1. 格式化代码

**1.1.** 空格 **VS** 制表符

只允许使用空格，将编辑器设置为1个TAB = 4个字符缩进

**1.2.** 每行的长度

每行最多不得超过120个字符，建议通过 “Xcode => Preferences => Text Editing => 勾选Show Page Guide” 来设置提醒。

**1.3.** 方法的声明和定义

在 – 或 + 和返回值之间留1个空格，方法名和第一个参数间不留空格。如：

- (void)doSomethingWithString:(NSString \*)theString

{

...

}

当参数过长时，每个参数占用一行，以冒号对齐。如：

- (void)doSomethingWith:(GTMFoo \*)theFoo

rect:(NSRect)theRect

interval:(float)theInterval

{

...

}

如果方法名比参数名短，每个参数占用一行，至少缩进4个字符，且为垂直对齐（而非使用冒号对齐）。如：

- (void)short:(GTMFoo \*)theFoo

longKeyword:(NSRect)theRect

evenLongerKeyword:(float)theInterval

{

...

}

**1.4.** 方法的调用

调用方法沿用声明方法的习惯。例外：如果给定源文件已经遵从某种习惯，继续遵从那种习惯。

所有参数应在同一行中，或者每个参数占用一行且使用冒号对齐。如：

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

或：

[myObject doFooWith:arg1

name:arg2

error:arg3];

和方法的声明一样，如果无法使用冒号对齐时，每个参数一行、缩进4个字符、垂直对其（而非使用冒号对齐）。如：

[myObj short:arg1

longKeyword:arg2

evenLongerKeyword:arg3];

**1.5. @public** 和 **@private**

@public 和 @private使用单独一行，且缩进1个字符。

**1.6. Protocol**的格式

类型标示符、代理名称、尖括号间不留空格。

该规则同样适用于：类声明、实例变量和方法声明。如：

@interface MyProtocoledClass : NSObject<NSWindowDelegate>

{

@private

id<MyFancyDelegate> \_delegate;

}

- (void)setDelegate:(id<MyFancyDelegate>)aDelegate;

@end

**1.7.** 方法和类中左大括号的位置

方法和类定义中的左大括号统一另起一行，这样虽然降低了代码的紧凑性，但是会使代码结构更为清晰。

有一种例外的情况：Block语法块的左括号不必另起一行。

**1.8.** 判断和循环语句的格式

这些规则与公司的C\C++编程规范保持一致，原因不必赘述。

if、while、for、switch等关键字与后边的条件之间留一个空格，左大括号另起一行，如下：

if (result == 200)

{

…

}

while (need)

{

…

}

for (int i = 0; i < 100; ++i)

{

…

}

switch (value)

{

case 1:

…

break;

case 2:

…

break;

default:

…

break;

}

2. 命名规范

**2.1.** 标识符

标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。采用驼峰式写法，首字符小写，后续每个单词首字母大写。

命名中若使用特殊约定或缩写，则要有注释说明。

自己特有的命名风格，要自始至终保持一致，不可来回变化。

对于变量命名，禁止取单个字符，例如i、j、k等，但i、j、k作局部循环变量是允许的。

成员变量定义时使用“m\_”作为前缀，但是在暴露属性的时候，将“m\_”前缀去掉，例如：

NSString \*m\_name;

NSInteger m\_value;

// 类成员暴露为属性，在头文件中的写法

@property(nonatomic, retain) NSString \* name;

@property(nonatomic, retain) NSInteger value;

// 类成员暴露为属性，在实现文件中的写法

@synthesize name = m\_name;

@synthesize value = m\_value

全局变量使用“g\_”作为前缀，例如：

NSInteger g\_totalCount;

