

iOS开发编码规范及注意事项

标题含有(建议, 不是必需)时, 不是必须遵守。

语言

应该使用US英语。保持每个名称使用正确的英文单词, 避免使用单词缩写。
应该:

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

不应该:

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

代码组织

在函数分组和 `protocol / delegate` 实现中使用 `#pragma mark -` 来分类方法, 要遵循以下一般结构:

```
#pragma mark - Lifecycle
- (instancetype)init {}
- (void)dealloc {}
- (void)viewDidLoad {}
- (void)viewWillAppear:(BOOL)animated {}
- (void)didReceiveMemoryWarning {}
#pragma mark - UITextFieldDelegate
#pragma mark - UITableViewDataSource
#pragma mark - UITableViewDelegate
#pragma mark - Custom Protocol Impl
#pragma mark - IBActions & Action
- (IBAction)submitData:(id)sender {}
#pragma mark - Public
- (void)publicMethod {}
#pragma mark - Private
- (void)privateMethod {}
```

```
#pragma mark - setter & getter
- (void)setCustomProperty:(id)value {}
- (id)customProperty {}
#pragma mark - NSCopying
- (id)copyWithZone:(NSZone *)zone {}
#pragma mark - NSObject
- (NSString *)description {}
```

行宽

尽量让你的代码保持在80列之内。

我们深知Objective-C是一门繁冗的语言，在某些情况下略超80列可能有助于提高可读性，但这也只能是特例而已，不能成为开脱。

如果阅读代码的人认为把某行行宽保持在80列仍然有不失可读性，你应该按他们说的去做。

我们意识到这条规则是有争议的，但很多已经存在的代码坚持了本规则，我们觉得保证一致性更重要。

通过设置 `Xcode > Preferences > Text Editing > Show page guide`，来使越界更容易被发现。

方法声明和定义

+、-和返回类型之间须使用一个空格，参数列表中只有参数之间可以有空格。

方法应该像这样：

```
- (void)doSomethingWithString:(NSString *)theString
{
    // do Something
}
```

*前的空格是可选的。当写新的代码时，要与先前代码保持一致。

方法调用

方法调用应尽量保持与方法声明的格式一致。当格式的风格有多种选择时，新的代码要与已有代码保持一致。

调用时所有参数应该在同一行：

```
[myObject doFooWith:arg1 name:arg2 error:arg3];
```

不要使用下面的缩进风格:

```
[myObject doFooWith:arg1 name:arg2 // some lines with >1 arg
                                error:arg3];
[myObject doFooWith:arg1
                                name:arg2 error:arg3];
[myObject doFooWith:arg1
    name:arg2 // aligning keywords instead of colons
    error:arg3];
```

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网上的可读性。
例如：

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

一行很长的代码应该分成两行代码，下一行用两个空格隔开。

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

空格

- 缩进使用4个空格(1个Tab键)，确保在Xcode偏好设置来设置。
- 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：

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



```
}
```

不应该:

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

- 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，但通常都抽离出来成为一个新方法。
- 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。
- 应该避免以冒号对齐的方式来调用方法。因为有时方法签名可能有3个以上的冒号和冒号对齐会使代码更加易读。请不要这样做，尽管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。

应该:

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

不应该:

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

异常

每个@标签应该有独立的一行，在@与{}之间需要有一个空格，@catch与被捕捉到的异常对象的声明之间也要有一个空格。

如果你决定使用Objective-C的异常，那么就按下面的格式。不过你最好先了解下为什么不要使用异常。

```
@try {  
    foo();  
}  
@catch (NSException *ex) {  
    bar(ex);  
}  
@finally {  
    baz();  
}
```

