# 第零章 文档说明

该文档为iOS开发规范，包括基本编程规约、数据模型、日志处理、结构组织、数据缓存、第三方框架、辅助工具共七章，每章节包括若干小节详规范说明。该文档目的是为了统一基本编程风格和基本规范，便于高效高质的代码，更加有利于团队成员之间的沟通，增加代码可读性和降低其他同事阅读难度。

# 第一章 编程规范

## 一、命名规范

1. 驼峰式命名方法。变量名，方法名都必须遵循，以小写字母开头，英文单词进行拼接，首字母大写。宏定义为全大写，常量大写开头。但专有名字可以全部大写，不受限制，如HTTP、TCP、UDP等，只有表示专有名称时候才有效。
2. 命名必须具有意义。尽量避免缩写，Objective-C语言官方提供的标准是行为动词加名词进行组合为方法名称，变量名为表示的意义单词组合起来，添加行为动词组合后可以通顺阅读并能理解含义。
3. 前缀。如果过需要添加模块或项目标识，可以添加前缀，如SDWebImage框架的前缀是sd\_，用小写加下划线\_，IQKeyboardManager框架的前缀为IQ，采用两个缩写大写字母标记。
4. 不要与系统框架保留字冲突，如CG、NS、UI、CA等，这些都是被已知的系统框架，即使扩展系统框架也需要与系统的区分。
5. 尽量不要使用下划线做前缀，系统私有函数都是单(\_)或双(\_\_)下划线，可能会导致不可预见的逻辑甚至严重的bug。
6. 分类命名遵循变量命名规则，建议添加前缀以作区分。

// 正确示范

@property (nonatomic, strong) NSString \* title;

// 错误示范

@property (nonatomic, strong) NSString \* Date;

## 二、命名规则

1. 变量命名必须具有意义，规避无意义的变量名，如i、j、k等，只有在临时作用域内的临时变量可以使用，不能影响到阅读，一般只有简单的逻辑才可以使用。
2. 常量一般可以分为两种，全局使用和模块内。全局常量，一般是整个APP系统需要使用的，如当前环境变量，系统配置信息。模块内，使用常量供当前内部使用或对外暴露的变量，如NSNotification的名称。如果是全局的可以用单独文件，业务较多可以进行拆分成多个。
3. 宏一般定义为基础数据类型，与常量区别是宏仅仅做替换，没有内存分配。比如获取当前一些界面布局尺寸，分页数据大小，线程总数等等，大部分宏定义目的是与一些编译相关的信息。
4. 方法遵循表义功能，通过其名可以清楚了解其含义，常用行为动词有set、get、add、remove等。
5. 需要供外部使用方法和变量才写到.h文件，并且添加必要的注释，表达使用方法，在必要场景还需要说明可能出现的异常信息和特定状况下的使用说明。其他内部使用属性和方法都放在.m文件内。

## 三、书写规范

1. 每一行的长度设置为80(直接在Xcode里面设置Text Editing -> Editing -> Page guide at column)，便于阅读，在大部分计算机编程s语言里面设置行长度为80，过长会导致阅读不方便，可以使用换行。
2. 函数参数过多时候以:对齐左右k和v进行分行，在调用时候时候也可以遵循，Xcode目前都是支持直接空格换行对齐。
3. 大括号对齐{和}，在Objective-C里面建议是不换行，在if里面即使只有一行也要使用{}将内容包含，防止扩展时候出现条件遗漏导致逻辑被更改。
4. 集合类NSArray和NSDictionary在添加时候需要空格。

NSDictionary \* dict = @{ @"k": @"v", @"k1": @"v1"};

NSArray \* list = @[ @{"k": @"v"}, @{@"k1": @"v1"}];

1. 空属性修饰。目前有三种方式可以修饰是否为空，可以为空nullable、\_\_nullable、\_Nullable，不允许为空nonnull、\_\_nonnull、\_Nonnull，示例如下：

@property (nonatomic, strong, nullable) NSString \* title;

@property (nonatomic, strong) NSString \* \_\_nullable title;

@property (nonatomic, strong) NSString \* \_Nullable title;

