1. 代码规范

文件书写及存放规范 3

.h文件 3

.m文件 4

.xib文件 5

资源文件 5

存放 6

类命名规范 6

方法命名及存放规范 6

命名 6

存放 7

属性命名及存放规范 8

命名 8

存放 8

统一代码书写习惯 8

禁止魔鬼数字 8

禁止超过3000行的类（category/utility及一些特殊情况除外） 8

禁止超过200行的方法 8

关于废弃的方法使用 8

注释！注释！还是注释！ 9

尽量换行 9

大括号要求 9

方法调用 10

界面细化 10

提交代码前保证各个iOS版本，屏幕尺寸以及retina/非retina的兼容 10

重载 10

不要使用NSLog 11

缩进 11

KISS原则（Keep it simple stupid） 11

一致性原则 12

复用原则 12

常见的引起bug的代码案例 12

案例1 12

案例2 12

案例3 13

案例4 13

案例5 14

案例6 15

案例7 15

常见的坏味道 18

魔鬼数字 19

重复代码 19

过长的判断语句 19

过长的参数列表 19

过长函数 19

过大的类 20

过多的临时变量 20

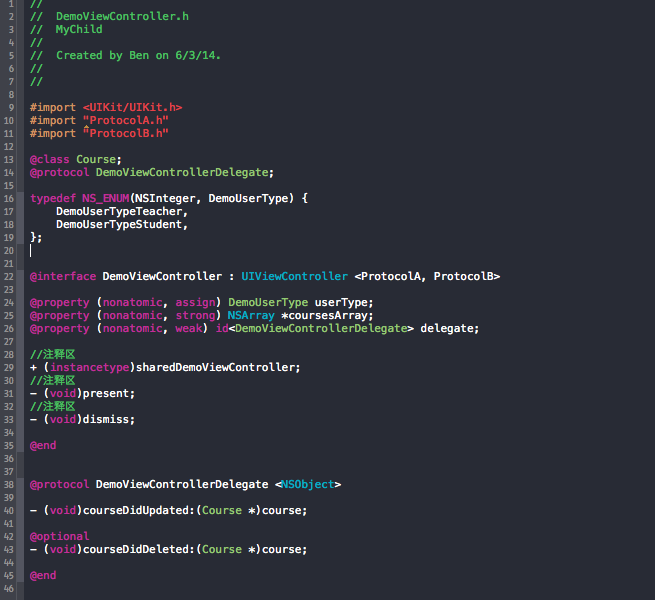
多处相同分支判断 20

不正确的注释，翻译性的注释 20

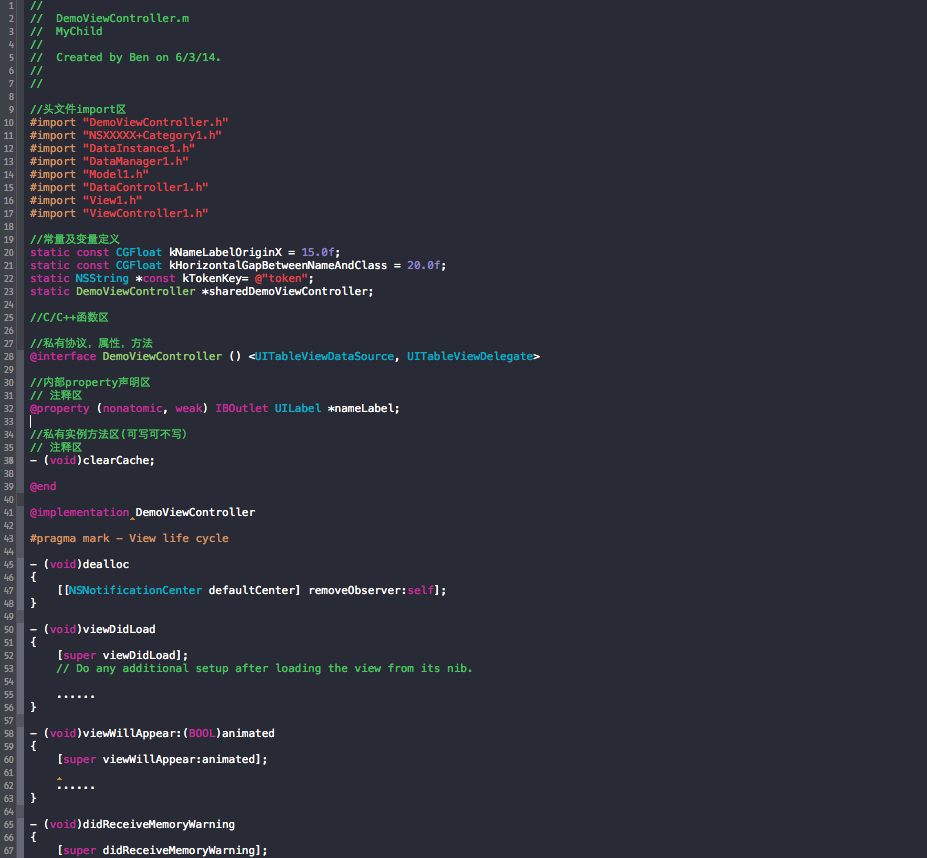
继承，包装，category使用的选择 20

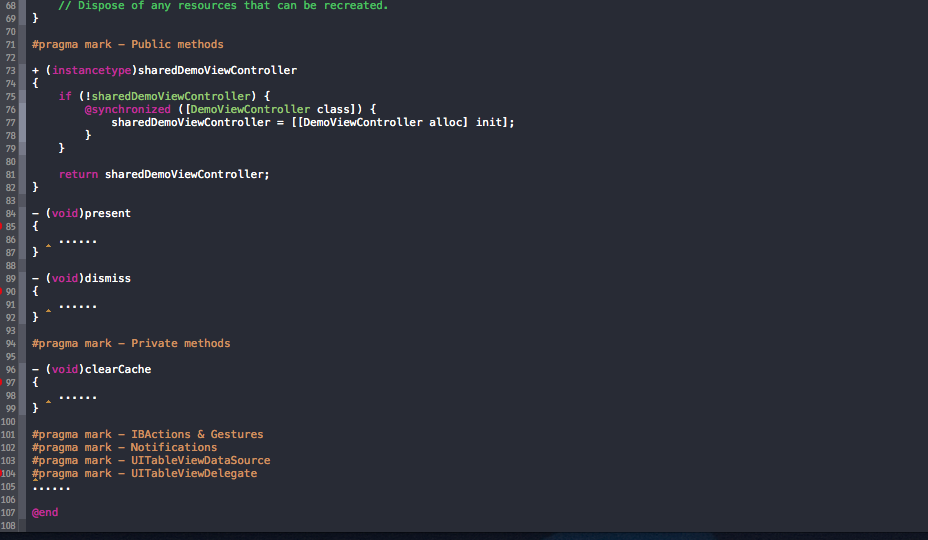
# 文件书写及存放规范

## .h文件

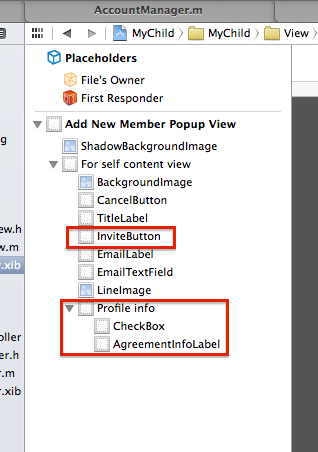
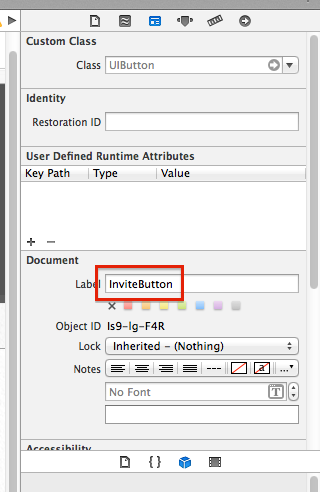


