# 文档前言

## 基本描述

俗话说的好：“好记性，不如烂笔头”。说明的一个道理就是纪录的重要性，哪怕是一个记忆天才，纪录在其他地方，可以随时看到也是非常重要的。何况我是如此一个平凡的人啊！谦虚学习，踏踏实实，不能好高骛远，天道酬勤才是王道。

本文档旨在纪录开发过程中遇到的一些重要点、难点，也能帮助我纪录我的成长过程。

少波，好好纪录，这是很重要的一个事情，不要觉得我写了一些知道的东西，没什么价值，磨刀不误砍柴工。

## 内容概括

本文包括如下内容：iOS开发、设计模式、数据结构、java服务器开发、前段开发等，若有新技术的添加会及时的添加到文档中。

IOS开发部分

# ****一键自动生成 IOS 所有尺寸的 ICON 【效率工具】****http://www.ui.cn/detail/40848.html

# 我的计划项目：

### 关于运动的视频app

### ~~电子商务类~~

### 社交聊天类

# 开发工具

前言

第三方库是现在的程序员离不开的东西 不光是APP开发 基本上所有的商业项目 都会或多或少的使用到第三方库

Github上Star>100的开源库数量如下

可以看到JS以绝对的优势排名第一 一定程度上也说明了JS在这几年为什么发展得这么迅速 不会点JS都不好意思说自己是码农 不过现在JS圈的造轮子浪潮也是愈演愈烈了 自己不写个框架都不好意思打招呼

OC排名第四 相信这个排名还会上升 Swift暂时还挤不进前十 不过超过OC 也是指日可待(毕竟亲儿子)

Xcode基本是Mac/iOS程序员的必备工具(AppCode我没用过 不知道有多少人用?) 但是能说有多好用..这就仁者见仁了(跟Visual Studio是没得比了) 不过在用了插件以后 倒也能让生产力提升一大截

接下来我会介绍一些我本人常用的第三方库和插件 也许不多 但是一定是久经考验 用了一定不会错

介绍

第三方库

CocoaPod

CocoaPod并不是iOS上的第三方库 而是大名鼎鼎的第三方库的管理工具

在CocoaPod没有出现之前 第三方库的管理是非常痛苦的 尤其是一些大型的库(比如nimbus) 每次对库进行更新 都可能会非常的痛苦

CocoaPod的出现解决了这些问题 以Framework的方式引入第三方库 极大的节约了集成的时间 而且通吃Objective-C和Swift(Swift上的Cathatage我没有实际用过 但是它的那种集成方式还是比CocoaPod麻烦点)

iCarousel

在iOS 4.x的时代(我也是从4.0开始接触iOS开发的) Coverflow的效果可谓是风靡一时 初出茅庐的我当年对如何实现几乎是束手无策(当年所有的电影资讯类的APP 如布丁爱生活等 都需要实现Coverflow选片的效果 碰巧我也要做一个) iCarousel的出现替我解决了一个大的难题

从此iCarousel成为了我每个项目的必备良药(也是我最喜欢的库) 不管是Coverflow还是轮播广告 都能轻描淡写的搞定 其内置的十来种显示类型 基本可以解决90%的UI需求 而其强大的自定义功能 则可以解决剩余的10% :)

不熟悉的朋友可以尝试一下其精美的demo(pod try iCarousel) 所有的功能都在demo中一览无余

作者nicklockwood也是一个高产的大神 同时维护着数量众多且优质的开源库(比如iVersion iRate) 大家不妨去看一看 淘一淘

AFNetworking/Alamofire

HTTP框架的龙头老大 当年在与ASIHTTPRequest的竞争中笑到了最后(当然也是因为ASIHTTPRequest的作者不维护了 不过国内很多公司 因为历史原因都在自行维护) 由于及时维护和拥抱语言的新特性 迅速被大家所接受和喜爱

相信每个人都用过 这里就不赘述了

MKNetworkKit

如果说AFNetworking是老大 那么MKNetworkKit可以说是小弟了 不过也因为比AFNetworking轻量的原因 也获得了许多开发者的青睐

当年因为ASIHTTPRequest停止维护了 在机缘巧合之下 我在AFNetworking和MKNetworkKit之间选择了后者 并在很多项目中进行了使用

不过也许是因为AFNetworking发展得更快更迅速 而作者本人是单兵作战的原因 作者也基本放弃了更新(版本号在停留在0.8x) 十分可惜

SDWebImage

图片异步下载和缓存管理的集大成者 UITableView的黄金搭档 其代码被开发者研究分享过无数次喵大的Kingfisher(可以说是Swift版的SDWebImage)也是深受其影响

异步下载AFNetworking和MKNetworkKit都有实现 数据缓存也有FastImageCache与TMCache等 但是将其融会贯通的 唯有SDWebImage了

除了简单的使用sd\_setImageWithURL之外 SDImageCache也是可以独立使用的 功能也相当之强大

ZXingObjC

zxing是Google出品的二维码扫描组件 原本是Java编写的 现在也有了各种语言的移植版 而ZXingObjC就是其中之一

zxing支持各种主流的一维码二维码扫描 简单易用 小厂如果要实现二维码扫描这个功能 基本都会选择zxing

不过iOS7已内置了二维码摄像头扫描 而iOS8也已内置了二维码静态图扫描 以后可能再也不需要用到ZXing了 :(

MBProgressHUD

简单易用且稳定的HUD组件 半透明黑底白字的风格也经久不衰 loading提示的最佳选择

Masonry/SnapKit

最好用的Autolayout手写库(Cartography也不错啦 但是还是用不惯) 帮助我轻易的跨越了Autolayout这道坎

我也曾多次在文章中提到过关于Masonry的使用方法 如果还没有用过的朋友 不妨看一看

pop

Facebook的工程师一直是神一般的存在 对开源社区的贡献也是巨大的 极大的推动了各种变成语言的发展 比如HipHop之于PHP react之于JavaScript pop之于Objective-C等等

不管是HipHop react Facebook的工程师总是抱着颠覆的态度来开源 pop也不例外 这点之前我也简单介绍过 而以pop为基础的paper一发布就震惊整个APP届 在这点上pop也是厥功甚伟

pop对自定义动画也支持得很好 我也以pop的自定义动画为基础写过MMTweanAnimation

ReactiveCocoa

说起来惭愧 大名鼎鼎的RAC 我只使用了点皮毛(只拿来做输入验证了)

暂时还没有进行深层次的使用 对RAC的理解也停留在表面阶段 不过这篇文章介绍得非常详细 值得一看

GPUImage

如果你要做图像(照片或者视频)的相关处理 或者只是简单的想做个像Camera360一样的拍照滤镜 那么你一定要研究一下GPUImage

如它的名字所述 GPUImage是基于GPU的图像处理框架 我们都知道 GPU是提升性能的关键 这也就是GPUImage如此重要 如此受欢迎的原因

Lumberjack

log系统是每个项目都应该有的东西 而Lumberjack是log系统中的翘楚

你可以简单的把它当成NSLog的替代品(简单来说 Lumberjack比NSLog速度更快) 或者根据你的需要来打造一个更强大的日志系统

NSLogger

从名字可以看出 NSLogger也是一个log系统 其特点是附带了一个功能强大的Desktop Viewer 可以让你方便的查看APP产生的日志(支持分级筛选等等 甚至可以直接log一张图片)

AwesomeMenu

当年横空出世的Path 其优美的设计 精彩的动画 不知让多少人嘴巴都合不拢 而最赞的 就是它的弹出菜单 一时成为了每个APP争相模仿的对象

有了AwesomeMenu 你可以轻易的实现它

MMDrawerController

普通的侧滑菜单 用MMDrawerController就搞定了

realm

作为数据存储的一等公民 CoreData的地位不言而喻 不过也因为使用起来不够方便 才会出现MagicalRecord这种辅助类 甚至fmdb这种基于纯sqlite的库

而realm以挑战者的身份闪亮登场 不仅读取性能更快(据说数倍于CoraData) 接口简单易用(以对象的形式使用数据 这点和leancloud的思路很相似) 并且还跨平台(iOS/Android均可使用 OC/Swift/Java都支持)

如果你习惯使用Mantle之类的Modal转换的话Realm-JSON肯定能讨你欢心

作为YCombinator的孵化项目 其质量还是能得以保证的 至少我试用下来 确实给我以很大的惊喜 不管是API的设计 还是数据对象的定义 就连数据库的版本升级 都非常的方便)

Xcode插件

Alcatraz

与CocoaPod类似 Alcatraz是Xcode的插件管理器 能够让你方便的管理Xcode的插件(不仅可以管理插件 还可以管理主题等等)

FuzzyAutocomplete

如果只让我选一个插件留下 那必须是FuzzyAutocomplete 强大的模糊匹配输入 让你写代码的时候再也不用费脑子去记住名字那么长的对象或者函数名了 好用到让你想哭

XAlign

作为有洁癖的码农 看到不对齐的代码一定是不能忍的 XAlign可以轻松解决你的烦恼

VVDocumenter-Xcode

喵大的又一力作 能够识别当前函数的参数和返回类型 帮助你快速编写符合规范的注释(目前是以Javadoc为标准)

deriveddata-exterminator

如果你老是遇到Xcode抽风 提示你要因为某个原因要删除某个目录下的文件 否则编译不过 那你一定会被这个插件感动 因为说明了遇到这个事的人不只你一个

Xcode-Quick-Localization

多语言在iOS开发中一直不是很方便 有了它 你可以省不少事

Backlight-for-XCode

就如果Xcode默认的80个字的分页提示一样 高亮当前正在编辑的一行 也是一种友好的提示 喜不喜欢也就因人而异了

# 行多多上班坚持任务纪录

## 看第三方代码

# ios代码书写规范

## 他人代码规范1

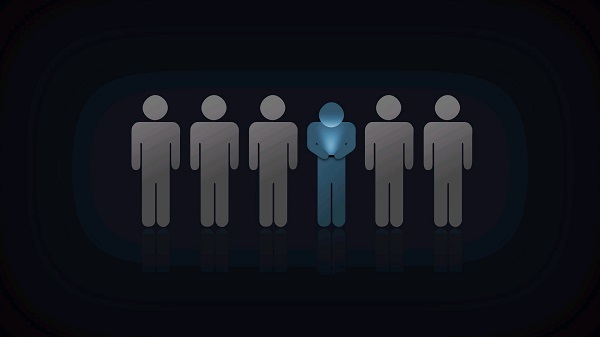
<http://www.cocoachina.com/ios/20160324/15727.html>

iOS开发编码建议与编程经验（持续更新中）

2016-03-24 06:02 编辑： suiling 分类：[iOS开发](http://www.cocoachina.com/ios/) 来源：[乞力马扎罗的雪的博客](http://blog.csdn.net/chenyufeng1991/article/details/50261901)

50 11427

[iOS开发](http://www.cocoachina.com/cms/tags.php?/iOS%E5%BC%80%E5%8F%91/)[编程经验](http://www.cocoachina.com/cms/tags.php?/%E7%BC%96%E7%A8%8B%E7%BB%8F%E9%AA%8C/)[编码建议](http://www.cocoachina.com/cms/tags.php?/%E7%BC%96%E7%A0%81%E5%BB%BA%E8%AE%AE/)

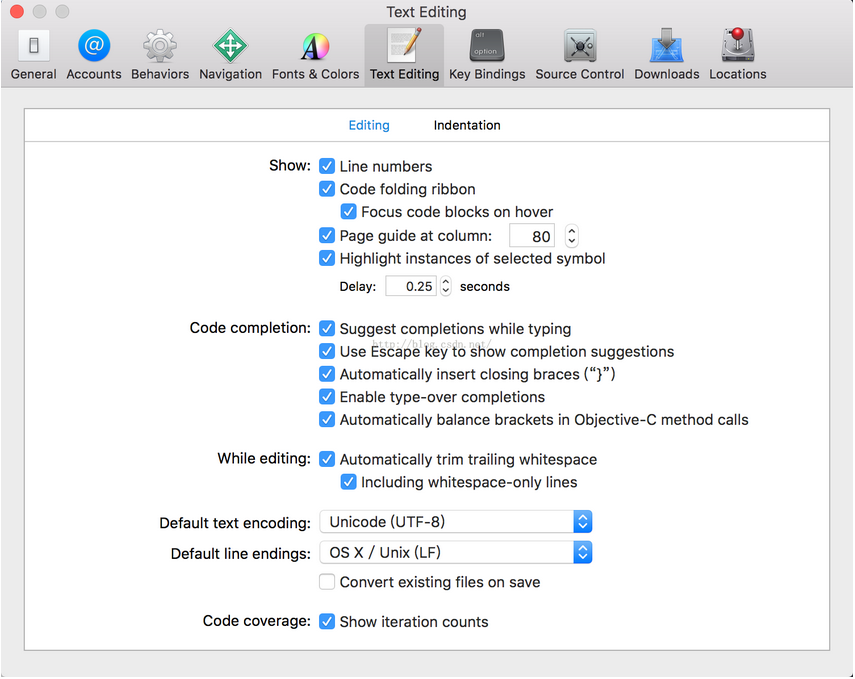


作者：乞力马扎罗的雪（[GitHub](https://github.com/chenyufeng1991)） [原文](http://blog.csdn.net/chenyufeng1991/article/details/50261901)