注意：建议不要随意抛出异常，除非必要场景。

块（闭包）

块（block）适合用在target/selector模式下创建回调方法时，因为它使代码更易读。块中的代码应该缩进1个Tab。

取决于块的长度，下列都是合理的风格准则：

- 如果一行可以写完块，则没必要换行。
- 如果不得不换行，关括号应与块声明的第一个字符对齐。
- 块内的代码须按Tab缩进。
- 如果块太长，比如超过20行，建议把它定义成一个局部变量，然后再使用该变量。
- 如果块不带参数，^{之间无须空格。如果带有参数，^(之间无须空格，但){之间须有一个空格。
- 块内允许按两个空格缩进，但前提是和项目的其它代码保持一致的缩进风格。

```
// The entire block fits on one line.  
[operation setCompletionBlock:^( [self onOperationDone]; ]];  
  
// The block can be put on a new line, indented four spaces, with the  
// closing brace aligned with the first character of the line on which  
// block was declared.  
[operation setCompletionBlock:^(  
    [self.delegate newDataAvailable];  
)];
```

```

// Using a block with a C API follows the same alignment and spacing
// rules as with Objective-C.
dispatch_async(fileIOQueue_, ^{
    NSString *path = [self sessionFilePath];
    if (path) {
        // ...
    }
});

// An example where the parameter wraps and the block declaration fits
// on the same line. Note the spacing of |^(SessionWindow -window) {|
// compared to |^{| above.
[[SessionService sharedService] loadWindowWithCompletionBlock:^(Session
Window *window) {
    if (window) {
        [self windowDidLoad:window];
    } else {
        [self errorLoadingWindow];
    }
}];

// An example where the parameter wraps and the block declaration does
// not fit on the same line as the name.
[[SessionService sharedService] loadWindowWithCompletionBlock:^(Session
Window *window) {
    if (window) {
        [self windowDidLoad:window];
    } else {
        [self errorLoadingWindow];
    }
}];

// Large blocks can be declared out-of-line.
void (^largeBlock)(void) = ^{
    // ...
};
[operationQueue_ addOperationWithBlock:largeBlock];

```

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，因为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。

可以使用[VVDocumenter](#)插件生成注释。

命名

对于易维护的代码而言，命名规则非常重要。Objective-C 的方法名往往十分长，但代码块读起来就像散文一样，不需要太多的代码注释。

Apple命名规则尽可能坚持，特别是与这些相关的memory management rules (MRC/ARC)。

Apple风格使用驼峰命名法，这在 Objective-C 社区中非常普遍。

任何的类、类别、方法以及变量的名字中都使用全大写的**首字母缩写**

<http://en.wikipedia.org/wiki/Initialism>。这遵守了苹果的标准命名方式，如 URL、TIFF 以及 EXIF。

长的，描述性的方法和变量命名是好的。

应该：

```
UIButton *settingsButton;
```

不应该：

```
UIButton *setBut;
```

协议名

类型标识符和尖括号内的协议名之间，不能有任何空格。

这条规则适用于类声明、实例变量以及方法声明。例如：

```
@interface MyProtocoledClass : NSObject<NSWindowDelegate> {  
    @private  
    id<MyFancyDelegate> _delegate;  
}  
- (void)setDelegate:(id<MyFancyDelegate>)aDelegate;  
@end
```

类名

类名应该有两三个字母的前缀以表示类别是项目的一部分或者该类别是通用的。

类名（以及类别、协议名）应首字母大写，并以驼峰格式分割单词。

类别名

类别名应该有两三个字母的前缀以表示类别是项目的一部分或者该类别是通用的。类别名应该包含它所扩展的类的名字。

方法名

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。

方法名应尽量读起来就像句子，这表示你应该选择与方法名连在一起读起来通顺的参数名。（例如：`convertPoint:fromRect:` 或 `replaceCharactersInRange:withString:`）。

详情参见 Apple's Guide to Naming Methods

<http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingMethods.html>。

访问器方法应该与他们要获取的成员变量的名字一样，但不应该以`get`作为前缀。例如：

```
- (id)getDelegate; // AVOID
- (id)delegate;    // GOOD
```

变量名

变量名应该以小写字母开头，并使用驼峰格式。类的成员变量应该以下划线作为前缀。例如：`_myInstanceVariable`。

如果不能使用 Objective-C 2.0 的 `@property`，使用 KVO/KVC 绑定的成员变量可以以一个下划线作为前缀。

宏

- 全部使用大写字母
- 单词之间使用下划线分隔
- 宏名称前缀使用项目前缀

例如：AB_SCREEN_WIDTH

常用命名规范

- UIButton：变量名+Button
- UITextField：变量名+TextField
- UILabel：变量名+Label
- UITextView：变量名+TextView
- UIImageView：变量名+ImageView

- UIView: 变量名+View
- UITableViewCell/UICollectionViewController: 变量名+Cell
- UIViewController: 功能名+ViewController
- 协议(Protocol): 事件回调的协议使用 `Delegate` 结尾; 数据源的协议使用 `DataSource` 结尾; 其他的协议使用 `Protocol` 结尾
- ...

