，选择付费类型，通常可选的是可重复消费的（Consumable）和永久有效的（Non-Consumable）两种，然后设置好价格、Product ID、购买介绍和截图，这里的 Product ID 是需要记住的，后面开发的时候需要用到，如下图所示。

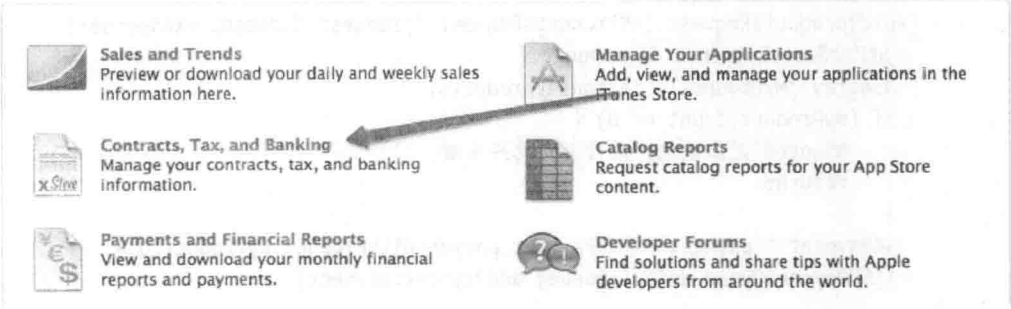


3. 添加一个用于在 sandbox 付费的测试用户，如下图所示。注意苹果对该测试用户的密码要求和正式账号一样，至少要 8 位，并且要同时包含数字和大小写字母。





4. 填写相关的税务、银行和联系人信息，如下图所示。



14.2 iOS 端开发

1. 在工程中引入 storekit.framework 和 #import (StoreKit/StoreKit.h)。
2. 获得所有的付费 Product ID 列表。这个可以用常量存储在本地，也可以由自己的服务器返回。
3. 制作一个界面，展示所有的应用内付费项目。这些应用内付费项目的价格和介绍信息可以是自己的服务器返回。但如果是不带服务器的单机游戏应用或工具类应用，则可以通过向 App Store 查询获得。我在测试时发现，向 App Store 查询的速度非常慢，通常需要 2~3 秒钟，所以不建议这么做，最好还是搞个自己的服务器。
4. 当用户点击了一个 IAP 项目，我们需要先查询用户是否允许应用内付费，如果不允许则不用进行以下步骤了，代码如下所示：

```
if ([SKPaymentQueue canMakePayments]) {
    // 执行下面提到的第5步：
    [self getProductInfo];
} else {
    NSLog(@"失败，用户禁止应用内付费购买。");
}
```

5. 我们先通过该 IAP 的 Product ID 向 App Store 查询，获得 SKPayment 实例，然后通过 SKPaymentQueue 的 addPayment 方法发起一个购买的操作，代码如下所示：

```
// 下面的ProductId应该是事先在itunesConnect中添加好的，已存在的付费项目。否则查
    询会失败。
- (void)getProductInfo {
        NSSet * set = [NSSet setWithArray:@[@"ProductId"]];
        SKProductsRequest * request = [[SKProductsRequest alloc]
            initWithProductIdentifiers:set];
        request.delegate = self;
        [request start];
}
```

```
// 以上查询的回调函数
- (void)productsRequest:(SKProductsRequest *)request didReceiveResponse:(
    SKProductsResponse *)response {
    NSArray *myProduct = response.products;
    if (myProduct.count == 0) {
        NSLog(@"无法获取产品信息，购买失败。");
        return;
    }
    SKPayment * payment = [SKPayment paymentWithProduct:myProduct[0]];
    [[SKPaymentQueue defaultQueue] addPayment:payment];
}
```

6. 在 viewDidLoad 方法中，将购买页面设置成购买的 Observer，代码如下所示：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 监听购买结果
    [[SKPaymentQueue defaultQueue] addTransactionObserver:self];
}

- (void)viewDidUnload {
    [super viewDidUnload];
    [[SKPaymentQueue defaultQueue] removeTransactionObserver:self];
}
```

7. 当用户购买的操作有结果时，就会触发下面的回调函数，相应进行处理即可：

```
- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray *)
    transactions {
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased://交易完成
                NSLog(@"transactionIdentifier = %@", transaction.
                    transactionIdentifier);
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed://交易失败
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored://已经购买过该商品
                [self restoreTransaction:transaction];
                break;
            case SKPaymentTransactionStatePurchasing:      //商品添加进列表
                NSLog(@"商品添加进列表");
                break;
            default:
```

```
                break;
            }
        }

    }

    - (void)completeTransaction:(SKPaymentTransaction *)transaction {
        // Your application should implement these two methods.
        NSString * productIdentifier = transaction.payment.productIdentifier;
        NSString * receipt = [transaction.transactionReceipt base64EncodedString];
        if ([productIdentifier length] > 0) {
            // 向自己的服务器验证购买凭证
        }

        // Remove the transaction from the payment queue.
        [[SKPaymentQueue defaultQueue] finishTransaction: transaction];

    }

    - (void)failedTransaction:(SKPaymentTransaction *)transaction {
        if(transaction.error.code != SKErrorPaymentCancelled) {
            NSLog(@"购买失败");
        } else {
            NSLog(@"用户取消交易");
        }
        [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
    }

    - (void)restoreTransaction:(SKPaymentTransaction *)transaction {
            // 对于已购商品，处理恢复购买的逻辑
        [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
    }
```

8. 服务器验证凭证（可选项）。如果购买成功，我们需要将凭证发送到服务器上进行验证。考虑到网络异常情况，iOS 端的发送凭证操作应该可以持久化，如果程序退出、崩溃或网络异常，可以恢复重试。

14.3 服务端开发

服务端后台的工作比较简单，分为 4 步：

1. 接收 iOS 端发过来的购买凭证。
2. 判断凭证是否已经存在，是否验证过，然后存储该凭证。
3. 将该凭证发送到苹果的服务器验证，并将验证结果返回给客户端。

4. 如果需要，修改用户相应的会员权限。

考虑到网络异常情况，服务器的验证应该是一个可恢复的队列，如果失败了，应该进行重试。

与苹果的验证接口文档在https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/VerifyingStoreReceipts/VerifyingStoreReceipts.html#//apple_ref/doc/uid/TP40008267-CH104-SW3。简单来说就是将该购买凭证用 Base64 编码，然后 POST 给苹果的验证服务器，苹果将验证结果以 JSON 形式返回。

苹果 App Store 线上的购买凭证验证地址是https://buy.itunes.apple.com/verifyReceipt，测试的验证地址是https://sandbox.itunes.apple.com/verifyReceipt。

14.4 注意事项

大家提交给苹果审核的应用肯定是正式版，但如果你以为苹果审核时你的应用应该连接苹果的线上验证服务器来验证购买凭证，那就错了。

苹果在审核应用时，只会在沙盒（sandbox）环境购买，其产生的购买凭证，也只能连接苹果的测试验证服务器。但是审核的应用又是连接的我们的线上服务器，那应该怎么处理呢？

解决方法是判断苹果正式验证服务器的返回状态码，如果是 21007，则再一次连接测试服务器进行验证即可。苹果的文档 http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/RenewableSubscriptions/RenewableSubscriptions.html上有对返回的状态码的详细说明。

第15章 基于UIWebView的混合编程

基于UIWebView的混合编程是指同时使用原生的控件和UIWebView来展现应用界面。合理地使用该方案可以保证应用既有原有界面的流畅交互效果，又有Web界面的良好的动态修改和多平台复用的优势。

15.1 混合编程简介

基于UIWebView的混合编程本来是一个挺普通和常见的技术框架，但是自从国外开始用Hybird来称呼它时，这个技术突然间就变得“高大上”起来。[1]

国内的许多应用都采用了基于UIWebView的混合编程技术，例如微信公众号的内容页面、微博的详情页面、网易新闻客户端的内容页面、支付宝客户端的彩票页面等。下图是支付宝客户端的彩票页面（左半边是手机上的应用截图，右半边是浏览器打开该HTML文件的截图），从中可以看到其界面就是用UIWebView实现的。

[1] 这类事情的发生并不鲜见，在几年前的前端开发领域中，用JavaScript异步请求XML数据来渲染HTML界面的技术方案，也因为有了AJAX的名字而引起巨大反响。

我所参与开发的有道云笔记的笔记查看和编辑界面，也是基于 UIWebView 实现的。这些页面都具有以下共同的特点：

1. 排版复杂。通常包括图片和文字的混排，还有可能有链接需要支持点击。如果不用 UIWebView，自己用原生控件通过拼装来实现，由于界面元素太多，做起来会很困难，而如果用 CoreText 来实现，就需要自己实现相当多的复杂排版逻辑。
2. 界面的变化需求频繁。例如淘宝的彩票页面，可能常常需要更新界面以推出不同的活动。采用 UIWebView 实现后，这类页面就可以动态地更新而不用向 App Store 提交新的版本，而原生实现的界面很难达到如此的灵活性。
3. 界面对用户的交互需求不复杂。因为 UIWebView 实现的交互效果与原生效果相比还是会大打折扣，所以这类界面通常都没有复杂的交互效果。这也是主流应用大多采用混合 UIWebView 来实现应用界面而不是使用纯 UIWebView 来实现界面的原因。

如果你的应用界面也具有以上属性，那么你也可以考虑使用 UIWebView 来实现该界面。下面我们来看看使用该技术方案需要考虑哪些问题。

15.2 使用模板引擎渲染 HTML 界面

在实际开发中，UIWebView 控件接受一个 HTML 内容，用于呈现相应的界面，下面是该 API 的接口：

```
- (void)loadHTMLString:(NSString *)string baseURL:(NSURL *)baseURL;
```

由于 HTML 内容通常是变化的，所以我们需要在内存中生成该 HTML 内容。比较简单粗暴的做法是将该 HTML 的基本内容定义在一个 NSString 中，然后用 [NSString stringWithFormat] 方法将内容进行格式化，下面是一个示例：

```
- (NSString *)demoFormatWithName:(NSString *)name value:(NSString *)value {
    NSString *html =
    @"<HTML>"
    "<HEAD>"
    "</HEAD>"
    "<BODY>"
    "<H1>%@</H1>"
    "<P>%@</P>"
    "</BODY>"
    "</HTML>";

    NSString *content = [NSString stringWithFormat:html, name, value];
    return content;
}
```

但其实我们可以看出，这样写并不舒服，因为：

1. 模板内容和代码混在一起，既不方便阅读，也不方便更改。
2. 模板的渲染逻辑使用简单的 [NSString stringWithFormat] 来完成，功能单一。在实际开发中，我们很可能需要将原始数据进行二次处理，而这些如果模板渲染模块不能支持，我们就只能自己手工写这部分数据二次处理，费时费力。例如：微博的详情页面，如果微博的发送时间小于 1 天，则需要显示成“×× 小时前”，如果小于 1 分钟，则需要显示成“刚刚”。这些界面渲染方面的逻辑如果能够抽取到专门的排版代码中，则会清晰很多。

所以我们需要一个模板引擎，专门负责这类渲染的工作。

我个人使用过的模板引擎是 MGTemplateEngine（http://mattgemmell.com/mgtemplateengine-templates-with-cocoa/ ），它的模板语言比较像 Smarty、FreeMarker 和 Django。另外它可以自定义 Filter，以便实现上面提到的自定义渲染逻辑。它需要依赖 RegexKit，RegexKit 是一个正则表达式工具类，提供强大的正则表达式匹配和替换功能。

不喜欢模板引擎功能太过于复杂的朋友，也可以尝试 GRMustache（https://github.com/groue/GRMustache），它比 MGTemplateEngine、GRMustache 的功能更简单。另外 GRMustache 在开源社区更加活跃，更新更频繁。

对于上面的示例代码，在使用 GRMustache 模板后，我们首先需要调整模板的内容：

1. 将模板内容放在另一个单独的文件中，方便日后更改。
2. 将原来的 %@, 替换成 {{ name }} 的形式。

模板调整后变成了如下内容（文件名为 template.html）：

```
<HTML>
  <HEAD>
  </HEAD>
  <BODY>
    <H1> {{ name }} </H1>
    <P> {{ content }}</P>
  </BODY>
</HTML>
```

然后我们在代码中将该文件读取到内存中，再使用 GRMustache 的 renderObject 方法生成渲染后的 HTML 内容，示例代码如下：

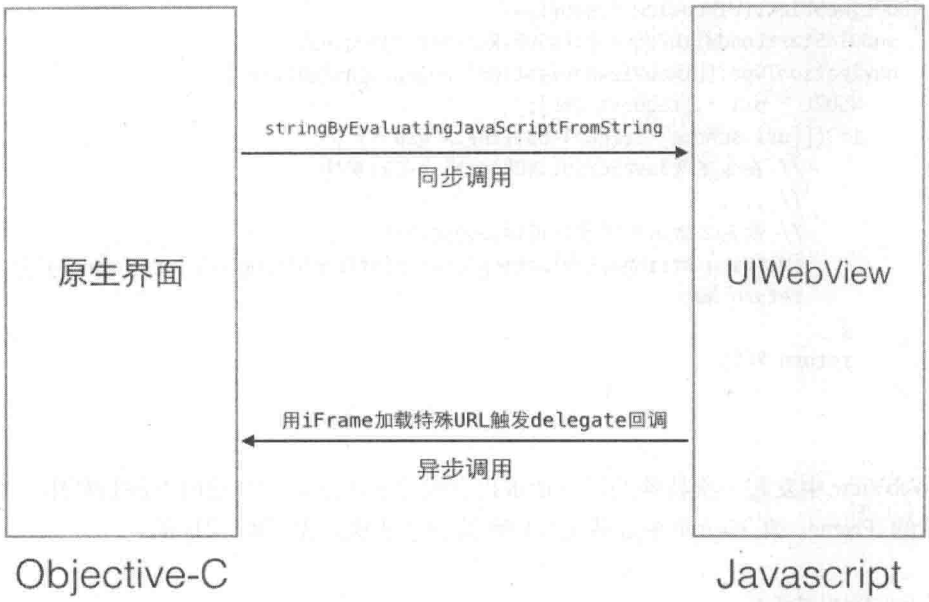
```
- (NSString *)demoFormatWithName:(NSString *)name value:(NSString *)value {
    NSString *fileName = @"template.html";
    NSString *path = [[[NSBundle mainBundle] bundlePath]
        stringByAppendingPathComponent:fileName];
    NSString *template = [NSString stringWithContentsOfFile:path encoding:
        NSUTF8StringEncoding error:nil];
    NSDictionary *renderObject = @{ @"name": name , @"conent": value };
    NSString *content = [GRMustacheTemplate renderObject:renderObject fromString:
        template error:nil];
    return content;
}
```

这样，我们使用 GRMustache 模板引擎成功完成了 HTML 内容渲染工作，之后就可以通过如下代码来让 UIWebView 加载 HTML 的内容了：

```
NSString * path = [[NSBundle mainBundle] bundlePath];
NSURL * baseURL = [NSURL fileURLWithPath:path];
// 通过模板渲染得到内容
NSString * htmlString = [self htmlContent];
[self.webView loadHTMLString:htmlString baseURL:baseURL];
```

15.3 Objective-C 语言和 JavaScript 语言相互调用

原生界面与 UIWebView 相互调用示意图如下所示。



在 UIWebView 通过一次性加载 HTML 内容获得初始界面后，对于复杂的应用，我们还需要在原生界面和 UIWebView 界面相互调用传递数据。但是，iOS 的 UIWebView 控件在与原生界面交互数据这方面功能较弱，所以我们需要详细看一下原生界面和 UIWebView 界面是如何做到相互调用的。因为原生界面是用 Objective-C 语言写的，而 UIWebView 界面是用 JavaScript 写的，所以我们讨论的主要就是如何做 Objective-C 语言和 JavaScript 语言之间的跨语言相互调用。

Objective-C 语言调用 JavaScript 语言，是通过 UIWebView 的`- (NSString *)stringByEvaluatingJavaScriptFromString:(NSString *)script;`方法实现的。该方法向 UIWebView 传递一段需要执行的 JavaScript 文件，最后获得执行结果。

JavaScript 语言调用 Objective-C 语言，并没有现成的 API，但是业界有一种“曲线救国”的方法，间接达到了调用的效果。该方法是利用 UIWebView 的特性：在 UIWebView 内发起的所有网络请求，都可以通过 delegate 函数在原生界面得到通知。这样我们在 UIWebView 内发起一个特殊的网络请求，请求加载的网址内容通常不是真实的地址，地址常常类似这样：`gap://methodname?argument`。

于是在 UIWebView 的 delegate 函数中，我们只要发现是`gap://`开头的地址，就不进行内容的加载，转而执行相应的调用逻辑。这也是著名的 Cordova （PhoneGap 的核心代码，贡献给了 Apache 基金会）框架调用原生逻辑的机制。以下是原生端截获特殊网络请求进行处理的示例代码：

```
// Objective-C语言
- (BOOL)webView:(UIWebView *)webView
    shouldStartLoadWithRequest:(NSURLRequest *)request
    navigationType:(UIWebViewNavigationType)navigationType {
       NSURL * url = [request URL];
       if ([[url scheme] isEqualToString:@"gap"]) {
            // 在这里做JavaScript调Objective-C的事情
            // ....
            // 做完之后用如下方法调回JavaScript
            [webView stringByEvaluatingJavaScriptFromString:@"alert('done')"];
            return NO;
        }
        return YES;
}
```

在 UIWebView 中发起一次特殊的网络请求也有很多种办法，最合适的办法是创建一个临时的隐藏的 iFrame，在 iFrame 中加载这个特殊的网络请求。代码如下所示：

