Table of Contents

iOS-Style-Guide

前言

> 注：本文转载自外刊IT评论的翻译。原文是MarkCC的 Stuff Everyone Should Do (part 2): Coding Standards。

编码规范：大家都应该做的事

我们在Google所做的事情中另外一个让我感到异常有效、有用的制度是严格的编码规范。

在到Google工作之前，我一直认为编码规范没有什么用处。我坚信这些规范都是官僚制度下产生的浪费大家的编程时间、影响人们开发效率的东西。

我是大错特错了。

在谷歌，我可以查看任何的代码，进入所有谷歌的代码库，我有权查看它们。事实上，这种权限是很少人能拥有的。但是，让我感到惊讶的却是，如此多的编码规范 —— 缩进，命名，文件结构，注释风格 —— 这一切让我出乎意料的轻松的阅读任意一段代码，并轻易的看懂它们。这让我震惊 —— 因为我以为这些规范是微不足道的东西。它们不可能有这么大的作用 —— 但它们却起到了这么大的作用。当你发现只通过看程序的基本语法结构就能读懂一段代码，这种时间上的节省不能不让人震撼！

反对编码规范的人很多，下面是一些常见的理由，对于这些理由，我以前是深信不疑。

这是浪费时间！

我是一个优秀的程序员，我不愿意浪费时间干这些愚蠢的事。我的技术很好，我可以写出清晰的、易于理解的代码。为什么我要浪费时间遵守这些愚蠢的规范？答案是：统一是有价值的。就像我前面说的 —— 你看到的任何的一行代码 —— 不论是由你写的，还是由你身边的同事，还是由一个跟你相差11个时区的距离人写的 —— 它们都有统一的结构，相同的命名规范 —— 这带来的效果是巨大的。你只需要花这么少的功夫就能看懂一个你不熟悉(或完全未见过)的程序，因为你一见它们就会觉得面熟。

我是个艺术家！

这种话很滑稽，但它反映了一种常见的抱怨。我们程序员对于自己的编码风格通常怀有很高的自负。我写出的的代码的确能反映出我的一些特质，它是我思考的一种体现。它是我的技能和创造力的印证。如果你强迫我遵守什么愚蠢的规范，这是在打压我的创造力。可问题是，你的风格里的重要的部分，它对你的思想和创造力的体现，并不是藏身于这些微

不足道的句法形式里。(如果是的话，那么，你是一个相当糟糕的程序员。)规范事实上可以让人们可以更容易的看出你的创造力 —— 因为他们看明白了你的作品，人们对你的认识不会因不熟悉的编码形式而受到干扰。

所有人都能穿的鞋不会合任何人的脚！

如果你使用的编码规范并不是为你的项目专门设计的，它对你的项目也许并不是最佳方案。这没事。同样，这只是语法：非最优并不表示是不好。对你的项目来说它不是最理想的，但并不能表明它不值得遵守。不错，对于你的项目，你并没有从中获得该有的好处，但对于一个大型公司来说，它带来的好处是巨大的。除此之外，专门针对某个项目制定编码规范一般效果会更好。一个项目拥有自己的编码风格无可厚非。但是，根据我的经验，在一个大型公司里，你最好有一个统一的编码规范，特定项目可以扩展自己特定的项目方言和结构。

我善长制定编码规范！

这应该是最常见的抱怨类型了。它是其它几种反对声音的混合体，但它却有自身态度的直接表现。有一部分反对者深信，他们是比制定编码规范的人更好的程序员，俯身屈从这些小学生制定的规范，将会降低代码的质量。对于此，客气点说，就是胡扯。纯属傲慢自大，荒唐可笑。事实上他们的意思就是，没有人配得上给他们制定规范，对他们的代码的任何改动都是一种破坏。如果参照任何一种合理的编码规范，你都不能写出合格的代码，那只能说你是个烂程序员。

当你按照某种编码规范进行编程时，必然会有某些地方让你摇头不爽。肯定会在某些地方你的编码风格会优于这些规范。但是，这不重要。在某些地方，编码规范也有优于你的编程风格的时候。但是，这也不重要。只要这规范不是完全的不可理喻，在程序的可理解性上得到的好处会大大的补偿你的损失。

但是，如果编码规范真的是完全不可理喻呢?

