

# Objective-C 代码风格指南

版本	日期	作者
1.0.0	2016-01-23	张艺麟

## 目录

---

- [留白和格式](#)
  - [空格 vs. 制表符\(M\)](#)
  - [换行\(M\)](#)
- [注释](#)
  - [声明部分的注释\(M\)](#)
  - [实现部分的注释\(O\)](#)
- [预编译宏](#)
  - [屏蔽代码\(M\)](#)
  - [条件编译\(M\)](#)
  - [嵌套缩进\(M\)](#)
  - [宏定义\(O\)](#)
- [命名](#)
  - [方法命名\(M\)](#)
  - [变量命名\(M\)](#)
- [属性特性\(M\)](#)
- [语法糖\(M\)](#)
- [Nullability Annotations\(O\)](#)
- [泛型与类型延拓\(O\)](#)
- [类别\(M\)](#)
- [常量\(O\)](#)
- [私有属性\(M\)](#)
- [枚举类型\(M\)](#)
- [布尔值\(O\)](#)
- [条件语句\(M\)](#)

- [三元操作符\(M\)](#)

- [Init方法\(M\)](#)
- [黄金路径\(M\)](#)
- [错误处理\(M\)](#)
- [单例模式\(M\)](#)
- [代码组织\(O\)](#)
- [Xcode工程\(O\)](#)

