移动团队Objective-C 规范指南

目录

点语法

应该 **始终** 使用点语法来访问或者修改属性，访问其他实例时首选括号。

推荐：

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

反对：

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

条件判断

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 语句的一部分时。还有另外一种更危险的，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。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;
}
```

反对：

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

或

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

三目运算符

三目运算符，？，只有当它可以增加代码清晰度或整洁时才使用。单一的条件都应该优先考虑使用。多条件时通常使用 if 语句会更易懂，或者重构为实例变量。

推荐：

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

反对：

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

方法

在方法签名中，在 -/+ 符号后应该有一个空格。方法片段之间也应该有一个空格。

推荐：

```
- (void)setExampleText:(NSString *)text image:(UIImage *)image;
```

变量

变量名应该尽可能命名为描述性的。除了 `for()` 循环外，其他情况都应该避免使用单字母的变量名。星号表示指针属于变量，例如：`NSString *text` 不要写成 `NSString* text` 或者 `NSString * text`，常量除外。尽量定义属性来代替直接使用实例变量。除了初始化方法（`init`，`initWithCoder:`，等），`dealloc` 方法和自定义的 setters 和 getters 内部，应避免直接访问实例变量。更多有关在初始化方法和 dealloc 方法中使用访问器方法的信息，参见这里。

推荐：

```
@interface NYTSection : NSObject

@property (nonatomic) NSString *headline;

@end
```

反对：

```
@interface NYTSection : NSObject {
    NSString *headline;
}
```

变量限定符

当涉及到在 ARC 中被引入变量限定符时， 限定符 (`__strong`, `__weak`, `__unsafe_unretained`, `__autoreleasing`) 应该位于星号和变量名之间，如：`NSString * __weak text`。

命名

尽可能遵守苹果的命名约定，尤其那些涉及到内存管理规则，（NARC）的。

长的和描述性的方法名和变量名都不错。

推荐：

```
UIButton *settingsButton;
```

反对：

```
UIButton *setBut;
```

类名和常量应该始终使用三个字母的前缀（例如 `NYT`），但 Core Data 实体名称可以省略。为了代码清晰，常量应该使用相关类的名字作为前缀并使用驼峰命名法。

推荐：

```
static const NSTimeInterval
NYTArticleViewControllerNavigationFadeAnimationDuration = 0.3;
```

反对：

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

属性和局部变量应该使用驼峰命名法并且首字母小写。

为了保持一致，实例变量应该使用驼峰命名法命名，并且首字母小写，以下划线为前缀。这与 LLVM 自动合成的实例变量相一致。 **如果 LLVM 可以自动合成变量，那就让它自动合成。**

推荐：

```
@synthesize descriptiveVariableName = _descriptiveVariableName;
```

反对：

```
id varnm;
```

注释

当需要的时候，注释应该被用来解释 **为什么** 特定代码做了某些事情。所使用的任何注释必须保持最新否则就删除掉。

通常应该避免一大块注释，代码就应该尽量作为自身的文档，只需要隔几行写几句说明。这并不适用于那些用来生成文档的注释。

init 和 dealloc

`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。

`init` 方法的结构应该像这样：

```
- (instancetype)init {
    self = [super init]; // 或者调用指定的初始化方法
    if (self) {
        // Custom initialization
    }
    return self;
}
```

字面量

每当创建 `NSString`， `NSDictionary`， `NSArray`，和 `NSNumber` 类的不可变实例时，都应该使用字面量。要注意 `nil` 值不能传给 `NSArray` 和 `NSDictionary` 字面量，这样做会导致崩溃。

推荐：

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

反对：

```
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 *buildingZIPCode = [NSNumber numberWithInteger:10018];
```

CGRect 函数

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.

推荐：

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

CGFloat x = CGRectGetMinX(frame);
CGFloat y = CGRectGetMinY(frame);
CGFloat width = CGRectGetWidth(frame);
CGFloat height = CGRectGetHeight(frame);
```

反对：

```
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;
```

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 NYTAboutViewControllerCompanyName = @"The New York Times Company";

static const CGFloat NYTImageThumbnailHeight = 50.0;
```

反对：

```
#define CompanyName @"The New York Times Company"