遵循以上命名规范即可。

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。对于官方的raywenderlich.com书、初学者工具包或教程, 前缀'RW'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。

应该:

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

不应该:

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

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:

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

不应该:

```
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来说明，就像

`initWithWidth:height` 以下这个例子：

应该：

```
- (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;
```

不应该：

```
-(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。

通过使用 `_variable`(变量名前面有下划线)直接访问实例变量应该尽量避免，除了在初始化方法(`init`，`initWithCoder:`，等...), `dealloc` 方法和自定义的 `setter` 和 `getter`。

应该：

```
@interface RWTTutorial : NSObject
@property (nonatomic, copy) NSString *tutorialName;
@end
```

不应该:

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

属性特性

所有属性特性应该显式地列出来，有助于新手阅读代码。属性特性的顺序应该是 `atomicity`、`storage`、`setter`、`getter`、`nullable` (`IBOutlet` 修饰除外)。

应该:

```
@property (weak, nonatomic) IBOutlet UIView *containerView;
@property (nonatomic, copy) NSString *tutorialName;
```

不应该:

```
@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic) NSString *tutorialName;
```

NSString应该使用copy而不是strong的属性特性。

为什么？即使你声明一个NSString的属性，有人可能传入一个NSMutableString的实例，然后在你没有注意的情况下修改它。

应该:

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

不应该:

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


点符号语法

点语法是一种很方便封装访问方法调用的方式。当你使用点语法时，通过使用getter或setter方法，属性仍然被访问或修改。

想了解更多，阅读：

<https://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html>。

点语法应该总是被用来访问和修改属性，因为它使代码更加简洁。[]符号更偏向于用在其他例子。

应该：

```
NSInteger arrayCount = [self.array count];
view.backgroundColor = [UIColor orangeColor];
[UIApplication sharedApplication].delegate;
```

不应该：

```
NSInteger arrayCount = self.array.count;
[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。

应该：

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

不应该：

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

常量

常量是容易重复被使用和无需通过查找和代替就能快速修改值。常量应该使用 `static` 来声明而不是使用 `#define`，除非显式地使用宏。
应该：

```
static NSString * const RWTAboutViewControllerCompanyName = @"RayWender
    lich.com";
static CGFloat const RWTImageThumbnailHeight = 50.0;
```

不应该：

```
#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()` 来帮助和鼓励你使用固定的基本类型。
例如：

```
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 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。
不应该：

```
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;
}
```

有很多次，当相同代码被多个case使用时，一个fall-through应该被使用。一个fall-through就是在case最后移除 `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（.m）中的类扩展(匿名分类)中声明，命名分类(比如 RWTPrivate或private)应该从不使用除非是扩展其他类。

例如：

```

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

```

布尔值

Objective-C使用 YES 和 NO。因为 true 和 false 应该只在CoreFoundation, C或C++代码使用。既然nil解析成NO, 所以没有必要在条件语句比较。不要拿某样东西直接与YES比较, 因为YES被定义为1和一个BOOL能被设置为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。

应该:

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

不应该:

```
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, 但要指定get访问器的惯用名称。例如:

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

文字和例子从这里引用Cocoa Naming Guidelines。

条件语句

条件语句主体为了防止出错应该使用大括号包围, 即使条件语句主体能够不用大括号编写(如, 只用一行代码)。这些错误包括添加第二行代码和期望它成为if语句。

还有, even more dangerous defect可能发生在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。除此之外, 这种风格与其他条件语句的风格保持一致, 所以更加容易阅读。

应该:

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

不应该:

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

或

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

三元操作符

当需要提高代码的清晰性和简洁性时，三元操作符 `?:` 才会使用。单个条件求值常常需要它。多个条件求值时，如果使用if语句或重构成实例变量时，代码会更加易读。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。

Non-boolean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。

应该:

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

不应该:

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

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;  
}
```

或

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

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(建议，不是必需)

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，在计算它们结果时隐式地标准化这些 `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。

应该：

```
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);
```

不应该:

```
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“golden”或“happy”路径。也就是不要嵌套if语句，多个返回语句也是OK。

应该:

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

不应该:

```
- (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。

应该:

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

不应该:

```
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non-NULL)，那么判断错误值会导致false负值和crash。

单例模式

单例对象应该使用线程安全模式来创建共享实例。

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

这会防止possible and sometimes prolific crashes。

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分组的创建应该在物理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。

