

目录

● 概述	1.1
● 编程语言	1.2
Obj-C	1.2.1
Runtime	1.2.1.1
Method_Swizzling详解	1.2.1.1.1
多线程	1.2.1.2
Swift	1.2.2
Swift3的UsedResult警告	1.2.2.1
JavaScript	1.2.3
基础	1.2.3.1
js-Native交互	1.2.3.2
weex	1.2.3.3
rn	1.2.3.4
其它	1.2.4
Unix指令等脚本	1.2.4.1
xcodebuild&xctool	1.2.4.2
● 开发&进阶	2.1
UI控件	2.1.1
自定义View的几种方式	2.1.1.1
功能	2.1.2
iphone-x适配(完善中)	2.1.2.1
模块化方案	2.1.3
多媒体开发	2.1.3.1
网络	2.1.3.2
app内部路由策略	2.1.3.3
国际化方案	2.1.3.4
帐号体系	2.1.3.5
封装静态库&bundle	2.1.3.6
APP功能服务化组件开发	2.1.3.7
APPLauncher启动组件开发	2.1.3.8

填坑笔记	2.1.4
UIKit相关	2.1.4.1
Object相关	2.1.4.2
其它	2.1.4.3
● 开源库	3.1
Charts	3.1.1
Charts源码解析（一）	3.1.1.1
● 逆向与安全	4.1
工具	4.1.1
hopper	4.1.1.1
● 架构设计	5.1
设计模式	5.1.1
UML简介	5.1.1.1
1. uml类图手册	5.1.1.1.1
2. uml时序	5.1.1.1.2
3. uml活动图	5.1.1.1.3
4. uml状态机图	5.1.1.1.4
AOP	5.1.1.2
架构	5.1.2
项目模块化开发	5.1.2.1
模块化开发概述	5.1.2.1.1
子工程模板搭建	5.1.2.1.2
模块化开发详情	5.1.2.1.3
项目部署	5.1.2.1.4
组件实现细节	5.1.2.1.5
CocoaPods管理子工程	5.1.2.1.6
CocoaPods私有库framework创建和使用	5.1.2.1.7
MVC	5.1.2.1.8
MVP模式	5.1.2.1.9
MVIP模式	5.1.2.1.10
MVVM	5.1.2.1.11
● QA	6.1
性能优化	6.1.1
启动优化	6.1.1.1

耗电优化	6.1.1.2
内存优化	6.1.1.3
网络优化	6.1.1.4
优化工具	6.1.1.5
<u>测试</u>	6.1.2
单测	6.1.2.1
自动化测试	6.1.2.2
机型测试（分布调研）	6.1.2.3
其它（AB测试等）	6.1.2.4
<u>其它</u>	6.1.3
CodeReview	6.1.3.1
[代码规范度]()	6.1.3.1.1
[安全等各方面]()	6.1.3.1.2
安全扫描	6.1.3.2
● 代码管理	7.1
IDE	7.1.1
Xcode	7.1.1.1
包管理	7.1.2
CocoaPods	7.1.2.1
macOS之Carthage使用	7.1.2.2
Swift Package Manager	7.1.2.3
版本控制	7.1.3
Mac_For_Git_Server简易配置	7.1.3.1
SVN	7.1.3.2
● 打包&发布	8.1
帐号相关	8.1.1
申请帐号	8.1.1.1
IAP	8.1.1.2
iAd	8.1.1.3
持续集成	8.1.2
Jenkins	8.1.2.1
fastlane	8.1.2.2
优化	8.1.3

● 灰度分发&发布	8.1.4
● DMG打包流程	8.1.4.1
● 发布前CheckList-通用版	8.1.4.2
● 规范	9.1
● 工程规范	9.1.1
● 脚手架规范 (crash、数据统计等)	9.1.1.1
● 机型屏幕适配规范	9.1.1.2
● 代码规范	9.1.2
● Obj-C规范	9.1.2.1
● Swift规范1：linkedin代码规范译文	9.1.2.2
● Swift规范2：raywenderlich代码规范译文	9.1.2.3
● UED规范	9.1.3
● CodeReview规范	9.1.4
● 产品管理	10.1
● 项目把控	10.1.1
● Teambition、Scrum	10.1.1.1
● checklist，项目复盘	10.1.1.2
● PM	10.1.2
● 术语解读	10.1.2.1
● UED	10.1.3
● 工具使用：PaintCode、Sketch	10.1.3.1
● 其它	11.1
● Gitbook初探	11.1.1
● OmniPlan工具等	11.1.2
● DeepLearning	11.1.2.1
● 第三方服务	11.1.2.2
● 优质站点风云榜	11.1.2.3

● 概述

- iOS 开发手册
 - 提要
 - 花名册
 - 贡献方式
 - 文章来源及格式
 - I. 投递来源
 - II. 投递格式
 - 手册大纲

iOS 开发手册

提要

- 倾力构建一套完善的iOS知识体系，旨在使开发技能的梳理和获取井然有序，系统化。
- [gitbook 地址](#)
- [阅读模式](#)

花名册

开发人员	站点	其它
pan zhow	IT互联网自习室(公众号)	github
GoodbyeCain	.	github

贡献方式

1. 搜索并添加微信（[chowpan](#)），入伙花名册，一起完善该手册。
2. `pull request` 足下高作至[手册Github](#)。
 - i. Master 分支锁定
 - ii. 提交至 Dev 分支
 - iii. review、合并操作
3. 想得到的其它任意途径均可。

文章来源及格式

I. 投递来源

1. 自己原创。
2. 技术网站、公众号、简书、各大博客等优质文章均可。

II. 投递格式

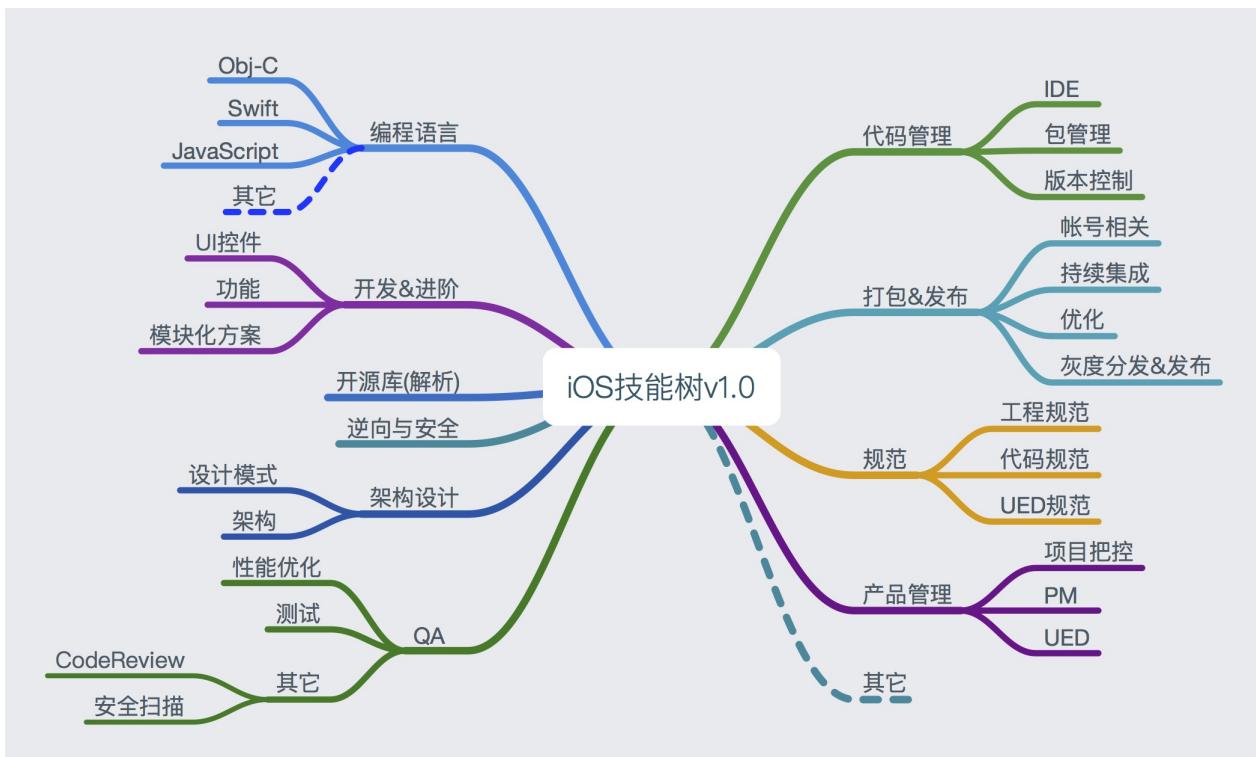
1. Markdown格式的文章（推荐）。

文章头部标有作者信息（姓名、微信、简书、公众号等）

2. 直接以文章外接、或者其它任意方式投递均可。

附：因文章入库格式为markdown，故而推荐。若含有图片等资源，最好以单一文件夹的方式管理。

手册大纲



Copyright © iOS 开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 编程语言

编程语言

- Obj-C
- Swift
- JavaScript
 - 基础
 - js-Native交互
 - weex
 - rn
- 其它

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- Obj-C

Obj-C

- Runtime
- 多线程

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- Runtime

Runtime

- Method_Swizzling详解

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

Method Swizzling

提要

1. 本文译自：<http://nshipster.com/method-swizzling/>
2. 和另一篇文章[associated objects](#)一起，让我们来共同认识下Object-c runtime的黑科技。

释义

1. 表现：可以更改现存方法（selector）具体的实现（Imp）
2. 功能：在runtime运行时，更改方法实现
3. 原理：在类的分发列表中，更改selectors与Imp的映射

举例

如我们想在一个**APP**中，记录每个**ViewController**被**push**的次数：

方案一：在每个**ViewCtrl**中，我们需要在**viewDidAppear**中添加**track**代码

缺点：产生大量的重复**track**代码

方案二：子类化**ViewController**

需要子类化**UIViewController**、**UITableViewController**、**UINavigationController**以及其他基类等。

