一份走心的iOS开发规范

本文的开发规范由很多item组成，不同的item描述了不同的问题。每一个item就是一条具体的开发规范，违反不同的开发规范，也会引起不同严重程度的后果。

前言

说是前言，其实也是本文诞生的目的。随着公司业务的不断增加，功能的快速迭代，app的业务线越来越多，代码体积变得越来越庞大。同时，app投入的开发者也也越来越多，不同的开发者的code风格千差万别。加之公司开发者人员变动，为了保证app稳定性，保证开发效率，统一开发风格。于是，这篇iOS开发规范应运而生。

因笔者现在所就职公司的开发规范主导编写，目前公司业务的迭代都在按照这个规范在有条不紊的进行。综合之前编写规范的经验，历时一个月的时间，断断续续重新梳理了一份比较全面、比较完整的iOS开发者规范，希望这些条条框框能够给正在阅读的你提供一些参考的价值。也希望越来越多的iOS开发者能够养成优秀的编码习惯。如果你觉得个别地方不妥或者有需要补充的规范，请留言或者私信，我会第一时间响应。

约定

在我看来，开发规范像是一条可供参考的标准线。不同开发者可以根据这条标准线来规范自己的开发行为，尤其是在大的项目中，开发规范可以约束不同开发者的开发风格，使项目从细节到整体上都能达到风格统一，利于维护。

本文的开发规范由很多item组成，不同的item描述了不同的问题。每一个item就是一条具体的开发规范，违反不同的开发规范，也会引起不同严重程度的后果。就像法律和道德的差异一样，我们必须遵守法律，不然可能带来损人不利己的严重后果，但有些人虽然没有触犯法律，却违背了道德，虽然暂时没有产生严重的后果，长此以往，也会形成一种坏的风气。所以，无论法律和道德，我们都该鞭策自己成为优秀的人，而不该止步于一个合格的人。同理，开发规范也是如此，我们必须遵守那些必须要遵守的开发规范，提倡遵守那些建议你遵守的开发规范。所以，根据约束力度，我们把开发规范暂时划分成两个等级，分别是【必须】、【建议】。

1. 【必须】：必须遵守。是不得不遵守的约定，一旦违反极有可能引起严重后果。
2. 【建议】：建议遵守。长期遵守这样的约定，有助于维护系统的稳定和提高合作效率。

本文参考了苹果官方编码指南和github上一些知名的编码规范，也算是取众

人之所长，集百家之精华的一篇文章。读者可以根据自己的实际需要和兴趣点来选择性的阅读。本文主题部分主要由以下两章(共32节)构成：

(一) 命名规范

1. 通用命名规范(讲述命名的一些通用规范)
2. 缩写规范(讲述常见的缩写以及缩写规范)
3. Method命名规范(讲述方法命名的具体规范)
4. Accessor命名规范(讲述set和get方法的命名规范)
5. Parameter命名规范(讲述参数命名规范)
6. Delegate方法命名规范(讲述delegate方法的命名规范)
7. Private方法命名规范(讲述私有方法的命名规范)
8. Category命名规范(讲述分类的命名规范)
9. Class命名规范(讲述类命名规范)
10. Protocol命名规范(讲述协议的命名规范)
11. Notification命名规范(讲述通知的命名规范)
12. Constant命名规范(讲述枚举常量以及const常量的命名规范)
13. Exception命名规范(讲述异常的命名规范)

(二)编码规范

1. Initialize方法(讲述类的initialize方法的使用规范)
2. Init方法(讲述初始化方法的设计规范包括designated init方法和secondary init方法)
3. Init error(讲述init方法初始化对象失败时的错误处理)
4. Dealloc规范(讲述dealloc方法的使用规范)
5. Block规范(讲述block的使用规范)
6. Notification规范(讲述通知的使用规范)
7. UI规范(讲述开发UI时的一些规范)
8. IO规范(讲述读写文件时的一些注意事项)
9. Collection规范(讲述集合类型的使用规范)
10. 分支语句规范(讲述常用的分支语句if、switch语句的编码规范)
11. 对象判等规范(讲述常用的判定对象等同性的方法使用规范)
12. 懒加载规范(讲述懒加载的使用规范)
13. 多线程规范(讲述多线程环境下的一些编码规范)
14. 内存管理规范(讲述编码过程中常见的内存管理注意点)

15. 延迟调用规范(讲述使用延迟方法时注意事项)
16. 注释规范(讲述编码中注释的使用规范)
17. 类的设计规范(讲述类的设计规范)
18. 代码组织规范(讲述类中的代码组织规范)
19. 工程结构规范(讲述工程的文件组织规范)

(一)命名规范

根据Cocoa编码规范里的描述，以前情况下，命名应该遵循以下基本原则：Clarity、Consistency、No Self Reference。

(1.1) 通用命名规则

一般情况下，通用命名规则适用于变量、常量、属性、参数、方法、函数等。当然也有例外，下面我们会针对于每一种情况一一列举。

【必须】自我描述性。属性/函数/参数/变量/常量/宏 的命名必须具有自我描述性。杜绝中文拼音、过度缩写、或者无意义的命名方式。

【必须】禁止自我指涉。属性/局部变量/成员变量不要自我指涉。通知和掩码常量(通常指那些可以进行按位运算的枚举值) 除外。

通俗的讲，自我指涉是指在变量末尾增加了自己类型的一个后缀。

命名	说明
NSString	规范的写法
NSStringObject	自我指涉（不规范）

掩码常量、通知除外：

命名	说明
NSUnderlineByWordMask	规范的写法
NSTableViewColumnDidMoveNotification	规范的写法

【必须】驼峰命名方式。参数名、成员变量、局部变量、属性名都要采用小写字母开头的驼峰命名方式。如果方法名以一个众所周知的大写缩略词开始，可以不适用驼峰命名方式。比如FTP、WWW等。

【建议】一致性。属性/函数/参数/变量/常量/宏 的命名应该具有上下文或者全局的一致性，相同类型或者具有相同作用的变量的命名方式应该相同或者类似。

说明：具体来讲，不同文件中或者不同类中具有相同功能或相似功能的属性的命名应该是相同的或者相似的。好处在于：方便后来的开发者减少代码的阅读量和提高对代码的理解速度。比如：

```
1. // count同时定义在NSDictionary、NSArray、NSSet这三个集合类中。且这三个集合类中
2. @property (readonly) NSUInteger count;
```

【必须】清晰性。属性/函数/参数/变量/常量/宏 的命名应该保持清晰+简洁，如果鱼和熊掌不能兼得，那么清晰更重要。

命名	说明
insertObject:atIndex:	规范的写法
insert:at:	不清晰，插入什么？at代表什么？
removeObjectAtIndex:	规范的写法
removeObject:	规范的写法，因为参数指明了要移除一个对象
remove:	不清晰，移除什么？