Cocoa 和 Objective-C 特性

成员变量应该是 @private

```
@interface MyClass : NSObject {
    @private
    id _myInstanceVariable;
}
// public accessors, setter takes ownership
- (id)myInstanceVariable;
- (void)setMyInstanceVariable:(id)theVar;
@end
```

重载指定构造函数

当你写子类的时候，如果需要 `init...` 方法，记得重载父类的指定构造函数。如果你没有重载父类的指定构造函数，你的构造函数有时可能不会被调用，这会导致非常隐秘而且难以解决的bug。

初始化

不要在 `init` 方法中，将成员变量初始化为 `0` 或者 `nil`，毫无必要。刚分配的对象，默认值都是 `0`，除了 `isa` 指针（译者注：NSObject 的 `isa` 指针，用于标识对象的类型）。所以不要在初始化器里面写一堆将成员初始化为 `0` 或者 `nil` 的代码。

避免 +new

不要调用 `NSObject` 类方法 `new`，也不要子类中重载它。使用 `alloc` 和 `init` 方法创建并初始化对象。

现代的Objective-C代码通过调用 `alloc` 和 `init` 方法来创建并 `retain` 一个对象。由于类方法 `new` 很少使用，这使得有关内存分配的代码审查更困难。

保持公共API简单

保持类简单，避免“厨房水槽（kitchen-sink）”式的 API。如果一个函数压根没必要公开，就不要这么做。用私有类别保证公共头文件整洁。

Objective-C没有方法来区分公共的方法和私有的方法，-所有的方法都是公共的（译者注：这取决于Objective-C运行时的方法调用的消息机制）。因此，除非客户端的代码期望使用某个方法，不要把这个方法放进公共API中。尽可能的避免了你你不希望被调用的方法却被调用到。这包括重载父类的方法。对于内部实现所需要的方法，在实现的文件中定义一个类别，而不是把它们放进公有的头文件中。

```
#import "GTMFoo.h"
```

```

@interface GTMFoo ()
- (NSString *)doSomethingWithDelegate; // Declare private method
@end

@implementation GTMFoo()
...
- (NSString *)doSomethingWithDelegate {
    // Implement this method
}
...
@end

```

init和dealloc内避免使用访问器

在 `init` 和 `dealloc` 方法执行的过程中，子类可能会处在一个不一致的状态，所以这些方法中的代码应避免调用访问器。

子类尚未初始化，或在 `init` 和 `dealloc` 方法执行时已经被销毁，会使访问器方法很可能不可靠。实际上，应在这些方法中直接对 `ivars` 进行赋值或释放操作。

正确：

```

- (id)init
{
    if (self = [super init]) {
        _bar = [[NSMutableString alloc] init]; // good
    }
    return self;
}

- (void)dealloc
{
    [_bar release]; // good
    [super dealloc];
}

```

错误：

```

- (id)init {
    self = [super init];
    if (self) {
        self.bar = [NSMutableString string]; // avoid
    }
    return self;
}

- (void)dealloc {

```

```
self.bar = nil; // avoid
[super dealloc];
}
```

Nullability检测的支持

在swift语言中，通过!和?可以将对象声明成Optional，用于在开发中标记这个对象是否可以为空。在OC中，以前是没有这样的功能的，因此我们在开发中会经常遇到因为某个函数应该返回实例而返回了空导致的崩溃。Nullability的主要用武之地，就是在这里，它可以起到提示开发者做是否为空得判断的提示。

打开Xcode7，系统的框架中已经支持了Nullability，如下：

```
@property (nullable, nonatomic, readonly) ObjectType firstObject;
@property (nullable, nonatomic, readonly) ObjectType lastObject;
```

这是NSArray中的两个属性，其中nullable关键字说明了这里可能返回空的值。如果仅仅是在返回值中给开发者一些提示，你可能觉得应用并不大，是的，对开发者最大的帮助是这一特性可以用于函数的参数中，这样我们在调用函数时起到的提示作用，将是非常重要的，越是多人合作的项目，作用也越大。

例如：

```
-(void)setValue:(NSNumber * _Nonnull )number
{
}
```

我们在调用函数时，如果传入了空值，编译器会给我们警告：

A screenshot of an Xcode warning message. On the left, there is a yellow warning icon followed by the text '22' and '22' on separate lines. In the center, the code snippet '[self setValue:nil];' is shown with 'nil' highlighted in red. On the right, the warning text reads 'Null passed to a callee that requires a non-null argument'.