在开发过程中，我们不仅要去看别人的代码，也要让别人看我们的代码。那么，有一个良好的编码习惯将会非常重要。下面将会罗列使用Objective-C来开发iOS的编码建议。

【1】任意函数长度不得超过50行。（其实很容易就超过50行，这就要考虑代码抽取了。）

【2】任意行代码不能超过80字符。（其实也很容易超过80字符，可以考虑多行显示，比如有多个参数时，可以每个参数放一行。）可以在Xcode中设置超过80个字符的提醒，选中“Page guide at column”.设置完之后就会在代码80个字符处有一条竖线。



【3】在每个方法的定义前留白一行，也就是在方法和方法之间留空一行。

【4】功能相近的方法要放在一起，并推荐使用#pragma mark - \*\*\*来导航代码，切分代码块。这样可以方便函数的查找。并且可以使用快捷键control+6 来快速查找方法的位置。

【5】二元运算符和参数之间要有一个空格，如赋值号=左右各留一个空格。

|  |  |
| --- | --- |
| 1 | self.myString = @"235423rew523452345"; |

【6】一元运算符和参数之间不放置空格，比如！非运算符，&按位与，|按位或。

|  |  |
| --- | --- |
| 1  2 | BOOL isOpen = **true**;  BOOL isClose = !isOpen; |

【7】强制类型转换和参数之间不放置空格。

|  |  |
| --- | --- |
| 1 | NSString \*str3 = (NSString\*)self.myString; |

【8】长的变量值应该拆分为多行。尤其体现在使用数组或者字典。以下也分别是快速声明数组@[]和字典@{}的方法。

|  |  |
| --- | --- |
| 1  2  3  4  5 | NSArray \*array = @[@"111",                       @"2222222222",                       @"3333333",                       @"wwwwwwwwwwww"                       ]; |
| 1  2  3  4 | @"age":@"20",                         @"gender":@"female",                         @"isMarried":@"false"                         }; |

【9】尽量使用有意义的名字命名，拒绝使用i,j等无意义字符命名。类的命名首字母大写，其他变量的命名首字符小写，并使用驼峰式分割单词。

【10】尽量减少在代码中直接使用数字常量，而使用宏定义等方式。如：MAX\_NUMBER\_PHONE替代8等等。这样我们搜索也比较方便。

【11】尽量减少代码中的重复计算，比如代码中多处要使用屏幕宽度，然后计算：[[UIScreenmainScreen] bounds].size.width ,很多次，闲得很繁琐，代码也冗长。不如直接宏定义：

|  |  |
| --- | --- |
| 1 | #define SCREEN\_WIDTH ([[UIScreen mainScreen] bounds].size.width) |

【12】合理使用约定俗成的缩略词：

* alloc:分配；
* alt：轮流，交替；
* app:应用程序；
* calc：计算；
* dealloc：销毁、析构；
* func：函数、方法；
* horiz：水平的；
* info：信息；
* init：初始化；
* max：最大的；
* min：最小的；
* msg：消息；
* nib：Interface Builder；
* rect：矩形；
* temp：暂时的；
* vert：垂直的；

【13】宏定义全部字母大写。

【14】函数长度不要超过50行，小函数比大函数可读性更强。函数的参数不宜过多，零元函数最好，一元函数也不错，高于三元的函数虚重构。

【15】合理范围内使用链式编程:

|  |  |
| --- | --- |
| 1 | NSString \*myName = [[NSString alloc] init]; |

但是嵌套不宜超过3层，超过3层需进行重构。

【16】函数调用时所有参数在同一行。如果参数过多，则可以每行一个参数，每个参数以冒号对齐。

【17】对传入参数的保护或者说是否为空的判断，尽量不要使用if(!obj),而使用NSAssert断言来处理。NSAssert是系统定义的宏。

|  |  |
| --- | --- |
| 1 | NSAssert(myName != nil, @"myName参数为空"); |

* 如果条件判断为真，则程序继续执行。
* 如果判断条件为假，则抛出异常，异常内容为后面定义的字符串。

【18】方法参数名前一般使用"an","the","new"来进行修饰。如：

|  |  |
| --- | --- |
| 1 | -(void)setPersonInfo:(NSString\*)theID theName:(NSString\*)theName theAge:(NSInteger\*)theAge |

【19】if-else超过四层的时候，就要考虑重构，多层的if-else结构很难维护。

【20】当需要一定条件才执行某项操作时，最左边的应该是最重要的代码，不要将最重要的代码内嵌到if中。如良好的风格是：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | - (void) someMethod {  **if**(![someOther boolValue]) {  **return**;    }  //最重要的代码写在这里；  } |

反面教材:

|  |  |
| --- | --- |
| 1  2  3  4  5 | - (void) someMethod {  **if**([someOther boolValue]) {       //重要代码；    }  } |

【21】所有的逻辑块都使用{}花括号包围，就算只是一行代码。

【22】明确指定构造函数，并有适当的注释。

【23】不要在init方法中把变量或者说属性初始化为0或者nil，因为没有必要。

【24】UIView的子类初始化的时候，不要进行任何的布局操作。布局操作应该在layoutSubviews里面做；需要重新布局的时候调用setNeedsLayout，而不要直接调用layoutSubviews。

【25】保持公共API简单，也就是保持.h文件简单。放在.h中声明的函数都是会被公开的，如果根本就没必要对其他类公开，再不要在.h中声明。OC中的方法都是公有方法，没有私有方法一说。

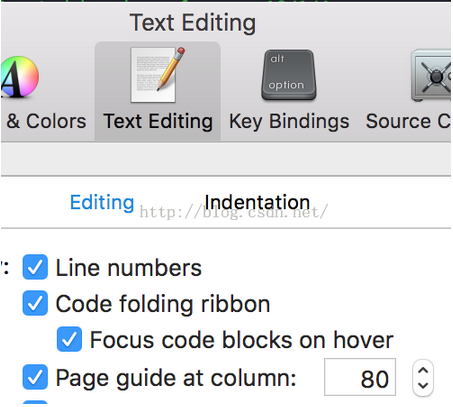
【26】一个文件只实现一个类。同一个文件中不要有多个类。

【27】Protocol单独用一个文件来创建，尽量不要与相关类混在一个文件中。

【28】在类定义中使用到自己定义类的时候，尽量不要在头文件中引入自己定义类的头文件，使用@class替代。而在实现文件中引入头文件。

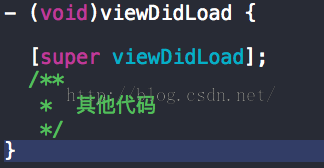
【29】布局时尽量使用相对布局，比如使用子View在父View中的相对位置。

【30】代码折叠，这个可能是关于开发效率的，我也写在编码规范中，因为这个很有用。Xcode7默认没有开启代码折叠，如果你的方法体行数很长，看起来会很不方便，此时你就可以把方法“收起来”，一个类中的结构就会很清晰。开启方法如下：Xcode菜单-->Preferences-->Text Editing-->勾选Code folding ribbon.如图：



【31】推荐方法的第一个花括号直接跟在方法体后，而不是另起一行，这样可以减少代码行。

【32】推荐方法体中的第一行留空，最后一行不留空，这样一个方法就会比较清晰。如图：



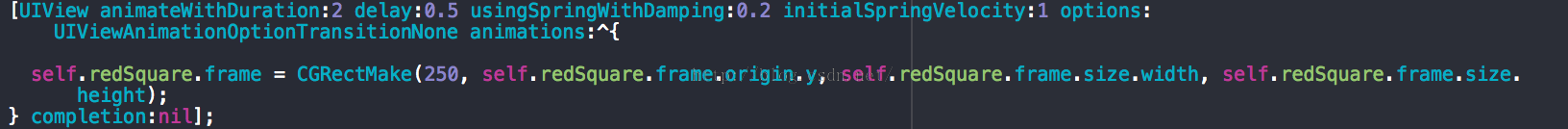
但是如果该花括号里面又是一个if，for之类的带花括号的语句块，那么上述的第一行可以不留空。

同样，如果花括号内第一行是注释的话，第一行也可以不留空。注释也起到了分隔代码的作用，看起来比较清晰。

再者，如果花括号内只有一行代码，第一行可以不留空。

【33】block中第一行也要留空，同方法体中的第一行留空，使代码清晰。

【34】代表类方法和实例方法的"+"加号,"-"减号后需要一个空格。这是一个非常小的细节，系统默认的方法都是这样的，我们自己声明或者实现一个方法的时候也需要这样：



【35】这一条有点像编程经验了，就是为解决某个问题估算时间。比如要开发某个功能、调试某个bug、给自己一个时间限制，如果在这期间不能解决问题，那么就去寻求帮助。这既是给自己一个压力，也为了不浪费时间。虽然，这一条其实很难做到，我往往由于不甘心而无限拖延时间去解决问题。

【36】由于提到编程经验，就不得不提到版本控制。务必去学会SVN或者Git，就算你是独立开发，也要学会控制自己的代码，当然，你要经常备份你的代码。

## 枚举的写法：

学习官方写法

# CocoaPods安装和使用

## 检查是否有ruby环境

检查一下是否安装正确 ：输入命令 rvm -v ，如果正确会列出RVM版本 如下所示：

$ rvm –v

## 步骤1 － 安装 RVM

$ curl -L https://get.rvm.io | bash -s stable

然后，载入 RVM 环境

　　　　$ source ~/.rvm/scripts/rvm

检查一下是否安装正确 ：输入命令 rvm -v

$ rvm –v

## 步骤2 － 用 RVM 安装 Ruby 环境

输入命令　　$ rvm install 2.0.0

　　　　$ rvm install 2.0.0

同样继续等待漫长的下载，编译过程，完成以后就安装好了。

## 步骤3 － 设置 Ruby 版本

$ rvm 2.0.0 --default

下载完成后使用如下：ruby –v 检查是否安装成功，再使用gem –v 查看gem环境是否安装成功。

## 步骤4 –替换ruby源

这有可能是因为Ruby的默认源使用的是cocoapods.org，国内访问这个网址有时候会有问题，网上的一种解决方案是将远替换成淘宝的，替换方式如下：

　　 $gem source -r https://rubygems.org/

　　　　$ gem source -a https://ruby.taobao.org

lihuifengdeMacBook-Pro:~ lihuifeng$ gem source -r https://rubygems.org/

https://rubygems.org/ removed from sources

lihuifengdeMacBook-Pro:~ lihuifeng$ gem source -a https://ruby.taobao.org

https://ruby.taobao.org added to sources

要想验证是否替换成功了，可以执行：

　　　　$ gem sources -l

# MVVM架构模式

[[bjc中国](http://objccn.io/) **objc中国**](http://objccn.io/) [**书籍**](https://store.objccn.io/products/)

# MVVM 介绍

 朱宏旭  10 Jun 2014

###### 分享文章

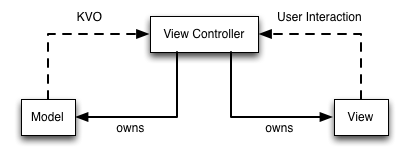
我于 2011 年在 500px 找到自己的第一份 iOS 开发工作。虽然我已经在大学里做了好几年 iOS 外包开发，但这才是我的一个真正的 iOS 开发工作。我被作为唯一的 iOS 开发者被招聘去实现拥有漂亮设计的 iPad 应用。在短短七周里，我们就发布了 1.0 并持续迭代，添加了更多特性，但从本质上，代码库也变得更加复杂了。

有时我感觉就像我不知道在做什么。虽然我知道自己的设计模式——就像任何好的编程人员那样 —— 但我太接近我在做的产品以至于不能客观地衡量我的架构决策的有效性。当队伍中来了另外一位开发者时，我意识到我们陷入困境了。

从没听过 MVC ？有人称之为 Massive View Controller（重量级视图控制器），这就是我们那时候的感觉。我不打算介绍令人汗颜的细节，但说实在的，如果我不得不再次重来一次，我绝对会做出不同的决策。

