|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 修订记录 | | | | | | | | | | | | |
| 版 本 | 修 订 描 述 | | | | | | | | | | 修订日期 | |
| A0 | 初次建立 | | | | | | | | | | 2020/1/18 | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
|  |  | | | | | | | | | |  | |
| 评审 | | | | | | | | | | | | |
| 部门 | | 评审 | | | 部门 | | 评审人 | | 部门 | | | 评审人 |
| 平台研发中心 | | 程绪伟 | | | TrackSolid 产品PDT经理 | | 陈新军 | | 有线定位产品PDT经理 | | | 张宁 |
| 项目管理中心 | | 段小权 | | | ISD产品PDT 经理 | | 王胜 | | 无线定位产品PDT经理 | | | 王阳 |
| 规划中心 | | 郑旻琪 | | | Tracker产品PDT经理 | | 李胜如 | | 个人穿戴产品PDT经理 | | | 龚超 |
| 云平台智慧车联产品 | | 武宇飞 | | | DVR 产品PDT经理 | | 赵志文 | | 几米圈产品PDT经理 | | | 林时配 |
| 分发部门 | | | | | | | | | | | | |
| PDT管理 | | | 规划中心 | | | 项目管理中心 | | | | 国内营销--解决方案部 | | |
| 研发中心-测试中心 | | | 研发中心--云平台 | | | 国内营销-研发管理中心 | | | |  | | |
| 编制: 谢洪涛 | | | | 审核:－－ | | | | 批准:程绪伟 | | | | |

**目录**

**[1. 引言 3](#_Toc536016739)**

[1.1. 更新记录](#_Toc536016740) **[3](#_Toc536016740)**

**[2. 编程规约 3](#_Toc536016741)**

[2.1. 命名规约](#_Toc536016742) **[4](#_Toc536016742)**

[2.2. 常量定义](#_Toc536016743) **[6](#_Toc536016743)**

[2.3. 格式规约](#_Toc536016744) **[7](#_Toc536016744)**

[2.4. OOP规约](#_Toc536016745) **[12](#_Toc536016745)**

[2.5. 控制语句](#_Toc536016746) **[11](#_Toc536016746)**

[2.6. 注释规约](#_Toc536016747) **[14](#_Toc536016747)**

[2.7. swift规约](#_Toc536016748) **[15](#_Toc536016748)**

[2.8. 其他](#_Toc536016749) **[22](#_Toc536016749)**

[2.9. 编程习惯](#_Toc536016750) **[23](#_Toc536016750)**

**1.0引言**

**1.1更新记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **更新人** | **更新日期** | **说明** |
| **1.0** | 陈梦宣 | 2018/05/02 | 新建 |
| **1.1** | 李治健 | 2019/01/23 | 补充及完善 |
| **1.2** | 李治健 | 2020/01/13 | 添加仓库规范，完善编程思维 |

**2.0编程规约**

**2.1命名规约**

2.1.1【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。

反例： \_name / \_\_name / $Object / name\_ / name$ / Object$

2.1.2【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。

说明：正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，即使纯拼音命名

方式也要避免采用。

反例： DaZhePromotion [打折] / getPingfenByName() [评分] / int某变量 = 3

正例： alibaba / taobao / youku / hangzhou等国际通用的名称，可视同英文。

2.1.3【强制】类名使用UpperCamelCase风格，必须遵从大驼峰形式。

正例：MarcoPolo / XmlService / TcpUdpDeal / TaPromotion

反例：macroPolo / XMLService / TCPUDPDeal / TAPromotion

2.1.4【强制】方法名、参数名、成员变量、局部变量都统一使用lowerCamelCase风格，必须

遵从小驼峰形式。

正例： localValue / getHttpMessage() / inputUserId

2.1.5【强制】数值型常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要

嫌名字长。

正例： MAX\_STOCK\_COUNT

反例： MAX\_COUNT

2.1.6【强制】抽象类命名使用Abstract或Base开头；异常类命名使用Exception结尾；测试

类命名以它要测试的类的名称开始，以Test结尾。

2.1.7【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一

使用单数形式，但是类名如果有复数含义，类名可以使用复数形式。

正例：应用工具类包名为com.alibaba.open.util、类名为MessageUtils（此规则参考

spring的框架结构）

2.1.8【强制】杜绝完全不规范的缩写，避免望文不知义。

说明：类名、结构体、枚举等定义不允许类名使用此类缩写(遵守第14条)；

内部变量名称可允许程序员间可通用、可识别的缩写(允许但不推荐)，至少能确保缩写

能被明确知晓具体含义。比如:

允许：var myHomeController: HomeController? -> var myHomeCtl: HomeController?

