Objective-C开发编码规范

2015-05-07 16:15 编辑：suiling 分类：iOS开发 来源：QianKaiLu 5 3586

Objective-C 开发 编码规范

Objective-C 编码规范，内容来自苹果、谷歌的文档翻译，自己的编码经验和对其它资料的总结。

概要

Objective-C 是一门面向对象的动态编程语言，主要用于编写 iOS 和 Mac 应用程序。关于 Objective-C 的编码规范，苹果和谷歌都已经有很好的总结：

- Apple Coding Guidelines for Cocoa
- Google Objective-C Style Guide

本文主要整合了对上述文档的翻译、作者自己的编程经验和其他的相关资料，为公司总结出一份通用的编码规范。

代码格式

使用空格而不是制表符 Tab

不要在工程里使用 Tab 键，使用空格来进行缩进。在 Xcode > Preferences > Text Editing 将 Tab 和自动缩进都设置为 **4** 个空格。（Google 的标准是使用两个空格来缩进，但这里还是推荐使用 Xcode 默认的设置。）

每一行的最大长度

同样的，在 Xcode > Preferences > Text Editing > Page guide at column: 中将最大行长设置为 **80** ，过长的一行代码将会导致可读性问题。

热门资讯


13个小技巧帮你征服Xcode
点击量 14393


几个iOS工程通用模块介绍
点击量 8759

10个iOS开发常见错误，你中招了吗
点击量 6959

流行iOS网络通信库AFNetworking曝
点击量 6823


App Store审核更新：报时Watch
点击量 5460

让程序员变懒的工具：Jenkins + 蒲公
点击量 5452


看一款简单的汇率应用如何席卷App
点击量 5373


有什么好的方案可以替代抽屉式导
点击量 4272


iOS开发的一些小技巧
点击量 4198


源码推荐（4.24）：iOS开发
点击量 3786

综合评论

mark !!!
拎壶冲 评论了 iOS音频播放(一)：概述

yes
拎壶冲 评论了 如何优化 iOS 通知

如果是整数，都必须用NSInteger，有木有很死板和弱智
丹心一片 评论了 Objective-C开发编码规范

微软真有钱

函数的书写

一个典型的 Objective-C 函数应该是这样的：

```
- (void)writeVideoFrameWithData:(NSData *)frameData timeStamp:(int)timeStamp {
...
}
```

在 - 和 (void) 之间应该有一个空格，第一个大括号 { 的位置在函数所在行的末尾，同样应该有一个空格。（我司的 C 语言规范要求是第一个大括号单独占一行，但考虑到 OC 较长的函数名和苹果 SDK 代码的风格，还是将大括号放在行末。）

如果一个函数有特别多的参数或者名称很长，应该将其按照 : 来对齐分行显示：

```
-(id)initWithModel:(IPCModle)model
ConnectType:(IPCConnectType)connectType
Resolution:(IPCResolution)resolution
AuthName:(NSString *)authName
Password:(NSString *)password
MAC:(NSString *)mac
AzIp:(NSString *)az_ip
AzDns:(NSString *)az_dns
Token:(NSString *)token
Email:(NSString *)email
Delegate:(id)delegate;
```

在分行时，如果第一段名称过短，后续名称可以以 Tab 的长度（ 4 个空格）为单位进行缩进：

```
- (void)short:(GTMFoo *)theFoo
longKeyword:(NSRect)theRect
evenLongerKeyword:(float)theInterval
error:(NSError **)theError {
...
}
```

函数调用

函数调用的格式和书写差不多，可以按照函数的长短来选择写在一行或者分成多行：

```
// 写在一行
[myObject doFooWith:arg1 name:arg2 error:arg3];
// 分行写，按照 ':' 对齐
[myObject doFooWith:arg1
name:arg2
error:arg3];
// 第一段名称过短的话后续可以进行缩进
[myObj short:arg1
longKeyword:arg2
evenLongerKeyword:arg3
error:arg4];
```

以下写法是错误的：

```
// 错误，要么写在一行，要么全部分行
[myObject doFooWith:arg1 name:arg2
error:arg3];
[myObject doFooWith:arg1
name:arg2 error:arg3];
// 错误，按照 ':' 来对齐，而不是关键字
[myObject doFooWith:arg1
name:arg2
error:arg3];
```