其中nullable/nonull用法不一样(如果是双指针、block的返回值和参数则不能使用)，\_\_nullable/\_\_nonnull已经被最新的\_Nullable/\_Nonnull替代，在一些旧的项目里还存在。三者功能都是一样的，官方提供了两个宏可以直接指定默认为nonnull：

NS\_ASSUME\_NONNULL\_BEGIN

@interface myClass ()

@property (nonatomic, copy) NSString \*aString;

- (id)methodWithString:(nullable NSString \*)str;

@end

NS\_ASSUME\_NONNULL\_END

1. 判空处理。尽量在使用数据时候添加分类方法去判断，再调用系统方法，不建议直接使用runtime进行拦截处理。常见的有NSArray、NSDictionary、NSAttributeString进行存取操作时候都不允许为空，系统函数并未做判空处理，可以将对应的方法添加到分类做统一处理。
2. 常量命名。数字类型的使用枚举类型NS\_ENUM和NS\_OPTIONS，不要使用1.2.3去比较判断。其中NS\_OPTIONS使用的是bitmask，使用&和|进行判断，示例：

typedef NS\_OPTIONS(NSUInteger, SDWebImageDownloaderOptions) {

SDWebImageDownloaderLowPriority = 1 << 0,

SDWebImageDownloaderProgressiveLoad = 1 << 1,

SDWebImageDownloaderUseNSURLCache = 1 << 2,

}

// how to use

SDWebImageDownloaderOptions option;

if (option & SDWebImageDownloaderUseNSURLCache) {

// other

}

1. KVO和NSNotification必须主动清除。注入添加了对象观察或监听，必须手动清除，否则观察者销毁后，发送者会调用已销毁对象导致崩溃。
2. BOOL类型判断，不要直接使用YES == isGet进行判断，BOOL定义的是signed char类型，YES、NO定义的是1和0，如下示例：

int temp = 2;

if (temp == YES) {

NSLog(@"2 is YES");

} else {

NSLog(@"2 is NO");

}

输出结果为2 is NO，很显然不是我们想要的结果。

1. CoreFoundation和CoreGraphics框架的内存需要手动处理，对应的申请和释放接口都有提供，与C相关的部分内存不归Objective-C管理，所以ARC无效，并且与Objective-C赋值时候需要桥接(bridge)模式。

## 四、注释

1. 文档必须要有注释。如果有特殊场景或异常状况都需要写在文档说明里面，因业务扩展某些相似功能的页面因为命名问题可能导致含糊不清，如果没有文档说明很难区分具体调用场景。
2. 属性注释，因为命名规范已经可以体现含义，只需要简单说明即可，在特定场景或异常则需要另外添加说明。
3. 方法注释，简短说明作用，特殊场景或异常信息需要给出说明
4. 常见注释参数，一些工具可以自动生成对应的文档，可以查看官方[文档](https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/HeaderDoc/intro/intro.html)更详细说明，虽然较长时间没有更新，依然具有一定的参考意义。

* @brief：简短解释说明
* @discussion：详细的描述
* @param：参数说明
* @return：返回值说明
* @see：查看参考的字段或方法说明
* @sa：与see一样
* @code：嵌入示例代码
* @remark：附加的一些特殊说明

# 第二章 数据模型

主要是针对model类型，包含三种：JSON、自定义和界面布局模型。

## 一、JSON数据模型

一般是通过第三方包对服务器返回数据进行解析到model，属性可以通用处理，需要支持编码和解码以便保存到数据库或磁盘文件做缓存。

1. 属性名不能与保留字冲突，虽然可以编译但是不利于区分，可以在模型框架里面设置映射参数。
2. 在复制操作时候，默认是浅拷贝，导致在更改时候会直接修改到源数据值，如深拷贝需要实现NSCopying协议方法进行修改。
3. 模型命名遵循基本命名规范，可以添加model做后缀名以便统一区分。
4. model层应该包含网络请求和缓存数据处理，对外暴露是获取数据和操作数据，数据来源应该是透明的。
5. 如果需要使用泛型模式，可以通过协议功能，只需要实现该协议即可，否则会出现大量的警告信息
6. 针对同一个项目数据来源，后台一般具有通用格式，可以使用基础model统一继承。

