新道互联网事业部 iOS 代码规范

一般原则

清晰性

- 最好是既清晰又简短,但不要为简短而丧失清晰性

代码	点评
insertObject: atIndex:	好
insert: at:	不清晰：要插入什么？“at”表示什么？
removeObjectAtIndex:	好
removeObject:	这样也不错,因为方法是移除作为参数的对象
remove:	不清晰：要移除什么?

- 名称通常不缩写,即使名称很长,也要拼写完全（禁止拼音！！！）

代码	点评
destinationSelection:	好
destSel:	不好
setBackgroundColor:	好
setBkgdColor:	不好

你可能会认为某个缩写广为人知,但有可能并非如此,尤其是当你的代码被来自不同文化和语言背景的开发人员所使用时。

- 然而,你可以使用少数非常常见,历史悠久的缩写。请参考“可接受的缩略名”一节
- 避免使用有歧义的 API 名称,如那些能被理解成多种意思的方法名称

代码	点评
sendPort:	是发送端口还是返回一个发送端口？
displayName:	是显示一个名称还是返回用户界面中控件的标题？

一致性

- 尽可能使用与 Cocoa 编程接口命名保持一致的名称。如果你不太确定某个命名的一致性，请浏览一下头文件或参考文档中的范例。
- 在使用多态方法的类中，命名的一致性非常重要，在不同类中实现相同功能的方法应该具有相同的名称。

代码	点评
- (int) tag	在 NSView，NSCell，NSControl 中有定义
- (void)setStringValue:(NSString *)	在许多 Cocoa 类中有定义

更多请看 函数参数

前缀

通常，软件会被打包成一个框架或多个紧密相关的框架(如 Foundation 和 Application Kit 框架)。但由于 Cocoa 没有像 C++一样的命名空间概念，所以我们只能用前缀来区分软件的功能范畴，防止命名冲突。

- 类名和常量应该始终使用三个字母的前缀（**注意，新道互联网事业部类名前缀一律使用 STT，Seentao Technology 的缩写**），因为苹果保留所有两个字母的前缀的使用权，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。

下面是常见的苹果官方的前缀

前缀	Cocoa 框架
NS	Foundation
NS	Application Kit
AB	Address Book
IB	Interface Builder

比如在我们的 APP 中蓝牙模块（Bluetooth low energy）管理类

```
@interface BLEManager : NSObject
@property (assign) BLEDeviceType deviceType;
@end
```

书写约定

在为 API 元素命名时,请遵循如下一些简单的书写约定

- 对于包含多个单词的名称,不要使用标点符号作为名称的一部分或作为分隔符(下划线,破折号等); 此外,大写每个单词的首字符并将这些单词连续拼写在一起。请注意以下限制:
    - 方法名小写第一个单词的首字符,大写后续所有单词的首字符。方法名不使用前缀。如: fileExistsAtPath:isDirectory: 如果方法名以一个广为人知的大写首字母缩略词开头,该规则不适 用, 如:NSImage 中的 TIFFRepresentation
    - 函数名和常量名使用与其关联类相同的前缀,并且要大写前缀后面所有单词的首字符。如: NSRunAlertPanel, NSCellDisabled
    - 避免使用下划线来表示名称的私有属性。苹果公司保留该方式的使用。如果第三方这样使用可能会导致命名冲突,他们可能会在无意中用自己的方法覆盖掉已有的私有方法,这会导致严重的后 果。请参考“私有方法”一节以了解私有 API 的命名约定的建议

类与协议命名

类名应包含一个明确描述该类(或类的对象)是什么或做什么的名词。类名要有合适的前缀(请参考“前缀” 一节)。Foundation 及 Application Kit 有很多这样例子,如:NSString, NSData, NSScanner, NSApplication, NSButton 以及 UIButton。

协议应该根据对方法的行为分组方式来命名。

- 大多数协议仅组合一组相关的方法,而不关联任何类,这种协议的命名应该使用动名词(ing),以不与 类名混淆。

代码	点评
NSLocking	good
NSLock	糟糕,它看起来像类名

- 有些协议组合一些彼此无关的方法(这样做是避免创建多个独立的小协议)。这样的协议倾向于与某个类关联在一起,该类是协议的主要体现者。在这种情形,我们约定协议的名称与该类同名。NSObject 协议就是这样一个例子。这个协议组合一组彼此无关的方法,有用于查询对象在其类层次中位置的方法,有使之能调用特殊方法的方法以及用于增减引用计数的方法。由于 NSObject 是这些方法的主要体现者,所以我们用类的名称命名这个协议。