2.1.9【推荐】如果使用到了设计模式，建议在类名中体现出具体模式。

说明：将设计模式体现在名字中，有利于阅读者快速理解架构设计思想。

正例：public class OrderFactory;

public class LoginProxy;

public class ResourceObserver;

2.1.10【参考】枚举类名建议带上Enum后缀，枚举成员名称需要全大写，单词间用下划线隔开。

说明：枚举其实就是特殊的常量类，且构造方法被默认强制是私有。

正例：枚举名字：DealStatusEnum，成员名称：SUCCESS / UNKOWN\_REASON。

2.1.11【强制】对于不能暴露的方法和属性一律加private或fileprivate修饰

例：private和fileprivate区别

private：访问级别所修饰的属性或者方法只能在当前类里访问。

fileprivate：访问级别所修饰的属性或者方法在当前的Swift源文件里可以访问

2.1.12【强制】首字母缩略词在命名中一般来说都是全部大写，例外的情形是如果首字母缩略

词是一个命名的开始部分，而这个命名需要小写字母作为开头，这种情形下首字母缩略

词全部小写。

// "HTML" 是变量名的开头, 需要全部小写 "html"

let htmlBodyContent: String = "<p>Hello, World!</p>"

// 推荐使用 ID 而不是 Id

let profileID: Int = 1

// 推荐使用 URLFinder 而不是 UrlFinder

class URLFinder {

/\* ... \*/

}

2.1.13【强制】使用前缀 k + 大骆驼命名法 为所有非单例的静态常量命名。

class MyClassName {

// 基元常量使用 k 作为前缀

static let kSomeConstantHeight: CGFloat = 80.0

// 非基元常量也是用 k 作为前缀

static let kDeleteButtonColor = UIColor.redColor()

// 对于单例不要使用k作为前缀

static let sharedInstance = MyClassName()

/\* ... \*/

}

2.1.14【强制】命名应该具有描述性 和 清晰的。

// 推荐

class RoundAnimatingButton: UIButton { /\* ... \*/ }

// 不推荐

class CustomButton: UIButton { /\* ... \*/ }

2.1.15【强制】当给函数参数命名时，要确保函数能理解每个参数的目的。

//推荐

- (void)configWithkey:(NSString \*\_Nonnull)key secret:(NSString \*\_Nonnull)secret

//不推荐

- (void)config:(NSString \*\_Nonnull)key :(NSString \*\_Nonnull)srt;

2.1.16【推荐】obj-C枚举成员需要统一的起始名称，多用下划线区分类型或功能，尽量使用大

写或大驼峰命名。Swift则不需要按照则规范，但是命名需要体现成员的功能或状态。

/\*播放器视频播放状态\*/

typedef enum : NSUInteger {

STREAM\_VIDEO\_STATUS\_NONE = 0, //未初始化状态

STREAM\_VIDEO\_STATUS\_PREPARE, //正在准备播放

STREAM\_VIDEO\_STATUS\_START, //正在播放

STREAM\_VIDEO\_STATUS\_STOP, //播放结束

} STREAM\_PLAY\_STATUS;

**2.2常量定义**

2.2.1【强制】不允许出现任何魔法值（即未经定义的常量）直接出现在代码中。

反例： String key="Id#taobao\_"+tradeId；

cache.put(key, value);

