iOS代码质量

# 官方代码规范

<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.pdf>

# 本人总结

## 在.h文件，也就是头文件少应用其他的文件，使用@class代替

## 2. 多类型处理，少用＃define预处理，在.h文件中声明

extern NSString \*const XFExternalConst;

在.m文件中描述

NSString \*const XFExternalConst = @"ko";

为避免冲突，一般都用类名做前缀。

## 3. 用枚举表示状态、选项、状态码

## 4. 使用清晰而协调的命名方式

驼峰命名法

方法与变量以小写字母开头

类名以大写字母开头

不要使用str这种简称，而用string这样的全称

Boolean属性应该加is前缀，如果某方法返回非属性的Boolean值，应该根据功能选用has或者is当前缀

## 5. 多用块枚举，少用 for 循环

## 6. 将类的实现代码分散到便于管理的数个分类之中

把一个类中的几个不同模块方法写到别的文件中，合理使用category。

# 选自博客一

**前言**

Segment Fault：<http://segmentfault.com/a/1190000003694112>

（首发在 Segment Fault）

这是一篇读书笔记，快速记录各种高效率编程的技巧和方法。这些方法是为了提升编码质量和效率，高质量代码利于后期的维护和更新，毕竟不能一份代码到永远。

由于是记录形式，当然不能把整篇内容都写下来，只记录关键性的内容，长期更新。

**正文**

Objective-C使用了消息机制代替调用方法。

区别：使用消息结构的语言，其运行时缩影执行的代码由运行环境来决定。而使用函数调用的语言，则又编译器决定。

**头文件中少引用其他文件**

在头文件中使用@Class代替直接引用其他头文件

**多使用字面量语法**

NSNumber \*intNumber = @1;

NSNumber \*floatNumber = @2.5f;

NSNumber \*doubleNumber = @3.1415926;

NSNumber \*boolNumber = @YES;

NSNumber \*charNumber = @'a';

int a = 3;

float b = 2.1;

NSNumber \*c = @(a\*b);

NSArray \*animals = @[@"cat",@"dog",@"monkey"];

NSString \*dog = animals[1];

NSDictionary \*dataDict = @{ @"firstName" : @"aa",

@"lastName" : @"bb",

@"age" : @20 };

NSString \*lastName = dataDict[@"lastName"];

NSMutableArray \*mutableArray = animals.mutableCopy;

**多用类型常量，少用#define预处理**

如果只在本类使用的常量，使用static const关键字来定义常量。

如果多个类都需使用到某一常量，则需将常量定义成公开的，具体方式是在类的声明文件中使用extern const关键字声明常量，在类的实现文件中使用const关键字定义常量，这样任何类只要导入了声明常量的头文件就可以直接使用定义好的常量了。

在.h文件中声明

extern NSString \*const XFExternalConst;

在.m文件中描述

NSString \*const XFExternalConst = @"ko";

为避免冲突，一般都用类名做前缀。

**用枚举表示状态、选项、状态码**

枚举只是一种常量命名方式，某个对象所经历的各种状态可以定义为一个枚举集。

编译器会为枚举分配一个独有的编号，从0开始每个递增加1.实现枚举所用的数据类型取决于编译器，不过其二进制位的个数必须能完全表示枚举编号才行。

enum ConnectionState {

ConnectionStateDisconnected,

ConnectionStateConnecting,

ConnectionStateConnected

};

typedef enum ConnectionState ConnectingState;

还可以不使用编译器所分配的编号，手工指定某个枚举成员所对应的值。

还有一种情况应该使用枚举类型，那就是定义选项的时候。若这些选项可以彼此组合，则更应该如此。只要枚举定义的对，各选项之间就可以通过“按位或操作符”来组合。

凡是需要以按位或操作来组合的枚举都应该用NS\_OPTIONS宏，如果没有组合需求，就用NS\_ENUM宏。

typedef NS\_ENUM(NSInteger, UIViewAnimationTransition) {

UIViewAnimationTransitionNone,

UIViewAnimationTransitionFlipFromLeft,

UIViewAnimationTransitionFlipFromRight,

UIViewAnimationTransitionCurlUp,

UIViewAnimationTransitionCurlDown,

};

typedef NS\_OPTIONS(NSUInteger, UIViewAutoresizing) {

UIViewAutoresizingNone = 0,

UIViewAutoresizingFlexibleLeftMargin = 1 << 0,

UIViewAutoresizingFlexibleWidth = 1 << 1,

UIViewAutoresizingFlexibleRightMargin = 1 << 2,

UIViewAutoresizingFlexibleTopMargin = 1 << 3,

UIViewAutoresizingFlexibleHeight = 1 << 4,

UIViewAutoresizingFlexibleBottomMargin = 1 << 5

};

枚举在switch语句里面的时候，不需要加default分支。

**属性的概念**

基本方法就不描述了。

@dynamic关键字，表示不要自动创建实现属性所有的实例变量，也不要为其创建存取方法。即使编译器没有发现定义存取方法，也不会报错，它相信这些方法能在运行期找到。