@public 和 @private 标记符

@public 和 @private 标记符应该以**一个空格**来进行缩进：

```
@interface MyClass : NSObject {
@public
...
@private
...
```

```
}
@end
```

协议（Protocols）

在书写协议的时候注意用 <> 括起来的协议和类型名之间是没有空格的，比如 IPCConnectHandler(), 这个规则适用所有书写协议的地方，包括函数声明、类声明、实例变量等等：

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

闭包（Blocks）

根据 block 的长度，有不同的书写规则：

- 较短的 block 可以写在一行内。
- 如果分行显示的话， block 的右括号 } 应该和调用 block 那行代码的第一个非空字符对齐。
- block 内的代码采用 **4 个空格** 的缩进。
- 如果 block 过于庞大，应该单独声明成一个变量来使用。
- ^ 和 (之间， ^ 和 { 之间都没有空格，参数列表的右括号) 和 { 之间有一个空格。

```
// 较短的 block 写在一行内
[operation setCompletionBlock:^{ [self onOperationDone]; }];
// 分行书写的 block ，内部使用 4 空格缩进
[operation setCompletionBlock:^{
[self.delegate newDataAvailable];
}];
// 使用 C 语言 API 调用的 block 遵循同样的书写规则
dispatch_async(_fileIOQueue, ^{
NSString* path = [self sessionFilePath];
if (path) {
// ...
}
});
// 较长的 block 关键字可以缩进后在新行书写，注意 block 的右括号 '}' 和调用 block 那行代码
[[SessionService sharedService]
loadWindowWithCompletionBlock:^(SessionWindow *window) {
if (window) {
[self windowDidLoad:window];
} else {
[self errorLoadingWindow];
}
}];
// 较长的 block 参数列表同样可以缩进后在新行书写
[[SessionService sharedService]
loadWindowWithCompletionBlock:
^(SessionWindow *window) {
if (window) {
[self windowDidLoad:window];
} else {
[self errorLoadingWindow];
}
}];
// 庞大的 block 应该单独定义成变量使用
void (^largeBlock)(void) = ^{
// ...
};
[_operationQueue addOperationWithBlock:largeBlock];
// 在一个调用中使用多个 block ，注意到他们不是像函数那样通过 ':' 对齐的，而是同时进行了 4 个
[myObject doSomethingWith:arg1
firstBlock:^(Foo *a) {
// ...
}
secondBlock:^(Bar *b) {
// ...
}];
```

数据结构的语法糖

应该使用可读性更好的语法糖来构造 NSArray， NSDictionary 等数据结构，避免使用冗长的 alloc,init 方法。

如果构造代码写在一行，需要在括号两端留有一个空格，使得被构造的元素于与构造语法区分开来：

```
// 正确，在语法糖的 "[]" 或者 "{}" 两端留有空格
NSArray *array = @[ [foo description], @"Another String", [bar description] ];
NSDictionary *dict = @{ NSForegroundColorAttributeName : [NSColor redColor] };
// 不正确，不留有空格降低了可读性
NSArray* array = @[[foo description], [bar description]];
NSDictionary* dict = @{NSForegroundColorAttributeName: [NSColor redColor]};
```

如果构造代码不写在一行内，构造元素需要使用**两个空格**来进行缩进，右括号] 或者 } 写在新的一行，并且与调用语法糖那行代码的第一个非空字符对齐：

```
NSArray *array = @[
@"This",
@"is",
@"an",
@"array"
];
NSDictionary *dictionary = @{
NSFontAttributeName : [NSFont fontWithName:@"Helvetica-Bold" size:12],
NSForegroundColorAttributeName : fontColor
};
```

构造字典时，字典的 Key 和 Value 与中间的冒号 : 都要留有一个空格，多行书写时，也可以将 Value 对齐：

```
// 正确，冒号 ':' 前后留有一个空格
NSDictionary *option1 = @{
NSFontAttributeName : [NSFont fontWithName:@"Helvetica-Bold" size:12],
NSForegroundColorAttributeName : fontColor
};
// 正确，按照 Value 来对齐
NSDictionary *option2 = @{
NSFontAttributeName : [NSFont fontWithName:@"Arial" size:12],
NSForegroundColorAttributeName : fontColor
};
// 错误，冒号前应该有一个空格
NSDictionary *wrong = @{
AKey: @"b",
BLongerKey: @"c",
};
// 错误，每一个元素要么单独成为一行，要么全部写在一行内
NSDictionary *alsoWrong= @{ AKey : @"a",
BLongerKey : @"b" };
// 错误，在冒号前只能有一个空格，冒号后才可以考虑按照 Value 对齐
NSDictionary *stillWrong = @{
AKey : @"b",
BLongerKey : @"c",
};
```