头文件

头文件的命名方式很重要,我们可以根据其命名知晓头文件的内容。

- 声明孤立的类或协议:将孤立的类或协议声明放置在单独的头文件中,该头文件名称与类或协议同名

头文件	声明
NSApplication.h	NSApplication 类

- 声明相关联的类或协议:将相关联的声明(类,类别及协议) 放置在一个头文件中,该头文件名称与 主要的类/类别/协议的名字相同。

头文件	声明
NSString.h	NSString 和 NSMutableString 类
NSLock.h	NSLocking 协议和 NSLock, NSConditionLock, NSRecursiveLock 类

- 包含框架头文件:每个框架应该包含一个与框架同名的头文件,该头文件包含该框架所有公开的头文件。

头文件	声明
Foundation.h	Foundation.framework

- 为已有框架中的某个类扩展 API:如果要在一个框架中声明属于另一个框架某个类的范畴类的方法, 该头文件的命名形式为:原类名+“Additions”。如 Application Kit 中的 NSBundleAdditions.h
- 相关联的函数与数据类型:将相联的函数,常量,结构体以及其他数据类型放置到一个头文件中,并以合适的名字命名。如 Application Kit 中的 NSGraphics.h

方法命名

一般性原则

为方法命名时,请考虑如下一些一般性规则:

- **方法名不要使用 `new` 作为前缀。**
- 小写第一个单词的首字符,大写随后单词的首字符,不使用前缀。请参考“书写约定”一节。有两种例 外情况:1,方法名以广为人知的大写字母缩略词(如 TIFF or PDF)开头;2,私有方法可以使用统一的前缀来分组和辨识,请参考“私有方法”一节
- 表示对象行为的方法,名称以动词开头:

```
- (void) invokeWithTarget:(id)target:
- (void) selectTabViewItem:(NSTableViewItem *)tableViewItem
```

名称中不要出现 do 或 does,因为这些助动词没什么实际意义。也不要在动词前使用副词或形容词修饰。

- 如果方法返回方法接收者的某个属性,直接用属性名称命名。不要使用 get， 除非是间接返回一个或多个值。请参考“访问方法”一节。

代码	点评
- (NSSize) cellSize;	对
- (NSSize) calcCellSize;	错
- (NSSize) getCellSize;	错

- 参数要用描述该参数的关键字命名

代码	点评
- (void) sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;	对
- (void) sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;	错

- 参数前面的单词要能描述该参数。

代码	点评
- (id) viewWithTag:(int)aTag;	对
- (id) taggedView:(int)aTag;	错

- 细化基类中的已有方法:创建一个新方法,其名称是在被细化方法名称后面追加参数关键词

```
- (id)initWithFrame:(CGRect)frameRect;//NSView, UIView.
- (id)initWithFrame:(NSRect)frameRect mode:(int)aMode  cellClass:(Class)factoryId numberOfRows:(int)rowsHigh
numberOfColumns (int)colsWide;//NSMatrix, a subclass of NSView
```

- 不要使用 and 来连接用属性作参数的关键字

代码	点评
- (int)runModalForDirectory:(NSString *)path file:(NSString *)name types:(NSArray *)fileTypes;	对
- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes;	错

虽然上面的例子中使用 add 看起来也不错,但当你方法有太多参数关键字时就有问题。

- 如果方法描述两种独立的行为,使用 and 来串接它们

```
- (BOOL) openFile:(NSString *)fullPath withApplication:(NSString NSWorkspace *)appName
andDeactivate:(BOOL)flag;//NSWorkspace.
```

访问方法

访问方法是对象属性的读取与设置方法。其命名有特定的格式依赖于属性的描述内容。

- 如果属性是用名词描述的,则命名格式为:

```
- (void) setNoun:(type)aNoun;
- (type) noun;
```

例如:

```
- (void) setgColor:(NSColor *)aColor;
- (NSColor *) color;
```

- 如果属性是用形容词描述的,则命名格式为:

```
- (void) setAdjective:(BOOL)flag;
- (BOOL) isAdjective;
```