属性的四种特质

* 原子性

默认情况下，编译器合成的方法会锁定机制保持atomic。如果使用nonatomic，则不使用同步锁。

* 读写权限

readwrite的属性具有getter和setter方法

readonly的属性仅具有getter方法

* 内存管理语义

assign只针对“纯量类型”，比如CGFloat或者NSInteger

strong表示该属性定义了一种拥有关系。为这种属性设置新值时，设置方法会先保留新值，并释放旧值，然后将新值设置上去

weak表示该属性定义另一种非拥有关系。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。和assign类似，然而在属性所指的对象遭到摧毁时，属性值也会清空

unsafe\_unretained这个和assign相同，但是它适用于对象类型，该特质表达一种非拥有关系，当目标对象遭到摧毁时，属性值不会自动清空，是不安全的

copy表达的所属关系和strong类型。然后设置方法并不保留新值，而是将其拷贝。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可能指向一个NSMutableString类的实例。如果不是拷贝的花，那么设置完属性以后，字符串的值可能会在对象不知情的情况下遭人更改。所以这个时候需要拷贝一份不可变的字符串。

* 方法名

getter=<name> 指定getter的方法名。如果属性是Boolean型，在方法名加上is前缀，就可以用这个方法来指定。

setter=<name> 指定setter的方法名。这个不常见。

**在对象内部尽量直接访问实例变量**

懒加载是重写getter方法

**理解对象等同性的概念**

按照==操作符比较出来的结果未必是我们想要的，因为该操作符比较出来的是两个指针本身，而不是指针所指的对象。应该是用NSObject协议中声明的isEqual方法来判断两个对象的等同性。来办来说两个类型不同的对象总是不相等的。

NSString \*oneStr = @"aaa 21";

NSString \*twoStr = [NSString stringWithFormat:@"aaa %d",21];

BOOL equalA = (oneStr == twoStr);//NO

BOOL equalB = [oneStr isEqual:twoStr];//YES

BOOL equalC = [oneStr isEqualToString:twoStr];//YES

两个用于判断等同性的关键方法

- (BOOL)isEqual:(id)object;

@property (readonly) NSUInteger hash;

默认实现是：当且仅当其指针值完全相等时，这两个对象才相等。

**几个要点**

* 若想监测对象的等同性，提供isEqual:与 hash 方法
* 相同的对象必须具有相同的哈希码，但是两个哈希码相同的对象却未必相同
* 不要盲目逐个检测每条属性，而是应该依照具体需求来制定监测方案
* 编写 hash 方法时，应该是用计算速度快而且碰撞低的算法

**以“类族模式”隐藏实现细节**

核心套路就是类似UIButton，创建的时候传入一个枚举值，根据枚举值来创建子类。（这里的笔记是我看懂以后写的，不知道的朋友先搜索一下工厂模式，其实就是那个意思）

**在既有类中使用关联对象存放自定义数据**

有时候需要在对象中存放相关信息，这时候我们通常会从对象所属的类中继承一个子类，然后改用这个子类对象。然而并非所有情况下都能这么做，有时候类的实例可能由某种机制创建。Objective-C有一种强大机制叫关联对象。

这种机制要小心使用，因为会使代码失控。

**理解objc\_msgSend的作用**

原型

void objc\_msgSend(id self, SEL cmd, ...)

一个例子

id returnValue = [someObject messageName:parameter];

编译器会把它转换为以下函数

id returnValue = objc\_msgSend(someObject,@selector(messageName:),parameter);

为了完成调用方法，该方法需要在接受者所属的类中搜寻其方法列表，如果能找到，就跳转过去。如果找不到，就沿着继承体系向上继续查找，等找到合适的再跳转。如果最终还是找不到，就执行消息转发的操作。

一些**边界情况**，则交由另一些函数处理

* objc\_msgSend\_stret 如果待发送的消息要返回结构体，可交此函数处理。
* objc\_msgSend\_fpret 如果消息返回的是浮点数，可交由此函数处理。
* ojbc\_msgSendSuper 如果要给超类发送消息，例如[super message:parameter]，那么就就交由此函数处理。

**理解消息转发机制**

消息转发分为两大阶段。

第一阶段先问接受者，所属的类，看其是否能动态添加方法，以处理当前这个unknown selector，这称为dynamic method resolution。

第二阶段涉及full forwarding mechanism。如果运行期系统已经把第一阶段执行完了，那么接受者自己就无法再以动态新增方法的手段来响应包含该selector的消息了。此时，运行期系统会请求接受者以其他手段来处理与消息相关的方法调用。然后又分两部。

首先，让接受者看看有没有其他对象能处理这条消息。如果有，就转发给那个对象。

如果没有，就会启动完整的消息转发机制，运行期系统会把消息有关的全部细节封装到NSInvocation对象中，再给接受者最后一次机会，让它设法解决当前还未处理的这条消息。

**动态方法解析**

对象收到无法解读的消息后，先调用

+ (BOOL)resolveInstanceMethod:(SEL)sel

该方法的参数就是那个未知的selector，返回Boolean类型，表示这个类是否能新增一个实例方法用来处理这个selector。在继续走下去之前，这有个机会新增一个处理的方法。

如果尚未实现的不是实例方法而是类方法，则调用

+ (BOOL)resolveClassMethod:(SEL)sel

使用他们的前提是，相关方法的实现代码已经写好，只等着运行的时候动态插入在类里面。

这个常常用来实现@dynamic属性。

**后备接收者**

当前接收者还有第二次机会处理，能不能把消息转发给其他接收者

- (id)forwardingTargetForSelector:(SEL)aSelector

找得到就返回对象，找不到就返回nil。

**完整的消息转发**

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

先创建NSInvocation对象，把尚未处理的那条消息有关的全部细节都封在其中。此对象包含selector，target以及参数。

继承体系中的每个类都有机会处理此调用请求，直到NSObject。如果还没有找到，那么该方法还会继续调用doesNotRecognizeSelector:抛出异常，此异常表示最终未能处理。

