iOS 软件开发代码规范

作者：李世平

微信：lishiping83118274

目　　录

前　言

本规范针对于 iOS 的 Object-C 开发语言。对于 iOS 开发者来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。

我本人是一个非常热爱 iOS 开发的开发者，之所以写这篇命名规范是因为去过小公司，来过大公司，对公司开发的流程有不同的见解，很多公司在开发的时候都是为了追求效率，因为需求来的太急，所以也没有时间顾虑代码的结构了，就是急匆匆的开发，测试，修 bug，对付能上线就能休息了。所以导致某些项目，在以后的维护中，不容易维护。

每一个开发者都是一个从懵懂到成熟的过程，记得刚刚学习开发的时候，也不知道开发规范，主要也是项目太小，后来进入公司，发现一个项目要做的东西很多，参与开发的人也很多，人多了之后，每个人写的风格都不一样，这样一个项目中有很多风格的代码，很多逻辑不统一的代码，当一群人集体开发的时候，就会有这样那样的问题，尤其是，来了就维护一个旧的项目的时候，代码都是别人写的，如果写的很多，看着太费时间，而且有时候找不到，下面我将要讲解一下，iOS 开发在整个大项目上遵循软件工程的开发流程，以及图形化视图软件的 MVC 模式怎么更好的理解并学习。

我接触的一个项目，很多人写的代码，就是把很多可以抽离的复杂的 View 模块依然写在 VC 类里面，也许是需求太急，不顾虑代码质量了，还有把整个数据的获取，转换成模型，过滤，以及判断都写在了 VC 里面，这样的 view 和 model 都写在了 VC 里面根本没体现出 MVC 模式的好处，导致一个 VC 类.m 文件代码冗余度太高，因为我之前跟过一个技术经理，他有代码洁癖，每次 pull request 提交代码，他都给你挑毛病，开始也是不厌其烦，后来养成习惯也就好了，所以这种洁癖传染给了我们下面的开发者，下面的内容我要介绍怎么能更好的利用 MVC 模式，让你写的代码，可读性，可维护性更强。美观的代码看着也让人感觉到舒服。

1 指导原则

【原则 1-1】首先是为人编写程序，其次才是计算机。

说明：这是软件开发的基本要点，软件的生命周期贯穿产品的开发、测试、生产、用户使用、版本升级和后期维护等长期过程，只有易读、易维护的软件代码才具有生命力。

【原则 1-2】保持代码的简明清晰，避免过分的编程技巧。

说明：简单是最美。保持代码的简单化是软件工程化的基本要求。不要过分追求技巧，否则会降低程序的可读性。

【原则 1-3】编程时首先达到正确性，其次考虑效率。

说明：编程首先考虑的是满足正确性、健壮性、可维护性、可移植性等质量因素，最后才考虑程序的效率和资源占用。

为保证代码的可靠性，编程时请遵循如下基本原则，优先级递减：

- 正确性，指程序要实现设计要求的功能。
- 稳定性、安全性，指程序稳定、可靠、安全。
- 可测试性，指程序要方便测试。
- 规范/可读性，指程序书写风格、命名规则等要符合规范。
- 全局效率，指软件系统的整体效率。
- 局部效率，指某个模块/子模块/函数的本身效率。
- 个人表达方式/个人方便性，指个人编程习惯。

【原则 1-4】编写代码时要考虑到代码的可测试性。

说明：不可以测试的代码是无法保障质量的，开发人员要牢记这一点来设计、编码。实现设计功能的同时，要提供可以测试、验证的方法。

【原则 1-5】函数（方法）是为一特定功能而编写，不是万能工具箱。

说明：方法是一个处理单元，是有特定功能的，所以应该很好地规划方法，不能是所有东西都放在一个方法里实现

【原则 1-6】鼓励多加注释。

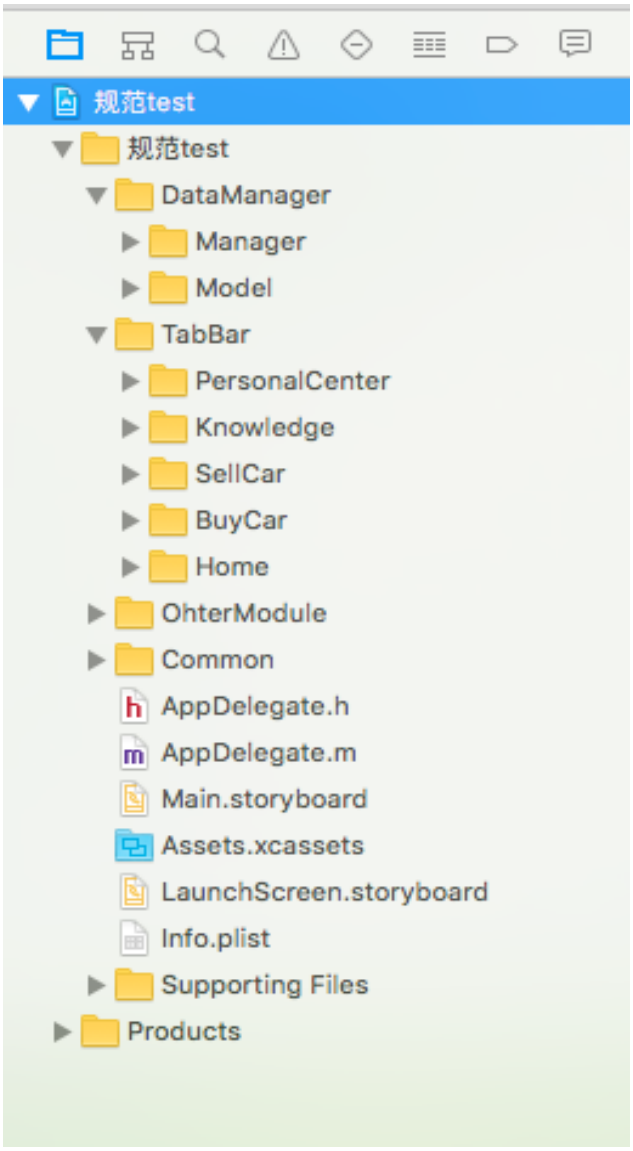
2 布局

程序布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性、可维护性。更重要的是，统一的程序布局和编程风格，有助于提高整个项目的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。因此，统一的、良好的程序布局和编程风格不仅仅是个人主观美学上的或是形式上的问题，而且会涉及到产品质量，涉及到个人编程能力的提高，必须引起大家重视。