【建议】一般情况下，不要缩写或省略单词，建议拼写出来，即使它有点长。当然，在保证可读性的同时，for循环中遍历出来的对象或者某些方法的参数可以缩写。

命名	说明
destinationSelection	规范写法
destSel	不清晰
setBackgroundColor:	规范写法
setBkgdColor:	不清晰

(1.2) 缩写规范

通常，我们都不应该缩写命名。然而，下面所列举的都是一些众所周知的缩写，我们可以继续使用这些古老的缩写。在其他情况下，我们需要遵循下面两条缩写建议：

- 允许使用那些在C语言时代就已经在使用的缩写，比如alloc和getc。
- 我们可以在命名参数的时候使用缩写。其他情况，尽量不要使用缩写。

我们也可以使用计算机行业通用的缩写。包括但不限于HTML、URL、RTF、HTTP、TIFF、JPG、PNG、GIF、LZW、ROM、RGB、CMYK、MIDI、FTP。

(1.3) Method命名规范

【必须】方法名也要采用小写字母开头的驼峰命名方式。如果方法名以一个中所周知的大写缩略词开头(比如HTTP)，该规则可以忽略。

【建议】一般情况下，不要在方法名称中使用前缀，因为他存在于特定类的命名空间中。

【建议】类、协议、函数、常量、枚举等全局可见内容需要添加三个字符作为前缀。苹果保留对任意两个字符作为前缀的使用权。所以尽量不要使用两个字符作为前缀。禁止使用的前缀包括但不限于：
NS,UI,CG,CF,CA,WK,MK,CI,NC。

【必须】禁止在方法前面加下划线“_”。Apple官网团队经常在方法前面加下划线"_"。为了避免方法覆盖，导致不可预知的意外，禁止在方法前面加下划线。

【必须】自我描述性。方法的命名也应该具有自我描述性。杜绝中文拼音、过度缩写、或者无意义的命名方式。

【建议】一致性。方法的命名也应该具有上下文或者全局的一致性，相同类型或者具有相同作用的方法的命名方式应该相同或者类似。

```
1. // 该方法同时定义在NSView、NSControl、NSCell这三个类里面。
2. - (NSInteger)tag;
3. // 该属性同时定义在NSDcitionary和NSArray中。
4. @property (readonly) NSUInteger count;
```

【必须】苹果爸爸说：如果一个方法代表某个名词执行的动作，则该方法应该以一个动词开头。如下：

```
1. - (void)invokeWithTarget:(id)target;
2. - (void)selectTabViewItem:(NSTabViewItem *)tabViewItem
```

【必须】苹果爸爸还说：如果方法代表对象接收的动作，那么方法一动词开头。但不要使用“do”或者"does"作为方法名称的一部分，因为这些助动词不能为方法名称增加太多的意义，反而让方法看起来更加臃肿。同时，也请不要在动词前面使用副词或者形容词。

【必须】如果方法返回接收者的某个属性，那么请直接以属性名作为方法名。如果方法间接的返回一个或多个值，我们可以使用“getxxx”的方式来命名方法。相反，无需额外的在方法名前面添加"get"。

命名	说明
- (NSSize)cellSize;	OK
- (NSSize)calcCellSize;	不OK
- (NSSize)getCellSize;	不OK

【必须】只有当方法间接的返回对象或数值，才有必要在方法名称中使用“get”，这种格式只适用于返回多个数据项的情况。如下：

```
1. // 通过传入指针，来获得多个值
2. - (void)getLineDash:(float *)pattern count:(int*)count phase:(float *)
3. // NSURLCache (NSURLSessionTaskAdditions)中声明的方法
4. - (void)getCachedResponseForDataTask:(NSURLSessionDataTask *)dataTask
```

【必须】所有参数前面都应该添加关键字，除非你能保证每个人都能意会到你的精神。

命名	说明
- (void)sendAction:(SEL)aSelector toObject:(id)anObject forAllCells:(BOOL)flag;	OK
- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;	不OK

【建议】苹果爸爸说：参数之前的单词尽量能描述参数的意义。

命名	说明
- (id)viewWithTag:(NSInteger)aTag;	OK
- (id)taggedView:(int)aTag;	不OK

【必须】如果当前子类创建的方法比从父类继承来的方法更加具体明确。本身提供的方法更具有针对性。则不该重写类本身提供的方法。而是应该单独的提供一个方法，并在新的方法后面添加上必要的关键参数。

命名	说明
- (id)initWithFrame:(CGRect)frameRect;	NSView, UIView.
- (id)initWithFrame:(NSRect)frameRect mode:(int)aMode cellClass:(Class)factoryId numberOfRows:(int)rowsHigh numberOfColumns:(int)colsWide;	NSMatrix, a subclass of NSView

```
1. // UIView提供的方法
2. - (instancetype)initWithFrame:(CGRect)frame
3. // 更具针对性的方法
4. - (instancetype)initWithFrame:(CGRect)frame mode:(int)aMode cellClass:
```

【建议】请不要使用“and”连接接收者属性。尽管and在下面的例子中读起来还算顺口，但随着你创建的方法参数的增加，这将会带来一系列的问题。

命名	说明
- (int)runModalForDirectory:(NSString *)path file:(NSString *) name types:(NSArray *)fileTypes;	OK
- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes;	不OK

【建议】如果方法描述了两个独立的动作，可以使用“and”连接起来。

命名	说明
- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes;	OK（NSWorkspace.）

(1.4) Accessor命名规范

Accessor Methods是指set、get方法。这些方法有一些推荐写法格式：

【建议】如果属性是名词，推荐格式如下：

```
1. - (type)noun;
2. - (void)setNoun:(type)aNoun;
3. 例如：
4. - (NSString *)title;
5. - (void)setTitle:(NSString *)aTitle;
```

【建议】如果属性表示一个形容词，推荐格式如下：

```
1. - (BOOL)isAdjective;
2. - (void)setAdjective:(BOOL)flag;
3. 例如：
4. - (BOOL)isEditable;
5. - (void)setEditable:(BOOL)flag;
```

【建议】如果属性是一个动词，动词使用一般现在时。推荐格式如下：

```
1. - (BOOL)verbObject;
2. - (void)setVerbObject:(BOOL)flag;
3. 例如：
4. - (BOOL)showsAlpha;
5. - (void)setShowsAlpha:(BOOL)flag;
```