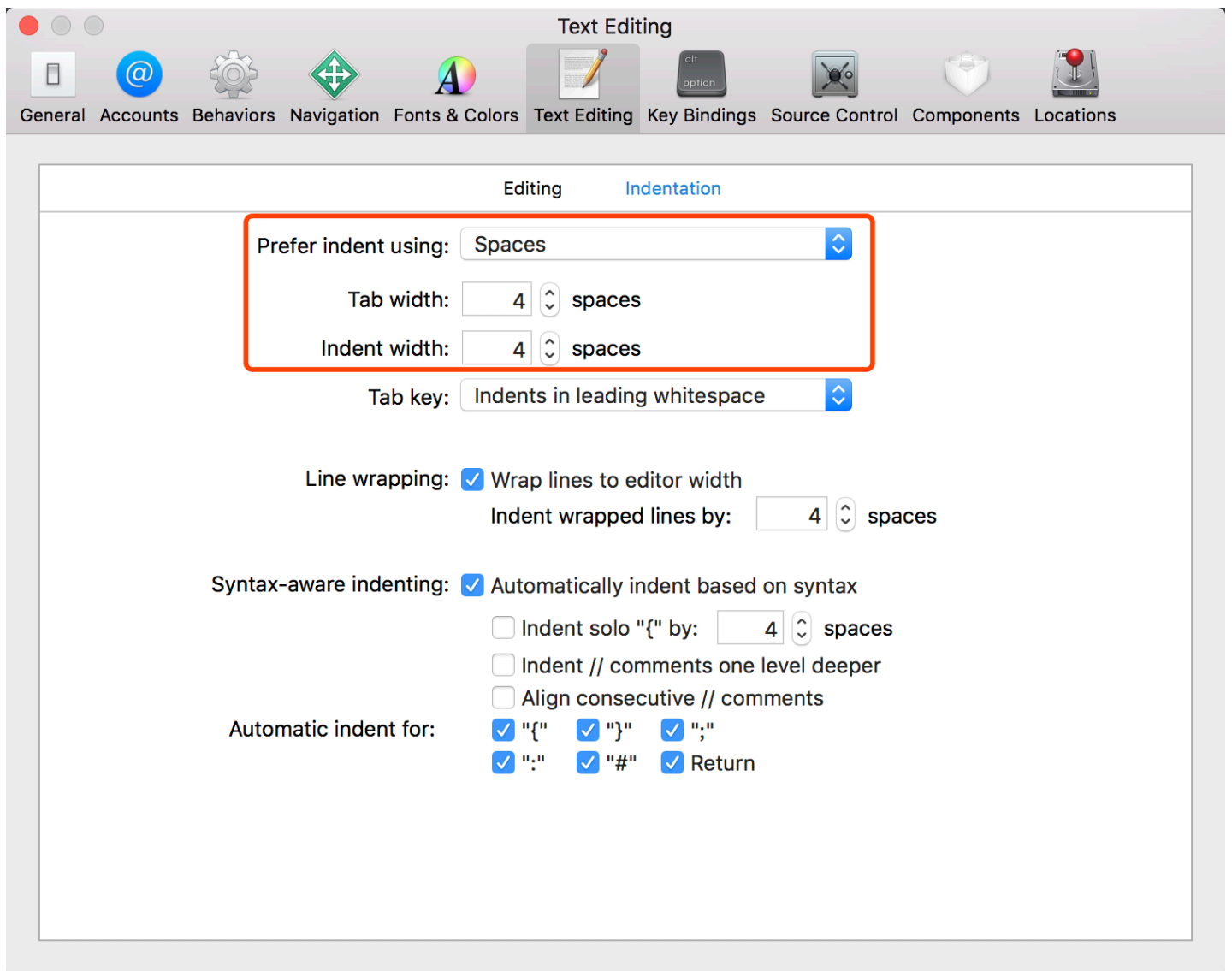
M:建议遵守 O:可选

## 1.留白和格式

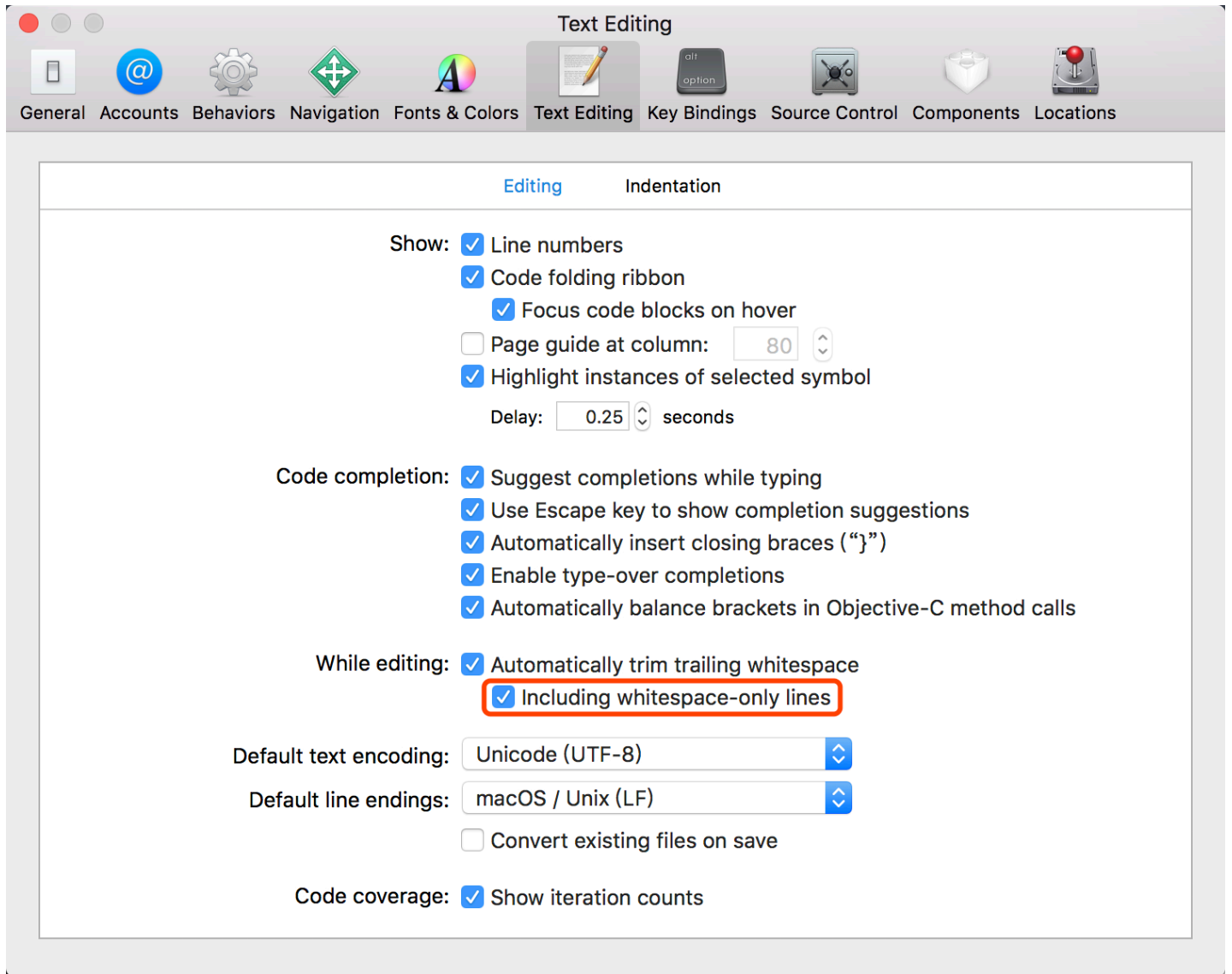
### 1.1空格 vs. 制表符

只使用空格，且一次缩进四个空格。

我们使用空格缩进。不要在代码中使用制表符。你应该将编辑器设置成自动将制表符替换成空格。



删除没有代码空行的空格。



## 1.2换行

- 方法大括号和其他大括号(`if` / `else` / `switch` / `while` 等.)总是在同一行语句打开但在新行中关闭。

应该:

```
1 | if (user.isHappy) {  
2 |     //Do something  
3 | } else {  
4 |     //Do something else  
5 | }
```

不应该:

```
1 | if (user.isHappy)
2 | {
3 |     //Do something
4 | }
5 | else {
6 |     //Do something else
7 | }
```

- 在方法之间应该有且只有一行，这样有利于在视觉上更清晰和更易于组织。在方法内的空白应该分离功能。
- 优先使用auto-synthesis。但如果有必要，@synthesize 和 @dynamic应该在实现中每个都声明新的一行。

## 2.注释

### 2.1声明部分的注释

每个接口、类别以及协议应辅以注释，以描述它的目的及与整个项目的关系。

例子:

```
1 | /**
2 |  <#Description#>
3 |
4 |  @param scid <#scid description#>
5 |  @param videoPath <#videoPath description#>
6 |  @param thumbPath <#thumbPath description#>
7 |  @return <#return value description#>
