iOS开发总结之代码规范

2015-10-14 09:47 编辑： [suiling](http://www.cocoachina.com/bbs/u.php?action=feed&uid=340873" \t "_blank) 分类：[iOS开发](http://www.cocoachina.com/ios/" \t "_blank) 来源：[RylanJIN投稿](http://www.jianshu.com/p/414bb5a53139" \t "_blank)

0 7713

[代码规范](http://www.cocoachina.com/cms/tags.php?/%E4%BB%A3%E7%A0%81%E8%A7%84%E8%8C%83/" \t "_blank)[iOS代码规范](http://www.cocoachina.com/cms/tags.php?/iOS%E4%BB%A3%E7%A0%81%E8%A7%84%E8%8C%83/" \t "_blank)[命名规范](http://www.cocoachina.com/cms/tags.php?/%E5%91%BD%E5%90%8D%E8%A7%84%E8%8C%83/" \t "_blank)



本文是投稿文章，作者：[RylanJIN](http://www.jianshu.com/users/e08797423685/latest_articles" \t "_blank)

最近被安排fix项目的随机crash问题, 大大小小修复了差不多10个issue, 总结一下发现这些问题或多或少都是由代码习惯和编程规范引起的, 可见一个好的编码习惯是多么的重要! 趁着这两天休假将自己所认为的一些比较好的代码规范整理一下, 并结合之前遇到的实际case跟大家分享一下.

**命名规范**

总的来说, iOS命名两大原则是:可读性高和防止命名冲突(通过加前缀来保证). Objective-C 的命名通常都比较长, 名称遵循驼峰式命名法. 一个好的命名标准很简单, 就是做到在开发者一看到名字时, 就能够懂得它的含义和使用方法. 另外, 每个模块都要加上自己的前缀, 前缀在编程接口中非常重要, 可以区分软件的功能范畴并防止不同文件或者类之间命名发生冲突, 比如相册模块(PhotoGallery)的代码都以PG作为前缀: PGAlbumViewController, PGDataManager.

**1). 常量的命名**

对于常量的命名最好在前面加上字母k作为标记. 如:

|  |  |
| --- | --- |
| 1 | static const NSTimeInterval kAnimationDuration = 0.3; |

定义作为NSDictionary或者Notification等的Key值字符串时加上const关键字, 以防止被修改. 如:

|  |  |
| --- | --- |
| 1 | NSString \*const UIApplicationDidEnterBackgroundNotification |

Tips:

I. 若常量作用域超出编译单元(实现文件), 需要在类外可见时, 使用extern关键字, 并加上该类名作为前缀. 如 extern NSString \*const PGThumbnailSize

II.全局常量(通知或者关键字等)尽量用const来定义. 因为如果使用宏定义, 一来宏可能被重定义. 二来引用不同的文件可能会导致宏的不同. P.S. 对于＃define也添加一下前缀k(强迫症, 哈哈...)

**2). 枚举的命名**

对于枚举类型, 经常会看到之前的C的定义方式:

|  |  |
| --- | --- |
| 1  2  3  4  5 | typedef enum : {      CameraModeFront,      CameraModeLeft,      CameraModeRight,  } CameraMode; |

不知道是肿么了, 每次看到这种定义方式总是感觉怪怪的, 作为一个正宗的iOS开发者当然要以Objective-C的方式来定义啦, 哈哈... 那Objective-C是怎么定义的呢? 很简单, 到SDK里面看看Apple是怎么做滴:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | typedef NS\_ENUM(NSInteger, UIViewAnimationTransition) {      UIViewAnimationTransitionNone,      UIViewAnimationTransitionFlipFromLeft,      UIViewAnimationTransitionFlipFromRight,      UIViewAnimationTransitionCurlUp,      UIViewAnimationTransitionCurlDown,  }; |

这边需要注意的是: 枚举类型命名要加相关类名前缀并且枚举值命名要加枚举类型前缀.

**3). 变量和对象的命名**

给一个对象命名时建议采用修饰+类型的方式. 如果只用修饰命名会引起歧义, 比如title (这个到底是个NSString还是UILabel?). 同样的, 如果只用类型来命名则会缺失作用信息, 比如label (好吧, 我知道你是个UILabel, 但是我不知道它是用来做什么的呀?). So, 正确的命名方式为:

|  |  |
| --- | --- |
| 1  2 | titleLabel    //表示标题的label,  是UILabel类型  confirmButton //表示确认的button, 是UIButton类型 |

对于BOOL类型, 应加上is前缀, 比如- (BOOL)isEqualToString:(NSString \*)aString这样会更加清晰. 如果某方法返回非属性的 BOOL 值, 那么应根据其功能, 选用 has 或 is 当前缀, 如- (BOOL)hasPrefix:(NSString \*)aString