```
// JavaScript语言
// 通知iPhone UIWebView 加载url对应的资源
// url的格式为: gap://something
function loadURL(url) {
    var iFrame;
    iFrame = document.createElement("iFrame");
    iFrame.setAttribute("src", url);
    iFrame.setAttribute("style", "display:none;");
    iFrame.setAttribute("height", "0px");
    iFrame.setAttribute("width", "0px");
    iFrame.setAttribute("frameborder", "0");
    document.body.appendChild(iFrame);
    // 发起请求后这个iFrame就没用了，所以把它从dom上移除掉
    iFrame.parentNode.removeChild(iFrame);
    iFrame = null;
}
```

需要特别注意的是，通过修改 document.location 也可以达到发起网络请求的效果。但是经过我们试验，修改 document.location 有一个很严重的问题，就是如果我们连续两次修改 document.location 的话，在原生界面的 delegate 方法中，只能截获后面那次请求，前一次请求由于很快被替换掉，所以被忽略掉了。所以该方法无法稳定地多次向原生界面发起调用。

如果你不想自己实现具体的相互调用，你也可以使用开源的 WebViewJavaScriptBridge (https://github.com/marcuswestin/WebViewJavaScriptBridge)，它能帮你实现上面我们提到的

相互调用功能。[2]

15.4 如何传递参数

以上的示例代码为了讲清楚机制，所以只是示例了最简单的相互调用。但实际上 Objective-C 语言和 JavaScript 语言相互调用时，常常需要传递参数。

例如，有道云笔记 iPad 版用 UIWebView 显示笔记的内容，当用户点击了笔记中的附件时，JavaScript 需要通知应用到后台下载这个笔记附件，同时通知当前的下载进度。对于这个需求，JavaScript 层获得用户点击事件后，就需要把当前点击的附件的 ID 传递给原生界面，这样原生界面才能知道下载哪个附件。

参数传递最简单的方式是将参数作为 URL 的一部分，放到 iFrame 的 src 里面。这样 UIWebView 通过截取分析 URL 后面的内容即可获得参数。但是这样的问题是，该方法只能传递简单的参数信息，如果参数是一个很复杂的对象，那么这个 URL 的编码将会很复杂。对此，有道云笔记和 Cordova 采用了不同的技术方案。

- 我们的技术方案是将参数以 JSON 的形式传递，但是因为要附加在 URL 之后，所以我们将 JSON 进行了 Base64 编码，以保证 URL 中不会出现一些非法的字符。
- Cordova 的技术方案，也是用 JSON 传递参数，但是将 JSON 放在 UIWebView 中的一个全局数组中，当 UIWebView 需要读取参数时，通过读取这个全局数组来获得相应的参数。

相比之下，应该说 Cordova 的方案更加全面，适用于多种场景。而我们的方案简洁高效，满足了我们自己产品的需求。读者可以根据具体的需求，自行决定自己的参数传递方案。

15.5 同步和异步

因为 iOS SDK 并非“天生”就支持 UIWebView 和原生界面相互调用，所以这里面还有同步异步的问题。细心的读者就能发现，UIWebView 调用原生界面是通过插入一个 iFrame，这个 iFrame 插入后就完了，执行的结果需要原生界面另外用 `stringByEvaluatingJavaScriptFromString` 方法通知 UIWebView，所以这是一个异步的调用。

而 `stringByEvaluatingJavaScriptFromString` 方法本身会直接返回一个 NSString 类型的执行结果，所以这显然是一个同步调用。

所以 UIWebView 调用原生界面是异步，原生界面调用 UIWebView 是同步。在处理一些逻辑的时候，不可避免地需要考虑这个特点。

[2] Objective-C 与 JavaScript 语言的相互操作，如果抛开了 UIWebView，我们还可以使用 iOS7 新增的关于 JavaScriptCore 相关的 API。它可以用来获得一个 JavaScript 语言的执行环境，但由于其执行环境含有 HTML 文档对象模型（DOM），所以它仅可以用来实现纯业务逻辑而不能操作 HTML DOM。

15.6 注意事项

15.6.1 线程阻塞问题

我们在开发中发现，当在 Objective-C 语言中调用 stringByEvaluatingJavaScriptFromString 方法时，可能由于 JavaScript 是单线程的原因，会阻塞原有 JavaScript 代码的执行。这里我们的解决办法是在 JavaScript 端用 defer 将 iFrame 的插入延后执行。

15.6.2 主线程的问题

UIWebView 的 stringByEvaluatingJavaScriptFromString 方法必须在主线程中执行，而主线程的执行时间过长就会阻塞 UI 的更新。所以我们应该尽量让 stringByEvaluatingJavaScriptFromString 方法执行时间短。

有道云笔记在保存的时候，需要调用 JavaScript 代码获得笔记的完整 HTML 内容，这个时候如果笔记内容很复杂，就会执行很长一段时间，而因为这个操作必须是主线程执行，所以我们要显示“正在保存”的 UIAlertView 完全无法正常显示，整个 UI 界面就完全卡住了。在新的编辑器里，我们更新了获得 HTML 内容的代码，才将这个问题解决。

15.6.3 键盘控制

做 iOS 开发的都知道，当我们需要键盘显示在某个控件上时，可以调用 [obj becomeFirstResponder] 方法来让键盘出来，并且光标输入焦点出现在该控件上。

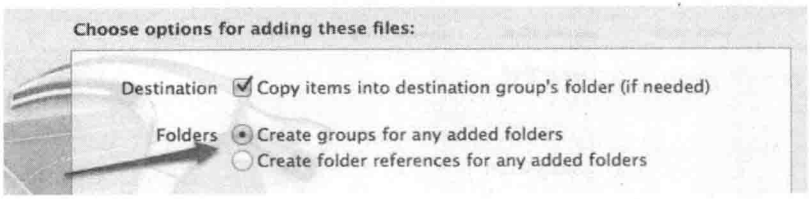
但是这个方法对于 UIWebView 并不适用。也就是说，我们无法通过程序控制让光标输入焦点出现在 UIWebView 上。关于这个问题，我在 Stack Overflow 上专门问了一下，还是没有得到很好的解决办法。

15.6.4 CommonJS 规范

CommonJS 是一个模块加载的规范，而 AMD 是该规范的一个草案。CommonJS AMD 规范描述了模块化的定义、依赖关系、引用关系及加载机制，其规范原文在这里：http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition 。它被 requireJS、NodeJs、Dojo、jQuery 等开源框架广泛使用。

AMD 规范需要用目录层级当作包层次，这一点就象 Java 一样。所以我们需要 iOS 打包后的 ipa 资源文件中具有资源目录层级关系。具体做法非常简单：只需要将该目录拖入工程中，

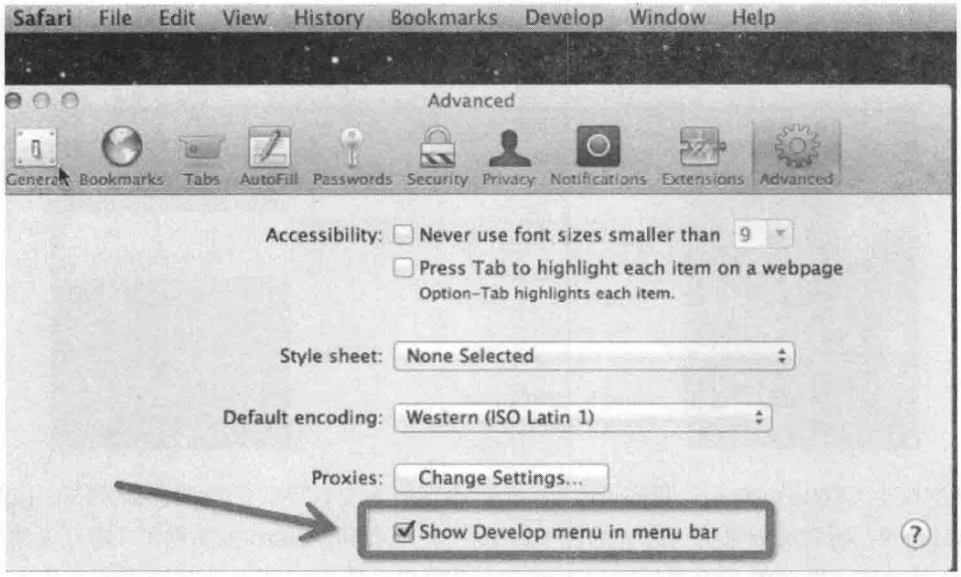
然后选择“Create groups for any added folders”。如下图所示，这样目录层级就能够打包到 ipa 文件中。



15.7 使用 Safari 进行调试

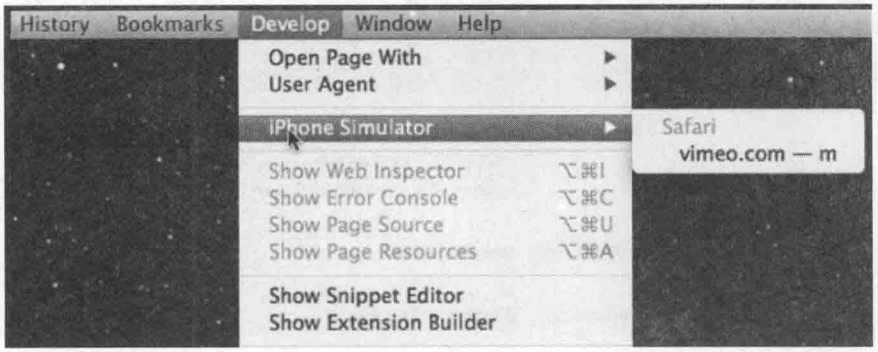
由于 UIWebView 的使用实在广泛，所以从 2012 年开始，苹果支持用 Safari 浏览器直接连接到模拟器或真机中的 UIWebView 来进行相关 HTML 页面，以及 JavaScript 逻辑的调试。

在使用之前，首先需要打开 Safari 的调试模式，在 Safari 的菜单中，选择“Safari”→“Preferences”→“Advanced”，勾选上“Show Develop menu in menu bar”选项，如下图所示。

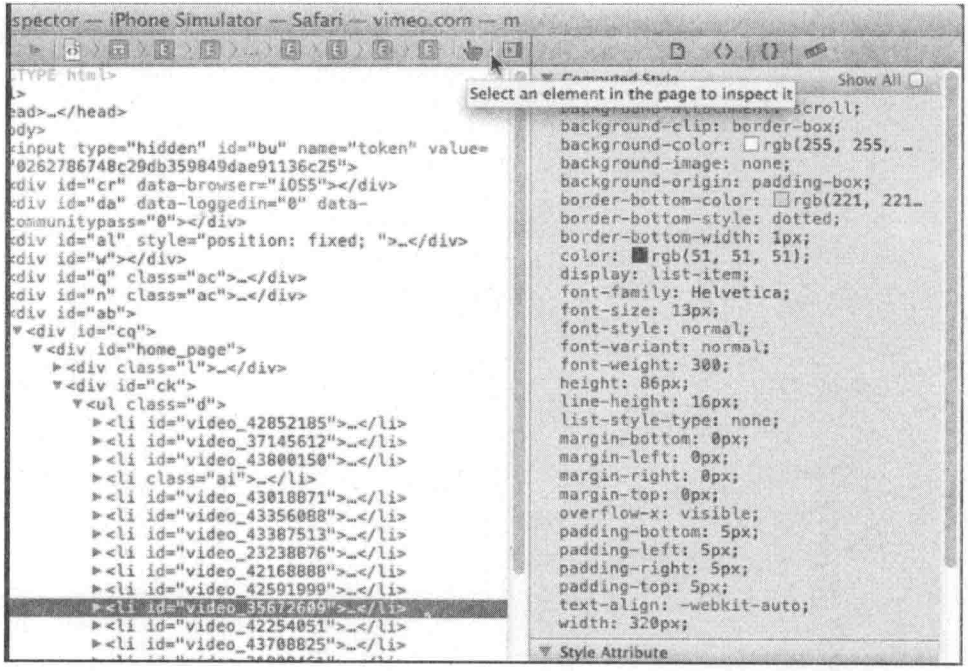


同时需要在 iPhone 模拟器或真机的设置上把调试模式打开，在 iPhone 模拟器或真机中打开应用设置界面，选择“Safari”→“高级”→“Web 检查器”即可，如下图所示。

之后启动模拟器或者真机，通过 USB 连上电脑时，Safari 的“Develop”菜单下就会多出相应的菜单项，如下图所示。



Safari 连接上 UIWebView 后，我们可以在 Safari 中直接修改 HTML 的代码、css 效果，以及调试 JavaScript。所有的效果都可以立即在 UIWebView 上看到。Safari 也提供了“审查元素”功能，可以通过下图中的小手图标完成，单击小手图标，然后在 Web 上想查看的元素上单击一下，就可以跳到该元素对应的 DOM 节点上。


spector — iPhone Simulator — Safari — vimeo.com — m
Select an element in the page to inspect it
Show All
background-clip: border-box;
background-color: rgb(255, 255, 255);
background-image: none;
background-origin: padding-box;
border-bottom-color: rgb(221, 221, 221);
border-bottom-style: dotted;
border-bottom-width: 1px;
color: rgb(51, 51, 51);
display: list-item;
font-family: Helvetica;
font-size: 13px;
font-style: normal;
font-variant: normal;
font-weight: 300;
height: 86px;
line-height: 16px;
list-style-type: none;
margin-bottom: 0px;
margin-left: 0px;
margin-right: 0px;
margin-top: 0px;
overflow-x: visible;
padding-bottom: 5px;
padding-left: 5px;
padding-right: 5px;
padding-top: 5px;
text-align: -webkit-auto;
width: 320px;
Style Attribute

第16章 安全性问题

16.1 前言

在移动互联网快速发展的今天，iOS 应用由于直接运行在用户的手机上，与运行在服务器的后台服务相比，更有可能被黑客攻击。本文接下来将从三个方面概述 iOS 移动应用在安全方面所面临的挑战及应对措施。

16.2 网络安全

16.2.1 安全地传输用户密码

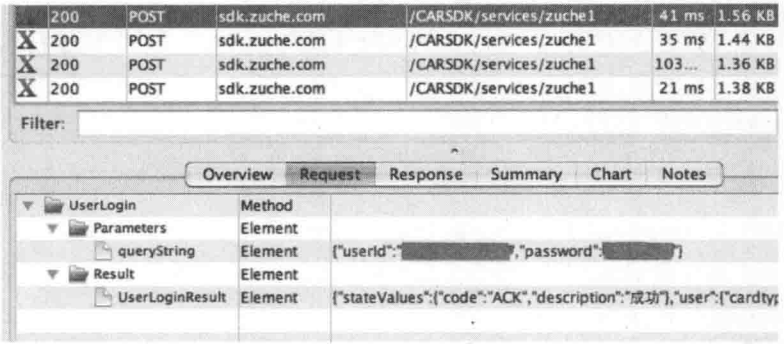
大部分 iOS 应用都需要联网，通过和服务器端进行通讯，获得最新的信息并将内容展现给用户。由于网络传输过程中有可能经过不安全的中间节点，所以我们应该对数据加密保持敏感，以保证用户信息的安全。黑客可以在受害者的手机上设置网络通讯的代理服务器，从而截获所有的网络请求。即使是 HTTPS 的加密通讯，黑客也可以通过中间人攻击（Man-in-the-middle attack，是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制）来截取通讯内容。

黑客会在 Mac 上使用 Charles（http://www.charlesproxy.com/ ）软件来将自己的电脑设置成代理服务器，从而截取应用的网络请求，分析目标应用在通讯协议上是否有安全问题。如果是在 Windows 环境中，则会使用 Fiddler（http://www.telerik.com/fiddler ）软件。为了测试，我选取了国内最大的两家租车公司神州租车和一嗨租车的 iOS 应用。

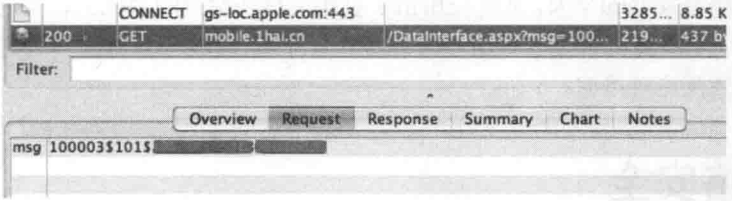
从下面两幅图可以看到，神州租车和一嗨租车在用户登录时，均采用明文的方式，将密码直接发送给服务器。其中一嗨租车不但采用明文方式发送密码，而且在发送时使用了 HTTP GET 的方式，而 GET 的 URL 数据一般都会保存在服务器的 access log 中，所以黑客一旦攻

破服务器，只需要扫描 acesss log，则可以轻易获得所有用户的明文密码。一嗨租车虽然已经修改了登录协议，采用了 POST 的方式来登录，但传递的仍然是明文密码。

神州租车登录协议：



一嗨租车登录协议：



如果每一个应用都像以上两种那样明文传输用户密码，那么我们可以想象这样一个场景，黑客在咖啡馆或机场等一些公共场所，用自己的电脑设置一个与该场所名字相同的免费 WiFi，那么受害者只要不小心使用了该 WiFi，就可能泄漏自己的明文密码。大多数普通人，都会使用一样的密码登录他的所有的账号，这就意味着他的其他账号，例如淘宝或网上银行账号也有被盗的风险。

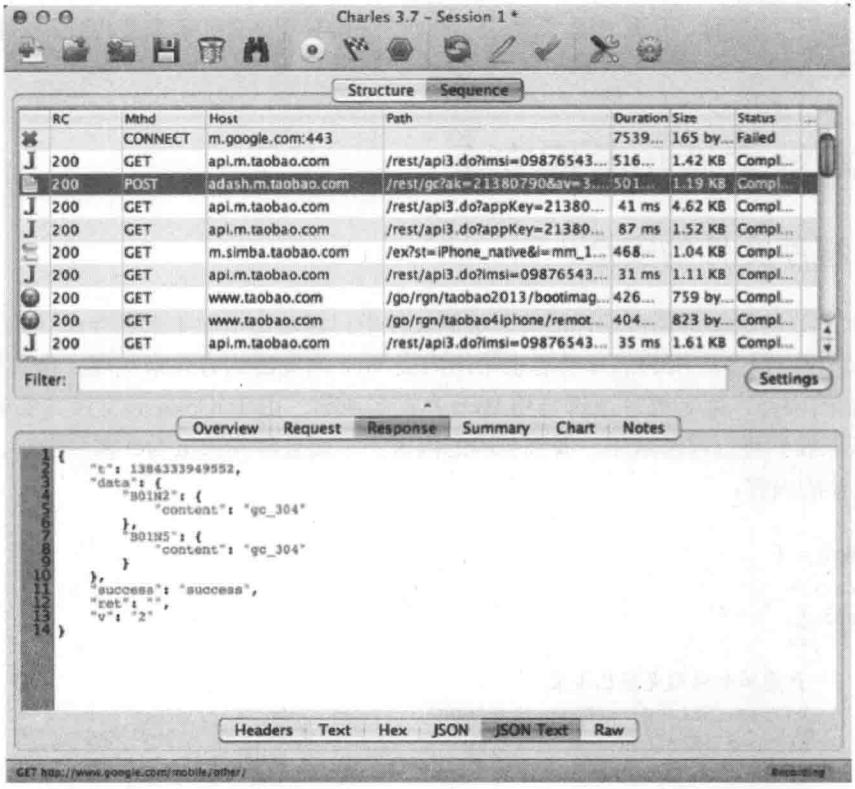
正确的做法应该是这样：事先生成一对用于加密的公私钥，客户端在登录时，使用公钥将用户的密码加密后，将密文传输到服务器。服务器使用私钥将密码解密，然后加盐 (Salt：在密码学中，是指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符) 之后再多次求 MD5，之后再和服务器原来存储的用同样方法处理过的密码匹配，如果一致，则登录成功。这样，黑客即使截获了加密后的密文，由于没有私钥，也无法还原出原始的密码。而服务器即使被黑客攻陷，黑客除了暴力尝试，也无法从加盐和多次 MD5 后的密码中还原出原始的密码。这样就保证了用户密码的安全。

16.2.2 防止通讯协议被轻易破解

除了上面提到的明文传输密码的问题外，移动端应用还要面对黑客破解通讯协议的威胁。在成功破解了通讯协议后，黑客可以模拟客户端登录，进而伪造一些用户行为，可能对用户数据造成危害。例如腾讯出品的消除游戏“天天爱消除”，在淘宝上就有很多售价仅为 1 元的代练服务，如果真正是人工代练，是不可能卖这么便宜的，只可能是该游戏的通讯协议被破解，黑客制作出了代练的机器人程序。

通讯协议被破解除了对于移动端游戏有严重危害外，对于应用也有很大的危害。例如针对微信，黑客可以制作一些“僵尸”账号，通过向微信公共账号后台发送垃圾广告，达到赢利目的。而 iPhone 设备上的 iMessage 通讯协议据说也被破解了，所以很多 iPhone 用户会收到来自 iMessage 的垃圾广告。

对于以上提到的问题，开发者可以选择类似 Protobuf（https://code.google.com/p/protobuf/protobuf，Google 提供的一个开源数据交换格式，其最大的特点是基于二进制，因此比传统的 JSON 格式要小得多）之类的二进制通讯协议或者自己实现通讯协议，对于传输的内容进行一定程度的加密，以增加黑客破解协议的难度。下图是我截取的淘宝客户端的通讯数据，可以看到其中的值都不能直观地猜出内容，所以这对于通讯协议有一定的保护作用。



16.2.3 验证应用内支付的凭证

iOS 应用内支付（IAP）是众多应用赢利的方式，先让用户免费试用或试玩，然后提供应用内支付来为愿意付费的用户提供更强大的功能，这种模式特别适合不习惯一开始就掏钱的中国用户。但是，由于国内“越狱”用户的比例比较大，所以我们也需要注意应用内支付环节中的安全问题。

简单来说，越狱后的手机由于没有沙盒作为保护，黑客可以对系统进行任意地修改，所以在支付过程中，苹果返回的已付款成功的凭证可能是伪造的。客户端拿到付款凭证之后，还需要将凭证上传到自己的服务器上，进行二次验证，以保证凭证的真实性。

另外，我们发现越狱用户的手机，很可能被黑客用中间人攻击技术来劫持支付凭证。这对于黑客有什么好处呢？因为苹果为了保护用户的隐私，支付凭证中并不包含用户的任何账号信息，所以我们的应用和服务器无法知道这个凭证是谁买的，而只能知道这个凭证是真的还是假的。所以在验证凭证时，哪个账号发起了验证请求，我们就默认这个凭证是该账号拥有的。如果黑客将凭证截获，就可以伪装成真实用户来验证凭证或者转手出售获利。

打个比方，这就类似于很多商场的购物卡，由于它们是不记名的，如果黑客将你买的购物卡偷窃然后去刷卡购物，商场是无法简单地区分出来的。

所以，对于应用内支付，开发者除了需要仔细地验证购买凭证外，也需要告知用户在越狱手机上进行支付的风险。

16.3 本地文件和数据安全

16.3.1 程序文件的安全

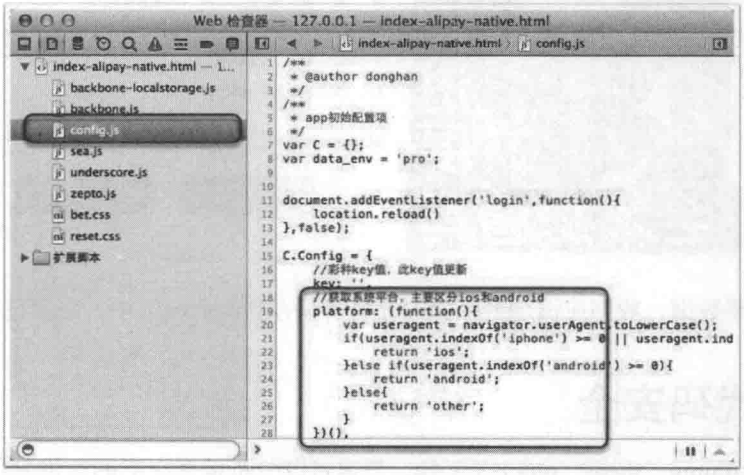
iOS 应用大部分的逻辑都是在编译后的二进制文件中，但是由于近年来混合式（Hybrid）编程方式的兴起，很多应用的部分功能也采用内嵌 Web 浏览器的方式来实现。例如腾讯 QQ iOS 客户端的内部，就有部分逻辑是用 Web 方式实现的。由于 iOS 安装文件其实就是一个 zip 包，所以我们可以通过解压，看到包内的内容。下面是解开腾讯 QQ 客户端后，看到的 qqapi.js 文件的内容：

```
iOSQQApi = {
    // ...
    app: {
        /**
         查询单个应用是否已安装
         @param {String} scheme 比如'mqq'
         @return {Boolean}
         */
```

```
        isAppInstalled: function(scheme) {
            return iOSQQApi._invokeClientMethod(
                'app', 'isInstalled',
                {'scheme':scheme});
        },

        /**
         批量查询指定应用是否已安装
         @param {Array<String>} schemes
                比如['mqq', 'mqqapi']
         @return {Array<Boolean>}
         */
        isAppInstalledBatch: function(schemes) {
            return iOSQQApi._invokeClientMethod(
                'app','batchIsInstalled',
                {'schemes':schemes});
        }
    },
    // ...
}
```

可以看到，这些文件都有着完整清晰的注释。通过分析这些 JavaScript 文件，黑客可以比较轻松地知道其调用逻辑，在越狱手机上，还可能修改这些 JavaScript 代码，以达到攻击的目的。

我也曾经尝试查看支付宝客户端中的彩票功能，通过分析，也可以找到其完整的、带着清晰注释的 JavaScript 代码，如下图所示。（注：支付宝现在已经对相应代码进行了加密）



通过将 JavaScript 源码进行混淆和加密，可以防止黑客轻易地阅读和篡改相关的逻辑，也可以防止自己的 Web 端与 Native 端的通讯协议泄漏。

16.3.2 本地数据安全

iOS 应用的数据在本地通常保存在本地文件或本地数据库中。如果本地的数据未进行加密处理，很可能被黑客篡改。以下是一个名为 LepsWorld3 的游戏，打开它的本地文件，可以很容易地看到，它使用了一个名为 ItempLifes 的变量保存生命数，如下图所示。

于是我们就可以简单修改该值，达到修改游戏参数的目的。而在淘宝上，也可以找到许多以此挣钱的商家，如下图所示。

对于本地的重要数据，我们应该加密存储或者将其保存到 keychain 中，以保证其不被篡改。

16.4 源代码安全

通过 file、class-dump、theos、otool 等工具，黑客可以分析编译之后的二进制程序文件，不过相对于这些工具来说，IDA 的威胁最大。

IDA（https://www.hex-rays.com/products/ida/）是一个收费的反汇编工具，对于 Objective-C 代码，它常常可以反汇编到可以方便阅读的程度，这对于程序的安全性，也是一个很大的危害。因为通过阅读源码，黑客可以更加方便地分析出应用的通讯协议和数据加密方式。

下面分别示例了一段代码的原始内容，以及通过 IDA 反汇编之后的结果。可以看到，IDA 几乎还原了原本的逻辑，而且可读性也非常强。

原始代码：

```
if ([[VersionAgent sharedInstance] isUpgraded]) {
    UpdateMigrationAgent *agent =
            [[UpdateMigrationAgent alloc] init];
    [FileUtils clearCacheDirectory];
    [[VersionAgent sharedInstance] saveAppVersion];
}
```

反汇编后：

```
v6 = _objc_msgSend(&OBJC_CLASS___VersionAgent,
                   "sharedInstance");
v7 = objc_retainAutoreleasedReturnValue(v6);
v41 = _objc_msgSend(v7, "isUpgraded");
objc_release(v7);
if ( v41 )
{
    NSLog(CFSTR("app is upgraded"), v41);
    _objc_msgSend(&OBJC_CLASS___FileUtils,
                  "clearCacheDirectory");
    v8 = _objc_msgSend(&OBJC_CLASS___VersionAgent,
                       "sharedInstance");
    v9 = objc_retainAutoreleasedReturnValue(v8);
    _objc_msgSend(v9, "saveAppVersion");
    objc_release(v9);
}
```

获得反汇编的代码后，由于软件内部逻辑相对汇编代码来说可读性强了很多，黑客可以用来制作软件的注册机，也可以更加方便地破解网络通讯协议，从而制作出机器人（“僵尸”）账号。在最极端的情况下，黑客可以将反汇编的代码稍加修改，植入木马，然后重新打包发布在一些越狱渠道上，这将对用户产生巨大的危害。

对于 IDA 这类工具，我们的应对措施就比较少了。除了可以用一些宏来简单混淆类名外，我们也可以将关键的逻辑用纯 C 实现。例如微信的 iOS 端的通讯底层，就是用 C 实现的。这样的方式除了能保证通讯协议安全外，也可以在 iOS 和 Android 等多个平台使用同一套底层通讯代码，达到复用的目的。

16.5 总结

由于移动互联网的快速发展，人们的购物、理财等需求也在移动端出现，这使得移动应用的安全性越来越重要。由于部署在用户终端上，移动应用比服务器应用更容易被攻击，大家也需要在移动应用的网络通讯、本地文件和数据、源代码三方面做好防范，只有这样才能保证应用的安全。

第17章 基于CoreText的排版引擎

使用CoreText技术，我们可以对富文本进行复杂的排版。经过一些简单的扩展，我们还可以实现对于图片、链接的点击效果。CoreText技术相对于UIWebView，有内存占用少，以及可以在后台渲染的优点，非常适合排版工作。

我们将从最基本的开始，一步一步完成一个支持图文混排、支持图片和链接点击的排版引擎。

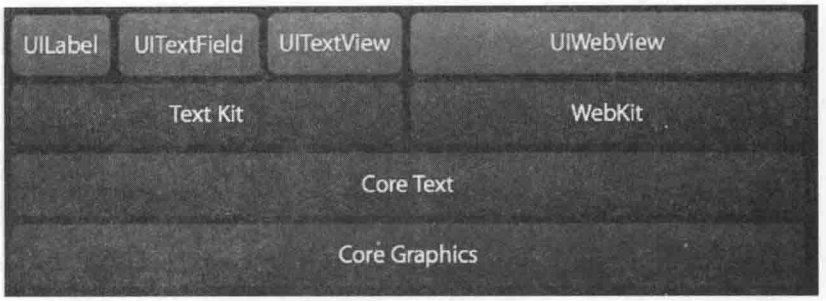
17.1 CoreText简介

CoreText是用于处理文字和字体的底层技术。它直接和Core Graphics（又被称为Quartz）打交道。Quartz是一个2D图形渲染引擎，能够处理OSX和iOS中的图形显示问题。

Quartz能够直接处理字体（font）和字形（glyphs），将文字渲染到界面上，它是基础库中唯一能够处理字形的模块。因此，CoreText为了排版，需要将显示的文本内容、位置、字体、字形直接传递给Quartz。与其他UI组件相比，由于CoreText直接和Quartz来交互，所以它具有高效的排版功能。

下图是CoreText的架构图，可以看到，CoreText处于非常底层的位置，上层的UI控件（包括UILabel、UITextField及UITextView）和UIWebView都是基于CoreText来实现的。[1]

[1] 这个是iOS7之后的架构图，在iOS7以前，并没有图中的Text Kit类，不过CoreText仍然是处在最底层直接和Core Graphics打交道的模块。



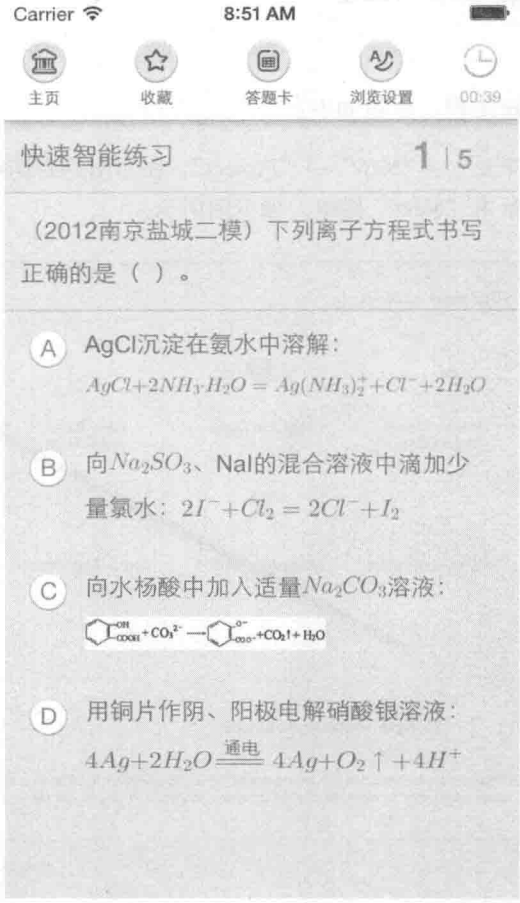
UIWebView 也是处理复杂的文字排版的备选方案。对于排版，基于 CoreText 和基于 UIWebView 相比，具有以下不同之处：

- CoreText 占用的内存更少，渲染速度更快，UIWebView 占用的内存多，渲染速度慢。
- CoreText 在渲染界面前就可以精确地获得显示内容的高度（只要有了 CTFrame 即可），而 UIWebView 只有渲染出内容后，才能获得内容的高度（而且还需要用 JavaScript 代码来获取）。
- CoreText 的 CTFrame 可以在后台线程渲染，UIWebView 的内容只能在主线程（UI 线程）渲染。
- 基于 CoreText 可以做更好的原生交互效果，交互效果可以更细腻。而 UIWebView 的交互效果都是用 JavaScript 来实现的，在交互效果上会有一些卡顿情况存在。例如，在 UIWebView 下，一个简单的按钮按下操作，都无法做出原生按钮的即时和细腻的按下效果。

当然，基于 CoreText 的排版方案也有一些劣势：

- CoreText 渲染出来的内容不能像 UIWebView 那样方便地支持内容的复制。
- 基于 CoreText 来排版需要自己处理很多复杂逻辑，例如需要自己处理图片与文字混排相关的逻辑，也需要自己实现链接点击操作的支持。

在业界，很多应用都采用了基于 CoreText 技术的排版方案，例如：新浪微博客户端、多看阅读客户端。我所在的创业公司的猿题库，也使用了自己基于 CoreText 技术实现的排版引擎，下图是我们产品的一个图文混排的界面（其中反应式是用图片的方式呈现的），可以看到，图片和文字排版效果很好。



17.2 基于 CoreText 的基础排版引擎

不带图片的排版引擎

下面我们来尝试完成一个基于 CoreText 的排版引擎。我们将从最简单的排版功能开始，然后逐步支持图文混排、链接点击等功能。

首先我们来尝试完成一个不支持图片内容的纯文字排版引擎。[2]

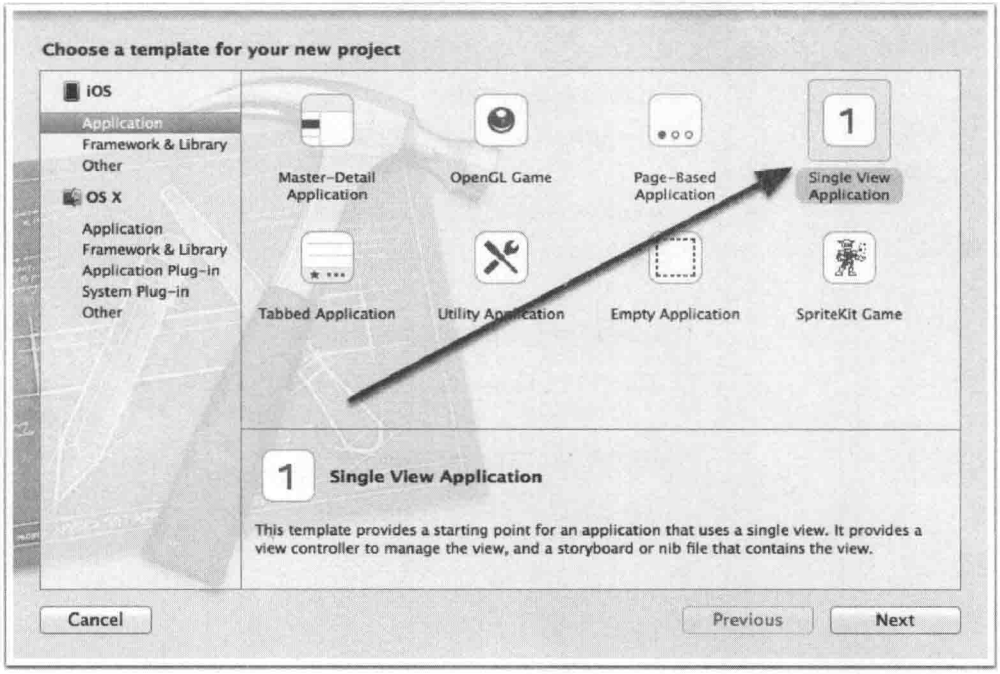
[2] 由于整个排版引擎的代码太多，为方便读者阅读，文章中只会列出最关键的核心代码，完整的代码请参考本书对应的 github 项目，项目地址是：https://github.com/tangqiaoboy/iOS-Pro 。

能输出“Hello World”的 CoreText 工程

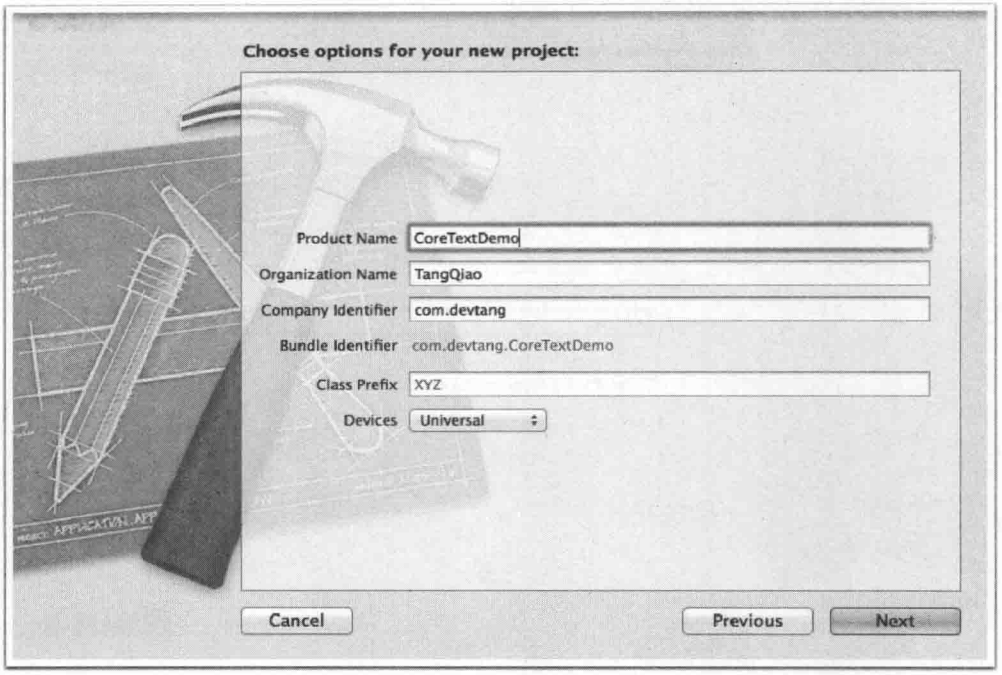
操作步骤

我们首先新建一个 Xcode 工程，步骤如下：

1. 打开 Xcode，选择“File”→“New”→“Project”，在弹出的对话框中，选择“Single View Application”，然后单击“Next”按钮，如下图所示。

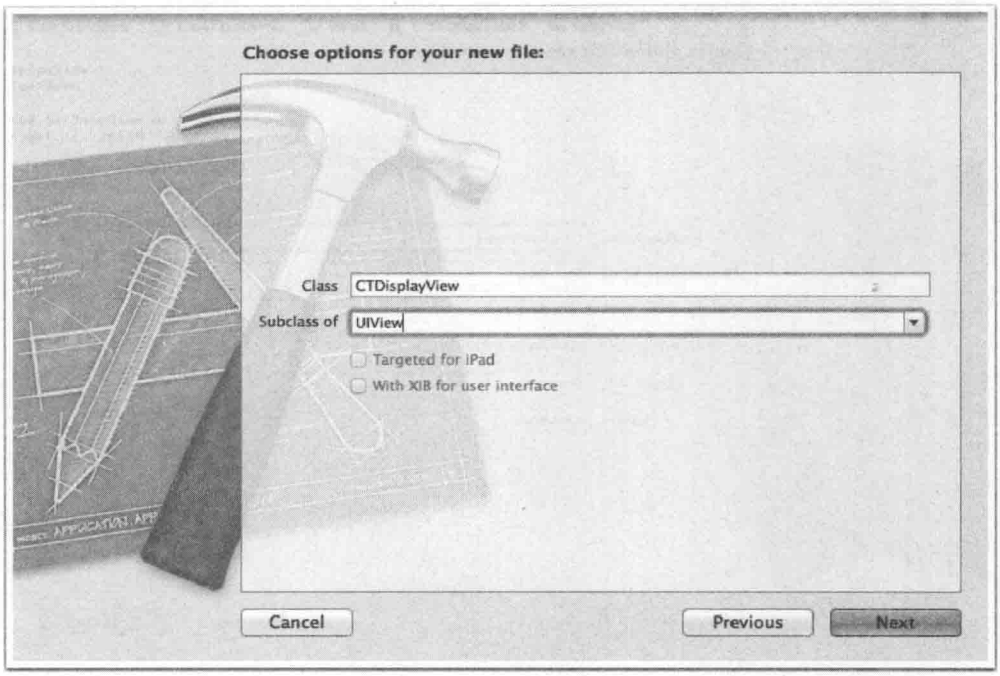


2. 接着填上项目名“CoreTextDemo”，然后单击“Next”按钮，如下图所示。



3. 选择保存目录后，我们就成功创建了一个空的工程。

在工程目录“CoreTextDemo”上单击右键，选择“New File”，然后填入类名“CTDisplayView”，并且让它的父类是 UIView，如下图所示。



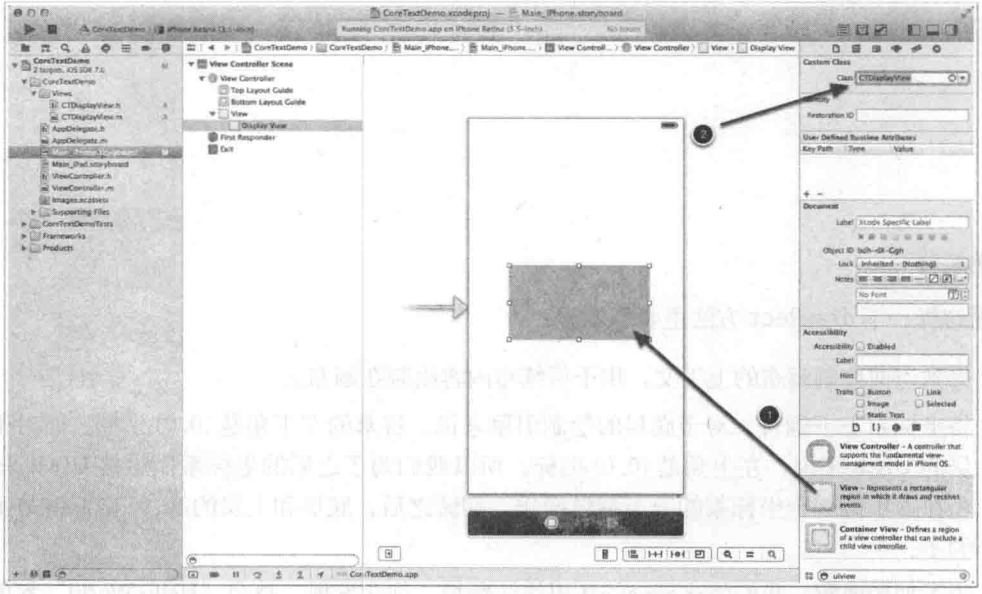
接着，我们在 CTDisplayView.m 文件中，让其 import 头文件为 CoreText/CoreText.h，接着输入以下代码来实现其 drawRect 方法：