此方案也会产生大量重复的**track**代码

方案三：使用类别（**category**）+ **method swizzling**

方案三：category + method swizzling

示例

```

#import <objc/runtime.h>
@implementation UIViewController (Tracking)

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Class class = [self class];
        SEL originalSelector = @selector(viewWillAppear:);
        SEL swizzledSelector = @selector(xxx_viewWillAppear:);

        Method originalMethod = class_getInstanceMethod(class, originalSelector);
        Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector);

        // When swizzling a class method, use the following:
        // Class class = object_getClass((id)self);
        // ...
        // Method originalMethod = class_getClassMethod(class, originalSelector);
        // Method swizzledMethod = class_getClassMethod(class, swizzledSelector);

        BOOL didAddMethod =
            class_addMethod(class,
                           originalSelector,
                           method_getImplementation(swizzledMethod),
                           method_getTypeEncoding(swizzledMethod));

        if (didAddMethod) {
            class_replaceMethod(class,
                               swizzledSelector,
                               method_getImplementation(originalMethod),
                               method_getTypeEncoding(originalMethod));
        } else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}

#pragma mark - Method Swizzling
- (void)xxx_viewWillAppear:(BOOL)animated {
    [self xxx_viewWillAppear:animated];
    NSLog(@"%@", self);
}

@end

```

注：在计算机技术中，引用转换pointer swizzling是基于名称和位置来指向指针引用的。尽管Object-C的原始细节不可考，但道理是相通的。因为method swizzling通过selector来更改函数指针的引用。（这段翻译的不好）

现在，任何基于UIViewController的实例或其子类实例，在调用viewDidAppear时，均会执行track代码。

詳解

一、使用场景：

在ViewController中：事务注入、响应事件、视图渲染、网络接口分发等。

二、使用地点

Swizzling 应该一直在 `+(void)load` 方法中执行。

在Obj-C类的 runtime中，有2个方法会被自动调用：

1. `+ (void)load`:类初始加载完成之后被调用。
2. `+ (void)initialize`:对于类或者类实例的所有方法，在被APP调用其中任一之前，调用该方法。
3. 因为method swizzling影响全局，所以尽量最少化的执行此种方案。
4. `+load`：可以保证类的初始化过程成功，并且在此之后执行。所以适合扩展系统行为。相对的，`+initialize`在执行时并不能保证类成功初始化。如果app从不直接发消息给该类，它将不会被调用。

三、`dispatch_once`

swizzling替换细节应一直在 `dispatch_once` 中实现。

再次重申，因为**swizzling**属于全局性的变更，所以我们要考虑到**runtime**的各种安全问题。原子性是其之一，也是保证代码只执行一次的方案。

即使在不同线程之间，GCD的`dispatch_once`也能保证代码只执行一次。也被认为是单例实现的标准化方案。

四、`Selectors`、`Methods`、`Implementations`

在Obj-C中，`Selectors`、`Methods`、`Implementations` 是运行时的主要组成部分。（在一般情况下，我们只是简单的将它们理解为同一概念：函数）

下面我们来看下Apple给出的概念解释：

1. `Selector`(`typedef struct objc_selector *SEL`):在**runtime**期间，代表方法的名称，作为一个C字符串注册（映射）在**runtime**中。

当相关类加载完成之后，编译器生成所属`Selectors`，并在**runtime**期间自动完成映射。

2. Method(**typedef struct objc_method *Method**):不透明类型，在类的定义中，表示一个方法。
3. Implementation(**typedef id (*IMP)(id, SEL, ...)**)：该指针类型指向method方法实体（具体实现）的初始位置。
 - i. id:指向自己的指针。实例方法：指向类的实例；类方法：则指向metaClass。
 - ii. 方法的Selector
 - iii. 后续参数：method

关系总结：

1. 一个类含有一个分发列表（**dispatch table**）：在运行时处理消息发送机制。
2. **dispatch table**中每个条目为一个Method，含有Key-Value：**Sel-IMP**。
3. IMP：指针，指向一个具体的方法实现

swizzling实质：改变一个类的**dispatch table**，即更改一个存在的**selector**，去映射新的**IMP**。也即一个方法实现**IMP**映射为一个新的**Selector**。

五、cmd调用

```
- (void)xxx_viewWillAppear:(BOOL)animated {
    [self xxx_viewWillAppear:animated];
    NSLog(@"%@", NSStringFromClass([self class]));
}
```

一个疑惑：正常情况下，上述代码将导致一个循环引用错误。

但在**swizzling**中，并没有这个错误。

因为**swizzling**在**runtime**中，`xxx_viewWillAppear:`已被重新指定到原始实现**Imp** (**UIViewController**的 `-viewWillAppear:`)。

故系统级别调用**viewWillAppear**，实际执行的是**xxx_viewWillAppear**的**IMP**；而此时 `[self xxx_viewWillAppear:]` 则是执行**viewWillAppear**的**IMP**了。

注：在**swizzling**方法中，记得添加前缀。避免与其它category中造成冲突。

注意事项

Swizzling被认为是黑科技，容易引发不可预知的后果。尽管如此，注意以下几项，**Method Swizzling**还是安全的。

1. 除非必要，应始终调用方法的原始**IMP**（即不使用**swizzling**）

APIs提供了输入输出的约定，并且IMPs属于黑盒范畴。Swizzling更改了原生IMPs，破坏了这种私有状态，进而影响到全局APP。

2. 避免冲突：针对category中的方法添加前缀
3. 知其然知其所以然：不理解swizzling的工作原理，只是简单的复制其实现代码，不仅危险，而且也浪费了深度学习runtime的机会。

通过阅读官方文档[Objective-C Runtime Reference](#)并阅读\文件，对swizzling的工作原理有较深的理解。学习替代想象。
4. 谨慎运用：无论多么熟悉swizzling技术，在下一版本任何问题都有可能发生。

资源库

1. [JRSwizzle](#):三方库，并针对上述问题，做了相关的安全措施。

结语

同[associated objects](#)一样，Method Swizzling功能强大，利弊皆有，需要谨慎的使用之。

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 多线程

多线程

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- [Swift](#)

Swift

- [Swift3的UsedResult警告](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 标题：Swift3 的Used Result 警告
 - Swift2.x时期
 - Swift3的变更
 - Swift3消除警告
 - Optional Chaining中属性丢失
 - 延伸阅读：

原文连接：<https://useyourloaf.com/blog/swift-3-warning-of-unused-result/>

译文如下：

标题：Swift3 的Used Result 警告

使用Xcode8的迁移工具将项目转换成Swift3的语法格式。转换之后，却产生了许多类似 `Result of call to someFunction` 形式的警告。

在Swift3中，调用一个具有返回值的函数时，若不使用其返回值，默认是给出警告的。这篇文章将针对如何忽略警告，做一些总结：

Swift2.x时期

在Swift2.x中，我们可以告知编译器，当忽略函数的返回值时，给予警告。

使用关键字 `@warn_unused_result`，如

```
@warn_unused_result func doSomething() -> Bool {  
    return true  
}
```

直接调用上述函数，而不处理返回值时，则会得到如下警告：

Result of call to 'doSomething()' is unused

同理在OC中使用属性标注，可以得到类似效果：

```
- (BOOL)doSomething __attribute__((warn_unused_result)) {  
    return YES;  
}
```

直接调用上述函数，而不处理返回值时，则会得到如下警告：

Ignoring return value of function declared with warn_unused_result attribute

Swift3的变更

在**Swift3**中不使用返回值，默认是给予警告的。而在**Obj-C**和**Swift2.x**中，不使用返回值，默认是不给予警告的。

这也是为什么项目从**Swift2.x**迁移至**Swift3**时，会产生很多新的警告。

Swift3消除警告

1. 声明函数时使用关键字 `@discardableResult` 重写之。

```
@discardableResult func doSomething() -> Bool {
    return true
}
```

意即不使用返回值时，不显示（丢弃）警告。

同理在**Obj-C**中，所有返回非空的函数默认均自动添加了 `@discardableResult` 属性，而不是 `warn_unused_result` 属性了。

2. 调用返回值非空函数时，使用 `_` 明确丢弃返回值。

```
_ = doSomething()
```

Optional Chaining 中属性丢失

在**Xcode8 beta3**中有种情景，即使用 `@discardableResult` 属性却没有得到预期效果。如推出 `ctrl`：

```
navigationController?.popViewController(animated: true)
```

警告如下(经测试，在正式版中也有该警告)：

```
Expression of type 'UIViewController?' is unused
```

原因：

1. `popViewController(animated: Bool)` 函数这是一个**Obj-C**的API，故默认添加了 `@discardableResult` 属性。
2. `navigationController`必须是可选类型（Optional Chaining），取值可以为 `nil`

3. @discardableResult 属性在遇到Optional Chaining之后会丢失

解决方案：

```
_ = navigationController?.popViewController(animated: true)
```

延伸阅读：

- SE-0047 Defaulting non-Void functions so they warn on unused results
- Swift Bug SR-1681 spurious unused result warning in Swift 3
- Swift Evolution Proposal Status

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- [JavaScript](#)

JavaScript

- [基础](#)
- [js-Native交互](#)
- [weex](#)
- [rn](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 基础

基础

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- [js-Native交互](#)

js-Native交互

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- [weex](#)

weex

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- [rn](#)

rn

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 其它

其它

- Unix指令等脚本
- xcodebuild&xctool

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

- [Unix指令等脚本](#)

Unix指令等脚本

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- [xcodebuild&xctool](#)

xcodebuild&xctool

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- 开发&进阶

开发&进阶

- UI控件
- 功能
- 模块化方案
- 填坑笔记

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- [UI控件](#)

UI控件

- [自定义View的几种方式](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

自定义**View**的几种方式

todo

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 功能

功 能

- iphone-x适配(完善中)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

一、参考链接

1. <https://developer.apple.com/cn/ios/update-apps-for-iphone-x/>
2. 链接：[WWDC__Designing for iPhoneX](#)
3. 链接：[Building Apps for iPhone X](#)
4. 链接：[Safe Area](#)

二、概述

1. 使原有项目支持iphone-x：启动画面若是Launch Image，需要添加针对iphone-x的启动画面，才能保证在模拟器或在真机上以iphone-x的屏幕大小运行。（Launch Storyboard设置正确，应该不会有问题是）

UIScreen的大小是在初始化时根据我们进入的第一个页面去进行参数化的。故更改启动图来适配不同的屏幕大小。

2. 所有UIs检测：元素错位、重叠、缩放错误、切割等各种问题

解决方案：Safe Area Guides、边距布局

3. 在实际适配中，发现使用约束情况下比非约束情况更好适配。

主要原因：相对布局，改动较少；内部subViews的布局很少以全局UIScreen的高度和宽度计算，基本上self.view的布局正确就Ok了。

三、方案详情

1. Safe Area：

1. 全页面

- iOS 11.0+:topLayoutGuide、bottomLayoutGuide
- iOS 11.0+:safeAreaInsets（非约束情况）、safeAreaLayoutGuide（使用约束时）
- 苹果官方的设计指导是使用以safeArea为框，以layoutMargin为间距来进行UI布局。
- 其它：扩展Safe Area，如additionalSafeAreaInsets定制Safe Area