【必须】不要把动词的过去分词形式当做形容词来使用。

命名	说明
- (void)setAcceptsGlyphInfo:(BOOL)flag;	OK
- (BOOL)acceptsGlyphInfo;	OK
- (void)setGlyphInfoAccepted:(BOOL)flag;	不OK
- (BOOL)glyphInfoAccepted;	不OK

命名	说明
- (void)setCanHide:(BOOL)flag;	OK
- (BOOL)canHide;	OK
- (void)setShouldCloseDocument:(BOOL)flag;	OK
- (BOOL)shouldCloseDocument;	OK
- (void)setDoesAcceptGlyphInfo:(BOOL)flag;	不OK
- (BOOL)doesAcceptGlyphInfo;	不OK

【建议】可以使用情态动词(can、should、will等)明确方法意义，但不要使用do、does这类无意义的情态动词。

命名	说明
- (void)setCanHide:(BOOL)flag;	OK
- (BOOL)canHide;	OK
- (void)setShouldCloseDocument:(BOOL)flag;	OK
- (BOOL)shouldCloseDocument;	OK
- (void)setDoesAcceptGlyphInfo:(BOOL)flag;	不OK
- (BOOL)doesAcceptGlyphInfo;	不OK

【建议】只有方法间接的返回一个数值，或者需要多个数值需要被返回的时候，才有必要在方法名称中使用“get”。

像这种接收多个参数的方法应该能够传入nil，因为调用者未必对每个参数都感兴趣

```
1. - (void)getLineDash:(float *)pattern count:(int *)count phase:(float *
```

(1.5) Parameter命名规范

【必须】不要使用”pointer”或”ptr”命名参数，应该使用参数类型而非它的名字来代表他是否是一个指针。

(1.6) Delegate方法命名规范

delegate methods 又叫做delegation methods，如果delegate对象实现了另一个对象的delegate方法，那么这个对象就可以在它自己某个指定的事件发生时调用delegate对象的delegate方法。delegate方法的命名有一些与众不同的格式：

【建议】以触发消息的对象名开头，省略类名前缀并且首字母小写：

```
1. - (BOOL)tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
2. - (BOOL)application:(NSApplication *)sender openFile:(NSString *)filena
```

【建议】除非delegate方法只有一个参数，即触发delegate方法调用的delegating对象，否则冒号是紧跟在类名后面的。

```
1. - (BOOL)applicationOpenUntitledFile:(NSApplication *)sender;
```

【建议】发送通知后再触发delegate方法是一个例外：当delegate方法的调用是为了告诉delegate对象，某个通知已经被发送时，这个delegate方法的参数应该是通知对象，而非触发delegate方法的对象。

```
1. - (void)windowDidChangeScreen:(NSNotification *)notification;
```

【建议】使用did或will这两个情态动词通知delegate对象某件事已经发生或将要发生。

```
1. - (void)browserDidScroll:(NSBrowser *)sender;
2. - (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window;
```

【建议】虽然我们可以在delegate方法中使用did和will来询问delegate是否可以代替另一个对象做某件事情，但是使用should看起来更加完美。

```
1. - (BOOL)windowShouldClose:(id)sender;
```

(1.7) Private方法命名规范

大部分情况下，私有方法的命名和公有方法的命名规则是一样的。然而，通常情况下应该给私有方法添加一个前缀，目的是和公有方法区分开。尽管这样，这种给私有方法加前缀的命名方式有可能引起一些奇怪的问题。问题就是：当你从Cocoa framework(即Cocoa系统库)中的某个类派生出来一个子类时，你并不知道你的子类中定义的私有方法是否覆盖了父类的私有方法，即有可能你自己在子类中实现的私有方法和父类中的某个私有方法同名。在运行时，这极有可能导致一些莫名其妙的问题，并且调试追踪问题的难度也是相当大。

Cocoa frameworks(Cocoa系统库)中的私有方法通常以一个下划线“_”开头，用于标记这些方法是私有的(比如，_fooData）。不要问我为什么他们这么做，这大概就是Apple工程师的开发习惯。基于这个事实，提供以下两条建议：

【必须】禁止使用下划线“_“作为私有方法的开头。Apple已经预留这种私有方法的命名习惯。

【建议】如果你是要子类化Cocoa Frameworks中的一个非常庞大复杂的类(比如NSView或UIView)，并且你想绝对的确保你自己的子类中的私有方法名和父类中的私有方法名不重复。你可以添加一个你自己的前缀作为私有方法的前缀，这个前缀应该尽可能的独特。也许这个前缀是基于你公司或者项目的缩写，比如”XX_“。

尽管给私有方法增加前缀看起来和”方法存在于他们的类的命名空间中“这一之前的说法有些冲突，但此处的意图是：为子类私有方法添加前缀仅仅是为了保证子类方法和父类方法名称不冲突。

【必须】不要在参数的名称中使用“pointer”或者"ptr"。应该使用参数的类型来说明参数是否是一个指针。

【必须】不要使用一到两个字符作为参数名。

【必须】不要对参数的每个单词都缩写。

【建议】如果调用某个方法是为了通知delegate某个事件"即将"发生或者"已经"发生，则请在方法名称中使用“will”或者“did”这样的助动词。例如：

```
1. - (void)applicationWillResignActive:(UIApplication *)application;
2. - (void)applicationDidEnterBackground:(UIApplication *)application;
```

【建议】如果调用某个方法是为了要求delegate代表其他对象执行某件事情，我们应该在方法中使用“should”这样的情态动词。当然，也可以在方法中使用“did”或者“will”这样的字眼，但更倾向于前者。

```
1. - (BOOL)tableViewSholdScroll:(id)sender;
```

(1.8) Category命名规范

【必须】category中不要声明属性和成员变量。

【必须】避免category中的方法覆盖系统方法。可以使用前缀来区分系统方法和category方法。但前缀不要仅仅使用下划线”_“。

【建议】如果一个类比较复杂，建议使用category的方式组织代码。具体可以参考UIView。

1.9 Class命名规范

【必须】class的名称应该由两部分组成，前缀+名称。即，class的名称应该包含一个前缀和一个名词。

(1.10) Protocol命名规范

命名	说明
NSLocking	OK
NSLock	不好，看起来像是一个类名

【建议】有时候protocol只是声明了一堆相关方法，并不关联class。这种不关联class的protocol使用ing形式以和class区分开来。比如NSLocking而非NSLock。

命名	说明
NSLocking	OK
NSLock	不好，看起来像是一个类名

命名	说明
UITableViewDelegate	OK
NSObjectProtocol	OK

【建议】如果proctocol不仅声明了一堆相关方法，还关联了某个class。这种关联class的protocol的命名取决于关联的class，然后再后面再加上protocol或delegate用于显示的声明这是一份协议。