命名规范

基本原则

清晰

命名应该尽可能的清晰和简洁，但在 Objective-C 中，清晰比简洁更重要。由于 Xcode 强大的自动补全功能，我们不必担心名称过长的问题。

```
// 清晰
insertObject:atIndex:
// 不清晰， insert 的对象类型和 at 的位置属性没有说明
insert:at:
// 清晰
removeObjectAtIndex:
// 不清晰， remove 的对象类型没有说明，参数的作用没有说明
remove:
```

不要使用单词的简写，拼写出完整的单词：

```
// 清晰
destinationSelection
setBackgroundColor:
// 不清晰，不要使用简写
destSel
setBkgdColor:
```

然而，有部分单词简写在 Objective-C 编码过程中是非常常用的，以至于成为了一种规范，这些简写可以在代码中直接使用，下面列举了部分：

```
alloc == Allocate max == Maximum
alt == Alternate min == Minimum
app == Application msg == Message
calc == Calculate nib == Interface Builder archive
dealloc == Deallocate pboard == Pasteboard
func == Function rect == Rectangle
horiz == Horizontal Rep == Representation (used in class name such as NSBitmapI
info == Information temp == Temporary
init == Initialize vert == Vertical
int == Integer
```

命名方法或者函数时要避免歧义

```
// 有歧义，是返回 sendPort 还是 send 一个 Port ？
sendPort
// 有歧义，是返回一个名字属性的值还是 display 一个 name 的动作?
displayName
```

一致性

整个工程的命名风格要保持一致性，最好和苹果 SDK 的代码保持统一。不同类中完成相似功能的方法应该叫一样的名字，比如我们总是用 count 来返回集合的个数，不能在 A 类中使用 count 而在 B 类中使用 getNumber 。

使用前缀

如果代码需要打包成 Framework 给别的工程使用，或者工程项目非常庞大，需要拆分成不同的模块，使用命名前缀是非常有用的。

- 前缀由大写的字母缩写组成，比如 Cocoa 中 NS 前缀代表 Founation 框架中的类， IB 则代表 Interface Builder 框架。
- 可以在为类、协议、函数、常量以及 typedef 宏命名的时候使用前缀，但注意**不要**为成员变量或者方法使用前缀，因为他们本身就包含在类的命名空间内。
- 命名前缀的时候不要和苹果 SDK 框架冲突。

命名类和协议（ Class&Protocol ）

类名以大写字母开头，应该包含一个名词来表示它代表的对象类型，同时可以加上必要的前缀，比如 NSString,NSDate,NSScanner,NSApplication 等等。

而协议名称应该清晰地表示它所执行的行为，而且要和类名区别开来，所以通常使用 ing 词尾来命名一个协议，比如 NSCopying,NSLocking 。

有些协议本身包含了很多不相关的功能，主要用来为某一特定类服务，这时候可以直接用类名来命名这个协议，比如 NSObject 协议，它包含了 id 对象在生存周期内的一系列方法。

命名头文件（ Headers ）

源码的头文件名应该清晰地暗示它的功能和包含的内容：

- 如果头文件内只定义了单个类或者协议，直接用类名或者协议名来命名头文件，比如 NSLocale.h 定义了 NSLocale 类。
- 如果头文件内定义了一系列的类、协议、类别，使用其中最主要的类名来命名头文件，比如 NSString.h 定义了 NSString 和 NSMutableString 。
- 每一个 Framework 都应该有一个和框架同名的头文件，包含了框架中所有公共类头文件的引用，比如 Foundation.h

- Framework 中有时候会实现在别的框架中类的类别扩展，这样的文件通常使用被扩展的框架名 +Additions 的方式来命名，比如 NSBundleAdditions.h 。

