iOS 开发手册(V1.0.0)

行代码者，如切如磋，如琢如磨。

前言

本手册目前为 beta 版，还处于完善中，主要包含可以快速执行的内容。各小伙伴可以提出自己想法和建议，有必要可以开个小会进行讨论，达成共识和公约，确认最终可以执行的版本。确认后本手册还会不断完善和修正，但已存在的部分不会有大的修改，主要在于增加新的内容。

本手册默认为必须执行，只是建议而非强制执行的用红色“(不强制)”文字标出。

本手册将作为项目的一部分，放在项目的根目录，方便小伙伴执行和新伙伴学习。

在新版本迭代开发过程，新加代码应严格按照本手册执行，在对之前文件进行修改时，也要积极主动的根据本手册对文件进行局部重构，提高代码质量和可读性。

一、编程规约

Swift

(一)命名风格

Swift 的命名尽量简洁清晰

所有命名尽量使用英文，万不得已可以使用拼音。使用拼音时，类、协议可以用拼音首字母做前缀，如:TM;特卖，资源图片，变量等使用拼音时应用全拼，并写上对应的注释。

1. 类、协议命名

类的命名以首字母大写开始，多个单词以单词首字母大写进行分割。类的命名首先应保证表达意思明确，能一眼看出这个类是做什么任务的。

协议的命名和类基本相同，协议命名末尾一般加上单词 Protocol/Delegate/DataSource，普通情况下使用 Protocol，表示操作、事件相关的代理协议使用 Delegate，表示资源、数据相关的代理协议使用 DataSource，大部分情况下协议名可以直接在其相应的类名的尾部加上Delegate 即可，示例:

```
// 正例
EmptyView，OrderEventProtocol，MenuItemDelegate，MenuItemDataSource
```

继承自 UITableViewCell 的类，以 Cell 为后缀；继承自 UICollectionViewCell 的类，以 CCell 为后缀，示例:

```
// 正例
Class HomeAdsBannerCCell: UICollectionViewCell
Class HomeAdsBannerCell: UITableViewCell
```

2. 变量命名

变量名应使用容易意会的应用全称，以首字母小写开始，多个单词以单词首字母大写进行分割

```
class Person {
    var userName: String = ""
    var age: UInt = 0
}
```

变量名称应直接体现出变量的类型，当变量名能比较清晰的识别类型则可以省略，变量名不能清晰识别类型时则要加后缀，比如 UILabel 以 Label 为后缀，UIView 以 View 为后缀，UIImageView 以 ImageView 为后缀...，禁止让变量命名产生歧义或误导，如将一个 label 命名为 view，将一个NSArray 命名为 dic 等，示例:

```
// 正例
var orders: [OrderInfoModel] = []
var title: String = "标题"
var titleLabel: UILabel = UILabel()
var orderParamDic: [String: AnyObject] = [:]
var red: UIColor = .red

// 反例
var orderArray: [OrderInfoModel] = []
var orderDic: [OrderInfoModel] = []
```

```
var titleString: String = "标题"
var title: UILabel = UILabel()
var titleView: UILabel = UILabel()
var orderParam: [String: AnyObject] = [:]
var redColor: UIColor = .red
```

禁止让变量命名作用不清，不好搜索定位和识别，示例:

```
// 正例
var orders: [OrderInfoModel] = []
var ordersTableView: UITableView

// 反例
var datas: [OrderInfoModel] = []
var tableView: UITableView
```

3. 枚举的命名

枚举类型的命名与类相似，以首字母大写开始，多个单词以单词首字母大写进行分割。

枚举值的命名以首字母小写开始，多个单词以单词首字母大写进行分割，禁止以枚举名作为前缀。示例:

```
// 正例
enum AppThemeStyle {
    case white
    case black
    case blue
    case red
    case lightOrange
}

// 反例
enum AppThemeStyle {
    case AppMainStyleWhite
    case AppMainStyleBlack
    case AppMainStyleBlue
    case AppMainStyleRed
    case AppMainStyleLightOrange
}
```

4. 方法函数命名

方法名以及参数名的命名都以首字母小写开始，多单词以首字母大写的形式分割单词，能够清晰的表达出这个方法和参数所要完成的功能和意义，示例:

```
// 正例
func removeObject(forKey: String, style: AppThemeStyle)

// 反例
func remove(forKey: String, style: AppThemeStyle)
```

5. Extention 命名

Extention 的命名和类一致，文件名为主类类名 + “+” + 扩展名，扩展名首字母大写，原则上 Extention 命名只需要一个单词，应该体现出该扩展主要功能。 示例：

```
// 正例
UIView+Radius.swift
extenison UIView {

}

// 反例
UIViewRadius.swift
UIView+Extention.swift
```

6. 常量命名

常量名与变量名命名规则一致，即以首字母小写开始，多个单词以单词首字母大写进行分割，示例如下

```
// 正例
let blueText: UIColor = .blue

// 反例
let TEXT_COLOR_BLUE: UIColor = .blue
```

需要时可以通过 struct 来对常量进行归类，示例：

```
struct SizeStyle {
    static let mainFontSize14: Float = 14.0
    static let designScreenWidth: Float = 375.0
}
```

注：常量定义放在当前文件的顶部，不要放在文件中间，以方便查看

7. 资源图片命名

资源图片业务或者页面进行分组，每组建一个 folder，其名称为对应业务功能或页面名称，所有文件夹以及图片的名称禁止使用中文，只能使用英文字符。

资源图片的命名规则为：业务/页面 _ 图片名称 _ 状态；

业务/页面：为图片所对应的业务或页面名称，首字母小写并以首字大写的形式分割多个单词；

图片名称：为图片真实名称，首字母小写并以首字大写的形式分割多个单词；

状态：主要包含四种：正常、按下、选中、不可点击，分别用 normal、pressed、selected、disable 来标记，在没有不同状态，即只有正常状态的情况下可以省略状态这一栏，示例:

```
// 商品详情加购按钮，正常状态
goodsDetail_addBagButton_normal
// 商品详情加购按钮，不可点击状态
goodsDetail_addBagButton_disable
// Tabbar 首页正常状态图标
tabbar_home_normal
// Tabbar 首页选中状态图标
tabbar_home_selected
```

空状态页面上显示的图片，可以统一定为“空态”这个业务，且这些图片没有多种状态，故可以省略状态一栏，示例:

```
// 购物车页面为空状态
empty_goodsBag
// 空订单列表
empty_orderList
```

8. 国际化多语言命名

国际化多语言命名规则为：GM_业务/页面 _内容含义，业务/页面、内容含义以首字母大写并以首字母大写的形式分割单词，示例:

```
// 个人中心标题（实际开发中是不需要写注释的，这里只是为了方便理解作用与命名规则）
"GM_UserCenter_Title" = "个人中心";
// 分类
"GM_Tabbar_Category" = "分类";
```

通用常用的内容属于"公共"业务，命名为：GM_Common _内容含义，示例：

```
// 确定
"GM_Common_Sure" = "确认";
// 取消
"GM_Common_Cancel" = "取消";
```

国际化多语言命名内容按迭代版本号分块新增，版本之间隔一行，并以迭代版本号加以注释，避免迭代开发中的代码冲突，示例：

```
/// V1.0.0
"GM_UserCenter_Title" = "个人中心";
"GM_Tabbar_Home" = "首页";

/// V1.1.0
"GM_Common_Sure" = "确认";
"GM_Common_Cancel" = "取消";
```

(二)代码格式

1. 空格和空行的使用

1.1 运算符的两侧使用空格

所有的双目运算符，包括:加(+)、减(-)、乘(*)、除(/)、取余(%)、赋值 (=、+=、-=等)、比较运算符(>、<、==、!=等)、逻辑运算(&&、||)、位运算符 (|、&、<<、>>)等的两侧都应该添加空格，示例:

```
// 双目运算符左右加空格
result *= (param1 + param2) % 3 - param3 / 5
yesOrNo = (param1 >= param2) && param3 < 100
```