## 二、数据库模型

本地缓存数据库，如sqlite和Realm缓存数据，需要将mode和数据库交互部分封装对外透明，建议加一层中间层，可以切换缓存类型不影响业务层的使用。

## 三、UI布局数据模型

1. 在比较复杂的界面布局时候可以使用model做辅助，可以将布局和计算逻辑分开，定义一定的规则即可适配大部分的布局调整。
2. 在做大量数据数据界面布局时候，也可以将布局模型进行缓存处理，减少计算量加快渲染速度。

# 第三章 日志处理

日志可以记录调试信息、数据状况和用户操作信息，在必要情况下将用户端的日志文件上传到服务器可以更加精准定位到特定场景下的系统崩溃信息，对系统稳定和版本覆盖都有极高的帮助。

## 一、调试日志

1. 调试阶段的日志只出现在测试包，提交到App Store时候不允许携带，大量的日志会影响到执行速度，可以使用DEBUG宏进行控制，也可以自定义其他参数，在编译里面进行设置，可以设置多个Configuration
2. 针对不同模块的日志，需要对外提供日志开关，特别是在多人开发时候，日志过多会影响到其他模块调试速度。
3. 调试日志在功能开发完成之后只需要保留必要的部分，否则也会造成过多累计，影响到其他部分。只有在功能开发或追踪bug时候才可能需要大量日志进行分析。

## 二、线上日志

只有极少特殊数据才需要在发布的release包里面打印，大部分的线上日志需要保存到文件或上传到服务器日志系统(如果有)。尽量不要在线上版本进行NSLog打印日志，还会存在敏感数据风险。

## 三、日志存储

1. 如有专门日志服务器，可以将一些特殊数据或出现错误时候的数据进行上传到服务器共分析，或者追踪在线bug的时候在某些特定版本也可以将必要的日志打包上传到服务器
2. 可以可用日志工具(如CocoaLumberjack)存储到本地文件，可以将崩溃信息一并保存，并提供对应界面让用户主动提交到服务器。
3. 在特殊情况下可能需要用户的全部日志，可以在经用户同意情况下直接上传手机上面的日志文件，里面系统有记录详细的报错信息以及手机当时所处环境。

# 第四章 结构组织

## 一、MVC设计模式

Apple默认提供的是MVC，对应的view为storyboard文件，大部分的逻辑都写在Controller里面，如果使用代码编写控件的时候会导致Controller更加臃肿，可以对控件进行封装减少控制器的负载。

## 二、MVVM设计模式

在业务功能逐渐增多时候，出现了MVVM替代MVC解放Controller部分功能，这其中较常用的是结合ReactiveObjC框架进行数据传递，将之前的事件用信号量方式进行替代。可以调整目录结构为

ViewController // 与控制器相关回调事件，导航栏部分

View //视图部分，包括布局

ViewModel // view和model之间的数据处理部分，回调view部分刷新界面，获取view事件拉取model的数据

Model // 包含网络获取数据，缓存数据和对数据操作

## 三、目录结构

下面的项目结构具有通用性，根据不同规模显示会有一定的差异，只显示了业务模块部分，并不是工程项目结构，未包含Pods、Frameworks和XXXTest等系统生成的部分。

* AppDelegate -- 系统入口信息，如果功能不断扩展也可以添加分类，如下添加调试信息和推送部分
  + AppDelegate+Debug.h
  + AppDelegate+Debug.m
  + AppDelegate+Push.h
  + AppDelegate+Push.m
* Main(Class) -- 主要业务模块，包含整个系统的业务部分代码，建议以首页模块划分为基准，然后随着业务扩展
  + Base -- 基础部分，包含BaseView、BaseModel、BaseController等基类信息
  + Tabbar -- 具有UITabBar项目
  + Home -- 首页信息
  + User -- 用户模块
* Component -- 组件部分，一般是与业务无关的部分封装成模块，如果模块具有更通用性可以跨项目时候，可以考虑提取出去封装为framework模式
  + Cache -- 缓存模块
  + Networking -- 网络模块
  + MapView -- 地图模块
* Category -- 分类，对原有累的功能扩展，如果功能具有明显的不同也可以扩展多个分类，下面只针对不同类在分组，防止目录深度扩展
  + NSString -- 对字符串的扩展，下面的示例显示的是对两个功能进行扩展
    - NSString+init.h
    - NSString+init.m
    - NSString+number.h
    - NSString+number.m
* Service -- 具有摸个独立功能可以单独设计成服务模式，如系统的定位模块、蓝牙模块、加密模块等，
* Vendor -- 第三方框架，虽然大部分都可以用Cocoapods和Carthage管理，部分厂家SDK并未发布到管理平台
* Supporting Files -- 系统默认创建的，包含入口storyboard、Assets、Info.plist文件
* Resource -- 资源文件，图片资源，文件资源，建议按照模块进行划分，有些图片具有公用属性可以单独分类