8 |  */
9 | - (instancetype)initWithScid:(NSString *)scid
10 |    withVideoPath:(NSString *)videoPath
11 |    withThumbPath:(NSString *)thumbPath;
```

快捷键: Command + Shift + /

### 2.2实现部分的注释

当需要注释时，注释应该用来解释这段特殊代码为什么要这样做。任何被使用的注释都必须保持最新或被删除。

一般都避免使用块注释，代码尽可能做到自解释，只有当断断续续或几行代码时才需要注释。

注释 `Why` 而不是 `How`，解释它要完成的任务是什么，而不是只解释一个程序功能的细节

## 3.预编译宏

### 3.1屏蔽代码

我们需要屏蔽多行代码时, 使用条件编译来屏蔽, 并且写在行首.

应该:

```
1 void foo ()
2 {
3     int i = 0;
4     int j = 1;
5     #if 0
6         if (i == 0) {
7             NSLog(@"i is zero.");
8         } else {
9             NSLog(@"i is not zero.");
10        }
11    #endif
12 }
```

不应该:

```
1 void foo ()
2 {
3     int i = 0;
4     int j = 1;
5
6     //if (i == 0) {
7         //NSLog(@"i is zero.");
8     //} else {
9         //NSLog(@"i is not zero.");
10    //}
11 }
```

### 3.2条件编译

条件编译的代码如果超过一屏幕, 约 20 行代码, 需要在条件编译的 `else`, `end` 等分支部分添加注释代码.