?: 三目运算符在 ? 和 : 的两侧都应该添加空格，示例:

```
// ?: 三目运算符空格使用
yesOrNo = param1 > 40 ? param2 : param3;
```

单目运算符(!)无需添加空格，示例:

```
// 单目运算符无需添加空格
param1 = !param2
```

1.2. 方法命中空格使用

方法中参数类型和":"之间应该添加 一个空格，示例:

```
// 正例
func test(string: String)

// 反例
func test(string:String)
func test(string : String)
func test(string :String)
```

多个方法的实现之间应该空一行，示例:

```
// 正例
func test1() {
```

```
}

func test2() {

}

// 反例
func test1() {

}
func test2() {

}
```

方法定义时，左大括号可以和方法名在同一行，左大括与方法的"()"之间应该空一格，示例：

```
// 正例
func test() {

}

// 反例
func test(){

}
```

1.3. 属性定义时空格的使用

属性定义时和函数参数一致，属性类型与":"之间都应该添加空格，示例:

```
// 正例
var userName: String = ""

// 反例
var userName:String = ""
var userName :String = ""
var userName : String = ""
```

2. 行数和列数的限制

2.1. 一个函数的长度不应该超过 50 行

当函数长度超过 50 行后，应该将内部一些复杂的逻辑提炼出来，原则上完整功能或需要注释的代码块都可以构成一个函数，形成新的函数后，然后调用，微型重构工作也应该无处不在，而不是等项目完成后再来重构

2.2. 每行代码长度建议不超过 120

建议每一行代码的长度超过 120 字符时做折行处理，处理时请以结构清晰为原则。通过“Xcode => Preferences => TextEditing => 勾选Show Page Guide / 输入120 => OK” 来设置提醒

2.3. 每个类原则上不超过 600 行

一个类不应该将很多复杂的逻辑揉合到一起来实现， 我们约定当 .m 文件超过 600 行时，要考虑将这个文件进行拆分，可以使用类目(Category)的方法来分离功能代码。如果逻辑过于复杂，则应该考虑从设计上将一些内部可以独立的逻辑提炼出来，形成新的类，以减轻单一类的复杂度

3. 方法的声明、定义和调用

3.1. 方法的声明和定义

正常情况下，所有参数应在同一行中，当参数过长时，每个参数占用一行，以参数首字母对齐，示例:

```
func refreshData(emptyImage: UIImage?,
                 imageWidth: Float?,
                 imageHeight: Float?,
                 description: String?,
                 subDescription: String?,
                 buttonTitle: String?)
```

3.2. 方法的调用

方法调用和方法声明规则一致，示例:

```
emptyView.refreshData(emptyImage: emptyImage,
                      imageWidth: imageWidth,
                      imageHeight: imageHeight,
                      description: description,
                      subDescription: subDescription,
                      buttonTitle: buttonTitle)
```

4. 代码分段的使用

Swift 的源码文件通常会比较大，代码行较多，推荐使用 // MARK: - XXX 将代码不同的处理段分隔开，分段名称的首母大写，方便在编辑器中快速定位到需要查看的代码。

4.1. Lifecycle 分段

在 UIViewController 的子类实现文件中，建议添加一个// MARK: - Life cycle 分段，该分段包含，init、loadView、viewDidLoad、viewWillAppear、viewDidAppear、viewWillDisappear、viewDidDisappear、viewWillLayoutSubviews、dealloc 等方法，并建议将 dealloc 方法放到实现文件最前面。

建议将 Lifecycle 分段放到文件的最前端，因为对一个类的阅读，往往是从 init、 viewDidLoad 等方法开始的，示例:

```
// MARK: - Life Cycle
override func viewDidLoad() {
        super.viewDidLoad()
}
```

4.2. Protocol 分段

为每一个代理添加一个分段，并且分段的名称应该是代理名称，代理名称务必正确拼写，方便查看代理的定义，并且分段要通过 extension 来实现，必要时可以新建对应类的 extension 文件，其他分段可视情况而定是否需要新建 extension 文件，示例:

```
// MARK: - UITableViewDelegate/UITableViewDataSource
extension HomeViewController: UITableViewDelegate, UITableViewDataSource {
}
```

4.3. Property 分段

Property 分段是类定义或声明非 UI 属性和变量的代码区域，放在类的最前面，最好是把数据属性和操作(闭包)属性用一个空行区分开来，示例:

```
// 正例
class HomeViewController: BaseViewController {
    // MARK: - Property
    /// 商品数集
    private var goodsModels: [GoosInfoModel]?

    /// 刷新回调
    var refreshHandle: (()->Void)?
}
```

4.4. Override 分段

对从父类继承的自定义的方法，单独添加一个 Override 分段，示例:

```
// 父类
class GMBaseObject {
    func test() {
    }
}

// 子类
class GMSubBaseObject: GMBaseObject {
    // MARK: - Override
    func test() {
    }
}
```

4.5. Events 分段

为 button 的点击事件，geture 的响应方法，KVO 的回调，Notification 的回调方法添加一个 events 分段，事件方法名结尾要能提现方法的来源，比如 button 对应 Action，手势对应 Gesture，通知对应 Notifaction，示例：

```
// 正例
// MARK: - Events
// MARK: -- Action
/// 登录按钮事件
@objc private func loginAction(sender: UIbutton) {

}

// MARK: -- UIGestureRecognizer
/// 长按手势事件
@objc private func longPressGesture(longPressGesture: UILongPressGestureRecognizer) {

}

// MARK: -- NSNotification
/// 通知事件
@objc private func updateViewWithNotification(note: NSNotification) {
}
```

4.6. Public 分段

public 段的方法，为需要暴露给其他类，供其他类调用的方法，示例:

```
// 正例
class GMBaseObject {
    // MARK: - Public
    func test() {
    }
}
```

4.7. Private 分段

Private 分段的方法，大多为实现本类业务而添加的方法，不需要暴露接口给其他类，只提供给本类的方法调用，示例:

```
// 正例
class GMBaseObject {
    // MARK: - Private
    private func requestGoodsList() {
    }
}
```

4.8. Lazy 分段

Lazy 分段用来对 UI 属性的声明定义，放在类的后面，示例:

```
// 正例
class HomeViewController: BaseViewController {
    // MARK: - Life Cycle
    ...
    ...
    // MARK: - Lazy
    private var goodsTableView: UITableView = {
      return UITableView()
    }()
}
```

(三)注释

1. 注释的格式

类、属性、方法等的注释应该使用 xcode 的标准方式，方便在使用到该类或属性或方法时，在联想词列表中看到对应的注释。

使用 xcode 快捷键 command + option + / 注释，示例:

```
/// 商品详情页
class GoodsDetailViewController: BaseViewController {
    /// 商品名
    private var goodsName: String = ""
}
```

2. 类的注释

在每个类的头文件的顶部，添加注释，简单的说明这个类是完成什么功能的，如果在新的版本开发对类进行比较重要的修改，也应该简单的注释下在哪个版本做了哪些修改，便于以后维护代码时能够更多的定位问题，示例:

```
//
//  GMGoodsDetailViewController.swift
//  商品详情页
/**
  1. V1.1.0 增加折扣价，阶梯价
  2. V1.2.0 增加活动折扣
**/

//  Created by 罗坤 on 2021/10/16.
//  Copyright © 2021 com.gree. All rights reserved.

/// 商品详情页
class GoodsDetailViewController: BaseViewController {
}
```

3. 属性和方法的注释

对自定义的属性和方法进行注释，以便在调用这些属性和方法进行查看和理解，使用 Xcode 快捷键 command + option + / 注释，示例:

```
/// 商品详情页
class GoodsDetailViewController: BaseViewController {
    /// 商品名
    private var goodsName: String = ""

    /// 根据活动类型刷新商品价格
    /// - Parameter activityType: 活动类型
    func refreshGoodsPriceWithType(activityType: ActivityType) {

    }
}
```

4. 枚举注释

对枚举和枚举类型的每一个枚举值给出注释，示例:

```
/// 活动类型
enum ActivityType {
    /// 普通商品
    case normal
    /// 折扣商品
    case discount
    /// 拼团商品
    case groupBuy
}
```