我会修改一个关键架构，并将其带入我从那时起就在开发的各种应用，即使用一种叫做 Model-View-ViewModel 的架构替换 Model-View-Controller。

所以，MVVM 到底是什么？与其专注于说明 MVVM 的来历，不如让我们看一个典型的 iOS 是如何构建的，并从那里了解 MVVM：



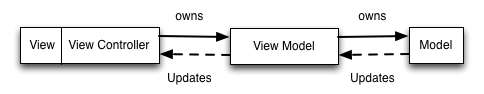
我们看到的是一个典型的 MVC 设置。Model 呈现数据，View 呈现用户界面，而 View Controller 调节它两者之间的交互。Cool！

稍微考虑一下，虽然 View 和 View Controller 是技术上不同的组件，但它们几乎总是手牵手在一起，成对的。你什么时候看到一个 View 能够与不同 View Controller 配对？或者反过来？所以，为什么不正规化它们的连接呢？



这更准确地描述了你可能已经编写的 MVC 代码。但它并没有做太多事情来解决 iOS 应用中日益增长的重量级视图控制器的问题。在典型的 MVC 应用里，许多逻辑被放在 View Controller 里。它们中的一些确实属于 View Controller，但更多的是所谓的“表示逻辑（presentation logic）”，以 MVVM 属术语来说，就是那些将 Model 数据转换为 View 可以呈现的东西的事情，例如将一个**NSDate** 转换为一个格式化过的 **NSString**。

我们的图解里缺少某些东西，那些使我们可以把所有表示逻辑放进去的东西。我们打算将其称为 “View Model” —— 它位于 View/Controller 与 Model 之间：



看起好多了！这个图解准确地描述了什么是 MVVM：一个 MVC 的增强版，我们正式连接了视图和控制器，并将表示逻辑从 Controller 移出放到一个新的对象里，即 View Model。MVVM 听起来很复杂，但它本质上就是一个精心优化的 MVC 架构，而 MVC 你早已熟悉。

现在我们知道了什么是 MVVM，但为什么我们会想要去使用它呢？在 iOS 上使用 MVVM 的动机，对我来说，无论如何，就是它能减少 View Controller 的复杂性并使得表示逻辑更易于测试。通过一些例子，我们将看到它如何达到这些目标。

此处有三个重点是我希望你看完本文能带走的：

* MVVM 可以兼容你当下使用的 MVC 架构。
* MVVM 增加你的应用的可测试性。
* MVVM 配合一个绑定机制效果最好。

如我们之前所见，MVVM 基本上就是 MVC 的改进版，所以很容易就能看到它如何被整合到现有使用典型 MVC 架构的应用中。让我们看一个简单的 **Person** Model 以及相应的 View Controller：

**@interface Person : NSObject**

**- (instancetype)initwithSalutation:(NSString \*)salutation firstName:(NSString \*)firstName lastName:(NSString \*)lastName birthdate:(NSDate \*)birthdate;**

**@property (nonatomic, readonly) NSString \*salutation;**

**@property (nonatomic, readonly) NSString \*firstName;**

**@property (nonatomic, readonly) NSString \*lastName;**

**@property (nonatomic, readonly) NSDate \*birthdate;**

**@end**

Cool！现在我们假设我们有一个 **PersonViewController** ，在 **viewDidLoad** 里，只需要基于它的 **model** 属性设置一些 Label 即可。

**- (void)viewDidLoad {**

**[super viewDidLoad];**

**if (self.model.salutation.length > 0) {**

**self.nameLabel.text = [NSString stringWithFormat:@"%@ %@ %@", self.model.salutation, self.model.firstName, self.model.lastName];**

**} else {**

**self.nameLabel.text = [NSString stringWithFormat:@"%@ %@", self.model.firstName, self.model.lastName];**

**}**

**NSDateFormatter \*dateFormatter = [[NSDateFormatter alloc] init];**

**[dateFormatter setDateFormat:@"EEEE MMMM d, yyyy"];**

**self.birthdateLabel.text = [dateFormatter stringFromDate:model.birthdate];**

**}**

这全都直截了当，标准的 MVC。现在来看看我们如何用一个 View Model 来增强它。

**@interface PersonViewModel : NSObject**

**- (instancetype)initWithPerson:(Person \*)person;**

**@property (nonatomic, readonly) Person \*person;**

**@property (nonatomic, readonly) NSString \*nameText;**

**@property (nonatomic, readonly) NSString \*birthdateText;**

**@end**

我们的 View Model 的实现大概如下：

**@implementation PersonViewModel**

**- (instancetype)initWithPerson:(Person \*)person {**

**self = [super init];**

**if (!self) return nil;**

**\_person = person;**

**if (person.salutation.length > 0) {**

**\_nameText = [NSString stringWithFormat:@"%@ %@ %@", self.person.salutation, self.person.firstName, self.person.lastName];**

**} else {**

**\_nameText = [NSString stringWithFormat:@"%@ %@", self.person.firstName, self.person.lastName];**

**}**

**NSDateFormatter \*dateFormatter = [[NSDateFormatter alloc] init];**

**[dateFormatter setDateFormat:@"EEEE MMMM d, yyyy"];**

**\_birthdateText = [dateFormatter stringFromDate:person.birthdate];**

**return self;**

**}**

**@end**

Cool！我们已经将 **viewDidLoad** 中的表示逻辑放入我们的 View Model 里了。此时，我们新的 **viewDidLoad** 就会非常轻量：

**- (void)viewDidLoad {**

**[super viewDidLoad];**

**self.nameLabel.text = self.viewModel.nameText;**

**self.birthdateLabel.text = self.viewModel.birthdateText;**

**}**

所以，如你所见，并没有对我们的 MVC 架构做太多改变。还是同样的代码，只不过移动了位置。它与 MVC 兼容，带来[更轻量的 View Controllers](http://objccn.io/issue-1/)。

可测试，嗯？是怎样？好吧，View Controller 是出了名的难以测试，因为它们做了太多事情。在 MVVM 里，我们试着尽可能多的将代码移入 View Model 里。测试 View Controller 就变得容易多了，因为它们不再做一大堆事情，并且 View Model 也非常易于测试。让我们来看看：

**SpecBegin(Person)**

**NSString \*salutation = @"Dr.";**

**NSString \*firstName = @"first";**

**NSString \*lastName = @"last";**

**NSDate \*birthdate = [NSDate dateWithTimeIntervalSince1970:0];**

**it (@"should use the salutation available. ", ^{**

**Person \*person = [[Person alloc] initWithSalutation:salutation firstName:firstName lastName:lastName birthdate:birthdate];**

**PersonViewModel \*viewModel = [[PersonViewModel alloc] initWithPerson:person];**

**expect(viewModel.nameText).to.equal(@"Dr. first last");**

**});**

**it (@"should not use an unavailable salutation. ", ^{**

**Person \*person = [[Person alloc] initWithSalutation:nil firstName:firstName lastName:lastName birthdate:birthdate];**

**PersonViewModel \*viewModel = [[PersonViewModel alloc] initWithPerson:person];**

**expect(viewModel.nameText).to.equal(@"first last");**

**});**

**it (@"should use the correct date format. ", ^{**

**Person \*person = [[Person alloc] initWithSalutation:nil firstName:firstName lastName:lastName birthdate:birthdate];**

**PersonViewModel \*viewModel = [[PersonViewModel alloc] initWithPerson:person];**

**expect(viewModel.birthdateText).to.equal(@"Thursday January 1, 1970");**

**});**

**SpecEnd**

如果我们没有将这个逻辑移入 View Model，我们将不得不实例化一个完整的 View Controller 以及伴随的 View，然后去比较我们 View 中 Label 的值。这样做不只是会变成一个麻烦的间接层，而且它只代表了一个十分脆弱的测试。现在，我们可以按意愿自由地修改视图层级而不必担心破坏我们的单元测试。使用 MVVM 带来的对于测试的好处非常清晰，甚至从这个简单的例子来看也可见一斑，而在有更复杂的表示逻辑的情况下，这个好处会更加明显。

注意到在这个简单的例子中， Model 是不可变的，所以我们可以只在初始化的时候指定我们 View Model 的属性。对于可变 Model，我们还需要使用一些绑定机制，这样 View Model 就能在背后的 Model 改变时更新自身的属性。此外，一旦 View Model 上的 Model 发生改变，那 View 的属性也需要更新。Model 的改变应该级联向下通过 View Model 进入 View。

在 OS X 上，我们可以使用 Cocoa 绑定，但在 iOS 上我们并没有这样好的配置可用。我们想到了 KVO（Key-Value Observation），而且它确实做了很伟大的工作。然而，对于一个简单的绑定都需要很大的样板代码，更不用说有许多属性需要绑定了。作为替代，我个人喜欢使用 ReactiveCocoa，但 MVVM 并未强制我们使用 ReactiveCocoa。MVVM 是一个伟大的典范，它自身独立，只是在有一个良好的绑定框架时做得更好。

我们覆盖了不少内容：从普通的 MVC 派生出 MVVM，看它们是如何相兼容的范式，从一个可测试的例子观察 MVVM，并看到 MVVM 在有一个配对的绑定机制时工作得更好。如果你有兴趣学习更多关于 MVVM 的知识，你可以看看[这篇博客](http://www.teehanlax.com/blog/model-view-viewmodel-for-ios/)，它用更多细节解释了 MVVM 的好处，或者[这一篇](http://www.teehanlax.com/blog/krush-ios-architecture/)关于我们如何在最近的项目里使用 MVVM 获得巨大的成功的文章。我同样还有一个经过完整测试，基于 MVVM 的应用，叫做 [C-41](https://github.com/AshFurrow/C-41) ，它是开源的。去看看吧，如果你有任何疑问，请[告诉我](https://twitter.com/ashfurrow)。

[话题 #13 下的更多文章](http://objccn.io/issue-13)

原文 [Introduction to MVVM](http://www.objc.io/issue-13/mvvm.html)

### 关于译者



### 朱宏旭

留着长发的NIX，用过几年Linux，目前做 iOS 开发。喜欢科幻小说和科幻电影，因为幻想和梦境能带给我异世界体验。偶尔半梦半醒之间会质疑人生的意义，不知活着的所谓。

<https://twitter.com/nixzhu>

© 2016 [objc中国](http://objccn.io/)  
本站由 [@onevcat](http://im.onevcat.com/) 创建，文章源自 [objc.io](http://objc.io/)，由 [objc中国 项目组](https://github.com/objccn)整理维护

# Socket（长链接，短链接，心跳）

# 网络部分

## 实现离线断点下载功能（2016年4月19日星期二）

### 基本描述：

实现大文件的离线重新下载，在程序运行过程可以暂停、继续以及在程序app下一次启动后，可以继续下载，上次下载的文件进行保存，下次从上次下载完成的部分继续下载。

基本的实际案例：百度音乐下载、喜马拉雅音乐下载、百度云文件的下载。

其实本人认为，网络部分断点续传算是难度和复杂度都比较高的部分。

### 解决方案

#### 解决方案理论

用户发送下载文件请求（如下载音乐、大文件等），当用户首次发送下载文件，将请求头部设置为请求的文件二进制位置为0，在下载过程中不断将下载的二进制文件存储在本地，并且记录下载量，一旦用户暂停下载，app记录下最后下载完成的数量以及下载请求的位置，从多少字节开始下载。服务器端则根据客户端的请求，从head区域的文件请求位置，开始传输二进制数据，发送给客户端，在完成总的文件下载后，客户端取消任何关于本文件的下载信息。

当前，一般的网络处理，都会用到AFNetworking 网络请求库，但是此框架对实现离线形式（重启app，从上次的下载点继续下载）的断点续传支持不好。因此我此处使用到的是apple系统api原生的网络请求方式。

#### 解决实践

1. 用到NSURLSession ->session，调用含有代理设置的初始化方法:

+ (NSURLSession \*)sessionWithConfiguration:(NSURLSessionConfiguration \*)configuration delegate:(nullable id <NSURLSessionDelegate>)delegate delegateQueue:(nullable NSOperationQueue \*)queue;

2. 使用NSMutableRequest->request，设置头部文件为：

//设置表头信息

NSString \*rang = [NSString stringWithFormat:”@bytes=%zd-”,currrentSize];