命名	说明
UITableViewDelegate	OK
NSObjectProtocol	OK

(1.11) Notification命名规范

【建议】苹果爸爸说：如果一个类声明了delegate属性，通常情况下，这个

类的delegate对象可以通过实现的delegate方法收到大部分通知消息。那么，这些通知的名称应该反映出对应的delegate方法。比如，application对象发送的NSApplicationDidBecomeActiveNotification通知和对应的applicationDidBecomeActive:消息。其实，这也算是命名的一致性要求。

【必须】notification的命名使用全局的NSString字符串进行标识。命名方式如下：

```
1. [Name of associated class] + [Did | Will] + [UniquePartOfName] + Notif
```

例如：

```
1. NSApplicationDidBecomeActiveNotification
2. NSWindowDidMiniaturizeNotification
3. NSTextViewDidChangeSelectionNotification
4. NSColorPanelColorDidChangeNotification
```

【必须】object通常是指发出notification的对象，如果在发送notification的同时要传递一些额外的信息，请使用userInfo，而不是object。

【必须】如果某个通知是为了告知外界某个事件"即将"发生或者"已经"发生，则请在通知名称中使用“will”或者“did”这样的助动词。例如：

```
1. UIKeyboardWillChangeFrameNotification;
2. UIKeyboardDidChangeFrameNotification;
```

(1.12) Constant命名规范

(1.12.1) 枚举常量

【必须】使用枚举类型来表示一组相关的整型常量。

【建议】枚举常量和typedef定义的枚举类型的命名规范同函数的命名规范一致。

```
1. typedef enum _NSMatrixMode {
2.     NSRadioModeMatrix           = 0,
3.     NSHighlightModeMatrix       = 1,
```

```
4.      NSListModeMatrix            = 2,
5.      NSTrackModeMatrix           = 3
6.  } NSMatrixMode;
```

注意：上面枚举typeof中的_NSMatrixMode是无用的。

我们可以像位掩码(bit masks)一样创建一个匿名枚举，如下：

```
1.  enum {
2.      NSBorderlessWindowMask      = 0,
3.      NSTitledWindowMask          = 1 << 0,
4.      NSClosableWindowMask        = 1 << 1,
5.      NSMiniaturizableWindowMask  = 1 << 2,
6.      NSResizableWindowMask       = 1 << 3
7.  };
```

(1.12.2) 使用const关键字创建常量

【必须】使用const关键字创建浮点型常量。你也可以使用const来创建和其他常量不相关的整型常量。否则，请使用枚举类型来创建。即，如果一个整型常量和其他常量不相关，可以使用const来创建，否则，使用枚举类型表示一组相关的整型常量。

以下例子声明了const常量的格式：

```
1.  const float NSLightGray;
```

1.12.3 其他常量类型

【必须】通常情况下，不要使用#define预处理命令(preprocessor command)创建常量。正如上面所说，对于整型常量，使用枚举创建;对于浮点型常量，使用const修饰符创建。

【必须】有些符号需要使用大写字母标识。预处理器需要根据这个符号进行计算以便决定是否要对某一块代码进行处理。比如：

```
1.  #ifdef DEBUG
```

注意：那些编译器定义的宏，左侧和右侧各有两个下划线。如下：

```
1. __MACH__
```

【必须】通知的名字和字典的key，应该使用字符串常量来定义。使用字符串常量编译器可以进行检查，这样可以避免拼写错误。Cocoa 系统库提供了许多字符串常量的例子，比如：

```
1. APPKIT_EXTERN NSString *NSPrintCopies;
```

字符串常量应该在.h头文件中暴露给外部，而字符串常量真正的赋值是在.m文件中。如下：

```
1. .h文件
2. extern NSString *const WSNetworkReachablityStatusDidChangedNotificatio
3. .m文件
4. NSString * const WSNetworkReachablityStatusDidChangedNotification = @"
```

(1.13) Exception命名规范

上面已经有一节介绍过通知的命名规范。异常和通知的命名遵循相似的规则，但又各有不同。

【必须】和Notification的命名规范一样(可参考Notification命名规范一节)，异常也是用全局的NSString字符串进行标识。命名方式如下：

[Prefix] + [UniquePartOfName] + Exception

相当于异常由前缀、名称中能够标识异常唯一性的那部分、Exception。如下：

```
1. NSColorListIOException
2. NSColorListNotEditableException
3. NSDraggingException
4. NSFontUnavailableException
5. NSIllegalSelectorException
```

(二)编码规范

(2.1) Initialize规范

(void)initialize类方法先于其他的方法调用。且initialize方法给我们提供了一个让代码once、lazy执行的地方。initialize通常被用于设置class的版本号(参考 Versioning and Compatibility)。
(https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/FrameworkImpl.html#//apple_ref/doc/uid/20001286-1001777)

initialize方法的调用遵循继承规则(所谓继承规则，简单来讲是指：子类方法中可以调用到父类的同名方法，即使没有调用[super xxx])。如果我们没有实现initialize方法，运行时初次调用这个类的时候，系统会沿着继承链(类继承体系)，先后给继承链上游中的每个超类发送一条initialize消息，直到某个超类实现了initlialize方法，才会停止向上调用。因此，在运行时，某个类的initialize方法可能会被调用多次(比如，如果一个子类没有实现initialize方法)。

比如：有三个类：SuperClass、SubClass和FinalClass。他们的继承关系是这样的FinalClass->SubClass->SuperClass，现只实现了SuperClass方法的initialize方法。

```
1.  // SuperClass
2.  @implementation SuperClass
3.  + (void)initialize {
4.      NSLog(@"superClass initalize");
5.  }
6.  @end
7.  // 初始化FinalClass
8.  - (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)e
9.      FinalClass *finalC = [FinalClass new];
10. }
11. // 控制台输出结果
12. 2018-01-27 22:11:03.130365+0800 Demo[67162:11721965] superClass inital
13. 2018-01-27 22:11:03.130722+0800 Demo[67162:11721965] superClass inital
14. 2018-01-27 22:11:03.130815+0800 Demo[67162:11721965] superClass inital
```

解释：

因为FinalClass继承自SubClass，SubClass继承自SuperClass。因为继承体系中只有SuperClass实现了initialize方法，导致初始化FinalClass这个子类时，FinalClass会调用他的父类(SubClass)中的initialize方法。又因为他(FinalClass)的父类(SubClass)也没有实现initialize方法，又会继续沿着继承体系，向上游寻找，最后找到SubClass的父类(SuperClass)。因为SuperClass实现了这个initialize方法，所以调用结束。至于为什么是连续调用了三次SuperClass的initialize方法。因为子类FinalClass的初始化触发了超类SubClass、SuperClass的初始化。所以初始化FinalClass时，实际上使这三个类都得到了初始化的机会，自然就会连续调用三次SuperClass的initialize方法。