命名方法（Methods）

Objective-C 的方法名通常都比较长，这是为了让程序有更好地可读性，按苹果的说法 “ 好的方法名应当可以以一个句子的形式朗读出来 ” 。

方法一般以小写字母打头，每一个后续的单词首字母大写，方法名中不应该有标点符号（包括下划线），有两个例外：

- 可以用一些通用的大写字母缩写打头方法，比如 PDF,TIFF 等。
- 可以用带下划线的前缀来命名私有方法或者类别中的方法。

如果方法表示让对象执行一个动作，使用动词打头来命名，注意不要使用 do， does 这种多余的关键字，动词本身的暗示就足够了：

```
// 动词打头的方法表示让对象执行一个动作
- (void)invokeWithTarget:(id)target;
- (void)selectTabViewItem:(NSTabViewItem *)tabViewItem;
```

如果方法是为了获取对象的一个属性值，直接用属性名称来命名这个方法，注意不要添加 get 或者其他的动词前缀：

```
// 正确，使用属性名来命名方法
- (NSSize)cellSize;
// 错误，添加了多余的动词前缀
- (NSSize)calcCellSize;
- (NSSize)getCellSize;
```

对于有多个参数的方法，务必在每一个参数前都添加关键词，关键词应当清晰说明参数的作用：

```
// 正确，保证每个参数都有关键词修饰
- (void)sendAction:(SEL)aSelector toObject:(id)anObject forAllCells:(BOOL)flag;
// 错误，遗漏关键词
- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;
// 正确
- (id)viewWithTag:(NSInteger)aTag;
// 错误，关键词的作用不清晰
- (id)taggedView:(int)aTag;
```

不要用 and 来连接两个参数，通常 and 用来表示方法执行了两个相对独立的操作（从设计上来说，这时候应该拆分成两个独立的方法）：

```
// 错误，不要使用 "and" 来连接参数
- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(
// 正确，使用 "and" 来表示两个相对独立的操作
- (BOOL)openFile:(NSString *)fullPath withApplication:(NSString *)appName andDea
```

方法的参数命名也有一些需要注意的地方：

- 和方法名类似，参数的第一个字母小写，后面的每一个单词首字母大写
- 不要再方法名中使用类似 pointer,ptr 这样的字眼去表示指针，参数本身的类型足以说明
- 不要使用只有一两个字母的参数名
- 不要使用简写，拼出完整的单词

下面列举了一些常用参数名：

```
...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
```

存取方法（Accessor Methods）

存取方法是指用来获取和设置类属性值的方法，属性的不同类型，对应着不同的存取方法规范：

```
// 属性是一个名词时的存取方法范式
- (type)noun;
- (void)setNoun:(type)aNoun;
// 栗子
- (NSString *)title;
- (void)setTitle:(NSString *)aTitle;
// 属性是一个形容词时存取方法的范式
- (NSString *)title;
- (void)setTitle:(NSString *)aTitle;
// 栗子
- (BOOL)isAdjective;
- (void)setAdjective:(BOOL)flag;
// 属性是一个动词时存取方法的范式
- (BOOL)verbObject;
- (void)setVerbObject:(BOOL)flag;
// 栗子
- (BOOL)showsAlpha;
- (void)setShowsAlpha:(BOOL)flag;
```

命名存取方法时不要将动词转化为被动形式来使用：

```
// 正确
- (void)setAcceptsGlyphInfo:(BOOL)flag;
- (BOOL)acceptsGlyphInfo;
// 错误，不要使用动词的被动形式
- (void)setGlyphInfoAccepted:(BOOL)flag;
- (BOOL)glyphInfoAccepted;
```

可以使用 can,should,will 等词来协助表达存取方法的意思，但不要使用 do, 和 does：

```
// 正确
- (void)setCanHide:(BOOL)flag;
- (BOOL)canHide;
- (void)setShouldCloseDocument:(BOOL)flag;
- (BOOL)shouldCloseDocument;
// 错误，不要使用 "do" 或者 "does"
- (void)setDoesAcceptGlyphInfo:(BOOL)flag;
- (BOOL)doesAcceptGlyphInfo;
```

为什么 Objective-C 中不适用 get 前缀来表示属性获取方法？因为 get 在 Objective-C 中通常只用来表示从函数指针返回值的函数：

```
// 三个参数都是作为函数的返回值来使用的，这样的函数名可以使用 "get" 前缀
- (void)getLineDash:(float *)pattern count:(int *)count phase:(float *)phase;
```