例如:

```
1  #ifdef CONDITION_FOO
2  void foo1 ()
3  {
4      int i = 0;
5      int j = 1;
6
7      if (i == 0) {
8          NSLog(@"i is zero.");
9      } else {
10         NSLog(@"i is not zero.");
11     }
12 }
13
14 void foo2 ()
15 {
16     int i = 0;
17     int j = 1;
18
19     if (i == 0) {
20         NSLog(@"i is zero.");
21     } else {
22         NSLog(@"i is not zero.");
23     }
24 }
25
26 #else /* !CONDITION_FOO */
27
28 void foo1 ()
29 {
30     int i = 0;
31     int j = 1;
32
33     if (i == 0) {
34         NSLog(@"i is zero.");
35     } else {
36         NSLog(@"i is not zero.");
37     }
38 }
39
40 void foo2 ()
41 {
42     int i = 0;
43     int j = 1;
44
45     if (i == 0) {
46         NSLog(@"i is zero.");
47     } else {
48         NSLog(@"i is not zero.");
```

```
49     }
50 }
51
52 #endif /* CONDITION_FOO */
```

### 3.3 嵌套缩进

如果条件编译宏需要缩进, 则在 `#` 字符后添加2个空格表示一层缩进. 如果嵌套层级 大于 3 层, 则需要 在条件编译结束或者分支后面添加条件注释.

例如:

```
1  #ifdef USE_FOO
2  void foo1 ()
3  {
4      int i = 0;
5      int j = 1;
6      # ifdef CONDITION_IF
7          if (i == 0) {
8              #   ifdef CONDITION_USE_LOG
9                  NSLog(@"i is zero.");
10             #   endif /* CONDITION_USE_LOG */
11             # else /* !CONDITION_IF */
12                 } else {
13                     #   ifdef CONDITION_USE_LOG
14                         NSLog(@"i is not zero.");
15                     #   endif /* CONDITION_USE_LOG */
16                 }
17             # endif /* CONDITION_IF */
18         }
19     #endif /* USE_FOO */
```

### 3.4 宏定义

宏定义函数或者代码块, 对于定义的部分需要遵守缩进规范.

例如:

```
1 | #define FOO_FUNC(arg) \  
2 |     { \  
3 |         int i = 0; \  
4 |         if (i == 0) { \  
5 |             NSLog(@"i is zero"); \  
6 |         } \  
7 |     } \
```

## 4.命名

我们应该遵守标准的 [Coding Guidelines for Cocoa](#)。

### 4.1方法命名

方法名应该以小写字母开头，并混合驼峰格式。每个具名参数也应该以小写字母开头。

`+/-` 与 `(returnType)` 之间有一个空格，`(returnType)` 与方法名之间没有空格

例如:

```
1 | - (void)scrollViewDidScroll:(UIScrollView *)scrollView;
```

例外: 当使用熟知的缩写时，缩写字母全部使用大写。如 `URL`、`TIFF` 以及 `EXIF`

### 4.2变量命名

变量尽量以描述性的方式来命名。除了在 `for()` `while()` 循环中，单个字符的变量命名应该尽量避免。

`*` 与变量名紧靠在一起。类型与 `*` 之间应该有一个空格

应该:

```
1 | UIButton *settingsButton;
```

不应该:

```
1 | UIButton * setBtn;
```

属性也是使用驼峰式，但首单词的首字母小写。对属性使用 `auto-synthesis`，而不是手动编写 `@synthesize` 语句，除非你有一个好的理由。



应该:

```
1 | @property (strong, nonatomic) NSString *descriptiveVariableName;
```