例如:

```
- (void) setEditable:(BOOL)flag;
- (BOOL) isEditable;
```

- 如果属性是用动词描述的,则命名格式为:(动词要用现在时时态)

```
- (void) setVerbObject:(BOOL)flag;
- (BOOL) verbObject;
```

例如:

```
- (void) setShowAlpha:(BOOL)flag;
- (BOOL) showsAlpha;
```

- 不要使用动词的过去分词形式作形容词使用

```
- (void)setAcceptsGlyphInfo:(BOOL)flag;  //对
- (BOOL)acceptsGlyphInfo;               //对
- (void)setGlyphInfoAccepted:(BOOL)flag; //错
- (BOOL)glyphInfoAccepted;              //错
```

- 可以使用情态动词(can, should, will 等)来提高清晰性,但不要使用 do 或 does

```
- (void) setCanHide:(BOOL)flag;            //对
```

```
- (BOOL) canHide;                          //对
- (void) setShouldCloseDocument:(BOOL)flag; //对
- (void) shouldCloseDocument;               //对
- (void) setDoseAcceptGlyphInfo:(BOOL)flag; //错
- (BOOL) doseAcceptGlyphInfo;               //错
```

- 只有在方法需要间接返回多个值的情况下,才使用 get

```
- (void) getLineDash:(float *)pattern count:(int *)count phase:(float *)phase;  //NSBezierPath
```

像上面这样的方法,在其实现里应允许接受 NULL 作为其 in/out 参数,以表示调用者对一个或多个返回 值不感兴趣。

委托方法

委托方法是那些在特定事件发生时可被对象调用,并声明在对象的委托类中的方法。它们有独特的命名约 定,这些命名约定同样也适用于对象的数据源方法。

- 名称以标示发送消息的对象的类名开头,省略类名的前缀并小写类第一个字符

```
- (BOOL) tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;
```

- 冒号紧跟在类名之后(随后的那个参数表示委派的对象)。该规则不适用于只有一个 sender 参数的方法

```
- (BOOL) applicationOpenUntitledFile:(NSApplication *)sender;
```

- 上面的那条规则也不适用于响应通知的方法。在这种情况下,方法的唯一参数表示通知对象

```
- (void) windowDidChangeScreen:(NSNotification *)notification;
```

- 用于通知委托对象操作即将发生或已经发生的方法名中要使用 did 或 will

```
- (void) browserDidScroll:(NSBrowser *)sender;
- (NSUndoManager *) windowWillReturnUndoManager:(NSWindow *)window;
```

- 用于询问委托对象可否执行某操作的方法名中可使用 did 或 will,但最好使用 should

```
- (BOOL) windowShouldClose:(id)sender;
```

集合方法

管理对象(集合中的对象被称之为元素)的集合类,约定要具备如下形式的方法:

```
- (void) addElement:(elementType)adObj;
- (void) removeElement:(elementType)anObj;
- (NSArray *)elements;
```

例如:

```
- (void) addLayoutManager:(NSLayoutManager *)adObj;
- (void) removeLayoutManager:(NSLayoutManager *)anObj;
- (NSArray *)layoutManagers;
```

集合方法命名有如下一些限制和约定:

- 如果集合中的元素无序,返回 NSSet,而不是 NSArray
- 如果将元素插入指定位置的功能很重要,则需具备如下方法:

```
- (void) insertElement:(elementType)anObj atIndex:(int)index;
- (void) removeElementAtIndex:(int)index;
```

集合方法的实现要考虑如下细节:

- 以上集合类方法通常负责管理元素的所有者关系,在 add 或 insert 的实现代码里会 retain 元素,在 remove 的实现代码中会 release 元素
- 当被插入的对象需要持有指向集合对象的指针时,通常使用 set... 来命名其设置该指针的方法,且不 要 retain 集合对象。比如上面的 insertLayerManager:atIndex: 这种情形,NSLayoutManager 类使 用如下方法:

```
- (void) setTextStorage:(NSTextStorage *)textStorage;
- (NSTextStorage *)textStorage;
```