还是上面那三个类，如果我们又给SubClass实现了initialize方法，那么控制台将会输出如下结果(至于为什么，前面已经介绍过了，大家可以自己分析下)：

```
1. 2018-01-27 22:34:54.697952+0800 Load[67652:11780578] superClass inital
2. 2018-01-27 22:34:54.698118+0800 Load[67652:11780578] subClass initiali
3. 2018-01-27 22:34:54.698472+0800 Load[67652:11780578] subClass initiali
```

基于上面陈述的这些事实，我们得出一个结论：

【必须】如果我们想要让initialize方法仅仅被调用一次，那么需要借助于GCD的dispatch_once()。如下：

```
1. + (void)initialize {
2.     static dispatch_once_t onceToken = 0;
3.     dispatch_once(&onceToken, ^{
4.         // the initializing code
5.     }
6. }
```

【建议】如果我们想在继承体系的某个指定的类的initialize方法中执行一些初始化代码，可以使用类型检查和而非dispatch_once()。如下：

```
1. if (self == [NSFoo class]) {
2.     // the initializing code
3. }
```

说了这么多，总而言之，由于任何子类都会调用父类的initialize方法，所以可能会导致某个父类的initialize方法会被调用多次，为了避免这种情况，我们可以使用类型判等或dispatch_once()这两种方式，以保证initialize中的代码不会被无辜调用。

initialize是由系统自动调用的方法，我们不应该显示或手动调用initialize方法。如果我们要触发某个类的初始化行为，应该调用这个类的一些无害的方法。比如：

```
1. [NSImage self];
```

(2.2)Init方法规范

Objective-C有designated Initializers和secondary Initializers的概念。designated Initializers叫做指定初始化方法。《Effective Objective-C 2.0 编写高质量iOS与 OS X代码的52个有效方法》中将designated Initializers翻译为”全能初始化方法“。designated Initializers方法是指类中为对象提供必要信息以便其能完成工作的初始化方法。一个类可以有一个或者多个designated Initializers。但是要保证所有的其他secondary initializers都要调用designated Initializers。即：只有designated Initializers才会存储对象的信息。这样的好处是：当这个类底层的某些数据存储机制发生变化时(可能是一些property的变更)，只需要修改这个designated Initializers内部的代码即可。无需改动其他secondary Initializers初始化方法的代码。

【必须】所有secondary 初始化方法都应该调用designated 初始化方法。

【必须】所有子类的designated初始化方法都要调用父类的designated初始化方法。使这种调用关系沿着类的继承体系形成一条链。

【必须】如果子类的designated初始化方法与超类的designated初始化方法不同，则子类应该覆写超类的designated初始化方法。(因为开发者很有可能直接调用超类的某个designated方法来初始化一个子类对象，这样也是合情合理的，但使用超类的方法初始化子类，可能会导致子类在初始化时缺失一些必要信息)。

【必须】如果超类的某个初始化方法不适用于子类，则子类应该覆写这个超

类的方法，并在其中抛出异常。

【必须】禁止子类的designated初始化方法调用父类的secondary初始化方法。否则容易陷入方法调用死循环。如下：

```
1.  // 超类
2.  @interface ParentObject : NSObject
3.  @end
4.   @implementation ParentObject
5.      //designated initializer
6.      - (instancetype)initWithURL:(NSString*)url title:(NSString*)title
7.          if (self = [super init]) {
8.              _url = [url copy];
9.              _title = [title copy];
10.         }
11.         return self;
12.     }
13.     //secondary initializer
14.     - (instancetype)initWithURL:(NSString*)url {
15.         return [self initWithURL:url title:nil];
16.     }
17.     @end
18. // 子类
19. @interface ChildObject : ParentObject
20. @end
21.     @implementation ChildObject
22.     //designated initializer
23.     - (instancetype)initWithURL:(NSString*)url title:(NSString*)title
24.         //在designated intializer中调用 secondary initializer, 错误的
25.         if (self = [super initWithURL:url]) {
26.         }
27.         return self;
28.     }
29.     @end
30.  @implementation ViewController
31.     - (void)viewDidLoad {
32.         [super viewDidLoad];
33.         // 这里会死循环
34.         ChildObject* child = [[ChildObject alloc] initWithURL:@"url" t
35.     }
36.     @end
```

【必须】另外禁止在init方法中使用self.xxx的方式访问属性。如果存在继承的情况下，很有可能导致崩溃。

(2.3) Init error

一个好的初始化方法应该具备以下几个方面，在初始化阶段就能够发现错误并给予处理，也就是初始化方法应该具备一些必要的容错功能。

【必须】调用父类的designated初始化方法初始化本类的对象。

【必须】校验父类designated初始化方法返回的对象是否为nil。

【建议】如果初始化当前对象的时候发生了错误，应该给予对应的处理：释放对象，并返回nil。

以下实例列举类初始化阶段可能会存在的错误：

```
1. - (id)init {
2.     self = [super init];  // Call a designated initializer here.
3.     if (self != nil) {
4.         // Initialize object  ...
5.         if (someError) {
6.             [self release];
7.             self = nil;
8.         }
9.     }
10.    return self;
11. }
```

(2.4) dealloc规范

【必须】不要忘记在dealloc方法中移除通知和KVO。

【建议】dealloc 方法应该放在实现文件的最上面，并且刚好在 @synthesize 和 @dynamic 语句的后面。在任何类中，init 都应该直接放在 dealloc 方法的下面。

【必须】在dealloc方法中，禁止将self作为参数传递出去，如果self被retain住，到下个runloop周期再释放，则会造成多次释放crash。如下：

```
1. -(void)dealloc{
2.     [self unsafeMethod:self];
3.     // 因为当前已经在self这个指针所指向的对象的销毁阶段，销毁self所指向的对象已经
4.     // 到了下一个runloop周期，因为self所指向的对象已经被销毁，会因为非法访问而造
```

```
5. }
```

【必须】和init方法一样，禁止在dealloc方法中使用self.xxx的方式访问属性。如果存在继承的情况下，很有可能导致崩溃。

(2.5) Block规范

【必须】调用block时需要对block判空。

【必须】注意block潜在的引用循环。

(2.6) Notification规范

前面在命名规范一章中已经介绍了通知的命名规范，这里解释的是通知的使用规范。

通知作为观察者模式的一个落地产物，在开发中能够实现一对多的通信。所有可以使用delegate和block实现的通信和传值，都可以使用通知实现。正因通知如此灵活，我们更应该弄清楚通知适合使用的场景，避免把通知和delegate以及block等进行混淆。