不应该:

```
1 | id varnm;
```

## 5.属性特性

当使用 `NSString`，请使用 `copy` 而不是 `strong`。

当使用 `NSDictionary` `NSArray` 集合类型时，请使用 `strong` 而不是 `copy`。

将属性 `@property` 声明为 `nonatomic`，除非你需要原子性。

## 6.语法糖

`NSString`，`NSDictionary`，`NSArray`，和 `NSNumber` 的字面值应该在创建这些类的不可变实例时被使用。请特别注意 `nil` 值不能传入 `NSArray` 和 `NSDictionary` 字面值，因为这样会导致crash。

这是 `xcode 4.4` 中 `LLVM compiler 4.0` 引入的新特性

应该:

```
1 | NSArray *names = @[@"Brian", @"Matt", @"Chris",  
2 |                   @"Alex", @"Steve", @"Paul"];  
3 | NSDictionary *productManagers = @{@"iPhone": @"Kate",  
4 |                                   @"iPad": @"Kamal",  
5 |                                   @"Mobile Web": @"Bill"};  
6 | NSNumber *shouldUseLiterals = @YES;  
7 | NSNumber *buildingStreetNumber = @10018;
```

不应该:

```

1 | NSArray *names = [NSArray arrayWithObjects:
2 |     @"Brian", @"Matt", @"Chris",
3 |     @"Alex", @"Steve", @"Paul", nil];
4 | NSDictionary *productManagers =
5 |     [NSDictionary dictionaryWithObjectsAndKeys:
6 |     @"Kate", @"iPhone", @"Kamal", @"iPad",
7 |     @"Bill", @"Mobile Web", nil];
8 | NSNumber *shouldUseLiterals = [NSNumber numberWithBool:YES];
9 | NSNumber *buildingStreetNumber = [NSNumber numberWithInt:10018];

```

## 7.Nullability Annotations

在XCode6.3中，为了与Swift统一，苹果引入了Nullability Annotations来标记对象是optional还是non-optional。



编译器会默认所有的参数和返回值是 `Nullable` 的，当我们需要显示指定一个参数或返回值非空，可以用 `__nonnull` 修饰。

例如：

```

1 | @property (nonatomic, copy) NSArray * __nonnull items;
2 |
3 | - (NSString * __nonnull)nullableTest:(NSString * __nonnull)test;

```

1. 当传参 or 赋值时，传入 `nil`，则编译器会产生一个 。
2. 当函数返回值是 `nonnull`，并返回 `nil`，则编译器会产生一个 。
3. 苹果还提供了两个宏来减轻我们的工作量：`NS_ASSUME_NONNULL_BEGIN` 和 `NS_ASSUME_NONNULL_END`。包含在这两个宏中的参数和返回值都是 `nonnull` 的。

## 8.泛型与类型延拓

泛型和Nullability一样，只作用于编译期，是为我们开发者服务的另一重要特性。可以在集合中提示集合中存储的数据类型。

例如：

```

1 | NSMutableArray<NSString *> *array = [[NSMutableArray alloc] init];
2 |
3 | NSDictionary<NSString *, NSNumber *> *dict = @{@"key": @(1)};

```

当我们向集合中添加不符合类型标记的对象时，编译器会给出⚠️。

类型延拓是告诉编译器，这里可以返回某一类的子类的指针。

在开发中，我们经常会遇到这样的情况：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton * btn;
    MyArray * array = [[MyArray alloc] init];
    [array.viewArray addObject:btn];

    UIButton * button = [array.viewArray firstObject];
    ⚠️ Incompatible pointer types initializing 'UIButton *' with an expression of type 'UIView * _Nullable'
}
```

以前需要强转，但是以后就不需要了。我们在声明这个数组时加上一个\_\_typeof修饰符：

```
1 | @property (nonnull, strong, nonatomic) NSMutableArray<__typeof UIView *> *viewA
```

警告消失了。

## 9.类别

扩展系统API以及需要在多个程序中复用的类别命名以 `yx_` 为前缀，避免与其他SDK的类别中自定义方法名冲突。

## 10.常量

常量使用 `static` 与 `const` 创建，`#define` 可以用于定义便利方法，但尽量使用 `inline` 代替。

应该：