如果是这样，那就麻烦了：你被糟蹋了。但这并不是因为这荒谬的编码规范。这是因为你在跟一群蠢货一起工作。想通过把编码规范制定的足够荒谬来阻止一个优秀的程序员写出优秀的代码，这需要努力。这需要一个执著的、冷静的、进了水的大脑。如果这群蠢货能强行颁布不可用的编码规范，那他们就能干出其它很多傻事情。如果你为这群蠢货干活，你的确被糟蹋了 —— 不论你干什么、有没有规范。(我并不是说罕有公司被一群蠢货管理；事实很不幸，我们这个世界从来就不缺蠢货，而且很多蠢货都拥有自己的公司。)

序言

代码风格的重要性对于一个团队和项目来说不言而喻。网上有许多 Objective-C 的代码风格，这篇编码风格指南基于raywenderlich.com的编码规范（有些删减或修改），简洁而又符合苹果的规范，同时有助于养成良好的代码习惯。

原文在这里。 本人才疏学浅，如果有任何翻译不当欢迎在 Issues 中反馈或者直接 Fork 。

背景

这里有些关于编码风格的Apple官方文档，如果有些东西此文档没有提及，可以从以下文档来获取更多细节：

- The Objective-C Programming Language
- Cocoa Fundamentals Guide
- Coding Guidelines for Cocoa
- iOS App Programming Guide

目录

语言

推荐使用美式英语、简洁清晰.

推荐:

```
UIColor *myColor = [UIColor whiteColor];
```

不推荐:

```
UIColor *myColour = [UIColor whiteColor];
```

代码组织

`#pragma mark -` 是一个在类内部组织代码并且帮助你分组方法实现的好办法。 我们建议使用 `#pragma mark -` 来分离:

- 不同功能组的方法
- 协议/代理 的实现

参考如下结构:

Objective-C

```
#pragma mark - Lifecycle
- (instancetype)init {}
- (void)dealloc {}
- (void)viewDidLoad {}
- (void)viewWillAppear:(BOOL)animated {}
- (void)didReceiveMemoryWarning {}

#pragma mark - Custom Accessors
- (void)setCustomProperty:(id)value {}
- (id)customProperty {}

#pragma mark - IBActions/Event Response
- (IBAction)submitData:(id)sender {}
- (void)someButtonDidPressed:(UIButton*)button

#pragma mark - Protocol conformance
#pragma mark - UITextFieldDelegate
#pragma mark - UITableViewDataSource
#pragma mark - UITableViewDelegate

#pragma mark - Public
- (void)publicMethod {}

#pragma mark - Private
- (void)privateMethod {}

#pragma mark - NSCopying
- (id)copyWithZone:(NSZone *)zone {}

#pragma mark - NSObject
- (NSString *)description {}
```

间距

- 推荐缩进使用**TAB**,确保在Xcode偏好设置来设置 TAB为4个空格。(不推荐使用空格如使用空格 一个缩进使用 **4** 个空格, raywenderlich.com使用**2**个空格)
- 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。

推荐:

```
if (user.isHappy) {
    //Do something
} else {
    //Do something else
}
```

不推荐:

Objective-C

```
if (user.isHappy)
{
  //Do something
}
else {
  //Do something else
}
```

- 在方法之间应该有且只有一行，这有助于视觉清晰度和代码组织性。在方法中的功能块之间应该使用空白分开，但通常都抽离出来一个新方法。
- 优先使用auto-synthesis。但如果有必要，`@synthesize` 和 `@dynamic` 在实现中每个声明都应新起一行。
- 应该避免以冒号对齐的方式来调用方法。有时方法签名可能有3个以上的冒号而冒号对齐会使代码更加易读。请**不要**这样做，尤其是用冒号对齐的方法包含代码块，因为Xcode的对齐方式令它难以辨认。

推荐:

```
// blocks are easily readable
[UIView animateWithDuration:1.0 animations:^{
  // something
} completion:^(BOOL finished) {
  // something
}];
```