通常你不会直接调用 setTextStorage:,而是覆写它。

另一个关于集合约定的例子来自 NSWindow 类:

```
- (void) addChildWindow:(NSWindow *)childWin ordered:(NSWindowOrderingMode)place;
- (void) removeChildWindow:(NSWindow *)childWin;
- (NSArray *)childWindows;
- (NSWindow *) parentWindow;
- (void) setParentWindow:(NSWindow *)window;
```

方法参数

命名方法参数时要考虑如下规则:

- 如同方法名,参数名小写第一个单词的首字符,大写后继单词的首字符。如:removeObject:(id)anObject
- 不要在参数名中使用 pointer 或 ptr,让参数的类型来说明它是指针
- 避免使用 one, two,...,作为参数名
- 避免为节省几个字符而缩写

按照 Cocoa 惯例,以下关键字与参数联合使用:

```
...action:(SEL)aSelector
..alignment:(int)mode
...atIndex:(int)index
...content:(NSRect)aRect
...doubleValue:(double)aDouble
...floatValue:(float)aFloat
...font:(NSFont *)fontObj
...frame:(NSRect)frameRect
...intValue:(int)anInt
...keyEquivalent:(NSString *)charCode
...length:(int)numBytes
...point:(NSPoint)aPoint
...stringValue:(NSString *)aString
...tag:(int)anInt
...target:(id)anObject
...title:(NSString *)aString
```

私有方法

大多数情况下,私有方法命名相同与公共方法命名约定相同,但通常我们约定给私有方法添加前缀,以便 与公共方法区分开来。即使这样,私有方法的名称很容易导致特别的问题。当你设计一个继承自 Cocoa framework 某个类的子类时,你无法知道你的私有方法是否不小心覆盖了框架中基类的同名方法。

Cocoa framework 的私有方法名称通常以下划线作为前缀(如:_fooData),以标示其私有属性。基于这 样的事实,遵循以下两条建议:

- 不要使用下划线作为你自己的私有方法名称的前缀,Apple 保留这种用法。
- 若要继承 Cocoa framework 中一个超大的类(如:NSView),并且想要使你的私有方法名称与基类中的区别开来,你可以为你的私有方法名称添加你自己的前缀。这个前缀应该具有唯一性，建议用"p_Method"格式， p 代表 private。

尽管为私有方法名称添加前缀的建议与前面类中方法命名的约定冲突,这里的意图有所不同:为了防止不 小心地覆盖基类中的私有方法。

函数命名

Objective-C 允许通过函数(C 形式的函数)描述行为,就如成员方法一样。如果隐含的类为单例或在处理函数子系统时,你应当优先使用函数,而不 是类方法。

函数命名应该遵循如下几条规则:

- 函数命名与方法命名相似,但有两点不同:
    1. 它们有前缀,其前缀与你使用的类和常量的前缀相同
    2. 大写前缀后紧跟的第一个单词首字符
- 大多数函数名称以动词开头,这个动词描述该函数的行为

```
NSHighlightRect
NSDeallocateObject
```

查询属性的函数有个更多的规则要遵循:

- 查询第一个参数的属性的函数,省略动词
- `unsigned int NSEventMaskFromType(NSEventType type)`
- `float NSHeight(NSRect rect)`
- 返回值为引用的方法,使用 Get

```
const char *NSGetSizeAndAlignment(const char *typePtr, unsigned int *sizep, unsigned int *alignp)
```

- 返回 boolean 值的函数,名称使用判断动词 is/does 开头

```
BOOL NSDecimalIsNotANumber(const NSDecimal *decimal)
```

属性与数据类型命名

这一节描述属性，实例变量，常量，异常以及通知的命名约定。

属性声明与实例变量

属性声明的命名大体上和访问方法的命名是一致的。假如属性是动词或名词，格式如下

```
@property (...) typenounOrVerb;
```

例如：

```
@property (strong) NSString *title;
@property (assign) BOOL showsAlpha;
```

如果属性是形容词，名字去掉"is"前缀，但是要特别说明一下符合规范的 get 访问方法，例如

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

多数情况下，你声明了一个属性，那么就会自动生成对应的实例变量。

确保实例变量名简明扼要地描述了它所代表的属性。通常，你应该使用访问方法，而不是直接访问实例变量（除了在 init 或者 dealloc 方法里）。为了便于标识实例变量，在名字前面加个下划线"_"，例如:

```
@implementation MyClass {
    BOOL _showsTitle;
}
```