通知是一把双刃剑，让你欢喜让你忧。开发中，当你走投无路将要崩溃时，可以考虑使用通知;而当你频繁使用通知时，同样会让你崩溃到走投无路。所以，在每个应用中，我们应该时刻留意并控制通知的数量，避免通知满天飞的现象。

曾经有一个项目摆在我面前，我却无法珍惜，因为通知太多了，几乎有代码的地方就有通知。如果现在同样有一个充满通知的项目摆在我面前，我知道是时候该优化它了。

【必须】基于以上的陈述，当我们使用通知时，必须要思考，有没有更好的办法来代替这个通知。禁止遇到问题就想到通知，把通知作为备选项而非首选项。

【必须】post通知时，object通常是指发出notification的对象，如果在发送notification的同时要传递一些额外的信息，请使用userInfo，而不是object。

【必须】NSNotificationCenter在iOS8及更老的系统有一个多线程bug，

selector执行到一半可能会因为self的销毁而引起crash，解决的方案是在selector中使用weak_strong_dance。如下：

```
1. - (void)onMultiThreadNotificationTrigged:(NSNotification *)notify {
2.       __weak typeof(self) wself = self; __strong typeof(self) sself = ws
3.       if (!sself) { return; }
4.       [self doSomething];
5. }
```

【必须】在多线程应用中，Notification在哪个线程中post，就在哪个线程中被转发，而不一定是在注册观察者的那个线程中。如果post消息不在主线程，而接受消息的回调里做了UI操作，需要让其在主线程执行。

说明：每个进程都会创建一个NotificationCenter，这个center通过NSNotificationCenter defaultCenter获取，当然也可以自己创建一个center。

NoticiationCenter是以同步(非异步，当前线程，会等待，会阻塞)的方式发送请求。即，当post通知时，center会一直等待所有的observer都收到并且处理了通知才会返回到poster。如果需要异步发送通知，请使用notificationQueue，在一个多线程的应用中，通知会发送到所有的线程中。

(2.7) UI规范

【必须】如果想要获取window，不要使用view.window获取。请使用[[UIApplication sharedApplication] keyWindow]。

【必须】在使用到 UIScrollView，UITableView，UICollectionView 的 Class 中，需要在 dealloc 方法里手动的把对应的 delegate, dataSouce 置为 nil。

【必须】UITableView使用self-sizing实现不等高cell时，请在-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath;中给cell设置数据。不要在-(void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell forRowAtIndexPath:(NSIndexPath *)indexPath;方法中给cell设置数据。

【建议】当访问一个 CGRect 的 x， y， width， height 时，应该使用 CGGeometry 函数代替直接访问结构体成员。苹果的 CGGeometry 参考中说到：

- All functions described in this reference that take CGRect data structures as inputs implicitly standardize those rectangles before calculating their results. For this reason, your applications should avoid directly reading and writing the data stored in the CGRect data structure. Instead, use the functions described here to manipulate rectangles and to retrieve their characteristics.

因此，推荐的写法是这样的：

```
1. CGRect frame = self.view.frame;
2.
3. CGFloat x = CGRectGetMinX(frame);
4. CGFloat y = CGRectGetMinY(frame);
5. CGFloat width = CGRectGetWidth(frame);
6. CGFloat height = CGRectGetHeight(frame);
```

反对这样的写法：

```
1. CGRect frame = self.view.frame;
2.
3. CGFloat x = frame.origin.x;
4. CGFloat y = frame.origin.y;
5. CGFloat width = frame.size.width;
6. CGFloat height = frame.size.height;
```

(2.8) IO规范

【建议】尽量少用NSUserDefaults。

说明：[[NSUserDefaults standardUserDefaults] synchronize] 会block住当前线程，知道所有的内容都写进磁盘，如果内容过多，重复调用的话会严重影响性能。

【建议】一些经常被使用的文件建议做好缓存。避免重复的IO操作。建议只有在合适的时候再进行持久化操作。

2.9 Collection规范

【必须】不要用一个可能为nil的对象初始化集合对象，否则可能会导致

crash。

```
1. // 可能崩溃
2. NSObject *obj = somOjbcetMaybeNil;
3. NSMutableArray *arrM = [NSMutableArray arrayWithObject:obj];
4. // 崩溃信息：
5. *** Terminating app due to uncaught exception 'NSInvalidArgumentExcept
```

```
1. // 改进办法：
2. NSObject *obj = somOjbcetMaybeNil;
3. NSMutableArray *arrM = nil;
4. if (obj && [obj isKindOfClass:[NSObject class]]) {
5.     arrM = [NSMutableArray arrayWithObject:obj];
6. } else {
7.     arrM = nil;
8. }
```

【必须】同理，对插入到集合对象里面的对象也要进行判空。

【必须】注意在多线程环境下访问可变集合对象的问题，必要时应该加锁保护。不可变集合(比如NSArray)类默认是线程安全的，而可变集合类(比如NSMutableArray)不是线程安全的。

【必须】禁止在多线程环境下直接访问可变集合对象中的元素。应该先对其进行copy，然后访问不可变集合对象内的元素。

```
1. // 正确写法
2. - (void)checkAllValidItems{
3. NSArray *array = [array copy];
4. [array enumerateObjectsUsingBlock:^(id _Nonnull obj, NSUInteger idx, B
5. //do something using obj
6. }]; }
7.
8. // 错误写法
9. - (void)checkAllValidItems{
10. [self.allItems enumerateObjectsUsingBlock:^(id _Nonnull obj, NSUIntege
11.     //do something using obj
12.     // 如果在enumerate过程中，其他线程对allItems这个可变集合进行了变更操作，这
13. }]; }
```

【必须】注意使用enumerateObjectsUsingBlock遍历集合对象中的对象时，关键字return的作用域。block中的return代表的是使当前的block返回，而非使

当前的整个函数体返回。以下使用NSArray举例，其他集合类型同理。如下：

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)e
    NSArray *array = [NSArray arrayWithObject:@"1"];
    [array enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger id
        // excute some code...
        return;
    }];
    // 依然会执行到这里
    NSLog(@"fall through");
}

// 执行结果：
// fall through
```

当然，两个enumerateObjectsUsingBlock嵌套，如果仅在最内层的block中return，外层block的代码还是会被执行。如下：

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)e
    NSArray *arr1 = [NSArray arrayWithObject:@"1"];
    NSArray *arr2 = [NSArray arrayWithObject:@"2"];
    [arr2 enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger idx
        [arr1 enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUIntege
            // do something
            return;
        }];
        NSLog(@"fall through");
    }];
    NSLog(@"fall through");
}

// 执行结果：
// fall through
// fall through
```