不推荐:

```
// colon-aligning makes the block indentation hard to read
[UIView animateWithDuration:1.0
                 animations:^{
                     // something
                 }
                 completion:^(BOOL finished) {
                     // something
                 }];
```

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同步维护更新最新或者干脆就删除，错误的注释不如不要。

通常应避免使用块注释，代码本身应尽可能像文档一样表示意图，做到自解释，只需少量的打断注释。例外：这不适用于生成文档的注释

命名

尽可能遵守Apple的命名约定，尤其是和 内存管理规则 (NARC) 相关的地方。

推荐长的，描述性的方法和变量命名。

推荐:

```
UIButton *settingsButton;
```

不推荐:

```
UIButton *setBut;
```

三个字符前缀应该经常用在类和常量命名，但在Core Data的实体名中应被忽略。如raywenderlich.com官方的书、初学者工具包或教程，前缀'RWT'应该被使用。

常量应该使用驼峰式命名规则，所有的单词首字母大写和加上与类名有关的前缀。

推荐:

```
static NSTimeInterval const RWTTutorialViewControllerNavigationFadeAnimationDuration = 0.3
;
```

不推荐:

```
static NSTimeInterval const fadetime = 1.7;
```

属性也应使用驼峰式，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。

推荐:

```
@property (nonatomic, copy) NSString *descriptiveVariableName;
```

不推荐:

```
id varnm;
```

下划线

当使用属性时，实例变量应该使用 `self.` 来访问和改变。这就意味着所有属性将会视觉效果不同，因为它们前面都有 `self.` 。

但有一个特例：在初始化方法里，实例变量(例如，_variableName)应该直接被使用来避免getters/setters潜在的副作用。

局部变量不应该包含下划线。

方法

在方法签名中，应该在方法类型(-/+ 符号)之后有一个空格。在方法各个段之间应该也有一个空格(符合Apple的风格)。在参数之前应该包含一个具有描述性的关键字来描述参数。

"and"这个词的用法应该保留。它不应该用于多个参数说明，就像 `initWithWidth:height` 这个例子：

推荐:

```
- (void)setExampleText:(NSString *)text image:(UIImage *)image;
- (void)sendAction:(SEL)aSelector to:(id)anObject forAllCells:  (BOOL)flag;
- (id)viewWithTag:(NSInteger)tag;
- (instancetype)initWithWidth:(CGFloat)width height:(CGFloat)height;
```

不推荐:

```
-(void)setT:(NSString *)text i:(UIImage *)image;
- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;
- (id)taggedView:(NSInteger)tag;
- (instancetype)initWithWidth:(CGFloat)width andHeight:(CGFloat)height;
- (instancetype)initWith:(int)width and:(int)height;  // Never do this.
```

变量

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

星号 `*` 表示变量是指针。例如， `NSString *text` 既不是 `NSString* text` 也不是 `NSString * text` ，除了一些特殊情况下常量。

私有变量 应该尽可能代替实例变量的使用。尽管使用实例变量是一种有效的方式，但更偏向于使用属性来保持代码一致性。

应该尽量避免通过使用'back'属性(_variable，变量名前面有下划线)直接访问实例变量，除了在初始化方法(`init` , `initWithCoder:` , 等...), `dealloc` 方法和自定义的setters和getters。想了解关于如何在初始化方法和dealloc直接使用Accessor方法的更多信息，查看这里

推荐:

```
@interface RWTTutorial : NSObject

@property (nonatomic, copy) NSString *tutorialName;

@end
```

不推荐:

```
@interface RWTTutorial : NSObject {
    NSString *tutorialName;
}
```

属性特性修饰符

属性特性应该显式地列出来，有助于新手阅读代码。属性特性的顺序推荐是原子性、内存管理、读写性。

推荐:

```
@property (nonatomic, strong, readonly) NSObject *object;
@property (nonatomic, assign) NSTimeInterval duration;
```

不推荐:

```
@property (strong, nonatomic, readonly) NSObject *object;
@property (nonatomic) NSTimeInterval duration;
```

NSString应该使用 `copy` 而不是 `strong` 的属性特性, 为什么?