这个机制属于底层机制，可以动态注入方法，甚至之前的可以动态注入属性，云后端服务商可以说基本就靠这个套路，通过KVC的样子往类里面添加属性。

**用方法调配技术调试黑盒方法**

黑科技。

IMP指针，改方法实现，替换系统方法，可以多添加日志打印。

**类对象**

OC 是一门极其动态的语言。

每个 OC 对象实例都是指向某块内存数据的指针。

typedef struct objc\_object {

Class isa;

} \*id;

每个对象结构体的首个成员是Class类的变量。该变量定义了对象所属的类，通常称为is a指针。

typedef struct objc\_class \*Class;

struct objc\_class {

Class isa;

Class super\_class;

const char \*name;

long version;

long info;

long instance\_size;

struct objc\_ivar\_list \*ivars;

struct objc\_method\_list \*\*methodLists;

struct objc\_cache \*cache;

struct objc\_protocol\_list \*protocols;

};

此结构体存放类的元数据，例如类的实例实现了几个方法，具备多少个实例变量等信息。 首个变量是isa指针，说明Class本身也是 OC 对象。

super\_class定义了本类的超类。类对象所属的类型是另一个类，叫做超类。

每个类仅有一个类对象，而每个类对象仅有一个与之相关的元类。

class方法所返回的类表示发起代理的对象，而非接受代理的对象。

**用前缀避免命名空间冲突**

开发者可能会忽视另外一个容易引发命名冲突的地方，那就是类的实现文件中所用的纯 C 函数及全局变量。

**提供全能初始化方法**

所有对象均要初始化。

提供一个全能初始化方法，其他的几种初始化方法调用它。

如果全能初始化方法与超类不同，则需覆写超类中的对应方法。

**实现description方法**

重写- (NSString \*)description

控制台- (NSString \*)debugDescription

**尽量使用不可变对象**

尽量把对外公布出来的属性设为只读，只在必要时候对外公布。

有时候想修改封装在对象内部的数据，但是却不想让外人所改动。这种情况需要将readonly在.m文件中重新生成readwrite。但是为了避免产生意外，需要在必要时通过dispatch queue来实现。

不要把可变的内容作为属性公开，而是提供相关方法，以此修改对象中的可变内容。

**使用清洗而协调的命名方式**

驼峰命名法

方法与变量以小写字母开头

类名以大写字母开头

不要使用str这种简称，而用string这样的全称

Boolean属性应该加is前缀，如果某方法返回非属性的Boolean值，应该根据功能选用has或者is当前缀

**类与协议的命名**

为类与协议的名称加上前缀，以避免命名空间的冲突

委托一般使用委托的发起方名称后面跟一个Delegate

**为私有方法名加前缀**

一般可以使用p\_作为前缀，表示私有方法

不要用一个单独的下划线作为私有方法的前缀

**理解Objective-C错误模型**

异常NSException应该用于极其严重的错误，比如编写了某个抽象基类，它的正确用法是先从重继承一个子类，然后再使用这个子类。在这种情况下，如果有人直接使用了这个抽象基类，那么可以考虑抛出异常。

NSError的用法很灵活，封装了三条信息

* Error domain 错误范围，类型为字符串 错误发生的范围，通常用一个特有的全局变量来定义。
* Error code 错误码，类型为证书 独有的错误代码。这种错误通常采用enum来定义，比如 HTTP 请求返回的状态码。
* User info 用户信息，类型为字典 有关此错误的额外信息，其中或许包含一段*本地化描述*，或许还包含导致该错误发生的另外一个错误，经由此种信息，可将相关错误传承一条chain of errors。

**理解NSCopying协议**

使用对象时经常需要拷贝它。如果想令自己的类支持拷贝操作

- (id)copyWithZone:(NSZone \*)zone;

为什么会出现NSZone，以前开发的时候，会把内存分成不同的zone，而对象会创建在某个区里面。现在不用了，每个程序只有一个default zone

另外一个NSMutableCopying协议，返回可变的副本

- (id)mutableCopyWithZone:(NSZone \*)zone;

**深拷贝** 在拷贝对象自身时，将底层数据也一并复制过去

**浅拷贝** Foundation框架中所有的容器类默认情况下执行浅拷贝，只拷贝对象本身，不复制数据 因为不是所有对象都能拷贝，而且调用者也未必需要都一一拷贝。

**通过委托与数据源协议进行对象间通信**

委托属性要定义成weak，因为两者之间必须为非拥有关系

- (BOOL)respondsToSelector:(SEL)aSelector;

也可以用协议定义一套接口，令某类从该接口获取所需的数据。委托模式的这种用法是向类提供数据，所以成为dataSource。在这种模式中，信息从数据源流向类。而在常规的代理模式中，信息则从类流向受委托者。