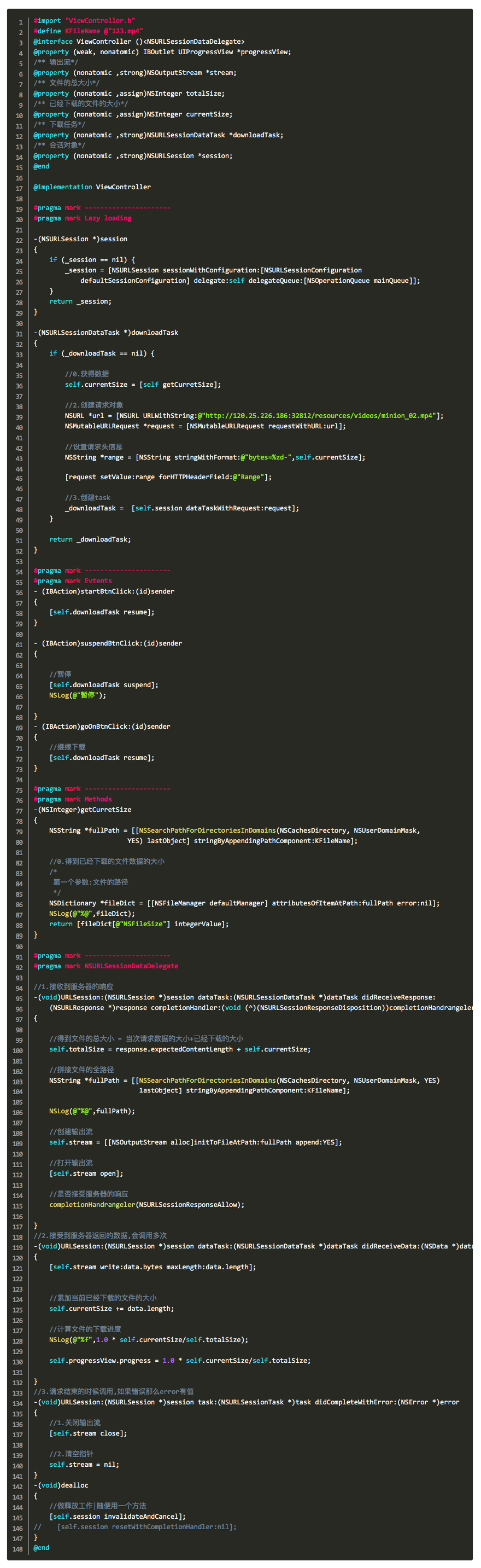
[request setValue:rang forHTTPHeanderField:”@Range”];

3.调用创建task方法

//创建task

NSURLSessionDataTask \*task = [session dataTaskWithRequest:request];

下图是参考别人写的，我觉的很好，直接贴上来，如果觉的图片不清楚，可以通过放大文档的形式查看，如下图：



# 数据存储

# 多线程

## NSThread

是三种方法里面相对轻量级的，但需要管理线程的生命周期、同步、加锁问题，这会导致一定的性能开销

### 优缺点

1.优点：NSThread比其他两种多线程方案较轻量级，更直观地控制线程对象

2.缺点：需要自己管理线程的生命周期，线程同步。线程同步对数据的加锁会有一定的系统开销

## Cocoa Operations(NSOperationQueue)

是基于OC实现的，NSOperation以面向对象的方式封装了需要执行的操作，不必关心线程管理、同步等问题。NSOperation是一个抽象基类，iOS提供了两种默认实现：NSInvocationOperation和NSBlockOperation，当然也可以自定义NSOperation

### 注意点：

1. 运行中的任务不能够取消，只能够取消还没有执行的任务。
2. 分为两个队列，主线程队列和新线程队列。
3. 在NSOperationQueue中, 可以指定各个NSOperation之间的依赖关系.
4. 主队列只是串行执行
5. operation的executing和finished状态量需要用willChangeValueForKey/didChangeValueForKey来触发KVO消息.

### 与gcd对比

NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

GCD 只支持FIFO 的队列， 而NSOperationQueue可以调整队列的执行顺序。（通过调整权重）

NSOperationQueue可以在Operation间设置依赖关系，而GCD不可以。 如果一个Operation依赖另一个Operation所产生的数据的化，你可以设置一个Operation依赖于另一个Operation来实现， NSOperationQueue可以根据依赖关系，可以以正确的顺序执行Queue中的Operation。

NSOperationQueue支持KVO。 这就意味着你可以观察任务的状态属性。

以上几点并不是说在任何处理多任务时一定要选择NSOperationQueue， 因为NSOperationQueue在执行速度上会比GCD慢。

## GCD（grand gentral dispatch）

简称GCD，iOS4才开始支持)：提供了一些新特性、运行库来支持多核并行编程，它的关注点更高：如何在多个cpu上提升效率

/\*

### GCD的优势－－－－－－－－

GCD是苹果公司为多核的并行运算提出的解决方案

GCD会自动利用更多的CPU内核（比如双核、四核）

GCD会自动管理线程的生命周期（创建线程、调度任务、销毁线程）

程序员只需要告诉GCD想要执行什么任务，不需要编写任何线程管理代码

### 任务和队列－－－－－－－－－

GCD中有2个核心概念

（1）任务：执行什么操作

（2）队列：用来存放任务

队列－－－－－－－－－

GCD的队列分为2大类型

（1）并发队列（concurrent dispatch queue）

可以让多个任务并发（同时）执行（自动开启多个线程同时执行任务）并发功能，只有在异步(dispatch\_async)函数下才有效。

（2）串行队列 (serial dispatch queue)

让任务一个接一个执行

### 补充说明－－－－－－－－－

4个术语：同步 异步 并发 串行

同步和异步决定是不是要开启新线程

同步：在当前线程中执行，不具备开启新线程的能力

异步：在新线程中执行，具备开启新线程的能力

并发和串行决定任务的执行方式

并发：多个任务并发（同时）执行

串行：一个任务执行后，再执行下一个任务

### 串行队列

GCD中获得串行队列有2种途径

（1）使用dispatch\_queue\_create 函数创建

dispatch\_queue\_t dispatch\_queue\_create(const char \*label, dispatch\_queue\_attr\_t attr); // 队列名称， 队列属性，一般用NULL即可

示例：

dispatch\_queue\_t queue = dispatch\_queue\_create("wendingding", NULL); // 创建

dispatch\_release(queue); // 非ARC需要释放手动创建的队列

（2）使用主队列（跟主线程相关联的队列）

主队列是GCD自带的一种特殊的串行队列,放在主队列中的任务，都会放到主线程中执行

使用dispatch\_get\_main\_queue()获得主队列

示例：

dispatch\_queue\_t queue = dispatch\_get\_main\_queue();

### 并发队列－－－－－－－

GCD默认已经提供了全局的并发队列，供整个应用使用，不需要手动创建

使用dispatch\_get\_global\_queue函数获得全局的并发队列

dispatch\_queue\_t dispatch\_get\_global\_queue(dispatch\_queue\_priority\_t priority,unsigned long flags); // 此参数暂时无用，用0即可

示例：

这个参数是留给以后用的，暂时用不上，传个0。

第一个参数为优先级，这里选择默认的。获取一个全局的默认优先级的并发队列。

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0); // 获得全局并发队列

说明：全局并发队列的优先级

#define DISPATCH\_QUEUE\_PRIORITY\_HIGH 2 // 高

#define DISPATCH\_QUEUE\_PRIORITY\_DEFAULT 0 // 默认（中）

#define DISPATCH\_QUEUE\_PRIORITY\_LOW (-2) // 低

#define DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND INT16\_MIN // 后台

### 各种队列执行的效果

### 小结

说明：同步函数不具备开启线程的能力，无论是什么队列都不会开启线程；异步函数具备开启线程的能力，开启几条线程由队列决定（串行队列只会开启一条新的线程，并发队列会开启多条线程）。

同步函数

（1）并发队列：不会开线程

（2）串行队列：不会开线程

异步函数

（1）并发队列：能开启N条线程

（2）串行队列：开启1条线程

### 补充：

凡是函数中，各种函数名中带有create\copy\new\retain等字眼，都需要在不需要使用这个数据的时候进行release。

GCD的数据类型在ARC的环境下不需要再做release。

CF（core Foundation）的数据类型在ARC环境下还是需要做release。

异步函数具备开线程的能力，但不一定会开线程

### GCD调度组

有这么一个需要，分别执行2个耗时的异步操作，等2个异步操作都执行完毕后在回到主线程执行操作。如果想要快速高效地实现这个需求，可以考虑使用调度组。 调度组的创建代码如下：

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_group\_async(dispatch\_group\_t group, dispatch\_queue\_t queue, ^(void)block)

第一个参数group是一个调度组，queue是一个异步队列，这里如果使用同步队列意义就不大了，不过还是能达到效果，第三个参数是一个执行代码块，任务都添加到该代码块里面。 往调度组里面添加任务并获得通知的代码如下

// 群组－统一监控一组任务

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_queue\_t q = dispatch\_get\_global\_queue(0, 0);

// 添加任务

// group 负责监控任务，queue 负责调度任务

dispatch\_group\_async(group, q, ^{

[NSThread sleepForTimeInterval:1.0];

NSLog(@"任务1 %@", [NSThread currentThread]);

});

dispatch\_group\_async(group, q, ^{

NSLog(@"任务2 %@", [NSThread currentThread]);

});

dispatch\_group\_async(group, q, ^{

NSLog(@"任务3 %@", [NSThread currentThread]);

});

// 监听所有任务完成 － 等到 group 中的所有任务执行完毕后，"由队列调度 block 中的任务异步执行！"

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

// 修改为主队列，后台批量下载，结束后，主线程统一更新UI

NSLog(@"OK %@", [NSThread currentThread]);

});

NSLog(@"come here");

GCD调度组中还有一个添加任务的函数： dispatch\_group\_enter(dispatch\_group\_t group) dispatch\_group\_leave(dispatch\_group\_t group) 这两个函数要配对出现 例如：

// 群组－统一监控一组任务

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_queue\_t q = dispatch\_get\_global\_queue(0, 0);

// 1> 入组 -> 之后的 block 会被 group 监听

// dispatch\_group\_enter 一定和 dispatch\_group\_leave 要配对出现

dispatch\_group\_enter(group);

dispatch\_async(q, ^{

NSLog(@"task1 %@", [NSThread currentThread]);

// block 的末尾，所有任务执行完毕后，添加一个出组

dispatch\_group\_leave(group);

});

// 再次入组

dispatch\_group\_enter(group);

dispatch\_async(q, ^{

[NSThread sleepForTimeInterval:1.0];

NSLog(@"task2 %@", [NSThread currentThread]);

// block 的末尾，所有任务执行完毕后，添加一个出组

dispatch\_group\_leave(group);

});

// 群组结束

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

NSLog(@"OVER");

});

NSLog(@"come here");

代码执行的效果与刚才那种方式相同！ GCD中还有一个函数

dispatch\_group\_wait(dispatch\_group\_t group, dispatch\_time\_t timeout)

该函数是个阻塞式的等待，当我们将第二个参数设置为DISPATCH\_TIME\_FOREVER时，群组任务不执行完，后续代码就会无法执行，也就是说只要任务不执行完come here就不会打印。

### dispatch\_apply

执行某个代码片段若干次。

dispatch\_apply(10, globalQ, ^(size\_t index) {

// do sth. 10 times

});

### dispatch\_once

它可以保证整个应用程序生命周期中某段代码只被执行一次！

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

// code to be executed once

});

举例：

＋(MyInstance\*)shareMyInstance（）

｛

static MyInstance \*inst=nil;

static dispatch\_once\_t p;

dispatch\_once(&p,^{

inst=[[MyInstance alloc]init];

});

return inst;

｝

### dispatch\_after

有时候我们需要等个几秒钟然后做个动画或者给个提示，这时候可以用dispatch\_after这个函数：

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\_time

或者说同步地等待一段时间看是否结束：

dispatch\_time\_t time = dispatch\_time(DISPATCH\_TIME\_NOW, 1ull \* NSEC\_PER\_SEC);

dispatch\_group\_wait(group, time);

### dispatch\_barrier\_async

通过dispatch\_barrier\_async函数提交的任务会等它前面的任务执行结束才开始，然后它后面的任务必须等它执行完毕才能开始。

dispatch\_async(concurrentQ, blk0);

dispatch\_async(concurrentQ, blk1);

dispatch\_barrier\_async(concurrentQ, blk\_barrier);

dispatch\_async(concurrentQ, blk2);

\*/

## 同步锁（NSLock）

## @ synchronized

使用方法；

在防止多个线程访问的对象和代码处使用如下：

@synchronized (\_elements) {

[\_elements addObject:element];

}

”synchronized block” 与 [\_lock lock] 和 [\_lock unlock] 效果相同。你可以把它当成是锁住 self，仿佛 self 就是个 NSLock。锁在左括号 { 后面的任何代码运行之前被获取到，在右括号 } 后面的任何代码运行之前被释放掉。这爽就爽在妈妈再也不用担心我忘记调用 unlock 了！