```
#import "CTDisplayView.h"
#import "CoreText/CoreText.h"

@implementation CTDisplayView

- (void)drawRect:(CGRect)rect
{
    [super drawRect:rect];

    // 步骤1
    CGContextRef context = UIGraphicsGetCurrentContext();

    // 步骤2
    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);

    // 步骤3
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddRect(path, NULL, self.bounds);
```

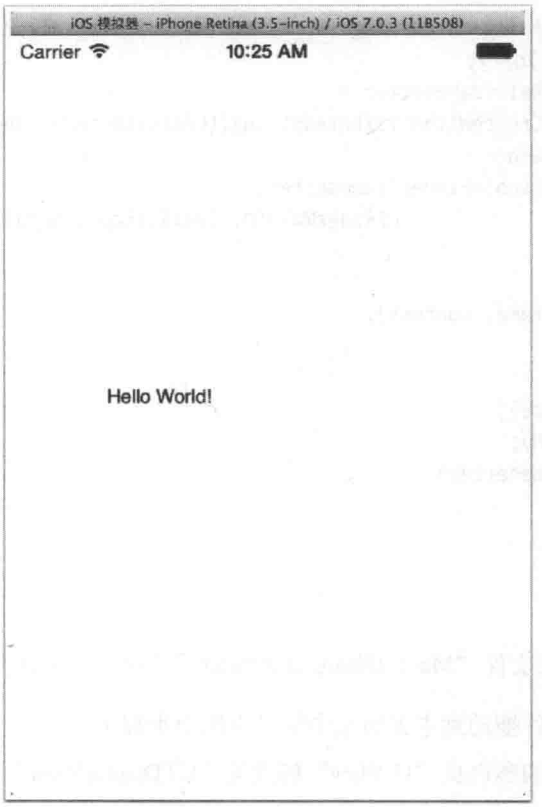
```
    // 步骤4
    NSAttributedString *attString = [[NSAttributedString alloc] initWithString:@"
        Hello World!"];
    CTFramesetterRef framesetter =
    CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attString);
    CTFrameRef frame =
    CTFramesetterCreateFrame(framesetter,
                             CFRangeMake(0, [attString length]), path, NULL);

    // 步骤5
    CTFrameDraw(frame, context);

    // 步骤6
    CFRelease(frame);
    CFRelease(path);
    CFRelease(framesetter);
}

@end
```

打开程序的 storyboard 文件“Main_iPhone.storyboard”，执行下面两步：

1. 将一个 UIView 控件拖动到主界面正中间（下图中步骤 1）。
2. 将该 UIView 控件的类名从“UIView”修改为“CTDisplayView”（下图中步骤 2）。



之后，我们运行程序，就可以看到，“Hello World”出现在程序正中间了，如下图所示。



代码解释

下面解释一下 drawRect 方法主要的步骤：

1. 得到当前绘制画布的上下文，用于后续将内容绘制在画布上。
2. 将坐标系上下翻转。对于底层的绘制引擎来说，屏幕的左下角是 (0, 0) 坐标。而对于上层的 UIKit 来说，左上角是 (0, 0) 坐标。所以我们为了之后的坐标系描述按 UIKit 来做，先在这里做一个坐标系的上下翻转操作。翻转之后，底层和上层的 (0, 0) 坐标就是重合的了。

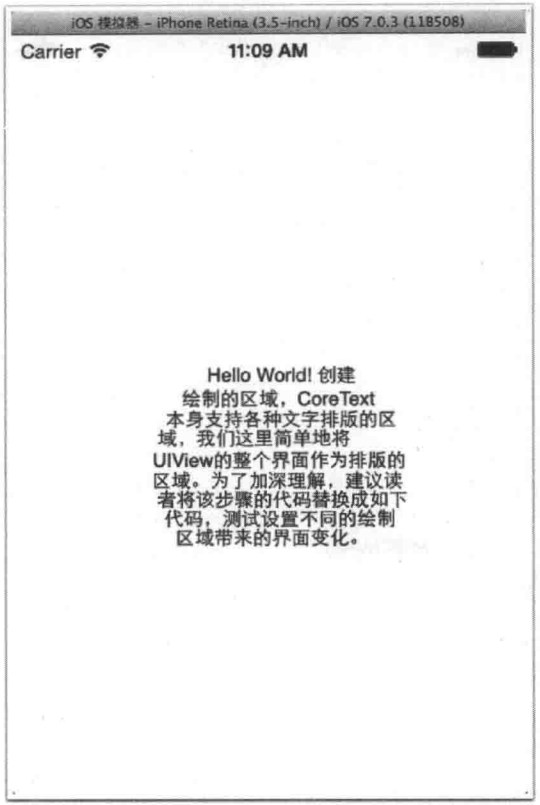
   为了加深理解，我们将这部分的代码块注释掉，你会发现，整个“Hello World”界面将上下翻转，如下图所示。

3. 创建绘制的区域，CoreText 本身支持各种文字排版的区域，我们这里简单地将 UIView 的整个界面作为排版的区域。

为了加深理解，我们将该步骤的代码替换成如下代码，测试设置不同的绘制区域带来的界面变化。

```
// 步骤3
CGMutablePathRef path = CGPathCreateMutable();
CGPathAddEllipseInRect(path, NULL, self.bounds);

// 步骤4
NSAttributedString *attString =
    [[NSAttributedString alloc] initWithString:@"Hello World! "
                "创建绘制的区域，CoreText本身支持各种文字排版的区域，"
                "我们这里简单地将UIView的整个界面作为排版的区域。"
                "为了加深理解，建议读者将该步骤的代码替换成如下代码，"
                "测试设置不同的绘制区域带来的界面变化。"];
```

执行结果如下图所示。



代码基本的宏定义和 Category

为了方便我们的代码编写，我在 CoreTextDemo-Prefix.pch 文件中增加了以下基本的宏定义，以方便我们使用 NSLog 和 UIColor。

```
#ifdef DEBUG
#define debugLog(...) NSLog(__VA_ARGS__)
#define debugMethod() NSLog(@"%s", __func__)
#else
#define debugLog(...)
#define debugMethod()
#endif

#define RGB(A, B, C)    [UIColor colorWithRed:A/255.0 green:B/255.0 blue:C/255.0
    alpha:1.0]
```

我也为 UIView 的 frame 调整增加了一些扩展，这样可以方便地调整 UIView 的 x、y、width、height 等值。部分关键代码如下（完整的代码请查看示例工程）：

UIView+frameAdjust.h 文件：

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

@interface UIView (frameAdjust)

- (CGFloat)x;
- (void)setX:(CGFloat)x;

- (CGFloat)y;
- (void)setY:(CGFloat)y;

- (CGFloat)height;
- (void)setHeight:(CGFloat)height;

- (CGFloat)width;
- (void)setWidth:(CGFloat)width;

@end
```

UIView+frameAdjust.m 文件：

```
@implementation UIView (frameAdjust)
- (CGFloat)x {
    return self.frame.origin.x;
}

- (void)setX:(CGFloat)x {
    self.frame = CGRectMake(x, self.y, self.width, self.height);
}

- (CGFloat)y {
    return self.frame.origin.y;
}

- (void)setY:(CGFloat)y {
    self.frame = CGRectMake(self.x, y, self.width, self.height);
}

- (CGFloat)height {
    return self.frame.size.height;
}
- (void)setHeight:(CGFloat)height {
    self.frame = CGRectMake(self.x, self.y, self.width, height);
}

- (CGFloat)width {
    return self.frame.size.width;
```

```
}
- (void)setWidth:(CGFloat)width {
    self.frame = CGRectMake(self.x, self.y, width, self.height);
}

@end
```

文章中的其余代码默认都 #import 了以上提到的宏定义和 UIView Category。

排版引擎框架

上面的“Hello World”工程仅仅展示了 Core Text 排版的基本能力。但是要制作一个较完善的排版引擎，我们不能简单地将所有代码都放到 CTDisplayView 的 drawRect 方法里面。根据设计模式中的“单一功能原则 (Single responsibility principle)”[3]，我们应该把功能拆分，把不同的功能都放到各自不同的类里面。

对于一个复杂的排版引擎来说，可以将其功能拆成以下几个类来完成：

1. 一个显示用的类，仅负责显示内容，不负责排版。
2. 一个模型类，用于承载显示所需要的所有数据。
3. 一个排版类，用于实现文字内容的排版。
4. 一个配置类，用于实现一些排版时的可配置项。

按照以上原则，我们将 CTDisplayView 中的部分内容拆开，分成 4 个类：

1. CTFrameParserConfig 类，用于配置绘制的参数，例如文字颜色、大小，行间距等。
2. CTFrameParser 类，用于生成最后绘制界面需要的 CTFrameRef 实例。
3. CoreTextData 类，用于保存由 CTFrameParser 类生成的 CTFrameRef 实例，以及 CTFrameRef 实际绘制需要的高度。
4. CTDisplayView 类，持有 CoreTextData 类的实例，负责将 CTFrameRef 绘制到界面上。

关于这 4 个类的关键代码如下。

CTFrameParserConfig 类：

```
#import <Foundation/Foundation.h>
@interface CTFrameParserConfig : NSObject

@property (nonatomic, assign) CGFloat width;
@property (nonatomic, assign) CGFloat fontSize;
```

[3] “单一功能原则”(Single responsibility principle) 参考链接：http://zh.wikipedia.org/wiki/%E5%8D%95%E4%B8%80%E5%8A%9F%E8%83%BD%E5%8E%9F%E5%88%99。

```
@property (nonatomic, assign) CGFloat lineSpace;
@property (nonatomic, strong) UIColor *textColor;

@end

#import "CTFrameParserConfig.h"

@implementation CTFrameParserConfig

- (id)init {
    self = [super init];
    if (self) {
        _width = 200.0f;
        _fontSize = 16.0f;
        _lineSpace = 8.0f;
        _textColor = RGB(108, 108, 108);
    }
    return self;
}

@end
```

CTFrameParser 类：

```
#import <Foundation/Foundation.h>
#import "CoreTextData.h"
#import "CTFrameParserConfig.h"

@interface CTFrameParser : NSObject

+ (CoreTextData *)parseContent:(NSString *)content config:(CTFrameParserConfig*)
    config;

@end

#import "CTFrameParser.h"
#import "CTFrameParserConfig.h"

@implementation CTFrameParser

+ (NSDictionary *)attributesWithConfig:(CTFrameParserConfig *)config {
    CGFloat fontSize = config.fontSize;
    CTFontRef fontRef = CTFontCreateWithName((CFStringRef)@"ArialMT", fontSize, NULL
        );
    CGFloat lineSpacing = config.lineSpace;
    const CFIndex kNumberOfSettings = 3;
    CTParagraphStyleSetting theSettings[kNumberOfSettings] = {
```

```
        { kCTParagraphStyleSpecifierLineSpacingAdjustment, sizeof(CGFloat), &
            lineSpacing },
        { kCTParagraphStyleSpecifierMaximumLineSpacing, sizeof(CGFloat), &
            lineSpacing },
        { kCTParagraphStyleSpecifierMinimumLineSpacing, sizeof(CGFloat), &
            lineSpacing }
    };

    CTParagraphStyleRef theParagraphRef = CTParagraphStyleCreate(theSettings,
        kNumberOfSettings);

    UIColor * textColor = config.textColor;

    NSMutableDictionary * dict = [NSMutableDictionary dictionary];
    dict[(id)kCTForegroundColorAttributeName] = (id)textColor.CGColor;
    dict[(id)kCTFontAttributeName] = (__bridge id)fontRef;
    dict[(id)kCTParagraphStyleAttributeName] = (__bridge id)theParagraphRef;

    CFRelease(theParagraphRef);
    CFRelease(fontRef);
    return dict;
}

+ (CoreTextData *)parseContent:(NSString *)content config:(CTFrameParserConfig*)
    config {
    NSDictionary *attributes = [self attributesWithConfig:config];
    NSAttributedString *contentString =
        [[NSAttributedString alloc] initWithString:content
                                        attributes:attributes];

    // 创建CTFramesetterRef实例
    CTFramesetterRef framesetter = CTFramesetterCreateWithAttributedString((
        CFAttributedStringRef)contentString);

    // 获得要绘制的区域的高度
    CGSize restrictSize = CGSizeMake(config.width, CGFLOAT_MAX);
    CGSize coreTextSize = CTFramesetterSuggestFrameSizeWithConstraints(framesetter,
        CFRangeMake(0,0), nil, restrictSize, nil);
    CGFloat textHeight = coreTextSize.height;

    // 生成CTFrameRef实例
    CTFrameRef frame = [self createFrameWithFramesetter:framesetter config:config
        height:textHeight];

    // 将生成好的CTFrameRef实例和计算好的绘制高度保存到CoreTextData实例中，最后返回
        CoreTextData实例
```

```
    CoreTextData *data = [[CoreTextData alloc] init];
    data.ctFrame = frame;
    data.height = textHeight;

    // 释放内存
    CFRelease(frame);
    CFRelease(framesetter);
    return data;
}

+ (CTFrameRef)createFrameWithFramesetter:(CTFramesetterRef)framesetter
                                  config:(CTFrameParserConfig *)config
                                  height:(CGFloat)height {

    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddRect(path, NULL, CGRectMake(0, 0, config.width, height));

    CTFrameRef frame = CTFramesetterCreateFrame(framesetter, CFRangeMake(0, 0), path
        , NULL);
    CFRelease(path);
    return frame;
}

@end
```

CoreTextData 类：

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

@interface CoreTextData : NSObject

@property (assign, nonatomic) CTFrameRef ctFrame;
@property (assign, nonatomic) CGFloat height;

@end

#import "CoreTextData.h"

@implementation CoreTextData

- (void)setCtFrame:(CTFrameRef)ctFrame {
    if (_ctFrame != ctFrame) {
        if (_ctFrame != nil) {
            CFRelease(_ctFrame);
        }
        CFRetain(ctFrame);
        _ctFrame = ctFrame;
```

```
    }
}

- (void)dealloc {
    if (_ctFrame != nil) {
        CFRelease(_ctFrame);
        _ctFrame = nil;
    }
}

@end
```

CTDisplayView 类：

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

@interface CTDisplayView : UIView

@property (strong, nonatomic) CoreTextData * data;

@end

#import "CTDisplayView.h"

@implementation CTDisplayView

- (void)drawRect:(CGRect)rect
{
    [super drawRect:rect];
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);

    if (self.data) {
        CTFrameDraw(self.data.ctFrame, context);
    }
}

@end
```

以上 4 个类中的逻辑与之前“Hello World”那个项目的逻辑基本一致，只是分拆到了 4 个类中完成。另外，CTFrameParser 增加了方法来获得要绘制的区域的高度，并将高度信息保存到 CoreTextData 类的实例中。之所以要获得绘制区域的高度，是因为在很多实际使用场景中，我们需要先知道所要显示内容的高度，之后才可以进行绘制。

例如，UITableView 在渲染时，UITableView 首先会向 delegate 回调如下方法来获得每个将要渲染的 Cell 的高度：

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath
    *)indexPath;
```

之后，UITableView 会计算当前滚动的位置具体需要绘制的 UITableViewCell 是哪些，然后对于那些需要绘制的 Cell，UITableView 才会继续向其 data source 回调如下方法来获得 UITableViewCell 实例：

```
- (UITableViewCell *)cellForRowAtIndexPath:(NSIndexPath *)indexPath;
```

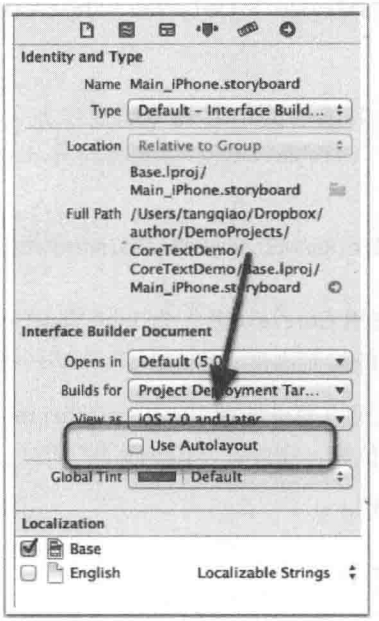
对于上面的情况，如果我们使用 CoreText 来作为 TableViewCell 的内容，那么就必须在每个 Cell 绘制之前，就知道其需要的绘制高度，否则 UITableView 将无法正常工作。

完成以上 4 个类之后，我们就可以简单地在 ViewController.m 文件中，加入如下代码来配置 CTDisplayView 的显示内容、位置、高度、字体、颜色等信息。代码如下所示：

```
#import "ViewController.h"

@interface ViewController ()

@property (weak, nonatomic) IBOutlet CTDisplayView *ctView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    CTFrameParserConfig *config = [[CTFrameParserConfig alloc] init];
    config.textColor = [UIColor redColor];
    config.width = self.ctView.width;

    CoreTextData *data = [CTFrameParser parseContent:@"按照以上原则，我们将`
        CTDisplayView`中的部分内容拆开。" config:config];
    self.ctView.data = data;
    self.ctView.height = data.height;
    self.ctView.backgroundColor = [UIColor yellowColor];
}

@end
```

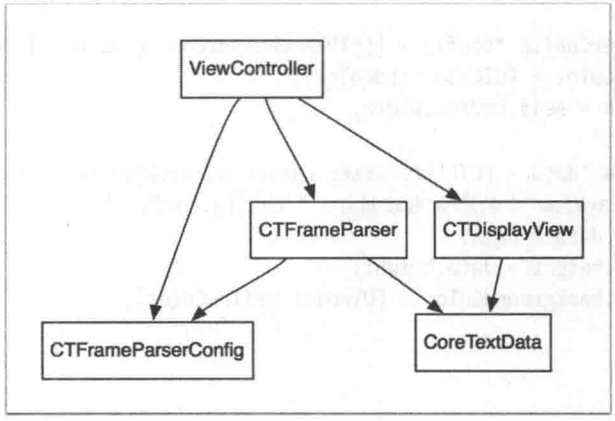
从 Xcode4.0 开始，默认的界面编辑就开启了对于 Use Autolayout 的使用，但因为我们在代

码中直接修改了变量 ctView 的 frame 信息，所以需要在 Main_iPhone.storyboard 中将 Use Autolayout 这一项取消勾选，如下图所示。



下面是本框架的 UML 示意图，从图中我们可以看出，这 4 个 Core Text 类的关系是这样的：

1. CTFrameParser 通过 CTFrameparserConfig 实例来生成 CoreTextData 实例。
2. CTDisplayView 通过持有 CoreTextData 实例来获得绘制所需要的所有信息。
3. ViewController 类通过配置 CTFrameparserConfig 实例，进而获得生成的 CoreTextData 实例，最后将其赋值给它的 CTDisplayView 成员，达到将指定内容显示在界面上的效果。



说明 1：整个工程代码在名为“basic_arch”的分支下，读者可以在示例的源代码工程中使用

git checkout basic_arch 来切换到当前讲解的工程示例代码。

说明2：为了方便操作UIView的frame属性，项目中增加了一个名为“UIView+frameAdjust.m”的文件，它通过Category来给UIView增加了直接设置height属性的方法。

定制排版文件格式

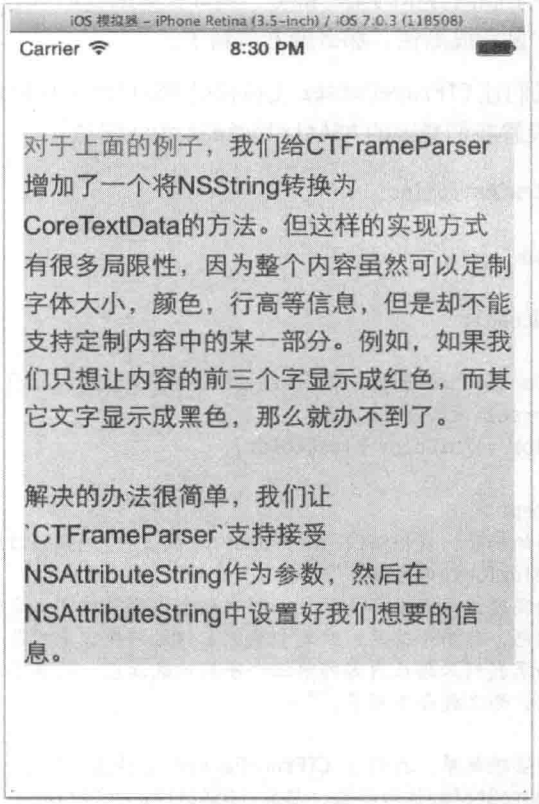
对于上面的例子，我们给CTFrameParser增加了一个将NSString转换为CoreTextData的方法。但这样的实现方式有很多局限性，因为整个内容虽然可以定制字体大小、颜色、行高等信息，但是却不能支持定制内容中的某一部分。例如，如果我们只想让内容的前三个字显示成红色，而让其他文字显示成黑色，那么就办不到了。

解决的办法很简单，我们让CTFrameParser支持接受NSAttributeString作为参数，然后在ViewController类中设置我们想要的NSAttributeString信息。

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    CTFrameParserConfig *config = [[CTFrameParserConfig alloc] init];
    config.width = self.ctView.width;
    config.textColor = [UIColor blackColor];

    NSString *content =
        @"对于上面的例子，我们给CTFrameParser增加了一个将NSString转"
         "换为CoreTextData的方法。"
         "但这样的实现方式有很多局限性，因为整个内容虽然可以定制字体"
         "大小，颜色，行高等信息，但是却不能支持定制内容中的某一部分。"
         "例如，如果我们只想让内容的前三个字显示成红色，而其它文字显"
         "示成黑色，那么就办不到了。"
         "\n\n"
         "解决的办法很简单，我们让`CTFrameParser`支持接受"
         "NSAttributeString作为参数，然后在NSAttributeString中设置好"
         "我们想要的信息。";
    NSDictionary *attr = [CTFrameParser attributesWithConfig:config];
    NSMutableAttributedString *attributedString =
        [[NSMutableAttributedString alloc] initWithString:content
                                                attributes:attr];
    [attributedString addAttribute:NSForegroundColorAttributeName
                             value:[UIColor redColor]
                             range:NSMakeRange(0, 7)];

    CoreTextData *data = [CTFrameParser parseAttributedContent:attributedString
                                                        config:config];
```

```
    self.ctView.data = data;
    self.ctView.height = data.height;
    self.ctView.backgroundColor = [UIColor yellowColor];
}

@end
```

结果如下图所示，我们很方便地就把前面 7 个字变成了红色。



更进一步地，实际工作中，我们更希望通过一个排版文件，来设置需要排版的文字的内容、颜色、字体大小等信息。我在开发猿题库应用时，自己定义了一个基于 UBB 的排版模板，但是实现该排版文件的解析器要占用大量的篇幅，考虑到这并不是本章的重点，所以我们以一个较简单的排版文件来讲解其思想。

我们规定排版的模板文件为 JSON 格式。JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于阅读和编写，同时也易于机器解析和生成。iOS 从 5.0 版开始，提供了名为 NSJSONSerialization 的类库来方便开发者对 JSON 的解析。在 iOS5.0 之前，业界也有很多相关的 JSON 解析开源库，例如 JSONKit 等，可供大家使用。