2.2.2【推荐】建议使用类型常量，不建议使用#define预处理命令。

首先比较一下这两种声明常量的区别：

预处理命令：简单的文本替换，不包括类型信息，并且可被任意修改。

类型常量：包括类型信息，并且可以设置其使用范围，而且不可被修改。

使用预处理虽然能达到替换文本的目的，但是本身还是有局限性的：

不具备类型信息。

可以被任意修改。

2.2.3【强制】~~常量以相关类名作为前缀~~。常量或变量前面需要体现使用功能或需求，后面需要

添加具体类型。

推荐这样写：

OC中：

~~static const NSTimeInterval HomeViewControllerFadeOutAnimationDuration = 0.4;~~

~~Swift中：~~

~~Let HomeViewControllerFadeOutAnimationDuration = 0.4;~~

static const NSTimeInterval fadeOutAnimationDuration = 0.4;或 static const NSTimeInterval fadeOutAnimationTimeInterval = 0.4;

@property (nonatomic,strong) NSArray bookArray;

Swift中：

let fadeOutAnimationDuration = 0.4;

或 let fadeOutAnimationTimeInterval = 0.4;

var bookArray: NSArray?;

不推荐这样写：

OC中：

static const NSTimeInterval fadeOutTime = 0.4;

@property (nonatomic,strong) NSArray books;

或@property (nonatomic,strong) NSArray bookArr;

Swift中：

let fadeOutTime = 0.4;

var bookArr: NSArray? 或var books: NSArray?

**2.3格式规约**

2.3.1【强制】大括号的使用约定。如果是大括号内为空，则简洁地写成{}即可，不需要换行；

如果是非空代码块则：

1）左大括号前不换行。

2）左大括号后换行。

3）右大括号前换行。

4）右大括号后还有else等代码则不换行；表示终止右大括号后必须换行。

正例：

class SomeClass {

func someMethod() {

if x == y {

/\* ... \*/

} else if x == z {

/\* ... \*/

} else {

/\* ... \*/

}

}

/\* ... \*/

}

2.3.2【强制】左括号和后一个字符之间不出现空格；同样，右括号和前一个字符之间也不出现空格。详见第5条下方正例提示。

2.3.3【强制】if/for/while/switch/do等保留字与左右括号之间都必须加空格。

2.3.4【强制】任何运算符左右必须加一个空格。

说明：运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号、三目运行符等。

2.3.5【推荐】没有必要增加若干空格来使某一行的字符与上一行的相应字符对齐。

正例：

int a = 3;

long b = 4L;

float c = 5F;

StringBuffer sb = new StringBuffer();

说明：增加sb这个变量，如果需要对齐，则给a、b、c都要增加几个空格，在变量比较多

的情况下，是一种累赘的事情。

2.3.6【推荐】方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语

义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

说明：没有必要插入多行空格进行隔开。

2.3.7【推荐】遵守Xcode内置的缩进格式（包括数组的缩进）。如果已经遵守，按下CTRL+i 进

行格式化。

2.3.8【推荐】使用类型定义简洁的版本，而不是全称通用语法。

例：推荐做法：

1 var deviceModels: [String]

2 var employees: [Int: String]

3 var faxNumber: Int?

不推荐做法：

1 var deviceModels: Array<String>

2 var employees: Dictionary<Int, String>

3 var faxNumber: Optional<Int>

2.3.9【推荐】通知标识的命名，k+事件命名+Notification。

2.3.10【强制】确保每个文件结尾都有空白行。

2.3.11【强制】 确保每行都不以空白字符作为结尾。

Xcode->Preferences->Text Editing->Automatically trim trailing whitespace + Including whitespace-only lines。

2.3.12【强制】分号前不添加空格，分号后必须加空格。

当在写一个变量类型，一个字典里的主键，一个函数的参数，遵从一个协议，或一个父

类，不用在分号前添加空格。

// 指定类型

let pirateViewController: PirateViewController

// 字典语法(注意这里是向左对齐而不是分号对齐)

let ninjaDictionary: [String: AnyObject] = [

"fightLikeDairyFarmer": false,

"disgusting": true

]

// 调用函数

someFunction(someArgument: "Kitten")

// 父类

class PirateViewController: UIViewController {

/\* ... \*/

}

2.3.13【强制】在逗号后面加空格。

let myArray = [1, 2, 3, 4, 5]

2.3.14【推荐】当遇到需要处理的数组或字典内容较多是应多行显示，需把 [ 和 ] 类似于方法

体里的括号，方法体里的闭包也要做类似处理。

ssomeFunctionWithABunchOfArguments(

someStringArgument: "hello I am a string",

someArrayArgument: [

"dadada daaaa daaaa dadada daaaa daaaa dadada daaaa daaaa",

"string one is crazy - what is it thinking?"