**将类的实现代码分散到便于管理的数个分类之中**

把一个类中的几个不同模块方法写到别的文件中，合理使用category。

**不要在分类中声明属性**

除了extension外，其他的分类都无法向类中新增实例变量

声明为@dynamic，然后动态添加

**使用extension隐藏实现细节**

**通过协议提供匿名对象**

使用匿名对象来隐藏类型名称

**理解引用计数**

retain 增计数 release 减计数 autorelease 待稍后清理autorelease pool时，再减少计数

对象创建出来时，其保留计数至少为1

**自动释放池**

**循环引用**

**以ARC简化引用计数**

若方法名以下列词语开头，则返回的对象归调用者所有

* alloc
* new
* copy
* mutableCopy

在应用程序中，可用下列修饰符来改变局部变量与实例变量的语义

\_\_strong 默认语义，保留这个值

\_\_unsafe\_unretained 不保留这个值，这么做可能不安全，因为等到再次使用变量时，其对象可能已经回收了

\_\_weak 不保留这个值，但是变量可以安全使用，因为如果系统把这个对象回收了，那么变量也会自动清空

\_\_autoreleasing 把对象*按引用传递*给方法时，使用这个特殊的修饰符，此值在方法返回时自动释放

比如，想令实例变量的语义与不使用 ARC 时相同，可以使用\_\_weak或\_\_unsafe\_unretained修饰符

block 块会自动保留其所捕获的全部对象，而如果这其中有某个对象又保留了块本身，那么就可能导致循环引用，可以用\_\_weak局部变量来打破这种循环引用

注意：CoreFoundation对象不归 ARC 管理，开发者必须适时调用CFRetain/CFRelease

**在dealloc方法中只释放引用并解除监听**

把原来配置过的观测行为都清除掉，如果使用NSNotificationCenter给此对象注册过某种通知，那么一般应该在这里注销

**使用弱引用来避免循环引用**

**理解Block**

如果block所捕获的变量是对象类型，那么就会自动保留它。系统在释放这个块的时候，也会将其一并释放。这引出一个重要问题。block块本身可视为对象，也有引用计数。

如果将block块定义在实例方法中，那么除了可以访问类的所有实例变量之外，还可以使用 self 变量，块总能修改实例变量，所以在声明时无需加\_\_block。不过，如果通过读取或者写入操作捕获了实例比那两，那么也会自动把self变量一并捕获了，因为实例变量是与self所指代的实例关联在一起的。

**全局块**

定义块的时候，占的内存区域是分配在*栈*中的

给块发送copy消息拷贝，这样就可以把块从栈复制到堆了。

全局块不会捕捉任何状态，运行时也无须有状态来参与。

**为常用的块类型创建typedef**

**用handler块降低代码分散程度**

**使用block块引用所属对象不要出现引用循环**

**多用派发队列，少用同步锁**

@synchronized(self)根据给定的对象，自动创建一个锁，并等待块中的代码执行完毕

NSLock

不过最好使用 GCD，它能更简单，更搞笑的形式为代码加锁

使用**串行同步队列**，将读取操作以及写入操作都安排在同一个队列里，可保证数据同步

\_syncQueue = dispatch\_queue\_create("com.xx", NULL);

dispatch\_sync(\_syncQueue, ^{

\_someString = someString;

});

把设置和获取操作都安排在序列化的队列里执行，这样的花所有针对属性的访问操作都同步了

dispatch\_barrier\_async(dispatch\_queue\_t queue,disaptch\_block\_t block);在barrier中必须单独执行，不能与其他块并行

**多使用 GCD，少用performSelector系列方法**

延后执行某个任务

dispatch\_after(dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)(3.0 \* NSEC\_PER\_SEC)), dispatch\_get\_main\_queue(), ^{

[self doSomething];

});

想把任务放在主线程上执行

dispatch\_async(dispatch\_get\_main\_queue(), ^{

[self doSomething];

});

**掌握 GCD 及操作队列的使用时机**

在执行后台任务时，GCD 并不一定是最佳方式，还有一种技术叫做NSOperationQueue

比如，从服务器端下载并处理文件的动作，可以用操作来表示，而在处理其他文件之前，必须先下载清单文件，后续的下载操作，都要依赖于先下载清单文件这一操作。如果操作队列允许并发的话，那么后续的多个下载操作就可以同时执行，但前提是它们所依赖的那个清单文件下载操作已经执行完毕

NSOperation对象有许多属性都适合 KVO 来监听，isCancelled来判断任务是否取消，或者通过isFinished来判断任务是否完成。

NSOperation对象也有线程优先级

**通过dispatch group机制，根据系统资源状况来执行任务**

**使用dispatch\_once来执行只需一次的线程安全代码**

单例使用

**不要使用dispatch\_get\_current\_queue**

**熟悉系统框架**

CFNetwork，此框架提供了 C 语言级别的网络通信能力，它将 BSD 抽象成易于使用的网络接口。而 Foundation 则将该框架李的部分内容封装为 OC 的接口

CoreAudio，此框架提供的 C 语言 API 可以用来操作设备商的音频硬件

AVFoundation，用来回访并录制音频及视频

CoreData，将对象放入数据库

CoreText，可以高效执行文字排版及渲染操作

**多用块枚举，少用 for 循环**

**构建缓存时选用NSCache而非NSDictionary**

**精简initialize与load的实现代码**

+ (void)load，只调用一次。

+ (void)initialize，该方法在程序首次使用该类之前调用，且只调用一次

**NSTimer会保留其目标对象**

NSTimer很容易出现引用循环

（未完待续）

**总结**

纯属个人笔记，特别是底层机制很有作用，如今iOS开发不再仅仅是把一个内容展现出来，里面还有涉及到各种安全性能，了解根本才是持续发展之道。

# 选择github

https://github.com/raywenderlich/objective-c-style-guide#the-official-raywenderlichcom-objective-c-style-guide

**The official raywenderlich.com Objective-C style guide.**

This style guide outlines the coding conventions for raywenderlich.com.