#define thumbnailHeight 2
```

枚举类型

当使用 `enum` 时，建议使用新的基础类型规范，因为它具有更强的类型检查和代码补全功能。现在 SDK 包含了一个宏来鼓励使用使用新的基础类型 - `NS_ENUM()`

推荐：

```
typedef NS_ENUM(NSInteger, NYTAdRequestState) {
    NYTAdRequestStateInactive,
    NYTAdRequestStateLoading
};
```

位掩码

当用到位掩码时，使用 `NS_OPTIONS` 宏。

举例：

```
typedef NS_OPTIONS(NSUInteger, NYTAdCategory) {
    NYTAdCategoryAutos      = 1 << 0,
    NYTAdCategoryJobs       = 1 << 1,
    NYTAdCategoryRealState  = 1 << 2,
    NYTAdCategoryTechnology = 1 << 3
};
```

为什么使用位掩码的形式来定义枚举值？可以参考这篇帖子。

私有属性

私有属性应该声明在类实现文件的延展（匿名的类目）中。

推荐：

```
@interface NYTAdvertisement ()

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

@end
```

图片命名

图片名称应该被统一命名以保持组织的完整。它们应该被命名为一个说明它们用途的驼峰式字符串，其次是自定义类或属性的无前缀名字（如果有的话），然后进一步说明颜色 和/或 展示位置，最后是它们的状态。

推荐：

- `RefreshBarButtonItem` / `RefreshBarButtonItem@2x` 和 `RefreshBarButtonItemSelected` / `RefreshBarButtonItemSelected@2x`
- `ArticleNavigationBarWhite` / `ArticleNavigationBarWhite@2x` 和 `ArticleNavigationBarBlackSelected` / `ArticleNavigationBarBlackSelected@2x`.

图片目录中被用于类似目的的图片应归入各自的组中。

布尔

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` 进行比较，因为 `YES` 被定义为 1，而 `BOOL` 可以多达 8 位。

这使得整个文件有更多的一致性和更大的视觉清晰度。

推荐：

```
if (!someObject) {
}
```

反对：

```
if (someObject == nil) {
}
```

对于 `BOOL` 来说，这有两种用法：

```
if (isAwesome)
if (![someObject boolValue])
```

反对：

```
if ([someObject boolValue] == NO)
if (isAwesome == YES) // 永远别这么做
```

如果一个 `BOOL` 属性名称是一个形容词，属性可以省略 “is” 前缀，但为 get 访问器指定一个惯用的名字，例如：

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

内容和例子来自 Cocoa 命名指南 。

单例

单例对象应该使用线程安全的模式创建共享的实例。

```
+ (instancetype)sharedInstance {
    static id sharedInstance = nil;

    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });

    return sharedInstance;
}
```

这将会预防有时可能产生的许多崩溃。

导入

如果有一个以上的 import 语句，就对这些语句进行分组。每个分组的注释是可选的。
注：对于模块使用 @import 语法。

```
// Frameworks
@import QuartzCore;

// Models
#import "NYTUser.h"

// Views
#import "NYTButton.h"
#import "NYTUserView.h"
```

黄金路径

对于明确条件可以尽早结束函数返回的，放在函数最前面写。

- 推荐写法

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

  //Do something important
}
```

- 不推荐的写法

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

页面绘制方式

为了便于通过代码管理工具进行版本管理，减少冲突出现的几率，页面不要使用storyboard或xib进行生成，统一view层采用代码进行绘制。

代码结构的规定

1、页面的绘制通过单独定义的view文件来实现，controller通过引入view文件完成页面的展示。

2、使用`#pragma mark`来分类方法，参考以下结构

```
#pragma mark — Life Cycle

#pragma mark - UITextFieldDelegate

#pragma mark - UITableViewDataSource

#pragma mark - UITableViewDelegate

#pragma mark - Custom Delegates

#pragma mark - Events

#pragma mark — Private Methods

#pragma mark — Getters and Setters
```

Controller中的代码组织顺序按照上面的分类进行。