其实上面的\_elements可以设置为当前的类对象，self，关键是通过@synchronized关键字会将过好内的代码进行加锁，等代码执行完成后才允许其他线程访问这些代码。

# IOS动画

## 动画内容学习地址

1 <https://zsisme.gitbooks.io/ios-/content/chapter1/layers-and-trees.html>

2 <https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html>

## 我的动画研究之旅

### 研究前言

“研究”这个词语，说明一件事情，要研究的东西并不是很简单，否则谈不上“研究”二字。“研究”说明的是要沉下心来，深入思考，而不是浅尝辄止，追求数量和速度，否则那就偏离了研究的意义。

# NSRuntime

<http://ios.jobbole.com/84028/#comment-90464>

* [如何使用这个 Playground](https://github.com/Tuccuay/RuntimeSummary#Playground)
* [消息机制介绍 / Messaging](https://github.com/Tuccuay/RuntimeSummary#Messaging)
* [方法交换 / MethodSwizzling](https://github.com/Tuccuay/RuntimeSummary#MethodSwizzling)

* [动态加载方法 / ResolveInstanceMethod](https://github.com/Tuccuay/RuntimeSummary" \l "ResolveInstanceMethod)
* [消息转发 / ForwardMessage](https://github.com/Tuccuay/RuntimeSummary" \l "ForwardMessage)
* [动态关联属性 / AssociatedObject](https://github.com/Tuccuay/RuntimeSummary#AssociatedObject)
* [字典转模型 / MakeModel](https://github.com/Tuccuay/RuntimeSummary#MakeModel)
* [对象归档、解档 / ObjectArchive](https://github.com/Tuccuay/RuntimeSummary#ObjectArchive)
* [更多](https://github.com/Tuccuay/RuntimeSummary#More)

### 消息发送

无论何时，要调用objc\_msgSend函数，必须要将函数强制转换成合适的函数指针类型才能调用。

从objc\_msgSend函数的声明来看，它应该是不带返回值的，但是我们在使用中却可以强制转换类型，以便接收返回值。另外，它的参数列表是可以任意多个的，前提也是要强制函数指针类型。

#### 学习使用

我们建立一个类，就专门学习如何运用objc\_msgSend函数来发送消息。我们建立了一个HYBMsgSend类来学习。

#### 1.创建并初始化对象

我们一直以来都是使用类似这样的[[HYBMsgSend alloc] init]来创建并初始化对象吧，其实在编译时，这一行代码也会转换成类似如下的代码：

// 1.创建对象

HYBMsgSend \*msg = ((HYBMsgSend \* (\*)(id, SEL))objc\_msgSend)((id)[HYBMsgSend class], @selector(alloc));

// 2.初始化对象

msg = ((HYBMsgSend \* (\*)(id, SEL))objc\_msgSend)((id)msg, @selector(init));

要发送消息给msg对象，并将创建的对象返回，那么我们需要强转函数指针类型。(HYBMsgSend \* (\*)(id, SEL)这是带一个对象指针返回值和两个参数的函数指针，这样就可以调用了。

#### 2.发送无参数无返回值消息

我们先定义一个方法：

- (void)noArgumentsAndNoReturnValue {

NSLog(@"%s was called, and it has no arguments and return value", \_\_FUNCTION\_\_);

}

然后，我们发送消息，测试一下结果。

// 2.调用无参数无返回值方法

((void (\*)(id, SEL))objc\_msgSend)((id)msg, @selector(noArgumentsAndNoReturnValue));

结果如下，说明成功地接收到消息并处理了：

-[HYBMsgSend noArgumentsAndNoReturnValue] was called, and it has no arguments and return value

#### 3.带参数不带返回值消息

我们定义一个方法只带参数，不带返回值：

- (void)hasArguments:(NSString \*)arg {

NSLog(@"%s was called, and argument is %@", \_\_FUNCTION\_\_, arg);

}

然后尝试发送消息试试：

// 3.调用带一个参数但无返回值的方法

((void (\*)(id, SEL, NSString \*))objc\_msgSend)((id)msg, @selector(hasArguments:), @"带一个参数，但无返回值");

同样，我们也是需要强转函数指针类型，否则会报错的。其实，只有调用runtime函数来发送消息，几乎都需要强转函数指针类型为合适的类型。

其打印结果说明成功接收到消息并处理了：

-[HYBMsgSend hasArguments:] was called, and argument is 带一个参数，但无返回值

#### 4.带返回值不带参数消息

当消息带有返回值时，我们如何接收呢？我们先声明一个方法，让它带有返回值，但是不带参数：

- (NSString \*)noArgumentsButReturnValue {

NSLog(@"%s was called, and return value is %@", \_\_FUNCTION\_\_, @"不带参数，但是带有返回值");

return @"不带参数，但是带有返回值";

}

然后发送消息，并接收返回值：

// 4.调用带返回值，但是不带参数

NSString \*retValue = ((NSString \* (\*)(id, SEL))objc\_msgSend)((id)msg, @selector(noArgumentsButReturnValue));

NSLog(@"5. 返回值为：%@", retValue);

打印结果说明成功地发送了消息并得到处理，且成功地获取到了返回值：

-[HYBMsgSend noArgumentsButReturnValue] was called, and return value is 不带参数，但是带有返回值

#### 5.带参数带返回值的消息

定义一个带参数带普通返回值的方法：

- (int)hasArguments:(NSString \*)arg andReturnValue:(int)arg1 {

NSLog(@"%s was called, and argument is %@, return value is %d", \_\_FUNCTION\_\_, arg, arg1);

return arg1;

}

发送消息，并接收返回值：

// 6.带参数带返回值

int returnValue = ((int (\*)(id, SEL, NSString \*, int))

objc\_msgSend)((id)msg,

@selector(hasArguments:andReturnValue:),

@"参数1",

2016);

NSLog(@"6. return value is %d", returnValue);

其结果如下，说明调用成功，参数也传过去了，返回值也接收到了：

-[HYBMsgSend hasArguments:andReturnValue:] was called, and argument is 参数1, return value is 2016

6. return value is 2016

#### 6.动态添加方法再调用

我们声明一个C语言函数：

// C函数

int cStyleFunc(const void \*arg1, const void \*arg2) {

NSLog(@"%s was called, arg1 is %s, and arg2 is %s", \_\_FUNCTION\_\_, arg1, arg2);

return 1;

}

这个函数并不属性对象方法，因此我们不能直接调用，但是我们可以动态地添加方法到对象中，然后再发送消息：

class\_addMethod(msg.class, NSSelectorFromString(@"cStyleFunc"), (IMP)cStyleFunc, "v@:");

returnValue = ((int (\*)(id, SEL, const void \*, const void \*))

objc\_msgSend)((id)msg,

NSSelectorFromString(@"cStyleFunc"),

"参数1",

"参数2");

NSLog(@"7. return value is %d", returnValue);

#### 7.带浮点返回值的消息

对于发送带浮点返回值类型的消息，我们可以使用objc\_msgSend\_fpret也可以使用objc\_msgSend。不过这两个函数返回结果是一样的。笔者并不是很清楚，这两个函数的主要区别是什么。根据注释说明，只是说有一些处理器上，需要使用objc\_msgSend\_fpret来发送带浮点返回值类型的消息。

float retFloatValue = ((float (\*)(id, SEL))objc\_msgSend\_fpret)((id)msg, @selector(returnFloatType));

NSLog(@"%f", retFloatValue);

retFloatValue = ((float (\*)(id, SEL))objc\_msgSend)((id)msg, @selector(returnFloatType));

NSLog(@"%f", retFloatValue);

#### 8.带结构体返回值的消息

对于返回值类型为结构体的消息，我们必须使用objc\_msgSend\_stret而不能直接使用objc\_msgSend函数，否则会crash：

// 9.返回结构体时，不能使用objc\_msgSend，而是要使用objc\_msgSend\_stret，否则会crash

CGRect frame = ((CGRect (\*)(id, SEL))objc\_msgSend\_stret)((id)msg, @selector(returnTypeIsStruct));

NSLog(@"9. return value is %@", NSStringFromCGRect(frame));

#### 案例：

// 创建一个对象

// Cat \*harlan = [[Cat alloc] init];

// 使用 Runtime 创建一个对象

// 根据类名获取到类

Class catClass = objc\_getClass("Cat");

// 同过类创建实例对象

// 如果这里报错，请将 Build Setting -> Enable Strict Checking of objc\_msgSend Calls 改为 NO

Cat \*harlan = objc\_msgSend(catClass, @selector(alloc));

// 初始化对象

// harlan = [harlan init];

// 通过 Runtime 初始化对象

harlan = objc\_msgSend(harlan, @selector(init));

// 调用对象方法

// [harlan eat];

// 通过 Runtime 调用对象方法

// 调用的这个方法没有声明只有实现所以这里会有警告

// 但是发送消息的时候会从方法列表里寻找方法

// 所以这个能够成功执行

objc\_msgSend(harlan, @selector(eat));

// 输出： 2016-04-21 21:10:20.733 Messaging[20696:1825249] burbur~

// 当然，objc\_msgSend 可以传递参数

Cat \*alex = objc\_msgSend(objc\_msgSend(objc\_getClass("Cat"), sel\_registerName("alloc")), sel\_registerName("init"));

objc\_msgSend(alex, @selector(run:), 10);

### 方法交换 / MethodSwizzling

+ (void)load {

// 获取到两个方法

Method imageNamedMethod = class\_getClassMethod(self, @selector(imageNamed:));

Method tuc\_imageNamedMethod = class\_getClassMethod(self, @selector(tuc\_imageNamed:));

// 交换方法

method\_exchangeImplementations(imageNamedMethod, tuc\_imageNamedMethod);

}

+ (UIImage \*)tuc\_imageNamed:(NSString \*)name {

// 因为来到这里的时候方法实际上已经被交换过了

// 这里要调用 imageNamed: 就需要调换被交换过的 tuc\_imageNamed

UIImage \*image = [UIImage tuc\_imageNamed:name];

// 判断是否存在图片

if (image) {

NSLog(@"加载成功");

} else {

NSLog(@"加载失败");

}

return image;

}

### [动态加载方法 / ResolveInstanceMethod](https://github.com/Tuccuay/RuntimeSummary#ResolveInstanceMethod)

void run(id self, SEL \_cmd, NSNumber \*metre) {

NSLog(@"跑了%@米",metre);

}

// 当调用了一个未实现的方法会来到这里

+ (BOOL)resolveInstanceMethod:(SEL)sel {

if (sel == NSSelectorFromString(@"run:")) {

// 动态添加 run: 方法

class\_addMethod(self, @selector(run:), run, "v@:@");

return YES;

}

return [super resolveInstanceMethod:sel];

}

### 消息转发 / ForwardMessage

#pragma mark - 实例方法

// 第一步

// 在没有找到方法时，会先调用此方法，可用于动态添加方法

// 返回 YES 表示相应 selector 的实现已经被找到并添加到了类中，否则返回 NO

+ (BOOL)resolveInstanceMethod:(SEL)sel {

return YES;

}

// 第二步

// 如果第一步的返回 NO 或者直接返回了 YES 而没有添加方法，该方法被调用

// 在这个方法中，我们可以指定一个可以返回一个可以响应该方法的对象

// 如果返回 self 就会死循环

- (id)forwardingTargetForSelector:(SEL)aSelector {

return nil;

}

// 第三步

// 如果 `forwardingTargetForSelector:` 返回了 nil，则该方法会被调用，系统会询问我们要一个合法的『类型编码(Type Encoding)』

// https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html

// 若返回 nil，则不会进入下一步，而是无法处理消息

- (NSMethodSignature \*)methodSignatureForSelector:(SEL)aSelector {

return [NSMethodSignature signatureWithObjCTypes:"v@:"];

}

// 当实现了此方法后，-doesNotRecognizeSelector: 将不会被调用

// 如果要测试找不到方法，可以注释掉这一个方法

// 在这里进行消息转发

- (void)forwardInvocation:(NSInvocation \*)anInvocation {

// 我们还可以改变方法选择器

[anInvocation setSelector:@selector(touch)];

// 改变方法选择器后，还需要指定是哪个对象的方法

[anInvocation invokeWithTarget:self];

}

- (void)touch {

NSLog(@"Cat 没有实现 -stoke 方法，并且成功的转成了 -touch 方法");

}

- (void)doesNotRecognizeSelector:(SEL)aSelector {

NSLog(@"无法处理消息：%@", NSStringFromSelector(aSelector));

}

### 动态关联属性 / AssociatedObject

- (void)setName:(NSString \*)name {

// 把属性关联给对象

objc\_setAssociatedObject(self, "name", name, OBJC\_ASSOCIATION\_RETAIN\_NONATOMIC);