**Introduction**

The reason we made this style guide was so that we could keep the code in our books, tutorials, and starter kits nice and consistent - even though we have many different authors working on the books.

This style guide is different from other Objective-C style guides you may see, because the focus is centered on readability for print and the web. Many of the decisions were made with an eye toward conserving space for print, easy legibility, and tutorial writing.

**Credits**

The creation of this style guide was a collaborative effort from various raywenderlich.com team members under the direction of Nicholas Waynik. The team includes: [Soheil Moayedi Azarpour](https://github.com/moayes), [Ricardo Rendon Cepeda](https://github.com/ricardo-rendoncepeda), [Tony Dahbura](https://github.com/tdahbura), [Colin Eberhardt](https://github.com/ColinEberhardt), [Matt Galloway](https://github.com/mattjgalloway), [Greg Heo](https://github.com/gregheo), [Matthijs Hollemans](https://github.com/hollance), [Christopher LaPollo](https://github.com/elephantronic), [Saul Mora](https://github.com/casademora), [Andy Pereira](https://github.com/macandyp), [Mic Pringle](https://github.com/micpringle), [Pietro Rea](https://github.com/pietrorea), [Cesare Rocchi](https://github.com/funkyboy), [Marin Todorov](https://github.com/icanzilb), [Nicholas Waynik](https://github.com/ndubbs), and [Ray Wenderlich](https://github.com/raywenderlich)

We would like to thank the creators of the [New York Times](https://github.com/NYTimes/objective-c-style-guide) and [Robots & Pencils'](https://github.com/RobotsAndPencils/objective-c-style-guide) Objective-C Style Guides. These two style guides provided a solid starting point for this guide to be created and based upon.

**Background**

Here are some of the documents from Apple that informed the style guide. If something isn't mentioned here, it's probably covered in great detail in one of these:

* [The Objective-C Programming Language](http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html)
* [Cocoa Fundamentals Guide](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html)
* [Coding Guidelines for Cocoa](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html)
* [iOS App Programming Guide](http://developer.apple.com/library/ios/#documentation/iphone/conceptual/iphoneosprogrammingguide/Introduction/Introduction.html)

**Table of Contents**

* Language
* Code Organization
* Spacing
* Comments
* Naming
  1. Underscores
* Methods
* Variables
* Property Attributes
* Dot-Notation Syntax
* Literals
* Constants
* Enumerated Types
* Case Statements
* Private Properties
* Booleans
* Conditionals
  1. Ternary Operator
* Init Methods
* Class Constructor Methods
* CGRect Functions
* Golden Path
* Error handling
* Singletons
* Line Breaks
* Smiley Face
* Xcode Project

**Language**

US English should be used.

**Preferred:**

UIColor \*myColor = [UIColor whiteColor];

**Not Preferred:**

UIColor \*myColour = [UIColor whiteColor];

**Code Organization**

Use #pragma mark - to categorize methods in functional groupings and protocol/delegate implementations following this general structure.

#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

- (IBAction)submitData:(id)sender {}

#pragma mark - Public

- (void)publicMethod {}

#pragma mark - Private

- (void)privateMethod {}

#pragma mark - Protocol conformance

#pragma mark - UITextFieldDelegate

#pragma mark - UITableViewDataSource

#pragma mark - UITableViewDelegate

#pragma mark - NSCopying

- (id)copyWithZone:(NSZone \*)zone {}

#pragma mark - NSObject

- (NSString \*)description {}

**Spacing**

* Indent using 2 spaces (this conserves space in print and makes line wrapping less likely). Never indent with tabs. Be sure to set this preference in Xcode.
* Method braces and other braces (if/else/switch/while etc.) always open on the same line as the statement but close on a new line.

**Preferred:**

if (user.isHappy) {

//Do something

} else {

//Do something else

}

**Not Preferred:**

if (user.isHappy)

{

//Do something

}

else {

//Do something else

}

* There should be exactly one blank line between methods to aid in visual clarity and organization. Whitespace within methods should separate functionality, but often there should probably be new methods.
* Prefer using auto-synthesis. But if necessary, @synthesize and @dynamic should each be declared on new lines in the implementation.
* Colon-aligning method invocation should often be avoided. There are cases where a method signature may have >= 3 colons and colon-aligning makes the code more readable. Please do **NOT** however colon align methods containing blocks because Xcode's indenting makes it illegible.

**Preferred:**

// blocks are easily readable

[UIView animateWithDuration:1.0 animations:^{

// something

} completion:^(BOOL finished) {

// something

}];

**Not Preferred:**

// colon-aligning makes the block indentation hard to read

[UIView animateWithDuration:1.0

animations:^{

// something

}

completion:^(BOOL finished) {

// something

}];

**Comments**

When they are needed, comments should be used to explain **why** a particular piece of code does something. Any comments that are used must be kept up-to-date or deleted.

Block comments should generally be avoided, as code should be as self-documenting as possible, with only the need for intermittent, few-line explanations. *Exception: This does not apply to those comments used to generate documentation.*

**Naming**

Apple naming conventions should be adhered to wherever possible, especially those related to [memory management rules](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html) ([NARC](http://stackoverflow.com/a/2865194/340508)).

Long, descriptive method and variable names are good.