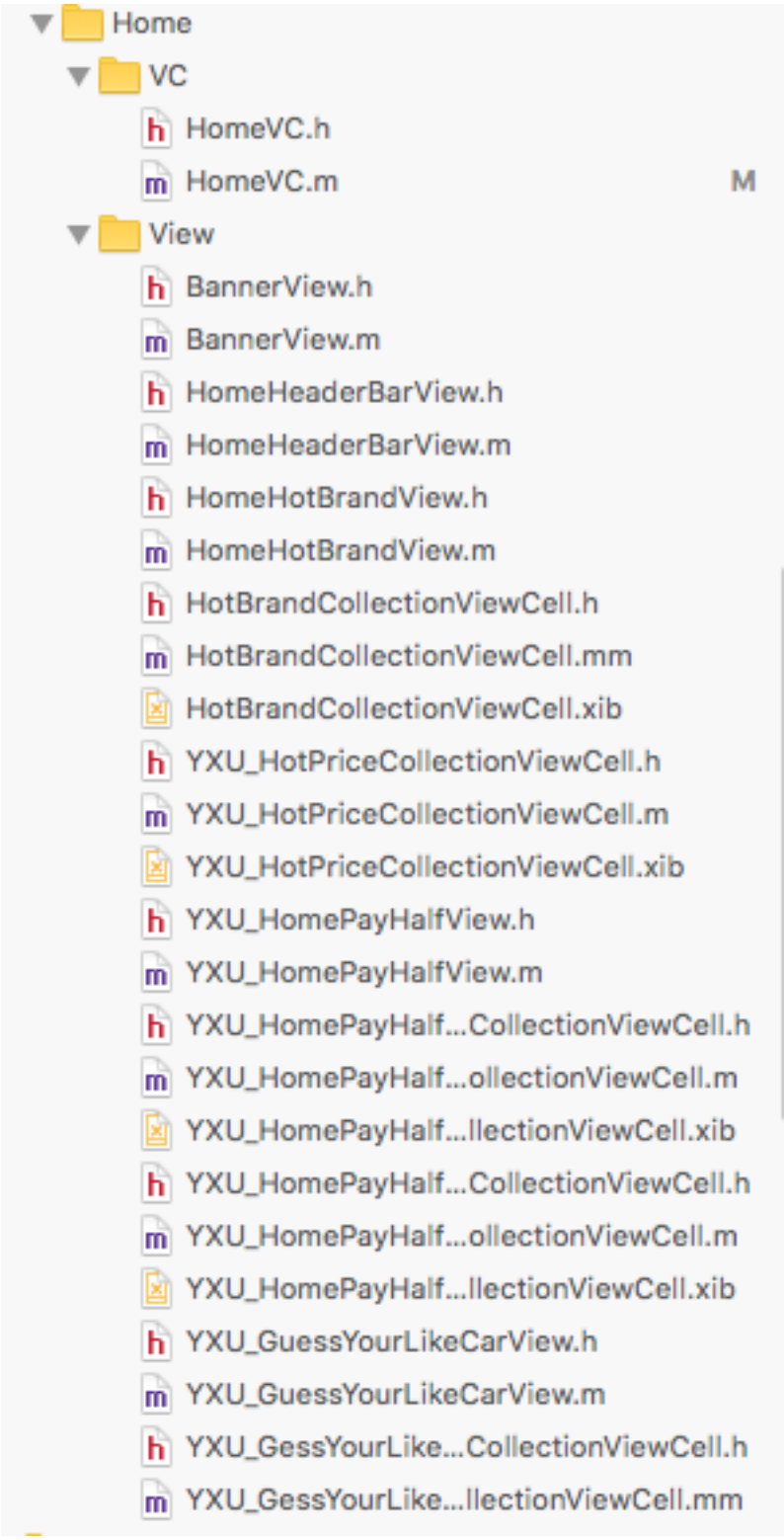
2.1 文件布局

【规则 2-1-1】遵循有逻辑的布局来排版文件目录和顺序。

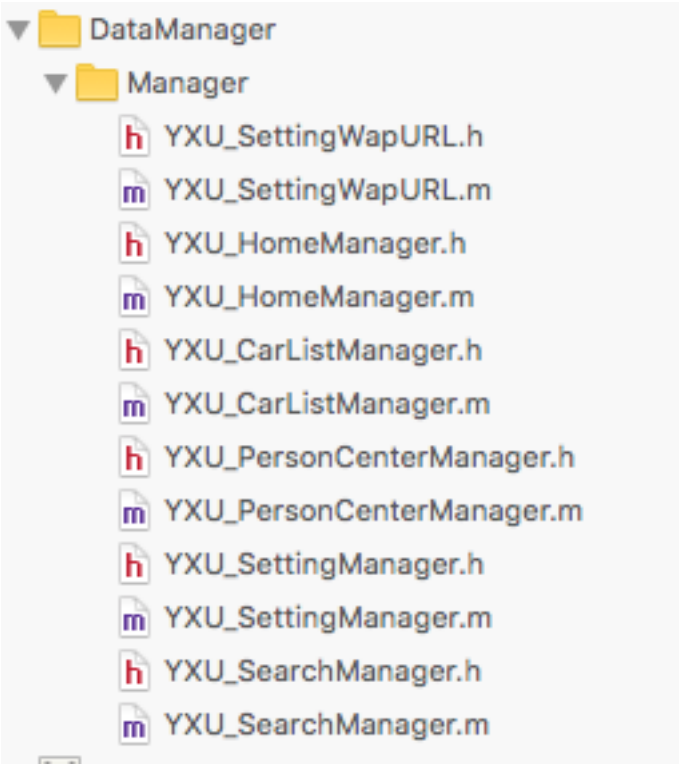
如图 2-1-1 所示为文件目录格式



如图 2-1-2 所示为一个 VC 控制整个复杂页面的 view 与数据的结合



如图 2-1-3 所示为数据的请求类放在了一起，方便查找



说明：以上截图 2-1-1 为文件目录排布样式，为什么要这么分，希望上面截图里面的文件夹是实体文件夹，不要只是　　一个 Group，避免在 Show In Finder 的时候指到莫名其妙的一个文件夹，下面讲一下好处：

好处 1：以上截图 2-1-1 至 2-1-3 显示的是文件的存放逻辑，我解释一下这样的好处，如果一个陌生的开发者接触一个旧项目的时候，去维护旧项目，怎么下手去找到你想要看的页面数据呢，首先我是这么做的，先找到当前 application 的 Window.rootViewController 是什么，如果是 TabBarVC,再去看 TabBar 的每一个 Tab 是什么，找到当前的 VC，从 VC 入口去查看 View 和 dataRequest，（为什么要这么做，有人会说，你直接找到文件目录的 VC 就行了啊，由于每个人编码风格不一样，你直接找到的 VC 不一定真是项目里面的你想要找的 VC，要从软件的根部入口才更准确）。

好处 2：如图 2-1-1 所示 Tabbar 模块包括四个 tab 模块分别对应四个页面，DataManager 管理数据，OtherModule 存放一些不在 tab 里面的模块，如 Push，Camera 等不在 tab 里面的模块，Common 存放一些公共类，类别，类扩展，以及接口定义类。也可以按照自己的意愿去分，但是最好把 VC 和 DataManager 区分开，以下会介绍好处。

好处 3：尽量用一个 VC 作为当前页面的入口 VC，当前 VC 不要继承一个带有复杂 UI 的 VC 类，而只是一个简单的 VC 类（加入了一些需要自定义控制 VC 的代码的 VC 类）。

好处 4：当要修改 VC 时候，首先找到 VC 之后，想要修改 UI，就去该 VC 找到那个复杂的自定义 View 去改，想要修改数据就去复杂的 data request 去改，这样是不是很方便，不用都在 VC 里面改，快速定位目标问题。

好处 5：DataManager 管理数据的文件夹，里面又分为 Model（里面存放一些 JSON 转 model 的数据模型类），Manager（里面存放一些数据请求类，并以规范的请求接口注释，