2. 状态栏、导航栏、TabBar、底部屏幕指示器

- statusBar+naviBar = 64.0 常量修正。
- iOS 11以前，导航栏的高度是64，其中状态栏statusBar的高度是20，底部tabbar的高度是49。
- iOS 11之后，iPhoneX的导航栏的高度是88，其中状态栏的statusBar的高度变成了44，底部的tabbar变成了83（添加了虚拟Home区）。

3. 内容元素如按钮避开屏幕角落和传感器槽，防止被切割。

4. 屏幕边缘手势

5. 适应不同的屏幕宽高比（横向、纵向）

6. 其中针对SafeArea，目前做了工具IMXSafeAreaKit:实现了safearea的获取，本地保存等定制功能。

2. 相册访问权限变更

- 即默认读权限，增强了写权限。
- www.jianshu.com/p/cd0f814a7ce9

3. iOS11适配UITableView

- <http://www.jianshu.com/p/73394f7518c8>
- UITableView偏移问题

iOS 11.0+, automaticallyAdjustsScrollViewInsets废弃，新增contentInsetAdjustmentBehavior。

- 在iOS11.0+：cell.contentView会和cell自身大小有变化，做了切割。

四、code snippets

1. API添加判断支持iOS11+：

```
if (@available(iOS 11.0, *)) {
    // 版本适配
}
```

2. 屏幕大小判断（尽量不用此类定量方式）

```
#definekStatusBarHeight[[UIApplication sharedApplication] statusBarFrame].size.height
#define kNavBarHeight 44.0

#define kTabBarHeight ([[UIApplication sharedApplication] statusBarFrame].size.height>20?83:49)//tabBar高

#define kTopHeight(kStatusBarHeight + kNavBarHeight)//导航栏高

//判断iPhoneX

#define isiPhone (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone)

#define iPhoneX [[UIScreen mainScreen] bounds].size.width >= 375.0f && [[UIScreen mainScreen] bounds].size.height >= 812.0f && isiPhone
```

3. UITableView偏移问题：

```
if (@available(iOS 11.0, *)) {
    self.tableView.contentInsetAdjustmentBehavior = UIScrollViewContentInsetAdjustmentNever;
} else {
    self.automaticallyAdjustsScrollViewInsets = NO;
}
```

4. 适配时机（在ViewController中）

```
- (void)viewDidLoad{
    [super viewDidLoad];
    //约束、frame布局设置
    //.....
}
```

五、UI细节

1. 屏幕大小（注：iphonex与iphone6等宽，却需要@3x的图片）

iOS Device Display Summary

Table 2-1 summarizes the physical dimensions of iOS displays and how those pixels are mapped to the logical coordinate system in UIKit.

Table 2-1 Screen Geometry

Device	Native Resolution (Pixels)	UIKit Size (Points)	Native Scale factor	UIKit Scale factor
iPhone X	1125 x 2436	375 x 812	3.0	3.0
iPhone 8 Plus	1080 x 1920	414 x 736	2.608	3.0
iPhone 8	750 x 1334	375 x 667	2.0	2.0
iPhone 7 Plus	1080 x 1920	414 x 736	2.608	3.0
iPhone 6s Plus	1080 x 1920	375 x 667	2.608	3.0
iPhone 6 Plus	1080 x 1920	375 x 667	2.608	3.0
iPhone 7	750 x 1334	375 x 667	2.0	2.0
iPhone 6s	750 x 1334	375 x 667	2.0	2.0
iPhone 6	750 x 1334	375 x 667	2.0	2.0
iPhone SE	640 x 1136	320 x 568	2.0	2.0

2. 资源图片：

手机型号	屏幕尺寸	屏幕密度	开发尺寸	像素尺寸	倍图
4/4S	3.5英寸	326ppi	320*480 pt	640*960 px	@2X
5/5S/5C	4英寸	326ppi	320*568 pt	640*1136 px	@2X
6/6S/7/8	4.7英寸	326ppi	375*667 pt	750*1334 px	@2X
6+/6S+/7+/8+	5.5英寸	401ppi	414*736 pt	1242*2208 px	@3X
X	5.8英寸	458ppi	375*812 pt	1125*2436 px	@3X

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- 模块化方案

模块化方案

- 多媒体开发
- 网络
- app内部路由策略
- 国际化方案
- 帐号体系
- 封装静态库&bundle
- APP功能服务化组件开发
- APPLauncher启动组件开发

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 多媒体开发

多媒体开发

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- 网络

网 络

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- app内部路由策略

app 内部路由策略

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- 国际化方案

国际化方案

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

账户体系

一、简介

1. 在TiketNew项目中主要负责关于账户体系的开发。
2. 因为账户体系比较繁杂，故在开发前做了一定的调研，产出了一个较为初级的账户体系架构
3. 随着后期的开发和不断深入，在一定程度上完善了该体系，并总结为文档，记录于此。
4. 以下介绍的有部分没有在TN一期中使用到

二、架构

1. 概览

1. 在app中，帐号的作用应该有2种：

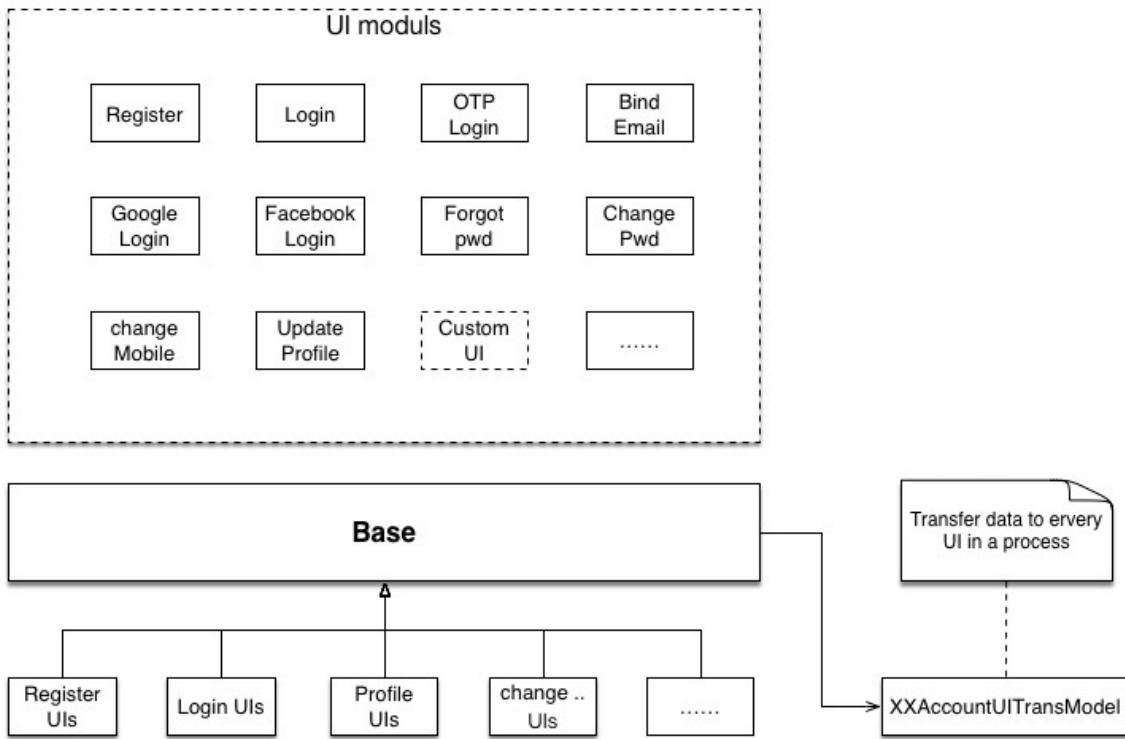
- 1) 购买订单时需要验证用户合法性。
- 2) 提供用户留存手段（签到、积分等）（TN项目无此功能）

2. 其它模块调用账户体系时表现形式：

- 1)如订单提交时未登录，弹出登录界面
- 2)用户手动调出登录界面

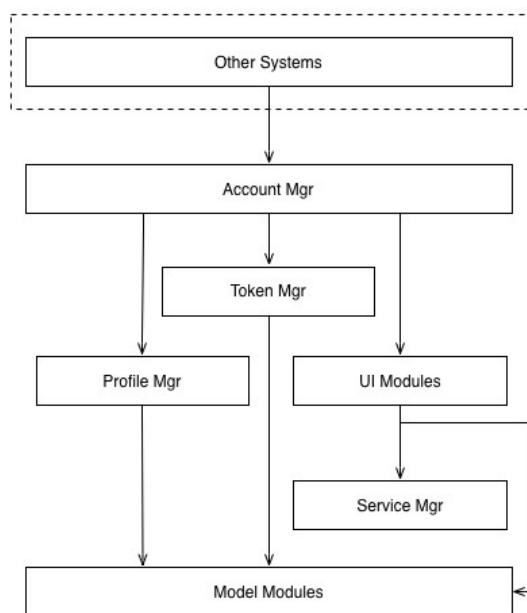
3. 登录体系目前细分如下：

- i. Model Modules : TokenModel和Profile Model集合，负责保存用户登录信息和用户属性
- ii. Token Mgr : 管理用户登录的Token Model : 更新、删除以及同步 (mem和Disk中均有一份完整的model)
- iii. Profile Mgr:管理用户属性集Profile Model : 更新、删除以及同步
- iv. Service Mgr:网络请求优化。具体的数据请求和账户体系中所需数据隔离
- v. UI Modules:登录、注册等具体界面集合
- vi. Account Mgr:帐号管理器，对外模块接口