## 四、文档结构

1. 文档注释，一般系统会默认创建文档相关信息，也可以设置模板，在.h文件中系统信息之后需要添加该类的功能说明，如果有特殊使用场景和异常信息都要添加说明。
2. .h文件需要将对外暴露的属性和方法给出对应的注释。
3. .m文件内容结构

@interface Demo ()

@property (nonatomic, strong) NSString \* title;

@end

@implementation Demo

#pragma mark - Class method

+ (void)getIdentifier { }

#pragma mark - LifeCycle

// 系统初始化方法

//

- (void)dealloc {

\_title = nil;

}

// 系统方法

#pragma mark - Action

// 事件处理

#pragma mark - Public

// 对外方法

#pragma mark - Private

// 私有方法

#pragma mark - delegate

// 代理事件

#pragma mark - Properties/Var

// 属性setter/getter

@end

* 使用#pragma mark -进行模块划分
* 将dealoc放在初始化方法后面
* 在init和dealloc不要使用存取方法获取变量(.语法)，还未初始化使用可能引发不可预见的问题，同样在销毁时候也不要再使用self值
* 释放资源之后注意先后顺序

# 第五章 数据缓存

缓存包括内存和文件缓存，大部分项目都会继承图片框架和对应的缓存框架，对于底层部分已经做好封装，常见的SDWebImage、YYKit等都会有图片缓存。

## 一、图片缓存

1. 第三方图片框架，都集成了缓存处理模块，在特殊场景可能需要自定义下载处理，可以利用框架的缓存模块进行扩展。针对图片下载必须进入后台一队列模式，防止占用过多资源，建议提供对外的缓存大小统计和清除操作。
2. 磁盘缓存放到Cache目录，如果系统空间不足会自动清理缓存和临时目录文件，保证系统的稳定性。
3. 对图片缓存需要限制大小，不能无休止占用磁盘空间，设计对应自动清除缓存机制(如LRU)，保证在可控范围内。
4. 大部分图片缓存只有当前界面使用时候才需要保存到内存缓存NSCache，内存相对磁盘空间更小、更昂贵，尽量控制好内存缓存大小防止过大被系统杀掉。

## 二、数据缓存

1. 少量的参数数据可以直接保存到NSUserDefault，不建议写入过多数据，再同步时候回block住，保证写入完成。
2. 缓存数据可以保存到数据库(sqlite)和文件，对外提供统计信息和清除操作，需要添加自动计算和大小控制，除一些必要的数据可以保存在Document之外都必须保存在Cache或Temp目录。
3. 缓存文件操作注意并发问题，读写问题可能导致数据错乱或系统崩溃。
4. 大部分情况不建议将数据缓存到内存，一般都是解析到model供前台使用。
5. 不要将数据缓存到NSDictionary和NSArray，使用NSCache替代。

## 三、临时缓存

如有一些临时缓存可以放到内存管理器使用，统一控制大小、清除操作，如果需要可以保存到临时目录temp，在APP重启时候，上一次的数据都会被清除。

# 第六章 第三方框架

第三方包管理工具有Cocoapods和Carthage，swift自3.0以后开始自带Package Manager，直接连接到开源框架地址即可，示例如下：

// swift-tools-version:4.0

import PackageDescription

let package = Package(

name: "DeckOfPlayingCards",

products: [

.library(name: "DeckOfPlayingCards", targets: ["DeckOfPlayingCards"]),

],

dependencies: [

.package(url: "https://github.com/apple/example-package-fisheryates.git", from: "2.0.0"),