],

someDictionaryArgument: [

"dictionary key 1": "some value 1, but also some more text here",

"dictionary key 2": "some value 2"

],

someClosure: { parameter1 in

print(parameter1)

})

2.3.15【推荐】当添加类的成员或属性时，尽量将属性归类并空行处理，而不是简单再后面追加，

且添加相关解释说明。

class JMRNEngineManager: RCTEventEmitter {

static var jmRNEngineManager: JMRNEngineManager? //全局静态的RN引擎模块管理类

var isHasListeners = false //JS是否有监听事件

let kRNSendJSCameraStateValue = JMH5BindMethodCode.getCameraState.rawValue //JS代码中监听事件kRNSendJSCameraState对应的接口名称getCameraState

let kRNSendJSCameraInfoValue = JMH5BindMethodCode.getCameraInfo.rawValue //JS代码中监听事件kRNSendJSCameraInfo对应的接口名称getCameraInfo

...

}

2.3.16【强制】成员函数直接需要用空格断开，禁止代码块出现连续。

public class MyBook: NSObject {

//......

/// 获取最喜欢的书的名称

///

/// - **Returns**: 书名字符串String，可能为nil

func getFavoriteBookName() -> String? {

return "Swift ++"

}

/// 获取书的数量

///

/// - **Returns**: 数量

func getCount() -> Int {

return bookArray?.count

}

//......

}

**禁止：**

public class MyBook: NSObject {

//......

func getFavoriteBookName() -> String? {

return "Swift ++"

}

func getCount() -> Int {

return bookArray?.count

}

//......

}

2.3.17【强制】在框架导入代码块之后后面需要空一行。

import Foundation

public class JMRNEngineTool: NSObject {

//......

}

**2.4 OOP规约**

2.4.1【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

2.4.2【强制】对外暴露的接口签名，原则上不允许修改方法签名，避免对接口调用方产生影响。

接口过时必须加DEPRECATED（…）注解，并清晰地说明采用的新接口或者新服务是什么。

2.4.3【强制】序列化类新增属性时，请不要修改serialVersionUID字段，避免反序列失败；如果完全不兼容升级，避免反序列化混乱，那么请修改serialVersionUID值。

2.4.4【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在init方法中。

2.4.5【推荐】使用索引访问用String的split方法得到的数组时，需做最后一个分隔符后有无

内容的检查，否则会有越界的风险。

说明：

let str = "a,b,c,,";

let ary = str.split(",");

//预期大于3，结果是3

print(ary.length);

**2.5控制语句**

2.5.1【强制】在if/else/for/while/do代码块中必须使用大括号，即使只有一行代码，也需要。

不推荐：if (bCondition) statements1

else statements2

推荐：

if (bCondition) {

statements1

} else {

statements2

}

2.5.2【强制】在接口代码块中顶层是使用条件选择语句时，推荐尽量少用else，在if-else的

方式可以改写成：

if(condition){

...

return obj;

}

//接着写else的业务逻辑代码;

说明：如果非得使用if()...else if()...else...方式表达逻辑，请勿超过 3层，超过请

使用状态设计模式。

正例：逻辑上超过 3层的if-else代码可以使用Swith语句。

2.5.3【强制】除常用方法（如 getXxx/isXxx）等外，不要在条件判断中执行其它复杂的语句，

将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明：很多 if语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确

什么样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢？

正例：

Bool existed = (file.open(fileName, "w") != nil) && (...) || (...);

if (existed) {

...

}

反例：

if ((file.open(fileName, "w") != null) && (...) || (...)) {

...