**Preferred:**

UIButton \*settingsButton;

**Not Preferred:**

UIButton \*setBut;

A three letter prefix should always be used for class names and constants, however may be omitted for Core Data entity names. For any official raywenderlich.com books, starter kits, or tutorials, the prefix 'RWT' should be used.

Constants should be camel-case with all words capitalized and prefixed by the related class name for clarity.

**Preferred:**

static NSTimeInterval const RWTTutorialViewControllerNavigationFadeAnimationDuration = 0.3;

**Not Preferred:**

static NSTimeInterval const fadetime = 1.7;

Properties should be camel-case with the leading word being lowercase. Use auto-synthesis for properties rather than manual @synthesize statements unless you have good reason.

**Preferred:**

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

**Not Preferred:**

id varnm;

**Underscores**

When using properties, instance variables should always be accessed and mutated using self.. This means that all properties will be visually distinct, as they will all be prefaced with self..

An exception to this: inside initializers, the backing instance variable (i.e. \_variableName) should be used directly to avoid any potential side effects of the getters/setters.

Local variables should not contain underscores.

**Methods**

In method signatures, there should be a space after the method type (-/+ symbol). There should be a space between the method segments (matching Apple's style). Always include a keyword and be descriptive with the word before the argument which describes the argument.

The usage of the word "and" is reserved. It should not be used for multiple parameters as illustrated in the initWithWidth:height: example below.

**Preferred:**

- (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;

**Not Preferred:**

-(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.

**Variables**

Variables should be named as descriptively as possible. Single letter variable names should be avoided except in for() loops.

Asterisks indicating pointers belong with the variable, e.g., NSString \*text not NSString\* text or NSString \* text, except in the case of constants.

Private properties should be used in place of instance variables whenever possible. Although using instance variables is a valid way of doing things, by agreeing to prefer properties our code will be more consistent.

Direct access to instance variables that 'back' properties should be avoided except in initializer methods (init, initWithCoder:, etc…), dealloc methods and within custom setters and getters. For more information on using Accessor Methods in Initializer Methods and dealloc, see [here](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-SW6).

**Preferred:**

@interface RWTTutorial : NSObject

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

@end

**Not Preferred:**

@interface RWTTutorial : NSObject {

NSString \*tutorialName;

}

**Property Attributes**

Property attributes should be explicitly listed, and will help new programmers when reading the code. The order of properties should be storage then atomicity, which is consistent with automatically generated code when connecting UI elements from Interface Builder.

**Preferred:**

@property (weak, nonatomic) IBOutlet UIView \*containerView;

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

**Not Preferred:**

@property (nonatomic, weak) IBOutlet UIView \*containerView;

@property (nonatomic) NSString \*tutorialName;

Properties with mutable counterparts (e.g. NSString) should prefer copy instead of strong. Why? Even if you declared a property as NSString somebody might pass in an instance of an NSMutableString and then change it without you noticing that.

**Preferred:**

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

**Not Preferred:**

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

**Dot-Notation Syntax**

Dot syntax is purely a convenient wrapper around accessor method calls. When you use dot syntax, the property is still accessed or changed using getter and setter methods. Read more [here](https://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html)

Dot-notation should **always** be used for accessing and mutating properties, as it makes code more concise. Bracket notation is preferred in all other instances.

**Preferred:**

NSInteger arrayCount = [self.array count];

view.backgroundColor = [UIColor orangeColor];

[UIApplication sharedApplication].delegate;

**Not Preferred:**

NSInteger arrayCount = self.array.count;

[view setBackgroundColor:[UIColor orangeColor]];

UIApplication.sharedApplication.delegate;

**Literals**

NSString, NSDictionary, NSArray, and NSNumber literals should be used whenever creating immutable instances of those objects. Pay special care that nil values can not be passed into NSArray and NSDictionary literals, as this will cause a crash.

**Preferred:**

NSArray \*names = @[@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul"];

NSDictionary \*productManagers = @{@"iPhone": @"Kate", @"iPad": @"Kamal", @"Mobile Web": @"Bill"};

NSNumber \*shouldUseLiterals = @YES;

NSNumber \*buildingStreetNumber = @10018;

**Not Preferred:**

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];

**Constants**

Constants are preferred over in-line string literals or numbers, as they allow for easy reproduction of commonly used variables and can be quickly changed without the need for find and replace. Constants should be declared as static constants and not #defines unless explicitly being used as a macro.

**Preferred:**

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

static CGFloat const RWTImageThumbnailHeight = 50.0;

**Not Preferred:**

#define CompanyName @"RayWenderlich.com"

#define thumbnailHeight 2

**Enumerated Types**

When using enums, it is recommended to use the new fixed underlying type specification because it has stronger type checking and code completion. The SDK now includes a macro to facilitate and encourage use of fixed underlying types: NS\_ENUM()

**For Example:**

typedef NS\_ENUM(NSInteger, RWTLeftMenuTopItemType) {

RWTLeftMenuTopItemMain,

RWTLeftMenuTopItemShows,

RWTLeftMenuTopItemSchedule

};

You can also make explicit value assignments (showing older k-style constant definition):

typedef NS\_ENUM(NSInteger, RWTGlobalConstants) {

RWTPinSizeMin = 1,

RWTPinSizeMax = 5,

RWTPinCountMin = 100,

RWTPinCountMax = 500,

};

Older k-style constant definitions should be **avoided** unless writing CoreFoundation C code (unlikely).