命名委托（Delegate）

当特定的事件发生时，对象会触发它注册的委托方法。委托是 Objective-C 中常用的传递消息的方式。委托有它固定的命名范式。

一个委托方法的第一个参数是触发它的对象，第一个关键词是触发对象的类名，除非委托方法只有一个名为 sender 的参数：

```
// 第一个关键词为触发委托的类名
- (BOOL)tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;
// 当只有一个 "sender" 参数时可以省略类名
- (BOOL)applicationOpenUntitledFile:(NSApplication *)sender;
根据委托方法触发的时机和目的，使用 should,will,did 等关键词
- (void)browserDidScroll:(NSBrowser *)sender;
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window; 、
- (BOOL)windowShouldClose:(id)sender;
```

集合操作类方法（Collection Methods）

有些对象管理着一系列其它对象或者元素的集合，需要使用类似“增删查改”的方法来对集合进行操作，这些方法的命名范式一般为：

```
// 集合操作范式
- (void)addElement:(elementType)anObj;
- (void)removeElement:(elementType)anObj;
- (NSArray *)elements;
// 栗子
- (void)addLayoutManager:(NSLayoutManager *)obj;
- (void)removeLayoutManager:(NSLayoutManager *)obj;
- (NSArray *)layoutManagers;
```

注意，如果返回的集合是无序的，使用 NSSet 来代替 NSArray 。如果需要将元素插入到特定的位置，使用类似于这样的命名：

```
- (void)insertLayoutManager:(NSLayoutManager *)obj atIndex:(int)index;
- (void)removeLayoutManagerAtIndex:(int)index;
```

如果管理的集合元素中有指向管理对象的指针，要设置成 weak 类型以防止引用循环。

下面是 SDK 中 NSWindow 类的集合操作方法：

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

命名函数（Functions）

在很多场合仍然需要用到函数，比如说如果一个对象是一个单例，那么应该使用函数来代替类方法执行相关操作。

函数的命名和方法有一些不同，主要是：

- 函数名称一般带有缩写前缀，表示方法所在的框架。
- 前缀后的单词以“驼峰”表示法显示，第一个单词首字母大写。

函数名的第一个单词通常是一个动词，表示方法执行的操作：

```
NSHighlightRect
NSDeallocateObject
```

如果函数返回其参数的某个属性，省略动词：

```
unsigned int NSEventMaskFromType(NSEventType type)
float NSHeight(NSRect aRect)
```

如果函数通过指针参数来返回值，需要在函数名中使用 Get：

```
unsigned int NSEventMaskFromType(NSEventType type)
float NSHeight(NSRect aRect)
```

函数的返回类型是 BOOL 时的命名：

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

命名属性和实例变量（Properties&Instance Variables）

属性和对象的存取方法相关联，属性的第一个字母小写，后续单词首字母大写，不必添加前缀。属性按功能命名成名词或者动词：

```
// 名词属性
@property (strong) NSString *title;
// 动词属性
```

```
@property (assign) BOOL showsAlpha;
```

属性也可以命名成形容词，这时候通常会指定一个带有 is 前缀的 get 方法来提高可读性：

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

命名实例变量，在变量名前加上 _ 前缀（有些有历史的代码会将 _ 放在后面 ），其它和命名属性一样：

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

一般来说，类需要对使用者隐藏数据存储的细节，所以不要将实例方法定义成公共可访问的接口，可以使用 @private ， @protected 前缀。

按苹果的说法，不建议在除了 init 和 dealloc 方法以外的地方直接访问实例变量，但很多人认为直接访问会让代码更加清晰可读，只在需要计算或者执行操作的时候才使用存取方法访问，我就是这种习惯，所以这里不作要求。

命名常量（ Constants ）

如果要定义一组相关的常量，尽量使用枚举类型（ enumerations ），枚举类型的命名规则和函数的命名规则相同：

```
// 定义一个枚举，注意带有 `_` 的名称是不会被使用的
typedef enum _NSMatrixMode {
NSRadioModeMatrix = 0,
NSHighlightModeMatrix = 1,
NSListModeMatrix = 2,
NSTrackModeMatrix = 3
} NSMatrixMode;
```

使用匿名枚举定义 bit map ：

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

使用 const 定义浮点型或者单个的整数型常量，如果要定义一组相关的整数常量，应该优先使用枚举。常量的命名规范和函数相同：

```
const float NSLightGray;
```