注意：

这一特性在Xcode6.3中就已经支持，但在Xcode7中又做了一些写法上的小改动，例如，在Xcode6.3中这样写：

```
-(void)setValue:(nonnull NSNumber * )number
{
}
```

而在Xcode7中提倡我们使用第一种写法。与之相关的几个关键字如下：

修饰参数

nonnull: 不可为空

nullable: 可以为空

null_unspecified: 不确定是否可以为空(极少情况)

在属性的声明中，还会有如下一个修饰符：

null_resettable: set方法可以为nil，get方法不可返回nil

一点提示：

你可以发现，iOS9的SDK中已经完全兼容使用了这些特性，并且**nonnull**的使用会比**nullable**广泛的多，因此，系统提供了这样一对宏：

```
#define NS_ASSUME_NONNULL_BEGIN _Pragma("clang assume_nonnull begin")

#define NS_ASSUME_NONNULL_END _Pragma("clang assume_nonnull end")
```

我们在这对宏之间定义的变量都会加上nonnull的修饰符，只有我们特殊声明nullable的才需要手动写。

泛型集合的支持

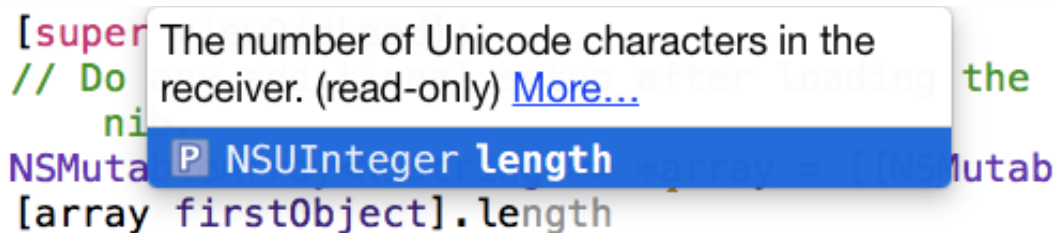
这一特性只作用于编译期，是为我们开发者服务的另一重要特性。在使用 `NSArray`、`NSDictionary` 时，集合中的元素尽量使用类型约定。这样编译器可以推断出集合中元素的类型，避免bug。

1、有类型约定的集合。

例如：

```
NSArray<NSString*> *list = @[@"A"];
```

声明了这样一个数组后，就好比告诉了编译器，这个数组中的数据类型都是 `NSString*` 类型的，现在非常好，如果我这个数组中元素的方法，会出现如下的提示：



The number of Unicode characters in the receiver. (read-only) [More...](#)

P NSInteger length

[array firstObject].length

激动吧，使用点语法可以访问到数组中泛型的方法了，还有更加诱人的：

```
array addObject:(nonnull NSString *)
}
M void addObject:(nonnull NSString *)
M void addObjectsFromArray:(nonnull NSArray<NSString *> *)
```

在我们向这个数组中追加元素的时候，编译器将元素的类型提示了出来，并且将FromArray方法中需要的元素类型也提示了出来。

同样，如果我们向这个数组中追加类型不匹配的元素，如下：

```
NSMutableArray<NSString *> *array = [[NSMutableArray alloc] init];
[array addObject:@1];
```

编译器会给我们一个这样的警告：

```
20     NSMutableArray<NSString *> *array = [[NSMutableArray alloc] init];
21     [array addObject:@1];
22 }                                     Incompatible pointer types sending 'NSNumber *' to parameter of type 'NSString * _Nonnull'
```

2、关于一个类型通配符

观察Xcode7中iOS系统的类，我们可以发现这么一个好玩的东西：`ObjectType`。它既不是一个类型，也不是关键字，然而却大量存在，如下是系统的NSMutableArray的头文件：

```
@interface NSMutableArray<ObjectType> : NSArray<ObjectType>
- (void)addObject:(ObjectType)anObject;
- (void)insertObject:(ObjectType)anObject atIndex:(NSUInteger)index;
- (void)removeLastObject;
- (void)removeObjectAtIndex:(NSUInteger)index;
- (void)replaceObjectAtIndex:(NSUInteger)index withObject:(ObjectType)anObject;
- (instancetype)init NS_DESIGNATED_INITIALIZER;
- (instancetype)initWithCapacity:(NSUInteger)numItems NS_DESIGNATED_INITIALIZER;
- (nullable instancetype)initWithCoder:(NSCoder *)aDecoder NS_DESIGNATED_INITIALIZER;
@end
```