并把所有的关于数据的过滤，转 model 等都放在数据请求的实现里面，这样就很好的把 Model 与 VC 分开，最好不要在 VC 里面再加入更多的 model 处理，体现 MVC 模式），同时把数据的获取放在子线程中执行，网络访问不会卡主线程，而且网络访问速度快，优化了整个软件的流畅度，当数据接收到再传给主线程去更新 UI。

好处 6：数据和 VC，View 分开，在代码重用方面更具有可重用行，比如同样的请求数据接口在 iPhone 端和 iPad 端都可以重用，而把数据请求写在 VC 里面就不能更好的为两个平台端重用，因为代码的耦合性太高，没法重用，代码要遵循高内聚，低耦合的逻辑结构。

好处 7：整个 VC 控制着数据请求和 View 的结合，在开发和查找问题等方面更体现优势，如果 bug 出现的是数据的问题，就去数据获取实现的方法里面找，看看数据的过滤转型时候是不是出现问题，如果是 UI 上的 bug，就去 View 层去找，看看是不是 UI 得布局写错了，或者计算错了，这样就会快速的定位问题，以便更快的解决问题。

坏处 1：很多把能写在 VC 的代码分出去了，会增加开发时间和开发成本，因为项目着急的时候，没有那么多时间让你去设计很好的结构，就是一股脑的写在了 VC 也是情理之中。因为没时间写，一个要长期维护的项目应该有一个很好的逻辑而不能和外包的项目比，外包只追求速度，做完就行，下一次谁做还不一定呢。

坏处 2：有的时候需要多些很多代码，但是只要是能重用的尽量分离出来为其他类重用。

2.2 代码布局

【规则 2-2-1】遵循 MVC 模式编写代码。

说明：以下内容如果有哪些不足，可以指出，方便作者更改，本人通过学习 iOS 开发指南，从零基础到 AppStore 上架，一本 iOS 架构设计图书中学习架构方面的知识。

如图 2-2-1 为 manager 文件夹获取数据的一个类的接口定义

```
/**
 *  @author lishiping, 16-05-23 18:05:21
 *
 *  获取热门车牌数据列表
 *
 *  @param callbackBlock 返回列表
 */
+ (void)requestHotBrandListArrayWithBlock:(void (^)(NSArray<
    YXUInfo_Brand*>* hotBrandArray, NSError* error))
    callbackBlock;

/**
 *  @author lishiping, 16-05-23 18:05:12
 *
 *  请求付一半的推荐车型数组，用户首页展示
 *
 *  @param callbackBlock 返回列表
 */
+ (void)requestPayHalfRecommendArrayWithBlock:(void (^)
    (NSArray* payHalfRecommendArray, NSError* error))
    callbackBlock;
```

如果 2-2-2 所示为 manager 类的接口定义

```
/**
 *  @author lishiping, 16-06-28 11:06:52
 *
 *  请求用户头像并缓存到本地
 *
 *  @param block 回调
 */
+(void)requestAvatarImageInBackgroundWithBlock:
    (ImageResultBlock)block;

/**
 *  @author lishiping, 16-06-03 20:06:53
 *
 *  请求当前用户的收藏数据
 *
 *  @param block 返回数据
 */
+(void)requestMyCollectCarListArrayWithBlock:
    (ArrayResultBlock)block;

```

如图 2-2-3 所示为预定义 BlockDefine.h

```
#import <UIKit/UIKit.h>

typedef void (^DidSelectItemCellClickedBlock)(NSIndexPath*
    indexPath);
typedef void (^ButtonClickedBlock)(id sender);

typedef void (^VoidResultBlock)(void);
typedef void (^BooleanResultBlock)(BOOL succeeded, NSError*
    error);
typedef void (^IntegerResultBlock)(NSInteger number, NSError*
    error);
typedef void (^ArrayResultBlock)(NSArray* objects, NSError*
    error);
typedef void (^DataResultBlock)(NSData* data, NSError* error)
    ;
typedef void (^ImageResultBlock)(UIImage* image, NSError*
    error);
typedef void (^StringResultBlock)(NSString* string, NSError*
    error);
typedef void (^IdResultBlock)(id object, NSError* error);
typedef void (^ProgressBlock)(NSInteger percentDone);
typedef void (^DictionaryResultBlock)(NSDictionary* dict,
    NSError* error);
typedef void (^ShowCarDetailsBlock)(YXUInfo_Car *info);
```

如图 2-2-1 为添加了明确注释的数据接口在返回的 Block 的数组中已经指定了数组内容为 YXUIn_Brand 类型的对象，这就告诉对接接口的开发者，数据里面存放的是车牌 Model 数据，可以按照 model 的属性做对接。

如图 2-2-2 所示为请求数据和预先定义的 Block 结合使用

还可以如图 2-2-3 所示，定义一个把能用到的 Block 都写在一个 BlockDefine.h 文件中，然后只要是哪个 manager 想要使用 block 就 import 这个 BlockDefine.h 文件，不用在每一个想要使用 Block 的时候又要重新定义一个，方便使用，又或者是把这个 BlockDefine.h 加入.pch 预编译头文件中，使用更方便，不过会造成不必要的类的引用，使编译时间过长，但是其实也没什么影响，作者本人就这把 BlockDefine.h 加入到预编译文件中。

如下图 2-2-4 所示为一个复杂 View 的初始化过程

```objc
#pragma mark - 热门车牌和热门价格
- (HomeHotBrandView*)homeHotBrandView
{
    _var_weakSelf;
    if (_homeHotBrandView == nil) {
        _homeHotBrandView = [[HomeHotBrandView alloc] init];
        _homeHotBrandView.backgroundColor = [UIColor
            whiteColor];
        _homeHotBrandView.frame = CGRectMake(0,
            BannerView_height, kScreenWidth,
            BrandView_height);
        [self.bgScrollView addSubview:_homeHotBrandView];

        _homeHotBrandView.didSelectItemCellClickedBlock = ^
            (NSIndexPath* indexPath) {
            //collectView第一行是热门车牌
            if (indexPath.section == 0) {
                if (indexPath.row == 0) {
                }
                else {
                    YXUInfo_Brand* info_Brand =
                        (YXUInfo_Brand*)[weakSelf.
                        homeHotBrandView.hotBrandArray
                        safe_objectAtIndex:indexPath.row - 1
                        ];
                    if (info_Brand) {

                        //热门车牌打点
                        _Click_Statistics(nil,
                            _DotName(@"home_recommendBrand")
                            , (@{@"品牌":info_Brand.brandname
                            ?:@"",@"位置":@(indexPath.row)}))
                            ;

                        [weakSelf
                            pushToCarListVCWithInfoBrand:
                            info_Brand];
                    }
                }
            }
            //collectView第二行热门价格
            else {
                if (indexPath.row == 0) {
                }
                else {
                    NSString* priceStr = [weakSelf.
                        homeHotBrandView.hotPriceArray
                        safe_objectAtIndex:indexPath.row - 1
                        ];
                    if (priceStr && ![priceStr
                        isEqualToString:@""]) {
                        [weakSelf pushToCarListVCWithPrice:
                            priceStr];
```