5. 一些必要的注释

对于一些逻辑比较复杂的代码，应该添加必要的注释，方便自己和他人理解与维护，示例:

```
func refreshGoodsPriceWithType(activityType: ActivityType) {
  ...
    // 根据活动类型更新商品价格
    switch activityType {
    case .normal:
          break
    case .discount:
          break
    case .groupBuy:
          break
    }
  ...
}
```

6. 开发状态注释 (不强制)

开发过程中因对接人、任务依赖相关人进度不同步，导致自己任务执行阻断时，使用 TODO: 加以注释；开发过程中发现原代码存在或可能存在 bug 和优化空间，自己不方便修改时，使用 FIXME: + 姓名 + 日期 + 注释 加以注释；原则上在当前迭代提交测试之前，应清除项目中所有的 TODO 和 FIXME。示例:

```
func loginAction() {
  // TODO: 对接登录接口，完成登录操作
}

func requestLogin() {
  // FIXME: lkun 2021.4.16 在网络请求回调 block 中应使用 __weak 定义 self，避免 block 循环引用
}
```

(四)其他规约

1. 集合类注明内部元素的类型

在定义声明属性或变量时，尽量避免使用 Any 或 AnyObject 类型

2. 分割线高度/宽度

分割线高度/宽度不能写成 0.5，需要根据屏幕 retain 倍数来设置最小尺寸，示例:

```
/// UI 尺寸设计规范
struct SizeDesign {
    /// 分割线高度
    static let splitLineHeight: CGFloat = 1.0 * UIScreen.main.scale / 3.0
}
```

3. 不允许使用魔法数字

对于商品状态，网络请求错误类型，UIView 及其子类的 tag 等，由不同数值进行区分的情况，一律使用枚举，不允许直接使用数字进行编码

4. 日志打印

代码中的日志打印应使用自定义的日志打印方法 DLog，禁止直接使用 NSLog 或 print 系列，NSLog 打印是耗时操作和不安全的，不应在线上出现，示例:

```
/// 日志打印
func DLog<T>(_ message: T, file: String = #file, funcName: String = #function, lineNum:
Int = #line) {
    #if DEBUG
    let fileName = (file as NSString).lastPathComponent
    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "HH:mm:ss.SSS"
    let dateString = dateFormatter.string(from: Date())
    print("[时间]\(dateString)\n[文件]\(fileName)\n[函数]\(funcName)\n[行号]\
(lineNum)\n[日志]\(message)")
    #endif
}
```

5. self 的使用

为了简洁，避免使用 self，即访问属性或调用方法时不要使用 self，只有在闭包或者初始化方法中避免与参数产生歧义时使用

6. 访问权限设置

当不需要在文件外部访问声明时，请使用 private 和 fileprivate 修饰，是使用 private 还是 fileprivate 视情况而定，示例：

```
// 正例
fileprivate var userName: String = ""
private func reloadData()

// 反例
var userName: String = ""
func reloadData()
```

当知道不需要重写声明(不需要继承)时，使用 final 修饰，示例：

```
// 正例
final class HomeViewController: BaseViewController {
}

// 反例
class HomeViewController: BaseViewController {
}
```

当某个属性要被外部访问但不希望被外部修改时，即属性只读，使用 private(set) 修饰，示例:

```
private(set) var token: String = ""
```

7. 可选类型拆包 (不强制)

在使用可选类型时，建议先拆包再使用，示例：

```
var homeViewController: BaseViewController?
guard homeViewController = HomeViewController() else {
  return
}
homeViewController.backgroundColor = .white
homeViewController.title = "首页"
```

8. 避免闭包的循环引用

在使用闭包需要避免循环引用时，要使用 [weak self]，在不完全确定 self 的存在时，不要用 [unowned self]，示例：

```
NetWorkRequest(LoginAPI.password(phoneNum: phoneNum,
                                 password: password),
                                modelType: ExampleModel.self) { [weak self] (model,
responseModel) in
    guard let `self` = self else {
        return
    }
    self.emptyViewStatus = .clear
} failureCallback: { (responseModel) in

}
```

Objective-C

(一)命名风格

Objective-C 中所有命名，应以能够清晰表达其含义为第一标准，命名长一点没关系，最主要是清晰。

所有命名尽量使用英文，万不得已可以使用拼音。使用拼音时，类、协议可以用拼音首字母做前缀，如:TM;特卖，资源图片，变量等使用拼音时应用全拼，并写上对应的注释。

1. 类、协议命名

类的命名以前缀开始，如: GM，除前缀外的第一单词首字母也需要大写，多个单词以单词首字母大写进行分割。类的命名首先应保证表达意思明确，能一眼看出这个类是做什么任务的，然后才考虑名字的长度(不要怕长)。

协议的命名和类基本相同，协议命名末尾一般加上单词 Protocol/Delegate/DataSource，普通情况下使用 Protocol，表示操作、事件相关的代理协议使用 Delegate，表示资源、数据相关的代理协议使用 DataSource，大部分情况下协议名可以直接在其相应的类名的尾部加上Delegate 即可，示例:

```
// 正例
GMEmptyView, GMOrderEventProtocol, GMMenuItemDelegate, GMMenuItemDataSource
// 反例
EmptyView, OrderEventProtocol, MenuItemDelegate, MenuItemDataSource
```

继承自 UITableViewCell 的类，以 Cell 为后缀；继承自 UICollectionViewCell 的类，以 CCell 为后缀，示例:

```
// 正例
@interface GMHomeAdsBannerCCell : UICollectionViewCell
@interface GMHomeAdsBannerCell : UITableViewCell

// 反例
@interface GMHomeAdsBannerCollectionViewCell : UICollectionViewCell
@interface GMHomeAdsBannerTableViewCell : UITableViewCell
@interface GMHomeAdsBannerView : UICollectionViewCell
@interface GMHomeAdsBannerView : UITableViewCell
```

2. 变量属性命名

变量名应使用容易意会的应用全称，且首字母小写，且使用首字母大写的形式分割单词。成员变量使用“_”做为前缀，以便和临时变量与属性区分，示例:

```
// 正例
@interface GMPerson : NSObject {
   NSString *_userName;
   NSInteger _age;
}

// 反例
@interface GMPerson : NSObject {
   NSString *userName;
   NSInteger age;
}
```

变量名称应直接体现出变量的类型，禁止让变量命名产生歧义或误导，如将一个 label 命名为 view，将一个 NSArray 命名为 dic 等，示例:

```
// 正例
@property (nonatomic, strong) NSArray *orderDataArray;
@property (nonatomic, strong) UILabel *countLabel;

// 反例
@property (nonatomic, strong) NSArray *orderDataDic;
@property (nonatomic, strong) UILabel *countView;
```

禁止让变量命名作用不清，不好搜索定位和识别，示例:

```
// 正例
@property (nonatomic, strong) NSArray *orderDataArray;
@property (nonatomic, strong) UITableView *ordersTableView;

// 反例
@property (nonatomic, strong) NSArray *dataArray;
@property (nonatomic, strong) UITableView *tableView;
```

3. 枚举的命名

枚举类型的命名与类相似，以前缀开始，如: GM，除前缀外的第一单词首字母也需要大写，多个单词以单词首字母大写进行分割。

枚举值的命名是在枚举类型之后加上表示该值的一个或多个单词，每个单词首字母大写。示例:

```
// 正例
typedef NS_ENUM(NSInteger, GMRefreshType) {
    GMRefreshTypeNormal = 0, // 停止刷新
    GMRefreshTypeNoMore, // 没有更多
};

// 反例
typedef NS_ENUM(NSInteger, RefreshType) {
    RefreshTypeNormal = 0, // 停止刷新
    RefreshTypeNoMore, // 没有更多
};
```

