iOS代码规范

iOS代码规范

一、文件书写规范

1、设置代码每一行的合适长度

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。

二、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。

1、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 too
ls. One can easily include
        library headers he wants by declaring the corresponding macros.
            I hope this file is not only a header, but also a useful Lin
ux library note.

    History:
        2012-??-??: On about come date around middle of Year 2012, file
```

```
created as "commonLib.h"
        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 AM
CDataTypes.h

    Copyright information:
            This file was intended to be under GPL protocol. However, I
may use this library
        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 contro
l.

*************************************************************************
********/
```

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。

> Apple支持的注释常用关键词：
> @brief 是标题
> @discussion 是具体说明
> @code 代码
> @see 同上
> @param 参数
> @return 返回

2、代码注释：`option + command + /`

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。

> 如果是双斜线 `"//"` 注释，需要在斜线后面加上一个空格

一些良好的注释：

```
/**
 *  Create a new preconnector to replace the old one with given mac addr
ess.
 *  NOTICE: We DO NOT stop the old preconnector, so handle it by yoursel
f.
 *
 *  @param type       Connect type the preconnector use.
 *  @param macAddress Preconnector's mac address.
 */
- (void)refreshConnectorWithConnectType:(IPCConnectType)type mac:(NSStri
ng *)macAddress;

/**
 *  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 *)cloudDeviceWithMac:(NSString *)macAddress;

// A delegate for NSApplication to handle notifications about app
// launch and shutdown. Owned by the main app controller.
@interface MyAppDelegate : NSObject {
  ...
}
@end

// A Method
- (void)someMethod;
```

3、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！

4、pragma的使用

善用使用 `pragma` 的标记，把method按照使用类型分类，如一个controller从上到下有类本身的生命周期的method，依次往下是外部调用的method，delegate的method，私有方法method等，将不同的API进行归类，增加定位方法的效率。

三、命名规范

1、命名基本规范

1.1、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
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
rect    == Rectangle
Rep     == Representation (used in class name such as NSBitmapImageRep)
temp    == Temporary
vert    == Vertical
```

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
```

1.2、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` 。

1.3、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公共方法使用前缀，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，命名以>=3个字母作为前缀，因为苹果保留了所有2个字母的前缀命名，指不定就会与其冲突。

2、文件名命名规范

2.1、文件名**UXIN_**开头标注通用类库

如果是跨项目通用类库，请用 `UXIN_` 开头，例如通用网络库 `UXIN_AFNetWork` 。

2.2 各自业务类库，用各自业务标识开头

比如二手车业务，用 `YXU_` 开头，如车市列表页使用 `YXU_CarGridVC` 文件名。

3、方法命名规范（Methods）

3.1、方法名

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。
- 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:(BOO
L)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 an
dTypes:(NSArray *)fileTypes;

// 正确，使用"and"来表示两个相对独立的操作
- (BOOL)openFile:(NSString *)fullPath withApplication:(NSString *)appNam
e andDeactivate:(BOOL)flag;
```

3.2、方法参数命名

命名方法参数时要考虑如下规则:

- 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它是指针
- 不要使用只有一两个字母的参数名，如one，two，……
- 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
```

3.3、私有方法命名

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，如基于你公司的名称，或工程的名称，并以 `“XX_”` 形式给出。比如你的工程名为 `"Byte Flogger"` ，那

么就可以是 “BF_addObject:”，”yxu_thisIsPrivateMethod“。

4、存取方法命名规范（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
- (BOOL)isAdjective;
- (void)setAdjective:(BOOL)flag;
// 栗子
- (BOOL)isEditable;
- (void)setEditable:(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;
```

4.1、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;
```

4.2、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 *)p
hase;
```

5、函数命名规范（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`：

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

- 函数的返回类型是BOOL时的命名：

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

6、委托命名规范（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 *)filenam
```

```
e;

// 当只有一个"sender"参数时可以省略类名
- (BOOL)applicationOpenUntitledFile:(NSApplication *)sender;
```

用于表示已经执行或将要执行某些操作可使用 `did` 或 `will` ，询问委托对象可否执行某操作的方法名最好使用 `should` ：

```
- (void)browserDidScroll:(NSBrowser *)sender;

- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window;、

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

7、属性和实例变量命名规范（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，不能以 `new` 为前缀。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。