如图 2-2-5 所示为一个模块的方法集中放在一起

如图 2-2-6 所示为点击事件执行的方法

```
//点击热门车牌
- (void)pushToCarListVCWithInfoBrand:(YXUInfo_Brand*)
    info_Brand
{
    LOG(@"热门车牌%@", info_Brand.brandname);
    if (info_Brand.brandname) {
        YXU_SearchVC* searchVC = [YXU_SearchVC new];
        searchVC.isShowCarListVC = YES;
        searchVC.carListVC.brandDic = @{@"brandname":
            info_Brand.brandname,@"brandid":@(info_Brand.
            brandid)};
        [self.navigationController pushViewController:
            searchVC animated:YES];
    }
}
```

如上图 2-2-4 所示为一个复杂的 View 结合到 VC 里面的初始化的方法，在 VC 中简短的 50 行代码包含了该 View 的初始化和该 View 的点击事件的处理，那么当前的 View 上复杂的子视图添加以及布局就都在这个自定义 View 的内部了，代码分布更加清晰，想要更改子视图就去内部去看。在此我要强调一点，一个点击事件无论在子视图的子视图里面还是更复杂，一定要把点击事件的回调给 VC 去做点击的跳转以及其他操作，除非是特殊的点击控制内部视图变化，因为要遵循 MVC 模式 VC 是一切 push，Present 的控制呀。复杂的视图只负责显示数据和点击事件的回调，跳转和其他点击之后的处理要交给 VC。下面要说几点：

1. 复杂视图单独写在一个自定义 View 类里面，把不必要写的 VC 中的各种复杂视图的子视图的添加写到其他模块视图，使整个 VC 变得清晰简洁，更体现了代码的美观。
2. 重要部分的代码一定要加上注释，你写的代码你懂得地方不一定别人也很快的懂，多人协作开发就是要更好的合作，给别人开辟方便就是为自己得到赞扬。
3. 点击事件的打点一定要加入 XXX 打点的注释，这样在其他开发者全局搜索打点的时候很快能找到。
4. 要把模块的代码放在一起如图 2-2-5，有些人喜欢把初始化放在一起，点击事件放在一起，这样也可以。
5. 在联合开发的时候，开发者一定要想到和别人合作开发，要给别人一个能直接对接上的接口。非常强调的一点，要给他人提供方便，不要让别人进入你这个 VC 再自己写这个跳转方法，如图 2-1-6 所示，这个方法要跳转通过热门车牌进入车市的过程，那么通过车牌进入车市所需要的参数一定要有我刚点击的是哪个热门车牌呀。所以把刚才点击的热门车牌传给需要对接的参数。

如图 2-2-7 所示为一个.h 文件所写的接口

```
//

#import <UIKit/UIKit.h>

@interface BannerView : UIView

@property(nonatomic,strong)UIImageView *imageView;//banner图
    片
@property(nonatomic,strong)UILabel *titleLabel;//标题
@property(nonatomic,strong)UILabel *carNumberLabel;//城市车辆
    总数

/**
 *  @author lishiping, 16-07-14 15:07:46
 *
 *  设置banner图片上城市有多少真实车源
 *
 *  @param carNumber 车辆数
 *  @param city      城市
 */
-(void)setCarNumber:(NSInteger)carNumber city:(NSString*)
    city;

@end
```

如图 2-2-8 所示把属性写在{}里面

```
#import <UIKit/UIKit.h>

@interface BannerView : UIView
{
    UILabel *carNumberLabel;//城市车辆
}

@property(nonatomic,strong)UIImageView *imageView;//banner图
    片
@property(nonatomic,strong)UILabel *titleLabel;//标题

/**
 *  @author lishiping, 16-07-14 15:07:46
 *
 *  设置banner图片上城市有多少真实车源
 *
 *  @param carNumber 车辆数
 *  @param city      城市
 */
-(void)setCarNumber:(NSInteger)carNumber city:(NSString*)
    city;

@end
```

如图 2-2-7 所示每一个类的属性最好写成带有 property 的属性访问的成员变量，要比图 2-2-8 的方式更好一些，因为加了属性访问的成员变量在内存管理方面更好，一下几点需要重视：

1. 图 2-2-7 中每一个属性都是完整单词定义的，这是模仿 iOS 官方 SDK 的写法，iOS 官方的 SDK 都是单词的全拼没有简写，iOS 官方 SDK 没有 Btn，lbl，field 这样的写法，为了就是让开发者一目了然的懂得这个成员变量的意思。
2. 第二每一个成员变量后面都加了注释也增强了可读性，如果属性非常易懂，也可以不加注释。
3. 每个 public 方法都加了注释，增加了该方法的可读性，给其他调用该方法的开发者提供简单易懂的注释。

2.3 文件头布局

【规则 2-3-1】遵循统一的布局顺序来书写头文件。

说明：以下内容如果某些节不需要，可以忽略。但是其它节要保持该次序。

头文件布局:

文件头（参见“注释”一节）
#import （依次为标准库头文件、非标准库头文件）
全局宏
常量定义
全局数据类型
类定义
属性定义
公开方法定义

【规则 2-3-2】遵循统一的布局顺序来书写实现文件。

说明：以下内容如果某些节不需要，可以忽略。但是其它节要保持该次序。

实现文件布局：

文件头（参见“注释”一节）
#import （依次为标准库头文件、非标准库头文件）
文件内部使用的宏
文件内部常量定义
文件内部使用的数据类型
文件内部全局变量
文件内部本地变量（即静态全局变量）
类的实现

【规则 2-3-3】包含标准库头文件用尖括号 <>，包含非标准库头文件用双引号“”。

正例：

```
#import <stdio.h>
#import “heads.h”
```

2.4 基本格式

【规则 2-4-1】if、else、else if、for、while、do 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加 { }。

说明：这样可以防止书写失误，也易于阅读。

正例：

```
if (varible1 < varible2)
{
    varible1 = varible2;
}
```

反例：下面的代码执行语句紧跟 if 的条件之后，而且没有加{}，违反规则。

```
if (varible1 < varible2) varible1 = varible2;
```

【规则 2-4-2】定义指针类型的变量，*应放在变量前。

正例：

```
float    *pfBuffer;
```

反例：

```
float*    pfBuffer;
```

〖建议 2-4-3〗源程序中关系较为紧密的代码应尽可能相邻。

说明：这样便于程序阅读和查找。

正例：

```
iLength      = 10;
iWidth       = 5;       // 矩形的长与宽关系较密切，放在一起。
StrCaption   = “Test”;
```

反例：

```
iLength = 10;
strCaption   = “Test”;
iWidth       = 5;
```

2.5 对齐

【规则 2-5-1】程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。{ }之内的代码块使用缩进规则对齐。

说明：这样使代码便于阅读，并且方便注释。

do while 语句和结构的类型化时可以例外，while 条件和结构名可与 } 在同一行。

正例：

```
(void)Function:(int)iVar
{                                   // 独占一行并与引用语句左对齐。
    while (condition)
    {
        doSomething();     // 与{ }缩进 4 格
    }
}
```

反例：

```
void Function(int iVar){
    while (condition){
        DoSomething();
    }}
```

【规则 2-5-2】结构型的数组、多维的数组如果在定义时初始化，按照数组的矩阵结构分行书写。

正例：

```
int aiNumbers[4][3] =
{
    1, 1, 1,
    2, 4, 8,
    3, 9, 27,
    4, 16, 64
}
```