在为类添加实例变量是要注意:

- 避免创建 public 实例变量
- 使用 @private,@protected 显式限定实例变量的访问权限
- 如果实例变量别设计为可被访问的,确保编写了访问方法

常量

常量命名规则根据常量创建的方式不同而大不同。

枚举常量

- 使用枚举来定义一组相关的整数常量
- 枚举常量与其 typedef 命名遵守函数命名规则。如:来自 NSMatrix.h 中的例子

```
typedef NS_ENUM(NSInteger, _NSMatrixMode) {
    NSRadioModeMatrix       = 0,
    NSHighlightModeMatrix    = 1,
    NSListModeMatrix        = 2,
    NSTrackModeMatrix       = 3,
} NSMatrixMode;
```

- 位掩码常量可以使用不具名枚举。如:

```
enum {
    NSBorderlessWindowMask      = 0,
    NSTitledWindowMask          = 1 << 0,
    NSClosableWindowMask        = 1 << 1,
    NSMiniaturizableWindowMask  = 1 << 2,
    NSResizableWindowMask       = 1 << 3
};
```

const 常量

- 尽量用 const 来修饰浮点数常数，以及彼此没有关联的整数常量（否则使用枚举）
- const 常量命名范例：

```
const float NSLightGray;
```

枚举常量命名规则与函数命名规则相同。

其他常量

- 通常不使用 #define 来创建常量。如上面所述，整数常量请使用枚举，浮点数常量请使用 const
- 使用大写字母来定义预处理编译宏。如：

```
#ifdef  DEBUG
```

- 编译器定义的宏名首尾都有双下划线。 如：

```
__MACH__
```

- 为 notification 名及 dictionary key 定义字符串常量,从而能够利用编译器的拼写检查,减少书写错误。Cocoa 框架提供了很多这样的范例:

```
APPKIT_EXTERN NSString *NSPrintCopies;
```

实际的字符串值在实现文件中赋予。(注意: APPKIT_EXTERN 宏等价于 Objective-C 中 extern)

异常与通知

异常与通知的命名遵循相似的规则,但是它们有各自推荐的使用模式。

异常

虽然你可以处于任何目的而使用异常(由 NSException 类及相关类实现),Cocoa 通常不使用异常来处 理常规的,可预料的错误。在这些情形下,使用诸如 nil, NULL, NO 或错误代码之类的返回值。异常的典 型应用类似数组越界之类的编程错误。

异常由具有如下形式的全局 NSString 对象标识:

```
[Prefix] + UniquePartOfName + Exception
```

UniquePartOfName 部分是有连续的首字符大写的单词组成。例如:

```
NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException
```

通知

如果一个类有委托,那它的大部分通知可能由其委托的委托方法来处理。这些通知的名称应该能够反应其 响应的委托方法。比如,当应用程序提交 NSApplicationDidBecomeActiveNotification 通知时,全局 NSApplication 对象的委托会注册从而能够接收 applicaitonDidBecomeActive: 消息。

通知由具有如下形式的全局 NSString 对象标识:

```
[相关联类的名称] + [Did 或 Will] + [UniquePartOfName] + Notification
```

例如:

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

图片命名

图片文件命名采用 `module_type_identifier_state` 规则，只需要@2x 和@3x 图片。
缩略 type 可用如下例子

- icon
- btn
- bg
- line

- logo
- pic
- img

使用图片时候不要有 `.png` 后缀
参考:

```
UIImage *settingIcon = [UIImage imageNamed:@"edu_icon_btn_setting"];
```

图片目录中被用于类似目的的图片应归入各自的组中。

可接受的缩略语

在设计编程接口时,通常名称不要缩写。然而,下面列出的缩写要么是固定下来的要么是过去被广泛使用 的,所以你可以继续使用。关于缩写有一些额外的注意事项:

- 标准 C 库中长期使用的缩写形式是可以接受的。如:"alloc", "getc"
- 你可以在参数名中更自由地使用缩写。如:imageRep, col(column), obj, otherWin

常见的缩写

缩写	含义
alloc	Allocate
alt	Alternate
app	Application
calc	Calculate
dealloc	Deallocate
func	Function
horiz	Horizontal
info	Information
init	Initialize
int	Integer
max	Maximum
min	Minimum
msg	Message
nib	Interface Builder archive
pboard	Pasteboard