枚举值的确定可以使用默认的数值，也可参照系统很多枚举的赋值，采用移位的方法来赋值，这样做的好处在于，可以使用按位或和按位与来进行逻辑判断和赋值，简化代码的编写。(不强制)

```
typedef NS_OPTIONS(NSInteger, GMGoodsAmountViewOption) {
    GMGoodsAmountViewOptionInputAmountDisable = 1 << 1, // 金额输入款不可编辑
    GMGoodsAmountViewOptionInputAmountEnable = 1 << 2, // 金额输入款可编辑
    GMGoodsAmountViewOptionAddEnable = 1 << 3, // 不可加
    GMGoodsAmountViewOptionAddDisable = 1 << 4, // 可加
    GMGoodsAmountViewOptionMinusDisable = 1 << 5, // 不可减
    GMGoodsAmountViewOptionMinusEnable = 1 << 6, // 可减
};
```

禁止在使用枚举匹配、比较时使用数字，应使用明确的枚举名，示例：

```
// 正例
if (refreshType == GMRefreshTypeNormal) {
}

switch (refreshType) {
  case GMRefreshTypeNormal:
    break;

  case GMRefreshTypeNoMore:
    break;

  default:
    break;
}

// 反例
if (refreshType == 0) {
}

switch (refreshType) {
        case 0:
            break;

        case 1:
            break;

        default:
            break;
}
```

4. 方法函数命名

方法名的首字母小写，且以首字母大写的形式分割单词，方法名和参数加起来应该尽可能的像一句话，能够清晰的表达出这个方法所要完成的功能，示例:

```
// 正例
- (void)removeObjectForKey:(NSString *)key withType:(GMObjectType)type;

// 反例
- (void)removeObject:(NSString *)key withType:(GMObjectType)type;
```

在 Category 里，定义的方法，方法名应该加上前缀: gm_，以表明这个方法是在 Category 里定义的。

```
// 正例
- (void)gm_relayoutIfNeed;

// 反例
- (void)relayoutIfNeed;
```

5. Category 命名

Category 的命名和类基本一致，通常不需要加前缀，首字母大写，原则上 Category 命名只需要一个单词，应该体现出该扩展主要功能。 示例：

```
// 正例
@interface UIView (Radius)

// 反例
@interface UIView (Category)
```

6. 宏和常量命名

宏的命名全部为大写，并以前缀 GM 开头，多个单词之间用下划线(_)分割。可以参考系统自带宏的命名，命名单词应该表达出该宏表示的是个什么值，示例:

```
// 正例
#define GM_COLOR_TEXT_BLUE  [UIColor blueColor]

// 反例
#define GMColorTextBlue  [UIColor blueColor]
#define COLOR_TEXT_BLUE  [UIColor blueColor]
```

定义宏时，如果宏的值是一个表达式，必须要用小括号将表达式括起来，否则会出现异常错误。示例:

```
#define GM_COUNT_YESTODAY 20
#define GM_COUNT_TODAY 20
// 正例
#define GM_COUNT_TOTAL (GM_COUNT_TODAY + GM_COUNT_TODAY)

// 反例
#define GM_COUNT_TOTAL GM_COUNT_TODAY + GM_COUNT_TODAY
```

常量的命名和变量基本一致，只是需要用小写字母 k 作为前缀，以 GM 为标识前缀，首字母大写来分割单词。

```
// 正例
static NSString *const kGMEvaluateSuccessClickBack = @"EVALUATE_SUCCESS_CLICK_BACK";

// 反例
static NSString *const GMEvaluateSuccessClickBack = @"EVALUATE_SUCCESS_CLICK_BACK";
static NSString *const kEvaluateSuccessClickBack = @"EVALUATE_SUCCESS_CLICK_BACK";
```

注：常量和宏的定义放在当前文件的顶部，不要放在文件中间，以方便查看。

7. 资源图片命名

资源图片业务或者页面进行分组，每组建一个 folder，其名称为对应业务功能或页面名称，所有文件夹以及图片的名称禁止使用中文，只能使用英文字符。

资源图片的命名规则为：业务/页面 _ 图片名称 _ 状态；

业务/页面：为图片所对应的业务或页面名称，首字母小写并以首字大写的形式分割多个单词；

图片名称：为图片真实名称，首字母小写并以首字大写的形式分割多个单词；

状态：主要包含四种：正常、按下、选中、不可点击，分别用 normal、pressed、selected、disable 来标记，在没有不同状态，即只有正常状态的情况下可以省略状态这一栏，示例:

```
// 商品详情加购按钮，正常状态
goodsDetail_addBagButton_normal
// 商品详情加购按钮，不可点击状态
goodsDetail_addBagButton_disable
// Tabbar 首页正常状态图标
tabbar_home_normal
// Tabbar 首页选中状态图标
tabbar_home_selected
```

空状态页面上显示的图片，可以统一定为“空态”这个业务，且这些图片没有多种状态，故可以省略状态一栏，示例:

```
// 购物车页面为空状态
empty_goodsBag
// 空订单列表
empty_orderList
```

8. 国际化多语言命名

国际化多语言命名规则为：GM_业务/页面 _内容含义，业务/页面、内容含义以首字母大写并以首字母大写的形式分割单词，示例:

```
// 个人中心标题（实际开发中是不需要写注释的，这里只是为了方便理解作用与命名规则）
"GM_UserCenter_Title" = "个人中心";
// 分类
"GM_Tabbar_Category" = "分类";
```

通用常用的内容属于"公共"业务，命名为：GM_Common _内容含义，示例：

```
// 确定
"GM_Common_Sure" = "确认";
// 取消
"GM_Common_Cancel" = "取消";
```

国际化多语言命名内容按迭代版本号分块新增，版本之间隔一行，并以迭代版本号加以注释，避免迭代开发中的代码冲突，示例：

```
/// V1.0.0
"GM_UserCenter_Title" = "个人中心";
"GM_Tabbar_Home" = "首页";

/// V1.1.0
"GM_Common_Sure" = "确认";
"GM_Common_Cancel" = "取消";
```

(二)代码格式

1. 空格和空行的使用

1.1 运算符的两侧使用空格

所有的双目运算符，包括:加(+)、减(-)、乘(*)、除(/)、取余(%)、赋值 (=、+=、-=等)、比较运算符(>、<、==、!=等)、逻辑运算(&&、||)、位运算符 (|、&、<<、>>)等的两侧都应该添加空格，示例:

```
// 双目运算符左右加空格
result *= (param1 + param2) % 3 - param3 / 5;
yesOrNo = (param1 >= param2) && param3 < 100;
self.view.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
```

?: 三目运算符在 ? 和 : 的两侧都应该添加空格，示例:

```
// ?: 三目运算符空格使用
yesOrNo = param1 > 40 ? param2 : param3;
```

单目运算符(!、++、--)无需添加空格，示例:

```
// 单目运算符无需添加空格
param1 = !param2;
param2 = param3++;
for (NSInterger i = 0; i < 10; i++) {
  NSLog(@"%ld", i);
}
```

1.2. 方法命中空格使用

方法名中“+”或“-”与方法返回类型之间应添加一个空格，示例:

```
// 正例
- (void)test;

// 反例
-(void)test;
```

方法中参数类型和“*”之间应该添加 一个空格，示例:

```
// 正例
- (void)test:(NSString *)string;

// 反例
- (void)test:(NSString*)string;
```

多个方法的实现之间应该空一行，示例:

```
// 正例
- (void)doSomethingWithString:(NSString *)string {
}

- (void)doSomethingWithObject:(NSObject *)object {
}

// 反例
- (void)doSomethingWithString:(NSString *)string {
}
- (void)doSomethingWithObject:(NSObject *)object {
}
```

方法定义时，左大括号可以和方法名在同一行，左大括与方法的最后一个参数之间应该空一格，示例:

```
// 正例
- (void)doSomethingWithString:(NSString *)string {
}

// 反例
- (void)doSomethingWithString:(NSString *)string{
}
```

方法连续调用时，在中括号后添加空格，示例:

```
// 正例
UIView *view = [[UIView alloc] initWithFrame:CGRectZero];

// 反例
UIView *view = [[UIView alloc]initWithFrame:CGRectZero];
```