说明：其实block相当于一个匿名函数，在block中使用return返回，仅是让当前这个匿名函数返回。

【必须】禁止返回mutable对象，禁止mutable对象作为入参传递。

【建议】如果使用NSMutableDictionary作为缓存，建议使用NSCache代替。

【建议】集合类使用泛型来指定对象的类型。

```
1.  @property(nonatomic,copy) NSArray<NSString *> *array;
2.  @property(nonatomic,strong) NSMutableDictionary<NSString *,NSString *>
```

(2.10) 分支语句规范

【建议】if条件判断语句后面必须要加大括号{}。不然随着业务的发展和代码迭代，极有可能引起逻辑问题。

```
1.  // 建议
2.  if (!error) {
3.      return success;
4.  }
5.  // 不建议
6.  if (!error)
7.      return success;
8.  if (!error)  return success;
```

【必须】多于3个逻辑表达式必须用参数分割成多个有意义的bool变量。

【建议】遵循gold path法则，不要把真正的逻辑写道括号内。

```
 1.  // 不建议
 2.  - (void)someFuncWith:(NSString *)parameter {
 3.      if (parameter) {
 4.          // do something
 5.          [self doSomething];
 6.      }
 7.  }
 8.  // 建议
 9.  - (void)someFuncWith:(NSString *)parameter {
10.      if (!parameter) {
11.          return;
12.      }
13.      // do something
14.      [self doSomething];
15.  }
```

【建议】对于条件语句的真假，因为 nil 解析为 NO，所以没有必要在条件中与它进行比较。永远不要直接和 YES 和 NO进行比较，因为 YES 被定义

为 1，而 BOOL 可以多达 8 位。

```
1. // 建议
2. if (isAwesome)
3. if (![someObject boolValue])
4. // 禁止这样做
5. if ([someObject boolValue] == NO) { }
6. if (isAwesome == YES) { }
```

【必须】使用switch...case...语句的时候，不要丢掉default:。除非switch枚举。

【必须】switch...case...语句的每个case都要添加break关键字，避免出现fall-through。

(2.11) 对象判等规范

isEqual:方法允许我们传入任意类型的对象作为参数，如果参数类型和receiver(方法调用者)类型不一致，会返回NO。而isEqualToString:和isEqualToArray:这两个方法会假设参数类型和receiver类型一致，也就是说，这两个方法不会对参数进行类型检查。因此这两个方法性能更好但不安全。如果我们是从外部数据源(比如info.plist或preferences)获取的数据，那么推荐使用isEqual:，因为这样更安全。如果我们知道参数的确切类型，那么可以使用类似于isEqualToString:这样的方法，因为性能更好。

(2.12) 懒加载规范

懒加载适合的场景：

1. 一个对象的创建依赖于其他对象。
2. 一个对象在整个app过程中，可能被使用，也可能不被使用。
3. 一个对象的创建需要经过大量的计算或者比较消耗性能。除以上三条之外，请不要使用懒加载。

【建议】懒加载本质上就是延迟初始化某个对象，所以，懒加载仅仅是初始化一个对象，然后对这个对象的属性赋值。懒加载中不应该有其他的不必要的逻辑性的代码，如果有，请把那些逻辑性代码放到合适的地方。

【必须】不要滥用懒加载，只对那些真正需要懒加载的对象采用懒加载。

【必须】如果一个对象在懒加载后，某些场景下又被设置为nil。我们很难保证这个懒加载不被再次触发。

(2.13) 多线程规范

【必须】禁止使用GCD的dispatch_get_current_queue()函数获取当前线程信息。

【必须】对剪贴板的读取必须要放在异步线程处理，最新Mac和iOS里的剪贴板共享功能会导致有可能需要读取大量的内容，导致读取线程被长时间阻塞。

【建议】仅当必须保证顺序执行时才使用dispatch_sync，否则容易出现死锁，应避免使用，可使用dispatch_async。

```
1.  - (void)viewDidLoad {
2.      [super viewDidLoad];
3.      // 错误。出现死锁，报错:EXC_BAD_INSTRUCTION。原因:在主队列中同步的添加一个b
4.      dispatch_queue_t mainQueue = dispatch_get_main_queue();
5.      dispatch_block_t block = ^() {
6.          NSLog(@"%@", [NSThread currentThread]);
7.      };
8.      dispatch_sync(mainQueue, block);
9.  }
```

```
1.  - (void)viewDidLoad {
2.      [super viewDidLoad];
3.      // 正确。异步执行。虽然还是把任务加到了主队列由主线程来执行，但因为是异步，此时
4.      dispatch_queue_t mainQueue = dispatch_get_main_queue();
5.      dispatch_block_t block = ^() {
6.          NSLog(@"%@", [NSThread currentThread]);
7.      };
8.      dispatch_async(mainQueue, block);
9.  }
10. // 打印结果:
11. // <NSThread: 0x600000073300>{number = 1, name = main}
```

【必须】禁止在非主线程中进行UI元素的操作。

【必须】在主线程中禁止进行同步网络资源读取，使用NSURLSession进行异

步获取。当然，你可以在子线程同步获取网络资源，但还是上面的那一条建议：避免使用dispatch_sync，尽量使用dispatch_async。因为死锁不一定只发生在主线程。

【必须】如果需要进行大文件或者多文件的IO操作，禁止主线程使用，必须进行异步处理。

【必须】对剪贴板的读取必须要放在异步线程处理，最新Mac和iOS里的剪贴板共享功能会导致有可能需要读取大量的内容，导致读取线程被长时间阻塞。

```
1.  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAU
2.    UIPasteboard *pasteboard = [UIPasteboard generalPasteboard];
3.    if (pasteboard.string.length > 0) {//这个方法会阻塞线程
4.       NSString *text = [pasteboard.string copy];
5.       [pasteboard setValue:@"" forPasteboardType:UIPasteboardNameGenera
6.       if (text == nil || [text isEqualToString:@""]) {
7.           return ;
8.       }
9.       dispatch_async(dispatch_get_main_queue(), ^{
10.          [self processShareCode:text];
11.      });
12.   }
13. });
```

2.14 内存管理规范

【建议】函数体提前return时，要注意是否有对象没有被释放掉(常见于CF对象)，避免造成内存泄露。

【建议】请慎重使用单例，避免产生不必要的常驻内存。

说明：我们不仅应该知道单例的特点和优势，也必须要弄明白单例适合的场景。UIApplication、access database 、request network 、access userInfo这类全局仅存在一份的对象或者需要多线程访问的对象，可以使用单例。不要仅仅为了访问方便就使用单例。