1. 因为父类指针可以指向子类对象,使用 copy 的目的是为了让本对象的属性不受外界影响,使用 copy 无论给我传入是一个可变对象还是不可对象,我本身持有的就是一个不可变的副本.
2. 如果我们使用是 strong ,那么这个属性就有可能指向一个可变对象,如果这个可变对象在外部被修改了,那么会影响该属性.

> copy 此特质所表达的所属关系与 strong 类似。然而设置方法并不保留新值，而是将其“拷贝” (copy)。 当属性类型为 NSString 时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个 NSMutableString 类的实例。这个类是 NSString 的子类，表示一种可修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变” (immutable)的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的” (mutable)，就应该在设置新属性值时拷贝一份。

推荐:

```
@property (nonatomic, copy) NSString *tutorialName;
```

不推荐:

```
@property (nonatomic, strong) NSString *tutorialName;
```

点符号语法

点语法是一种很便利保障访问方法调用的方式语法糖。当你使用点语法时，属性被访问或修改通过实际是使用getter或setter方法。进一步了解，阅读这里

点语法应该**总是**被用来访问和修改属性，[]符号更适用于于用在其他实例方法调用。使用点语法 会让代码更加简洁清晰并能帮助区分属性访问和普通方法调用

推荐:

```
view.backgroundColor = [UIColor orangeColor];
[UIApplication sharedApplication].delegate;
```

不推荐:

```
[view setBackgroundColor:[UIColor orangeColor]];
UIApplication.sharedApplication.delegate;
```

字面值

`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。

推荐:

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

不推荐:

```
NSArray *names = [NSArray arrayWithObjects:@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul", nil];
NSDictionary *productManagers = [NSDictionary dictionaryWithObjectsAndKeys: @"Kate", @"iPhone", @"Kamal", @"iPad", @"Bill", @"Mobile Web", nil];
NSNumber *shouldUseLiterals = [NSNumber numberWithBool:YES];
NSNumber *buildingStreetNumber = [NSNumber numberWithInteger:10018];
```

常量

常量首选内联字符串字面量或数字，因为常量可以轻易重用并且可以快速改变而不需要查找和替换。常量应该声明为 `static` 常量而不是 `#define` ，除非非常明确地要当做宏来使用。

推荐:

```
static NSString * const RWTAboutViewControllerCompanyName = @"RayWenderlich.com";

static CGFloat const RWTImageThumbnailHeight = 50.0;
```

不推荐:

```
#define CompanyName @"RayWenderlich.com"

#define thumbnailHeight 2
```

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。现在SDK有一个宏 `NS_ENUM()` 来促进和鼓励你使用固定的基本类型。

例如:

Objective-C

```
typedef NS_ENUM(NSInteger, RWTLeftMenuTopItemType) {
  RWTLeftMenuTopItemMain,
  RWTLeftMenuTopItemShows,
  RWTLeftMenuTopItemSchedule
};
```

你也可以显式地赋值(展示旧的k-style常量定义):

```
typedef NS_ENUM(NSInteger, RWTGlobalConstants) {
  RWTPinSizeMin = 1,
  RWTPinSizeMax = 5,
  RWTPinCountMin = 100,
  RWTPinCountMax = 500,
};
```

旧的k-style常量定义应该**避免**,除非编写Core Foundation C的代码（不太可能用到）。

不推荐:

```
enum GlobalConstants {
  kMaxPinSize = 5,
  kMaxPinCount = 500,
};
```

Case语句

大括号在case语句中并不是必须的，除非编译器强制要求。当一个case语句包含多行代码时，大括号应该加上。

```
switch (condition) {
  case 1:
    // ...
    break;
  case 2: {
    // ...
    // Multi-line example using braces
    break;
  }
  case 3:
    // ...
    break;
  default:
    // ...
    break;
}
```

Objective-C

有时可以使用 fall-through 在不同的case里执行同一段代码。一个fall-through是指移除case的'break'语句，这样就能够允许执行流程跳转到下一个case值。为了代码更加清晰，一个fall-through需要注释一下。

```
switch (condition) {
  case 1:
    // ** fall-through! **
  case 2:
    // code executed for values 1 and 2
    break;
  default:
    // ...
    break;
}
```