1.3. 属性定义时空格的使用

属性定义时，property关键字与左括号，右括号与属性类型，括号中逗号之后，属性类型与*之间都应该添加空格，示例:

```
// 正例
@property (nonatomic, copy) NSString *userName;

// 反例
@property(nonatomic,copy)NSString *userName;
```

属性定义时，小括号中各个部分的顺序约定如下:原子访问类型、读写权限、内存管理方式，示例:

```
// 正例
@property (nonatomic, readonly, copy) NSString *userName;

// 反例
@property (nonatomic, copy, readonly) NSString *userName;
```

2. 行数和列数的限制

2.1. 一个函数的长度不应该超过 50 行

当函数长度超过 50 行后，应该将内部一些复杂的逻辑提炼出来，原则上完整功能或需要注释的代码块都可以构成一个函数，形成新的函数后，然后调用，微型重构工作也应该无处不在，而不是等项目完成后再来重构

2.2. 每行代码长度建议不超过 120

建议每一行代码的长度超过 120 字符时做折行处理，处理时请以结构清晰为原则。通过“Xcode => Preferences => TextEditing => 勾选Show Page Guide / 输入120 => OK” 来设置提醒

2.3. 每个类原则上不超过 600 行

一个类不应该将很多复杂的逻辑揉合到一起来实现， 我们约定当 .m 文件超过 600 行时，要考虑将这个文件进行拆分，可以使用类目(Category)的方法来分离功能代码。如果逻辑过于复杂，则应该考虑从设计上将一些内部可以独立的逻辑提炼出来，形成新的类，以减轻单一类的复杂度。

3. 方法的声明、定义和调用

3.1. 方法的声明和定义

正常情况下，所有参数应在同一行中，当参数过长时，每个参数占用一行，以冒号对齐，示例:

```
+ (void)showEmptyOnView:(UIView *)superView
              emptyImage:(UIImage *)image
            emptyExplain:(NSString *)explain
              subExplain:(NSString *)subExpalin
                imageUrl:(NSString *)imageUrl
               retryText:(NSString *)retryText
            updateHandle:(void(^)(void))updateHandle {
}
```

当方法参数个数超过 3 个时，建议以 model 的形式传参，示例：(不强制)

```
- (void)updateInfoWithUserName:(NSString *)userName
                      nickName:(NSString *)nickName
                         email:(NSString *)email
                     avatarURL:(NSString *)avatarURL;

// 可以转为

@interface GMUserInfoModel: NSObject

/// 用户命名
@property (nonatomic, copy) NSString *userName;
/// 用户昵称
@property (nonatomic, copy) NSString *nickName;
/// 用户邮箱
@property (nonatomic, copy) NSString *email;
/// 用户头像
@property (nonatomic, copy) NSString *avatarURL;

@end

- (void)updateWithUserInfoModel:(GMUserInfoModel *)userInfoModel;
```

3.2. 方法的调用

方法调用和方法声明规则一致，示例:

```
[self refreshWithEmptyImage:image
                    explain:explain
                 subExplain:subExpalin
                   imageUrl:imageUrl
                  retryText:retryText];
```

4. 代码分段的使用

Objective-C 的源码文件通常会比较大，代码行较多，推荐使用 #pragma mark - XXX 将代码不同的处理段分隔开，分段名称的首母大写，方便在编辑器中快速定位到需要查看的代码。

4.1. Lifecycle 分段

在 UIViewController 的子类实现文件中，建议添加一个#pragma mark - Life cycle 分段，该分段包含，init、loadView、viewDidLoad、viewWillAppear、viewDidAppear、viewWillDisappear、viewDidDisappear、viewWillLayoutSubviews、dealloc 等方法，并建议将 dealloc 方法放到实现文件最前面。

建议将 Lifecycle 分段放到 .m 文件的最前端，因为对一个类的阅读，往往是从 init、 viewDidLoad 等方法开始的。示例:

```
#pragma mark - Life cycle
- (instancetype)init {
    if(self = [super init]) {
    }
    returen self;
}

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

4.2. 代理分段

建议为每一个代理添加一个分段，并且分段的名称应该是代理名称，代理名称务必正确拼写，方便查看代理的定义，示例:

```
#pragma mark - UITableViewDelegate
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
}

#pragma mark - UITableViewDataSource
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
(NSInteger)section {
}
```

4.3. Property 定义分段

每个类都会有属性，需要为属性添加一个分段，该分段包括属性的 get 和 set 方法，属性方法的初始化应尽量放到 get 方法中完成，而不应该全部放到 viewDidLoad 或者 UIView 的 init 方法中，这样会增加 viewDidLoad 和 init 方法的复杂影响可读性。

另外建议将 property 分段放置在 .m 文件的最后，因为该分段主要只包含一些属性的初始方法不需要太多查看，需要查看时可以直接command + 单击跳转到 get 方法查看。示例:

```
#pragma mark - Getter/Setter
- (UITableView *)ordersTableView {
    if (!_ordersTableView) {
    }
    return _ordersTableView;
}
```

4.4. override 分段

对从父类继承的方法，单独添加一个 override 分段，示例:

```
// 父类
@interface GMBaseObject : NSObject

- (void)test;

@end

// 子类
@implementation GMSubBaseObject

#pragma mark - Override
- (void)test {
}

@end
```

4.5. Events 分段

为 button 的点击事件、geture 的响应方法、KVO 的回调、Notification 的回调方法添加一个 events 分段，在该分段的方法中，调用对应业务的方法，对应业务的方法放在对应业务的分段，示例：

```
#pragma mark - Events
#pragma mark -- Action
// 登录按钮事件
- (void)loginAction:(id)sender {
}

#pragma mark -- UIGestureRecognizer
// 长按手势事件
```

```
- (void)longPressGesture:(UILongPressGestureRecognizer *)longPressGesture {
}

#pragma mark -- NSNotification
- (void)updateViewWithNotification:(NSNotification *)note {
}
```

4.6. Public 分段

public 段的方法，为需要暴露给其他类，供其他类调用的方法，示例:

```
#pragma mark - Public
@interface GMBaseObject : NSObject

- (void)test;

@end

@implementation GMBaseObject

#pragma mark - Public
- (void)test {
}

@end
```

4.7. Private 分段

Private 分段的方法，是为实现本类业务而添加的方法，不需要暴露接口给其他类，只提供给本类的方法调用，示例:

```

@interface GMBaseObject : NSObject

- (void)test;

@end

@implementation GMBaseObject

- (void)test {
}

#pragma mark - Private
- (void)updateData {
}

@end
```

5. 指定构造方法

要告诉调用者必须使用这个方法来初始化（构造）类对象时，使用 NS_DESIGNATED_INITIALIZER 宏来实现，示例:

```
@interface GMAdsBannerView : UIView

// 禁用从父类集成的初始化方法
- (instancetype)initWithFrame:(CGRect)frame NS_UNAVAILABLE;
- (instancetype)initWithCoder:(NSCoder *)coder NS_UNAVAILABLE;
// 告诉调用者，用此方法进行初始化
- (instancetype)initWithDelegate:(id<GMAdsBannerViewDelegate>)delegate
NS_DESIGNATED_INITIALIZER;

@end
```

(三)注释

1. 注释的格式

类、属性、方法等的注释应该使用 Xcode 的标准方式，方便在使用到该类或属性或方法时，在联想词列表中看到对应的注释。

使用 Xcode 快捷键 command + option + / 注释，示例:

```
/// 商品详情页
@interface GMGoodsDetailViewController : GMBaseViewController

/// 商品名
@property (nonatomic, copy) NSString *goodsName;

@end
```

2. 类的说明注释

在每个类的头文件的顶部，添加注释，简单的说明这个类是完成什么功能的，如果在新的版本开发对类进行比较重要的修改，也应该简单的注释下在哪个版本做了哪些修改，便于以后维护代码时能够更多的定位问题，示例:

```

//
//  GMGoodsDetailViewController.swift
//  商品详情页
/**
  1. V1.1.0 增加折扣价，阶梯价
  2. V1.2.0 增加活动折扣
**/