}

- (NSString \*)name {

// 取出属性

return objc\_getAssociatedObject(self, "name");

}

### 字典转模型 / MakeModel

+ (instancetype)modelWithDict:(NSDictionary \*)dict updateDict:(NSDictionary \*)updateDict {

id model = [[self alloc] init];

// 遍历模型中属性

unsigned int count = 0;

Ivar \*ivars = class\_copyIvarList(self, &count);

for (int i = 0 ; i < count; i++) {

Ivar ivar = ivars[i];

// 属性名称

NSString \*ivarName = [NSString stringWithUTF8String:ivar\_getName(ivar)];

ivarName = [ivarName substringFromIndex:1];

id value = dict[ivarName];

// 模型中属性名对应字典中的key

if (value == nil) {

if (updateDict) {

NSString \*keyName = updateDict[ivarName];

value = dict[keyName];

}

}

[model setValue:value forKeyPath:ivarName];

}

return model;

}

+ (instancetype)modelWithDict:(NSDictionary \*)dict {

return [self modelWithDict:dict updateDict:nil];

}

### 对象归档、解档 / ObjectArchive

- (void)tuc\_initWithCoder:(NSCoder \*)aDecoder {

// 不光归档自身的属性，还要循环把所有父类的属性也找出来

Class selfClass = self.class;

while (selfClass &&selfClass != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList(selfClass, &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

// 如果有实现忽略属性的方法

if ([self respondsToSelector:@selector(ignoredProperty)]) {

// 就跳过这个属性

if ([[self ignoredProperty] containsObject:key]) continue;

}

id value = [aDecoder decodeObjectForKey:key];

[self setValue:value forKey:key];

}

free(ivars);

selfClass = [selfClass superclass];

}

}

- (void)tuc\_encodeWithCoder:(NSCoder \*)aCoder {

Class selfClass = self.class;

while (selfClass &&selfClass != [NSObject class]) {

unsigned int outCount = 0;

Ivar \*ivars = class\_copyIvarList([self class], &outCount);

for (int i = 0; i < outCount; i++) {

Ivar ivar = ivars[i];

NSString \*key = [NSString stringWithUTF8String:ivar\_getName(ivar)];

if ([self respondsToSelector:@selector(ignoredProperty)]) {

if ([[self ignoredProperty] containsObject:key]) continue;

}

id value = [self valueForKeyPath:key];

[aCoder encodeObject:value forKey:key];

}

free(ivars);

selfClass = [selfClass superclass];

}

}

# NSRunLoop(较难)

暂放

# Block

## Block简介

Block是一种比较特殊的数据类型。它可以保存一段代码，在合适的时候取出来调用。

## Block的定义

Block的定义格式

返回值类型(^block变量名)(形参列表) = ^(形参列表) {

};

调用Block保存的代码

block变量名(实参);

默认情况下,Block内部不能修改外面的局部变量

Block内部可以修改使用\_\_block修饰的局部变量

Block的模式

1.无参数无返回值的Block

2.有参数无返回值的Block

3.有参数有返回值的Block

## Block的修饰

ARC情况下

1.如果用copy修饰Block，该Block就会存储在堆空间。则会对Block的内部对象进行强引用，导致循环引用。内存无法释放。

解决方法：

新建一个指针(\_\_weak typeof(Target) weakTarget = Target )指向Block代码块里的对象，然后用weakTarget进行操作。就可以解决循环引用问题。

2.如果用weak修饰Block，该Block就会存放在栈空间。不会出现循环引用问题。

MRC情况下

用copy修饰后，如果要在Block内部使用对象，则需要进行(\_\_block typeof(Target) blockTarget = Target )处理。在Block里面用blockTarget进行操作。

## Block简单用法举例

### 无参数无返回值的Block

/\*\*

\* 无参数无返回值的Block

\*/

-(void)func1{

/\*\*

\* void ：就是无返回值

\* emptyBlock：就是该block的名字

\* ()：这里相当于放参数。由于这里是无参数，所以就什么都不写

\*/

void (^emptyBlock)() = ^(){

NSLog(@"无参数,无返回值的Block");

};

emptyBlock();

}

### 有参数无返回值的Block

/\*\*

\* 调用这个block进行两个参数相加

\*

\* @param int 参数A

\* @param int 参数B

\*

\* @return 无返回值

\*/

void (^sumBlock)(int ,int ) = ^(int a,int b){

NSLog(@"%d + %d = %d",a,b,a+b);

};

/\*\*

\* 调用这个sumBlock的Block，得到的结果是20

\*/

sumBlock(10,10);

### 有参数有返回值的Block

/\*\*

\* 有参数有返回值

\*

\* @param NSString 字符串1

\* @param NSString 字符串2

\*

\* @return 返回拼接好的字符串3

\*/

NSString\* (^logBlock)(NSString \*,NSString \*) = ^(NSString \* str1,NSString \*str2){

return [NSString stringWithFormat:@"%@%@",str1,str2];

};

//调用logBlock,输出的是 我是Block

NSLog(@"%@", logBlock(@"我是",@"Block"));

### Block结合typedef使用

自己定义一个Block类型，用定义的类型去创建Block，更加简单便捷。  
这里举例一个Block回调修改上一下界面的背景颜色。  
ViewController1 控制器1，ViewController2 控制器2  
控制器1跳转到控制器2，然后在控制器2触发事件回调修改控制器1的背景颜色为红色。

ViewController2的实现

#import <UIKit/UIKit.h> @interface ViewController2 : UIViewController /\*\* \* 定义了一个changeColor的Block。这个changeColor必须带一个参数，这个参数的类型必须为id类型的 \* 无返回值 \* @param id \*/ typedef void(^changeColor)(id); /\*\* \* 用上面定义的changeColor声明一个Block,声明的这个Block必须遵守声明的要求。 \*/ @property (nonatomic, copy) changeColor backgroundColor; @end

-(void)touchesBegan:(NSSet<UITouch \*> \*)touches withEvent:(UIEvent \*)event{ //声明一个颜色 UIColor \*color = [UIColor redColor]; //用刚刚声明的那个Block去回调修改上一界面的背景色 self.backgroundColor(color); }

ViewController1的实现

-(void)touchesBegan:(NSSet<UITouch \*> \*)touches withEvent:(UIEvent \*)event{ ViewController2 \*vc =[[ViewController2 alloc]init]; // 回调修改颜色 vc.backgroundColor = ^(UIColor \*color){ self.view.backgroundColor = color; }; [self.navigationController pushViewController:vc animated:YES]; }

# 视频播放

### 相关博客地址

cocoachina比较多

<http://www.jianshu.com/p/3618a9116660>

http://www.jianshu.com/p/b8db6c142aad

<http://cnbin.github.io/blog/2015/05/19/iospei-zhi-ffmpegkuang-jia/>

<https://github.com/masterweb121/Eleven>

# 内存管理

## 检测内存泄漏

**Xcode结合Leaks检测内存泄露**

字数329 阅读503 评论1 喜欢18

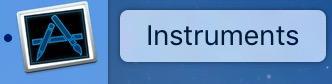
Collection/Bookmark/Share for width under 768px

**内存很重要！内存很重要！内存很重要！**

重要的事情要说三遍。虽然现在项目大多数是ARC模式，Xcode会帮我们管理内存，但并不意味着这样就不会存在内存泄露的现象。Xcode里面有一个工具Instruments，这个工具可以检测项目的性能。