【建议】单例初始化方法中尽量保证单一职责,尤其不要进行其他单例的调用。极端情况下，两个单例对象在各自的单例初始化方法中调用，会造成死锁。

【必须】在dealloc方法中，禁止将self作为参数传递出去，如果self被retain住，到下个runloop周期再释放，则会造成多次释放crash。这一点在dealloc一节中有说明。

【建议】除非你清除的知道自己在做什么。否则不建议将UIView类的对象加入到NSArray、NSDictionary、NSSet中。如有需要可以添加到NSMapTable和 NSHashTable。因为NSArray、NSDictionary、NSSet会对加入的对象做strong引用(即使你把加入的对象进行了weak)。而NSMapTable、NSHashTable会对加入的对象做weak引用。

说明：简单的说，NSHashTable相当于weak的NSMutableArray;NSMapTable相当于weak的NSMutableDictionary.

```
// 错误的例子：
@implementation WSObject
- (void)dealloc {
    NSLog(@"dealloc");
}
@end
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)e
    WSObject *object = [WSObject new];
    // 即使对object进行了weak弱化，数组也会强引用这个object对象。dealloc方法不
    __weak typeof(object) weakObject = object;
    [self.arrM addObject:weakObject];
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2 * NSEC
        NSLog(@"count = %ld",self.arrM.count);
    });
}
// 打印结果：
// count = 1
```

```
// 正确的例子：
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)e
    WSObject *object = [WSObject new];
    NSHashTable *hashTable = [NSHashTable weakObjectsHashTable];
    [hashTable addObject:object];
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2 * NSEC
        NSLog(@"count = %ld",hashTable.count);
    });
}
// 打印结果：
// dealloc
// count = 1
```

你可能对上面的例子有所疑惑，object已经释放了，但是控制台仍然输出hashTable.count == 1。但是请相信我，此时存在于hashTable中的那个object已经变成了nil。不信你继续看下面的例子：

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)e
    WSObject *object = [WSObject new];
    NSHashTable *hashTable = [NSHashTable weakObjectsHashTable];
    [hashTable addObject:object];
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2 * NSEC_
        NSLog(@"count = %ld",hashTable.count);
        if (hashTable && hashTable.count) {
            WSObject *object = [hashTable anyObject];
            NSLog(@"object = %@",[object self]);
        }
    });
}
// 打印结果：
2017-07-04 22:19:10.952139+0800 tst[46834:4305636] dealloc
2017-07-04 22:19:13.149903+0800 tst[46834:4305636] count = 1
2017-07-04 22:20:55.234522+0800 tst[46834:4305636] object = (null)
```

(2.15) 延迟调用规范

【必须】performSelector:withObject:afterDelay:要在有Runloop的线程里调用，否则调用无法生效。

- **说明：**异步线程默认是没有runloop的，除非手动创建;而主线程是系统会自动创建Runloop的。所以在异步线程调用是请先确保该线程是有Runloop的。

使用performSelector:withObject:afterDelay:和cancelPreviousPerformRequestsWithTarget组合的时候要小心：

- afterDelay会增加引用计数，而cancel会对引用计数减一

如果receiver在引用计数器为1的时候，调用cancel会立即回收receiver。后续再次调用receiver的方法就会crash。所以我们需要使用weakSelf并判空。如下：

```
__weak typeof(self) weakSelf = self;
[NSObject cancelPreviousPerformRequestsWithTarget:self];
```

```
3.  if (!weakSelf) {
4.      // NSLog(@"self dealloc");
5.      return;
6.   }
7.  [self doOther];
```

(2.16) 注释规范

【必须】如果方法、函数、类、属性等需要提供给外界或者他人使用，必须要加注释说明。

【必须】如果你的代码以SDK的形式提供给其他人使用，那么接口的注释是必须的。必须对暴露给外界的所有方法、属性、参数加以注释说明。

【建议】注释应该说明其作用以及注意事项(如果有)。

【建议】因为方法或属性本身就具有自我描述性，注释应该简明扼要，说明是什么和为什么即可。

(2.17) 类的设计规范

【建议】尽量减少继承，类的继承关系不要超过3层。可以考虑使用category、protocol来代替继承。

【建议】把一些稳定的、公共的变量或者方法抽取到父类中。子类尽量只维持父类所不具备的特性和功能。

【建议】.h文件中尽量不要声明成员变量。

【建议】.h文件中的属性尽量声明为只读。

【建议】.h文件中只暴露出一些必要的类、公开的方法、只读属性;私有类、私有方法和私有属性以及成员变量，尽量写在.m文件中。

(2.18) 代码组织规范

参考 raywenderlich/objective-c-style-guide(https://github.com/raywenderlich/objective-c-style-guide)

```
1.  #pragma mark - Lifecycle
```

```
 2. - (instancetype)init {}
 3. - (void)dealloc {}
 4. - (void)viewDidLoad {}
 5. - (void)viewWillAppear:(BOOL)animated {}
 6. - (void)didReceiveMemoryWarning {}
 7. #pragma mark - Custom Accessors
 8. - (void)setCustomProperty:(id)value {}
 9. - (id)customProperty {}
10. #pragma mark - IBActions
11. - (IBAction)submitData:(id)sender {}
12. #pragma mark - Public
13. - (void)publicMethod {}
14. #pragma mark - Private
15. - (void)privateMethod {}
16. #pragma mark - UITextFieldDelegate
17. #pragma mark - UITableViewDataSource
18. #pragma mark - UITableViewDelegate
19. #pragma mark - NSCopying
20. - (id)copyWithZone:(NSZone *)zone {}
21. #pragma mark - NSObject
22. - (NSString *)description {}
```

【建议】以上只是提供了组织代码的一种思路，如果有其他更好的组织方式，也不是不可以。

(2.19) 工程结构规范

【必须】为了避免文件杂乱，物理文件应该保持和 Xcode 项目文件同步。Xcode 创建的任何组(group)都必须在文件系统有相应的映射。为了更清晰，代码不仅应该按照类型进行分组，也可以根据业务功能进行分组。

【建议】合理组织工程的内的文件夹，工程中一般包括但不限于以下几个文件夹category(分类)、util/helper(工具类)、resource(资源)、const(常量)、third(第三方)。

【建议】尽可能一直打开 target Build Settings 中 "Treat Warnings as Errors" 以及一些额外的警告。如果你需要忽略指定的警告,使用 Clang 的编译特性。

【编辑推荐】

1. 美团iOS面试败北感悟

2. 几个步骤，让你的iOS代码容易阅读
3. 谷歌推出Android和iOS版Google Tasks
4. 研究表明：员工更喜欢 Mac 和 iOS 而非 Windows 或 Android
5. 关于 iOS 上的 PWA 应用，你需要知道些什么？

【责任编辑：未丽燕 TEL：（010）68476606】

点赞 0