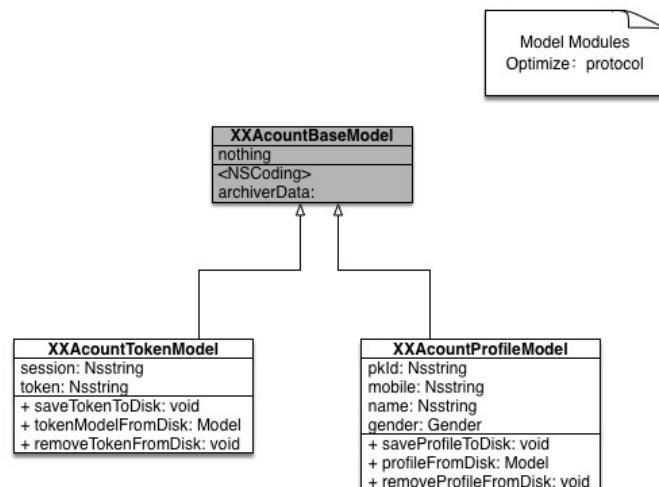


1. 下面给出各个模块间的关系图

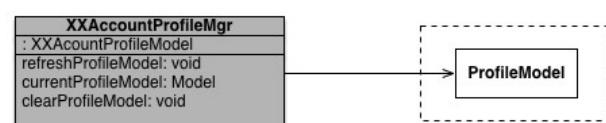
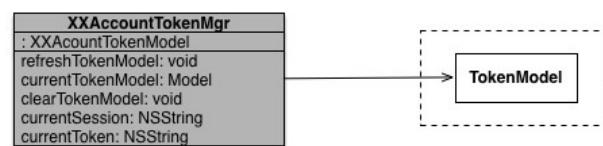
i. 模块间的关系图



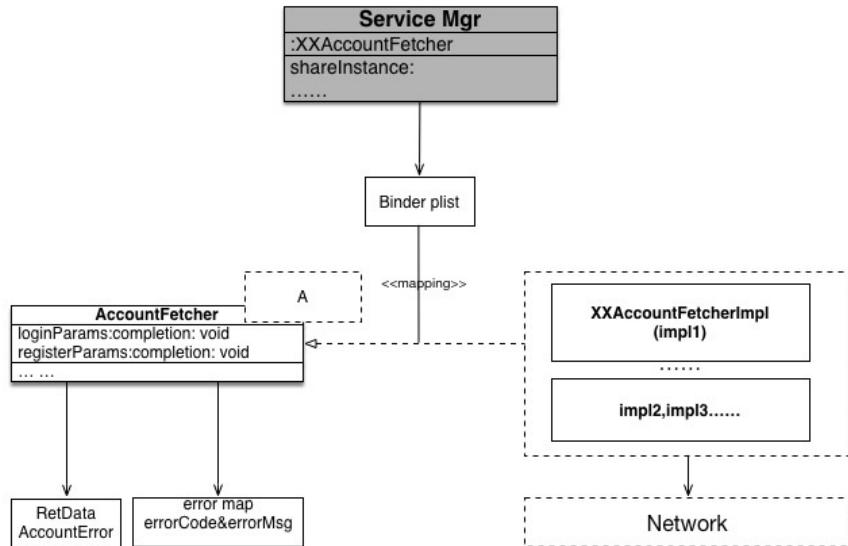
ii. Model Modules类图：其中待优化点为使用协议规范Model的各api



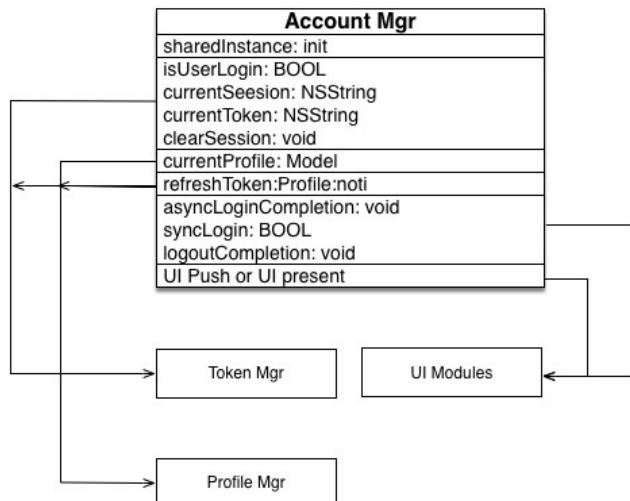
iii. Token Mgr和Profile Mgr 类图



iv. Service Mgr类图



v. Account Mgr类图



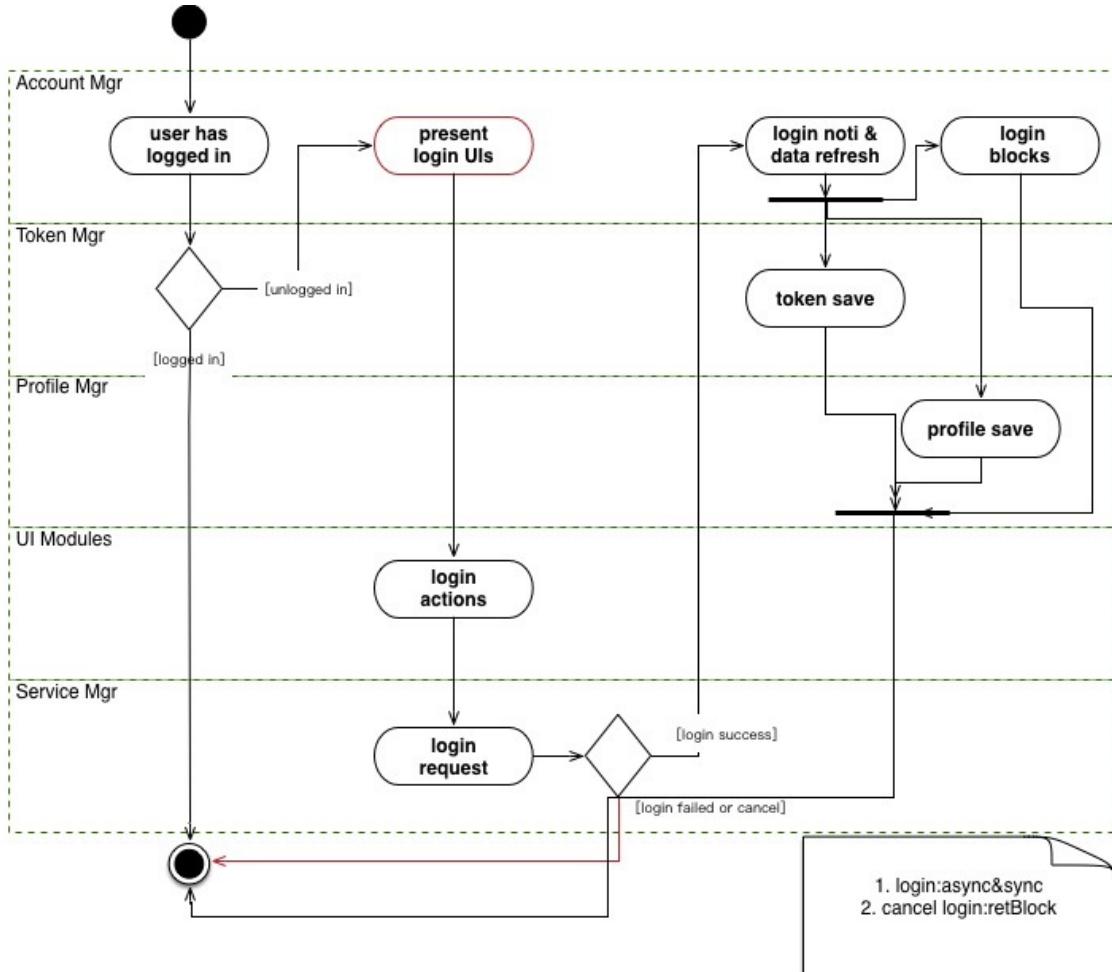
2. 账户体系协作流程

- 在 1.概览 中介绍了各子模块间的结构和相互关联，本小节将给出一个完整的登录事件。

2. 其它模块调用登录，有2种方式：

- i. API需要登录时，使用异步调用登录方式：同步方式，即异步线程下只允许一次登录
- ii. 人工登录，并获取回调：登录成功时，回调block集合

3. 下图中，当用户在登录界面点击cancel取消登录时，会有响应block激活，图中未标明

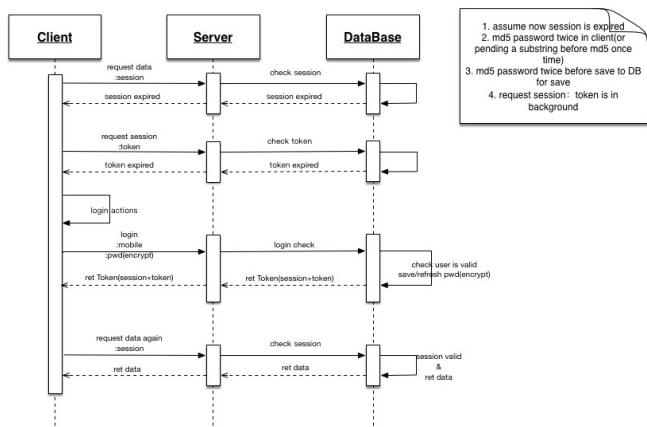


3. 一次完整的网络请求时序图

1. 附：

- i. 假定当前 `request data` 需要登录
- ii. 当前 `session` 过期
- iii. 密码登录时密码字段需要2次md5编码（也可以pwd+padding子串，执行一次 `md5`），登录成功后，保存DB前仍需要2次MD5编码（安全性考虑）
- iv. `session`过期后，client会用Token静默（后台）请求一次，如果token未过期，则此时可以在用户无感知情况下正常登录的。

2. 上图



三、其它

1. 关于登录注册中的Token和Session

1. request detail

- i. request param : user 、 pwd (MD5 or Encrypt)
- ii. response : userProfile 、 Token 、 Session
- iii. HTTPS protocol

2. Session

- i. 需要登录的请求，添加至header中或者参数中
- ii. expire过期时间较短（如24h）
- iii. 在24h(expire)之内，若被劫持，则仍有安全问题；但仍提高了安全性。

3. Token

- i. 长期存在本地DB。
- ii. 当Session过期，则以Token入参，请求新的session（针对用户透明，后台静默请求即可）。
- iii. expire时间较长（2~3Month，过期时，需用户重登录）