【规则 2-5-3】相关的赋值语句等号对齐。

正例：

```
tPDBRes.wHead      =  0;
tPDBRes.wTail      =  wMaxNumOfPDB - 1;
tPDBRes.wFree      =  wMaxNumOfPDB;
tPDBRes.wAddress   =  wPDBAddr;
tPDBRes.wSize      =  wPDBSize;
```

〖建议 2-5-6〗在 switch 语句中，每一个 case 分支和 default 要用{ }括起来，{ }中的内容需要缩进。

说明：使程序可读性更好。

正例：

```
switch (iCode)
{
    case 1:
    {
        DoSomething();        // 缩进 4 格
        break;
    }
    case 2:
    {                         // 每一个 case 分支和 default 要用{}括起来
        DoOtherThing();
        break;
    }
    …                         // 其它 case 分支
    default:
    {
        DoNothing();
        break;
    }
}
```

2.6 空行空格

【规则 2-6-1】函数(方法)块之间使用两个空行分隔。

说明：空行起着分隔程序段落的作用。适当的空行可以使程序的布局更加清晰。

正例：

```
(void)hey
{
    [hey 实现代码]
}
空一行
// 注释
(void)ack
{
    [ack 实现代码]
}
```

反例：

```
void Foo::Hey(void)
{
    [Hey 实现代码]
}
void Foo::Ack(void)
{
    [Ack 实现代码]
}
// 两个函数的实现是两个逻辑程序块，应该用空行加以分隔。
```

【规则 2-6-2】一元操作符如“!”、“~”、“++”、“--”、“*”、“&”（地址运算符）等前后不加空格。“[ ]”、“. ”、“->”这类操作符前后不加空格。

正例：

```
!bValue
~iValue
++iCount
*strSource
&fSum
aiNumber[i] = 5;
tBox.dWidth
tBox->dWidth
```

【规则 2-6-3】多元运算符和它们的操作数之间至少需要一个空格。

正例：

```
fValue    =    fOldValue;
fTotal    +    fValue
iNumber +=    2;
```

【规则 2-6-4】函数名之后不要留空格。

说明：函数名后紧跟左括号‘(’，以与关键字区别。

【规则 2-6-5】方法名与形参不能留空格，返回类型与方法标识符有一个空格。

说明：方法名后紧跟’:’，然后紧跟形参， 返回类型’(‘与’-‘之间有一个空格。

正例：

```
-␣(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations.
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

【规则 2-6-6】‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。‘,’之后要留空格。‘;’不是行结束符号时其后要留空格。

正例：

例子中的 ␣ 代表空格。

```
for␣(i␣=␣0;␣i␣<␣MAX_BSC_NUM;␣i++)
{
    DoSomething(iWidth,␣iHeight);
}
```

【规则 2-4-7】注释符与注释内容之间要用一个空格进行分隔。

正例：

```
/* 注释内容 */
// 注释内容
```

反例：

```
/*注释内容*/
//注释内容
```

2.7 断行

【规则 2-7-1】长表达式（超过 80 列）要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐。

说明：条件表达式的续行在第一个条件处对齐。
for 循环语句的续行在初始化条件语句处对齐。
函数调用和函数声明的续行在第一个参数处对齐。
赋值语句的续行应在赋值号处对齐。

正例：

```
if ((iFormat == CH_A_Format_M)
   && (iOfficeType == CH_BSC_M)) //  条件表达式的续行在第一个条件处对齐
{
    doSomething();
}

for (long_initialization_statement;
    long_condiction_statement;        // for 循环语句续行在初始化条件语句处对齐
    long_update_statement)
{
    doSomething();
}

// 函数声明的续行在第一个参数处对齐
BYTE ReportStatusCheckPara(HWND hWnd,
                           BYTE ucCallNo,
                           BYTE ucStatusReportNo);

// 赋值语句的续行应在赋值号处对齐
fTotalBill = fTotalBill + faCustomerPurchases[iID]
           + fSalesTax(faCustomerPurchases[iID]);
```

【规则 2-7-2】函数(方法)声明时，类型与名称不允许分行书写。

正例：

```
extern double FAR CalcArea(double dWidth, double dHeight);
```

反例：

```
extern double FAR
CalcArea(double dWidth, double dHeight);
```

3 注释

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

【规则 3-1】C 语言的注释符为“/* … */”。C++语言中，多行注释采用“/* … */”，单行注释采用“// …”。

〖建议 3-1〗单行注释采用//，多行采用注释符“/* … */”。

【规则 3-2】一般情况下，源程序有效注释量必须在 30%以上。

说明：注释的原则是有助于对程序的阅读理解，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息。

【规则 3-3】注释使用中文。

说明：对于特殊要求的可以使用英文注释，如工具不支持中文或国际化版本时。

【规则 3-4】文件头部建议进行注释，包括：.h 文件、.c 文件、.m 文件、.inc 文件、.def 文件、编译说明文件.cfg 等。

说明：注释必须列出：版权信息、文件标识、内容摘要、版本号、作者、完成日期、修改信息等。修改记录部分建议在代码做了大修改之后添加修改记录。备注：文件名称，内容摘要，作者等部分一定要写清楚。

正例：

下面是文件头部的中文注释：

```
/*********************************************************************
* 版权所有 (C)2011 中兴软件技术(南昌)有限公司
*
* 文件名称： // 文件名
* 文件标识： // 见配置管理计划书
* 内容摘要： // 简要描述本文件的内容，包括主要模块、函数及其功能的说明
* 其它说明： // 其它内容的说明
* 当前版本： // 输入当前版本
* 作　　者： // 输入作者名字及单位
* 完成日期： // 输入完成日期，例：2011 年 11 月 29 日
```

```
* 修改记录 1：// 修改历史记录，包括修改日期、修改者及修改内容
*     修改日期：
*     版 本 号：//或版本号
*     修 改 人：
*     修改内容：//修改原因以及修改内容说明
* 修改记录 2：…
**********************************************************************/
```

【规则 3-5】方法头部必须进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、访问和修改的表、修改信息等，除了函数(方法)名称和功能描述必须描述外，其它部分建议写描述。

说明：注释必须列出：函数名称、功能描述、输入参数、输出参数、返 回 值、修改信息等。备注：方法名称、功能描述要正确描述。

正例：

```
/**********************************************************************
* 方法名称：  // 方法名称
* 功能描述：  // 方法功能、性能等的描述
* 输入参数：  // 输入参数说明，包括每个参数的作用、取值说明及参数间关系
* 输出参数：  // 对输出参数的说明。
* 返 回 值：  // 方法返回值的说明
* 其它说明：  // 其它说明
**********************************************************************/
```

【规则 3-6】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。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

正例：

如下书写比较结构清晰

```
/* 获得子系统索引 */
iSubSysIndex = aData[iIndex].iSysIndex;

/* 代码段 1 注释 */
[ 代码段 1 ]

/* 代码段 2 注释 */
[ 代码段 2 ]
```

反例 1：

如下例子注释与描述的代码相隔太远。

```
/* 获得子系统索引 */

iSubSysIndex = aData[iIndex].iSysIndex;
```

反例 2：

如下例子注释不应放在所描述的代码下面。

```
iSubSysIndex = aData[iIndex].iSysIndex;
/* 获得子系统索引 */
```

反例 3：

如下例子，显得代码与注释过于紧凑。