我们的排版模板示例文件如下所示。

```
[ { "color" : "blue",
    "content" : "更进一步地，实际工作中，我们更希望通过一个排版文件，来设置需要排版
        的文字的 ",
    "size" : 16,
    "type" : "txt"
  },
  { "color" : "red",
    "content" : "内容、颜色、字体",
    "size" : 22,
    "type" : "txt"
  },
  { "color" : "black",
    "content" : "大小等信息。\n",
    "size" : 16,
    "type" : "txt"
  },
  { "color" : "default",
    "content" : "我在开发猿题库应用时，自己定义了一个基于UBB的排版模板，但是实现该排
        版文件的解析器要花费大量的篇幅，考虑到这并不是本章的重点，所以我们以一个较
        简单的排版文件来讲解其思想。",
    "type" : "txt"
  }
]
```

通过苹果提供的 NSJSONSerialization 类，我们可以将上面的模板文件转换成 NSArray 数组，每一个数组元素是一个 NSDictionary，代表一段相同设置的文字。为了简单，我们的配置文件只支持配置颜色和字号，但是读者可以依据同样的思想，很方便地增加其他配置信息。

接下来我们要为 CTFrameParser 增加一个方法，让其可以从如上格式的模板文件中生成 CoreTextData。最终我们的实现代码如下：

```
// 方法一
+ (CoreTextData *)parseTemplateFile:(NSString *)path config:(CTFrameParserConfig*)
    config {
    NSAttributedString *content = [self loadTemplateFile:path config:config];
    return [self parseAttributedContent:content config:config];
}

// 方法二
+ (NSAttributedString *)loadTemplateFile:(NSString *)path config:(
    CTFrameParserConfig*)config {
    NSData *data = [NSData dataWithContentsOfFile:path];
    NSMutableAttributedString *result = [[NSMutableAttributedString alloc] init];
    if (data) {
```

```
        NSArray *array =
            [NSJSONSerialization JSONObjectWithData:data
                                            options:NSJSONReadingAllowFragments
                                              error:nil];
        if ([array isKindOfClass:[NSArray class]]) {
            for (NSDictionary *dict in array) {
                NSString *type = dict[@"type"];
                if ([type isEqualToString:@"txt"]) {
                    NSAttributedString *as =
                        [self parseAttributedContentFromNSDictionary:dict
                                                              config:config];
                    [result appendAttributedString:as];
                }
            }
        }
    }
    return result;
}

// 方法三
+ (NSAttributedString *)parseAttributedContentFromNSDictionary:(NSDictionary *)dict
                                                        config:(CTFrameParserConfig
                                                            *)config {
    NSMutableDictionary *attributes = [self attributesWithConfig:config];
    // set color
    UIColor *color = [self colorFromTemplate:dict[@"color"]];
    if (color) {
        attributes[(id)kCTForegroundColorAttributeName] = (id)color.CGColor;
    }
    // set font size
    CGFloat fontSize = [dict[@"size"] floatValue];
    if (fontSize > 0) {
        CTFontRef fontRef = CTFontCreateWithName((CFStringRef)@"ArialMT", fontSize,
            NULL);
        attributes[(id)kCTFontAttributeName] = (__bridge id)fontRef;
        CFRelease(fontRef);
    }
    NSString *content = dict[@"content"];
    return [[NSAttributedString alloc] initWithString:content attributes:attributes
        ];
}

// 方法四
+ (UIColor *)colorFromTemplate:(NSString *)name {
    if ([name isEqualToString:@"blue"]) {
        return [UIColor blueColor];
```

```
    } else if ([name isEqualToString:@"red"]) {
        return [UIColor redColor];
    } else if ([name isEqualToString:@"black"]) {
        return [UIColor blackColor];
    } else {
        return nil;
    }
}

// 方法五
+ (CoreTextData *)parseAttributedContent:(NSAttributedString *)content config:(
    CTFrameParserConfig*)config {
    // 创建CTFramesetterRef实例
    CTFramesetterRef framesetter = CTFramesetterCreateWithAttributedString((
        CFAttributedStringRef)content);

    // 获得要缓制的区域的高度
    CGSize restrictSize = CGSizeMake(config.width, CGFLOAT_MAX);
    CGSize coreTextSize = CTFramesetterSuggestFrameSizeWithConstraints(framesetter,
        CFRangeMake(0,0), nil, restrictSize, nil);
    CGFloat textHeight = coreTextSize.height;

    // 生成CTFrameRef实例
    CTFrameRef frame = [self createFrameWithFramesetter:framesetter config:config
        height:textHeight];

    // 将生成好的CTFrameRef实例和计算好的缓制高度保存到CoreTextData实例中，
    // 最后返回CoreTextData实例
    CoreTextData *data = [[CoreTextData alloc] init];
    data.ctFrame = frame;
    data.height = textHeight;

    // 释放内存
    CFRelease(frame);
    CFRelease(framesetter);
    return data;
}

// 方法六
+ (CTFrameRef)createFrameWithFramesetter:(CTFramesetterRef)framesetter
                                  config:(CTFrameParserConfig *)config
                                  height:(CGFloat)height {

    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddRect(path, NULL, CGRectMake(0, 0, config.width, height));
```

```
    CTFrameRef frame = CTFramesetterCreateFrame(framesetter, CFRangeMake(0, 0), path
        , NULL);
    CFRelease(path);
    return frame;
}
```

以上代码主要由 6 个子方法构成：

- 方法一用于提供对外的接口，调用方法二实现从一个 JSON 的模板文件中读取内容，然后调用方法五生成 CoreTextData。
- 方法二读取 JSON 文件内容，并且调用方法三获得从 NSDictionary 到 NSAttributedString 的转换结果。
- 方法三将 NSDictionary 内容转换为 NSAttributedString。
- 方法四提供将 NSString 转换为 UIColor 的功能。
- 方法五接受一个 NSAttributedString 和一个 config 参数，将 NSAttributedString 转换成 CoreTextData 返回。
- 方法六是方法五的一个辅助函数，供方法五调用。

然后我们将 ViewController 中的调用代码做一下更改，使其从模板文件中加载内容。代码如下所示：

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    CTFrameParserConfig *config = [[CTFrameParserConfig alloc] init];
    config.width = self.ctView.width;
    NSString *path = [[NSBundle mainBundle] pathForResource:@"content" ofType:@"json
        "];
    CoreTextData *data = [CTFrameParser parseTemplateFile:path config:config];
    self.ctView.data = data;
    self.ctView.height = data.height;
    self.ctView.backgroundColor = [UIColor whiteColor];
}

@end
```

最后运行得到的结果如下图所示，可以看到，通过一个简单的模板文件，我们已经可以很方便地定义排版的配置信息了。

说明：读者可以在示例工程中使用 git checkout json_template，查看可以运行的示例代码。

支持图文混排的排版引擎

改造模板文件

下面我们来进一步改造，让排版引擎支持对于图片的排版。在上一小节中，我们在设置模板文件的时候，就专门在模板文件里面留了一个名为 type 的字段，用于表示内容的类型。之前的 type 的值都是 txt，这次，我们增加一个值为 img 的值，用于表示图片。

我们将上一节的 content.json 文件修改为如下内容，增加了两个 type 值为 img 的配置项。由于是图片的配置项，所以我们不需要设置颜色、字号这些图片不具有的属性，但是，我们另外增加了三个图片的配置属性：

1. 一个名为 width 的属性，用于设置图片显示的宽度。
2. 一个名为 height 的属性，用于设置图片显示的高度。
3. 一个名为 name 的属性，用于设置图片的资源名。

```
[ {
    "type" : "img",
```

```
    "width" : 200,
    "height" : 108,
    "name" : "coretext-image-1.jpg"
  },
  { "color" : "blue",
    "content" : "更进一步地，实际工作中，我们更希望通过一个排版文件，来设置需要排版
        的文字的 ",
    "size" : 16,
    "type" : "txt"
  },
  { "color" : "red",
    "content" : "内容、颜色、字体",
    "size" : 22,
    "type" : "txt"
  },
  { "color" : "black",
    "content" : "大小等信息。\n",
    "size" : 16,
    "type" : "txt"
  },
  {
    "type" : "img",
    "width" : 200,
    "height" : 130,
    "name" : "coretext-image-2.jpg"
  },
  { "color" : "default",
    "content" : "我在开发猿题库应用时，自己定义了一个基于UBB的排版模板，但是实现该排
        版文件的解析器要花费大量的篇幅，考虑到这并不是本章的重点，所以我们以一个较
        简单的排版文件来讲解其思想。",
    "type" : "txt"
  }
]
```

按理说，图片本身的内容信息中，是包含宽度和高度信息的，为什么我们要在这里指定图片的宽高呢？这主要是因为，在真实的开发中，应用的模板和图片通常是通过服务器获取的，模板是纯文本的内容，获取速度比图片快很多，图片不但获取速度慢，而且为了省流量，通常的做法是直到需要显示图片的时候，再加载图片内容。

如果我们不将图片的宽度和高度信息设置在模板里面，那么 CoreText 在排版的时候就无法知道绘制所需要的高度，我们就无法设置 CoreTextData 类中的 height 信息，没有高度信息，就会对 UITabView 一类的控件排版造成影响。所以，除非你的应用图片能够保证在绘制前全部在本地，否则就应该提前提供图片宽度和高度信息。

在完成模板文件修改后，我们选取两张测试用的图片，分别将其命名为 coretext-image-

1.jpg 和 coretext-image-2.jpg（和模板中的值一致），将其拖动增加到工程中。向 Xcode 工程增加图片资源是基础知识，在此就不详细介绍过程了。

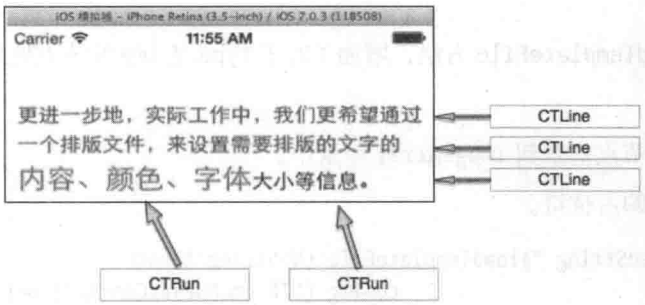
CTLine 与 CTRun

接下来我们需要改造的是 CTFrameParser 类，让解析模板文件的方法支持 type 为 img 的配置。

在改造前，我们先来了解一下 CTFrame 内部的组成。通过之前的例子，我们可以看到，我们首先通过 NSAttributeString 和配置信息创建 CTFrameSetter，然后，再通过 CTFrameSetter 来创建 CTFrame。

在 CTFrame 内部，是由多个 CTLine 来组成的，每个 CTLine 代表一行，每个 CTLine 又是由多个 CTRun 来组成，每个 CTRun 代表一组显示风格一致的文本。我们不用手工管理 CTLine 和 CTRun 的创建过程。

下图是一个 CTLine 和 CTRun 的示意图，可以看到，第三行的 CTLine 是由两个 CTRun 构成的，第一个 CTRun 为红色大字号的左边部分，第二个 CTRun 为右边字体较小的部分。



虽然我们不用管理 CTRun 的创建过程，但是我们可以设置某一个具体的 CTRun 的 CTRunDelegate 来指定该文本在绘制时的高度、宽度、排列对齐方式等信息。

对于图片的排版，其实 CoreText 本质上是不直接支持的，但是，我们可以在要显示文本的地方，用一个特殊的空白字符代替，同时设置该字体的 CTRunDelegate 信息为要显示的图片的宽度和高度信息，这样最后生成的 CTFrame 实例，就会在绘制时将图片的位置预留出来。

因为我们的 CTDisplayView 的绘制代码是在 drawRect 里面的，所以我们可以方便地把需要绘制的图片，用 CGContextDrawImage 方法直接绘制出来就行了。

改造模板解析类

在了解了以上原理后，我们就可以开始进行改造了。

我们需要做的工作包括：

1. 改造CTFrameParser的parseTemplateFile:(NSString *)path config:(CTFrameParserConfig*)config; 方法，使其支持对 type 为 img 的节点解析。并且对 type 为 img 的节点，设置其 CTRunDelegate 信息，使其在绘制时，为图片预留相应的空白位置。
2. 改造 CoreTextData 类，增加图片相关的信息，并且增加计算图片绘制区域的逻辑。
3. 改造 CTDisplayView 类，增加绘制图片相关的逻辑。

首先介绍对于 CTFrameParser 的改造。我们修改了 parseTemplateFile 方法，增加了一个名为 imageArray 的参数来保存解析时的图片信息。

```
+ (CoreTextData *)parseTemplateFile:(NSString *)path config:(CTFrameParserConfig*)
    config {
    NSMutableArray *imageArray = [NSMutableArray array];
    NSAttributedString *content = [self loadTemplateFile:path config:config
        imageArray:imageArray];
    CoreTextData *data = [self parseAttributedContent:content config:config];
    data.imageArray = imageArray;
    return data;
}
```

接着我们修改 loadTemplateFile 方法，增加了对于 type 是 img 的节点处理逻辑，该逻辑主要做两件事情：

1. 保存当前图片节点信息到 imageArray 变量中。
2. 新建一个空白的占位符。

```
+ (NSAttributedString *)loadTemplateFile:(NSString *)path
                                  config:(CTFrameParserConfig*)config
                              imageArray:(NSMutableArray *)imageArray {
    NSData *data = [NSData dataWithContentsOfFile:path];
    NSMutableAttributedString *result = [[NSMutableAttributedString alloc] init];
    if (data) {
        NSArray *array =
            [NSJSONSerialization JSONObjectWithData:data
                                            options:NSJSONReadingAllowFragments
                                              error:nil];
        if ([array isKindOfClass:[NSArray class]]) {
            for (NSDictionary *dict in array) {
                NSString *type = dict[@"type"];
                if ([type isEqualToString:@"txt"]) {
                    NSAttributedString *as =
                        [self parseAttributedContentFromNSDictionary:dict
                                                              config:config];
                    [result appendAttributedString:as];
```

```
            } else if ([type isEqualToString:@"img"]) {
                // 创建 CoreTextImageData
                CoreTextImageData *imageData = [[CoreTextImageData alloc] init];
                imageData.name = dict[@"name"];
                imageData.position = [result length];
                [imageArray addObject:imageData];
                // 创建空白占位符，并且设置它的CTRunDelegate信息
                NSAttributedString *as =
                    [self parseImageDataFromNSDictionary:dict
                                                  config:config];
                [result appendAttributedString:as];
            }
        }
    }
    return result;
}
```

最后我们新建一个最关键的方法：parseImageDataFromNSDictionary，生成图片空白的占位符，并且设置其 CTRunDelegate 信息，其代码如下：

```
static CGFloat ascentCallback(void *ref){
    return [(NSNumber*)[(__bridge NSDictionary*)ref objectForKey:@"height"]
        floatValue];
}

static CGFloat descentCallback(void *ref){
    return 0;
}

static CGFloat widthCallback(void* ref){
    return [(NSNumber*)[(__bridge NSDictionary*)ref objectForKey:@"width"]
        floatValue];
}

+ (NSAttributedString *)parseImageDataFromNSDictionary:(NSDictionary *)dict
                                                config:(CTFrameParserConfig*)config
                                                    {
    CTRunDelegateCallbacks callbacks;
    memset(&callbacks, 0, sizeof(CTRunDelegateCallbacks));
    callbacks.version = kCTRunDelegateVersion1;
    callbacks.getAscent = ascentCallback;
    callbacks.getDescent = descentCallback;
    callbacks.getWidth = widthCallback;
    CTRunDelegateRef delegate = CTRunDelegateCreate(&callbacks, (__bridge void *)(
        dict));
```

```
    // 使用0xFFFC作为空白的占位符
    unichar objectReplacementChar = 0xFFFC;
    NSString * content = [NSString stringWithCharacters:&objectReplacementChar
        length:1];
    NSDictionary * attributes = [self attributesWithConfig:config];
    NSMutableAttributedString * space =
        [[NSMutableAttributedString alloc] initWithString:content
                                                attributes:attributes];
    CFAttributedStringSetAttribute((CFMutableAttributedStringRef)space,
            CFRangeMake(0, 1), kCTRunDelegateAttributeName, delegate);
    CFRelease(delegate);
    return space;
}
```

接着我们对 CoreTextData 进行改造，增加了 imageArray 成员变量，用于保存图片绘制时所需的信息。

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

@interface CoreTextData : NSObject

@property (assign, nonatomic) CTFrameRef ctFrame;
@property (assign, nonatomic) CGFloat height;
// 新增加的成员
@property (strong, nonatomic) NSArray * imageArray;

@end
```

在设置 imageArray 成员时，我们还会调一个新创建的 fillImagePosition 方法，用于找到每张图片在绘制时的位置。

```
- (void)setImageArray:(NSArray *)imageArray {
    _imageArray = imageArray;
    [self fillImagePosition];
}

- (void)fillImagePosition {
    if (self.imageArray.count == 0) {
        return;
    }
    NSArray *lines = (NSArray *)CTFrameGetLines(self.ctFrame);
    int lineCount = [lines count];
    CGPoint lineOrigins[lineCount];
    CTFrameGetLineOrigins(self.ctFrame, CFRangeMake(0, 0), lineOrigins);

    int imgIndex = 0;
```

```
CoreTextImageData * imageData = self.imageArray[0];

for (int i = 0; i < lineCount; ++i) {
    if (imageData == nil) {
        break;
    }
    CTLineRef line = (__bridge CTLineRef)lines[i];
    NSArray * runObjArray = (NSArray *)CTLineGetGlyphRuns(line);
    for (id runObj in runObjArray) {
        CTRunRef run = (__bridge CTRunRef)runObj;
        NSDictionary *runAttributes = (NSDictionary *)CTRunGetAttributes(run);
        CTRunDelegateRef delegate = (__bridge CTRunDelegateRef)[runAttributes
            valueForKey:(id)kCTRunDelegateAttributeName];
        if (delegate == nil) {
            continue;
        }

        NSDictionary * metaDic = CTRunDelegateGetRefCon(delegate);
        if (![metaDic isKindOfClass:[NSDictionary class]]) {
            continue;
        }

        CGRect runBounds;
        CGFloat ascent;
        CGFloat descent;
        runBounds.size.width = CTRunGetTypographicBounds(run, CFRangeMake(0, 0),
             &ascent, &descent, NULL);
        runBounds.size.height = ascent + descent;

        CGFloat xOffset = CTLineGetOffsetForStringIndex(line,
            CTRunGetStringRange(run).location, NULL);
        runBounds.origin.x = lineOrigins[i].x + xOffset;
        runBounds.origin.y = lineOrigins[i].y;
        runBounds.origin.y -= descent;

        CGPathRef pathRef = CTFrameGetPath(self.ctFrame);
        CGRect colRect = CGPathGetBoundingBox(pathRef);

        CGRect delegateBounds = CGRectOffset(runBounds, colRect.origin.x,
            colRect.origin.y);

        imageData.imagePosition = delegateBounds;
        imgIndex++;
        if (imgIndex == self.imageArray.count) {
            imageData = nil;
            break;
```

```
            } else {
                imageData = self.imageArray[imgIndex];
            }
        }
    }
}
```

最后代码运行后的效果如下所示。[4]

添加对图片的点击支持

实现方式

为了实现对图片的点击支持，我们需要给 CTDisplayView 类增加用户点击操作的检测函数，在检测函数中，判断当前用户点击的区域是否在图片上，如果在图片上，则触发点击图片的逻辑。苹果提供的 UITapGestureRecognizer 可以很好地满足我们的要求，所以我们这里用

[4] 读者可以将示例工程用 git checkout image_support 切换到当前章节状态，查看相关代码逻辑。

它来检测用户的点击操作。

我们这里实现的是点击图片后，先用 NSLog 打印出一行日志。实际应用中，读者可以根据业务需求自行调整点击后的效果。

我们先为 CTDisplayView 类增加 UITapGestureRecognizer：

```
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        [self setupEvents];
    }
    return self;
}

- (void)setupEvents {
    UIGestureRecognizer * tapRecognizer =
        [[UITapGestureRecognizer alloc] initWithTarget:self
                                                action:@selector(
                                                    userTapGestureDetected:)];
    tapRecognizer.delegate = self;
    [self addGestureRecognizer:tapRecognizer];
    self.userInteractionEnabled = YES;
}
```

然后增加 UITapGestureRecognizer 的回调函数：

```
- (void)userTapGestureDetected:(UIGestureRecognizer *)recognizer {
    CGPoint point = [recognizer locationInView:self];
    for (CoreTextImageData * imageData in self.data.imageArray) {
        // 翻转坐标系，因为imageData中的坐标是CoreText的坐标系
        CGRect imageRect = imageData.imagePosition;
        CGPoint imagePosition = imageRect.origin;
        imagePosition.y = self.bounds.size.height - imageRect.origin.y - imageRect.
            size.height;
        CGRect rect = CGRectMake(imagePosition.x, imagePosition.y, imageRect.size.
            width, imageRect.size.height);
        // 检测点击位置 Point 是否在rect之内
        if (CGRectContainsPoint(rect, point)) {
            // 在这里处理点击后的逻辑
            NSLog(@"bingo");
            break;
        }
    }
}
```

事件处理

在界面上，CTDisplayView 通常在 UIView 的树形层级结构中，一个 UIView 可能是最外层 View Controller 的 View 的“孩子的孩子的孩子”，如下图所示。在这种多级层次结构中，很难通过 delegate 模式将图片点击的事件一层一层往外层传递，所以最好使用 NSNotification 来处理图片点击事件。

```
*** Printing out all the subviews of MKMapView ***
   [0]: class: 'UIView'
      [0]: class: 'MKScrollView'
         [0]: class: 'MKMapTileView'
            [0]: class: 'MKOverlayContainerView'
               [0]: class: 'MKOverlayClusterView'
                  [0]: class: 'MKPolylineView'
         [1]: class: 'MKAnnotationContainerView'
            [0]: class: 'MKUserLocationView'
            [1]: class: 'MKAnnotationView'
               [0]: class: 'UIImageView'
            [2]: class: 'MKPinAnnotationView'
            [3]: class: 'GSPinAnnotationView'
            [4]: class: 'MKPinAnnotationView'
            [5]: class: 'MKPinAnnotationView'
            [6]: class: 'MKPinAnnotationView'
               [0]: class: 'UICalloutView'
                  [0]: class: 'UIImageView'
                  [1]: class: 'UIImageView'
                  [2]: class: 'UIImageView'
                  [3]: class: 'UIImageView'
                  [4]: class: 'UIImageView'
                  [5]: class: 'UIImageView'
                  [6]: class: 'UILabel'
                  [7]: class: 'UILabel'
                  [8]: class: 'UILabel'
   [1]: class: 'UIImageView'
```

在 Demo 中，我们在最外层的 View Controller 中监听图片点击的通知，当收到通知后，进入到一个新的界面来显示图片内容。[5]

添加对链接的点击支持

修改模板文件

我们修改模板文件，增加一个名为“link”的类型，用于表示链接内容。代码如下所示：

```
[
  { "color" : "default",
    "content" : "这在这里尝试放一个参考链接：",
```

5 读者可以将 demo 工程切换到 image_click 分支，查看示例代码。

```
    "type" : "txt"
  },
  { "color" : "blue",
    "content" : "链接文字",
    "url" : "http://blog.devtang.com",
    "type" : "link"
  },
  { "color" : "default",
    "content" : "大家可以尝试点击一下",
    "type" : "txt"
  }
]
```

解析模板中的链接信息

我们首先增加一个 CoreTextLinkData 类，用于记录解析 JSON 文件时的链接信息：

```
@interface CoreTextLinkData : NSObject

@property (strong, nonatomic) NSString * title;
@property (strong, nonatomic) NSString * url;
@property (assign, nonatomic) NSRange range;

@end
```

然后我们修改 CTFrameParser 类，增加解析链接的逻辑：

```
+ (NSAttributedString *)loadTemplateFile:(NSString *)path
                                  config:(CTFrameParserConfig*)config
                              imageArray:(NSMutableArray *)imageArray
                               linkArray:(NSMutableArray *)linkArray {
    NSData *data = [NSData dataWithContentsOfFile:path];
    NSMutableAttributedString *result = [[NSMutableAttributedString alloc] init];
    if (data) {
        NSArray *array =
            [NSJSONSerialization JSONObjectWithData:data
                                            options:NSJSONReadingAllowFragments
                                              error:nil];
        if ([array isKindOfClass:[NSArray class]]) {
            for (NSDictionary *dict in array) {
                NSString *type = dict[@"type"];
                if ([type isEqualToString:@"txt"]) {
                    // 省略
                } else if ([type isEqualToString:@"img"]) {
                    // 省略
                } else if ([type isEqualToString:@"link"]) {
                    NSUInteger startPos = result.length;
```

```
                NSAttributedString *as =
                    [self parseAttributedContentFromNSDictionary:dict
                                                          config:config];
                [result appendAttributedString:as];
                // 创建 CoreTextLinkData
                NSUInteger length = result.length - startPos;
                NSRange linkRange = NSMakeRange(startPos, length);
                CoreTextLinkData *linkData = [[CoreTextLinkData alloc] init];
                linkData.title = dict[@"content"];
                linkData.url = dict[@"url"];
                linkData.range = linkRange;
                [linkArray addObject:linkData];
            }
        }
    }
    return result;
}
```

接下来，我们增加一个 Utils 类来专门检测用户的点击点是否在链接上。主要的方法是使用 CTLineGetStringIndexForPosition 函数来获得用户点击的位置与 NSAttributedString 字符串上的位置的对应关系。这样就知道是点击的哪个字符了。然后判断该字符串是否在链接上即可。该 Util 的实现逻辑如下：

```
// 检测点击位置是否在链接上
+ (CoreTextLinkData *)touchLinkInView:(UIView *)view atPoint:(CGPoint)point data:(
    CoreTextData *)data {
    CTFrameRef textFrame = data.ctFrame;
    CFArrayRef lines = CTFrameGetLines(textFrame);
    if (!lines) return nil;
    CFIndex count = CFArrayGetCount(lines);
    CoreTextLinkData *foundLink = nil;

    // 获得每一行的origin坐标
    CGPoint origins[count];
    CTFrameGetLineOrigins(textFrame, CFRangeMake(0,0), origins);

    // 翻转坐标系
    CGAffineTransform transform =  CGAffineTransformMakeTranslation(0, view.bounds.
        size.height);
    transform = CGAffineTransformScale(transform, 1.f, -1.f);

    for (int i = 0; i < count; i++) {
        CGPoint linePoint = origins[i];
        CTLineRef line = CFArrayGetValueAtIndex(lines, i);
        // 获得每一行的CGRect信息
```

```
            CGRect flippedRect = [self getLineBounds:line point:linePoint];
            CGRect rect = CGRectApplyAffineTransform(flippedRect, transform);

            if (CGRectContainsPoint(rect, point)) {
                // 将点击的坐标转换成相对于当前行的坐标
                CGPoint relativePoint = CGPointMake(point.x-CGRectGetMinX(rect),
                                                    point.y-CGRectGetMinY(rect));
                // 获得当前点击坐标对应的字符串偏移
                CFIndex idx = CTLineGetStringIndexForPosition(line, relativePoint);
                // 判断这个偏移是否在我们的链接列表中
                foundLink = [self linkAtIndex:idx linkArray:data.linkArray];
                return foundLink;
            }
        }
        return nil;
    }

    + (CGRect)getLineBounds:(CTLineRef)line point:(CGPoint)point {
        CGFloat ascent = 0.0f;
        CGFloat descent = 0.0f;
        CGFloat leading = 0.0f;
        CGFloat width = (CGFloat)CTLineGetTypographicBounds(line, &ascent, &descent, &
            leading);
        CGFloat height = ascent + descent;
        return CGRectMake(point.x, point.y - descent, width, height);
    }

    + (CoreTextLinkData *)linkAtIndex:(CFIndex)i linkArray:(NSArray *)linkArray {
        CoreTextLinkData *link = nil;
        for (CoreTextLinkData *data in linkArray) {
            if (NSLocationInRange(i, data.range)) {
                link = data;
                break;
            }
        }
        return link;
    }
```

最后改造一下 CTDisplayView，使其在检测到用户点击后，调用上面的 Util 方法即可。我们这里实现的是点击链接后，先用 NSLog 打印出一行日志。实际应用中，读者可以根据业务需求自行调整点击后的效果。[6]

```
    - (void)userTapGestureDetected:(UIGestureRecognizer *)recognizer {
```