四、番外

骨感的现实：目前由于与TN现有架构的冲突，做法妥协处：

1. AccountMgr :主要负责实现管理 TokenMgr 和 ProfileMgr ；登录注册等界面弹出消失，.....。

目前实现同等功能的是 TNAccountService 协议，实现为 TNAccountServiceImpl

2. 架构设计中 ServiceMgr 协议负责统一接口请求格式，具体实现可以由不同的IMPL文件来处理。

目前处理方案：`TNAccountFacade` 实现接口请求，未能做到接口的统一处理。

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 封装静态库&bundle

封装静态库**&bundle**

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

一、功能概述

- APP项目中有一些功能是全局性的，独立于业务逻辑、特定界面而存在。更确切来讲，它们是为业务逻辑、特定界面提供某项具体的服务。

特点：在多个业务逻辑、多个界面中需要随时唤起服务、使用服务、结束服务

服务示例：如定位功能、拍照相册资源获取、分享、二维码扫描等等。

二、原理

一、文件职能

1. `IMXServiceManager` : 服务管理器。负责服务的注册、服务对象获取、唤起、结束等功能。

服务对象获取时，会考虑是否懒加载，是否为单例模式等，并以此做出一定的操作

1. `IMXService` : 服务协议。规范一些服务的基本属性和基本操作

- 基本属性：如服务名称、是否懒加载、是否单例模式
- 基本操作：唤起、执行、结束服务等

2. 特定服务协议：继承于基本协议 `IMXService` , 强化特定服务功能。

- 不同的服务，其服务方式会有差别。如定位服务除了基本操作外，还可能具有逆向地理解析的功能。故可以继承该协议，创建具体的定位协议文件，规范定位服务。

3. `IMXServiceInfo` : 服务的基础属性定制文件

二、技术点

1. `service`的注册：采用主动注册方案

- 在特定服务文件中 `+ (void)load` : 实现本文件的注册，注册至管理器中，才能被执行。

2. 服务管理器初始化：将注册的`services`激活

根据不同服务属性，选择是否懒加载、是否单例等。

三、主要文件截图

1. `IMXServiceManager`头文件：

```

//  

#import <IMXLauncher/IMXBaseLauncher.h>  

/**  

 * 服务管理器  

 */  

@interface IMXServiceManager : IMXBaseLauncher  

/**  

 * 单例模式  

@return 实例  

*/  

+ (IMXServiceManager *)sharedInstance;  

/**  

 * 注册service至本管理器中  

@param serviceName 服务类名  

*/  

- (void)attachToServiceMgr:(NSString *)serviceName;  

/**  

 * 根据service名称，获取服务实例  

@param serviceName 服务类名  

@return 服务实例  

*/  

- (id)serviceByName:(NSString *)serviceName;  

@end

```

2. IMXService协议文件：

```

#import <Foundation/Foundation.h>  

/**  

 * 服务协议：定义服务属性、服务的基本操作方法  

 */  

@protocol IMXService <NSObject>  

@optional  

- (NSString *)serviceNickName;//default is classname  

- (BOOL)lazyLoading;  

- (BOOL)singleton;  

@required  

- (void)startService;  

- (void)executeService;  

- (void)stopService;  

@end

```

3. IMXServiceInfo头文件：

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

/**
 * 服务具备的属性
 */
@interface IMXServiceInfo : NSObject

@property(nonatomic, copy) NSString *name;
@property(nonatomic, copy) NSString *className;
@property(nonatomic, strong) id instance;
@property(nonatomic, assign,getter=isLazyLoading) BOOL lazyLoading;//YES default
@property(nonatomic, assign,getter=isSingleton) BOOL singleton;//NO default
@end
|
```

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook修订时间：2018-03-08 10:26:48

一、功能概述

- APP启动组件开发，也即APP初始化整合方案。

功能：将初始化文件 `APPDelegate` 中的功能拆分至多个实现文件中。实施单一功能原则，使每个文件只负责一个功能，使之服务化、定制化。如启动UIs、远程推送、Scheme跳转逻辑。

二、原理

一、文件职能

1. APP入口定制：定制的 `IMXEntryAPPDelegate`，从 `main()` 调动此文件。

该文件并不执行具体的功能逻辑，只负责接收 `UIApplicationDelegate` 协议事件并转发给 `IMXAPPServiceManager` 文件。

2. 文件管理器：`IMXAPPServiceManager` 主要负责功能文件管理。

- 功能文件的注册、分发执行
- 优先级分发策略具体实现、同步异步处理（TODO）

3. 启动文件协议：启动文件规范化、功能统一化：方便文件管理、执行。

- 协议除了继承 `UIApplicationDelegate` 外，自增了关于文件命名、优先级定义、模式（同步异步）定义。

二、技术点

1. 单一功能文件的执行：利用Obj-C中Runtime，实现多个文件的职能拆分。

- i. 单一功能文件中的 `+ (void)load`：实现本文件负责的服务注册，只有注册至管理器中，才能被执行
- ii. 在 `IMXEntryAPPDelegate` 中实现 `respondsToSelector:` 和 `forwardInvocation:` 方法，完成消息的转发；在 `IMXAPPServiceManager` 文件中，将转发进来的方法，批量交由 功能文件 来执行。
- iii. 附：使用 `NSProxy` 技术，同样可以实现服务分发功能。

2. 单一功能文件执行的优先级：目前使用Low、Default、High三种优先级来定义文件执行的顺序。

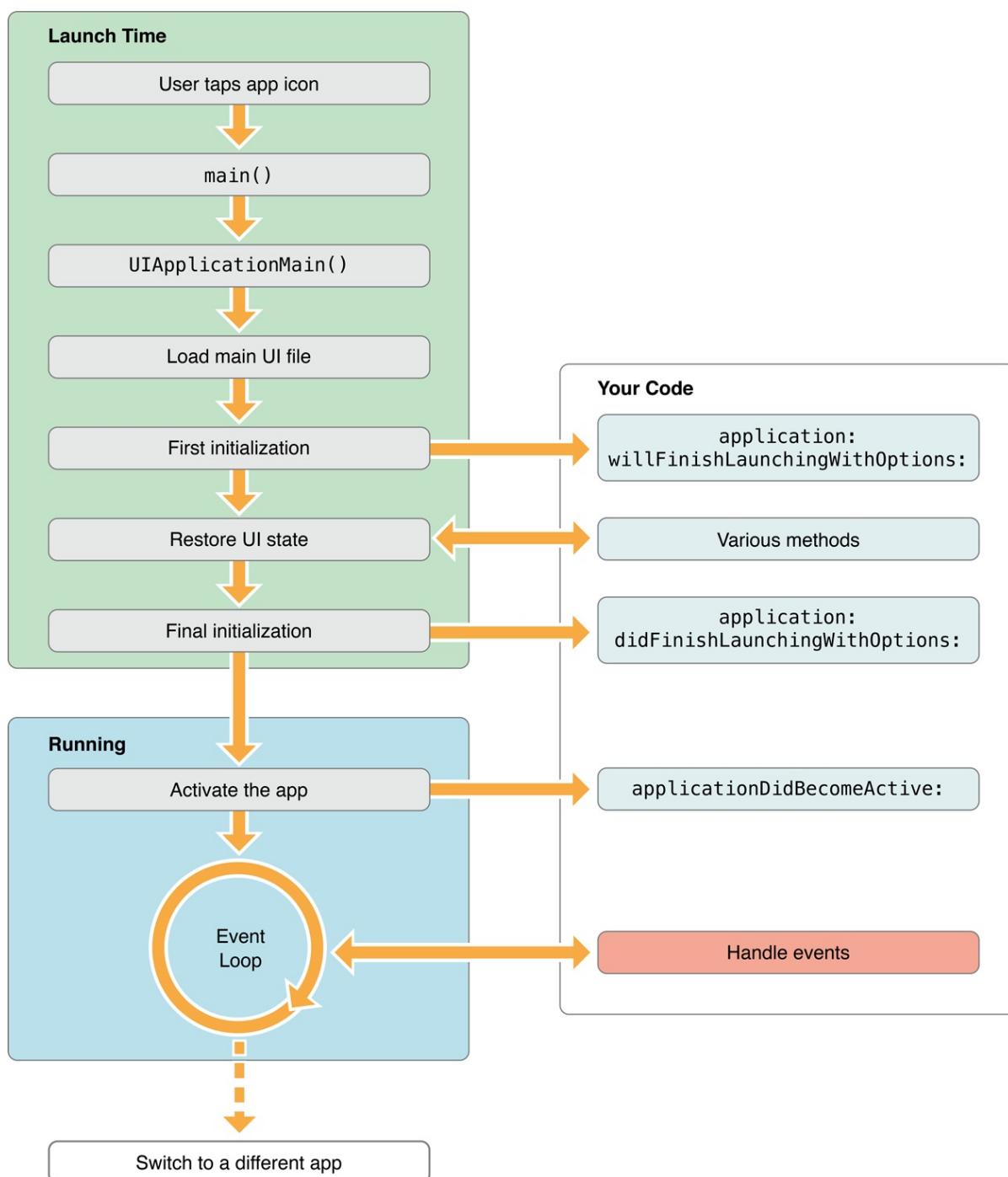
- i. 三个优先级队列，分别存放相应级别的启动文件。
- ii. 队列优先级不同，执行过程则分别依次执行。

- 3. 同步异步执行：本功能未实现，后续验证是否需要该功能。

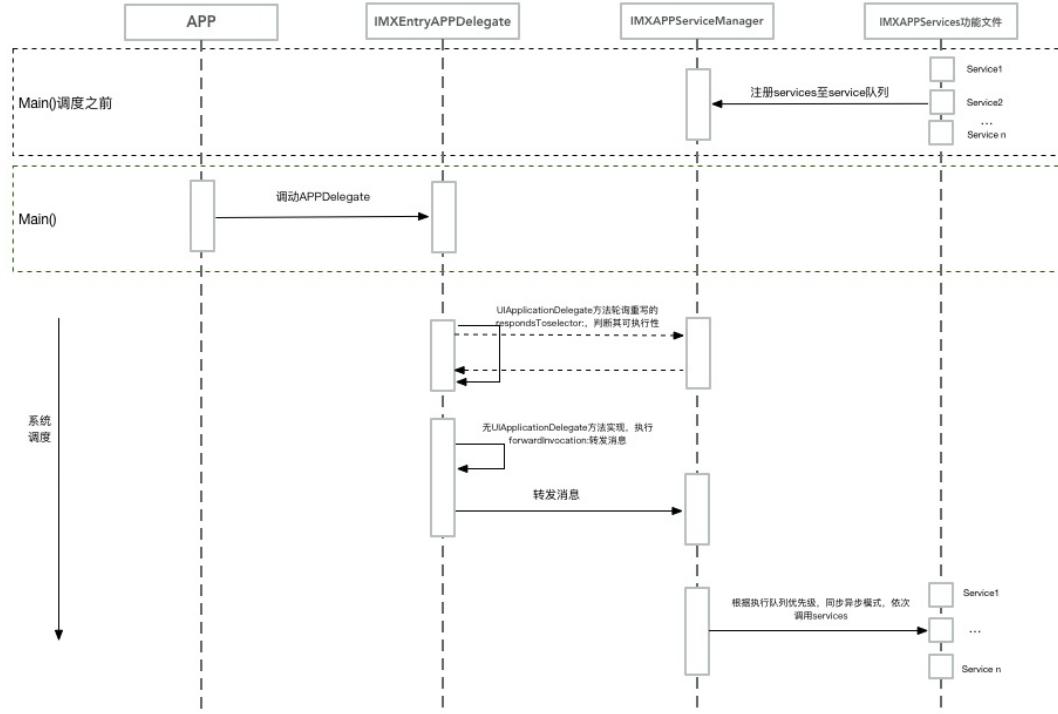
三、流程图

- 项目启动图：

自 main() 函数之后，便调用 AppDelegate 文件，进行 APP 初始化、事件监听等操作。本组件主要涵盖下图所示的 `Your Code` 块，并根据功能不同拆分为多个单一功能的文件。



- 执行流程图：



四、参考链接：

1. <http://www.qingpingshan.com/rjbc/ios/119485.html>
2. <http://southpeak.github.io/2015/01/31/cocoa-foundation-nsobject-class/>

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 填坑笔记

填坑笔记

- [UIKit相关](#)
- [Object相关](#)
- [其它](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 填坑笔记
 - 目录
 - [UIAlertController](#)

填坑笔记

目录

- [UIAlertController](#)

UIAlertController

1. 进行iPad开发时，使用UIAlertController的actionSheet属性时，注意与iPhone开发的区别。即需要添加sourceView、sourceRect。否则在ipad中运行时会crash。