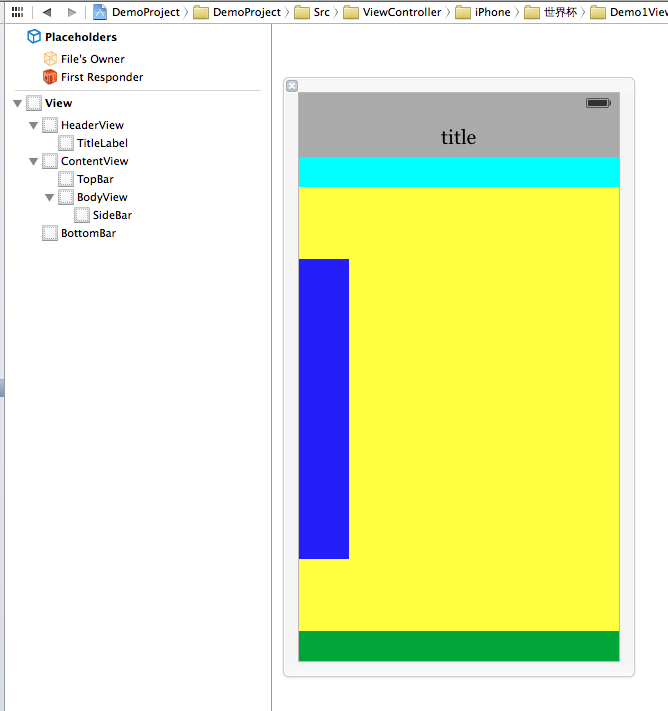
## .m文件

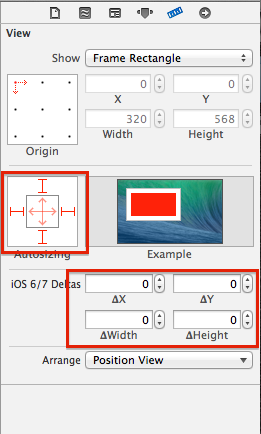
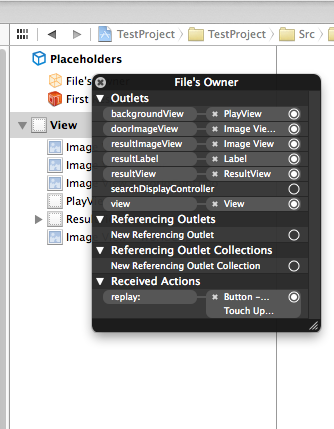




## .xib文件



## 资源文件

png/plist/bundle等，命名：iPhone/iPad\_模块名称\_界面或元素名称\_用途

若资源iPhone和iPad共享，前缀iPhone/iPad\_不需要

若资源各模块共享，模块名称可以省略

例如:

iPhone\_worldCup\_homePage\_background.png

login\_cancelButton\_background.png

navigationBarBackButton\_image.png

commonBackground.png

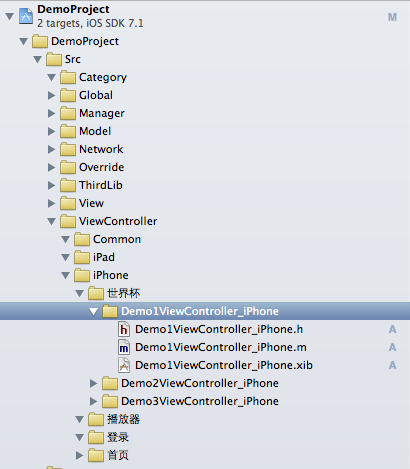
## 存放

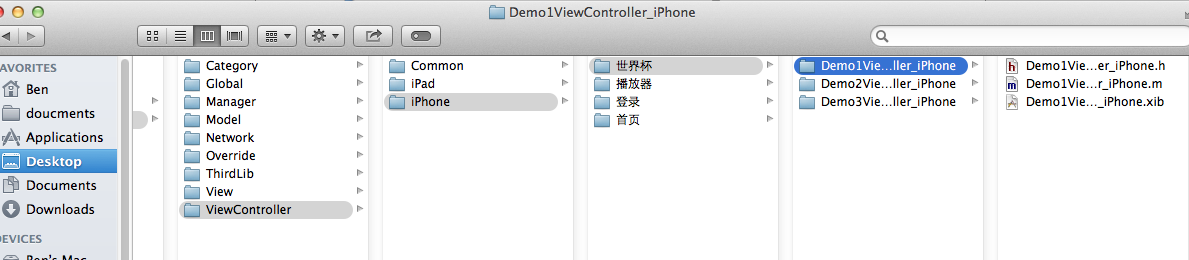
将相应的.h/.m/.xib放入一个文件夹

放入工程目录相应的分类位置下, 以方便查找维护

保持文件系统和工程的目录系统严格一致

资源文件尽量使用xcassets存放，放入新资源文件前注意重名问题





# 类命名规范

工程中自己的类（不包括第三方类，不包括公司公用iOS开发库中类）（包括Category的名称）统一命名规则为：

Sport\_ClassName或CategoryName\_iPhone/iPad

其中若不区分iPhone/iPad，可以不带\_iPhone/iPad后缀

主要考虑避免与第三方或系统未来的类重名

例如：

Sport\_HomeViewController\_iPhone

Sport\_CommonHeaderViewController

NSString+Sport\_Addition

# 方法命名及存放规范

## 命名

系统基类category或继承后新方法统一命名规则为：sport\_functionName (考虑防止与基类私有方法，第三方category方法或未来的方法重名)

自己的类方法统一命名规则为: functionName

其中functionName命名需要做到清晰明了，稍微长点没有关系，统一采用驼峰式命名，如：

小驼峰式命名法（lower camel case）：

第一个单字以小写字母开始；第二个单字的首字母大写，例如：firstName、lastName。

It is good to be both clear and brief as possible, but clarity shouldn’t suffer because of brevity:

|  |  |
| --- | --- |
| Code | Commentary |
| insertObject:atIndex: | Good. |
| insert:at: | Not clear; what is being inserted? what does “at” signify? |
| removeObjectAtIndex: | Good. |
| removeObject: | Good, because it removes object referred to in argument. |
| remove: | Not clear; what is being removed? |

In general, don’t abbreviate names of things. Spell them out, even if they’re long:

|  |  |
| --- | --- |
| Code | Commentary |
| destinationSelection | Good. |
| destSel | Not clear. |
| setBackgroundColor: | Good. |
| setBkgdColor: | Not clear. |

Avoid ambiguity in API names, such as method names that could be interpreted in more than one way.

|  |  |
| --- | --- |
| Code | Commentary |
| sendPort | Does it send the port or return it? |
| displayName | Does it display a name or return the receiver’s title in the user interface? |

关于命名强烈要求大家看看[Coding Guidelines for Cocoa](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html) ，学习一下Cocoa中的编码规范。

## 存放

方法能不暴露出来尽量不暴露，精简接口，方便使用该类

扩展已有类方法，根据具体场景决定使用category还是继承

修改已有类方法，根据具体场景决定使用换实现，category还是继承

方法应该放到相对应职责的类中，例如：和Model相关的构造，排序，拷贝，存档等方法应该放入Model类中

# 属性命名及存放规范

## 命名

如果实例是对象，需要在命名尾部添加对象的名称，前缀不需要。

@property (nonatomic, strong) UIWebView \*loadingWebView

@property (nonatomic, strong) NSArray \*studentArray;

@property (nonatomic, strong) NSMutableDictionary \*teacherMutableDict;

常量需要以k开头，以便区分。

static const CGFloat kTableViewCellHeight = 75.0f;

static NSString \*const kTokenKey = @”tk”;

static NSString \*const kUserInfoDidUpdatedNotification = @” kUserInfoDidUpdatedNotification”;

其它参阅[Coding Guidelines for Cocoa](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html)

## 存放

根据变量需要的作用域尽可能放入最小作用域范围内

对象的成员变量尽量不要暴露，精简接口，方便使用

全局的变量尽量用单件，方便调试和支持多线程

# 统一代码书写习惯

## 禁止魔鬼数字

## 禁止超过3000行的类（category/utility及一些特殊情况除外）

## 禁止超过200行的方法

## 关于废弃的方法使用

优先使用高版本SDK提供的方法。如：

if ([tabController respondsToSelector:@selector(presentViewController:animated:completion:)]) {