Tip: 如果某个命名已经很明确了, 为了简洁可以省去类型名. 比如scores, 很明显是个array了, 就不必命名成scoreArray了

**编码规范**

编码规范简单来说就是为了保证写出来的代码具备三个原则:可复用, 易维护, 可扩展. 这其实也是面向对象的基本原则. 可复用, 简单来说就是不要写重复的代码, 有重复的部分要尽量封装起来重用. 否则修改文件的时候得满地找相同逻辑的地方...这个就用no zuo no die来描述吧, 哈哈...易维护, 就是不要把代码复杂化, 不要去写巨复杂逻辑的代码, 而是把复杂的逻辑代码拆分开一个个小的模块, 这也是Do one thing的概念, 每个模块(或者函数)职责要单一, 这样的代码会易于维护, 也不容易出错. 可扩展则是要求写代码时要考虑后面的扩展需求, 这个属于架构层面的东东, 利用对应的设计模式来保证, 后面有机会单独写文探讨。

编码规范直接通过示例来介绍, 毕竟对于程序员来说一个Demo胜过千行文字(有同感的小伙伴让我看到你们的双手, 哈哈O(∩\_∩)O~~). 下面的部分示例选自richieyang博文, 写的很好的一篇文章, 推荐大家看一下, 我自己也是受益匪浅.

**1). 判断nil或者YES/NO**

Preferred:

|  |  |
| --- | --- |
| 1  2 | if (someObject) { ... }  if (!someObject) { ... } |

Not preferred:

|  |  |
| --- | --- |
| 1  2 | if (someObject == YES) { ...}  if (someObject != nil) { ...} |

if (someObject == YES)容易误写成赋值语句, 自己给自己挖坑了...而且if (someObject)写法很简洁, 何乐而不为呢?

**2). 条件赋值**

Preferred:

|  |  |
| --- | --- |
| 1 | result = object ? : [self createObject]; |

Not preferred:

|  |  |
| --- | --- |
| 1 | result = object ? object : [self createObject]; |

如果是存在就赋值本身, 那就可以这样简写, 多简洁啊, 哈哈...

**3). 初始化方法**

Preferred:

|  |  |
| --- | --- |
| 1  2  3  4 | NSArray \*names = @[@"Brian", @"Matt", @"Chris", @"Alex", @"Steve"];  NSDictionary \*productManagers = @{@"iPhone" : @"Kate", @"iPad" : @"Kamal"};  NSNumber \*shouldUseLiterals = @YES;  NSNumber \*buildingZIPCode = @10018; |

第一个好处还是简洁, 第二个好处是可以防止初始化进去nil值造成crash

**4). 定义属性**

Preferred:

|  |  |
| --- | --- |
| 1 | @property (nonatomic, readwrite, copy) NSString \*name; |

建议定义属性的时候把所有的参数写全, 尤其是如果想定义成只读的(防止外面修改)那一定要加上readonly, 这也是代码安全性的一个习惯.

如果是内部使用的属性, 那么就定义成私有的属性(定义到.m的class extension里面)

对于拥有Mutable子类型的对象(e.g. NSString, NSArray, NSDictionary)一定要定义成copy属性. Why? 示例: NSArray的array = NSMutableArray的mArray; 如果mArray在某个地方改变了, 那array也会跟着改变. So, make sense?

尽量不要暴露mutable类型的对象在public interface, 建议在.h定义一个Inmutable类型的属性, 然后在.m的get函数里面返回一个内部定义的mutable变量. Why? For security as well!

**5). BOOL赋值**

Preferred:

|  |  |
| --- | --- |
| 1 | BOOL isAdult = age > 18; |

Not preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | BOOL isAdult;  if (age > 18)  {      isAdult = YES;  }  else  {      isAdult = NO;  } |

为什么要这么写呢, 我不告诉你, 哈哈哈...

**6) 拒绝死值**

Preferred:

|  |  |
| --- | --- |
| 1  2  3 | if (car == Car.Nissan)  or  const int adultAge = 18; if (age > adultAge) { ... } |

Not preferred:

|  |  |
| --- | --- |
| 1  2  3 | if (carName == "Nissan")  or  if (age > 18) { ... } |

死值每次修改的时候容易被遗忘, 地方多了找起来就悲剧了. 而且定义成枚举或者static可以让错误发生在编译阶段. 另外仅仅看到一个数字, 完全不知道这个数字代表的意义. 纳尼?

**7). 复杂的条件判断**

Preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | if ([self canDeleteJob:job]) { ... }    - (BOOL)canDeleteJob:(Job \*)job  {      BOOL invalidJobState = job.JobState == JobState.New                            || job.JobState == JobState.Submitted                            || job.JobState == JobState.Expired;      BOOL invalidJob = job.JobTitle && job.JobTitle.length;        return invalidJobState || invalidJob;  } |

Not preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | if (job.JobState == JobState.New      || job.JobState == JobState.Submitted      || job.JobState == JobState.Expired      || (job.JobTitle && job.JobTitle.length))  {      //....  } |

清晰明了, 每个函数DO ONE THING!