当在switch使用枚举类型时，'default'是不需要的。例如：

```
RWTLeftMenuTopItemType menuType = RWTLeftMenuTopItemMain;

switch (menuType) {
  case RWTLeftMenuTopItemMain:
    // ...
    break;
  case RWTLeftMenuTopItemShows:
    // ...
    break;
  case RWTLeftMenuTopItemSchedule:
    // ...
    break;
}
```

私有属性

私有属性应该在类的实现文件中的类扩展(匿名分类)中声明，不应在命名分类(比如 `RWTPrivate` 或 `private`)中定义私有属性除非是扩展其他类。匿名分类可以通过使用+Private.h文件的命名规则暴露给测试。

例如:

```
@interface RWTDetailViewController ()

@property (nonatomic, strong) GADBannerView *googleAdView;
@property (nonatomic, strong) ADBannerView *iAdView;
@property (nonatomic, strong) UIWebView *adXWebView;

@end
```

布尔值

Objective-C使用 `YES` 和 `NO` 。因此 `true` 和 `false` 只应在CoreFoundation，C或C++代码使用。既然 `nil` 解析成 `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` 进行比较，因为 `YES` 被定义为 1，而 `BOOL` 可以多达 8 位。

这是为了在不同文件保持一致性和在视觉上更加简洁而考虑。

推荐:

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

不推荐:

```
if (someObject == nil) {}
if ([anotherObject boolValue] == NO) {}
if (isAwesome == YES) {} // Never do this.
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，但应为 get 访问器指定一个惯用的名字。例如：

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

内容和示例引用自Cocoa命名指南

条件语句

条件判断主体部分应该始终使用大括号括住来防止出错，即使它可以不用大括号（例如它只需要一行）。这些错误包括添加第二行（代码）并希望它是 if 语句的一部分。另外当 if 语句里面的一行被注释掉，下一行就会在不经意间成为了这个 if 语句的一部分时 可能导致更危险的影响。此外，这种风格也更符合所有其他的条件判断，因此也更容易检查。

推荐:

```
if (!error) {
  return success;
}
```

不推荐:

```
if (!error)
  return success;
```

或

```
if (!error) return success;
```

三元操作符

三元操作符`?:`,应该只用在它能让更加清晰或整洁的地方。单个条件求值常常需要它。多个条件求值通常会让语句更近难以理解，不如使用`if`语句或重构成实例变量。一般来说，最好使用三元操作符是在根据条件来赋值的情况下。

非布尔值类型的变量与某东西比较，加上括号()会提高可读性。如果被比较的变量是boolean类型，那么就不需要括号。

推荐:

```
NSInteger value = 5;
result = (value != 0) ? x : y;

BOOL isHorizontal = YES;
result = isHorizontal ? x : y;
```

不推荐:

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

当三元运算符的第二个参数（if 分支）返回和条件语句中已经检查的对象一样的对象的时候，下面的表达方式更灵巧:

推荐:

```
result = object ? : [self createObject];
```

不推荐:

```
result = object ? object : [self createObject];
```

Init方法

Init方法应该遵循Apple生成代码模板的命名规则。返回类型应该使用 `instancetype` 而不是 `id`

```
- (instancetype)init {
  self = [super init];
  if (self) {
    // ...
  }
  return self;
}
```

查看关于instancetype的文章Class Constructor Methods

类构造方法

当类构造方法被使用时，它应该返回类型是 `instancetype` 而不是 `id` 。这样确保编译器正确地推断结果类型。

```
@interface Airplane
+ (instancetype)airplaneWithType:(RWTAirplaneType)type;
@end
```

关于更多instancetype信息，请查看NSHipster.com

CGRect函数

当访问 `CGRect` 里的 `x` ， `y` ， `width` ，或 `height` 时，应该使用 `CGGeometry` 函数而不是直接通过结构体来访问。引用Apple的 `CGGeometry` :

> 在这个参考文档中所有的函数，接受CGRect结构体作为输入，在计算它们结果前隐式地标准化这些rectangles。因此，你的应用程序应该避免直接访问和修改保存在CGRect数据结构中的数据。相反，使用这些函数来操纵rectangles和获取它们的特性。

推荐:

```
CGRect frame = self.view.frame;