[tabController presentViewController:webViewController

animated:YES

completion:nil];

} else {

[tabController presentModalViewController:webViewController

animated:YES];

}

判断系统版本采用该方法

don’t：[[[UIDevice currentDevice] systemVersion] floatValue] >= 5.0

do：if ([self respondsToSelector:@selector(dismissViewControllerAnimated:completion:)]) {

} else {

}

注：dismissViewControllerAnimated:completion:IOS5.0以上调用该方法

## 注释！注释！还是注释！

平均10行有1行注释，欢迎更多注释。

特别是一些比较特殊的场景处理或业务逻辑处，应该通过注释方便其他开发阅读。

## 尽量换行

不要在一行上写多个内容，这样对代码合并有好处，另外生成的diff也比较内容分辨。

Bad：

@interface PPAppDelegate : UIResponder <UIApplicationDelegate, PPDataControllerDelegate,WXApiDelegate>

Good：

@interface PPAppDelegate : UIResponder <

UIApplicationDelegate,

PPDataControllerDelegate,

WXApiDelegate

>

Bad:

alertView = [[PPAlertViewPro alloc] initWithTitle:@"PPTV网络电视" message:notification.alertBody delegate:nil cancelButtonTitle:@"确定 "otherButtonTitles:nil];

Good:

alertView = [[PPAlertViewPro alloc] initWithTitle:@"PPTV网络电视"

message:notification.alertBody

delegate:nil

cancelButtonTitle:@"确定"

otherButtonTitles:nil];

## 大括号要求

正大括号不要另起一行，如：

if () {

}

if () {

} else {

}

for () {

}

只有一行语句的if else请不要省略括号

最外面的方法括号请统一换行



## 方法调用

不要在一行里写完所有的方法，直接按回车键，Xcode会自动与上一个”:”对齐，这样方便查看参数

[myObject doFooWith:arg1

name:arg2

error:arg3];

另外，参数和”:”之间不要加空格

## 界面细化

单个界面元素UIViewController化

## 提交代码前保证各个iOS版本，屏幕尺寸以及retina/非retina的兼容

## 重载

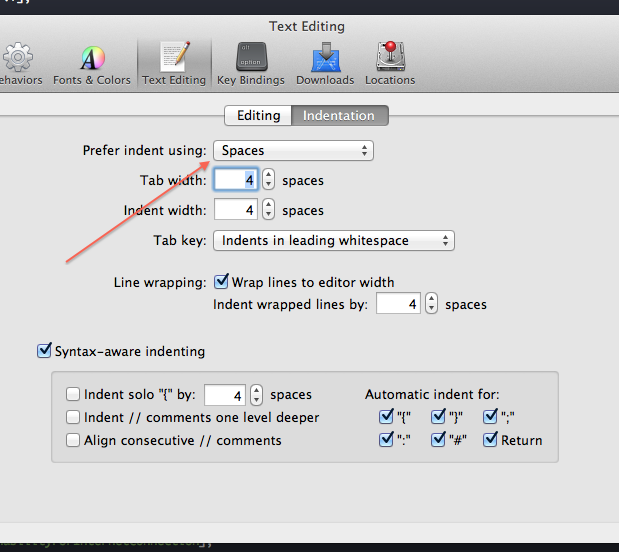
除特殊情况下，尽量在第一行调用父类的方法

## 不要使用NSLog

使用自定义的Log宏(Release版本不打印)，频繁直接使用NSLog会影响发布版本的性能。

## 缩进

统一使用空格缩进，不建议使用tab符缩进



# KISS原则（Keep it simple stupid）

一个临时／成员／全局变量一个用途，命名清晰易懂。

一个方法实现一个简单的功能，复杂的功能拆分成多个简单的功能步骤，避免过长的方法，方便复用。

一个类负责明确的职责，有职责外包的意识，各司其职， 避免超级类。

例如：ViewController中将与Model相关的职责（排序，拷贝，持久化，打印）外包给Model相关的类，将与网络请求数据相关的职责外包给PPDataController相关的类，将与自定义视图／控件相关的职责（由model刷新显示，观察model并实时刷新，根据相关值自定义显示，自定义的交互逻辑等），将与UI基本布局与适配相关的职责外包给UINib/xib，将与复杂的数据运算处理职责可外包给工具类／算法类(可以配合策略模式)／数据管理类／类的category中等等，这样最后放在ViewController中的主要就是核心的业务逻辑，其中主要是和各个外包类接口和回调打交道, 做到易读易扩展。