```
1 | static NSString *const SomeCellIdentifier = @"SomeCellIdentifier";
2 |
3 | static CGFloat const SomeCellHeight = 50.0;
```

不应该：

```
1 | #define cellHeight 2
```

## 11.私有属性

私有变量应该在类的私有类别中，不需要加 `private` 等词语来进行修饰。

私有类别可以在命名为 `<headerfile>+Private.h` 或 `<headerfile>-Private.h` 的文件里提供，也可以在 `.m` 文件中提供。

## 12.枚举类型

当使用 `enum` 时，推荐使用新的固定基本类型规格，因为它有更强的类型检查和代码补全。使用系统宏 `NS_ENUM()` 来帮助和鼓励你使用固定的基本类型。定义可同时生效的枚举，使用 `NS_OPTIONS`，用移位运算符定义枚举值。

应该:

```
1 | typedef NS_ENUM(NSInteger, YXLeftMenuTopItemType) {
2 |     YXLeftMenuTopItemTypeMain,
3 |     YXLeftMenuTopItemTypeSecond,
4 |     YXLeftMenuTopItemTypeThird
5 | };
```

不应该:

```
1 | enum YXLeftMenuTopItemType {
2 |     YXLeftMenuTopItemTypeMain,
3 |     YXLeftMenuTopItemTypeSecond
4 | };
```

## 13.布尔值

永远不要把对象或布尔值直接和 `YES` 进行比较。因为 `YES` 的值为1，`BOOL` 能被设置为8位。

可以重写类继承自 `NSObject` 下的 `-(BOOL)isEqual:(id)obj;` 方法来类的比较。

应该:

```
1 | if (someObject) {}
2 | if (![anotherObject boolValue]) {}
```

不应该:

```
1 | if (someObject == nil) {}
2 | if ([anotherObject boolValue] == NO) {}
3 | if (isAwesome == YES) {} // Never do this.
4 | if (isAwesome == true) {} // Never do this.
```

如果BOOL属性的名字是一个形容词，属性就能忽略"is"前缀，但要指定get访问器的惯用名称。

例如：

```
1 | @property (assign, getter=isEditable) BOOL editable;
```

## 14. 条件语句

一定使用 `{}` 来划定判断后的执行语句；即使执行语非常简单，也不能与条件判断在一行。

如果不这样做，会导致以下错误：

- 添加第二行代码和期望它成为if语句。
- if语句里面一行代码被注释了，然后下一行代码不知不觉地成为if语句的一部分。

### 14.1 三元操作符

只有在确定能够促进代码整洁与清晰的前提下才使用。

三元运算符最好在赋值需要判断时使用。

非BOOL类型变量必须要与其类型变量做出判断后才能使用。

应该：

```
1 | NSInteger value = 5;
2 | result = (value != 0) ? x : y;
3 |
4 | BOOL isHorizontal = YES;
5 | result = isHorizontal ? x : y;
```

不应该：

```
1 | result = a > b ? x = c > d ? c : d : y;
```

## 15. Init方法

`init` 方法中返回值使用 `instancetype` 取代 `id` 作为返回。这样确保编译器正确地推断结果类型。

```
1 - (instancetype)init {
2     self = [super init];
3     if (self) {
4         // ...
5     }
6     return self;
7 }
```

查看关于 [instancetype](#) 的文章

## 16. 黄金路径

条件判断的左侧空间被称为**黄金路径**；

为了减少 `if` 语句嵌套，将返回语句放到 `if` 语句中。

应该:

```
1 - (void)someMethod {
2     if (![someOther boolValue]) {
3         return;
4     }
5
6     //Do something important
7 }
```

不应该:

```
1 - (void)someMethod {
2     if ([someOther boolValue]) {
3         //Do something important
4     }
5 }
```

## 17. 错误处理

当方法通过引用来返回一个错误参数，判断返回值而不是错误变量。

应该:

```
1 | NSError *error;
2 | if (![self trySomethingWithError:&error]) {
3 |     // Handle Error
4 | }
```

不应该:

```
1 | NSError *error;
2 | [self trySomethingWithError:&error];
3 | if (error) {
4 |     // Handle Error
5 | }
```

在成功的情况下，有些Apple的APIs记录垃圾值(garbage values)到错误参数(如果non-NULL)，那么判断错误值会导致false负值和crash。

## 18.单例模式

单例必须线程安全。

不要随意使用单例，当某个对象被定义为static变量所引用，会一直在内存中驻留。

```
1 | + (instancetype)sharedInstance {
2 |     static id sharedInstance = nil;
3 |
4 |     static dispatch_once_t onceToken;
5 |     dispatch_once(&onceToken, ^{
6 |         sharedInstance = [[self alloc] init];
7 |     });
8 |
9 |     return sharedInstance;
10 | }
```

## 19.代码组织

### 19.1View层结构

viewController的代码应该差不多是这样：



```
@property (nonatomic, strong) UIButton *confirmButton
```

```
...
```

```
#pragma mark - life cycle
```

```
viewDidLoad
```

```
viewWillAppear
```

```
...
```

```
#pragma mark - UITableViewDelegate
```

```
methods
```

```
#pragma mark - CustomDelegate
```

```
methods
```

```
#pragma mark - event response
```

```
-(void)didTappedConfirmButton:(UIButton *)confirmButton
```

```
#pragma mark - private methods
```

```
methods
```

```
#pragma mark - getters and setters
```

```
-(UIButton *)confirmButton
```

```
-(UITableView *)tableView
```

```
...
```

要点如下：

所有的属性都使用getter和setter

在viewDidLoad里面只做addSubview和布局的事情，在viewWillAppear里面做Notification的监听之类的事情。至于属性的初始化，则交给getter去做。

比如这样：



```

1  #pragma mark - life cycle
2  - (void)viewDidLoad
3  {
4      [super viewDidLoad];
5
6      self.view.backgroundColor = [UIColor whiteColor];
7      [self.view addSubview:self.firstTableView];
8      [self.view addSubview:self.secondTableView];
9      [self.view addSubview:self.firstFilterLabel];
10     [self.view addSubview:self.secondFilterLabel];
11     [self.view addSubview:self.cleanButton];
12     [self.view addSubview:self.originImageView];
13     [self.view addSubview:self.processedImageView];
14     [self.view addSubview:self.activityIndicator];
15     [self.view addSubview:self.takeImageButton];
16 }
17
18 - (void)viewWillAppear:(BOOL)animated
19 {
20     [super viewWillAppear:animated];
21
22     CGFloat width = (self.view.width - 30) / 2.0f;
23
24     self.originImageView.size = CGSizeMake(width, width);
25     [self.originImageView topInContainer:70 shouldResize:NO];
26     [self.originImageView leftInContainer:10 shouldResize:NO];
27
28     self.processedImageView.size = CGSizeMake(width, width);
29     [self.processedImageView right:10 FromView:self.originImageView];
30     [self.processedImageView topEqualToView:self.originImageView];
31
32     CGFloat labelWidth = self.view.width - 100;
33     self.firstFilterLabel.size = CGSizeMake(labelWidth, 20);
34     [self.firstFilterLabel leftInContainer:10 shouldResize:NO];
35     [self.firstFilterLabel top:10 FromView:self.originImageView];
36
37     ... ...
38 }

```

这样即便在属性非常多的情况下，还是能够保持代码整齐，view的初始化都交给getter去做了。总之就是尽量不要出现以下的情况：