```
/* 代码段 1 注释 */
[ 代码段 1 ]
/* 代码段 2 注释 */
[ 代码段 2 ]
```

【规则 3-7】全局变量要有详细的注释，包括对其功能、取值范围、访问信息及访问时注意事项等的说明。

正例：

```
/*
* 变量作用说明
* 变量值说明
*/
BYTE g_ucTranErrorCode;
```

【规则 3-8】注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

正例：

如下注释结构比较清晰

```
- (int)doSomething
{
    /* 代码段 1 注释 */
    [ 代码段 1 ]

    /* 代码段 2 注释 */
    [ 代码段 2 ]
}
```

反例：

如下例子，排版不整齐，阅读不方便；

```
int DoSomething(void)
{
/* 代码段 1 注释 */
    [ 代码段 1 ]

/* 代码段 2 注释 */
    [ 代码段 2 ]
}
```

〖建议 3-9〗尽量避免在注释中使用缩写，特别是不常用缩写。

说明：在使用缩写时，应对缩写进行必要的说明。

4 命名规则

4.1 基本命名规则

好的命名规则能极大地增加可读性和可维护性。同时，对于一个有上百个人共同完成的大项目来说，统一命名约定也是一项必不可少的内容。本章对程序中的所有标识符（包括变量名、常量名、函数名、类名、结构名、宏定义等）的命名做出约定。

【规则 4-1-1】标识符要采用英文单词或其组合，便于记忆和阅读，切忌使用汉语拼音来命名。

说明：标识符应当直观且可以拼读，可望文知义，避免使人产生误解。程序中的英文单词一般不要太复杂，用词应当准确。

【规则 4-1-2】严格禁止使用连续的下划线，下划线也不能出现在标识符头或结尾（预编译开关除外）。

说明：这样做的目的是为了使程序易读。因为 variable_name 和 variable__name 很难区分，下划线符号‘_’若出现在标识符头或结尾，容易与不带下划线‘_’的标识符混淆。

【规则 4-1-3】程序中不要出现仅靠大小写区分的相似的标识符。

【规则 4-1-4】宏、常量名都要使用大写字母, 用下划线 ‘_’ 分割单词。预编译开关的定义使用下划线 ‘_’ 开始。

正例：如 DISP_BUF_SIZE、MIN_VALUE、MAX_VALUE 等等。

【规则 4-1-5】程序中局部变量不要与全局变量重名。

说明：尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

【规则 4-1-6】使用一致的前缀来区分变量的作用域。

说明：变量活动范围前缀规范如下（少用全局变量）：

g_	：	全局变量
s_	：	模块内静态变量
空	：	局部变量不加范围前缀

【规则 4-1-7】方法名用小写字母开头的单词组合而成，方法名最后是一段话的完整单词。

说明：方法名力求清晰、明了，通过方法名就能够判断方法的主要功能。方法名中不同意义字段之间不要用下划线连接，而要把每个字段的首字母大写以示区分。方法命名采用大小写字母结合的形式，但专有名词不受限制。

正例：

如下例子应该每个单词都是完整的单词写方法，避免使用 Btn，lbl，fed 简写方式，可以参考 iOS 官方 SDK 方法和属性的命名虽然有时候略显较长，但是更清晰的就能知道该方法的含义，也是遵循 iOS 官方命名的规范。

lookButtonClick：（id）sender；

反例：

lookBtn;

〖建议 4-1-8〗尽量避免名字中出现数字编号，如 Value1、Value2 等，除非逻辑上的确需要编号。

〖建议 4-1-9〗标识符前最好不加项目、产品、部门的标识。

说明：这样做的目的是为了代码的可重用性。

4.2 资源命名规则

iOS 出了 Assets.xcassets 文件归类之后可以更好地管理资源文件，Assets.xcassets 管理的文件在打包的时候自带压缩功能，可以降低文件包的大小，同时不用管理文件 2 倍图和 3 倍图的问题，都是自动匹配的。

可以吧某一模块的 UI 所需要的资源放在一个 group 里面，把公用的放在一个 common 的 group 里面。

规范的是以该 group_xx 为名字，其实在实际开发中都没有很规范，UI 给了什么样的图片就直接拉过来了，随便起了一个名字。如果说最好的命名那就是中英文都有吧，查找方便，又防止重复，我也没有规范的命名过。

4.3 变量，常量，宏命名规则

变量、常量和数据类型是程序编写的基础，它们的正确使用直接关系到程序设计的成败，变量包括全局变量、局部变量和静态变量，常量包括数据常量和指针常量，类型包括系统的数据类型和自定义数据类型。本节主要说明变量、常量与类型使用时必须遵循的规则和一些需注意的建议。

【规则 4-3-1】一个变量有且只有一个功能，尽量不要把一个变量用作多种用途。

说明：一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

【规则 4-3-2】循环语句与判断语句中，不允许对其它变量进行计算与赋值。

说明：循环语句只完成循环控制功能，if 语句只完成逻辑判断功能，不能完成计算赋值功能。

正例：

```
do
{
    [处理语句]
    cInput = GetChar();
} while (cInput == 0);
```

反例：

```
do
    {
    [处理语句]
} while (cInput = GetChar());
```

【规则 4-3-3】宏定义中如果包含表达式或变量，表达式和变量必须用小括号括起来。

说明：在宏定义中，对表达式和变量使用括号，可以避免可能发生的计算错误。

正例：

```
#define    HANDLE(A, B)     (( A ) / ( B ))
```

反例：

```
#define    HANDLE(A, B)     (A / B)
```

【规则 4-3-4】宏名大写字母。

正例：

```
#define    BUTTON_WIDTH       (int)320     //模仿 iOS SDK 的写法
```

反例：

```
#define    kButtonWidth       (int)320
```

【规则 4-3-5】宏常量要指定类型。

说明：不同的编译器，默认类型不一样。

正例：

```
#define    BUTTON_WIDTH       (int)320
```

反例：

```
#define    BUTTON_WIDTH       320
```

5 表达式与语句

表达式是语句的一部分，它们是不可分割的。表达式和语句虽然看起来比较简单，但使用时隐患比较多。本章归纳了正确使用表达式和 if、for、while、goto、switch 等基本语句的一些规则与建议。

【规则 5-1】在表达式中使用括号，使表达式的运算顺序更清晰。

说明：由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

正例：

```
if   (((iYear % 4 == 0) && (iYear % 100 != 0)) || (iYear % 400 == 0))
```

反例：

```
if (iYear % 4 == 0 && iYear % 100 != 0 || iYear % 400 == 0)
```

【规则 5-2】不可将布尔变量和逻辑表达式直接与 YES、NO 或者 1、0 进行比较。

说明：TURE 和 FALSE 的定义值是和语言环境相关的，且可能会被重定义的。

正例：

```
设 bFlag  是布尔类型的变量
if (bFlag)          // 表示 flag 为真
if (!bFlag)         // 表示 flag 为假
```

反例：

```
设 bFlag  是布尔类型的变量

if (bFlag == TRUE)
if (bFlag == 1)
if (bFlag == FALSE)
if (bFlag == 0)
```

【规则 5-3】在条件判断语句中，当整型变量与 0 比较时，不可模仿布尔变量的风格，应当将整型变量用“==”或“!=”直接与 0 比较。

正例：