8、常量命名规范（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。 建议使用 `NS_ENUM` 和 `NS_OPTIONS` 宏来定义枚举类型，参见官方的 Adopting

Modern Objective-C 一文：

```
// 定义一个枚举
typedef NS_ENUM(NSInteger, NSMatrixMode) {
    NSRadioModeMatrix,
    NSHighlightModeMatrix,
    NSListModeMatrix,
    NSTrackModeMatrix
};
定义bit map：

typedef NS_OPTIONS(NSUInteger, NSWindowMask) {
    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;
```

9、宏定义

- 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
  ```

- 严禁项目中滥用宏。重复定义或定义同名宏不同值会引发难以跟踪排查的问题，增加项目维护成本。也不能为了省事而引入某一个宏所在的头文件，导致引入非必要代码造成编译时间累加等问题。
- 能使用常量和函数定义的数据坚决不能使用宏，宏定义只能在“迫不得已”时才去使用。

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

10、通知命名规范（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
```

11、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，所以通常使用动名词(`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。

12、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` 。

四、代码格式

1、方法的书写

一个典型的Objective-C方法应该是这样的：

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

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。
考虑到OC较长的函数名和苹果SDK代码的风格，建议将大括号放在行末。

如果一个方法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<IPCConnectHandlerDelegate>)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 {
    ...
}
```

2、方法调用

方法调用的格式和书写差不多，可以按照方法的长短来选择写在一行或者分成多行：

```
// 写在一行
[myObject objectWith:arg1 name:arg2 error:arg3];

// 分行写，按照':'对齐
[myObject objectWith: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 objectWith:arg1 name:arg2
               error:arg3];
[myObject objectWith:arg1
                name:arg2 error:arg3];
```

```
// 错误，按照':'来对齐，而不是关键字
[myObject objectWith:arg1
           name:arg2
           error:arg3];
```

3、协议（Protocols）书写规范

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()<IPCPreconnectorDelegate>` ，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 <NSWindowDelegate> {
 @private
    id<MyFancyDelegate> _delegate;
}

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

4、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 `“,”` 或者 `“:”` 后面输入一个空格：

```
// 正确，“,”或者“:”后面加空格
NSArray *array = @[[foo description], @"Another String", [bar descriptio
n]];
NSDictionary *dict = @{NSForegroundColorAttributeName: [NSColor redColor
]};

// 不正确，可读性差
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':'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（键值对较多时建议使用）
NSDictionary *option2 = @{
  NSFontAttributeName:            [NSFont fontWithName:@"Arial" size:12]
,
  NSForegroundColorAttributeName: fontColor
};

// 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，Value对齐而不是冒号
NSDictionary *stillWrong = @{
  AKey      : @"b",
  BLongerKey: @"c",
};
```

5、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);
```

五、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。

1、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。

2、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。

3、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 <Cocoa/Cocoa.h>
#include <CoreFoundation/CoreFoundation.h>
#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。

4、引用框架的根头文件

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 <Foundation/Foundation.h>

// 错误，不要单独引用框架内的其它头文件
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
```

若引用的是一个框架/库内的文件，切记一定要引入框架/库名，指定文件所在的类库，这可以使得编译器能优先从缓存查找编译文件(.o)，减少编译时间，增加开发效率：

```
// 正确
#import <Foundation/Foundation.h>
```

```
// 错误
#import <Foundation.h>
```

5、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。

6、在init和dealloc中不要用存取方法访问实例变量

当 `init dealloc` 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];
}
```

7、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。

8、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];
}
```

9、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。

10、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) id <IPCConnectHandlerDelegate> delegate;
```

六、日常注意：

1、字典、数组的定义

诸如 `@{}` 这种方式实现， `@{@"key": value }` ， `value` 必须判定是否为空，为空必crash！
诸如 `@[]` 这种方式实现， `@[value, nil]` ， `value` 必须判定是否为空，为空必crash！

2、调试代码禁止上传到代码仓库

开发大忌：调试代码上传服务器
`#if define 0` 也不可以，个别情况下 `#if DEBUG 、TEST` 可以

3、区分接口调用顺序

接口拆分后，使用中涉及多个接口的，注意调用是否需要区分顺序，刷新数据是否跟接口调用顺序有关系