```
let alert = UIAlertController.init(title: nil, message: nil, preferredStyle: .actionSheet)
//...
if isPadDevice {
    alert.popoverPresentationController?.sourceView = baseCtrl?.view
    alert.popoverPresentationController?.sourceRect = CGRect.init(x: 10, y:UIScreen.main.bounds.height - 110 , width: 300, height: 400)
}
baseCtrl?.present(alert, animated: true, completion: nil)
```

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- Object相关
 - 目录
 - leak宏

Object相关

目录

- leak宏

leak宏

```
#define MJPerformSelectorLeakWarning(Stuff) \
do { \
_Pragma("clang diagnostic push") \
_Pragma("clang diagnostic ignored \"-Warc-performSelector-leaks\"") \
Stuff; \
_Pragma("clang diagnostic pop") \
} while (0)
```

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

目录

- Xcode配置
- DNS缓存
- Mac安装APP问题

Xcode配置

1. 配置文件：`~/Library/MobileDevice/Provisioning Profiles`
2. Archive文件：`~/Library/Developer/Xcode/Archives`

DNS缓存

1. 清理DNS缓存：

- win: `ipconfig /flushdns`
- mac : `sudo dscacheutil -flushcache; sudo killall -HUP mDNSResponder; say DNS cache flushed`

Mac安装APP问题

1. `xxx.app`已损坏,打不开.你应该将它移到废纸篓，并非你安装的软件已损坏，而是Mac系统的安全设置问题，因为这些应用都是破解或者汉化的,那么解决方法就是临时改变Mac系统安全设置。
 - 出现这个问题的解决方法：修改系统配置：系统偏好设置... -> 安全性与隐私。修改为任何来源
 - 如果没有这个选项的话（macOS Sierra 10.12）,打开终端，执行 `sudo spctl --master-disable` 即可。

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间：2018-03-08 10:26:48

- [开源库](#)

开源库

- [Charts](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

charts

- [Charts源码解析（一）](#)

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

Charts 简述

1. 源码地址：<https://github.com/danielgindi/Charts>
2. 源码是由swift编写，但可用于swift和Obj-C项目。
3. 更多简介可以进入上述源码区查看README.md

正文

1. 由于该三方库体量庞大，划分的非常细致。粗略数了下，总体约十多个文件组、合一百多个文件。优点是各功能划分清楚，结构清晰。缺点是像阅读wiki，跳转比较多。所以下面将以饼图作为切入点来分析一下源码，原因其一避免了散点解析，毫无头绪；其二是马上要用到这块的内容开发 ($O(N \cdot N)O \sim$) 。
2. 文件组结构简介：
 - i. Animation：各种动画的定义和实现
 - ii. Charts: 图表文件,含基类和具体类型的图表
 - iii. Component：组件（如X、Y轴定义等）
 - iv. Data:图表数据源定义（协议+实现）
 - v. Filter:RDP算法，减少曲线中的点（Whoops）
 - vi. Formatters:格式化value
 - vii. Highlight:选中图表条目实现方案
 - viii. Interfaces:
 - ix. Jobs: 提供控件的一些位置变换
 - x. Renderers：图表渲染器
 - xi. Utils:通用工具类（如角度和弧度转换）
3. 针对饼图解析，下面给出了必要的相关类图，其中还有很多基类和辅助性的工具类，如X、Y轴，Legend描述图表，都做了简化和忽略。

当一个类有多级父类，并且在最顶层父类才具有的属性，但在子类中进行实例化。目前这种类图的描述方案是：从父类指向引用，但实际引用的子类在同一级别（黄色背景）



效果图：



1. 饼图使用方法：

1. 初始化PieChartView，并设置相关属性

可以参见PieChartView的属性部分解析

2. 协议设置
3. 动态设置其值
4. Optional实时变更

```
_chartView.drawHoleEnabled = !_chartView.isDrawHoleEnabled;
[_chartView setNeedsDisplay];
```

2. PieChartView文件

属性（部分）

1. usePercentValuesEnabled：是否使用百分比来表示切片的值，NO：使用真实值显示
2. drawSlicesUnderHoleEnabled：切片是否要绘制于Hole之下；该属性需要和drawHoleEnabled一起使用。
3. holeRadiusPercent：饼图中心的圆形半径
4. transparentCircleRadiusPercent：透明圆形半径（通常和3中属性一起使用）
5. chartDescription（ChartDescription）：chartView.chartDescription.enabled = NO;是否启用饼图右下角的文本展示
6. drawCenterTextEnabled：是否展示饼图中心文本
7. centerAttributedText：设置饼图中心文本
8. rotationAngle：饼图旋转角度（默认270->top）；0~360°范围
9. rotationEnabled：是否可拖拽旋转
10. highlightPerTapEnabled：点击放大展示选中
11. legend（ChartLegend）：每个切片数据的占比-色值对；展示于图表的右上角。

```
ChartLegend *l = chartView.legend;
l.horizontalAlignment = ChartLegendHorizontalAlignmentRight;
l.verticalAlignment = ChartLegendVerticalAlignmentTop;
l.orientation = ChartLegendOrientationVertical;
l.drawInside = NO;
l.xEntrySpace = 7.0;
l.yEntrySpace = 0.0;
l.yOffset = 0.0;
```

12. data (PieChartData--PieDataSet) : 设置切片个数和每个切片的value值

初始化

1. 渲染器PieChartRenderer和设置IHighlighter协议实体

```
renderer = PieChartRenderer(chart: self, animator: _animator, viewPortHandler: _viewPortHandler)
_xAxis = nil

self.highlighter = PieHighlighter(chart: self)
```

2. 渲染器具体实施原理，详见下文
3. highlighter具体实施原理，详见下文

draw()

```

open override func draw(_ rect: CGRect)
{
    super.draw(rect)

    if _data === nil
    {
        return
    }

    let optionalContext = NSUIGraphicsGetCurrentContext()
    guard let context = optionalContext else { return }

    renderer!.drawData(context: context)

    if (valuesToHighlight())
    {
        renderer!.drawHighlighted(context: context, indices: _indicesToHighlight)
    }

    renderer!.drawExtras(context: context)

    renderer!.drawValues(context: context)

    _legendRenderer.renderLegend(context: context)

    drawDescription(context: context)

    drawMarkers(context: context)
}

```

代码解析：

1. renderer!.drawData(context: context)在上下文中使用渲染器绘制图表
2. renderer!.drawValues(context: context):一些图表的相关绘制

总结：renderer通过弱引用Chart实例，是Chart绘制的具体实施者，而数据源则是_data。

方法

1. 角度offset等具体的计算：（略）
2. 属性的具体设置（通过存储属性和计算属性的set方法）：如

```

open var drawEntryLabelsEnabled: Bool
{
    get
    {
        return _drawEntryLabelsEnabled
    }
    set
    {
        _drawEntryLabelsEnabled = newValue
        setNeedsDisplay()
    }
}

```

3. setExtraOffsetsWithLeft:top:right:bottom:

设置图表的padding offset

ChartViewBase中的方法

4. animateWithYAxisDuration:

渲染图表并动态展示

5. Options设置属性：

```

_chartView.drawEntryLabelsEnabled = !_chartView.isDrawEntryLabelsEnabled;
[_chartView setNeedsDisplay];

```

ChartViewDelegate 协议

1. 位于ChartViewBase文件中
2. 图表切片选中时反馈给调用者

3. PieChartRenderer文件

数据渲染器：将chart对应的数据绘制于Chart上

方法

1. drawData(context: CGContext) : 具体绘制
2. slice数学分析计算
3. 属性文本、图形等计算绘制

4. PieChartData文件

饼图数据源：提供最大值，最小值、移除value等操作功能。

属性

1. `internal var _dataSets = [IChartDataSet]()`: 图表具体切片 value；`IChartDataSet` 规范统一的计算方法。

Chart 中具体使用方法：

```

- (void)setDataCount:(int)count range:(double)range
{
    double mult = range;

    NSMutableArray *entries = [[NSMutableArray alloc] init];

    for (int i = 0; i < count; i++)
    {
        [entries addObject:[[PieChartDataEntry alloc] initWithValue:(arc4random_uniform(mult) + mult / 5) label:parties[i % parties.count]]];
    }

    PieChartDataSet *dataSet = [[PieChartDataSet alloc] initWithValues:entries label:@"Election Results"];
    dataSet.sliceSpace = 2.0;

    // add a lot of colors

    NSMutableArray *colors = [[NSMutableArray alloc] init];
    [colors addObjectsFromArray:ChartColorTemplates.vordiplom];
    [colors addObjectsFromArray:ChartColorTemplates.joyful];
    [colors addObjectsFromArray:ChartColorTemplates.colorful];
    [colors addObjectsFromArray:ChartColorTemplates.liberty];
    [colors addObjectsFromArray:ChartColorTemplates.pastel];
    [colors addObject:[UIColor colorWithRed:51/255.f green:181/255.f blue:229/255.f alpha:1.f]];

    dataSet.colors = colors;

    dataSet.valueLinePart1OffsetPercentage = 0.8;
    dataSet.valueLinePart1Length = 0.2;
    dataSet.valueLinePart2Length = 0.4;
    //dataSet.xValuePosition = PieChartValuePositionOutsideSlice;
    dataSet.yValuePosition = PieChartValuePositionOutsideSlice;