请不要使用下划线”\_”作为前缀来表示私有的属性，尤其是不要在类方法中使用。因为苹果公司保留使用这种方式，如果第三方再使用，就有可能会导致名称空间冲突。有可能

在无意中用自己的方法覆盖了一个已经存在的私有方法，这样做将会带来灾难性的后果。

混合编程中C/C++部分沿用C/C++编程规范中对应部分。

**2.2.** 类和方法

类名（及其category name）和协议名使用驼峰式写法，首字母大写，后续每个单词的首字母大写。

在开发静态库代码时，必须在类名之前使用特定前缀，以避免和其他静态库中的类命名冲突。如：MVStaticConfig。

类的方法名的首字母小写，后续每个单词的首字母大写。方法的参数使用相同的规则。

get和set在ObjC中有明确含义，在非自定义的获取器和设置器方法中，避免使用get和set开头的方法名称，以免造成和系统生成的方法冲突而歪曲了代码的意图。

方法命名应体现出该方法的内存管理信息，如果方法内部创建了对象并没有释放，需要由调用者释放，则方法名以alloc、create、new等单词开头，表示其内部创建了未释放的对象。

3. 注释规范

注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

**3.1.** 代码和注释的一致性

边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

注释的内容要清楚、明了，含义准确，防止注释二义性。

避免在注释中使用缩写，特别是非常用缩写。

**3.2.** 注释方式的选择

函数体中不允许使用块注释，只能使用行注释。

函数头说明及文件头说明可使用块注释。

**3.3.** 注释在代码中的位置

注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。示例：

// active statistic task number

#define MAX\_ACT\_TASK\_NUMBER 1000

#define MAX\_ACT\_TASK\_NUMBER 1000 // active statistic task number

数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。示例：

// sccp interface with sccp user primitive message name

enum SCCP\_USER\_PRIMITIVE

{

N\_UNITDATA\_IND, // sccp notify sccp user unit data come

N\_NOTICE\_IND, // sccp notify user the No.7 network can not

// transmission this message

N\_UNITDATA\_REQ, // sccp user's unit data transmission request

};

**3.4.** 全局变量的注释要求

全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。示例：

// The ErrorCode when SCCP translate

// Global Title failure, as follows // 变量作用、含义

// 0 － SUCCESS 1 － GT Table error

// 2 － GT error Others － no use // 变量取值范围

// only function SCCPTranslate() in

// this modual can modify it, and other

// module can visit it through call

// the function GetGTTransErrorCode() // 使用方法

BYTE g\_GTTranErrorCode;

**3.5.** 注释与所描述内容进行同样的缩排

如此可使程序排版整齐，并方便注释的阅读与理解。如下例子，排版不整齐，阅读稍感不方便。

void example\_fun( void )

{

// code one comments

CodeBlock One

// code two comments

CodeBlock Two

}

应改为如下布局。

void example\_fun( void )

{

// code one comments

CodeBlock One

// code two comments

CodeBlock Two

}

**3.6.** 文件头与函数头的注释格式

文件头使用如下形式注释，要包含如下必要信息：

文件简要说明

作者、创建时间

版权信息

样例如下：

//

// FrameworkX.h

// FrameworkX is an universal framework.

//

// Created by leonard on 12/13/12.

// Copyright (c) 2012 HUAWEI. All rights reserved.

//

函数头注释分为两种情况：如果是public函数，则使用appledoc风格注释格式，如下：

/\*\*

函数功能的简要说明，要剪短概括

函数功能的详细说明，可分多行写

@param 参数1名称 参数1说明

@param 参数2名称 参数2说明

@return 返回值说明

\*/

- (int) funcSampleWithObj1:(id)obj1 withObj2:(id)obj2;

如果是private函数，则在文件头使用简单的行注释即可。

// 本函数的作用是。。。。

- (void) privateFuncSample

{

…

}

4. 内存规范

**4.1.** 初始化

在类的初始化方法中，不要将变量初始化为“0”或“nil”，那是多余的，内存中所有的新创建的对象（isa除外）都是0，所以不需要重复初始化为“0”或“nil”。

与之相对的，在函数中，对于函数中声明的临时变量，必须在声明的同时进行初始化，

初始化的取值根据当前逻辑选择适合的取值。

**4.2. assign/ retain/ copy**属性的使用**.**

assign: 在进行赋值时内存引用计数不增加；用于基础类型，如NSInteger、NSUInteger、CGFloat、int、float、double、char等；