}

**2.6注释规约**

2.6.1【强制】所有的类都必须添加创建者信息。

2.6.2【强制】方法内部单行注释，在被注释语句上方另起一行(可以再代码后进行//注释)，使

用//注释。方法内部多行注释使用/\* \*/注释，注意与代码对齐。

2.6.3【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

2.6.4【强制】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心

逻辑等的修改。

说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严

重滞后，就失去了导航的意义。

2.6.5【强制】注释掉的代码尽量要配合说明，而不是简单的注释掉。

说明：代码被注释掉有两种可能性：1）后续会恢复此段代码逻辑。2）永久不用。前者如

果没有备注信息，难以知晓注释动机。后者建议直接删掉（代码仓库保存了历史代

码）

2.6.6【强制】注释模块中的空行不要使用 \* 来占位。

2.6.7【推荐】对于负责的类，在描述类的使用方法时可以添加一些合适的例子，请注意Swift

注释是支持 MarkDown 语法的。

/\*\*

## 功能列表

这个类提供下一下很赞的功能，如下:

- 功能 1

- 功能 2

- 功能 3

## 例子

这是一个代码块使用四个空格作为缩进的例子。

let myAwesomeThing = MyAwesomeClass()

myAwesomeThing.makeMoney()

## 警告

使用的时候总注意以下几点

1. 第一点

2. 第二点

3. 第三点

\*/

class MyAwesomeClass {

/\* ... \*/

}

2.6.8【强制】使用注释 // MARK: - xoxo 时, 下面一行保留为空行。

class Pirate {

// MARK: - 实例属性

private let pirateName: String

// MARK: - 初始化

init() {

/\* ... \*/

}

}

2.6.9【强制】SDK封装的接口原则上不更改，不调整，若需要调整或更改参数时，尽量保持原先的接口，新增新接口，并对原先接口进行NS\_DEPRECATED标识(Swift用DEPRECATED标识)。

2.6.10【强制】对不需要的代码或接口不能进行直接删除，需屏蔽了之后添加注释说明并提交

更新。对永久不用的代码删除时，原则上只允许删除自己屏蔽的代码，删除其他人屏蔽

的代码需进行沟通。

**2.7swift规约**

2.7.1【推荐】能用 let 尽量用 let 而不是 var。

尽可能的用 let foo = ... 而不是 var foo = ...（并且包括你疑惑的时候）。万不得已

的时候，再用 var（就是说：你 知道 这个值会改变，比如：有 weak 修饰的存储变量）。

理由： 这俩关键字 无论意图还是意义 都很清楚了，但是 let 可以产生安全清晰的代码。

let-有保障 并且它的值的永远不会变,对程序猿也是个 清晰的标记，对于它的用法，之后的代码可以做个强而有力的推断。猜测代码更容易了。不然一旦你用了 var，还要去推测值会不会变，这时候你就不得不人肉去检查。这样，无论何时你看到 var，就假设它会变，并问自己为啥。

2.7.2【强制】避免对 可选类型 强解包。

如果你有个 FooType? 或 FooType! 的 foo，尽量不要强行展开它以得到基本类型

（foo!）。

这样写：

if let foo = foo {

// Use unwrapped `foo` value in here

} else {

// If appropriate, handle the case where the optional is nil

}

或者使用可选链，比如：

// Call the function if `foo` is not nil. If `foo` is nil, ignore we ever tried to make the call

foo?.callSomethingIfFooIsNotNil()

理由： if let 绑定可选类型产生了更安全的代码，强行展开很可能导致运行时崩溃。

2.7.3【强制】避免毫无保留地展开可选类型。

如果 foo 可能为 nil ，尽可能的用 let foo: FooType? 代替 let foo: FooType!（注意：一般情况下，?可以代替!） 理由: 明确的可选类型产生了更安全的代码。无保留地展开可选类型也会挂。