[6] 在 Demo 中工程中，我们实现了点击链接跳转到一个新的界面，然后用 UIWebView 来显示链接内容的逻辑。读者可以将 Demo 工程切换到 link_click 分支，查看示例代码。

```
    CGPoint point = [recognizer locationInView:self];
    // 此处省略上一节中介绍的，对图片点击检测的逻辑

    CoreTextLinkData *linkData = [CoreTextUtils touchLinkInView:self atPoint:point
        data:self.data];
    if (linkData) {
        NSLog(@"hint link!");
        return;
    }
}
```

第18章 实战技巧

本章将介绍一些我在 iOS 开发中总结的实战技巧。

18.1 App Store 与审核

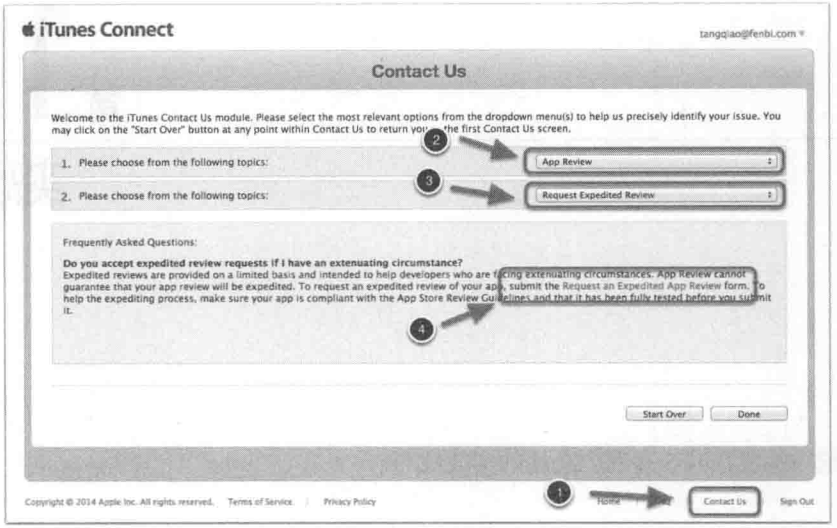
18.1.1 撤销正在审核的应用

正在审核（处于 In Review 状态）的应用其实也是能够撤销的，单击下图右上角灰色的“Reject This Binary”按钮即可。

18.1.2 申请加急审核

申请加急审核的步骤如下：

1. 访问 iTunes Connect 网站：https://itunesconnect.apple.com。
2. 单击网站底部的“Contact Us”按钮。
3. 在问题 1 中选择“App Review”。
4. 在问题 2 中选择“Request Expedited Review”。
5. 单击“Request an Expedited App Review”按钮即可填写加急审核的申请表，如下图所示。



注意，最好使用英文填写加急审核的申请表。申请加急审核时，最容易通过的理由是严重的崩溃 Bug（在理由一栏选择："Critical Bug Fix"），在原因处详细描述该 Bug 的重现步骤，则很容易使申请通过。加急审核申请的频率也对申请结果有一定影响，所以最好不要滥用，多次欺骗审核员可能造成真正有需要的时候申请被拒。

18.1.3 应用在市场的名字

不要取特别长的应用市场名字。虽然 App Store 上有很多特别长的名字通过了审核，但是由于苹果的审核员尺度并不完全一致，所以很可能造成你下次审核的时候通不过。

18.1.4 测试设备数的限制

测试设备 100 个数目限制的详细规则是什么？我总结出来的规则如下：

- 每一个开发者在一个 membership year（从交费日开始算起，一年之内算一个 membership year）中，只能有 100 个增加设备的名额。如果你增加一个设备，之后又将该设备删除，并不会将用掉的名额恢复。
- 开发者在每个 membership year 开始的时候，Team Agent 和 Admin 角色可以选择删掉一些设备来恢复资格，也可以清空所有设备来恢复到最多 100 次设备的名额。这个操作在 Team Agent 和 Admin 在一次新的 membership year 开始后即可使用。在使用时需要注意，先将需要删除的设备删掉，然后才能添加需要新增的设备。一旦开始增加新设备，删除设备以恢复名额的功能将不再可用。

- 在以后整个 membership year 中，删除设备不会增加新的名额。

直接看规则比较晦涩，举个例子：

- 假如第一年，你增加了 70 个设备，同时删除了 10 个设备，这个时候，虽然你的设备数是 60，但是可用的增加测试机的名额却只有 30 个了。
- 到了第二年，你延续了开发者身份，在你第一次登录进去后，你可以看到你的可用设备恢复成 100–60=40 个了。这个时候，你可以选择删除一些设备，例如你又删除了 20 个设备，这样你的名额数变成 60 个。之后你增加了 1 个设备，因为你选择了增加新设备，苹果认为你已经放弃了删除设备以恢复名额的机会，这样，你的名额就固定成 59 个。以后删除设备都不会增加新名额了，直到你的下一个 membership year 开始时才又会有这样的机会来删除设备释放名额。

我的设备数达到上限，我又急需增加新设备怎么办呢？对此你可以给苹果的技术客服发邮件要求他们帮助我们删除所有设备，并且恢复到增加 100 个测试设备的名额。具体做法是访问https://developer.apple.com/contact/，单击“Program Benefits”按钮，然后在新出来的提交界面中将需求填上。之后苹果会发邮件告诉你处理结果，你可能需要打电话过去和他们沟通一些细节。在沟通完成后，苹果就可以立即帮助你把状态修改到“可删除设备来增加测试设备名额”。这样，你就可以选择性地删除一些不需要的设备来释放一些名额了。

18.1.5 如何将应用下架

最简单的办法，是将应用的上架时间改成未来的一个时间，这样就会自动把已上架的应用下架。但是需要注意的是，由于 App Store 更新比较慢，整个下架操作完全生效可能会需要等待数小时之久。与此对应的是，审核通过的上架操作也取决于 App Store 的更新速度，有些极端情况下应用要花费 24 小时才能完成上架操作。

18.1.6 如何举报别的应用侵权

以英文邮件的方式与相关小组联系，这个小组会跟侵权的一方联系，邮件地址为：AppStoreNotices@apple.com。另外，也可以通过以下的渠道与苹果的相关小组联系：http://www.apple.com/legal/internet-services/itunes/appstorenotices/#/contacts。举报时需要提供侵权的相关证明文件，苹果会联系另一方提供自证材料，最终通过双方的材料来决定举报是否有效，对于有效的举报，侵权的应用会被苹果下架。

苹果非常重视保护创新和知识产权，对侵权者是不讲情面的，所以大家遇到侵权的行为，可以积极通过正常渠道举报，维护自己的权益。在国内也有许多成功举报的案例，如大众点评举报食神摇摇，Secret 举报无密，都使得侵权应用长时间下架。

18.1.7 iTunes Connect 后台操作出错

遇到 iTunes Connect 后台操作的任何问题，先换成 Safari 浏览器试试。苹果的 iTunes Connect 后台出现过多次在 Chrome 浏览器下面的 Bug。我用 Chrome 浏览器遇到过多次无法添加测试设备、无法上传 Push 证书等问题，换成 Safari 浏览器就好了。

如果你的应用处于“Upload Received”状态长达 24 小时，可以试试编辑一下应用截图介绍然后再保存，这是苹果的一个 Bug。

18.1.8 Metadata Reject

处于“Metadata Reject”状态的应用，通常不需要重新上传应用，只需要把相应的设置修改好后，在“Resolution Center”给审核员回复说明一下即可。如果重新上传应用，就会需要重新排队等待审核了。

18.2 开发技巧

18.2.1 UILabel 内容模糊

在非 Retina 的 iPad mini 的屏幕上，一个 UILabel 的 frame 的 origin 值如果有小数位数（例如 0.5），就会造成显示模糊。所以最好使用整数值的 origin 坐标。

18.2.2 收起键盘

在 UIViewController 中收起键盘，除了调用相应控件的 `resignFirstResponder` 方法外，还有另外三种办法：

1. 重载 UIViewController 中的 `touchesBegin` 方法，然后在里面执行 `[self.view endEditing:YES];`，这样单击 UIViewController 的任意地方，就可以收起键盘。
2. 直接执行 `[[UIApplication sharedApplication] sendAction:@selector(resignFirstResponder) to:nil from:nil forEvent:nil];`，用于在获得当前 UIViewController 比较困难的时候用。
3. 直接执行 `[[[UIApplication sharedApplication] keyWindow] endEditing:YES]`。

18.2.3 NSJSONSerialization 比 NSKeyedArchiver 更好

在选择持久化方案时，系统提供的 NSJSONSerialization 比 NSKeyedArchiver 在效率和体积上都更优。经过我的简单测试，NSJSONSerialization 比 NSKeyedArchiver 快了 7 倍，而且序列化之后的体积是 NSKeyedArchiver 的一半。

网上也有更详细的测试证明了该说法：https://github.com/randomsequence/NSSerialisationTests。

18.2.4 设置应用内的系统控件语言

在 iOS 应用中，有时候会需要调用系统的一些 UI 控件，例如：

1. 在 UIWebView 中长按会弹出系统的上下文菜单。
2. 在 UIImagePickerController 中会使用系统的照相机界面。
3. 在编译状态下的 UITableViewCell，处于待删除状态时，会有一个系统的删除按钮。

以上这些 UI 控件中，其显示的语言并不是和你当前手机的系统语言一致的，而是根据你的应用内部的语言设置来显示的。结果就是，如果你没有设置恰当的话，你的中文应用可能会出现一些英文的控件文字。

例如下图中，某中文应用在其软件界面中长按，就会出如下的菜单，可以看到，这个菜单的文字全是英文的。

而正常的菜单应该是中文的，下图是在新浪微博的正文中长按之后的效果。

如何解决这个问题呢？方法很简单，用 vim 直接打开工程的 Info.plist 文件，在文件中增加如下内容即可：

```
<key>CFBundleLocalizations</key>
<array>
        <string>zh_CN</string>
        <string>en</string>
</array>
```

巧用截屏功能

iOS 的截屏功能可以将当前界面中的 UI 元素保存成 UIImage。对于 iOS7 以后的系统，可以通过系统提供的 API- (UIView *)snapshotViewAfterScreenUpdates:(BOOL)afterUpdates 来实现截屏的功能。对于 iOS7 以前的系统，可以通过以下代码实现截屏的功能：

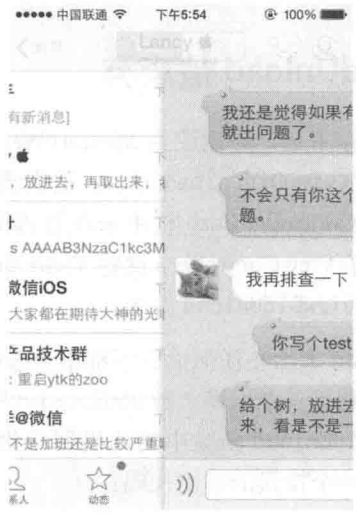
```
+ (UIImage *)captureImageFromView:(UIView *)view {
    CGRect screenRect = [view bounds];
    UIGraphicsBeginImageContext(screenRect.size);
    CGContextRef ctx = UIGraphicsGetCurrentContext();
    [view.layer renderInContext:ctx];
    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return image;
}
```

截屏功能看似无用，但是在一些情况下可以帮助我们实现一些特殊效果。下面我们来看看利用截屏功能可以做哪些事情。

用截屏功能来实现侧滑返回效果

苹果从 iOS7 开始提供了系统的侧滑返回功能，不过需要用户从左侧的的屏幕外边缘开始往右边滑动才能触发，并且为了兼容旧的 iOS 版本，很多应用都自己实现了侧滑返回功能。

下图是腾讯 QQ 自己实现的侧滑返回效果图，在侧滑到一半时，整个当前界面被移动到了右半部分，同时在左半部分以半透明的方式露出了上一个界面。



由于 ViewController 并不支持将自己的 view 设置成透明的，无法直接实现看到上一个 ViewController 的效果，所以我们需要自己实现这种半透明的“透视”功能，具体来说就是使用截屏。

为了使用截屏功能达到这种效果，我们在 Navigation Controller 进入到一个新的 ViewController 前，先进行截屏操作，保存当前的界面效果，然后将截到的当前界面作为参数，传递给目标 ViewController 当作背景。

这样，平时这个背景我们用内容遮挡住，当用户用手指向右滑时，我们将整个界面右移，露出这个背景，于是就会像看到了上一个 ViewController 一样。

用截屏功能来实现半透明效果

截屏功能除了可以完成侧滑返回效果，也可以做很多其他效果。猿题库 iPad 版的试卷列表页面，其左边部分以半透明的方式显示了底部的界面，这其实就是用截屏来实现的。有道云笔记的 iPad 版，其编辑笔记界面的左右边缘也可以以半透明的方式看到上一个界面，这也是通过截屏功能来实现的。

如果不用截屏来实现上述效果，我们还有另外两种可选的技术方案：

1. 使用 UIWindow 来实现半透明效果。但是这种方案不够灵活，当界面复杂后，多个 UIWindow 的处理也会比较麻烦。
2. 将新的界面的 UIViewController 作为上一层 ViewController 的子 ViewController，这样新界面通过 addSubview 方法来添加到父 ViewController 的 view 中。但是这种方案使得 ViewController 之间的耦合性就比较高了，如果新的界面可能被很多 ViewController 显示，那么就会需要写大量的高耦合性的代码了。

18.2.5 为什么 viewDidUnload 被废弃

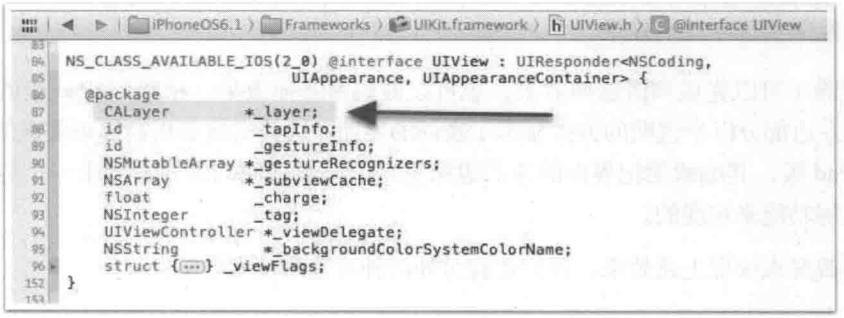
在 iOS4 和 iOS5 系统中，当内存不足，应用收到 MemoryWarning 时，系统会自动调用当前没在界面上的 ViewController 的 viewDidUnload 方法。通常情况下，这些未显示在界面上的 ViewController 是 UINavigationController Push 栈中未在栈顶的 ViewController，以及 UITabBarViewController 中未显示的子 ViewController。这些 ViewController 都会在 MemoryWarning 事件发生时，让系统自动调用 viewDidUnload 方法。

在 iOS6 中，由于 viewDidUnload 事件在任何情况下都不会被触发，所以苹果在文档中建议，应该将回收内存的相关操作移到另一个回调函数 didReceiveMemoryWarning 中。但是如果你以为仅仅是把以前写到 viewDidUnload 函数中的代码移动到 didReceiveMemoryWarning 函数中，那么你就错了。以下是一个错误的示例代码：

```
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    if([self isViewLoaded] && ![[self view] window]) {
        [self setView:nil];
    }
}
```

在 iOS6 以后的版本中不建议你将 view 置为 nil 的原因如下：

1. UIView 有一个 CALayer 的成员变量，CALayer 是具体用于将自己画到屏幕上的，如下图所示。



2. CALayer 是一个 bitmap 图像的容器类，当 UIView 调用自身的 drawRect 时，CALayer 才会创建这个 bitmap 图像类。
3. 具体占内存的其实是一个 bitmap 图像类，CALayer 只占 48Bytes, UIView 只占 96Bytes。而一个 iPad 的全屏 UIView 的 bitmap 类会占到 12MB 的大小！
4. 在 iOS6 中，当系统发出 MemoryWarning 时，系统会自动回收 bitmap 类，但是不回收 UIView 和 CALayer 类。这样既能回收大部分内存，又能在需要 bitmap 类时，通过调用 UIView 的 `drawRect:` 方法重建。

内存优化

苹果的操作系统对上面的内存回收还做了一个优化：

1. 当一段内存被分配时，它会被标记成“In use”，以防止被重复使用。当内存被释放时，这段内存会被标记成“Not in use”，这样，在有新的内存申请时，这块内存就可能被分配给其他变量。
2. CALayer 包括的具体的 bitmap 内容的私有成员变量类型为 CABackingStore（http://blog.spacemanlabs.com/2011/08/calayer-internals-contents/ ），当收到 MemoryWarning 时，CABackingStore 类型的内存区会被标记成 volatile 类型（这里的 volatile 和 C 及 Java 语言的 volatile 的意思不同），volatile 表示，这块内存可能再次被原变量使用。

这样，有了上面的优化后，当收到 MemoryWarning 时，虽然所有的 CALayer 所包含的 bitmap 内存都被标记成 volatile 了，但是只要这块内存没有被复用，当需要重建 bitmap 内存时，它就可以直接被复用，而避免了再次调用 UIView 的 `drawRect:` 方法。

所以，简单来说，对于 iOS6，你不需要做任何以前 `viewDidUnload` 的事情，更不需要把以前 `viewDidUnload` 的代码移动到 `didReceiveMemoryWarning` 方法中。

18.2.6 多人协作慎用 Storyboard

Storyboard 是苹果在 2011 年的 WWDC 中介绍的 Interface Builder 的功能。其基本想法是将原本的 xib 进行升级，引入一个容器用于管理多个 xib 文件，并且这个容器可以通过拖曳设置 xib 之间的界面跳转。这就是被苹果称做 Storyboard 的容器。

总体上来说，Storyboard 有以下好处：

1. 你可以从 Storyboard 中很方便地梳理出所有 ViewController 的界面间的调用关系。这一点对于新加入项目组的开发同事来说，比较友好。
2. 使用 Storyboard 时，也可以使用 Table ViewController 的 Static Cell 功能。对于开发一些 Cell 不多，但每个 Cell 都不一样的列表类设置界面会比较方便。

3. 通过实现- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender 方法，每个 ViewController 的跳转逻辑都聚集在一处，这方便我们统一管理界面跳转和传递数据。
4. Storyboard 可以方便将一些常用功能模块化和复用。例如 WWDC 2011 年介绍 Storyboard 的视频就将微博分享功能模块化成一个单独的 Storyboard。我在开发应用时，也将第三方注册登录模块做成一个单独的 Storyboard，便于以后复用。

在实际使用 Storyboard 时，会发现真正使用起来会有很多问题，我发现的问题有：

1. 首先它和 xib 一样，容易引起版本冲突。因为是它实际上的多个 xib 的集合，所以更容易让多人编辑产生冲突。苹果对 Storyboard 的设计也不好，很多时候你只要打开文件，即使什么都不做，这个文件也会被更改，所以冲突几乎是不可避免的，除非你不打开它。实在不小心打开看了，需要在提交前回退成服务器上的版本。
2. Storyboard 提供的 Static Cell 特性只适合于 UITableViewController 的子类。我很多时候的用法是一个 TableView 嵌套在另一个 UIView 中，Static Cell 就不能用了。
3. segue 的概念对于开发者来说并不省事。如果我们需要在程序中用代码触发一个 segue，那么就需要在另一个回调函数 prepareForSegue 处，用代码的方式设目标 View Controller 的参数信息。

所以如果读者坚持使用 Storyboard，首先必须做的是对 Storyboard 逻辑进行切分。我在实际使用时将应用界面按功能模块切分成了三个 Storyboard，但最后发现还是很容易出现多人同时修改的情况。

考虑到 Storyboard 带来的好处和坏处，我认为多人协作的团队应该慎用 Storyboard，以避免频繁的冲突处理。

真正遇到冲突时也不要着急，Storyboard 内部其实是 XML 格式，所以我们使用 vim 一类的文本编辑器将其打它，大多数情况下都能读懂冲突的原因，进而对冲突进行处理即可。

18.2.7 避免滥用 block

由于 block 定义方式简单，并且可以捕获上下文变量，所以合理地使用它，可以在很多开发场景下提高编程效率。

但是实际的使用经历告诉我们，block 经常被滥用。我在腾讯、百度及阿里的朋友都曾向我抱怨过团队内部有人滥用 block。其滥用主要体现在以下几点：

1. 对于 block 的内存管理不太熟悉，写出的代码有循环引用问题。这类人通常指望 Xcode 在编译时能够通过静态分析对循环引用提出警告。但是其实 Xcode 的静态分析只能发现非常少量的简单循环引用。

2. 对于 block 的生命期不太熟悉。这类人在使用 block 时不会去考虑 block 在执行时的线程和上下文变量可能的情况，造成程序出现问题时，调试变得异常困难。
3. 复杂的逻辑用多层 block 嵌套来实现，也给程序调试带来困难。

以下是一个滥用的具体例子，在该代码中，作者本意是想将 UITableView 做一层封装，以 block 的方式将接口暴露出来，他设计了两个 block：

1. 将 UITableView 的 Cell 创建用名为 ConfigureCellBlock 的 block 暴露出来。
2. 将 UITableView 的 Cell 点击用名为 SelectedCellBlock 的 block 暴露出来。

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

typedef void (^SelectedCellBlock)(UITableView *, NSIndexPath *);
typedef void (^ConfigureCellBlock)(UITableViewCell *, NSObject *);

@interface TreeTableViewAgent : NSObject <UITableViewDataSource, UITableViewDelegate
    >

@property (strong, nonatomic) NSMutableArray *treeCellDataArray;

@property (strong, nonatomic) ConfigureCellBlock configureCellBlock;
@property (strong, nonatomic) SelectedCellBlock selectedCellBlock;

- (id)initWithTableView:(UITableView *)tableView;

@end
```

看起来这么设计没什么问题，但是在实际使用时，这个 TreeTableViewAgent 类是被创建它的 UIViewController 所持有的，所以当两个 block 的代码中引用了 self，就会使得 TreeTableViewAgent 类引用到它所在的 UIViewController 类，从而形成循环引用，造成内存泄漏。

对于这种问题，需要将引用的一方变成 weak 的，从而避免循环引用，以下是具体使用时避免循环引用的例子：

```
__weak __typeof(self) weakSelf = self;
self.tableViewAgent.selectedCellBlock = ^(UITableView *tableView, NSIndexPath *
    indexPath) {
    [weakSelf someMethod];
};
```

而我认为，从软件架构层面来看，如果在使用 block 时需要时刻注意避免循环引用问题，那么还不如不使用 block。上面那个反例，用 delegate 来实现就要安全得多。

我们看到苹果在设计系统的 UITableView、UIActionSheet 和 UIAlertView 时，都用 delegate 而

不用 block，这也说明这些控件用 delegate 和 UIViewController 通讯是更加安全的。

而苹果在设计 UIView 的动画效果时，就提供了以下的 block 方式的接口，由于它是类方法，绝不会产生循环引用问题，所以这个时候使用 block 就显得更为合理。代码如下所示：

```
[UIView animateWithDuration:1.0 animations:^{
    // 动画效果
}];
```

18.2.8 合并工程文件的冲突

在多人协作开发的时候，iOS 项目的配置文件是最容易冲突的一个地方。工程配置文件的文件名通常是 project.pbxproj。

大部分工程文件的冲突都是由于大家同时添加了新的源码文件或资源文件，对于这种情况，我们只需要将这些改动合并即可。以下是一个例子，冲突之后从 git 给出的冲突信息来看，当前工作分支增加了一个名为 ApeDropDownRightArrow.png 的资源文件，而 master 分支增加了一个名为 ApeDropDownRightArrow.png 的资源文件。代码如下所示：

```
        451B0664174E1798002AF231 /* 1x */ = {
            isa = PBXGroup;
            children = (
<<<<<<< HEAD
                BD87C890196BE2C9007B9CC4 /* ApeDropDownRightArrow.png */,
=======
                BD714158196C1B9600DBF6C2 /* gDifficultyBar.pn */,
>>>>>>> master
```

对于上面这种冲突，我们只需要将内容简单合并即可，合并后的结果如下所示：