retain: 在进行赋值时会将入参retain一次，内存引用计数加1。在被赋值的变量使用完后，必须将其release掉。在对象需要保留时必须使用retain属性；

copy: 在进行赋值时进行深拷贝，它会将入参拷贝一份。通常用于异步操作或多线程。另外，对于Block类型的属性，必须使用copy。

**4.3.** 禁止对**delegate**进行**retain**

因为delegate中，一般会保存caller的实例变量；对delegate获取所有权，会造成delegate及caller之间的相互引用，双方都无法释放，造成内存泄漏。

@interface FooBar : NSObject

{

id m\_delegate;

// code continues...

}

@property(nonatomic, assign) id delegate; //cannot retain here

// code continues...

@end

**4.4.** 避免显式的调用 **+new** 方法

禁止直接调用 NSObject 的类方法 +new，也不要在子类中重载它。使用alloc和init方法。

**4.5. alloc**临时对象之后**,**一定要使用**release**将其释放

如果时alloc的一个临时变量,则在临时变量用完后立即释放。如:

customView = [[UICustomAlertView alloc] initDoubleWithMsg: @"确定要停止播放

么 ?" delegate: nil];

[customView show];

[customView release];

customView = nil;

**4.6. dealloc**的顺序要与**init**中的顺序相反

一般dealloc中释放对象的顺序应与init中顺序相反，[super dealloc]应当在子类dealloc方法的最后调用。

另外，在dealloc方法中，对于成员变量的释放，不要使用属性置空的方式，成员变量release之后，也不需要置为nil。

- (void)dealloc

{

self.customView = nil; //×！不要使用属性置空的方式

[m\_customView release]; // √ ！正确的写法

m\_customView = nil; // ×！对象释放之后，变量无需置空，没意义

[super dealloc];

}

**4.7.** 子线程和**NSAutoreleasePool**

如下方法创建了子线程：

[NSThread detachNewThreadSelector:@selector(doStuff) toTarget:self withObject:nil];

线程的工作方法中需要创建NSAutoreleasePool，并在最后释放，否则会导致内存泄露。

- (void)doStuff

{

NSAutoreleasePool \*pool = [[NSAutoreleasePool alloc] init];

// Do stuff

…

[pool release];

}

**4.8.** 处理**didReceiveMemoryWarning**

在移动开发中，当整个系统的内存占用到达临界值时，应用程序可以会被杀死，或出现一些异常的现象。因此，每个开发者都需要即时处理内存告警，释放不必要的资源。

UIViewController类提供了一个方法didReceiveMemoryWarning，当系统内存吃紧时，ViewController类会的此方法会被调用到。

每个ViewController类需要覆写didReceiveMemoryWarning方法，释放不必要的资源。

**4.9. autorelease**的资源需要尽早释放

对于autorelease的变量，只有在pool释放时才会释放。

因为移动设备上内存不会太多，因此，需要尽量早点释放不必要的资源，特别是在一个循环中或临时资源占用内存比较大时。

objective-c中支持多层级的autoreleasePool, 每一层级的pool在release时会释放自己范围内的autorelease变量。因此，在开发过程中，尽量在可控的范围内，使用创建并释放pool，例如，在一个大循环中创建临时的释放池：

for (int i = 0; i < 10000; ++i)

{

NSAutoreleasePool \*pool = [[NSAutoreleasePool alloc] init];

// Code benefitting from a local autorelease pool.