```
if (iValue == 0)

if (iValue != 0)
```

反例：

```
if (iValue)          // 会让人误解 iValue 是布尔变量

if (!iValue)
```

【规则 5-4】不可将浮点变量用“==”或“!=”与任何数字比较。

说明：无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该转化成“>=”或“<=”形式。

正例：

```
if ((fResult >= -EPSINON) && (fResult <= EPSINON))
```

反例：

```
if (fResult == 0.0)     // 隐含错误的比较
```

其中 EPSINON 是允许的误差（即精度）。

【规则 5-5】应当将指针变量用“==”或“!=”与 nil 比较。

说明：指针变量的零值是“空”（记为 NULL），即使 NULL 的值与 0 相同，但是两者意义不同。

正例：

```
if (pHead == nil)    // pHead 与 NULL 显式比较，强调 pHead 是指针变量
if (pHead != nil)
```

反例：

```
if (pHead == 0)          // 容易让人误解 pHead 是整型变量
if (pHead != 0)
```

或者

```
if (pHead)               // 容易让人误解 pHead 是布尔变量
if (!pHead)
```

【规则 5-6】在 switch 语句中，每一个 case 分支必须使用 break 结尾，最后一个分支必须是 default 分支。

说明：避免漏掉 break 语句造成程序错误。同时保持程序简洁。

对于多个分支相同处理的情况可以共用一个 break，但是要用注释加以说明。

正例：

```
switch (iMessage)
{
    case SPAN_ON:
    {
        [处理语句]
        break;
    }
```

```
        case SPAN_OFF:
        {
            [处理语句]
            break;
        }
        default:
        {
            [处理语句]
            break;
        }
    }
```

【规则 5-7】不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

〖建议 5-8〗循环嵌套次数不大于 3 次。

〖建议 5-9〗如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。

说明：下面两个示例中，反例比正例多执行了 NUM -1 次逻辑判断。并且由于前者总要进行逻辑判断，使得编译器不能对循环进行优化处理，降低了效率。如果 NUM 非常大，最好采用正例的写法，可以提高效率。

```
const    int NUM = 100000;
```

正例：

```
if (bCondition)
{
    for (i = 0; i < NUM; i++)
    {
        doSomething();
    }
}
else
{
    for (i = 0; i < NUM; i++)
    {
        doOtherthing();
    }
}
```

反例：

```
for (i = 0; i < NUM; i++)
{
    if (bCondition)
    {
        DoSomething();
    }
    else
    {
        DoOtherthing();
    }
}
```

〖建议 5-10〗for 语句的循环控制变量的取值采用“半开半闭区间”写法。

说明：这样做更能适应 c 语言数组的特点，c 语言的下标属于一个“半开半闭区间”。

正例：

```
int    aiScore[NUM];
…
for (i = 0; i < NUM; i++)
{
    printf(“%d\n”,aiScore[i])
}
```

反例：

```
int    aiScore[NUM];
…
for (i = 0; i <= NUM-1; i++)
{
    printf(“%d\n”,aiScore[i]);
}
```

相比之下，正例的写法更加直观，尽管两者的功能是相同的。

6 函数、方法、接口

〖规则 6-1〗方法不能为多个目的服务。

正例：

```
- (BOOL)sio_set_baud_rate:(int)arg;
- (BOOL)sio_set_stop_bits(byte)arg;
- (BOOL)sio_set_data_bits(byte)arg;
- (BOOL)sio_get_baud_rate:(int *)arg;
```

反例：

```
- (BOOL)sio_ ioctl:(void *)arg;
```

〖规则 6-2〗在组件接口中应该尽量少使用外部定义的类型（重用，减少耦合）。

〖建议 6-3〗避免函数有太多的参数，参数个数尽量控制在 5 个以内。

说明：如果参数太多，在使用时容易将参数类型或顺序搞错，而且调用的时候也不方便。如果参数的确比较多，而且输入的参数相互之间的关系比较紧密，不妨把这些参数定义成一个结构，然后把结构的指针当成参数输入。

【规则 6-4】对于有返回值的函数(方法)，每一个分支都必须有返回值。

说明：为了保证对被调用函数返回值的判断，有返回值的函数中的每一个退出点都需要有返回值。

【规则 6-5】对输入参数的正确性和有效性进行检查。

说明：很多程序错误是由非法参数引起的，我们应该充分理解并正确处理来防止此类错误。

【规则 6-6】防止将函数(方法)的参数作为工作变量。

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

〖建议 6-7〗函数(方法)的功能要单一，不要设计多用途的函数(方法)。

说明：多用途的函数往往通过在输入参数中有一个控制参数，根据不同的控制参数产生不同的功能。这种方式增加了函数之间的控制耦合性，而且在函数调用的时候，调用相同的一个函数却产生不同的效果，降低了代码的可读性，也不利于代码调试和维护。

〖建议 6-8〗函数(方法)体的规模不能太大，尽量控制在 200 行代码之内。

说明：冗长的函数不利于调试，可读性差。

【规则 6-9】避免设计多参数函数(方法)。

7 头文件和类继承引用

7.1 头文件引用

【规则 7-1-1】如果不是确实需要，应该尽量避免头文件包含其它的头文件。

说明：头文件中应避免包含其它不相关的头文件，一次头文件包含就相当于一次代码拷贝。

【规则 7-1-2】申明成员类，应该引用该类申明，而不是包含该类的头文件。

说明：正例：

```
@class SubClassName;
@interface ClassName : NSObject
{
    SubClassName *m_pSubClassName;
}
```

反例：

```
#import “SubClassName.h”;
@interface ClassName : NSObject
{
    SubClassName *m_pSubClassName;
}
```

7.2 类继承引用

【规则 7-2-1】在编写派生类的赋值时，注意不要忘记对基类的成员变量重新赋值。

说明：除非在派生类中调用基类的赋值函数，否则基类变量不会自动被赋值。

正例：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
}
```

【规则 7-2-2】私有方法应该在实现文件中申明。

正例：

```
@interface ClassName(Private)

- (void)test;

@end


- (void)test
{
}
```

8 内存和指针

8.1 内存使用

【规则 8-1-1】防止内存操作越界，尽量使用 safe_addobject 方式的类别方法。

说明：内存操作主要是指对数组、指针、内存地址等的操作，内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细。通常在 iOS 开发中使用类别扩展数组类增加 safe_addobject 之类的方法内判断数组，然后再做其他处理。

正例：

```
const int MAX_USE_NUM = 10                                  // 用户号为 1-10
unsigned char aucLoginFlg[MAX_USR_NUM + 1]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

- (void)ArrayFunction
{
    unsigned char ucUserNo;
    for (ucUserNo = 0; ucUserNo < MAX_USE_NUM; ucUserNo++)
    {
        aucLoginFlg[ucUser_No] = ucUserNo;
        … …
    }
}
```

反例：

```
const int MAX_USE_NUM = 10                                  // 用户号为 1-10
unsigned char aucLoginFlg[MAX_USR_NUM]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
- (void)ArrayFunction
{
    unsigned char ucUserNo;
    for (ucUserNo = 1; ucUserNo < 11; ucUserNo++)  // 10 已经越界了
    {
        aucLoginFlg[User_No] = ucUserNo;
        … …
    }
}
```