```
        451B0664174E1798002AF231 /* 1x */ = {
            isa = PBXGroup;
            children = (
                BD87C890196BE2C9007B9CC4 /* ApeDropDownRightArrow.png */,
                BD714158196C1B9600DBF6C2 /* gDifficultyBar.pn */,
```

对于一些特别复杂的工程设置修改，如果肉眼判断不出来冲突的内容应该如何合并，最后的办法是取其中一个分支的修改，然后把另一个分支所做的修改再重做一下。因为配置文件修改一般只是增加一些源文件或资源文件，所以重做一下相对来说也是可以接受的。

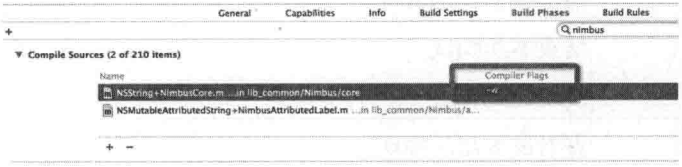
另外，有些时候还会出现合并的时候并没有报冲突，但是打开 Xcode 工程后编译失败的情况。对于这种情况要看具体的失败原因。如果报错的信息是一个类已存在，那么多是由于这个类在两个分支中同时添加了，解决办法是在工程文件中先删掉该文件，再添加即可。如果报错的信息是一个资源文件已存在，理由也同上，解决办法是把这个资源文件删掉再添加

一次。

18.2.9 忽略编译警告

由于 Xcode 和 LLVM 在不停地升级，有些时候我们使用的第三方的库会在新版的 Xcode 中产生一些编译的警告，对于我们确认没有问题的代码，我们可以加上-w 的编译参数，使得这些文件不产生编译警告的内容。

例如著名的开源库 Nimbus 为了兼容 iOS6 版本，使用了部分 iOS7 以后被废弃的 API，如果你的应用只支持 iOS7 以上版本，那么在编译时，相关 Nimbus 的代码会报警。对于这些源文件，我们加上-w 参数，则可以忽略掉相关的编译警告，如下图所示。



除了使用-w 禁止掉所有的编译警告外，我们也可以用-Wno-unused-variable 只禁掉未使用变量的编译警告。

18.3 Xcode 使用技巧

18.3.1 Xcode 快捷键

如果能慢慢熟悉一些快捷键，对于提高工作效率是有很大帮助的。虽然 Xcode 的设置页面有所有的快键键列表，但是估计没人能全部记住并且用上，我个人总结出来的常用的快捷键如下所示：

Cmd + Shift + O	快速查找类，通过这个可以快速跳转到指定类的源代码中
Ctrl + 6	列出当前文件中所有的方法，可以输入关键字来过滤。用它们可以快速定位想编辑的方法
Cmd + 1	切换成 Project Navigator (Cmd + 2 7 也可以做相应切换，不过不常用)
Cmd + Ctrl + Up	在.h 和.m 文件之间切换
Cmd + Enter	切换成 standard editor
Cmd + Opt + Enter	切换成 assistant editor
Cmd + Shift + Y	切换 Console View 的显示或隐藏

Cmd + 0	隐藏左边的导航 (Navigator) 区
Cmd + Opt + 0	隐藏右边的工具 (Utility) 区（可以直接把这个快捷键改成 Ctrl+O，这样用起来更顺手）
Cmd + Ctrl + Left/Right	到上/下一次编辑的位置，在两个编辑位置跳转的时候很方便
Cmd + Opt + J	跳转到文件过滤区
Cmd + Shift + F	在工程中查找
Cmd + R	运行，如果选上直接 kill 掉上次进程的话，每次直接一按就可以重新运行了
Cmd + B	编译工程
Cmd + Shift + K	清空编译好的文件
Cmd + .	结束本次调试
ESC	调出代码补全
Cmd + **单击**	查看该方法的实现
Opt + **单击**	查看该方法的文档
Cmd + T	新建一个 Tab 栏
Cmd + Shift + [	在 Tab 栏之间切换

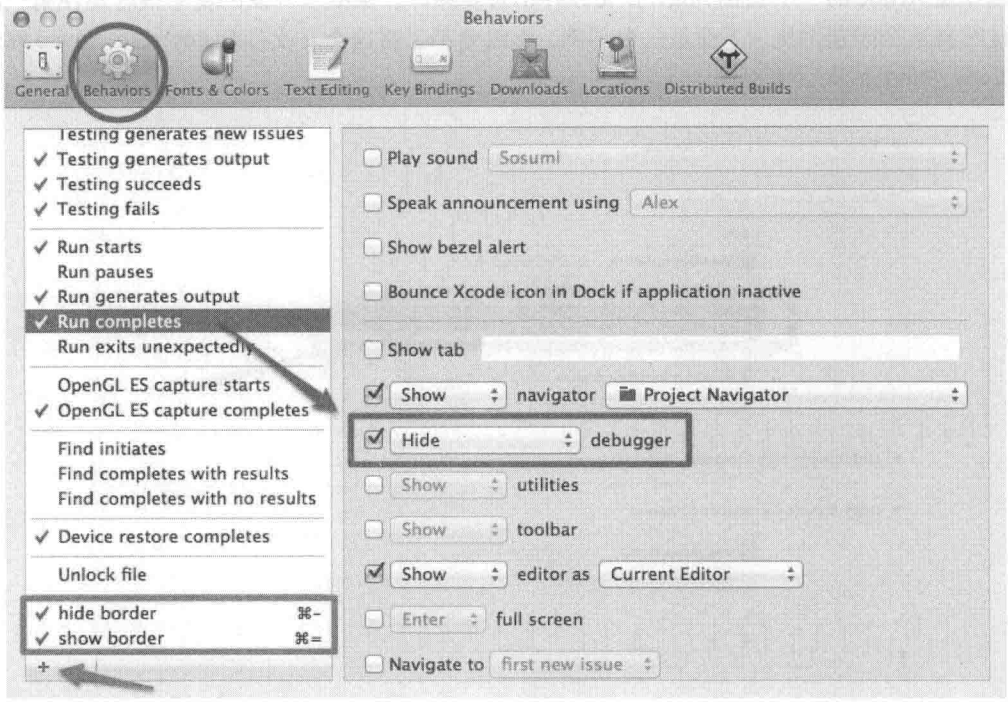
如果你实在不想记那么多，那也应该记住上面提到的 `Cmd + Shift + O` 和 `Ctrl + 6`，这是最能提高工作效率的两组快捷键。

Behaviors

Behaviors 是 Xcode 设置页面里的一栏，通过 `Cmd + ,` 可以调出设置页面看到它。在 Behaviors 里可以设置各种行为发生时界面应该做何种改变。

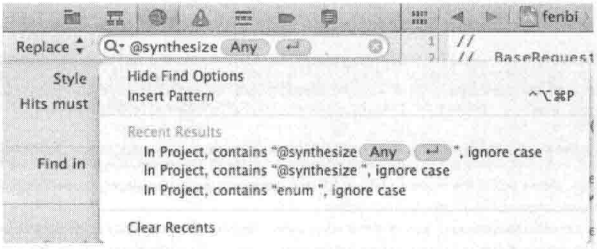
例如，如果我们要每次运行结束后就关闭 Console 界面，可以在 Behaviors 下面的“Run completes”一栏的右边，选中“Hide debugger”一项即可，如下图中上面方框所示。

另外，我们也可以单击左下角的“+”按钮来增加一些自定义的行为，例如我就增加了两个行为，分别用于把导航栏和工具栏同时隐藏和显示，如下图中下面方框所示。Behavior 的扩展性很强，你甚至可以定义执行一些脚本，所以可以做的事情很多。



18.3.2 查找技巧

Xcode 的查找替换功能提供了 Insert Pattern 的方式，方便你输入常见的查找规则。在查找时单击输入框左边的放大镜图标，选择“Insert Pattern”，即可选择回车等特殊字符。在下图的示例中，该查找替换功能将所有的以 @synthesize 开始的行删除。



18.3.3 JavaScript 文件设置调整

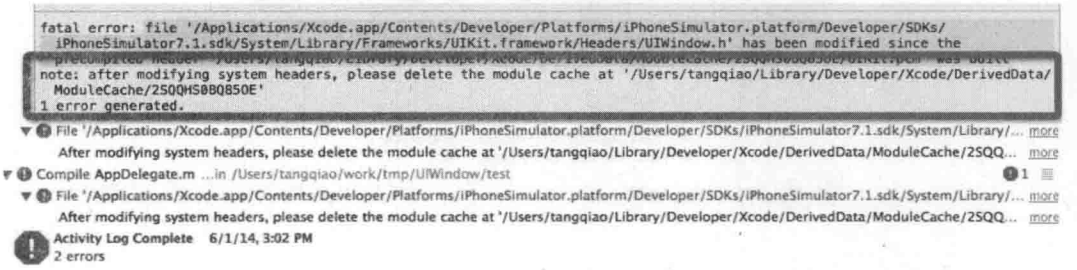
JavaScript 的 js 后缀的文件默认被拖动到工程中后，是在编译列表中，而不是资源列表中。你需要手工地调整其位置，否则它就不能打包到 ipa 文件中。

这应该是 Xcode 的一个长期 Bug，js 文件并不需要编译。做混合开发的新手很容易在第一次尝试时被这个问题困扰。下面是一个示例，你需要手工将这个 js 文件从 Compile Souce 移动到 Copy Bundle Resources。

18.3.4 清除 DerivedData

当多次重构工程造成代码没有错却编译失败时，可以尝试删除 DerivedData 目录。DerivedData 目录是 Xcode 的编译缓存，路径是 ~/Library/Developer/Xcode/DerivedData。

在工程代码进行 git merge 或 git rebase 的时候常常造成缓存异常。下图就是我某一次执行 git merge 后造成的工程编译信息异常，在清除 DerivedData 目录后重启 Xcode 就正常了。



18.3.5 target 信息异常

当工程的编译 target 信息异常的时候，可以删除 YourPjectName.xcodeproj/xcuserdata 目录。该目录下存有当前用户的各种工程状态信息，删除后重启 Xcode，Xcode 会自动重建该目录。

18.3.6 下载 Xcode

Xcode 除了能够在 App Store 直接下载外，还可以用开发者账号登陆开发者中心https://developer.apple.com/downloads/index.action下载。

在开发者中心下载的好处是，下载之后的 Xcode 是一个安装包，可以分享给其他同事。另外一个好处是，在下载时获得的下载地址可以复制到第三方的下载工具中使用，使下载速度更快一些。

在开发者中心下载的缺点是，不能享受 App Store 自动更新软件的方便了，并且 Mac App Store 更新是支持增量更新的，所以很多时候只需要下载很小的增量包即可完成更新。

18.4 调试技巧

18.4.1 模拟器快捷键

下面是我常用的模拟器快捷键：

- Cmd + 1/2/3 可以切换模拟器的显示比例。
- Opt + Shift 可以在模拟器中调出双指拖动效果。
- Opt 可以在模拟器中调出双指放大缩小效果。
- Cmd + Shift + H 是模拟器的 Home 键。
- Cmd + Left/Right 可以切换横竖屏。

如果你不想记这么多快捷键，那也一定要记住 Cmd + 1/2/3，因为 Retina 模拟器不能在 23 英寸的 iMac 显示器中完整显示。

18.4.2 覆盖安装注意事项

在模拟器或真机上进行应用调试时，如果是覆盖安装，新删除的资源文件并不会马上在模拟器或真机上被删掉。如果你的代码必须在没有该资源文件存在时才能正常工作，那么你

就需要手工强制把以前的应用删掉重装。

在有一些开发场景中比较容易触发该问题，例如你开始创建了一个带 xib 的 SampleViewController，然后通过 [[SampleViewController alloc] init] 这种方式初使化，那么系统就会默认查找有没有 xib 文件，有的话就从 xib 文件中加载界面元素。

如果这个时候你又改变主意在工程中把 SampleViewController.xib 文件删除了，用代码的方式来创建界面，这时新的调试默认就不会将目标机器上的 SampleViewController.xib 文件删除，所以会造成逻辑异常。

18.4.3 给模拟器相册增加图片

测试需要图片，但模拟器的相册是空的怎么办？可以把图片从 Finder 中拖动到模拟器中。用 Safari 打开这张图，然后长按这张图，在弹出的菜单中选择“Save Image”，就可以把图片保存到模拟器的相册中了，如下图所示。

18.4.4 获得模拟器中的程序数据

其实模拟器中的程序存储在/yourHome/Library/Application Support/iPhone Simulator/7.0/Applications 目录下，你可以直接进去找程序运行时保存的相关数据。

另外，如果要删除所安装的程序，也可以直接将 Applications 目录下的文件夹删掉，这比在模拟器中删更方便。

18.4.5 安装旧版本的模拟器

在 Xcode 中只会带最新版的模拟器，虽然有些模拟器可以在设置里面下载，但是更老的版本就无法在 Xcode 的设置页下载了。

这个问题的解决办法是把以前的 Xcode 和对应的模拟器装上，然后将模拟器复制或链接到最新的 Xcode 目录下。

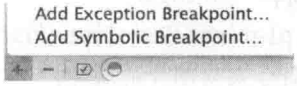
18.4.6 模拟慢速网络

在模拟器上可以通过之前介绍过的 Charles 工具来模拟慢速网络。

在真机上可以使用苹果为开发者提供的相关设置来实现，具体位置是在“设置”→“开发者”→“Status”→“Network Link Conditioner”中。如果你没有在手机设置中找到该选项，请尝试将手机连接到电脑，然后用 Xcode 的 Origanizer 将其设置成测试设备。

18.4.7 异常断点与符号断点

在 Xcode 中按 Cmd + 7 跳转到断点管理界面，单击左下角的“+”号，则可以添加异常断点或符号断点，如下图所示。



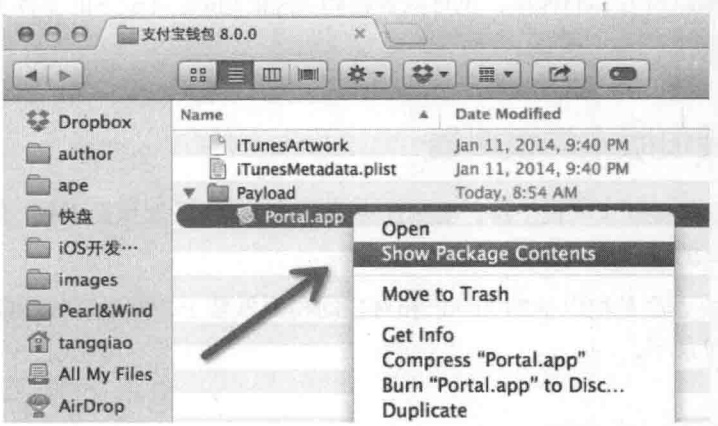
异常断点可以在程序因为异常崩溃退出前暂停，便于我们定位出错位置。

符号断点可以给指定的方法加断点，适于我们为系统的方法增加暂停位置。

18.5 ipa 文件格式

18.5.1 查看 ipa 的内容

经过 Xcode 编译生成的 ipa 文件实际上就是一个 zip 文件。我们将任意从 App Store 上下载的 ipa 文件名的后缀改成“zip”，然后双击即可解压打开。解压后的程序在 Payload 目录下，是一个“app”后缀的文件夹。我们在该文件上单击右键，选择“Show Package Contents”，即可看到所有的程序资源文件，如下图所示。



利用上面的办法，我们可以在一定程度上了解和学习别的应用的实现方式，例如我在开发有道云笔记的时候，就曾解开 Evernote 的 iPad 版，看到打包资源里面有大量的 js 文件。通过阅读这些 js 文件，学习到它们是用 UIWebView 的方式来实现的编译界面。

另外我们也要注意，打包时不要把不必要的文档包含进去，有些朋友不注意常常会把一些内部的 API 调用文档误打包进 ipa 文件中。在 2014 年初，国内的某互联网上市公司甚至出现了员工把项目源代码当作资源文件打包进 ipa 文件中的安全事件，这实在是不应该发生的。

18.5.2 查看 ipa 中的图片

默认情况下，ipa 文件中的图片会通过 Xcode 自带的图片压缩工具 pngcrush 压缩，pngcrush 程序位于/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/。如果你解开 ipa 之后无法查看图片内容，需要使用如下的命令来将图片解压缩：

```
pngcrush -revert-iphone-optimizations 源文件名 目的文件名
```

这里也为读者提供一个批量转换的脚本：

```
#!/bin/bash

#创建解压文件夹
mkdir OriginImages

for png in `find . -name '*.png'`
do
  # 获得文件名
  name=`basename $png`
  pngcrush -revert-iphone-optimizations $name OriginImages/$name
done
```

18.6 为工程增加 Daily Build

18.6.1 前言

为工程增加 Daily Build 是一件非常有意义的事情，也是敏捷开发中关于“持续集成”的一个实践。Daily Build 对于开发来说有如下好处：

- 保证了每次提交的代码可用，不会造成整个工程编译失败。
- 进度跟进。产品经理可以每天看到最新的开发进度，并且能试用产品，调整一些细节。很多时候，一个新功能，只有当你真正用了，才能体会到好或不好，所以 Daily Build 也给产品经理更多时间来调整他的设计。
- 需求确认。产品经理可以尽早确认开发的功能细节是否符合他的预期。因为互联网公司在开发时间上都比较紧凑，所以一般都没有传统的需求说明文档，所以 Daily Build 也能让产品经理尽早确认开发的功能细节是否符合他的预期，我就遇到过产品经理发现开发出的功能细节和他的预期不一致的情况，但是因为有 Daily Build，使得我可以尽早做修改，把修改的代价减小了。
- 测试跟进。如果功能点是独立的话，测试人员完全可以根据 Daily Build 来进行一些早期的测试，越早的 Bug 反馈可以使得修改 Bug 所需的时间越短。

18.6.2 步骤

xcodebuild 命令

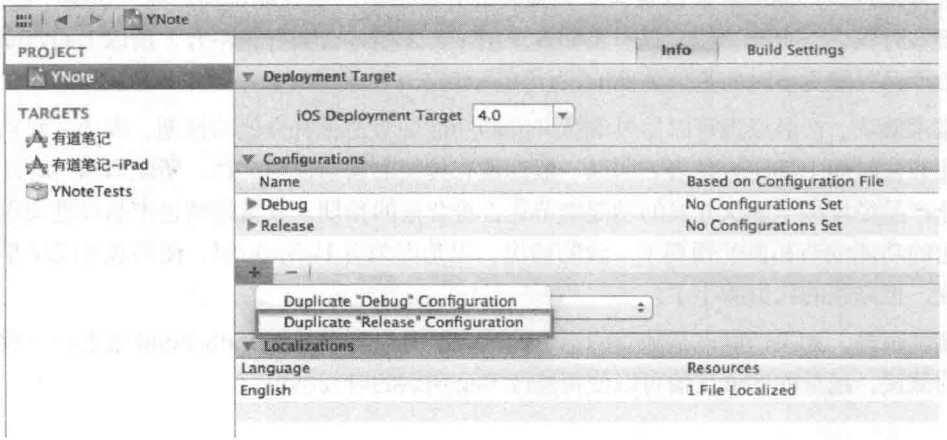
如何做 Daily Build 呢？其实 Xcode 就提供了命令行 build 的命令，这个命令是 xcodebuild，用 `xcodebuild -usage` 可以查看到所有的可用参数。代码如下所示：

```
[tangqiao ~]$xcodebuild -usage
Usage: xcodebuild [-project <projectname>] [[-target <targetname>]...|-alltargets]
    [-configuration <configurationname>] [-arch <architecture>]... [-sdk [<sdkname
    >|<sdkpath>]] [<buildsetting>=<value>]... [<buildaction>]...
       xcodebuild [-project <projectname>] -scheme <schemeName> [-configuration <
           configurationname>] [-arch <architecture>]... [-sdk [<sdkname>|<sdkpath
           >]] [<buildsetting>=<value>]... [<buildaction>]...
       xcodebuild -workspace <workspacename> -scheme <schemeName> [-configuration <
           configurationname>] [-arch <architecture>]... [-sdk [<sdkname>|<sdkpath
           >]] [<buildsetting>=<value>]... [<buildaction>]...
       xcodebuild -version [-sdk [<sdkfullpath>|<sdkname>] [<infoitem>] ]
       xcodebuild -list [[-project <projectname>]|[-workspace <workspacename>]]
       xcodebuild -showsdks
```

一般情况下的命令使用如下：

```
xcodebuild -configuration Release -target "YourProduct"
```

但在 Daily Build 中，把 Release 用为 Configuration 其实不是特别好。因为 Release 的证书可能经常会被修改。我们可以基于 Release 的 Configuration，建一个专门用于 Daily Build 的 Configuration。方法是：在工程详细页面中，选择“Info”一栏，在“Configurations”一栏的下方单击“+”号，然后选择“Duplicate "Release" Configuration”，新建名为“DailyBuild”的 Configuration，如下图所示。



之后就可以用如下命令来做 Daily Build 了：

```
xcodebuild -configuration DailyBuild -target "YourProduct"
```

执行完命令后，会在当前工程下的 build/DailyBuild-iphoneos/目录下生成一个名为“YourProduct.app”的文件。这个就是我们 build 成功之后的程序文件。

生成 ipa 文件

接下来我们需要生成 ipa 文件，因为 ipa 文件实际上就是一个 zip 文件，我们使用系统的 zip 命令来生成 ipa 文件即可。需要注意的是，ipa 文件并不是简单地将编辑好的 app 文件打包成 zip 文件，它需要将 app 文件放在一个名为“Payload”的文件夹下，然后将整个 Payload 目录打包成为 ipa 文件，命令如下所示：

```
cd $BUILD_PATH
mkdir -p ipa/Payload
cp -r ./DailyBuild-iphoneos/$PRODUCT_NAME ./ipa/Payload/
cd ipa
zip -r $FILE_NAME *
```

生成安装文件

苹果允许用 itms-services 协议来直接在 iPhone/iPad 上安装应用程序，我们可以直接生成该协议需要的相关文件，这样产品经理和测试人员都可以直接在设备上安装新版的应用了。

具体来说，就是需要生成一个带 itms-services 协议的链接的 HTML 文件，以及一个 plist 文件。需要注意的是，从 iOS7.1 开始，这个 plist 文件的地址必须是 HTTPS 的。

生成 HTML 的示例代码如下：

```
cat << EOF > install.html
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>安装此软件</title>
  </head>
  <body>
    <ul>
      <li>安装此软件:<a href="itms-services://?action=download-manifest&url=https%3A
          %2F%2Fwww.yourdomain.com%2Fynote.plist">$FILE_NAME</a></li>
    </ul>
    </div>
  </body>
</html>
EOF
```

生成 plist 文件的代码如下，注意，需要将下面的涉及www.yourdomain.com的地方换成你线上服务器的地址，将“ProductName”换成你的应用安装后的名字。

```
cat << EOF > ynote.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
    PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>items</key>
   <array>
       <dict>
           <key>assets</key>
           <array>
               <dict>
                   <key>kind</key>
                   <string>software-package</string>
                   <key>url</key>
                   <string>http://www.yourdomain.com/$FILE_NAME</string>
               </dict>
```

```
                <dict>
                    <key>kind</key>
                    <string>display-image</string>
                    <key>needs-shine</key>
                    <true/>
                    <key>url</key>
                    <string>http://www.yourdomain.com/icon.png</string>
                </dict>
            <dict>
                    <key>kind</key>
                    <string>full-size-image</string>
                    <key>needs-shine</key>
                    <true/>
                    <key>url</key>
                    <string>http://www.yourdomain.com/icon.png</string>
                </dict>
            </array><key>metadata</key>
            <dict>
                <key>bundle-identifier</key>
                <string>com.yourdomain.productname</string>
                <key>bundle-version</key>
                <string>1.0.0</string>
                <key>kind</key>
                <string>software</string>
                <key>subtitle</key>
                <string>ProductName</string>
                <key>title</key>
                <string>ProductName</string>
            </dict>
        </dict>
    </array>
</dict>
</plist>


EOF
```

定时运行

这一点非常简单，使用 crontab -e 命令即可。大家可以用 Google 随意搜索一下 crontab 命令，可以找到很多相关文档。假如我们要每周一到周五的早上 9 点钟执行 Daily Build，则 crontab 的配置如下：

```
0 9 * * * 1-5 /Users/tangqiao/dailybuild.sh >> /Users/tangqiao/dailybuild.log 2>&1
```

失败报警

在 Daily Build 脚本运行失败时，最好能发报警邮件或者短信，以便让我们能够尽早发现。发邮件可以用 python 的 smtplib 来写，示例如下：

```
import smtplib

sender = 'sender@devtang.com'
receivers = ['receiver@devtang.com']

message = """From: Alert <sender@devtang.com>
To: Some one <receiver@devtang.com>
Subject: SMTP email sample

Hope you can get it.
"""

try:
    obj = smtplib.SMTP('server.mail.devtang.com')
    obj.sendmail(sender, receivers, message)
    print 'OK: send mail succeed'
except Exception:
    print 'Error: unable to send mail'
```

上传

Daily Build 编译出来后，如果需要单独上传到另外一台 Web 机器上，可以用 FTP 或者 SCP 协议。如果 Web 机器是 Windows 机器的话，可以在 Windows 机器上开一个共享，然后用 `mount -t smbfs` 来将这个共享 mount 到本地，相关的示例代码如下：

```
mkdir upload
mount -t smbfs //$SMB_USERNAME:$SMB_PASSWORD@$SMB_TARGET ./upload
if [ "$?" -ne 0 ]; then
    echo "Failed to mount smb directory"
    exit 1
fi
mkdir ./upload/$FOLDER
cp $FILE_NAME ./upload/$FOLDER/
if [ "$?" -eq 0 ]; then
    echo "[OK] $FILE_NAME is uploaded to $SMB_TARGET"
else
    echo "[ERROR] $FILE_NAME is FAILED to  uploaded to $SMB_TARGET"
fi
umount ./upload
```

18.6.3 遇到的问题

如果你的脚本在手动执行时很正常，但是在用 crontab 启动时就会出现找不到开发者证书的错误，可以尝试用如下方法解决：在“钥匙串访问”中把开发者证书从“登录”栏拖动到“系统”栏，如下图所示。

18.6.4 总结

将以上各点结合起来，就可以用 bash 写出一个 Daily Build 脚本了。

如果你不想自己从头写 Daily Build 脚本，也可以参考我的好朋友 @lexrus 开源的自动打包脚本 ios-makefile（https://github.com/lexrus/ios-makefile）。该打包脚本虽然只有 300 多行代码，但是包含的功能相当完整，大家也可以在它的基础上做修改定制，以适自各自不同的需求。

18.7 使用脚本提高开发效率

18.7.1 删除未使用的图片资源

多次界面改版之后，如何知道自己的工程中有哪些图片资源没有使用？这里给大家提供一个小脚本，可以删除未使用的图片资源（使用前需要先 `brew install ack`，安装一个名为“ack”的命令行工具：

```
for i in `find . -name "*.png" -o -name "*.jpg"`; do
    file=`basename -s .jpg "$i" | xargs basename -s .png | xargs basename -s @2x`
```

```
    result=`ack -i "$file"`
    if [ -z "$result" ]; then
        echo "$i"
        # 如果需要，可以直接执行删除：
        # rm "$i"
    fi
done
```

18.7.2 用脚本自动生成小尺寸的图片

我们知道，在 iOS 开发中，为了使我们的应用能够同时支持 iPhone 的 Retina 屏幕和普通屏幕，美工需要对 UI 设计稿中的每个元素进行二次切图。苹果要求图片的命名分别为“name.png”和“name@2x.png”，带“@2x”的是 Retina 屏幕的贴图，不带“@2x”的文件为普通屏幕的贴图。

因为带 @2x 的图片大小是不带 @2x 图片的 2 倍，所以，我们完全可以让美工只切带 @2x 的大图，而我们使用脚本来生成小图。于是我写了下面这样的一个脚本，我只需要将所有的大图按照类似“name-1@2x.png”、“name-2@2x.png”的方式命名，然后脚本就会自动生成对应的名为“name-1.png”和“name-2.png”的小图。

使用该脚本时，请先用 `brew install imagemagick` 命令安装 imagemagick。imagemagick 是一个相当强大的图像处理库。

```
#! /bin/bash
# File name : convertImage.sh
# Author: Tang Qiao
#

# print usage
usage() {
    cat << EOF
    Usage:
        convertImage.sh <src directory> <dest directory>
EOF
}

if [ $# -ne 2 ]; then
    usage
    exit 1
fi

SRC_DIR=$1
DEST_DIR=$2
```

```
# check src dir
if [ ! -d $SRC_DIR ]; then
    echo "src directory not exist: $SRC_DIR"
    exit 1
fi

# check dest dir
if [ ! -d $DEST_DIR ]; then
    mkdir -p $DEST_DIR
fi

for src_file in $SRC_DIR/*.* ; do
    echo "process file name: $src_file"
    # 获得去掉文件名的纯路径
    src_path=`dirname $src_file`
    # 获得去掉路径的纯文件名
    filename=`basename $src_file`
    # 获得文件名字(不包括扩展名)
    name=`echo "$filename" | cut -d'.' -f1`
    # remove @2x in filename if there is
    name=`echo "$name" | cut -d"@" -f1`
    # 获得文件扩展名
    extension=`echo "$filename" | cut -d'.' -f2`
    dest_file="$DEST_DIR/${name}.${extension}"

    convert $src_file -resize 50% $dest_file
done
```

脚本使用方法：将以上代码另存为 convertImage.sh，然后用以下方式调用此脚本，即可将源文件夹中所有以 @2x 结尾的图片文件转换成一半大小的、去掉 @2x 的小图片。

```
convertImage.sh 源文件夹 目标文件夹
```

除了使用以上脚本外，你也可以选择第三方的软件帮助你缩图，例如免费的 XnConvert：http://www.xnview.com/en/xnconvert/ 。

18.7.3 检查图片

我发现美工给我的大图的长宽常常不是偶数，这样缩小的图就不是原图的整数分之一了。为了方便我检查美工给我的图片是否宽高都是偶数，我写了如下的检查脚本，这样就可以检查图片的宽高是否符合要求了：

```
#! /bin/bash
# File name : checkImageSize.sh
# Author: Tang Qiao
```

```
#

usage() {
    cat <<EOF
    Usage:
        checkImageSize.sh <directory>
EOF
}

if [ $# -ne 1 ]; then
    usage
    exit 1
fi

SRC_DIR=$1

# check src dir
if [ ! -d $SRC_DIR ]; then
    echo "src directory not exist: $SRC_DIR"
    exit 1
fi

for src_file in $SRC_DIR/*.png ; do
    echo "process file name: $src_file"
    width=`identify -format "%[fx:w]" $src_file`
    height=`identify -format "%[fx:h]" $src_file`
    # check width
    modValue=`awk -v a=$width 'BEGIN{printf "%d", a % 2}'`
    if [ "$modValue" == "1" ]; then
       echo "[Error], the file $src_file width is $width"
    fi
    # check height
    modValue=`awk -v a=$height 'BEGIN{printf "%d", a % 2}'`
    if [ "$modValue" == "1" ]; then
       echo "[Error], the file $src_file height is $height"
    fi
done
```

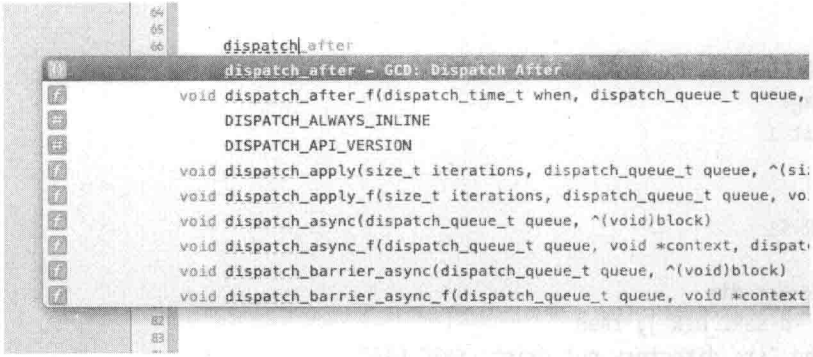
18.8 管理代码片段

18.8.1 代码片段介绍

Xcode 从 4.0 版本开始提供了代码片段（Code Snippets）功能，功能入口在 Xcode 整个界面的右下角，可以通过快捷键 Cmd + Ctrl + Opt + 2 调出来。代码片段是一些代码的模板，对

于一些常见的编程模式，Xcode 都将这些代码抽象成模板放到代码片段中，使用的时候，只需要按快捷键，就可以把模板的内容填到代码中。

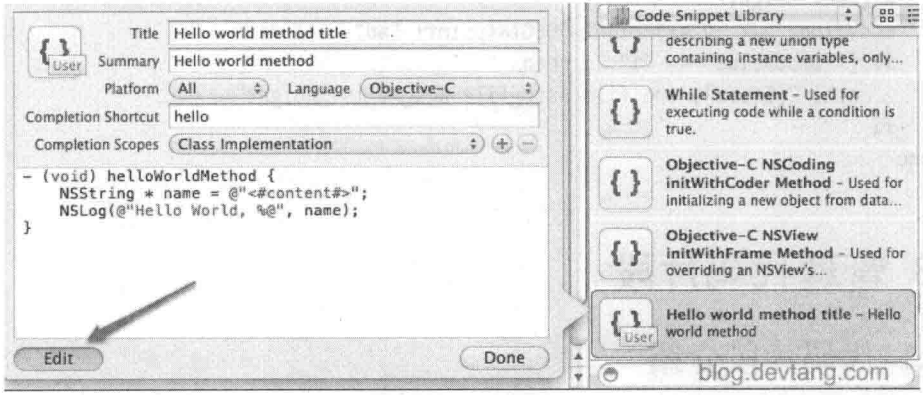
例如，在苹果引入 GCD（Grand Central Dispatch）后，当我们需要一个延时的操作时，只需要在 Xcode 中键入“dispatch”，就可以看到 Xcode 中弹出一个上下文菜单，第一项就是相应的代码片段，如下图所示。



18.8.2 定义自己的代码片段

那么如何自定义代码片段呢？相当简单，当你觉得某段代码很有用，可以当作模板的时候，将其整体选中，拖动到 Xcode 右下角的代码片段区域中即可。Xcode 会自动帮你创建一个新的代码片段。之后你可以单击该代码片段，在弹出的界面中选择“Edit”，即可为此代码片段设置快捷键等信息。

如果有些地方你想让用户替换掉，可以用 <# 被替换的内容 #> 的格式。这样在代码片段被使用后，焦点会自动移到该处，你只需要连贯地键入替换后的内容即可，如下图所示。



18.8.3 使用 Git 管理代码片段

在了解了代码片段之后，我就想能不能用 Git 来管理它。于是就研究了一下，我发现它都存放于目录 ~/Library/Developer/Xcode/UserData/CodeSnippets 中。于是，我就将这个目录设置成一个 Git 的版本库，然后将自己整理的代码片段都放到 Github 上了。

现在我有三台 Mac 机器，其中两台是笔记本，一台是 iMac，我常常在三台机器间切换着工作，由于将代码片段都放在 GitHub 上，所以我在任何一端有更新，另一端都可以很方便地用 Git Pull 将更新拉到本地。有一次我重装了 Mac OS X 操作系统，之后又重装了 Xcode，简单设置一下，所有代码片段都回来了，非常方便。

我的代码片段的地址是https://github.com/tangqiaoboy/Xcode_tool，要使用它非常方便，只需要如下三步即可：