缩写	含义
rect	Rectangle
Rep	Representation
temp	Temporary
vert	Vertical

常见的略写

```
ASCII,PDF,XML,HTML,URL,RTF,HTTP,TIFF JPG,GIF,LZW,ROM,RGB,CMYK,MIDI,FTP
```

代码注释规范

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。

文件注释

采用 Xcode 自动生成的注释格式，修改部分参数：

```
//
//  AppDelegate.m
//  LeFit
//
//  Created by Zhang Wen on 15-3-22.
//  Copyright (c) 2015 Appscomm LLC. All rights reserved.
//
```

其中**项目名称**、**创建人**、**公司版权**需要填写正确。

import 注释

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"
```

方法注释

采用 javadoc 的格式，可以使用 XCode 插件 VVDocumenter-Xcode 快速添加，只需输入`///`即可

```
/**
 *  功能描述
 *
 *  @param tableView 参数说明
 *  @param section   参数说明
 *
 *  @return 返回值说明
 */
- (NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section
{
    return [self.familyNames objectAtIndex:section];
}
```

代码块注释

单行的用`//`+空格开头，多汗的采用`/* */`注释

TODO 注释

TODO 很不错，有时候，注释确实是为了标记一些未完成的或完成的不尽如人意的地方，这样一搜索，就知道还有哪些活要干，日志都省了。

格式：`//TODO:说明`

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    //TODO:增加初始化
    return YES;
}
```

代码结构

实现文件中的代码结构，提倡以下约定：

- 用`#pragma mark -`将函数或方法按功能进行分组。
- dealloc 方法放到实现文件的最顶部。
  这样是为了时刻提醒你要记得释放相关资源。
- delgate 或协议相关方法放到一般内容之后。

```
#pragma mark - Lifecycle


- (void)dealloc {}
- (instancetype)init {}
- (void)viewDidLoad {}
- (void)viewWillAppear:(BOOL)animated {}
- (void)didReceiveMemoryWarning {}

#pragma mark - Custom Accessors


- (void)setCustomProperty:(id)value {}
- (id)customProperty {}
```

```
#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 {}
```

代码排版格式

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;
```

间距

- 一个缩进使用 4 个空格，永远不要使用制表符（tab）缩进。请确保在 Xcode 中设置了此偏好。
- 方法的大括号和其他的大括号（`if`/`else`/`switch`/`while` 等等）始终和声明在同一行开始，在新的一行结束。

推荐：

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

- 方法之间应该正好空一行，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，但往往可能应该创建一个新的方法。
- `@synthesize` 和 `@dynamic` 在实现中每个都应该占一个新行。

长度

- 每行代码的长度最多不超过 100 个字符
- 尝试将单个函数或方法的实现代码控制在 30 行内

  如果某个函数或方法的实现代码过长，可以考量下是否可以将代码拆分成几个小的拥有单一功能的方法。

  30 行是在 13 寸 macbook 上 XCode 用 14 号字体时，恰好可以让一个函数的代码做到整屏完全显示的行数。

- 将单个实现文件里的代码行数控制在 500～600 行内

  为了简洁和便于阅读，建议将单个实现文件的代码行数控制在 500～600 行以内最好。

  当接近或超过 800 行时，就应当开始考虑分割实现文件了。

  最好不要出现代码超过 1000 行的实现文件。

  我们一般倾向于认为单个文件代码行数越长，代码结构就越不好。而且，翻代码翻的手软啊。

  可以使用 Objective-C 的 Category 特性将实现文件归类分割成几个相对轻量级的实现文件。

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;
```

错误处理

当引用一个返回错误参数（error parameter）的方法时，应该针对返回值，而非错误变量。

推荐：

```
NSError *error;
if (![self trySomethingWithError:&error]) {
    // 处理错误
}
```

反对：

```
NSError *error;
[self trySomethingWithError:&error];
if (error) {
    // 处理错误
```

```
}
```

一些苹果的 API 在成功的情况下会写一些垃圾值给错误参数（如果非空），所以针对错误变量可能会造成虚假结果（以及接下来的崩溃）。

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;
```

- 实现文件中，方法的左花括号不另起一行，和方法名同行，并且和方法名之间保持 1 个空格

  此条是为了和 XCode6.1 模板生成的文件的代码风格保持一致。