CGFloat x = CGRectGetMinX(frame);
CGFloat y = CGRectGetMinY(frame);
CGFloat width = CGRectGetWidth(frame);
CGFloat height = CGRectGetHeight(frame);
CGRect frame = CGRectMake(0.0, 0.0, width, height);
```

不推荐:

```
CGRect frame = self.view.frame;

CGFloat x = frame.origin.x;
CGFloat y = frame.origin.y;
CGFloat width = frame.size.width;
CGFloat height = frame.size.height;
CGRect frame = (CGRect){ .origin = CGPointZero, .size = frame.size };
```

黄金路径

当使用条件语句编码时，左手边的代码应该是"黄金" 或 "快乐"的路径。也就是不要嵌套 `if` 语句。使用多个 return 可以避免增加循环的复杂度，并提高代码的可读性。因为方法的重要部分没有嵌套在分支里面，并且你可以清楚地找到相关的代码。

推荐:

```
- (void)someMethod {
  if (![someOther boolValue]) {
    return;
  }

  //Do something important
}
```

不推荐:

```
- (void)someMethod {
  if ([someOther boolValue]) {
    //Do something important
  }
}
```

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。

推荐:

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

不推荐:

```
NSError *error;
[self trySomethingWithError:&error];
if (error) {
  // Handle Error
}
```

在成功的情况下，有些Apple的APIs记录垃圾值(garbage values)到错误参数(如果非NULL)，所以判断 erroe变量值 可能会导致错误（甚至崩溃）。

单例模式

如果可能，请尽量避免使用单例而是依赖注入。

然而，如果一定要用，请使用一个线程安全的模式来创建共享的实例。对于 GCD，用 `dispatch_once()` 函数就可以咯。

```
+ (instancetype)sharedInstance
{
   static id sharedInstance = nil;
   static dispatch_once_t onceToken = 0;
   dispatch_once(&onceToken, ^{
      sharedInstance = [[self alloc] init];
   });
   return sharedInstance;
}
```

使用 dispatch_once()，来控制代码同步，取代了原来的约定俗成的用法。

```
+ (instancetype)sharedInstance
{
    static id sharedInstance;
    @synchronized(self) {
        if (sharedInstance == nil) {
            sharedInstance = [[MyClass alloc] init];
        }
    }
    return sharedInstance;
}
```

`dispatch_once()` 的优点是，它更快，而且语法上更干净，因为dispatch_once()的意思就是 “把一些东西执行一次”，就像我们做的一样。 这样同时可以避免 有时可能产生的许多崩溃.

经典的单例对象是：一个设备的GPS以及它的加速度传感器(也称动作感应器)。

虽然单例对象可以子类化，但这种方式能够有用的情况非常少见。

必须有证据表明，给定类的接口趋向于作为单例来使用。

所以，单例通常公开一个 `sharedInstance` 的类方法就已经足够了，没有任何的可写属性需要被暴露出来。

尝试着把单例作为一个对象的容器，在代码或者应用层面上共享，是一个糟糕和丑陋的设计。

> NOTE：单例模式应该运用于类及类的接口趋向于作为单例来使用的情况

换行符

换行符是一个很重要的主题，因为它的风格指南主要为了打印和在线浏览的可读性。

例如:

```
self.productsRequest = [[SKProductsRequest alloc] initWithProductIdentifiers:productIdenti
fiers];
```

一个像上面的长行的代码在第二行以一个间隔（2个空格）延续。

```
self.productsRequest = [[SKProductsRequest alloc]
  initWithProductIdentifiers:productIdentifiers];
```

Xcode工程

物理文件应该与Xcode工程文件保持同步来避免文件扩张。任何Xcode分组的创建应该在文件系统的文件体现。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。

尽可能在target的Build Settings打开"Treat Warnings as Errors，和启用以下additional warnings。如果你需要忽略特殊的警告，使用 Clang's pragma feature。

其他Objective-C编码规范

如果我们的编码规范不符合你的口味，可以查看其他的编码规范：

- Robots & Pencils
- New York Times

- Google
- GitHub
- Adium
- Sam Soffes
- CocoaDevCentral
- Luke Redpath
- Marcus Zarra

Swift

TODO