//  Created by 罗坤 on 2021/10/16.
//  Copyright © 2021 com.gree. All rights reserved.
```

```
/// 商品详情页
@interface GMGoodsDetailViewController : GMBaseViewController

@end
```

3. 属性和方法的注释

对自定义的属性和方法进行注释，以便在调用这些属性和方法进行查看和理解，使用 xcode 快捷键 command + option + / 注释，示例:

```
/// 商品详情页
@interface GMGoodsDetailViewController : GMBaseViewController

/// 商品名
@property (nonatomic, copy) NSString *goodsName;

/// 刷新商品价格
/// @param activityType 活动类型
- (void)refreshGoodsPriceWithType:(GMActivityType)activityType;
/// 重新加载数据
- (void)reloadData;

@end
```

4. 枚举注释

对枚举和枚举类型的每一个枚举值给出注释，示例:

```
/// 活动类型
typedef NS_ENUM(NSInteger, GMActivityType) {
    GMActivityTypeNormal = 0, // 非活动，正常形态
    GMActivityTypeDiscount,   // 折扣活动
    GMActivityTypeFull3Reduction20 // 满 3 件减 20
};
```

5. 一些必要的注释

对于一些逻辑比较复杂的代码，应该添加必要的注释，方便自己和他人理解与维护，示例:

```
- (void)refreshGoodsPriceWithType:(GMActivityType)activityType {
  ...

    // 根据活动类型更新商品价格
    switch (activityType) {
        case GMActivityTypeNormal:
            self.goodsPriceLabel.text = self.goodsModel.price;
            break;
```

```
        case GMActivityTypeDiscount:
            self.goodsPriceLabel.text = self.goodsModel.discoutPrice;
            break;
        case GMActivityTypeFull3Reduction20:
            self.goodsPriceLabel.text = self.goodsModel.FullReductionPrice;
            break;
        default:
            break;
    }

  ...
}
```

6. 开发状态注释 (不强制)

开发过程中因对接人、任务依赖相关人进度不同步，导致自己任务执行阻断时，使用 TODO: 加以注释；开发过程中发现原代码存在或可能存在 bug 和优化空间，自己不方便修改时，使用 FIXME: + 姓名 + 日期 + 注释 加以注释；原则上在当前迭代提交测试之前，应清除项目中所有的 TODO 和 FIXME。示例:

```
- (void)loginAction:(id)sender {
  // TODO: 对接登录接口，完成登录操作
}

- (void)reloadData {
  // FIXME: lkun 2021.4.16 在网络请求回调 block 中应使用 __weak 定义 self，避免 block 循环引用
}
```

(四)其他规约

1. 集合类注明内部元素的类型

```
// 正例
@property (nonatomic, strong) NSMutableArray<GMGoodsModel *> *goodsArray;
// 反例
@property (nonatomic, strong) NSMutableArray *goodsArray;

// 正例
- (void)configMenuWithItems:(NSArray<NSString *> *)menuItems;
// 反例
- (void)configMenuWithItems:(NSArray *)menuItems;
```

2. Readonly 属性

提供给外部调用，不能修改的属性，需要添加修饰符 readonly，示例:

```
// imageURL 仅提供外部读权限
@interface GMGoodsModel: NSObject

@property (nonatomic, readonly, copy) NSString *imageURL;

@end
```

3. 分割线高度/宽度

分割线高度/宽度不能写成 0.5，需要根据屏幕 retain 倍数来设置最小尺寸，示例:

```
/// 分隔线高度
#define GM_SEPARATED_LINE_HEIGHT  (1.0 / 3.0 * [UIScreen mainScreen].scale)
```

4. 头文件中应该暴露最少的接口

只将真正需要供其他类调用的方法和属性暴露在头文件中

5. 不允许使用魔法数字

对于商品状态，网络请求错误类型，UIView 及其子类的 tag 等，由不同数值进行区分的情况，一律使用枚举，不允许直接使用数字进行编码。

6. 日志打印

代码中的日志打印应使用自定义的日志打印宏 DLog，禁止直接使用 NSLog，NSLog 打印是耗时操作，不应在线上出现，示例:

```
#ifdef DEBUG
#define DLog(fmt,...) NSLog(@"\n[类名]%s\n[行号]%d\n[函数名]%s\n[日志]
%@\n",__FILE__,__LINE__,__FUNCTION__,[NSString stringWithFormat:(fmt), ##__VA_ARGS__]);
#else
#define DLog(fmt,...)
#endif
```

7. 头文件的依赖

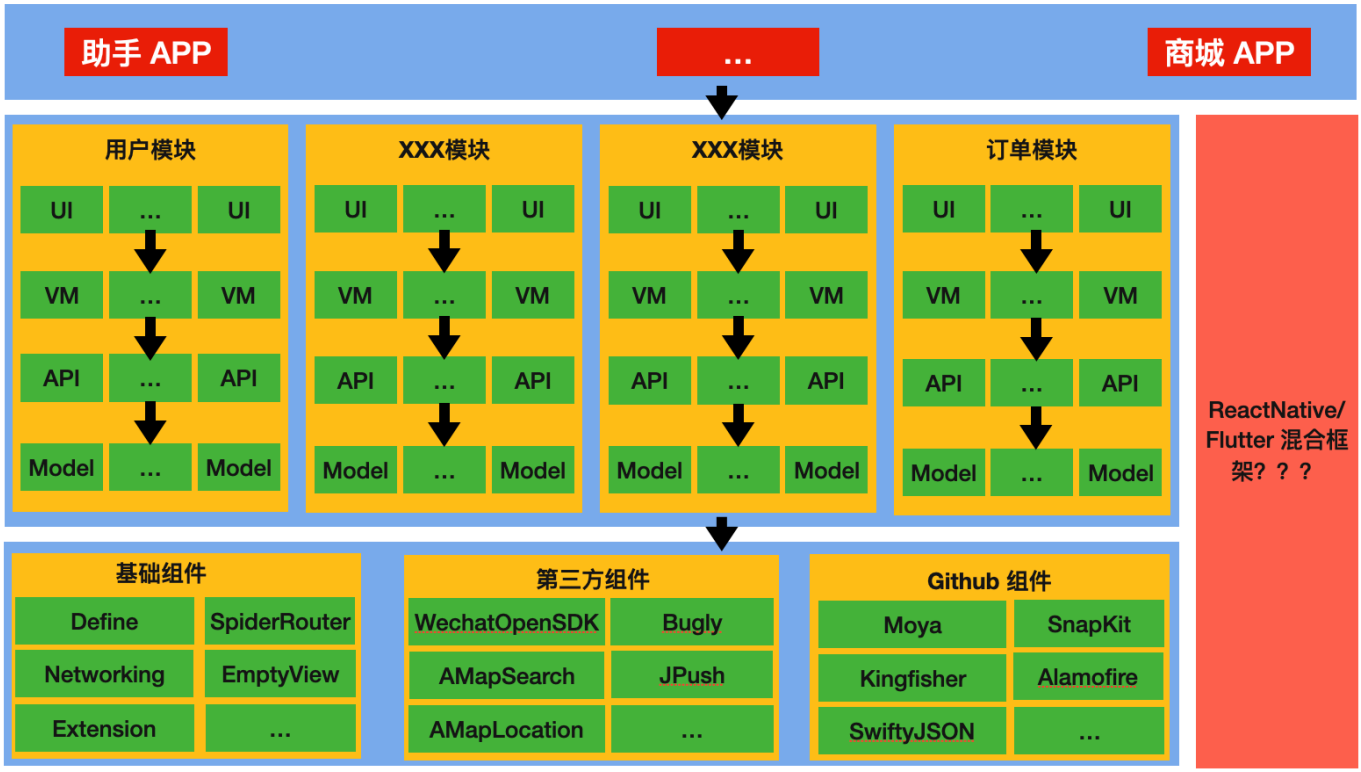
禁止使用 .pch 文件来引用头文件，只在需要的地方引用必要的头文件。

类之间如果没有必要间接依赖同个头文件，应在 .h 中通过 @class 来说明类的存在，在 .m 文件中引用其头文件(不强制)。

每一个组件都必须有一个以组件名命名的 .h 文件，该文件包含外部使用该组件时需要依赖的所有头文件。

二、工程结构

(一)项目架构

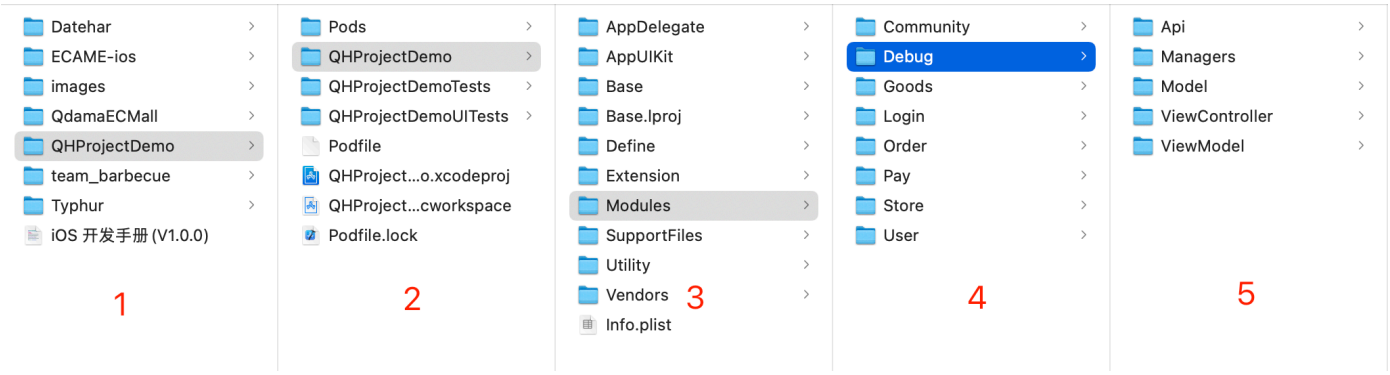


上图为项目的整体设计思想，采用 MVVM 的设计模式，箭头方向表示的是内部的依赖关系，即上层 APP 依赖各业务模块，各业务模块依赖各基础组件。虽然我们还没做组件化和模块化，但整体设计思想是不变的，在开发过程中要时刻考虑新增功能模块和组件能力的归类划分，即考虑怎样做能减少组件化或模块化的工作量，为后续往组件化或者模块化演进提供良好的基础。

备注：目前没有使用 ReactNative/Flutter 混合框架，具体会根据后期的业务需求做技术预研再来确定，是可以考虑的方向

(二)目录结构

1. 目录结构图



2. 目录结构说明

项目的目录结构划分为六个区域，对应上图的 1 / 2 / 3 / 4 / 5

2.1. 区域一

该区域为总项目文件目录，即对应从 gitlab clone 下来的项目

2.2. 区域二

该区域为代码项目的工程文件目录，存放工程文件、单元测试工程、组件依赖管理 Podfile、组件库 Pods，以及一些脚本文件，比如 bulgy 符号表上传脚本、自动化打包部署脚本、代码检测脚本等等

2.3. 区域三

该区域为代码项目的功能文件目录：

AppDelegate：app 生命周期管理，存放 UIApplication 代理、生命周期相关业务、TabbarController、引导页、广告页等文件

Base：存放自定义的一些基类，比如自定义 UIViewController、UIView、ViewModel、RequestApi 基类等

Define：存放自定义的全局常量和全局方法，比如设计规范、网络域名配置、第三方平台 key、路由配置、通知 key 等

Extension：存放 **全局** 基础功能类扩展，比如 UIView、UIImage、UIColor、UIViewController 等类的扩展，切记业务功能类的扩展不要放在这里

Modules：存放各个业务模块，例如用户模块、商品模块、订单模块、支付模块等等

SupportFiles：存放资源文件，包含图片、.plist 文件、Json 文件、国际化文件、桥接头文件等

Utility：存放基础组件，比如 Networking、SpiderRouter、EmptyView、AppCache、Managers 等。特别注意的是 Managers 文件夹，该文件夹存放的是 **全局** 基础功能的管理类，比如国际化文件读取管理、设备权限管理（比如相册权限、摄像头权限、日历权限、联系人权限等等）等，与具体业务无关的。原则上一个文件夹就是一个组件，做组件化时可以作为独立的组件的抽离出来

AppUIKit: 存放企业设计规范定义的基础组件，比如按钮、气泡弹窗等

Vendors: 存放的是不能通过 CocoaPods 管理的第三方平台库

2.4. 区域四

该区域为业务功能模块目录，原则上一个文件夹为一个业务功能模块，做模块化时可以作为独立的模块抽离出来使用

Debug: 调试模块，该模块负责 App 调试业务的实现，比如环境切换、组件的使用示例、业务功能的调试等，只在 Debug 环境下显示，Release 或线上隐藏

Login: 登录模块，该模块负责手机登录（验证码和密码）、第三方登录、修改密码、找回密码等业务

Goods: 商品模块，该模块负责首页、分类、商品搜索、商品详情等和商品有关的业务

Pay: 支付模块，该模块负责购物车、确认单页、支付页、订单完成页等和支付流程相关的业务

Order: 订单模块，该模块负责订单列表、订单详情、售后、评价等与订单相关的业务