一些底层的自定义view或control，不要掺杂任何业务逻辑，做到其它工程拿一个类或一个文件夹就可以直接使用。

# 一致性原则

同一个工程保持一致的命名规范，书写规范，存放规范

复用代码中已有的自定义或第三方view/control，已有的Utility/Category方法，已有的基类，已有的模块

当基类的抽象不能概括新需求时，优先考虑修改现有框架以适配新需求，实在无法修改时新写的自定义view/control, 基类，模块要考虑不仅满足当前使用场景，还要做到对未来需求中可能变化的因素封装以做到可配置，或者通过委托（datasource/delegate）自定义，或者通过继承重写，并且提供精简易用的接口，接口中多注释，复杂的最好提供使用样例代码

# 复用原则

公用的常量放到全局的头文件中或配置单件中，文件头部 ，方法中。例如：设计要求的界面一致性的字体大小，颜色，透明度，阴影，3.5与4.0屏的高度差，无效id，键盘高度，宏定义，全局的常量字符串（产品名称，notification name，网络请求中公用的常量参数，重用的标示符等）。

公用的变量放到全局的单件（Manager，Configuration）中，方便增删查改，方便复用，方便调试 。

公用的与业务逻辑无关的方法较短的用宏定义，较长的可选择放入工具类或Category中。例如：判断设备尺寸用宏定义，判断是否是合法email地址放入一个ValidateUtility工具类中，将date对象转换成制定的格式字符串放入NSDate+Formatter的category中。

类中与自身关联密切的重复代码应该提取成单独的方法，方便继承重写，模版方法可方便子类复用。

基类应该保持职责单一，接口易读易用，变化的因素可方便的通过配置，或通过继承重写，或通过委托自定义，从而避免工程中功能相似的类。

# 常见的引起bug的代码案例

## 案例1

-(void) changeDatasource\_backgroundThread //Wrong  
{  
 [self.dataSourceArray removeAllObjects];  //或者self.dataSourceArray = anotherNewArray;  
 [self.tableViewperformSelectorOnMainThread:@selector(reloadData) withObject:nil waitUntilDone:NO];  
}

-(void) changeDatasource\_backgroundThread //Right  
{  
 dispatch\_async(dispatch\_get\_main\_queue(), ^{  
         [self.dataSourceArray removeAllObjects];  //或者self.dataSourceArray = anotherNewArray;

[self.tableView reloadData];  
 });

}

## 案例2

- (void)testAccessVariable

{

NSInteger outsideVariable = 10;

void (^blockObject)(void) = ^(void) {

NSLog(@" > outside variable = %d", outsideVariable);

};

outsideVariable = 30;

blockObject();

} // 打印的内容是什么？

使用 \_\_block或static修饰NSInteger outsideVariable = 10; 打印内容是什么？

blockObject 在实现时会对 outside 变量进行只读拷贝，在 block 块内使用该只读拷贝。因此这里输出的是拷贝时的变量值 10。如果，我们想要让 blockObject 修改或同步使用 outside 变量就需要用 \_\_block 来修饰 outside 变量。(\_\_block 变量其实是一个结构体对象，拷贝的是指向该结构体对象的指针)（对于 static 变量，全局变量，在 block 中是有读写权限的，因为在 block 的内部实现中，拷贝的是指向这些变量的指针）

## 案例3

@interface KSViewController ()

{

id \_observer;

}

@end

@implementation KSViewController

- (void)viewDidLoad

{

[super viewDidLoad];

// Do any additional setup after loading the view, typically from a nib.