2.7.4【推荐】首选 structs 而非 classes。

除非你需要 class 才能提供的功能（比如 identity 或 deinitializers），不然就用

struct要注意到继承通常不是用 类 的好理由，因为多态可以通过协议实现，重用可以

通过组合实现。比如，这个类的分级:

class Vehicle {

let numberOfWheels: Int

init(numberOfWheels: Int) {

self.numberOfWheels = numberOfWheels

}

func maximumTotalTirePressure(pressurePerWheel: Float) -> Float {

return pressurePerWheel \* Float(numberOfWheels)

}

}

class Bicycle: Vehicle {

init() {

super.init(numberOfWheels: 2)

}

}

class Car: Vehicle {

init() {

super.init(numberOfWheels: 4)

}

}

可以重构成酱紫：

protocol Vehicle {

var numberOfWheels: Int { get }

}

func maximumTotalTirePressure(vehicle: Vehicle, pressurePerWheel: Float) -> Float {

return pressurePerWheel \* Float(vehicle.numberOfWheels)

}

struct Bicycle: Vehicle {

let numberOfWheels = 2

}

struct Car: Vehicle {

let numberOfWheels = 4

}

理由: 值的类型更简单，容易辨别，并且通过let关键字可猜测行为。

2.7.5【推荐】参数化类型(模板类型)能不写类型参数的就别写了。

参数化类型的方法可以省略接收者的类型参数，当他们对接收者来说一样时。比如：

struct Composite<T> {

…

func compose(other: Composite<T>) -> Composite<T> {

return Composite<T>(self, other)

}

}

可以改成这样：

struct Composite<T> {

…

func compose(other: Composite) -> Composite {

return Composite(self, other)

}

}

2.7.6【强制】在闭包里使用 self 的时候要注意出现循环引用，使用捕获列表可以避免这一点。

myFunctionWithClosure() { [weak self] (error) -> Void in

// 方案 1

self?.doSomething()

// 或方案 2

guard let strongSelf = self else {

return

}

strongSelf.doSomething()

}

2.7.7【推荐】Switch 模块中不用显式使用break。

2.7.8【强制】断言流程控制的时候不要使用小括号。

// 推荐

if x == y {

/\* ... \*/

}

// 不推荐

if (x == y) {

/\* ... \*/

}

2.7.9【推荐】在写枚举类型的时候，尽量简写。

// 推荐

imageView.setImageWithURL(url, type: .person)

// 不推荐

imageView.setImageWithURL(url, type: AsyncImageView.Type.person)

2.7.10【强制】在使用类方法的时候不能简写，因为类方法不如 枚举 类型一样，可以根据轻

易地推导出上下文。

// 推荐

imageView.backgroundColor = UIColor.whiteColor()

// 不推荐

imageView.backgroundColor = .whiteColor()

2.7.11【强制】在使用一些语句如 else，catch等紧随代码块的关键词的时候，确保代码块和关键词在同一行。下面 if/else 和 do/catch 的例子.

if someBoolean {

// 你想要什么

} else {

// 你不想做什么

}

do {

let fileContents = try readFile("filename.txt")

} catch {

print(error)

}

2.7.12【推荐】自定义操作符

不推荐使用自定义操作符，如果需要创建函数来替代。

2.7.13【推荐】如果对于一个变量你不打算声明为可选类型，但当需要检查变量值是否为 nil，推荐用当前值和 nil 直接比较，而不推荐使用 if let 语法。

// 推荐

if someOptional != nil {

// 你要做什么

}

// 不推荐

if let \_ = someOptional {

// 你要做什么

}

2.7.14【推荐】不要使用 unowned，unowned 和 weak 变量基本上等价，并且都是隐式拆包

( unowned 在引用计数上有少许性能优化)，由于不推荐使用隐式拆包，也不推

荐使用unowned 变量。

// 推荐

weak var parentViewController: UIViewController?