```

1 | - (void)viewDidLoad
2 | {
3 |     [super viewDidLoad];
4 |
5 |     self.textLabel = [[UILabel alloc] init];
6 |     self.textLabel.textColor = [UIColor blackColor];
7 |     self.textLabel ... ..
8 |     self.textLabel ... ..
9 |     self.textLabel ... ..
10 |    [self.view addSubview:self.textLabel];
11 | }

```

getter和setter全部都放在最后

按照顺序来分配代码块的位置，先是 `life cycle`，然后是 `delegate` 方法实现，然后是 `event response`，然后才是 `getters and setters`。这样后来者阅读代码时就能省力很多。

每一个delegate都把对应的protocol名字带上，delegate方法不要到处乱写，写到一块区域里面去

比如UITableViewDelegate的方法集就老老实实写上 `#pragma mark - UITableViewDelegate`。这样有个好处就是，当其他人阅读一个他并不熟悉的Delegate实现方法时，他只要按住command然后去点这个protocol名字，Xcode就能够立刻跳转到对应这个Delegate的protocol定义的那部分代码去，就省得他到处找了。

event response专门开一个代码区域

所有button、gestureRecognizer的响应事件都放在这个区域里面，不要到处乱放。

关于private methods，正常情况下ViewController里面不应该写

ViewController基本上是大部分业务的载体，本身代码已经相当复杂，所以跟业务关联不大的东西能不放在ViewController里面就不要放。另外一点，这个private method的功能这时候只是你用得到，但是将来说不定别的地方也会用到，一开始就独立出来，有利于将来的代码复用。

使用类簇分离独立功能

在视频详情、拍摄等大类中，增加独立功能，要使用类簇的方式。不能让一个文件太大。

## 19.2方法调用

我们在调用类的方法时经常会遇到方法和参数过长的情况，这时候要使用换行，按照冒号对齐

应该:

```
1 | [self appendDownloadJobWithID:name
2 |         downloadURL:url
3 |         outputPath:targetPath
4 |         andCompleteHandler:completeHandler
5 |         completionHandler:progressHandler];
```

不应该:

```
1 | [self appendDownloadJobWithID:name downloadURL:url outputPath:targetPath andCom
```

## 20.Xcode工程

应该使工程中文件的物理路径和逻辑路径统一

推荐使用 [synx](#) 整理工程目录

```
1 | $ synx XXX.xcodeproj
```

整理目录并删除项目中未引用的文件（记得先备份）

```
1 | $ synx -p XXX.xcodeproj
```

代码不仅是根据类型来分组，而且还可以根据功能来分组，这样代码更加清晰。

根据功能模块将代码分组，便于查找维护。

注意黄色warning

尽可能在 `target` 的 `Build Settings` 打开 `Treat Warnings as Errors`

如果要忽略特定warning，使用 [Clang's pragma feature](#)

例如：

```

1  {
2      // ...
3      size_t bytesPerRow = 0;
4      CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
5      CGColorSpaceModel colorSpaceModel = CGColorSpaceGetModel(colorSpace);
6      CGBitmapInfo bitmapInfo = CGImageGetBitmapInfo(imageRef);
7
8      if (colorSpaceModel == kCGColorSpaceModelRGB) {
9          uint32_t alpha = (bitmapInfo & kCGBitmapAlphaInfoMask);
10     #pragma clang diagnostic push
11     #pragma clang diagnostic ignored "-Wassign-enum"
12         if (alpha == kCGImageAlphaNone) {
13             bitmapInfo &= ~kCGBitmapAlphaInfoMask;
14             bitmapInfo |= kCGImageAlphaNoneSkipFirst;
15         } else if (!(alpha == kCGImageAlphaNoneSkipFirst || alpha == kCGImageAl
16             bitmapInfo &= ~kCGBitmapAlphaInfoMask;
17             bitmapInfo |= kCGImageAlphaPremultipliedFirst;
18         }
19     #pragma clang diagnostic pop
20     }
21     // ...
22 }

```

## 参考资料:

- [Google Objective-C 风格指南](#)
- [禅与Objective-C编程艺术](#)
- [Raywenderlich Objective-C Style Guide](#)
- [iOS应用架构谈 view层的组织和调用方案](#)
- [Coding Guidelines for Cocoa](#)
- [Nullability and Objective-C](#)
- [BOOL / bool / Boolean / NSCFBoolean](#)