    PieChartData *data = [[PieChartData alloc] initWithDataSet:dataSet];

    NSNumberFormatter *pFormatter = [[NSNumberFormatter alloc] init];
    pFormatter.numberStyle = NSNumberFormatterPercentStyle;
    pFormatter.maximumFractionDigits = 1;
    pFormatter.multiplier = @1.f;
    pFormatter.percentSymbol = @"%";
    [data setValueFormatter:[[ChartDefaultValueFormatter alloc] initWithFormatter:pFormatter]];
    [data setValueFont:[UIFont fontWithName:@"HelveticaNeue-Light" size:11.f]];
    [data setValueTextColor:UIColor.blackColor];

    _chartView.data = data;
    [_chartView highlightValues:nil];
}

```

基类

1. PieRadarChartViewBase

1. PieChartView的直接父类
2. 角度、padding offset、手势添加
3. Legend Renderer 设置

2. ChartViewBase

1. ChartView的基类
2. 主要涵盖数据源、view属性等基础性的配置

附：语法萃取

1. 导入模块：`import Foundation`
2. 判断开发平台宏定义：

```
#if !os(OSX)
    import UIKit
#endif
```

platform文件即统一了各平台的差异：

```
#if os(iOS) || os(tvOS)
    import UIKit
    .....
        (别名定义)
    public typealias NSUIFont = UIFont
    open class NSUIView: UIView
    {
        .....
    }
#endif
#ifndef tvOS
    ...
#endif
#ifndef OSX
    import Cocoa
    import Quartz
    //-
    public typealias NSUIFont = NSFont
    ....
        (扩展)
#endif
```

为支持 UIKit (iOS, tvOS) and Cocoa (OS X)，判断平台，并给予一个别名定义：

```
public typealias NSUIFont = UIFont
```

3. 扩展某些类的功能：

```
extension NSUITapGestureRecognizer
{
    final func nsuiNumberOfTouches() -> Int
    {
        return numberOfTouches
    }

    final var nsuiNumberOfTapsRequired: Int
    {
        get
        {
            return self.numberOfTapsRequired
        }
        set
        {
            self.numberOfTapsRequired = newValue
        }
    }
}
```

4. `open class` ** 修饰的类除了具有`public`的特点外，该类还可以被其它模块的类继承，而`public`修饰的类只能被模块外引用，却不可继承。

5. `fileprivate`: 访问控制。本模块本文件即文件内私有，只可在本源文件中使用

```
fileprivate var _drawHoleEnabled = true

fileprivate var _holeColor: NSUIColor? = NSUIColor.white
```

6. `==` 指针判定

- 7. Double 中的`infinity`代表无穷，它是类似 `1.0 / 0.0` 这样的数学表达式的结果，代表数学意义上的无限大。`NaN` 代表的是某些未被定义的或者出现了错误的运算，它是“Not a Number”的简写
- 8. `FLT_EPSILON` 和 `DBL_EPSILON`：`float` 和 `double` 类型的机器精度，即 `1.0` 与下一个可被 `double` 类型描述的值的差。
- 9. 关于类中的属性和计算属性，常使用私有的属性、`open`类型的计算属性来支持对外访问（同时可以设置其`setter`、`getter`方法，来监听值的变化做一些联动操作）。

```

fileprivate var _type: FillType = FillType.empty

// MARK: Properties
open var type: FillType
{
    return _type
}

```

10. `@nonobjc`、`@objc(**)`：Obj-C和Swift混编时使用

11. `fatalError`：Release编译时期抛出错误、`Assert`只在Debug时使用

某些方法要求子类必须override：`fatalError("这个方法必须在子类中被重写")`

12. `guard`：

```

guard let
    viewPortHandler = self.viewPortHandler
else { return }

```

13. `_displayLink?.remove(from: RunLoop.main, forMode: RunLoopMode.commonModes)`

14. 宏定义

```

#if os(tvOS)
    // 23 is the smallest recommended font size on the TV
    font = NSUIFont.systemFont(ofSize: 23)
#elif os(OSX)
    font = NSUIFont.systemFont(ofSize: NSUIFont.systemFontSize())
#else
    font = NSUIFont.systemFont(ofSize: 8.0)
#endif

```

15. `@available`: 声明这些类型的生命周期依赖于特定的平台和操作系统版本。

`#available` 用在条件语句代码块中，判断不同的平台下，做不同的逻辑处理