.package(url: "https://github.com/apple/example-package-playingcard.git", from: "3.0.0"),

],

targets: [

.target(

name: "DeckOfPlayingCards",

dependencies: ["FisherYates", "PlayingCard"]),

.testTarget(

name: "DeckOfPlayingCardsTests",

dependencies: ["DeckOfPlayingCards"]),

]

)

## 一、Cocoapods管理

系统安装Cocoapods，可以去官网[下载](https://cocoapods.org/)或使用sudo gem install cocoapods管理工具下载，初始化项目：

pod init

会生成.xcworkspace文件，包含该项目project和Pods.project，通过framework模式导入到项目里面，Podfile管理第三方包列表，可以配置不同模式和不同项目的依赖包，可以指定不同版本和自定义仓库地址。使用pod install安装，pod update进行更新。

## 二、Carthage管理

可以通过brew和port管理工具下载，也可以直接下载编译好的安装文件.pkg和源码进行安装，安装文档[地址](https://github.com/Carthage/Carthage#installing-carthage)。在项目目录创建文件Carthage，文件内容：

github "Alamofire/Alamofire" ~> 4.7.2

安装命令：carthage update，会将仓库源码checkout下来并编译成framework。

## 三、自定义框架

如果自定义模块具有通用性，可以作为第三方框架模式引入到项目中，可以直接指定仓库地址或自建cocoapods的spec仓库进行发布，具体可以参考官方[文档](https://guides.cocoapods.org/making/private-cocoapods.html)。

## 四、常见开源框架

Objective0-C：

* AFNetworking：网络框架
* SVProgressHUD：进度条hud
* MBProgressHUD：进度条hud
* YYKit：工具集合
* IQKeyboardManager：键盘管理工具
* Masonry：AutoLayout布局框架
* FMDB：sqilt数据库框架
* ReactiveObjC：Reactive的OC版本
* MJRefresh：下拉刷新
* FLEX：Flipboard出品的调试工具，hook到网络，界面，磁盘数据，界面查看
* Reveal-SDK：界面调试工具，付费软件
* LookinServer：微信出品的界面调试工具，免费
* SDWebImage：图片工具，包括缓存
* CocoaLumberjack：日志框架

Swift：

* Snapkit：AutoLayout布局框架
* Alamofire：网络请求框架

# 第七章 辅助工具

辅助工具包括数据调试、UI调试、内存检测等，这些辅助工具只在调试模式下使用，不能继承到上线版本。

## 一、FLEX

Flipboard发布的专门调试工具，官网[地址](https://github.com/Flipboard/FLEX)，可以提供文件内容查看、网络请求信息、界面布局查看等。使用方式为：

[[FLEXManager sharedManager] showExplorer];

提供功能：

* 查看界面布局，可以移动和修改界面控件。
* 查看沙盒内容，包括各种缓存数据和sqlite数据库，相当于文件浏览器。
* 查看系统日志文件。
* 查看网络请求信息。
* 查看当前内存情况。

## 二、Reveal

专业的界面调试工具，软件安装在Mac上面，通过WiFi或数据线直接远程查看APP界面状况，获取整个APP界面布局框架，分层显示具体的控件位置和属性，可以实时修改信息刷新到APP上面。官网[地址](https://revealapp.com/)，官网提供免费试用14天，只需要邮箱账号即可。  
主要提供功能：

* 界面布局
* UI控件属性
* 修改数据刷新到APP

## 三、Lookin

微信提供的类似Reveal工具，功能差不多类似，不过免费，具体查看[官网](https://lookin.work/)，xcode也提供了UI Hierarchy，不过功能太弱，调试起来很卡。

## 四、Instruments

Instruments为xcode自带的内存检测和性能优化工具，可以查找出系统存在的性能瓶颈，相对于第三方工具，Instrument是启动APP实时获取运行数据生成对应的曲线报告。目前提供功能常用的有：

* 内存分配
* 僵尸数据
* 内存溢出
* 网络数据量
* 响应时间
* 动画测试
* I/O性能