```
git clone https://github.com/tangqiaoboy/Xcode_tool
cd Xcode_tool
./setup_snippets.sh
```

大家也可以将我的 GitHub 项目 fork 一份，改成自己的。这样可以方便地增加和管理自己的代码片段。

18.8.4 其他代码片段管理工具

除了使用 Xcode 自带的代码片段外，我们还可以使用类似 dash（http://kapeli.com/dash）一类的第三方工具。dash 的代码片段功能虽然与 Xcode 的集成不如原生的更方便，但是 dash 可以与任何编辑器整合，并且可以管理任何文本片段。

第三部分：iOS 开发底层原理

本部分讲解 iOS 开发涉及的底层原理。主要涉及 Objective-C 对象模型、Tagged Pointer 对象，以及 Block 对象模型。

我们通过学习 iOS 开发中的内存管理方式，了解到引用计数的特点及局限，从而避免错误使用造成内存泄漏。

我们通过了解 Objective-C 的对象模型，来理解它如何支持 KVO、Method Swizzling、Block 等特性。

通过了解语言实现细节，将加深我们对于语言的理解，从而能更加深入地理解语言背后的本质。

第三部分：iOS 开发原理篇

第19章 Objective-C 对象模型

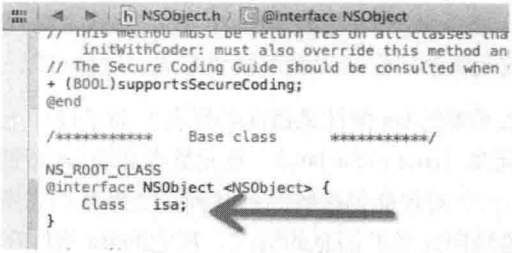
19.1 简介

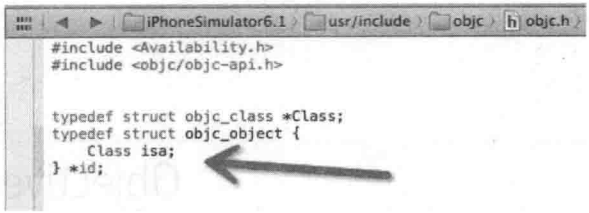
本章主要介绍 Objective-C 对象模型的实现细节，以及 Objective-C 语言对象模型中对 isa swizzling 和 method swizzling 的支持。希望本章能加深你对 Objective-C 对象的理解。

19.2 isa 指针

Objective-C 是一门面向对象的编程语言，每一个对象都是一个类的实例。在 Objective-C 语言的内部，每一个对象都有一个名为 isa 的指针，指向该对象的类。每一个类描述了一系列它的实例的特点，包括成员变量的列表、成员函数的列表等。每一个对象都可以接收消息，而对象能够接收的消息列表保存在它所对应的类中。

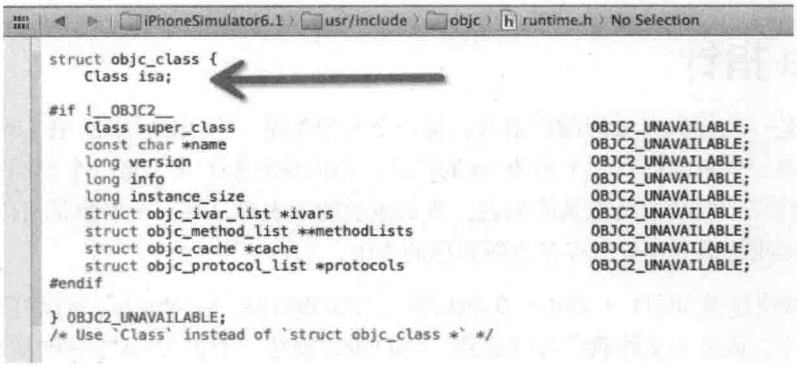
在 Xcode 中按快捷键 Shift + Cmd + O，然后输入“NSObject.h”和“objc.h”，可以打开 NSObject 的定义头文件，通过头文件我们可以看到，NSObject 就是一个包含 isa 指针的结构体，如下图所示。





按照面向对象语言的设计原则，所有事物都应该是对象（严格来说 Objective-C 并没有完全做到这一点，因为它有像 int、double 这样的简单变量类型，而类似 Ruby 一类的语言，连 int 变量也是对象）。在 Objective-C 语言中，每一个类实际上也是一个对象。每一个类也有一个名为 isa 的指针。每一个类也可以接收消息，例如代码 `[NSObject alloc]`，就是向 NSObject 这个类发送名为“alloc”的消息。

在 Xcode 中按快捷键 `Shift + Cmd + O`，然后输入“runtime.h”，可以打开 Class 的定义头文件，通过头文件我们可以看到，Class 也是一个包含 isa 指针的结构体，如下图所示（图中除了 isa 外还有其他成员变量，但那是为了兼容非 2.0 版的 Objective-C 的遗留逻辑，大家可以忽略它）。



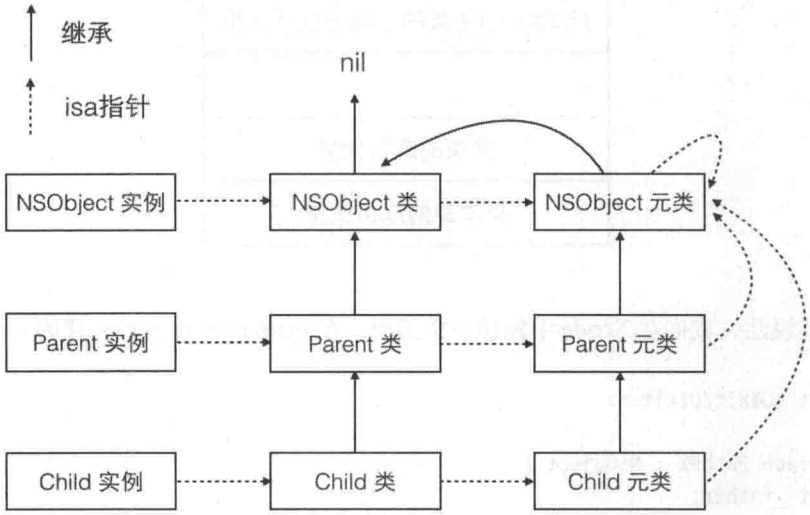
因为类也是一个对象，所以它也必须是另一个类的实列，这个类就是元类（metaclass）。元类保存了类方法的列表。当一个类方法被调用时，元类会首先查找它本身是否有该类方法的实现，如果没有，则该元类会向它的父类查找该方法，这样可以一直找到继承链的头。

元类也是一个对象，那么元类的 isa 指针又指向哪里呢？为了设计上的完整，所有的元类的 isa 指针都会指向一个根元类（root metaclass）。根元类本身的 isa 指针指向自己，这样就形成了一个闭环。上面提到，一个对象能够接收的消息列表是保存在它所对应的类中的。在实际编程中，我们几乎不会遇到向元类发消息的情况，那它的 isa 指针在实际上很少用到。不过这么设计保证了面向对象概念在 Objective-C 语言中的完整，即语言中的所有事物都是对象，都有 isa 指针。

我们再来看看继承关系，由于类方法的定义是保存在元类中，而方法调用的规则是，如果该

类没有一个方法的实现，则向它的父类继续查找。所以，为了保证父类的类方法在子类中可以被调用，所有子类的元类都会继承父类的元类，换而言之，类对象和元类对象有着同样的继承关系。

我很想把关系说清楚一些，但是这块儿确实有点绕，我们还是来看图吧，很多时候图像比文字表达起来更为直观。下面这张图或许能够让大家对 isa 和继承的关系看得更清楚一些。



我们可以从图中看出：

1. NSObject 的类中定义了实例方法，例如-(id)init 方法和-(void)dealloc 方法。
2. NSObject 的元类中定义了类方法，例如 +(id)alloc 方法、+(void)load 方法和 +(void)initialize 方法。
3. NSObject 的元类继承自 NSObject 类，所以 NSObject 类是所有类的根，因此 NSObject 中定义的实例方法可以被所有对象调用，例如-(id)init 方法和-(void)dealloc 方法。
4. NSObject 的元类的 isa 指向自己。

19.3 类的成员变量

如果把类的实例看成一个 C 语言的结构体（struct），上面说的 isa 指针就是这个结构体的第一个成员变量，而类的其他成员变量依次排列在结构体中。排列顺序如下图所示。

为了验证该说法，我们在 Xcode 中新建一个工程，在 main.m 中运行如下代码：

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

@interface Father : NSObject {
    int _father;
}

@end

@implementation Father
@end

@interface Child : Father {
    int _child;
}

@end

@implementation Child
@end

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

  Child * child = [[Child alloc] init];
  @autoreleasepool {
      // ...
  }
```

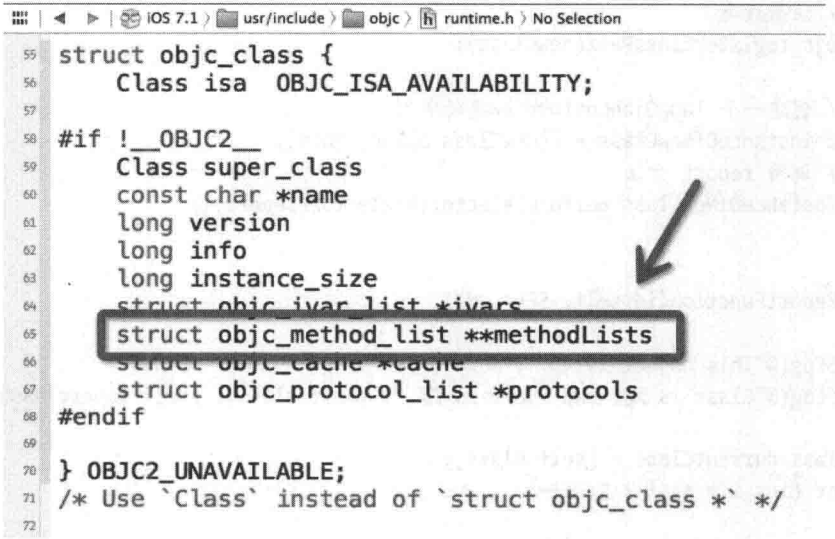
```
}
```

我们将断点下在 @autoreleasepool 处，然后在 Console 中输入 p *child，则可以看到 Xcode 输出如下内容，这与我们上面的说法一致。

```
(lldb) p *child
(Child) $0 = {
  (Father) Father = {
    (NSObject) NSObject = {
      (Class) isa = Child
    }
    (int) _father = 0
  }
  (int) _child = 0
}
```

因为对象在内存中的排布可以看成一个结构体，该结构体的大小并不能动态变化，所以无法在运行时动态地给对象增加成员变量。

相应地，对象的方法定义都保存在类的可变区域中。Objective-C 2.0 并未在头文件中将实现暴露出来，但在 Objective-C 1.0 中，我们可以看到方法的定义列表是一个名为 methodLists 的指针的指针，如下图所示。通过修改该指针指向的指针的值，就可以动态地为某一个类增加成员方法。这也是 Category 实现的原理。同时也说明了为什么 Category 只可为对象增加成员方法，却不能增加成员变量。[1]



[1] 需要特别说明一下，通过 objc_setAssociatedObject 和 objc_getAssociatedObject 方法可以变相地给对象增加成员变量，但由于实现机制不一样，所以并不是真正改变了对象的内存结构。

因为 isa 本身也只是一个指针，所以除了对象的方法可以动态地修改外，我们也可以在运行时动态地修改 isa 指针的值，达到替换对象整个行为的目的，不过该应用场景较少。

19.4 对象模型的应用

19.4.1 动态创建对象

我们可以使用 Objective-C 语言提供的与 runtime 相关的函数，动态地创建一个新的类，并且通过相关的方法来获得 isa 指针的值，从而了解对象的内部结构。

我们先来看动态创建类的代码：

```
#import <objc/runtime.h>

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 创建一个名为 TangQiaoCustomView 的类，它是UIView的子类
    Class newClass = objc_allocateClassPair([UIView class], "TangQiaoCustomView", 0)
        ;
    // 为该类增加一个名为 report 的方法
    class_addMethod(newClass, @selector(report), (IMP)ReportFunction, "v@:");
    // 注册该类
    objc_registerClassPair(newClass);

    // 创建一个 TangQiaoCustomView类的实例
    id instanceOfNewClass = [[newClass alloc] init];
    // 调用 report 方法
    [instanceOfNewClass performSelector:@selector(report)];
}

void ReportFunction(id self, SEL _cmd)
{
    NSLog(@"This object is %p.", self);
    NSLog(@"Class is %@, and super is %@.", [self class], [self superclass]);

    Class currentClass = [self class];
    for (int i = 1; i < 5; i++)
    {
        NSLog(@"Following the isa pointer %d times gives %p", i, currentClass);
        currentClass = object_getClass(currentClass);
    }
```

```
    NSLog(@"NSObject's class is %p", [NSObject class]);
    NSLog(@"NSObject's meta class is %p", object_getClass([NSObject class]));
}

@end
```

该代码的关键之处主要有以下几点：

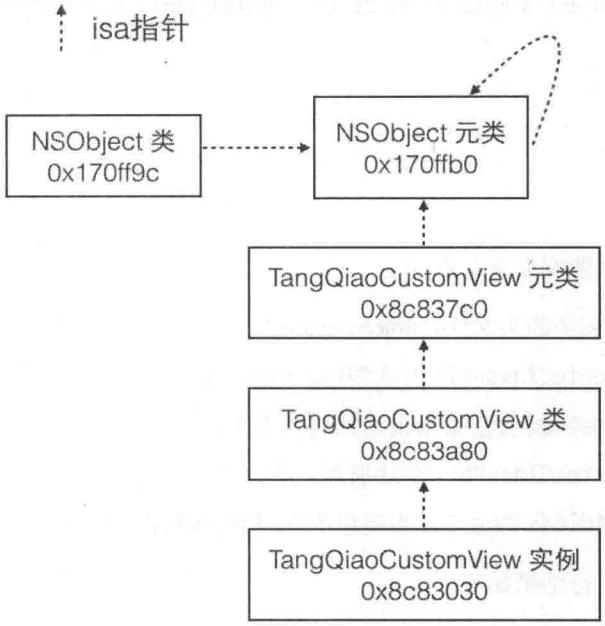
1. import rumtime 相关的头文件：objc/runtime.h。
2. 使用 objc_allocateClassPair 方法创建新的类。
3. 使用 class_addMethod 方法来给类增加新的方法。
4. 使用 objc_registerClassPair 来注册新的类。
5. 使用 object_getClass 方法来获得对象的 isa 指针所指向的对象。

最终，我们得到的运行结果如下：

```
This object is 0x8c83030.
Class is TangQiaoCustomView, and super is UIView.
Following the isa pointer 1 times gives 0x8c83a80
Following the isa pointer 2 times gives 0x8c837c0
Following the isa pointer 3 times gives 0x170ffb0
Following the isa pointer 4 times gives 0x170ffb0
NSObject's class is 0x170ff9c
NSObject's meta class is 0x170ffb0
```

我将上面的运行结果中的内存地址与对应的类画在了下面的图中，大家从中可以清楚地看到：

1. 从 TangQiaoCustomView 对象开始，在我们连续读取了 3 次 isa 指针所指向的对象后，isa 指针所指向的地址变成了 0x170ffb0，也就是我们说的 NSObject 元类的地址。之后我们第 4 次取 isa 指针所指的对象时，其结果仍然为 0x170ffb0，这说明 NSObject 元类的 isa 指针确实是指向它自己的。
2. 作为对比，我们在代码最后获取了 NSObject 类的 isa 指针地址，我们可以看到其值都是 0x170ffb0，这说明所有的元类对象的 isa 指针，都是指向 NSObject 元类的。



19.4.2 系统相关 API 及应用

isa swizzling 的应用

系统提供的 KVO 的实现，就利用了动态地修改 isa 指针的值的技术。在苹果的文档（https://developer.apple.com/library/ios/documentation/cocoa/conceptual/KeyValueObserving/Articles/KVOImplementation.html）中可以看到如下描述：

> Key-Value Observing Implementation Details > >Automatic key-value observing is implemented using a technique called isa-swizzling. > >The isa pointer, as the name suggests, points to the object's class which maintains a dispatch table. This dispatch table essentially contains pointers to the methods the class implements, among other data. > >When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class. As a result the value of the isa pointer does not necessarily reflect the actual class of the instance.
>
> You should never rely on the isa pointer to determine class membership. Instead, you should use the class method to determine the class of an object instance.

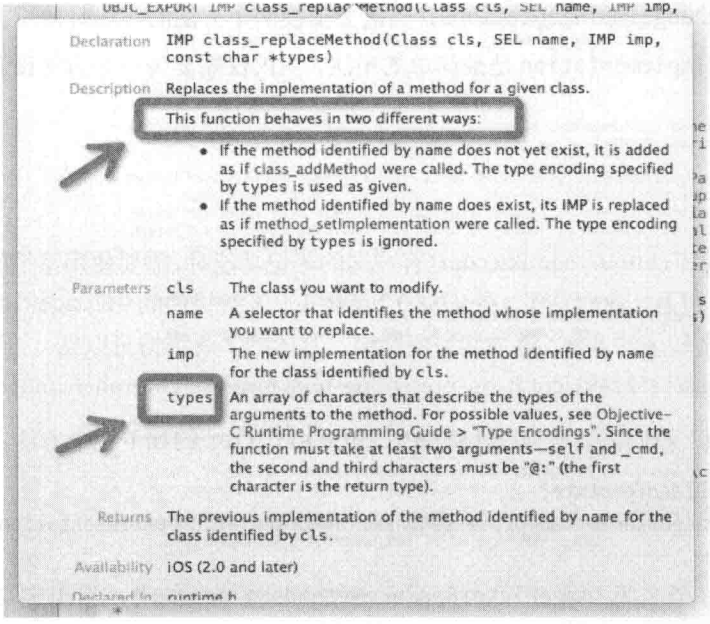
Method Swizzling API 说明

Objective-C 提供了以下 API 来动态替换类方法或实例方法的实现：

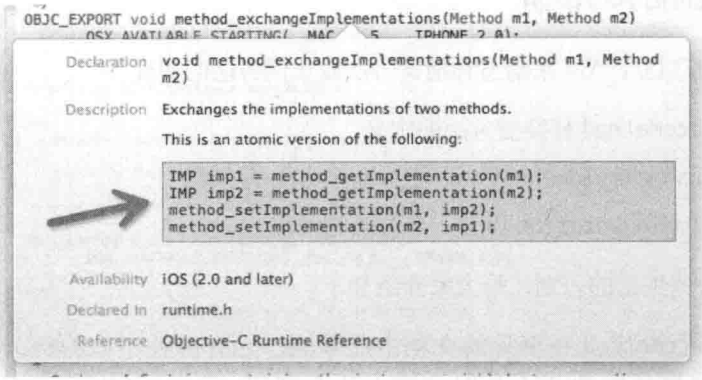
- class_replaceMethod 替换类方法的定义。
- method_exchangeImplementations 交换两个方法的实现。
- method_setImplementation 设置一个方法的实现。

这三个方法有一些细微的差别，给大家介绍如下：

- class_replaceMethod 在苹果的文档中能看到，它有两种不同的行为，如下图所示。当类中没有想替换的原方法时，该方法会调用 class_addMethod 来为该类增加一个新方法，也正因为如此，class_replaceMethod 在调用时需要传入 types 参数，而 method_exchangeImplementations 和 method_setImplementation 却不需要。



- method_exchangeImplementations 的内部实现其实是调用了两次 method_setImplementation 方法，从苹果的文档中能清晰地了解到，如下图所示。



从以上的区别我们可以总结出这三个 API 的使用场景：

- class_replaceMethod，当需要替换的方法有可能不存在时，可以考虑使用该方法。
- method_exchangeImplementations，当需要交换两个方法的实现时使用。
- method_setImplementation 是最简单的用法，当仅仅需要为一个方法设置其实现方式时使用。

使用示例

我们在开发猿题库(http://yuantiku.com)客户端的笔记功能时，需要使用系统的 UIImagePickerController。但是，我们发现，在 iOS6.0.2 系统下，系统提供的 UIImagePickerController 在 iPad 横屏下有转屏的 Bug，造成其方向错误。具体的 Bug 详情可以见：。(http://stackoverflow.com/questions/12522491/crash-on-presenting-uiimagepickercontroller-under-ios-6-0)

为了修复该 Bug，我们需要替换 UIImagePickerController 的如下两个方法：

```
- (BOOL)shouldAutorotate;
- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation;
```

我们先实现了一个名为 ImagePickerReplaceMethodsHolder 的类，用于定义替换后的方法和实现。代码如下所示：

```
// ImagePickerReplaceMethodsHolder.h
@interface ImagePickerReplaceMethodsHolder : NSObject

- (BOOL)shouldAutorotate;
- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation;

@end

// ImagePickerReplaceMethodsHolder.m
```

```
@implementation ImagePickerReplaceMethodsHolder

- (BOOL)shouldAutorotate {
    return NO;
}

- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation {
    return UIInterfaceOrientationPortrait;
}

@end
```

然后，我们在调用处，判断当前的 iOS 版本，对于 iOS6.0 及 6.0 之后、iOS6.1 之前的版本，我们将 UIImagePickerController 的有问题的方法替换。具体代码如下：

```
#define SYSTEM_VERSION_GREATER_THAN_OR_EQUAL_TO(v)  ([[[UIDevice currentDevice]
    systemVersion] compare:v options:NSNumericSearch] != NSOrderedAscending)
#define SYSTEM_VERSION_LESS_THAN(v)                 ([[[UIDevice currentDevice]
    systemVersion] compare:v options:NSNumericSearch] == NSOrderedAscending)

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        [self hackForImagePicker];
    });
}

+ (void)hackForImagePicker {
    // fix bug of image picker under iOS 6.0
    // http://stackoverflow.com/questions/12522491/crash-on-presenting-
        uiimagepickercontroller-under-ios-6-0
    if (SYSTEM_VERSION_GREATER_THAN_OR_EQUAL_TO(@"6.0")
        && SYSTEM_VERSION_LESS_THAN(@"6.1")) {
        Method oldMethod1 = class_getInstanceMethod([UIImagePickerController class],
            @selector(shouldAutorotate));
        Method newMethod1 = class_getInstanceMethod([ImagePickerReplaceMethodsHolder
            class], @selector(shouldAutorotate));
        method_setImplementation(oldMethod1, method_getImplementation(newMethod1));

        Method oldMethod2 = class_getInstanceMethod([UIImagePickerController class],
            @selector(preferredInterfaceOrientationForPresentation));
        Method newMethod2 = class_getInstanceMethod([ImagePickerReplaceMethodsHolder
            class], @selector(preferredInterfaceOrientationForPresentation));
        method_setImplementation(oldMethod2, method_getImplementation(newMethod2));
    }
}
```

通过以上代码，我们就针对 iOS 特定版本的有问题的系统库函数打了 Patch，使问题得到解决。

开源界的使用

有少量不明真相的人以为苹果在审核时会拒绝让应用使用以上 API，这其实是对苹果的误解。使用以上 API 是安全的。另外，开源界也对以上方法适当使用，例如：

- 著名的网络库 AFNetworking（https://github.com/AFNetworking/AFNetworking）。AFNetworking 网络库（V1.x 版本）使用了 `class_replaceMethod` 方法（AFHTTPRequestOperation.m 文件第 105 行）。
- Nimbus（https://github.com/jverkoey/nimbus）。Nimbus 是著名的工具类库，它在其 core 模块中提供了 NIRuntimeClassModifications.h 文件，用于提供上述 API 的封装。
- 国内的大众点评 iOS 客户端。该客户端使用了他们自己开发的基于 Wax 修改而来的 WaxPatch（https://github.com/mmin18/WaxPatch），WaxPatch 可以实现通过服务器更新来动态修改客户端的逻辑。而 WaxPatch 主要是修改了 Wax 中的 wax_instance.m 文件，在其中加入了 `class_replaceMethod` 来替换原始实现，从而修改客户端的原有行为。

19.4.3 参考文献

- http://www.sealiesoftware.com/blog/archive/2013/09/24/objc_explain_Non-pointer_isa.html
- http://www.cocoawithlove.com/2010/01/what-is-meta-class-in-objective-c.html

第20章 Tagged Pointer 对象

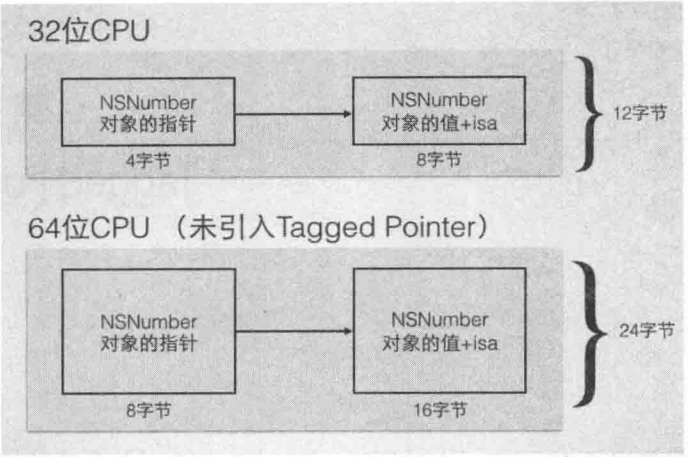
2013 年 9 月，苹果推出了 iPhone5s，与此同时，iPhone5s 配备了首个采用 64 位架构的 A7 双核处理器（http://en.wikipedia.org/wiki/Apple_A7），为了节省内存和提高执行效率，苹果提出了 Tagged Pointer 的概念。对于 64 位程序，引入 Tagged Pointer 后，相关逻辑能减少一半的内存占用，并有 3 倍的访问速度提升，以及 100 倍的创建、销毁速度提升。