// 不推荐

weak var parentViewController: UIViewController!

unowned var parentViewController: UIViewController

2.7.15【推荐】当拆包取值时，使用和被拆包取值变量相同的名称。

guard let myVariable = myVariable else {

return

}

2.7.16【强制】在创建类常量的时候，使用 static 关键词修饰。

class MyTableViewCell: UITableViewCell {

static let kReuseIdentifier = String(MyTableViewCell)

static let kCellHeight: CGFloat = 80.0

}

2.7.17【推荐】在解析可选类型时，推荐使用 guard 语句，而不是 if 语句，因为 guard 语

句可以减少不必要的嵌套缩进。

// 推荐

guard let monkeyIsland = monkeyIsland else {

return

}

bookVacation(onIsland: monkeyIsland)

bragAboutVacation(onIsland: monkeyIsland)

// 不推荐

if let monkeyIsland = monkeyIsland {

bookVacation(onIsland: monkeyIsland)

bragAboutVacation(onIsland: monkeyIsland)

}

// 禁止

if monkeyIsland == nil {

return

}

bookVacation(onIsland: monkeyIsland!)

bragAboutVacation(onIsland: monkeyIsland!)

2.7.18【推荐】当解析可选类型需要决定在 if 语句 和 guard 语句之间做选择时，最重要的

判断标准是是否让代码可读性更强，实际项目中会面临更多的情景，如依赖 2

个不同的布尔值，复杂的逻辑语句会涉及多次比较等，大体上说，根据你的判

断力让代码保持一致性和更强可读性， 如果你不确定 if 语句 和 guard 语句

哪一个可读性更强，建议使用 guard 。

// if 语句更有可读性

if operationFailed {

return

}

// guard 语句这里有更好的可读性

guard isSuccessful else {

return

}

// 双重否定不易被理解 - 不要这么做

guard !operationFailed else {

return

}

2.7.19【推荐】如果需要在2个状态间做出选择，建议使用if 语句，而不是使用 guard 语句。

// 推荐

if isFriendly {

print("你好, 远路来的朋友！")

} else {

print("穷小子，哪儿来的？")

}

// 不推荐

guard isFriendly else {

print("穷小子，哪儿来的？")

return

}

print("你好, 远路来的朋友！")

**2.8其他**

2.8.1【强制】缓存路径

Documents：应用中用户数据可以放在这里，iTunes备份和恢复的时候会包括此目录

tmp：存放临时文件，iTunes不会备份和恢复此目录，此目录下文件可能会在应用退出后

删除

Library/Caches：存放缓存文件，iTunes不会备份此目录，此目录下文件不会在应用退

出删除

2.8.2 【强制】为了避免文件杂乱，物理文件应该保持和 Xcode 项目文件同步。Xcode 创建的

任何组（group）都必须在文件系统有相应的映射。为了更清晰，代码不仅应该按照类型

进行分组，也可以根据业务功能进行分组

# 2.9 编程习惯

# 编程思维

1、应遵循模块化、功能化原则，降低耦合，提高模块及框架的可扩展性；  
2、应遵循面向对象、面向函数式编程，少用全局类型变量，少用临时参数；  
3、任何选择型代码块需要全部考虑选择类型，任何返回型代码块需要考虑所有返回的可能性；  
4、功能逻辑或需求应考虑所有异常或非正常调用及请求，而非简单的完成需求。  
5、一般情况下，任何请求和调用应该有相应的回调或反馈；  
6、你敲的每行代码，应思考是否是最优代码，是否是冗余代码，是否可以改善？

7、多与其他人沟通，多探讨代码逻辑、代码的严谨性；

8、新增功能及bug修复，应考虑修改的代码是否会影响其他功能；

# 代码仓库规范

**master**：主分支，用于备份及查看；  
**dev**：开发分支，用于日常项目开发及调试；  
**release**：项目结版及后期维护拉取代码发布等；  
**test**：日常测试及无关项目功能的临时使用分支；

1、项目代码在发版、重大更改、重要功能添加，都需要进行代码提交；  
2、在不影响项目其他功能及模块的原则上，必须每日提交代码推送远程仓库分支；  
3、提交代码，必须写清楚修改的内容，包括：新增或修改了什么功能、调整了什么地方、修 复了什么BUG等；  
4、项目发版必须打版本号Tag；  
5、在项目发版或重大更新及调整之后，需要发起master分支合并请求，用SE主导；

6、重要更新及发版之后，需要发起主分支Master合并请求；