【规则 8-1-2】必须对动态申请的内存做有效性检查，并进行初始化；动态内存的释放必须和分配成对以防止内存泄漏，释放后内存指针置为 nil。

说明：对嵌入式系统，通常内存是有限的，内存的申请可能会失败，如果不检查就对该指

针进行操作，可能出现异常，而且这种异常不是每次都出现，比较难定位。

指针释放后，该指针可能还是指向原有的内存块，可能不是，变成一个野指针，一般用户不会对它再操作，但用户失误情况下对它的操作可能导致程序崩溃。

正例：

```
- (void)memmoryFunction
{
    unsigned char *pucBuffer = NULL;
    pucBuffer = GetBuffer(sizeof(DWORD));
    if (NULL != pucBuffer)                  // 申请的内存指针必须进行有效性验证
    {
        // 申请的内存使用前必须进行初始化
        memset(pucBuffer, 0xFF, sizeof(DWORD));
    }
    ….
    FreeBuffer(pucBuffer);          // 申请的内存使用完毕必须释放
    pucBuffer = NULL;               // 申请的内存释放后指针置为空

    …
}
```

【规则 8-1-3】变量在使用前应初始化，防止未经初始化的变量被引用。

说明：不同的编译系统，定义的变量在初始化前其值是不确定的。有些系统会初始化为 0，而有些不是。

8.2 指针使用

【规则 8-2-1】指针不要进行复杂的逻辑或算术操作。

说明：指针加一的偏移，通常由指针的类型确定，如果通过复杂的逻辑或算术操作，则指针的位置就很难确定。

【规则 8-2-2】减少指针和数据类型的强制类型转化。

【规则 8-2-3】对变量进行赋值时，必须对其值进行合法性检查，防止越界等现象发生。

说明：尤其对全局变量赋值时，应进行合法性检查，以提高代码的可靠性、稳定性。

【规则 8-2-5】在哪申请在哪释放。

【规则 8-2-5】 非初始化方法中的 alloc 操作之前必须要 nil 判断；

【建议 8-2-6】全局指针释放后置为 nil 值。

9 断言与错误处理

断言是对某种假设条件进行检查（可理解为若条件成立则无动作，否则应报告）。它可以快速发现并定位软件问题，同时对系统错误进行自动报警。断言可以对在系统中隐藏很深，用其它手段极难发现的问题进行定位，从而缩短软件问题定位时间，提高系统的可测性。在实际应用时，可根据具体情况灵活地设计断言。

【规则 9-1】整个软件系统应该采用统一的断言。如果系统不提供断言，则应该自己构造一个统一的断言供编程时使用。

说明：整个软件系统提供一个统一的断言函数，如 Assert(exp)，同时可提供不同的宏进行定义（可根据具体情况灵活设计），如：

（1）#define ASSERT_EXIT_M 中断当前程序执行，打印中断发生的文件、行号，该宏一般在单调时使用。

（2）#define ASSERT_CONTINUE_M 打印程序发生错误或异常的文件,行号,继续进行后续的操作，该宏一般在联调时使用。

（3）#define ASSERT_OK_M空操作，程序发生错误情况时，继续进行，可以通过适当的方式通知后台的监控或统计程序，该宏一般在 RELEASE 版本中使用。

【规则 9-2】正式软件产品中应把断言及其它调测代码去掉（即把有关的调测开关关掉）。

说明：加快软件运行速度。

〖建议 9-3〗使用断言检查函数输入参数的有效性、合法性。

说明：检查函数的输入参数是否合法，如输入参数为指针，则可用断言检查该指针是否为空，如输入参数为索引，则检查索引是否在值域范围内。

正例：

```
BYTE    StoreCsrMsg(WORD wIndex,    T_CMServReq    *ptMsgCSR)
{
    WORD            wStoreIndex;
    T_FuncRet       tFuncRet;

    Assert (wIndex < MAX_DATA_AREA_NUM_A);   // 使用断言检查索引
    Assert (ptMsgCSR != NULL);                // 使用断言检查指针

    …                                         // 其它代码

    return OK_M;
}
```

10 其它补充

【规则 10-1】避免过多直接使用立即数。

正例：

```
ViewBounds.size.height = VIEW_BOUNDS_HEIGHT;
```

反例：

```
ViewBounds.size.height = 150;
Height = 150;
```

【规则 10-2】枚举第一个成员要赋初始值。

正例：

```
typedef enum
{
    WIN_SIZE_NORMAL = 0,
    WIN_SIZE_SMALL
}WinSize;
```

反例：

```
typedef enum
{
    WIN_SIZE_NORMAL,
    WIN_SIZE_SMALL
}WinSize;
```

【规则 10-3】addObject 之前要非空判断。

【规则 10-4】release 版本代码去掉 NSLog 打印。

【规则 10-5】禁止在代码中直接写死字符串资源，尽量要用字符串 ID 替代。

说明：应该要考虑多语言国际化，尽量使用 `NSLocalizedStringFromTable` 实现对字符串 `ID` 的引用

【规则 10-6】对于框架设计，数据逻辑层应尽量与 UI 层分离，降低耦合度。

【规则 10-7】同等难度下，优先考虑代码实现窗体创建。

说明：代码实现窗体创建容易移植，优先考虑代码实现来替代 xib 实现及 storyboard，因为 storyboard 只要打开以下，就会记录一个版本号，并将你记得 XCode 版本加入，容易造成不必要的麻烦。

11XCode 工具插件

Package Manager 的使用，为 XCode 安装 Package Manager 插件工具管理器之后，可以选择安装一些插件，向大家介绍一些非常好用的 Xcode 代码编辑插件，下面介绍几个很好的小工具：

1. ClangFormat 为代码自动排版，断行，按照规范重新排布代码，整理代码排版的功能。
2. KSImageNamed 当你要使用图片资源对象的时候，会自动检索当前功能中的所有图片推荐给你。
3. Peckham 是款 Xcode 插件，是 Xcode 自动补全功能很好的补充，可方便开发者添加导入语句，比如 Xcode 不能自动补全 pod 头部的导入，而 Peckham 可以很好地解决这个问题。按下 Command+Control+P 键，键入几个头部字母，从弹出的选项列表中进行选择。
4. VVDocumenter-XCode 很多时候，为了快速开发，很多的技术文档都是能省则省，这个时候注释就变得异常重要，再配合 Doxygen 这种注释自动生成文档的，就完美了。但是每次都要手动输入规范化的注释，着实也麻烦，但有了 VVDocumenter，只需要在要写文档的代码上面连打三个斜杠，就能自动提取参数等生成规范的 Javadoc 格式文档注释 作者也开源了相关代码，而且也可以说着每次打注释的作者非常好用。
5. **XAlign**，这个插件可以很快速地使代码对齐，有 3 种模式："="对齐、宏定义对齐和属性对齐。

还有其他好用的插件我还没有安装，大家可以自己去查询还有哪些更好的，顺便也告诉我一声，谢谢啦。

12 参考文档

《iOS 开发指南》
《iOS9 开发指南》
《iOS 开发进阶，唐巧》
《深入浅出 iPhone 开发》