```
//赞成的
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

//不赞成的
- (void)didReceiveMemoryWarning
{

    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

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`。

block

`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 `4` 个空格。 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 `4` 空格缩进。
- 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。

```
// The entire block fits on one line.
[operation setCompletionBlock:^{ [self onOperationDone]; }];


[operation setCompletionBlock:^{
    [self.delegate newDataAvailable];
}];

dispatch_async(fileIOQueue_, ^{
    NSString *path = [self sessionFilePath];
    if (path) {
        ...
    }
});



[[SessionService sharedService]
    loadWindowWithCompletionBlock:^(SessionWindow *window) {
        if (window) {
            [self windowDidLoad:window];
        }
        else {
            [self errorLoadingWindow];
        }
    }
];


[[SessionService sharedService]
    loadWindowWithCompletionBlock:
        ^(SessionWindow *window) {
            if (window) {
                [self windowDidLoad:window];
            }
            else {
                [self errorLoadingWindow];
            }
        }
];


void (^largeBlock)(void) = ^{
```

```
    ...
};
[operationQueue_ addOperationWithBlock:largeBlock];
```

开发实践

本章主要描述开发过程中一些比较固定的实践技巧，写代码时可以直接套用

初始化

- 初始化方法的返回类型用 `instancetype`，不要用 `id`
  关于 instancetype 的介绍参见 NSHipster.com。

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。

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][http://developer.apple.com/library/ios/#documentation/graphicsimaging/reference/CGGeometry/Reference/reference.html]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

- 定义常量时，除非明确的需要将常量当成宏使用，否则优先使用 `const`，而非`#define`
- 只在某一个特定文件里面使用的常量，用 static

```
    static CGFloat const RWImageThumbnailHeight = 50.0;
```

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()`

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

位掩码

当用到位掩码时，使用 `NS_OPTIONS` 宏。

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

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。有名字的类目（例如 `ASMPrivate` 或 `private`）永远都不应该使用，除非要扩展其他类。

```
@interface NYTAdvertisement ()


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


@end
```

布尔值

- Objective-C 的布尔值只使用 `YES` 和 `NO`
- `true` 和 `false` 只能用于 CoreFoundation，C 或 C++的代码中
- 禁止将某个值或表达式的结果与 `YES` 进行比较
  因为 BOOL 被定义成 signed char。这意味着除了 YES(1)和 NO(0)以外，它还可能是其他值。

  因此 C 或 C++中的非 0 为真并不一定就是 YES

```
//以下都是被禁止的
- (BOOL)isBold {
    return [self fontTraits] & NSFontBoldTrait;
}

- (BOOL)isValid {
    return [self stringValue];
}

if ([self isBold] == YES) {
    //...
}


//以下才是赞成的方式
- (BOOL)isBold {
    return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}

- (BOOL)isValid {
    return [self stringValue] != nil;
}

- (BOOL)isEnabled {
    return [self isValid] && [self isBold];
}

if ([self isBold]) {
    //...
}
```

- 虽然 `nil` 会被直接解释成 `NO`，但还是建议在条件判断时保持与 nil 的比较，因为这样代码更直观。

```
//比如，更直观的代码
if (someObject != nil) {
```

```
    //...
}

//没那么直观的代码
if (!someObject) {
    //...
}
```

- 在 C 或 C++代码中，要注意 NULL 指针的检测。

  向一个 nil 的 Objective-C 对象发送消息不会导致崩溃。但由于 Objective-C 运行时不会处理给 NULL 指针的情况，所以为了避免崩溃，需要自行处理对于 C/C++的 NULL 指针的检测。

- 如果某个 `BOOL` 类型的 property 的名字是一个形容词，建议为 getter 方法加上一个"is"开头的别名。

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

- 在方法实现中，如果有 block 参数，要注意检测 block 参数为 nil 的情况。

```
- (void)exitWithCompletion:(void(^)(void))completion {
    // 错误。 如果外部调用此方法时 completion 传入 nil，此处会发生 EXC_BAD_ACCESS
    completion();

    // 正确。如果 completion 不存在则不调用。
    if (completion) {
        completion();
    }
}
```