不要使用 #define 宏来定义常量，如果是整型常量，尽量使用枚举，浮点型常量，使用 const 定义。 #define 通常用来给编译器决定是否编译某块代码，比如常用的：

```
#ifdef DEBUG
```

注意到一般由编译器定义的宏会在前后都有一个 __ ，比如 __MACH__ 。

命名通知（ Notifications ）

通知常用于在模块间传递消息，所以通知要尽可能地表示出发生的事件，通知的命名范式是：

```
[ 触发通知的类名 ] + [Did | Will] + [ 动作 ] + Notification
```

栗子：

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

注释

读没有注释代码的痛苦你我都体会过，好的注释不仅能让人轻松读懂你的程序，还能提升代码的逼格。注意注释是为了让别人看懂，而不是仅仅你自己。

文件注释

每一个文件都必须写文件注释，文件注释通常包含

- 文件所在模块
- 作者信息
- 历史版本信息
- 版权信息
- 文件包含的内容，作用

一段良好文件注释的栗子：

```
/*********************************************************************************
Copyright (C), 2011-2013, Andrew Min Chang
File name: AMCCommonLib.h
Author: Andrew Chang (Zhang Min)
E-mail: LaplaceZhang@126.com
Description:
This file provide some covenient tool in calling library tools. One can easily
library headers he wants by declaring the corresponding macros.
I hope this file is not only a header, but also a useful Linux library note.
History:
2012-??-??: On about come date around middle of Year 2012, file created as "com
2012-08-20: Add shared memory library; add message queue.
2012-08-21: Add socket library (local)
2012-08-22: Add math library
2012-08-23: Add socket library (internet)
2012-08-24: Add daemon function
2012-10-10: Change file name as "AMCCommonLib.h"
2012-12-04: Add UDP support in AMC socket library
2013-01-07: Add basic data type such as "sint8_t"
2013-01-18: Add CFG_LIB_STR_NUM.
2013-01-22: Add CFG_LIB_TIMER.
2013-01-22: Remove CFG_LIB_DATA_TYPE because there is already AMCDataTypes.h
Copyright information:
This file was intended to be under GPL protocol. However, I may use this librar
in my work as I am an employee. And my company may require me to keep it secret
Therefore, this file is neither open source nor under GPL control.
*********************************************************************************/
```

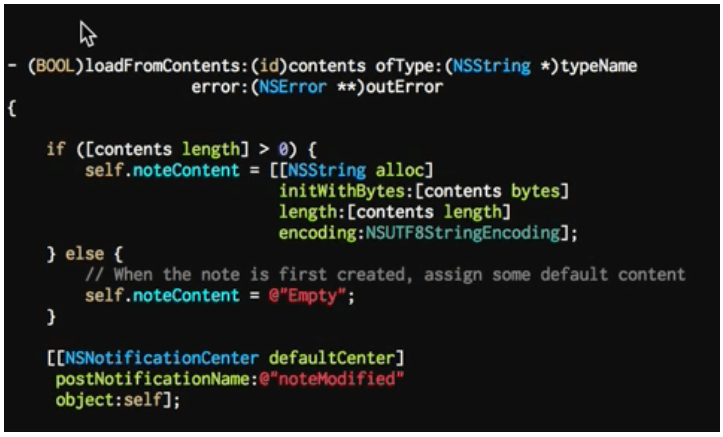
文件注释的格式通常不作要求，能清晰易读就可以了，但在整个工程中风格要统一。

代码注释

好的代码应该是“自解释”（self-documenting）的，但仍然需要详细的注释来说明参数的意义、返回值、功能以及可能的副作用。

方法、函数、类、协议、类别的定义都需要注释，推荐采用 Apple 的标准注释风格，好处是可以在引用的地方 alt+ 点击自动弹出注释，非常方便。

有很多可以自动生成注释格式的插件，推荐使用 VVDocumenter：



```
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
                   error:(NSError **)outError
{

    if ([contents length] > 0) {
        self.noteContent = [[NSString alloc]
                            initWithBytes:[contents bytes]
                            length:[contents length]
                            encoding:NSUTF8StringEncoding];
    } else {
        // When the note is first created, assign some default content
        self.noteContent = @"Empty";
    }

    [[NSNotificationCenter defaultCenter]
     postNotificationName:@"noteModified"
     object:self];
```