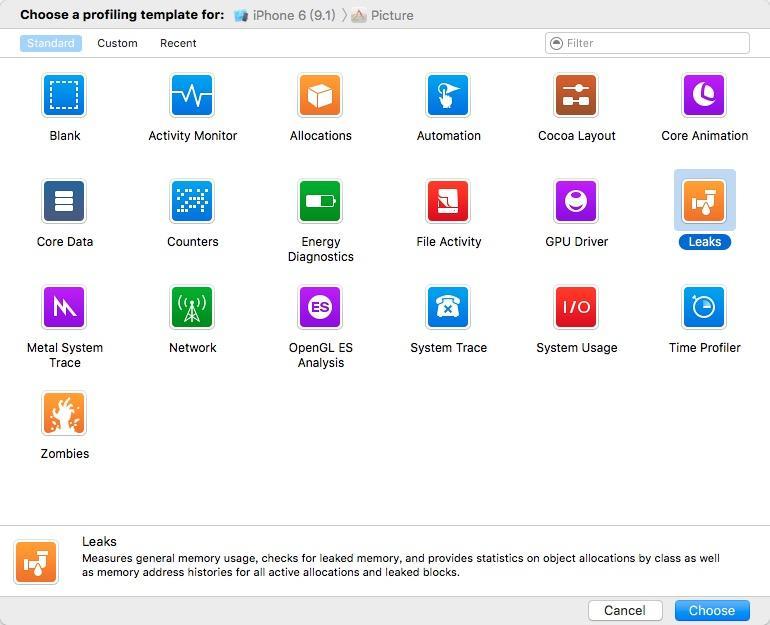
**Xcode结合Leaks检测内存泄露代码**

1.command+i 开启 Instruments工具



启动.png

2.选择Leaks，点choose



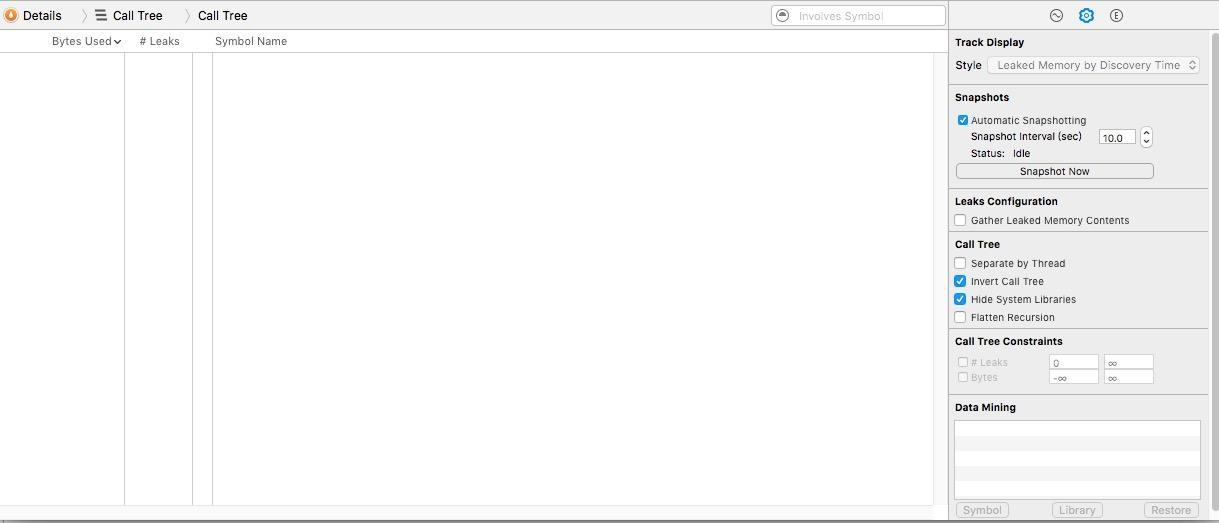
Leaks.png

3.选择Leaks，然后在Leaks那里选择**Call Tree**



xxxx.png

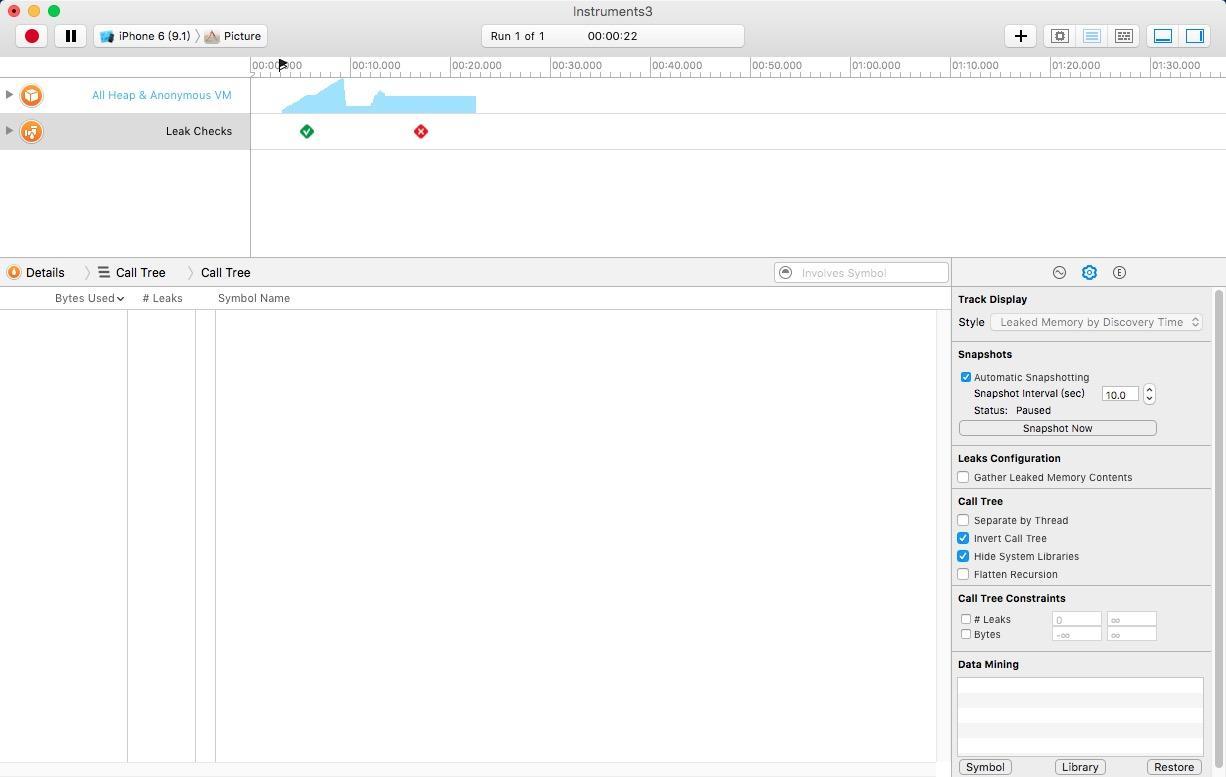
4.选择好Call Tree 之后选择设置，长得和齿轮一样



xxxxx.png

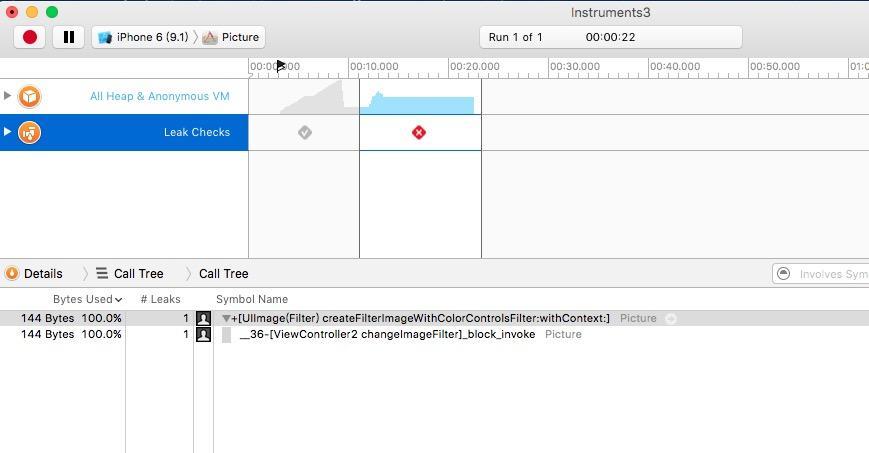
把Invert Call Tree 和 Hide system Libraries勾选

5.最后点左上角的红点Record，你就会发现模拟器启动了，然后随便你操作app。如果出现了红色的叉叉，就说明存在内存泄露。



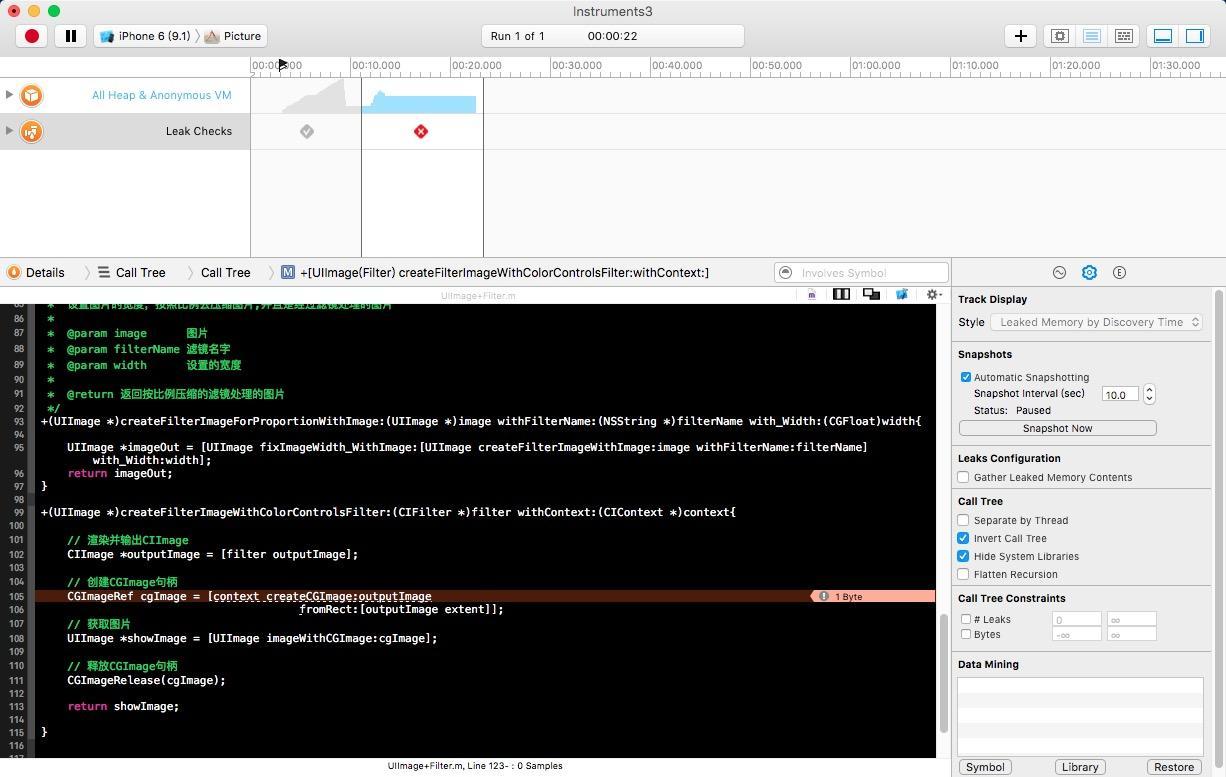
Bug.png

6.选中红色叉叉的区域,在Details里面就可以知道泄露的代码具体在哪一行。双击那段代码就可以去到泄露的位置。



findBug.png

7.找到泄露的代码，也可以一直点进去看。



Bug11.png

8.结束啦，找到泄露的地方，那就改去咯。

##### 内存管理

堆：需要手动释放。  
栈：系统自动管理。

##### 所谓内存管理,就是对内存进行管理,涉及的操作有:

分配内存 :比如创建一个对象,会增加内存占用  
清除内存 :比如销毁一个对象,能减小内存占用

##### 内存管理的管理范围

任何继承了NSObject的对象  
对其他非对象类型无效(int、char、float、double、struct、enum等)

##### 只有OC对象才需要进行内存管理的本质原因

OC对象存放于堆里面  
OC基础类型一般放在栈里面(栈内存会被系统自动回收)

##### 引用计数器的常见操作

给对象发送一条retain消息,可以使引用计数器值+1（retain方法返回对象本身）  
给对象发送一条release消息,可以使引用计数器值-1  
给对象发送retainCount消息,可以获得当前的引用计数器值

##### 需要注意的是: release并不代表销毁\回收对象,仅仅是计数器-1。

##### 空指针\野指针

僵尸对象  
已经被销毁的对象(不能再使用的对象)

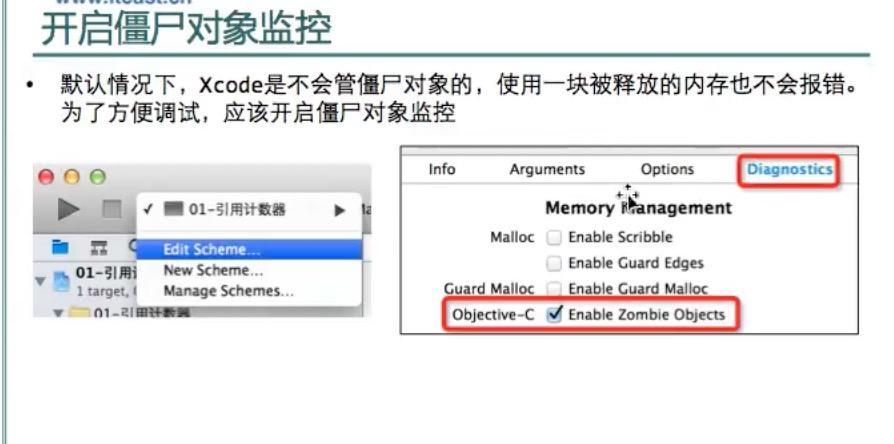
野指针  
指向僵尸对象(不可用内存)的指针  
给野指针发消息会报**EXC\_BAD\_ACCESS**错误

空指针  
没有指向存储空间的指针(里面存的是nil,也就是0)  
给空指针发消息是没有任何反应的。  
为了避免野指针错误的常见办法  
在对象被销毁之后,将指向对象的指针变为空指针

##### delloc方法注意细节

**当一个对象的引用计数器值为0时**  
这个对象即将被销毁，其占用的内存被系统回收  
系统会自动给对象发送一条dealloc消息  
(因此,从dealloc方法有没有被调用,就可以判断出对象是否被销毁)  
**dealloc方法的重写**  
一般会重写dealloc方法,在这里释放相关资源,dealloc就是对象的遗言  
一旦重写了dealloc方法,就必须调用[super dealloc],并且放在最后面调用  
使用注意  
**不能直接调用dealloc方法**  
一旦对象被回收了,它占用的内存就不再可用,坚持使用会导致程序崩溃（野指针错误）

##### Xcode开启僵尸对象的监控



截图1.png

##### 苹果官方规定的内存管理原则

谁创建谁release:如果你通过alloc、new或[mutable]copy来创建一个对象，那么你必须调用release或autorelease。  
谁retain谁release:只要你调用了retain，就必须调用一次release。  
总结一下就是  
**有加就有减**  
**曾经让对象的计数器+1，就必须在最后让对象计数器-1**

##### set方法的内存管理

- (void)setCar:(Car\*)car { if(car!= \_car) { // 对当前正在使用的车（旧车）做一次release [\_car release]; // 对新车做一次retain操作 \_car=[car retain]; } }

##### dealloc方法内存管理

- (void)dealloc { //当人不在了，代表不用车了 // 对车做一次release操作 [\_car release]; //调用dealloc方法时候一定要调用父类的dealloc的方法 [super dealloc]; }

##### @property参数修饰

##### 控制set方法的内存管理

retain：release旧值，retain新值（用于OC对象）  
assign：直接赋值，不做任何内存管理(默认，用于非OC对象类型)  
copy ：release旧值，copy新值（一般用于NSString\*,block）

##### 控制需不需生成set方法

readwrite：同时生成set方法和get方法（默认）  
readonly ：只会生成get方法

##### 多线程管理

atomic ：性能低（默认）  
nonatomic：性能高

##### 控制set方法和get方法的名称

setter：设置set方法的名称，一定有个冒号:  
getter：设置get方法的名称

##### 强引用strong与弱引用weak的广义区别：

强引用**strong**也就是我们通常所讲的引用，其存亡直接决定了所指对象的存亡。如果不存在指向一个对象的引用，并且此对象不再显示列表中，则此对象会被从内存中释放。  
弱引用**weak**除了不决定对象的存亡外，其他与强引用相同。即使一个对象被持有无数个若引用，只要没有强引用指向他，那麽其还是会被清除。没办法，还是 “强哥” 有面子。

**简单讲strong等同retain**  
ARC时候用strong,MRC时候用return  
weak比assign多了一个功能，当对象消失后自动把指针变成nil，好处不言而喻。所以，我觉得在delegate时候用weak会好过assign。

##### 至于block的话，就要结合考虑。（weak，copy，assign）这三个结合情况，在下一篇的介绍。

**\_\_weak** ,**\_\_strong** ,**\_\_unsafe\_unretained**, **\_\_autoreleasing**用来修饰变量.  
(\_\_strong) 是缺省的关键词。  
(\_\_weak) 声明了一个可以自动 nil 化的弱引用。  
(\_\_unsafe\_unretained) 声明一个弱应用，但是不会自动nil化，也就是说，如果所指向的内存区域被释放了，这个指针就是一个野指针了。  
(\_\_autoreleasing) 用来修饰一个函数的参数，这个参数会在函数返回的时候被自动释放。