**Not Preferred:**

enum GlobalConstants {

kMaxPinSize = 5,

kMaxPinCount = 500,

};

**Case Statements**

Braces are not required for case statements, unless enforced by the complier.

When a case contains more than one line, braces should be added.

switch (condition) {

case 1:

// ...

break;

case 2: {

// ...

// Multi-line example using braces

break;

}

case 3:

// ...

break;

default:

// ...

break;

}

There are times when the same code can be used for multiple cases, and a fall-through should be used. A fall-through is the removal of the 'break' statement for a case thus allowing the flow of execution to pass to the next case value. A fall-through should be commented for coding clarity.

switch (condition) {

case 1:

// \*\* fall-through! \*\*

case 2:

// code executed for values 1 and 2

break;

default:

// ...

break;

}

When using an enumerated type for a switch, 'default' is not needed. For example:

RWTLeftMenuTopItemType menuType = RWTLeftMenuTopItemMain;

switch (menuType) {

case RWTLeftMenuTopItemMain:

// ...

break;

case RWTLeftMenuTopItemShows:

// ...

break;

case RWTLeftMenuTopItemSchedule:

// ...

break;

}

**Private Properties**

Private properties should be declared in class extensions (anonymous categories) in the implementation file of a class. Named categories (such as RWTPrivate or private) should never be used unless extending another class. The Anonymous category can be shared/exposed for testing using the +Private.h file naming convention.

**For Example:**

@interface RWTDetailViewController ()

@property (strong, nonatomic) GADBannerView \*googleAdView;

@property (strong, nonatomic) ADBannerView \*iAdView;

@property (strong, nonatomic) UIWebView \*adXWebView;

@end

**Booleans**

Objective-C uses YES and NO. Therefore true and false should only be used for CoreFoundation, C or C++ code. Since nil resolves to NO it is unnecessary to compare it in conditions. Never compare something directly to YES, because YES is defined to 1 and a BOOL can be up to 8 bits.

This allows for more consistency across files and greater visual clarity.

**Preferred:**

if (someObject) {}

if (![anotherObject boolValue]) {}

**Not Preferred:**

if (someObject == nil) {}

if ([anotherObject boolValue] == NO) {}

if (isAwesome == YES) {} // Never do this.

if (isAwesome == true) {} // Never do this.

If the name of a BOOL property is expressed as an adjective, the property can omit the “is” prefix but specifies the conventional name for the get accessor, for example:

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

Text and example taken from the [Cocoa Naming Guidelines](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingIvarsAndTypes.html%23//apple_ref/doc/uid/20001284-BAJGIIJE).

**Conditionals**

Conditional bodies should always use braces even when a conditional body could be written without braces (e.g., it is one line only) to prevent errors. These errors include adding a second line and expecting it to be part of the if-statement. Another, [even more dangerous defect](http://programmers.stackexchange.com/a/16530) may happen where the line "inside" the if-statement is commented out, and the next line unwittingly becomes part of the if-statement. In addition, this style is more consistent with all other conditionals, and therefore more easily scannable.

**Preferred:**

if (!error) {

return success;

}

**Not Preferred:**

if (!error)

return success;

or

if (!error) return success;

**Ternary Operator**

The Ternary operator, ?: , should only be used when it increases clarity or code neatness. A single condition is usually all that should be evaluated. Evaluating multiple conditions is usually more understandable as an if statement, or refactored into instance variables. In general, the best use of the ternary operator is during assignment of a variable and deciding which value to use.

Non-boolean variables should be compared against something, and parentheses are added for improved readability. If the variable being compared is a boolean type, then no parentheses are needed.

**Preferred:**

NSInteger value = 5;

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

BOOL isHorizontal = YES;

result = isHorizontal ? x : y;

**Not Preferred:**

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

**Init Methods**

Init methods should follow the convention provided by Apple's generated code template. A return type of 'instancetype' should also be used instead of 'id'.

- (instancetype)init {

self = [super init];

if (self) {

// ...

}

return self;

}

See Class Constructor Methods for link to article on instancetype.

**Class Constructor Methods**

Where class constructor methods are used, these should always return type of 'instancetype' and never 'id'. This ensures the compiler correctly infers the result type.

@interface Airplane

+ (instancetype)airplaneWithType:(RWTAirplaneType)type;

@end

More information on instancetype can be found on [NSHipster.com](http://nshipster.com/instancetype/).

**CGRect Functions**

When accessing the x, y, width, or height of a CGRect, always use the [CGGeometry functions](http://developer.apple.com/library/ios/#documentation/graphicsimaging/reference/CGGeometry/Reference/reference.html) instead of direct struct member access. From Apple's CGGeometry reference:

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.

**Preferred:**

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);

**Not Preferred:**

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 };

**Golden Path**

When coding with conditionals, the left hand margin of the code should be the "golden" or "happy" path. That is, don't nest if statements. Multiple return statements are OK.

**Preferred:**

- (void)someMethod {

if (![someOther boolValue]) {

return;

}

//Do something important

}

**Not Preferred:**

- (void)someMethod {

if ([someOther boolValue]) {

//Do something important

}

}

**Error handling**

When methods return an error parameter by reference, switch on the returned value, not the error variable.

**Preferred:**

NSError \*error;

if (![self trySomethingWithError:&error]) {

// Handle Error

}

**Not Preferred:**

NSError \*error;

[self trySomethingWithError:&error];

if (error) {

// Handle Error

}

Some of Apple’s APIs write garbage values to the error parameter (if non-NULL) in successful cases, so switching on the error can cause false negatives (and subsequently crash).