一些良好的注释：

```
/**
* Create a new preconnector to replace the old one with given mac address.
* NOTICE: We DO NOT stop the old preconnector, so handle it by yourself.
*
* @param type Connect type the preconnector use.
* @param macAddress Preconnector's mac address.
*/
- (void)refreshConnectorWithConnectType:(IPCConnectType)type Mac:(NSString *)ma
/**
* Stop current preconnecting when application is going to background.
*/
-(void)stopRunning;
/**
* Get the COPY of cloud device with a given mac address.
*
* @param macAddress Mac address of the device.
*
* @return Instance of IPCCloudDevice.
*/
-(IPCCloudDevice *)getCloudDeviceWithMac:(NSString *)macAddress;
// A delegate for NSApplication to handle notifications about app
// launch and shutdown. Owned by the main app controller.
@interface MyAppDelegate : NSObject {
...
}
@end
```

协议、委托的注释要明确说明其被触发的条件：

```
/** Delegate - Sent when failed to init connection, like p2p failed. */
-(void)initConnectionDidFailed:(IPCConnectHandler *)handler;
```

如果在注释中要引用参数名或者方法函数名，使用 || 将参数或者方法括起来以避免歧义：

```
// Sometimes we need |count| to be less than zero.
// Remember to call |StringWithoutSpaces("foo bar baz")|
```

定义在头文件里的接口方法、属性必须要有注释！

编码风格

每个人都有自己的编码风格，这里总结了一些比较好的 Cocoa 编程风格和注意点。

不要使用 new 方法

尽管很多时候能用 new 代替 alloc init 方法，但这可能会导致调试内存时出现不可预料的问题。 Cocoa 的规范就是使用 alloc init 方法，使用 new 会让一些读者困惑。

Public API 要尽量简洁

共有接口要设计的简洁，满足核心的功能需求就可以了。不要设计很少会被用到，但是参数极其复杂的 API 。如果要

定义复杂的方法，使用类别或者类扩展。

#import 和 #include

#import 是 Cocoa 中常用的引用头文件的方式，它能自动防止重复引用文件，什么时候使用 #import ，什么时候使用 #include 呢?

- 当引用的是一个 Objective-C 或者 Objective-C++ 的头文件时，使用 #import
- 当引用的是一个 C 或者 C++ 的头文件时，使用 #include ，这时必须要保证被引用的文件提供了保护域（ #define guard ）。

栗子:

```
#import
#include
#import "GTMFoo.h"
#include "base/basictypes.h"
```

为什么不全部使用 #import 呢? 主要是为了保证代码在不同平台间共享时不出现问题。

引用框架的根头文件

上面提到过，每一个框架都会有一个和框架同名的头文件，它包含了框架内接口的所有引用，在使用框架的时候，应该直接引用这个根头文件，而不是其它子模块的头文件，即使是你只用到了其中的一小部分，编译器会自动完成优化的。

```
// 正确，引用根头文件
#import
// 错误，不要单独引用框架内的其它头文件
#import
#import
```

BOOL 的使用

BOOL 在 Objective-C 中被定义为 signed char 类型，这意味着一个 BOOL 类型的变量不仅仅可以表示 YES(1) 和 NO(0) 两个值，所以永远**不要**将 BOOL 类型变量直接和 YES 比较:

```
// 错误，无法确定 |great| 的值是否是 YES(1) ，不要将 BOOL 值直接与 YES 比较
BOOL great = [foo isGreat];
if (great == YES)
// ...be great!
// 正确
BOOL great = [foo isGreat];
if (great)
// ...be great!
```

同样的，也不要将其它类型的值作为 BOOL 来返回，这种情况下， BOOL 变量只会取值的最后一个字节来赋值，这样很可能会取到 0 （ NO ）。但是，一些逻辑操作符比如 &&,||,! 的返回是可以直接赋给 BOOL 的:

```
// 错误，不要将其它类型转化为 BOOL 返回
- (BOOL)isBold {
return [self fontTraits] & NSFontBoldTrait;
}
- (BOOL)isValid {
return [self stringValue];
}
// 正确
- (BOOL)isBold {
return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}
// 正确，逻辑操作符可以直接转化为 BOOL
- (BOOL)isValid {
return [self stringValue] != nil;
}
- (BOOL)isEnabled {
return [self isValid] && [self isBold];
}
```