本章从 Tagged Pointer 试图解决的问题入手，带领读者理解 Tagged Pointer 的实现细节和优势，最后指出了使用时的注意事项。

20.1 原有系统的问题

我们先看看原有的对象为什么会浪费内存。假设我们要存储一个 NSNumber 对象，其值是一个整数。正常情况下，如果这个整数只是一个 NSInteger 的普通变量，那么它所占用的内存与 CPU 的位数有关，在 32 位 CPU 下占 4 个字节，在 64 位 CPU 下是占 8 个字节的。而指针类型的大小通常也与 CPU 位数相关，一个指针所占用的内存在 32 位 CPU 下为 4 个字节，在 64 位 CPU 下是 8 个字节。

所以，一个普通的 iOS 程序，如果没有 Tagged Pointer 对象，从 32 位机器迁移到 64 位机器中后，虽然逻辑没有任何变化，但这种 NSNumber、NSDate 一类的对象所占用的内存会翻倍，如下图所示。



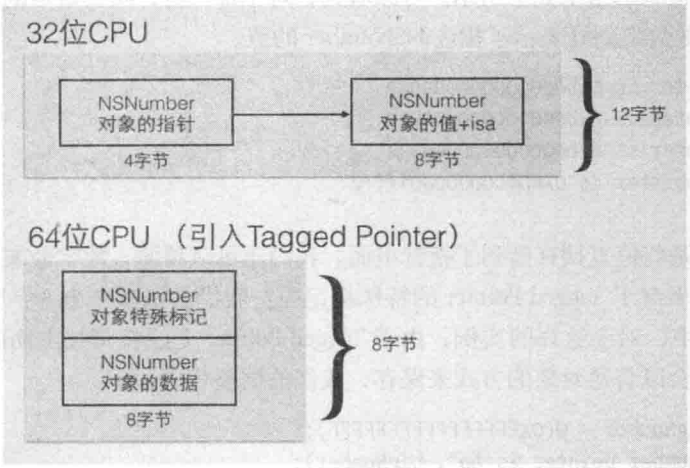
我们再来看看效率上的问题，为了存储和访问一个 NSNumber 对象，我们需要在堆上为其分配内存，另外还要维护它的引用计数，管理它的生命期。这些都给程序增加了额外的逻辑，造成运行效率上的损失。

20.2 Tagged Pointer 介绍

20.2.1 Tagged Pointer

为了改进上面提到的内存占用和效率问题，苹果提出了 Tagged Pointer 对象。由于 NSNumber、NSDate 一类的变量本身的值需要占用的内存大小常常不需要 8 个字节，拿整数来说，4 个字节所能表示的有符号整数就可以达到 20 多亿（注：2^31=2147483648，另外 1 位作为符号位），对于绝大多数情况都是可以处理的。

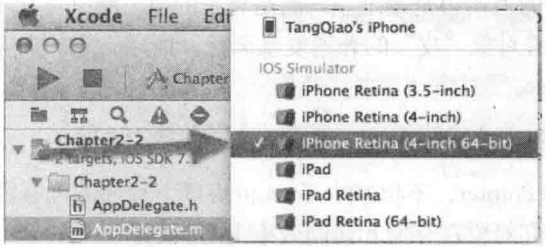
所以我们可以将一个对象的指针拆成两部分，一部分直接保存数据，另一部分作为特殊标记，表示这是一个特别的指针，不指向任何一个地址。所以，引入了 Tagged Pointer 对象之后，64 位 CPU 下 NSNumber 的内存图变成了下面这样。



对此，我们也可以用 Xcode 来验证，代码如下：

```
int main(int argc, char * argv[])
{
    @autoreleasepool {
        NSNumber *number1 = @1;
        NSNumber *number2 = @2;
        NSNumber *number3 = @3;
        NSNumber *numberFFFF = @(0xFFFF);

        NSLog(@"number1 pointer is %p", number1);
        NSLog(@"number2 pointer is %p", number2);
        NSLog(@"number3 pointer is %p", number3);
        NSLog(@"numberffff pointer is %p", numberFFFF);
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
            class]));
    }
}
```

在该代码中，我们将几个 Number 类型的指针的值直接输出。需要注意的是，我们需要将模拟器切换成 64 位的 CPU 来测试，如下图所示。



运行之后，我们得到的结果如下所示。可以看到，对于每一行，除去最后的数字末尾的 2 和开头的 0xb，其他数字刚好表示了相应 NSNumber 的值。

```
number1 pointer is 0xb000000000000012
number2 pointer is 0xb000000000000022
number3 pointer is 0xb000000000000032
numberFFFF pointer is 0xb00000000000ffff2
```

可见，苹果确实是将值直接存储到了指针里面。我们还可以猜测，数字最末尾的 2 和最开头的 0xb 是不是苹果对于 Tagged Pointer 的特殊标记呢？我们尝试放一个 8 字节的长的整数到 NSNumber 实例中，对于这样的实例，由于 Tagged Pointer 无法将其按上面的压缩方式来保存，那么应该就会以普通对象的方式来保存，我们的试验代码如下：

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

运行之后，结果如下，验证了我们的猜测，bigNumber 的地址更像是一个普通的指针地址，和它本身的值看不出任何关系：

```
bigNumber pointer is 0x10921ecc0
```

可见，当 8 字节可以承载用于表示的数值时，系统就会以 Tagged Pointer 的方式生成指针，如果 8 字节承载不了时，则又用以前的方式来生成普通的指针。关于以上关于 Tagged Pointer 的存储细节，我们也可以在这里（https://www.mikeash.com/pyblog/friday-qa-2012-07-27-lets-build-tagged-pointers.html）找到相应的讨论，但是其中关于 Tagged Pointer 的实现细节与我们的试验并不相符，我认为可能是苹果更改了具体的实现细节，并且这不影响我们讨论的 Tagged Pointer 本身的优点。

20.2.2 特点

我们也可以在 WWDC2013 的视频 Session 404 Advanced in Objective-C 中，看到苹果对于 Tagged Pointer 特点的介绍：

1. Tagged Pointer 专门用来存储小的对象，例如 NSNumber 和 NSDate。
2. Tagged Pointer 指针的值不再是地址了，而是真正的值。所以，实际上它不再是一个对象了，它只是一个披着对象“皮”的普通变量而已。所以，它的内存并不存储在堆中，也不需要 malloc 和 free。
3. 在内存读取上有着以前 3 倍的效率，创建时比以前快 106 倍。

因此，苹果引入 Tagged Pointer，不但减少了 64 位机器下程序的内存占用，还提高了运行效率，完美地解决了小内存对象在存储和访问效率上的问题。

20.3 注意事项和实现细节

20.3.1 isa 指针

Tagged Pointer 的引入也带来了问题，即 Tagged Pointer 并不是真正的对象，而是一个伪对象，所以你如果完全把它当成对象来使用，可能会让它“露马脚”。

在上一章中我们写道，所有对象都有 isa 指针，而 Tagged Pointer 其实是没有的，因为它不是真正的对象。因为不是真正的对象，所以如果你直接访问 Tagged Pointer 的 isa 成员的话，在编译时将会有如下警告。

```
-(BOOL)exampleTagUsage:(NSObject *)arg {
    if (((long)arg & 1) == 0) return arg->isa == cachedValue;
    else return [arg isKindOfClass: cachedValue];
}


warning: bitmasking for introspection of Objective-C object pointers is
strongly discouraged [-Wdeprecated-objc-pointer-introspection]
        if (((long)arg & 1) == 0) return arg->isa == cachedValue;
                   ~~~~~~~~~~ ^
error: direct access to Objective-C's isa is deprecated in favor of
object_getClass() [-Werror,-Wdeprecated-objc-isa-usage]
        if (((long)arg & 1) == 0) return arg->isa == cachedValue;
                                          ^
```

对于上面的写法，应该换成相应的方法调用，如 `isKindOfClass` 和 `object_getClass`。只要避免在代码中直接访问对象的 isa 变量，即可避免这个问题。

20.3.2 64 位下的 isa 指针优化

对于 64 位设备，苹果除了引入 Tagged Pointer 来优化小的对象外，对于普通的对象，其 isa 指针也进行了优化和调整。

在 32 位环境下，对象的引用计数都保存在一个外部的表中，每一个对象的 Retain 操作，实际上包括如下 5 个步骤：

1. 获得全局的记录引用计数的 hash 表。
2. 为了线程安全，给该 hash 表加锁。
3. 查找到目标对象的引用计数值。
4. 将该引用计数值加 1，写回 hash 表。
5. 给该 hash 表解锁。

从上面的步骤我们可以看出，为了保证线程安全，对引用计数的增减操作都要先锁定这个表，这从性能上看是非常差的。

而在 64 位环境下，isa 指针也是 64 位，实际作为指针部分只用到的其中 33 位，剩余的 31 位苹果使用了类似 Tagged Pointer 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修改这个指针即可。只有当引用计数超出 19 位，才会将引用计数保存到外部表，而这种情况是很少的，所以这样引用计数的更改效率会更高。

与前面的 5 个步骤对应，在 64 位环境下，新的 Retain 操作包括如下 5 个步骤：

1. 检查 isa 指针上面的标记位，看引用计数是否保存在 isa 变量中，如果不是，则使用以前的步骤，否则执行第 2 步。
2. 检查当前对象是否正在释放，如果是，则不做任何事情。
3. 增加该对象的引用计数，但是并不马上写回到 isa 变量中。
4. 检查增加后的引用计数的值是否能够被 19 位表示，如果不是，则切换成以前的办法，否则执行第 5 步。
5. 进行一个原子的写操作，将 isa 的值写回。

虽然步骤都是 5 步，但是由于没有了全局的加锁操作，所以引用计数的更改更快了。

20.3.3 isa 的 bit 位含义

下表是 isa 指针每个 bit 位具体的用处，按从低位到高位的顺序排列：

bit 位	变量名	意义
1 bit	indexed	0 表示普通的 isa，1 表示 Tagged Pointer
1 bit	has_assoc	表示该对象是否有过 associated 对象，如果没有，在析构释放内存时可以更快
1 bit	has_cxx_dtor	表示该对象是否有 C++ 或 ARC 的析构函数，如果没有，在析构释放内存时可以更快
30 bits	shiftcls	类的指针
9 bits	magic	其值固定为 0xd2，用于在调试时分辨对象是否未完成初使化
1 bit	weakly_referenced	表示该对象是否有过 weak 对象，如果没有，在析构释放内存时可以更快
1 bit	deallocating	表示该对象是否正在析构
1 bit	has_sidetable_rc	表示该对象的引用计数值是否大到无法直接在 isa 中保存
19 bits	extra_rc	表示该对象超过 1 的引用计数值，例如，如果该对象的引用计数是 6，则 extra_rc 的值为 5

20.3.4 总结

苹果引入 Tagged Pointer，给 64 位系统带来了内存的节省和运行效率的提高。Tagged Pointer 通过在其最后一个 bit 位设置一个特殊标记，用于将数据直接保存在指针本身中。因为 Tagged Pointer 并不是真正的对象，我们在使用时需要注意不要直接访问其 isa 变量。

20.3.5 参考文献

https://www.mikeash.com/pyblog/friday-qa-2013-09-27-arm64-and-you.html

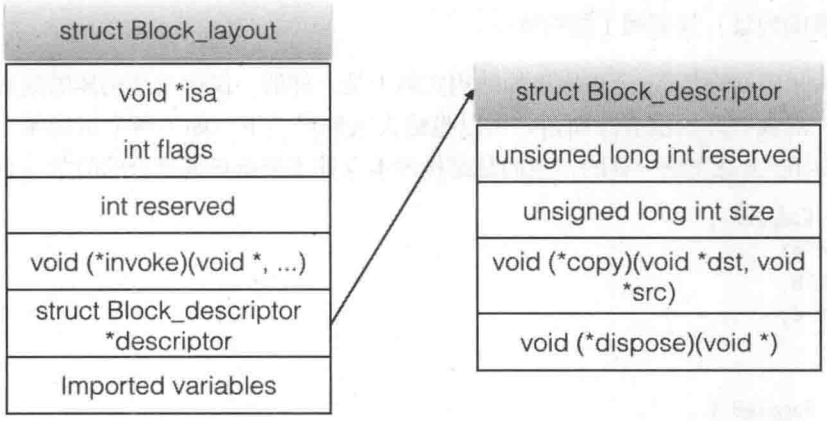
第21章 block对象模型

本章主要介绍 Objective-C 语言的 block 在编译器中的具体实现方式。

21.1 block 的内部数据结构定义

数据结构定义

从苹果的 llvm 项目的开源代码中（https://llvm.org/svn/llvm-project/compiler-rt/tags/Apple/Libcompiler_rt-10/BlocksRuntime/Block_private.h），我们可以得到 block 的数据结构定义，如下图所示。



https://llvm.org/svn/llvm-project/compiler-rt/tags/Apple/Libcompiler_rt-10/BlocksRuntime/Block_private.h

对应的结构体定义如下：

```
struct Block_descriptor {
    unsigned long int reserved;
```

```
    unsigned long int size;
    void (*copy)(void *dst, void *src);
    void (*dispose)(void *);
};

struct Block_layout {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor *descriptor;
    /* Imported variables. */
};
```

通过该图，我们可以知道，一个 block 实例实际上由 6 部分构成：

1. isa 指针，所有对象都有该指针，用于实现对象相关的功能。
2. flags，用于按 bit 位表示一些 block 的附加信息，在本文后面介绍的 block copy 的实现代码中可以看到对该变量的使用。
3. reserved，保留变量。
4. invoke，函数指针，指向具体的 block 实现的函数调用地址。
5. descriptor，表示该 block 的附加描述信息，主要是 size 大小，以及 copy 和 dispose 函数的指针。
6. variables，capture 过来的变量，block 能够访问它外部的局部变量，就是因为将这些变量（或变量的地址）复制到了结构体中。

该数据结构和后面的 clang 分析出来的结构实际上是一样的，仅仅是结构体的嵌套方式不一样。但这一点我一开始没有想明白，所以也给大家解释一下，如下两个结构体 SampleA 和 SampleB 在内存上是完全一样的，原因是结构体本身并不带有任何额外的附加信息。

```
struct SampleA {
    int a;
    int b;
    int c;
};

struct SampleB {
    int a;
    struct Part1 {
        int b;
    };
    struct Part2 {
        int c;
```

```
    };
};
```

在 Objective-C 语言中，一共有 3 种类型的 block：

1. _NSConcreteGlobalBlock，全局的静态 block，不会访问任何外部变量。
2. _NSConcreteStackBlock，保存在栈中的 block，当函数返回时会被销毁。
3. _NSConcreteMallocBlock，保存在堆中的 block，当引用计数为 0 时会被销毁。

我们在下面会分别来查看它们在实现方式上的差别。

21.2 用 clang 分析 block 实现

研究工具：clang

为了研究编译器是如何实现 block 的，我们需要使用 clang。clang 提供一个命令，可以将 Objetive-C 的源码改写成 C 语言，借此可以研究 block 具体的源码实现方式。该命令是：

```
clang -rewrite-objc block.c
```

NSConcreteGlobalBlock 类型的 block 的实现

我们先新建一个名为“block1.c”的源文件：

```
#include <stdio.h>

int main()
{
    ^{ printf("Hello, World!\n"); } ();
    return 0;
}
```

然后在命令行中输入 clang -rewrite-objc block1.c，即可在目录中看到 clang 输出了一个名为“block1.cpp”的文件，该文件就是 block 在 C 语言中的实现。将 block1.cpp 中一些无关的代码去掉，关键代码引用如下：

```
struct __block_impl {
    void *isa;
    int Flags;
    int Reserved;
    void *FuncPtr;
};
```

```
struct __main_block_impl_0 {
    struct __block_impl impl;
    struct __main_block_desc_0* Desc;
    __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int flags=0) {
        impl.isa = &_NSConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
    printf("Hello, World!\n");
}

static struct __main_block_desc_0 {
    size_t reserved;
    size_t Block_size;
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0) };

int main()
{
    (void (*)())&__main_block_impl_0((void *)__main_block_func_0, &
        __main_block_desc_0_DATA) ();
    return 0;
}
```

下面我们就具体看一下是如何实现的。`__main_block_impl_0` 就是该 block 的实现，从中我们可以看出：

1. 一个 block 实际是一个对象，它主要由一个 isa、一个 impl 和一个 descriptor 组成。
2. 由于这里没有开启 ARC。所以这里我们看到 isa 指向的还是 `_NSConcreteStackBlock`。但在开启 ARC 时，block 应该是 `_NSConcreteGlobalBlock` 类。
3. impl 是实际的函数指针，本例中，它指向 `__main_block_func_0`。这里的 impl 相当于之前提到的 invoke 变量，只是 clang 编译器对变量的命名不一样而已。
4. descriptor 是用于描述当前这个 block 的附加信息的，包括结构体的大小，需要 capture 和 dispose 的变量列表等。结构体大小需要保存的原因是，每个 block 会 capture 一些变量，这些变量会加到 `__main_block_impl_0` 这个结构体中，使其体积变大。在该例子中我们还看不到相关 capture 的代码，后面将会看到。

NSConcreteStackBlock 类型的 block 的实现

我们另外新建一个名为“block2.c”的文件，输入以下内容：

```
#include <stdio.h>

int main() {
    int a = 100;
    void (^block2)(void) = ^{
        printf("%d\n", a);
    };
    block2();

    return 0;
}
```

用之前提到的 clang 工具，转换后的关键代码如下：

```
struct __main_block_impl_0 {
    struct __block_impl impl;
    struct __main_block_desc_0* Desc;
    int a;
    __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int _a, int
        flags=0) : a(_a) {
        impl.isa = &_NSConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
    int a = __cself->a; // bound by copy
    printf("%d\n", a);
}

static struct __main_block_desc_0 {
    size_t reserved;
    size_t Block_size;
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};

int main()
{
    int a = 100;
    void (*block2)(void) = (void (*)())&__main_block_impl_0((void *)
        __main_block_func_0, &__main_block_desc_0_DATA, a);
    ((void (*)(__block_impl *))((__block_impl *)block2)->FuncPtr)((__block_impl *)
        block2);

    return 0;
}
```

在本例中，我们可以看到：

1. 本例中，isa 指向 _NSConcreteStackBlock，说明这是一个分配在栈上的实例。
2. __main_block_impl_0 中增加了一个变量 a，在 block 中引用的变量 a 实际是在申明 block 时，被复制到 __main_block_impl_0 结构体中的那个变量 a。因为这样，我们就能理解，在 block 内部修改变量 a 的内容，不会影响外部的实际变量 a。
3. __main_block_impl_0 中由于增加了一个变量 a，所以结构体变大了，该结构体大小被写在了 __main_block_desc_0 中。

我们修改上面的源码，在变量前面增加 __block 关键字：

```
#include <stdio.h>

int main()
{
    __block int i = 1024;
    void (^block1)(void) = ^{
        printf("%d\n", i);
        i = 1023;
    };
    block1();
    return 0;
}
```

生成的关键代码如下，可以看到，差异相当大：

```
struct __Block_byref_i_0 {
    void *__isa;
    __Block_byref_i_0 *__forwarding;
    int __flags;
    int __size;
    int i;
};

struct __main_block_impl_0 {
    struct __block_impl impl;
    struct __main_block_desc_0* Desc;
    __Block_byref_i_0 *i; // by ref
    __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc,
         __Block_byref_i_0 *_i, int flags=0) : i(_i->__forwarding) {
        impl.isa = &_NSConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};
```

```
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
    __Block_byref_i_0 *i = __cself->i; // bound by ref

    printf("%d\n", (i->__forwarding->i));
    (i->__forwarding->i) = 1023;
}

static void __main_block_copy_0(struct __main_block_impl_0*dst, struct
    __main_block_impl_0*src) {_Block_object_assign((void*)&dst->i, (void*)src->i, 8
    /*BLOCK_FIELD_IS_BYREF*/);}

static void __main_block_dispose_0(struct __main_block_impl_0*src) {
    _Block_object_dispose((void*)src->i, 8/*BLOCK_FIELD_IS_BYREF*/);}

static struct __main_block_desc_0 {
    size_t reserved;
    size_t Block_size;
    void (*copy)(struct __main_block_impl_0*, struct __main_block_impl_0*);
    void (*dispose)(struct __main_block_impl_0*);
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0),
    __main_block_copy_0, __main_block_dispose_0};

int main()
{
    __attribute__((__blocks__(byref))) __Block_byref_i_0 i = {(void*)0,(
        __Block_byref_i_0 *)&i, 0, sizeof(__Block_byref_i_0), 1024};
    void (*block1)(void) = (void (*)())&__main_block_impl_0((void *)
        __main_block_func_0, &__main_block_desc_0_DATA, (__Block_byref_i_0 *)&i,
        570425344);
    ((void (*)(__block_impl *))((__block_impl *)block1)->FuncPtr)((__block_impl *)
        block1);
    return 0;
}
```

从代码中我们可以看到：

1. 源码中增加一个名为 __Block_byref_i_0 的结构体，用来保存我们要 capture 并且修改的变量 i。
2. __main_block_impl_0 中引用的是 __Block_byref_i_0 的结构体指针，这样就可以起到修改外部变量的作用。
3. __Block_byref_i_0 结构体中带有 isa，说明它也是一个对象。
4. 我们需要负责 __Block_byref_i_0 结构体相关的内存管理，所以 __main_block_desc_0 中增加了 copy 和 dispose 函数指针，用于在调用前后修改相应变量的引用计数。

21.2.1 NSConcreteMallocBlock 类型的 block 的实现

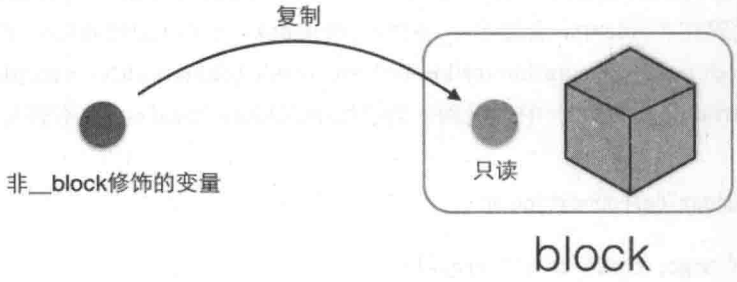
NSConcreteMallocBlock 类型的 block 通常不会在源码中直接出现，只有当一个 block 被调用其 copy 方法的时候，系统才会将这个 block 复制到堆中，从而产生 NSConcreteMallocBlock 类型的 block。以下是一个 block 被 copy 时的示例代码（来自http://www.galloway.me.uk/2013/05/a-look-inside-blocks-episode-3-block-copy/），可以看到，在第 8 步，目标的 block 类型被修改为 _NSConcreteMallocBlock。

```
static void *_Block_copy_internal(const void *arg, const int flags) {
    struct Block_layout *aBlock;
    const bool wantsOne = (WANTS_ONE & flags) == WANTS_ONE;

    // 1
    if (!arg) return NULL;

    // 2
    aBlock = (struct Block_layout *)arg;

    // 3
    if (aBlock->flags & BLOCK_NEEDS_FREE) {
        // latches on high
        latching_incr_int(&aBlock->flags);
        return aBlock;
    }

    // 4
    else if (aBlock->flags & BLOCK_IS_GLOBAL) {
        return aBlock;
    }

    // 5
    struct Block_layout *result = malloc(aBlock->descriptor->size);
    if (!result) return (void *)0;

    // 6
    memmove(result, aBlock, aBlock->descriptor->size); // bitcopy first

    // 7
    result->flags &= ~(BLOCK_REFCOUNT_MASK);    // XXX not needed
    result->flags |= BLOCK_NEEDS_FREE | 1;

    // 8
    result->isa = _NSConcreteMallocBlock;

    // 9
```

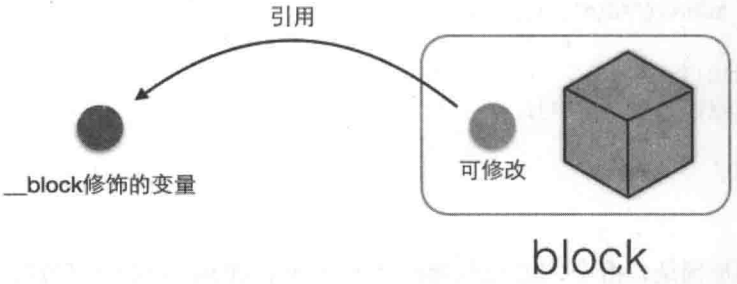
```
    if (result->flags & BLOCK_HAS_COPY_DISPOSE) {
        (*aBlock->descriptor->copy)(result, aBlock); // do fixup
    }

    return result;
}
```

21.2.2 变量的复制

对于 block 外的变量引用，block 默认是将其复制到其数据结构中来实现访问的。如果这个对象是一个引用类型，则 block 会将其引用计数加 1，如下图所示。



对于用 __block 修饰的外部变量引用，block 是复制其引用地址来实现访问的，如下图所示。



21.3 注意事项

21.3.1 避免循环引用

由于 block 会复制外部的变量，所以如果不注意，会比较容易造成循环引用。对于这种问题，需要将引用的一方变成 weak 的，从而避免循环引用，以下是具体使用时避免循环引用的例

子：

```
MyViewController *__weak weakSelf = self;
self.tableViewAgent.selectedCellBlock = ^(UITableView *tableView, NSIndexPath *
    indexPath) {
    [weakSelf someMethod];
};
```

21.3.2 ARC 对 block 类型的影响

在 ARC 开启的情况下，将只会有 NSConcreteGlobalBlock 和 NSConcreteMallocBlock 类型的 block。

原本的 NSConcreteStackBlock 的 block 会被 NSConcreteMallocBlock 类型的 block 替代。证明方式是以下代码在 Xcode 中，会输出 <__NSMallocBlock__: 0x100109960>。在苹果的官方文档（http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html）中也提到，当把栈中的 block 返回时，就不需要调用 copy 方法了。

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

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        int i = 1024;
        void (^block1)(void) = ^{
            printf("%d\n", i);
        };
        block1();
        NSLog(@"%@", block1);
    }
    return 0;
}
```

我认为这么做的原因是，由于 ARC 已经能很好地处理对象的生命周期的管理，这样所有对象都放到堆上管理，对于编译器实现来说，会比较方便。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

专家力荐

从2008年苹果发布第一代iOS SDK至今，已经有六个年头，整个移动开发的热潮也持续五六年了，但iOS开发者仍旧非常短缺，特别是优秀的iOS开发者。我一直在说，每一个行业里面的开发者，都应该有所追求，要变成行业的佼佼者，这不仅是因为我们都想有更高的收入，更是因为人本身就该有追求。唐巧就是我们这个行业中非常优秀的开发者，但是更值得称赞的是，他提供了这么一本书，让我们也有机会变成更专业的、更优秀的开发者，善莫大焉。

OurCoders站长，资深iOS开发者 tinyfool

作为一本面向中高级iOS开发者的书籍，《iOS开发进阶》一书汇总了很多进阶开发时所常用和必备的知识。阅读本书不但能帮你借助各类工具大幅提升开发效率，也能让你加深对iOS及Objective-C背后机制的理解。如果你想在iOS开发的道路上百尺竿头更进一步的话，这本凝聚了作者多年开发实战经验的进阶图书将是你不可错过的伙伴！

objccn.io创始人、《Swifter》作者 王巍（onevcat）

唐巧是国内较早从事iOS开发的资深工程师，并在个人博客、InfoQ、微信公众号等平台持续更新技术文章，我们也曾邀请他给微信iOS团队分享相关经验。目前市面上缺乏对Objective-C高级特性、Xcode插件等做全面剖析和整理的图书，《iOS开发进阶》很好地填充了此处的空白，相信对广大iOS开发者会有很大的帮助。

微信iOS客户端团队负责人 lylechen

本人有幸曾和作者唐巧共事，当时他还负责微博后台研发。四年后唐巧转做iOS开发，并成为国内资深专家。作者的工作经历本身就是一次完美的“iOS开发进阶”。如果你刚刚提交了一款App Store应用，那本书第一部分一定会有很多高效的工具可以帮你；如果你已有一两年iOS开发经验，那你一定会对第二部分的各种实战技巧相见恨晚；本书最后部分的底层原理是你成为资深工程师的必备课程。

网易新闻客户端技术负责人 王聪（robaggio）

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：徐津平
封面设计：李 玲

上架建议：iOS开发

ISBN 978-7-121-24745-3
9 787121 247453 >
定价：65.00元