这个ObjectType其实只是一个类型标识符，它具体怎么写并不重要，只是系统中都约定使用了ObjectType，你也可以在自己的类中按自己的喜好来命名，这个东西有怎样的用处，我用文字描述不清楚，我们可以通过自己来定义一个集合类来理解：创建一个类，继承于NSObject，我取名叫MyArray：

```

//这个类型通配符只能在interface里使用，作用域为@interface到@end之间
//这里我使用Type来做这个通配符
@interface MyArray<Type> : NSObject
@property(n nonatomic, strong, nonnull) NSMutableArray<Type> *array;
-(void)addObject:(nonnull Type)obj;
@end

```

实现如下:

```

- (instancetype)init
{
    self = [super init];
    if (self) {
        _array = [[NSMutableArray alloc] init];
    }
    return self;
}

- (void)addObject:(id)obj
{
    [_array addObject:obj];
}

- (NSString *)description
{
    NSMutableString * str = [[NSMutableString alloc] init];
    for (int i=0; i<_array.count; i++) {
        [str appendString:[NSString stringWithFormat:@"%@\n", _array[i]]];
    }
    return str;
}

```

我们在使用这个自定义的集合类型时，就会有和系统一样的效果了：

```

//ID.
MyArray<NSString *> * array = [[MyArray alloc] init];
[array addObject:@"123"];
[array addObject:(nonnull NSString *)];
M void addObject:(nonnull NSString *)

```

3、关于多参数的泛型集合

多参数的泛型集合，有一个非常好的例子，就是NSDictionary，在Xcode7中我们可以这样写字典：


```
//
NSMutableDictionary<NSString *,NSNumber *> * dic =
    [[NSMutableDictionary alloc] init];
dic setValue:(nullable NSNumber *) forKey:(nonnull NSString *)
M void setValue:(nullable NSNumber *) forKey:(nonnull NSString *)
```

可以看到，字典键值的类型编译器为我们提示了出来，结合上面类型通配符的使用，对于多参的集合，将参数类型用“,”隔开即可。

4、协变性与逆变性

因为有了泛型集合的概念，相比之前，我们的类型实际上更加复杂了，比如还拿我们自定义的集合类型来举例：

```
MyArray<NSString *> * array;
MyArray<NSMutableString *> * muArray;
```

array和muArray在编译器看来已经是不同的类型，如果我们强行转换，会报如下的警告：

```
MyArray<NSString *> * array;
MyArray<NSMutableString *> * muArray;
array = muArray;
⚠ Incompatible pointer types assigning to 'MyArray<NSString *> *' from 'MyArray<NSMutableString *> *'
```

因此，就有了逆变和协变这个概念：

__covariant :子类型指针可以向父类型指针转换

__contravariant:父类型指针可以向子类型转换

上面的情况，我们将自定义的类做如下修改，就不会出现警告：

```
@interface MyArray<__covariant Type> : NSObject
@property(nonatomic,strong,nonnull)NSMutableArray<Type> *array;
-(void)addObject:(nonnull Type)obj;
@end
```

四、类型延拓符的应用

在开发中，开发者经常会遇到这样的情况，例如通过tag获取某些UI控件时，viewWithTag方法通常会返回给我们一个UIView类型的指针，这就需要开发者手动的强转一下，十分麻烦。新增加的__typeof修饰符可以帮助我们解除这个烦恼。我们还从自定义的那个数组类开刀，对其添加一个属性：

```
@interface MyArray<__covariant Type> : NSObject
@property(nonatomic,strong,nonnull)NSMutableArray<Type> *array;
```

```
@property(nonnull,strong,nonatomic)NSMutableArray<UIView *> * viewArray
;
-(void)addObject:(nonnull Type)obj;
@end
```

创建一个自定义的数组对象，并向其中添加一个UIButton，我们会看到有如下一个警告：

```
- (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' 2

```
}
```

这也是我们开发中常遇到的问题，对吧，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：

```
@property(nonnull,strong,nonatomic)NSMutableArray<__typeof UIView *> *
viewArray;
```

警告就消失了，很cool吧。

这个修饰符就是告诉编译器，这里可以返回UIView的子类指针。