**8). 嵌套判断**

Preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if (!user.UserName) return NO;  if (!user.Password) return NO;  if (!user.Email) return NO;    return YES; |

Not preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | BOOL isValid = NO;  if (user.UserName)  {        if (user.Password)      {            if (user.Email) isValid = YES;      }    }  return isValid; |

一旦发现某个条件不符合, 立即返回, 条理更清晰

**9). 参数过多**

Preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | - (void)registerUser(User \*user)  {         // to do...    } |

Not preferred:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | - (void)registerUserName:(NSString \*)userName                  password:(NSString \*)password                     email:(NSString \*)email  {       // to do...  } |

当发现实现某一功能需要传递的参数太多时, 就预示着你应该聚合成一个model类了...这样代码更整洁, 也不容易因为参数太多导致出错。

**10). 回调方法**

Preferred:

|  |  |
| --- | --- |
| 1 | - (UITableViewCell \*)tableView:(UITableView \*)tableView cellForRowAtIndexPath:(NSIndexPath \*)indexPath; |

函数调用的可知性, 回调时被调用者要知道其调用者, 方便信息的传递, 所以建议在回调方法中第一个参数中加上调用者。

Well, 不知不觉已经整理了10个了, 额, 太多了, 不知道童鞋们还有木有耐心看了, 好吧, 这一段就到此为止吧, 下面写一下block的编码规范, 各位看官, 预知后事如何, 且继续look look, 哈哈...

**Block的循环引用问题**

Block确实是个好东西, 但是用起来一定要注意循环引用的问题, 否则一不小心你就会发现, Oh My God, 我的dealloc肿木不走了...

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \_\_weak typeof(self) weakSelf = self;  dispatch\_block\_t block =  ^{      [weakSelf doSomething]; // weakSelf != nil      // preemption, weakSelf turned nil      [weakSelf doSomethingElse]; // weakSelf == nil  }; |

如此在上面定义一个weakSelf, 然后在block体里面使用该weakSelf就可以避免循环引用的问题. 那么问题来了...是不是这样就完全木有问题了? 很不幸, 答案是NO, 还是有问题。问题是block体里面的self是weak的, 所以就有可能在某一个时段self已经被释放了, 这时block体里面再使用self那就是nil, 然后...然后就悲剧了...那么肿么办呢?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | \_\_weak typeof(self) weakSelf = self;  myObj.myBlock =  ^{      \_\_strong typeof(self) strongSelf = weakSelf;      if (strongSelf) {        [strongSelf doSomething]; // strongSelf != nil        // preemption, strongSelf still not nil        [strongSelf doSomethingElse]; // strongSelf != nil      }      else {          // Probably nothing...          return;        }  }; |

解决方法很简单, 就是在block体内define一个strong的self, 然后执行的时候判断下self是否还在, 如果在就继续执行下面的操作, 否则return或抛出异常.

什么情况下会出现block里面self循环引用的问题? 这个问题问的好, 哈哈...简单来说就是双边引用, 如果block是self类的property (此时self已经retain了block), 然后在block内又引用了self, 这个情况下就肯定会循环引用了...

P.S. RAC里面有定义好的@weakify(self)和@strongify(self), 用起来灰常灰常的方便, 劝君尝试一下^\_^

**那些年遇到的Crash**

* 多线程同步问题造成的Crash

这个其实还蛮常见的, 尤其是在多线程泛滥使用的今天...你可以使用多线程, 但你要知道保护它呀, 哈哈. 对于数据源或model类一定要注意多线程同时访问的情况, 我个人比较喜欢用GCD的串行队列来同步线程.

* Observer的移除

现在的代码里面很多需要用到Observer, 根据被观察对象的状态来相应的Update UI或者执行某个操作. 注册observer很简单, 但是移除的时候就出问题了, 要么是忘记移除observer了, 要么是移除的时机不对. 如果某个被观察对象已经被释放了, observer还在, 那结果只能是crash了, 所以切记至少在dealloc里面移除一下observer...

* NSArray, NSDictionary成员的判空保护

在addObject或insertObject到NSArray或者NSDictionary时最好加一下判空保护, 尤其是网络相关的逻辑, 如果网络返回为空(jason解析出来为空), 但你还是毅然决然的add到array里面, 那么...

最后一点就是commit代码之前一定要保证木有warning, 木有内存泄露, 确保都OK之后再上传代码. 其实很简单, 上传代码之前Command + Shift + B静态分析一下, 看看有木有什么issue...就先写这么多吧, 以后遇到更多的坑后, 我一定会爬出来再过来补充的, to be continued...