KSTester \* tester = [[KSTester alloc] init];

[tester run];

\_observer = [[NSNotificationCenter defaultCenter]

addObserverForName:@"TestNotificationKey"

object:nil queue:nil usingBlock:^(NSNotification \*n) {

NSLog(@"%@", self);

}];

}

- (void)dealloc

{

if (\_observer) {

[[NSNotificationCenter defaultCenter] removeObserver:\_observer];

}

}

使用\_\_weak KSViewController \* wself = self; 打破循环引用，同样问题在NSTimer中也经常遇到。

## 案例4

- (id) getBlockArray

{

int val = 10;

return [[NSArray alloc] initWithObjects:

^{ KSLog(@" > block 0:%d", val); }, // block on the stack

^{ KSLog(@" > block 1:%d", val); }, // block on the stack

nil];

}

- (void)testManageBlockMemory

{

id obj = [self getBlockArray];

typedef void (^BlockType)(void);

BlockType blockObject = (BlockType)[obj objectAtIndex:0];

blockObject();

}

// return [[NSArray alloc] initWithObjects:

// [^{ KSLog(@" > block 0:%d", val); } copy], // block copy to heap

// [^{ KSLog(@" > block 1:%d", val); } copy], // block copy to heap

// nil];

执行上面的代码中，在调用 testManageBlockMemory 时，程序会 crash 掉。因为从 getBlockArray 返回的 block 是分配在 stack 上的，但超出了定义 block 所在的作用域，block 就不在了。正确的做法（被屏蔽的那段代码）是在将 block 添加到 NSArray 中时先 copy 到 heap 上，这样就可以在之后的使用中正常访问。

在 ARC 下，对 block 变量进行 copy 始终是安全的，无论它是在栈上，还是全局数据段，还是已经拷贝到堆上。对栈上的 block 进行 copy 是将它拷贝到堆上；对全局数据段中的 block 进行 copy 不会有任何作用；对堆上的 block 进行 copy 只是增加它的引用记数。

如果栈上的 block 中引用了\_\_block 类型的变量，在将该 block 拷贝到堆上时也会将 \_\_block 变量拷贝到堆上如果该 \_\_block 变量在堆上还没有对应的拷贝的话，否则就增加堆上对应的拷贝的引用记数。

## 案例5

1. @property (nonatomic, strong) NSString \*string1;
2. @property (nonatomic, unsafe\_unretained) NSString \*string2;
3. self.string1 = @"String 1";
4. self.string2 = self.string1;
5. self.string1 = nil;
6. NSLog(@"String 2 = %@", self.string2);

ARC下的内存管理

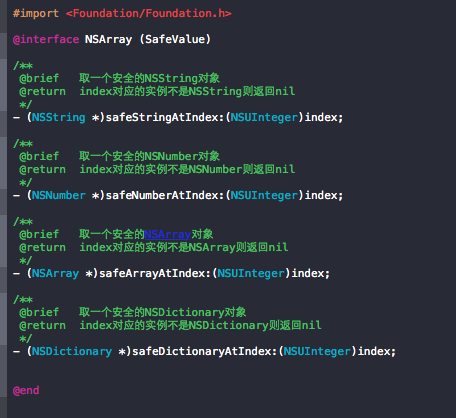
| 属性值 | 关键字 | 所有权 |
| --- | --- | --- |
| strong | \_\_strong | 有 |
| weak | \_\_weak | 无（iOS5~) |
| unsafe\_unretained | \_\_unsafe\_unretained | 无 |
| copy | \_\_strong | 有 |
| assign | \_\_unsafe\_unretained | 无 |
| retain | \_\_strong | 有 |

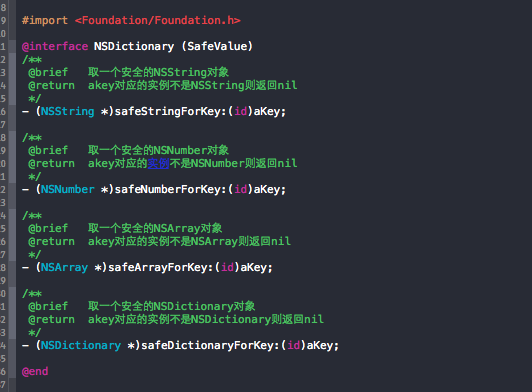
## 案例6

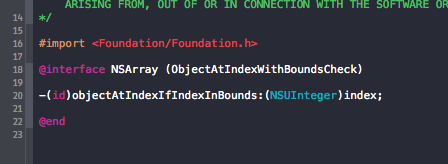
weak/strong, copy/mutableCopy, deepCopy/mutableDeepCopy