###### @class和#import的区别

作用上的区别  
**#import**会包含引用类的所有信息(内容), 包括引用类的变量和方法  
**@class**仅仅是告诉编译器有这么一个类,具体这个类里有什么信息, 完全不知道。

效率上的区别  
如果有上百个头文件都#import了同一个文件，或者这些文件依次被#import,那么一旦最开始的头文件稍有改动，后面引用到这个文件的所有类都需要重新编译一遍,编译效率非常低  
相对来讲，使用@class方式就不会出现这种问题了

##### MRC循环retian

循环retain的场景  
比如A对象retain了B对象，B对象retain了A对象

循环retain的弊端  
这样会导致A对象和B对象永远无法释放

循环retain的解决方案  
当两端互相引用时，应该一端用retain、一端用assign

**同理，在ARC下，如果两个对象相互引用，并且用strong修饰，那两个对象都无法释放。解决方案：其中一个对象用weak修饰，一个用strong修饰。那就可以让两个对象都可以释放。**

##### nil Nil NULL NSNull 之间的区别

**nil** 是对objective c id 对象赋空值  
**Nil**: 表示对类进行赋空值  
**NULL**: 用于对非对象指针赋空值，比如C指针  
**NSNull** 对于像NSArray这样的类型，nil或NULL不能做为加到其中的Object，如果定义了一个NSArray，为其分配了内存，又想设置其中的内容为空，则可以用[NSNULL null】返回的对象来初始化NSArray中的内容

# 数据结构

# 设计模式

## 设计原则

### 单一职责原则

一个类只负责一个功能领域中相应的职责

### 开闭原则

一个类或者软件实体尽量实现对扩展开发，对修改关闭

### 里氏代换

所有引用基类（父类）的地方，都能够使用子类的对象

### 依赖倒转

具体依赖抽象，而不是抽象依赖具体。

### 接口隔离

### 合成复用

### 迪米特法则

## 创建型

### 工厂模式

## 行为型

## 策略模式

### 组成成分：

1. 抽象策略类
2. 具体策略类
3. 策略上下文

# 技术论坛地址

1. <http://nshipster.cn/>

# 牛人技术博客地址

1. 苹果源码<http://opensource.apple.com/source/>
2. <http://www.cnblogs.com/ludashi/p/4925042.html>
3. <http://limboy.me/ios/2014/12/31/learning-ios.html>
4. <http://cnbin.github.io/blog/2015/05/19/iospei-zhi-ffmpegkuang-jia/>
5. <http://www.cocoachina.com/ios/20150104/10823.html>
6. <http://casatwy.com/iosying-yong-jia-gou-tan-wang-luo-ceng-she-ji-fang-an.html>
7. <http://www.henishuo.com/ios-block-memory-cycle/?utm_source=tuicool&utm_medium=referral>
8. <http://www.cnblogs.com/mjios/p/4409853.html>
9. <https://blog.cnbluebox.com/blog/2014/07/01/cocoashen-ru-xue-xi-nsoperationqueuehe-nsoperationyuan-li-he-shi-yong/>
10. <http://www.jianshu.com/p/c47c24ab1e76>
11. <http://weibo.com/5612984599/profile?rightmod=1&wvr=6&mod=personinfo&is_hot=1#_loginLayer_1464445659010>
12. <http://casatwy.com/zen-yao-mian-shi-jia-gou-shi.html>
13. <http://blog.cnbluebox.com>
14. <http://my.oschina.net/u/566401/blog/219568>
15. <http://blog.sunnyxx.com/archives/page/3/>
16. <http://my.oschina.net/caijunrong/blog/512372>
17. <http://www.jianshu.com/users/256fb15baf75/latest_articles>
18. <http://www.jianshu.com/p/8f16613861fa>
19. <http://www.jianshu.com/users/0a7ce85a2708/latest_articles>
20. <http://www.jianshu.com/p/2113ffe54b30>
21. <http://www.jianshu.com/collection/6930cb8ad81b>
22. <http://www.cnblogs.com/zy1987/>
23. <http://www.cnblogs.com/wengzilin/>
24. <http://www.jianshu.com/p/3ea737b1e2e8>

# 面试题集锦

1. iOS面试必看，最全梳理<http://www.cocoachina.com/ios/20160323/15770.html>

# 面试题学习记录

## 深复制和浅复制

深复制：复制内存对象，新旧互不影响。

浅复制：复制内存对象指针，两个指针同指向一个内存。

对于子类与父类的关系是：

现深拷贝需要实现NSCoying协议，实现- (id)copyWithZone:(NSZone \*)zone 方法。当对一个property属性含有copy修饰符的时候，在进行赋值操作的时候实际上就是调用这个方法。

### KVC KVO

Key Value Coding

Key Value Coding是cocoa的一个标准组成部分，它能让我们可以通过name(key)的方式访问property, 不必调用明确的property accssor, 如我们有个property叫做foo, 我们可以foo直接访问它，同样我们也可以用KVC来完成[Object valueForKey:@“foo”], 有同学就会问了， 这样做有什么好处呢？主要的好处就是来减少我们的代码量。

下面我们来看看几个例子，就明白了KVO的用法和好处了，假设这样个类叫做People,

@interface People: NSObject

@property (nonatomic, strong) NSString \*name;

@property (nonatomic, strong) NSNumber \*age;

@end

场景1，apple 官网的一个例子，当我们需要统计很多People的时候，每一行是一个人的实例，并且有2列属性，name, age, 这时候我们可以会这样做，

- (id)tableView:(NSTableView \*)tableview

objectValueForTableColumn:(id)column row:(NSInteger)row {

People \*people = [peoleArray objectAtIndex:row];

if ([[column identifier] isEqualToString:@"name"]) {

return [people name];

}

if ([[column identifier] isEqualToString:@"age"]) {

return [people age];

}

// And so on.

}

同样我们也可以用KVC，帮助我们化简这些if, 因为name, age其实都是property, 我们可以直接通过key来访问，所以整理过后是

People \*people = [peopleArray objectAtIndex:row];

return [people valueForKey:[column identifier]];

场景2，这下我们有了server, server的某个api(listPeople??), 会返回我们json格式一个数组，里面包含这样dict{name:xx, age:xx}这样的数据， 我们希望用这些dict数据构造出我们的people来，通常我们的做法是，为我们People类写一个static factory方法专门用来处理dict来， 把dict里面的数据取出来, 然后创建个空的People对象，然后依次设置property。然而当这样类似People的与server交互的类多了，我们就要为每个类都要加上这样的wrapper, 是否有种简单办法来设置这样的属性，当然就是我们的KVC了。

-(id) initWithDictionary:(NSMutableDictionary\*) jsonObject

{

if((self = [super init]))

{

[self init];

[self setValuesForKeysWithDictionary:jsonObject];

}

return self;

}

setValuesForKeysWithDictionary, 会为我们把和dictionary的key名字相同的class proerty设置上dict中key对应的value, 是不是很方便呀，但是有同学又要问了 如果json里面的某些key就是和object的property名字不一样呢，或者有些server返回的字段是objc保留字如”id”, “description”等， 我们也希望也map dict to object, 这时候我们就需要用上setValue:forUndefinedKey, 因为如果我们不处理这些Undefined Key，还是用setValuesForKeysWithDictionary就会 抛出异常。

- (void)setValue:(id)value forUndefinedKey:(NSString \*)key

{

if([key isEqualToString:@"nameXXX"])

self.name = value;

if([key isEqualToString:@"ageXXX"])

self.age = value;

else

[super setValue:value forKey:key];

}

所以只要重载这个方法，就可以处理了那些无法跟property相匹配的key了，默认的实现是抛出一个NSUndefinedKeyException，又有同学发问了如果 这时候server返回的People有了内嵌的json(如Products{product1{count:xx, sumPrice:xx}}, product2{} ….)，又该怎么办，能把这个内嵌的json转化成我们的客户端的Product类嘛， 当然可以这时候就需要重载setValue:forKey, 单独处理”Products”这个key, 把它wrapper成我们需要的class

-(void) setValue:(id)value forKey:(NSString \*)key

{

if([key isEqualToString:@"products"])

{

for(NSMutableDictionary \*productDict in value)

{

Prodcut \*product = [[Product alloc] initWithDictionary:prodcutDict];

[self.products addObject:product];

}

}

}

场景3，我们需要把一个数组里的People的名字的首字母大写，并且把新的名字存入新的数组， 这时候通常做法会是遍历整个数组，然后把每个People的name取出来，调用 capitalizedString 然后把新的String加入新的数组中。 有了KVC就有了新做法:

[array valueForKeyPath:@"name.capitalizedString"]

我们看到valueForKeyPath, 为什么用valueForKeyPath, 不用valueForKey, 因为valueForKeyPath可以传递关系，例如这里是每个People的name property的String的capitalizedString property， 而valueForKey不能传递这样的关系，所以对于dict里面的dict, 我们也只能用valueForKeyPath。这里我们也看到KVC对于array(set), 做了特殊处理，不是简单操作collection上，而是 针对这些collection里面的元素进行操作，同样KVC也提供更多地操作，例如@sum这些针对collection，有兴趣的同学可以去用下。

场景4，当我们执行NSArray \*products = [people valueForKey:@“products”]，我们希望的是[people products]，可是people没有这样的方法， KVC又会为我们带来些什么呢？

首先会去找getProdcuts or products or isProducts, 按照这样的顺序去查找，第一个找到的就返回

然后会去找countOfProducts and either objectInProductsAtIndex: or ProductsAtIndexes, 如果找到，就会去找countOfProducts and enumeratorOfProducts and memberOfProducts 这个2个方法都找到了，KVC才会给我们返回一个代理的NSKeyValueArray，用于我们后续的操作（addProduct之类的）。

如果有个变量叫做 products, isProducts, products or isProducts, KVC会直接就使用这样的变量，如果你觉得直接用这样的变量是破坏了封装， 可以禁止这样的行为发生，重载 +accessInstanceVariablesDirectly，返回NO。

简单来说，valueForKey, 会给我们带来一个代理array, 如果我们实现了某些方法，上诉的这些方法只是针对NSArray, 对于mutable的collection, 我们还需要提供其他 方法的实现才行。

Key Value Observing

Key Value Observing, 顾名思义就是一种observer 模式用于监听property的变化，KVO跟NSNotification有很多相似的地方， 用addObserver:forKeyPath:options:context:去start observer, 用removeObserver:forKeyPath:context去stop observer, 回调就是observeValueForKeyPath:ofObject:change:context:。

- (void)removeObservation {

[self.object removeObserver:self

forKeyPath:self.property];

}

- (void)addObservation {

[self.object addObserver:self forKeyPath:self.property

options:0

context:(\_\_bridge void\*)self];

}

- (void)observeValueForKeyPath:(NSString \*)keyPath

ofObject:(id)object

change:(NSDictionary \*)change

context:(void \*)context {

if ((\_\_bridge id)context == self) {

// 只处理跟我们当前class的property更新

}

else {

[super observeValueForKeyPath:keyPath ofObject:object

change:change context:context];

}

}

对于KVO来说，我们要做的只是简单update 我们的property数据，不需要像NSNotificationCenter那样关心是否有人在监听你的请求，如果没有人监听该怎么办， 所有addObserver, removeObserver, callback 都是想要监听的你的property的class做的事情。 曾经做个项目，用NSNotificationCenter post Notification在一个network callback里面，可是这时候因为最早的addObserver的class被释放了， 接着生成的addObserver的class, 就接受到了上一个observer该监听的事件，所以造成了错误，那时候的解决方案是为addObserve key做unique，不会2次addObserver 的key是相同的，但是有了KVO, 我们同样可以用KVO来完成，当addOberver的的object remove的时候，就不会有这样的callback被调用了。

KVO给我们提供了更少的代码，和比NSNotification好处，不需要修改被观察的class, 永远都是观察你的人做事情。 但是KVO也有些毛病， 1. 如果没有observer监听key path, removeObsever:forKeyPath:context: 这个key path, 就会crash, 不像NSNotificationCenter removeObserver。 2. 对代码你很难发现谁监听你的property的改动，查找起来比较麻烦。 3. 对于一个复杂和相关性很高的class，最好还是不要用KVO, 就用delegate 或者 notification的方式比较简洁。

Summary

尽量使用KVC可以大大地减少我们的代码量，当遇到property的时候，可以多想想是否可以KVC来帮助我，是否可以用KVC来重构代码， 当需要加入observer模式时，可以考虑下KVO, 在高性能的observer里面，KVO会给我们很好的帮助。