另外 BOOL 类型可以和 _Bool,bool 相互转化，但是不能和 Boolean 转化。

使用 ARC

除非想要兼容一些古董级的机器和操作系统，我们没有理由放弃使用 ARC 。在最新版的 Xcode(6.2) 中， ARC 是自动打开的，所以直接使用就好了。

在 init 和 dealloc 中不要用存取方法访问实例变量

当 initdealloc 方法被执行时，类的运行时环境不是处于正常状态的，使用存取方法访问变量可能会导致不可预料的结果，因此应当在这两个方法内直接访问实例变量。

```
// 正确，直接访问实例变量
- (instancetype)init {
self = [super init];
if (self) {
_bar = [[NSMutableString alloc] init];
}
return self;
}
- (void)dealloc {
[_bar release];
[super dealloc];
}
// 错误，不要通过存取方法访问
- (instancetype)init {
self = [super init];
if (self) {
self.bar = [NSMutableString string];
}
return self;
}
- (void)dealloc {
self.bar = nil;
[super dealloc];
}
```

按照定义的顺序释放资源

在类或者 Controller 的生命周期结束时，往往需要做一些扫尾工作，比如释放资源，停止线程等，这些扫尾工作的释放顺序应当与它们的初始化或者定义的顺序保持一致。这样做是为了方便调试时寻找错误，也能防止遗漏。

保证 NSString 在赋值时被复制

NSString 非常常用，在它被传递或者赋值时应当保证是以复制（ copy ）的方式进行的，这样可以防止在不知情的情况下 String 的值被其它对象修改。

```
- (void)setFoo:(NSString *)aFoo {
_foo = [aFoo copy];
}
```

使用 NSNumber 的语法糖

使用带有 @ 符号的语法糖来生成 NSNumber 对象能使代码更简洁：

```
NSNumber *fortyTwo = @42;
NSNumber *piOverTwo = @(M_PI / 2);
enum {
kMyEnum = 2;
};
NSNumber *myEnum = @(kMyEnum);
```

nil 检查

因为在 Objective-C 中向 nil 对象发送命令是不会抛出异常或者导致崩溃的，只是完全的 “ 什么都不干 ”，所以，只在程序中使用 nil 来做逻辑上的检查。

另外，不要使用诸如 nil == Object 或者 Object == nil 的形式来判断。

```
// 正确，直接判断
if (!objc) {
...
}
// 错误，不要使用 nil == Object 的形式
if (nil == objc) {
...
}
```

属性的线程安全

定义一个属性时，编译器会自动生成线程安全的存取方法（ Atomic ），但这样会大大降低性能，特别是对于那些需要频繁存取的属性来说，是极大的浪费。所以如果定义的属性不需要线程保护，记得手动添加属性关键字 nonatomic 来取消编译器的优化。

点分语法的使用

不要用点分语法来调用方法，只用来访问属性。这样是为了防止代码可读性问题。

```
// 正确，使用点分语法访问属性
NSString *oldName = myObject.name;
myObject.name = @"Alice";
// 错误，不要用点分语法调用方法
NSArray *array = [NSArray arrayWithObject:@"hello"];
NSUInteger numberOfItems = array.count;
array.release;
```

Delegate 要使用弱引用

一个类的 Delegate 对象通常还引用着类本身，这样很容易造成引用循环的问题，所以类的 Delegate 属性要设置为弱引用。

```
/** delegate */
@property (nonatomic, weak) iddelegate;
```


上一篇：实时显示iOS编写UI代码效果
下一篇：项目管理：CocoaPods建立私有仓库

相关资讯
Xcode 6 插件开发入门
iOS Crash文件的解析（一）
GCD使用经验与技巧浅谈
10个iOS开发常见错误，你中招了吗

Scrapy+Flask+Mongodb+Swift 开发全攻略（1）
iOS开发的一些小技巧
源码推荐（4.24）：iOS开发Categoy(扩展)大
Java转iOS-第一个项目总结(2)：遇到问题和解决
从Java转iOS第一个项目总结
苹果上线ResearchKit开发页面

文章评论 (1)

Azzinoth 2015-05-09 17:34:50

非常完整，非常规范，解决了很多规范性问题。赞！

支持(0) 回复(0)

对这篇文章有什么感想，写一下吧……

发表评论