…

[pool release];

}

**4.10.** 引用计数与**ARC**

同一个工程的代码，只使用引用计数或ARC机制中的一种，不允许同时使用，以避免因不同的内存使用方式降低代码的可维护性。

有一种例外的情况：如果在一个非ARC工程中引用了一份基于ARC的开源代码，或者在一个ARC工程引用了一份非ARC的开源代码，则不必刻意保持一种机制，可通过编译开关对

不同的源文件进行单独控制。

5. 其他注意事项

**5.1. #import VS #include**

使用 #import 引入Ojbective-C和Ojbective-C++头文件，使用 #include 引入C和C++头文件。

如果是在某个类的头文件中要引用另一个类，优先使用@class语法，而不要使用#import引入另一个类的头文件，以避免头文件过度包含导致的循环引用。

import根框架（root frameworks），而非各单个文件。虽然有时我们仅需要框架（如Cocoa 或 Foundation）的某几个头文件，但引入根文件编译会更快。因为根框架（root frameworks）一般会预编译，所以加载会更快。再次强调：使用 #import 而非 #include 来引入Objective-C框架。如：

#import <Foundation/NSArray.h> // 禁止

#import <Foundation/NSString.h>// 禁止

...

#import <Foundation/Foundation.h> // 推荐

**5.2. BOOL**陷阱

将int值转换为BOOL时应特别小心，避免直接和YES比较。

Objective-C中，BOOL被定义为unsigner char，这意味着除了 YES（1）和NO（0）外它还可以是其他值。禁止将int直接转换（cast or convert）为BOOL。

常见的错误包括：将数组的大小、指针值或位运算符的结果转换（cast or convert）为BOOL，因为该BOOL值的结果取决于整型值的最后一位。

将整型值转换为BOOL的方法：使用三元运算符返回YES/NO，或使用位运算符（&&，||，!）。

BOOL、\_Bool和bool之间的转换是安全的，但是BOOL和Boolean间的转换是不安全的，所以将Boolean堪称整形值

在Objective-C中，只允许使用BOOL。如：

// 禁止 - (BOOL)isBold

{

return [self fontTraits] & NSFontBoldTrait;

}

- (BOOL)isValid

{

return [self stringValue];

}

// 正确 - (BOOL)isBold

{

return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;

}

- (BOOL)isValid

{

return [self stringValue] != nil;

}

- (BOOL)isEnabled

{

return [self isValid] && [self isBold];

}

禁止直接将BOOL和YES/NO比较，如：

// 禁止

BOOL great = [foo isGreat];

if (great == YES)

...

// 正确

BOOL great = [foo isGreat];

if (great)

...

**5.3.** 给**nil**对象发消息

一般情况下给nil对象发消息相当于空操作，因此无需在发消息之前检查对象是否为nil，但要注意的是该方法的返回值不允许是struct类型的变量，否则会出现返回随机值的

情况。

// 禁止

CGRect rect = [obj fetchRect];

// 正确

[string1 replaceCharactersInRange:NSMakeRange(0, 4) withString:@"That"];

**5.4.** 不要在非**main**线程中调用**UI**控件

UI的操作只能在主线程执行，否则会引入很多未定义错误。

**5.5.** 调用未知对象的方法前，需要进行兼容性检查

调用conformsToProtocol检查对象是否兼容对应的protocol。

if ([req.m\_ReqBody conformsToProtocol:@protocol(HttpReqProtocol)])

{

sBody = [req.m\_ReqBody performSelector:@selector(httpBodyData)];

}

调用respondsToSelector检查是否可以执行对应的方法。

if ([viewDelegate respondsToSelector:@selector(stopMove)])

{

[viewDelegate stopMove];

}

**5.6.** 类的私有变量和方法不要暴露在头文件中

在面向对象的设计中，对象应只公开所必要的接口，私有接口应通过private来限定，表示变量或方法不对外公开，只用于类的内部实现。

objective-c在此基础上更进了一步，通过Category和Extension的机制，可以把私有的变量或方法写在.m文件中。这样类的接口定义更清晰，而且可以减少编译时间。

下面的例子中，私有的方法和变量写到了实现文件中。

MyClass.h文件:

@interface MyClass : NSObject

- (float)value;

@end

MyClass.m文件：

#import "MyClass.h"

@interface MyClass() //declare extension

{

float m\_value;

}

- (void)setValue:(float)newValue;

@end

@implementation MyClass

- (float)value

{

return m\_value;

}

- (void)setValue:(float)newValue

{

m\_value = newValue;

}

@end