**Singletons**

Singleton objects should use a thread-safe pattern for creating their shared instance.

+ (instancetype)sharedInstance {

static id sharedInstance = nil;

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

sharedInstance = [[self alloc] init];

});

return sharedInstance;

}

This will prevent [possible and sometimes prolific crashes](http://cocoasamurai.blogspot.com/2011/04/singletons-your-doing-them-wrong.html).

**Line Breaks**

Line breaks are an important topic since this style guide is focused for print and online readability.

For example:

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

A long line of code like this should be carried on to the second line adhering to this style guide's Spacing section (two spaces).

self.productsRequest = [[SKProductsRequest alloc]

initWithProductIdentifiers:productIdentifiers];

**Smiley Face**

Smiley faces are a very prominent style feature of the raywenderlich.com site! It is very important to have the correct smile signifying the immense amount of happiness and excitement for the coding topic. The end square bracket is used because it represents the largest smile able to be captured using ascii art. A half-hearted smile is represented if an end parenthesis is used, and thus not preferred.

**Preferred:**

:]

**Not Preferred:**

:)

**Xcode project**

The physical files should be kept in sync with the Xcode project files in order to avoid file sprawl. Any Xcode groups created should be reflected by folders in the filesystem. Code should be grouped not only by type, but also by feature for greater clarity.

When possible, always turn on "Treat Warnings as Errors" in the target's Build Settings and enable as many [additional warnings](http://boredzo.org/blog/archives/2009-11-07/warnings) as possible. If you need to ignore a specific warning, use [Clang's pragma feature](http://clang.llvm.org/docs/UsersManual.html#controlling-diagnostics-via-pragmas).

**Other Objective-C Style Guides**

If ours doesn't fit your tastes, have a look at some other style guides:

* [Robots & Pencils](https://github.com/RobotsAndPencils/objective-c-style-guide)
* [New York Times](https://github.com/NYTimes/objective-c-style-guide)
* [Google](http://google-styleguide.googlecode.com/svn/trunk/objcguide.xml)
* [GitHub](https://github.com/github/objective-c-conventions)
* [Adium](https://trac.adium.im/wiki/CodingStyle)
* [Sam Soffes](https://gist.github.com/soffes/812796)
* [CocoaDevCentral](http://cocoadevcentral.com/articles/000082.php)
* [Luke Redpath](http://lukeredpath.co.uk/blog/my-objective-c-style-guide.html)

[Marcus Zarra](http://www.cimgf.com/zds-code-style-guide/)

# 选择博客2

[**iPhone开发：Objective C 代码规范－iOS总结版**](http://www.cnblogs.com/pengyingh/articles/2405693.html)

一，关于空行

A：.h中的空行

1，文件说明与头文件包涵（#import）之间空1行

2，头文件包涵（#import）之间，如果需要分类区别，各类别之间空1行

3，头文件包涵（#import）与@class之间空2行

4，@interface与@class之间空一行

5，头文件｛｝里面，空1行开始声明对象成员，如果需要分类区别，各类别之间空1行

6，｛｝外空1行，书写属性，如果需要分类区别，各类别之间空1行

7，空1行开始写方法，如果需要分类区别，各类别之间空1行

8，方法完成后，空1行@end

9，如果需要声明protocal，空2行接着写。通常protocal写在@end后面，但是声明在@interface前面，也可以另写一个文件

B：.m中的空行

1，文件说明与头文件包涵（#import）之间空1行

2，头文件包涵（#import）之间，如果需要分类区别，各类别之间空1行

3，@implementation和@synthesize之间空一行，@synthesize不要使用逗号（，）如果需要分类区别，各类别之间空1行

4，@synthesize与方法之间空2行

5，各方法之间空2行

C：方法里面的空行

1，方法名后空1行开始写

2，变量声明后需要空1行，如果需要分类区别，各类别之间空1行

3，条件，循环，选择语句，整个语句结束，需要空1行

4，各功能块之间空1行

5，最后一个反括号之前不空行

6，注释与代码之间不空行

7，#pragma mark与方法之间空1行

二，关于空格

1，.h中协议<>前面有1个空格

2，.h中成员声明时，类型与变量之间有至少1各空格。星号（＊）靠近变量，不靠近类型。（部分习惯，所有变量可以以＊对齐，中间留空）

3，@property后有1各空格，（）里面，逗号后有1个空格，括号外，先留1个空格，再声明属性

4，方法＋，－后，与（）之间有1个空格

5，返回类型与＊之间有1个空格，方法参数中返回类型与＊之间有1个空格

6，多参数的方法，每一个参数后面都有1个空格

三，关于布尔值

1，不要用if(obj==nil){}，而用if(!obj){}

2，比较时把常量放前面可以避免错误

不要用if(aIntValue==255){}，而用if(255==aIntValue){}，避免漏掉一个“＝”，而变成赋值

3，不要用if(aBool==YES){}，直接用if(aBool){}或if(!!aBool){}

四，变量属性名和其它

1，尽量在使用的时候才声明变量，尽量少用全局变量

2，变量名不要只使用一个字母，尽量能表示变量的意思

3，@synthesize和@dynamic，应该放在类实现的最上面，每一个声明都单独一行

4，协议protocls需要加#pragam marks-来区分

5，释放时应该[obj release],obj = nil; delegate也应该在dealloc里面delegate ＝ nil；