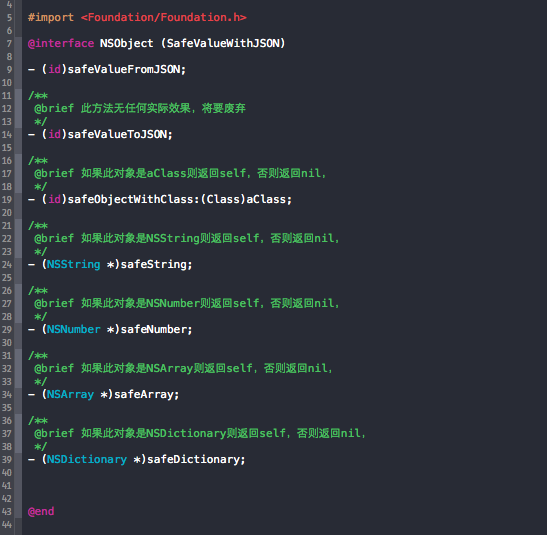
## 案例7

使用带有安全检查的扩展方法









例如：

addressInfo.telnum = [array objectAtIndex:0]; //Bad

addressInfo.telcell = [array objectAtIndex:1]; //Bad

addressInfo.telnum = [array objectAtIndexIfIndexInBounds:0]; //Good

addressInfo.telcell = [array objectAtIndexIfIndexInBounds:1]; //Good

[dictionary setObject:value forKey:key]; //Bad

if (value) {

[dictionary setObject:value forKey:key]; // Good1

}

[dictionary setObject:value ? value : @”” forKey:key]; // Good2

- (BOOL)parseContent:(NSString \*)

{

BOOL result = NO;

NSError \*error = nil;

NSDictionary \*dicContent = [NSJSONSerialization JSONObjectWithString:content

options:0

error:&error];

if (!error && ([[dicContent objectForKey:@”err”] intValue] == 0)) {

result = YES;

}

return result;

} //Bad

- (BOOL)parseContent:(NSString \*)

{

BOOL result = NO;

NSError \*error = nil;

NSDictionary \*dicContent = [NSJSONSerialization JSONObjectWithString:content

options:0

error:&error];

if (!error && [dicContent isKindOfClass:[NSDictionary class]] && ([[dicContent safeNumberForKey:@”err”] intValue] == 0)) {

result = YES;

}

return result;

} //Good

self.topicId = [[dicContent objectForKey:@"data"] integerValue]; //Bad

self.topicId = [[dicContent safeStringForKey:@"data"] integerValue]; //Good

# 常见的坏味道

## Macintosh HD:Users:Ben:Desktop:Untitled.png

## 魔鬼数字

定义命名清晰易懂的常量或变量并放到合适作用域的地方

## 重复代码

重新放置 (全局的常量，变量，方法)

提取方法(界面中刷新布局逻辑，界面计算逻辑，安全检查的代码，提取全能方法)

放入基类

提取模版方法

提取类

## 过长的判断语句

引入几个临时变量，通过变量名容易读懂逻辑

拆分成小的方法，通过方法名容易读懂逻辑

## 过长的参数列表

不要传递方法可自己获取的参数

利用/引入参数结构体或对象

## 过长函数

分解方法

外包方法（外包给category或工具类，或新建的类）

## 过大的类

引入新的类

引入子类

## 过多的临时变量

引入新的类（Model, View, Manager……)

## 多处相同分支判断

引入子类使用多态

引入新的基类和子类，使用多态，让当前类使用（策略模式）

前两条一起（桥接模式）

## 不正确的注释，翻译性的注释

## 继承，包装，category使用的选择

category中尽量不要掺杂任何业务逻辑，不依赖其它类

在简单的重写某些方法即可达到目的时，尽量使用继承，注意重写相关的基类方法