Store: 门店模块，该模块负责首页门店切换、门店搜索等相关业务

User: 用户模块，该模块负责个人中心、会员、优惠券、积分、分享等相关的业务

2.5. 区域五

该区域为功能模块实现目录:

Managers：存放该功能模块辅助业务实现的辅助类

Api：存放该功能模块网络接口请求类

Model：存放该功能模块的数据模型类

ViewModel：存放该功能模块的业务逻辑实现类

View：存放该功能模块自定义 UI 视图类，视图类代码不涉及业务实现，只涉及 UI 的布局、展示、以及事件响应

ViewController：存放该功能模块控制器类，控制器类代码不涉及业务实现，只涉及 UI 的布局、展示、以及事件响应

备注：业务类扩展，要新建一个文件夹来存放，例如 HomeViewController.Swift、HomeViewController+Ads.Swift、HomeViewController+Banner.Swift 应新建一个文件夹存放，文件夹名为 HomeViewController

三、版本管理规范

(一)git 版本迭代管理

1.1 分支命名规则

开发分支命名规则：**dev _ 版本号 _ [业务功能(如有独立功能版本)]**，帮忙后期分支管理

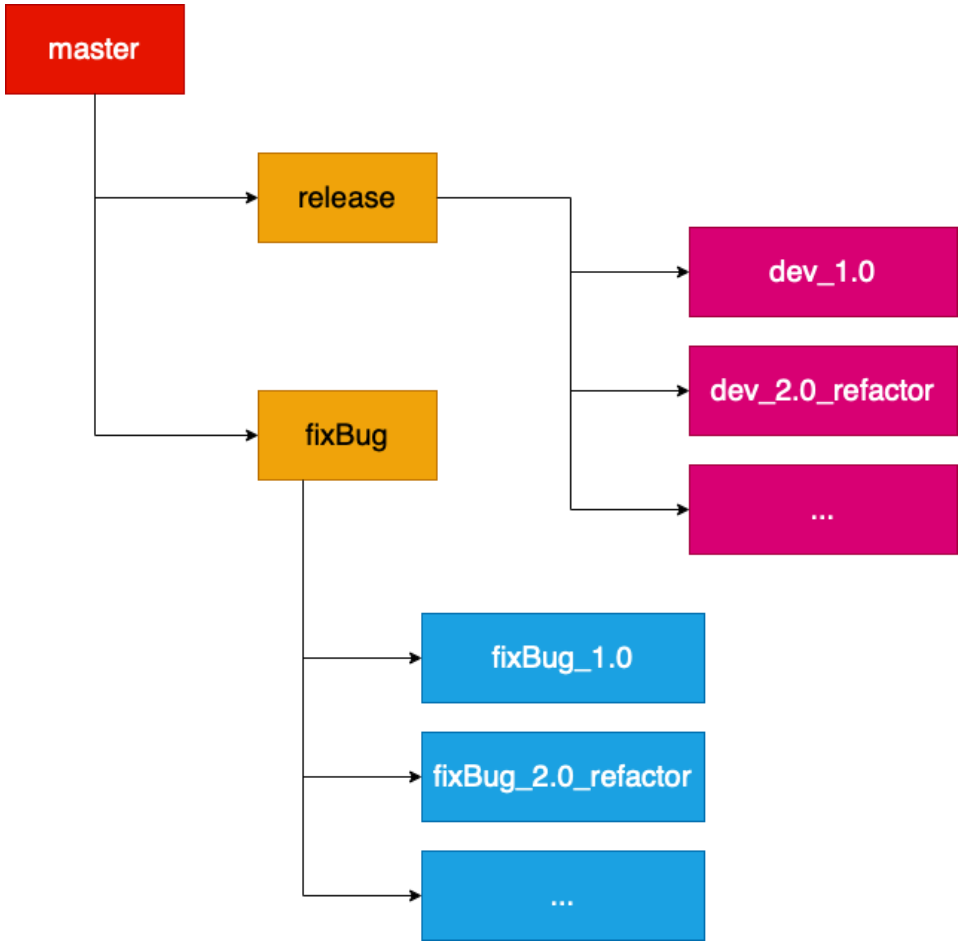
如：

业务迭代版本分支：dev_2.0

功能版本分支： dev_2.0_groupBuy

Bug 修复版本分支： fixBug_2.0，主要负责线上 bug 的修复

1.2 分支管理规则



上图为 git 版本管理分支之间的关系模型图

1. 每个版本的发布都要先将代码合并到 release 分支，然后在 release 分支打包发布
2. 当有线上 bug 并需要单独发版时，在 fixBug 相关联分支进行修复，然后在 fixBug 分支发布
3. 同级/跨级分支之间不能合并、拉取代码，只能合并代码到父分支，然后从父分支拉取到子分支
4. 同级的业务迭代版本分支只保留最近的 4、5 个分支，其他的分支删除

(二)git 提交格式

1. 功能业务代码提交

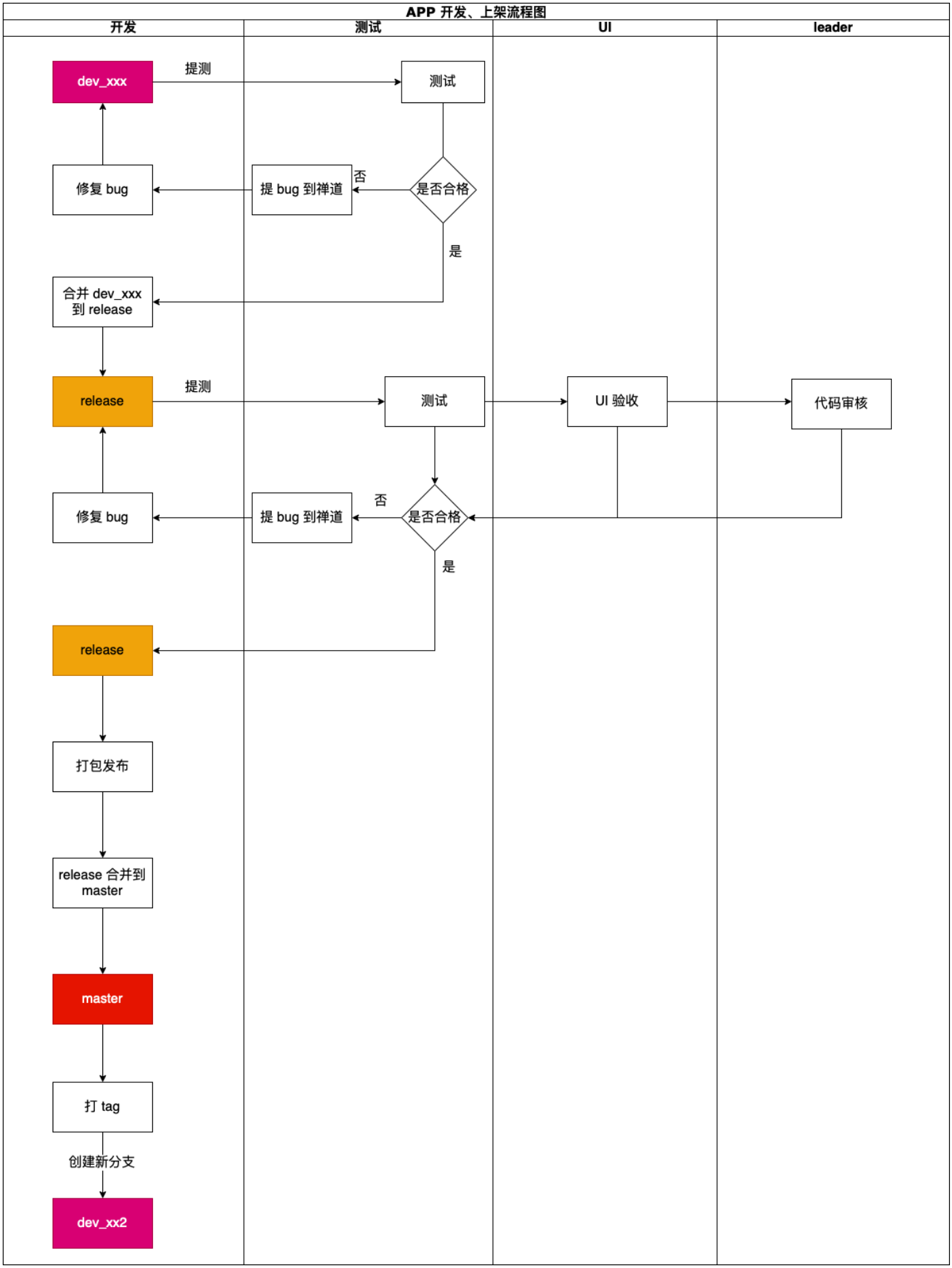
功能业务代码提交时，message 的说明格式为：

> **业务迭代：** feat: + [禅道任务编号] + 变更内容说明，例如 `git commit -m "feat:[5678]订单改造 2.0，增加退款二次弹窗提醒"`
>
> **非业务迭代：** feat + [组件名或文件夹名] + 变更内容说明，例如 `git commit -m "feat:[SpiderRouter] 增加通过 URL 路由能力"`

2. Bug 修复提交

> fix: + [禅道 bug 编号] + 修复内容说明，例如 `git commit -m "fix:[2345]修复订单改造 2.0，退款二次弹窗样式与设计图不一致问题"`

四、开发、上架流程


APP 开发、上架流程图
开发
测试
UI
leader
dev_xxx
提测
测试
修复 bug
提 bug 到禅道
否
是否合格
是
合并 dev_xxx 到 release
release
提测
测试
UI 验收
代码审核
修复 bug
提 bug 到禅道
否
是否合格
是
release
打包发布
release 合并到 master
master
打 tag
创建新分支
dev_xx2

五、公共组件使用说明

自己写公共组件要及时补充使用说明，并导出 PDF 格式版，覆盖原有版本

(一)网络组件

...

六、Code Review

(一)重复代码检测

重复代码检测工具使用的是 **PMD**，打开终端，执行以下命名安装:

```
brew install PMD
```

每个版本提测前执行一次检测脚本，检测结果输出在工程项目根目录 code_review_output.xml 文件里，重复代码尽量在每一个版本发布前解决

(二)代码规范检测

代码检测工具使用的是 **SwiftLint**，打开终端，执行以下命名安装:

```
brew install swiftlint
```

(三)内存泄漏检测

项目使用 Cocoapods 引入 **MLeaksFinder**， 开发过程中发现内存泄漏提示，需及时查找和修复

(四)代码代码复盘和 Review 会议

1. 出现线上 bug 时，必须召开复盘会议
2. 尽量每月开一次 Code Review 会议，每个成员对自己的代码做讲解，对代码规范和设计方案进行讨论和优化