```

if #available(iOS 8, *) {
    // iOS 8 及其以上系统运行
}

```

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- 逆向与安全
-

- 逆向与安全

逆向与安全

- 工具

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 工具

工具

- [hopper](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

工具

- [hopper](#)
- [Charles](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 架构设计

架构设计

- 设计模式
- 架构

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 设计模式

设计模式

- UML简介
- AOP

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

- [UML简介](#)

UML简介

- [1. uml类图手册](#)
- [2. uml时序](#)
- [3. uml活动图](#)
- [4. uml状态机图](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- UML之类图
 - 前情提要：
 - 概念
 - 属性
 - 关系
 - 直线关联
 - 带箭头关联
 - 包含关系
 - 继承关系
 - 依赖关系
 - 类的递归关系
 - 实战
 - 总结：
 - 类图分析需求步骤：

UML之类图

前情提要：

之前做账户体系总结时，需要用到UML来绘制图谱结构，所以在阅读相关书籍和网站时，将相谱做了简要记录，下次忘记时可以快速参阅，毕竟好记性不如烂笔头~~

概念

1. 需求中涉及到的业务概念、人物等都可以抽象为类。
2. 提炼类图元素：识别出类、提炼类的关键属性和操作、描绘类之间关系

属性

1. 类图即一个矩形方框：类的名字、中间为属性、下面为操作。



关系

类之间的关系（简化类图，只需给出类名就可以了）

直线关联

1. 关联关系：AB之间有关系，但又不确定具体是什么关系



2. 一对关系：一个C对应一个D的关系



3. 一对多关系：一个E对应0至多个F。（*表示0到多个）



4. 一对n关系：一个G对应0到n个H。（0..3：0到3个）

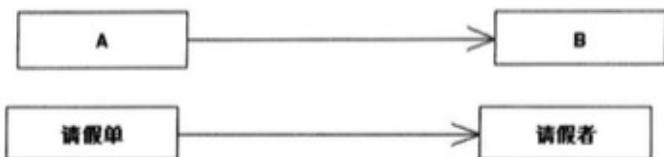


5. 角色关系：（+：public属性）



带箭头关联

1. 导航关系：A可找到B（如A有属性保存B的引用；1:A含有B的引用；2:A可追溯到B）



包含关系

1. 空心菱形：弱包含（聚合），（部门没有了，员工可以继续存在；员工可以有多个部门）

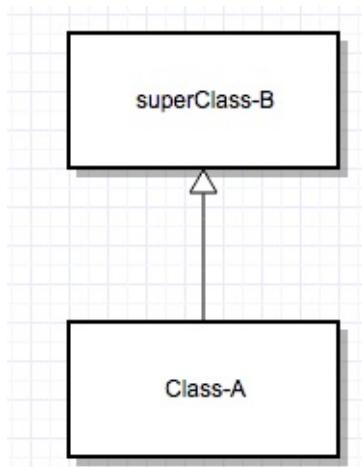


2. 实心菱形：强包含（组合），（部门没有了，员工不存在；员工只有一个部门对应）

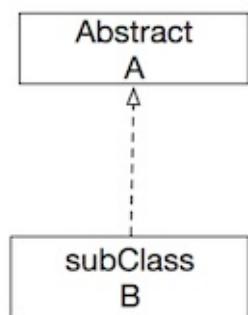


继承关系

1. A继承B（泛化）：A中具有了B的所有特点

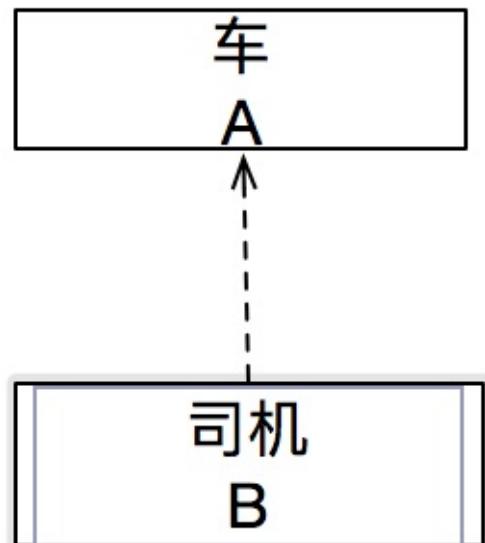


2. 实现关系用一条带空心箭头的虚线表示（B继承自A：A是接口，无具体实现的抽象类。即继承抽象类）



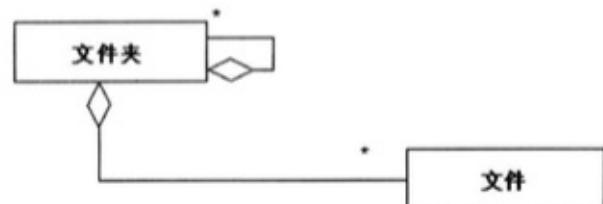
依赖关系

1. A依赖于B，也可理解为A需要依赖B的完成才能完成



类的递归关系

1. 自包含递归关系：常用于树形的业务结构（自关联关系同样适用）



2. 三角关系（关联类）：如表示公司和雇员关系的直线上，拉出一条虚线，另一端连接同类，合同类即为关联类。

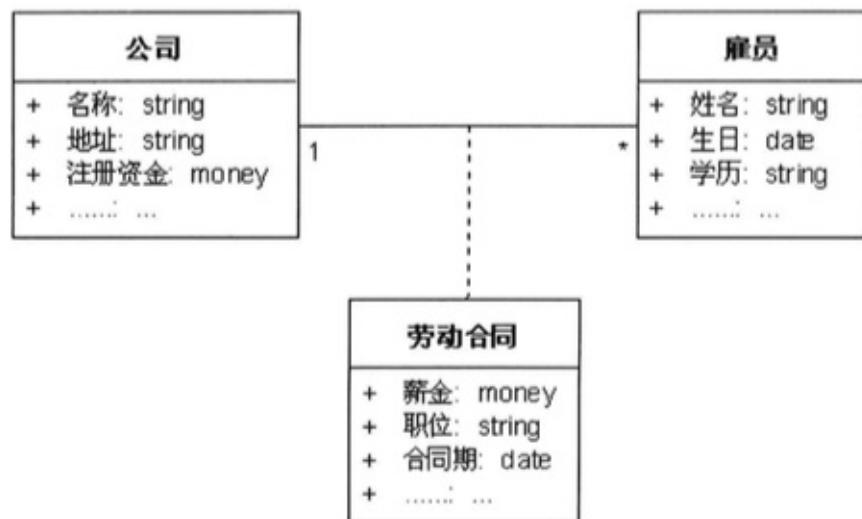
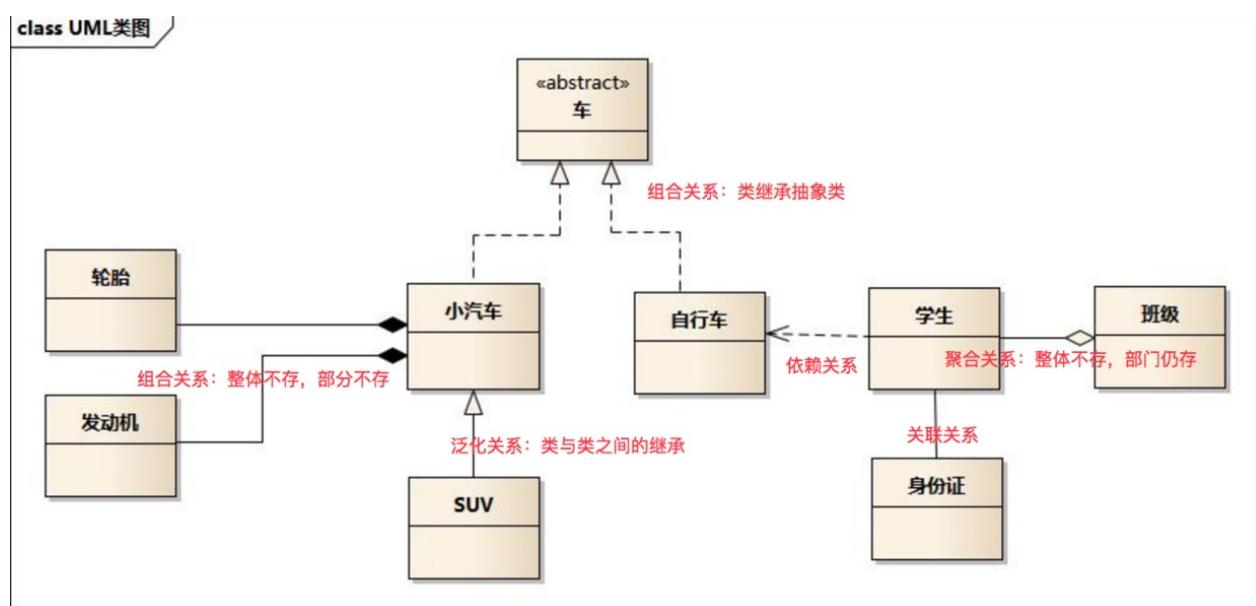


图 3.33 公司、雇员、劳动合同的关系 I

实战



总结：

表 3.1 类图基本语法

通俗叫法	学术名称	图例
直线关系	关联 (Association)	<pre> classDiagram class G class H G "1" -- "0..3" H class A class B A --> "0..3" B </pre>
包含关系	聚合 (Aggregation) 组合 (Composition)	<pre> classDiagram class 部门 class 员工 部门 o-- "*" 员工 部门 *-- "*" 员工 </pre>
继承关系	泛化 (Generalization)	<pre> classDiagram class A class B A --> "0..3" B </pre>
依赖关系	依赖 (Dependency)	<pre> classDiagram class A class B A -.-> "0..3" B </pre>

类图分析需求步骤：

1. 识别出类
2. 识别出类的属性操作
3. 描绘类的关系（直接关系）
4. 对各类进行分析、抽象、整理

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- UML之时序图
 - 提要
 - 图解
 - 实践建议
 - 详解

UML之时序图

提要

UML烂笔头之二

图解

1. 正常版(顺序图基本语法)

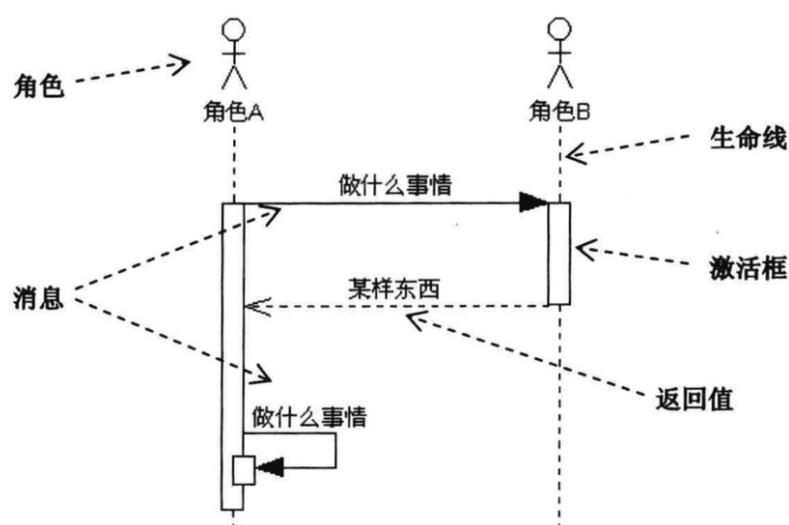
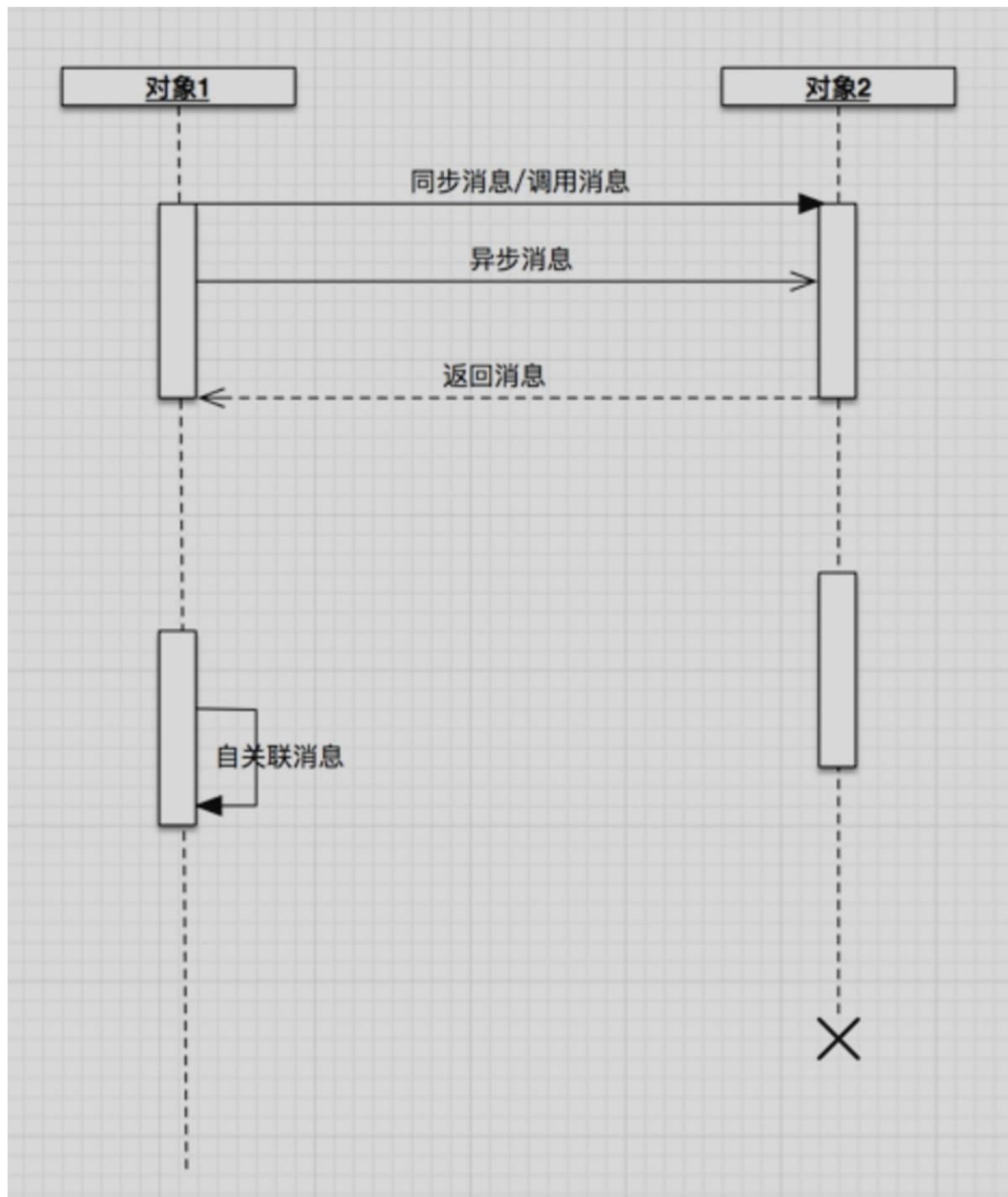


图 6.3 顺序图基本语法

2. 循环、条件、可选分支

! [u2](uml时序图/u2.png)

1. 含自关联



实践建议

1. 流程中提取分解一个一个的子流程
2. 子流程参与角色
3. 两两角色之间的交互组织

详解

1. 定义：显示对象以及对象之间交互的顺序。
2. 建模元素
 - i. 角色（Actor）：人、系统或者子系统等
 - ii. 对象（Object）：
 - iii. 生命线（LifeLine）：从对象图标向下延伸的一条虚线，表示对象存在的时间
 - iv. 控制焦点（Focus of Control）：表示时间段的符号，在该时间段内对象将执行某些操作，用小矩形表示
 - v. 消息：一般分同步消息（syn）、异步消息（Asyn）、返回消息（return）。
 - vi. 自关联消息：表示方法的自身调用以及一个对象内的一个方法调用另外一个方法。

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间：2018-03-08 10:26:48

- UML之活动图
 - 概念
 - 图详解
 - 文详解
 - 示例
 - 结语

UML之活动图

概念

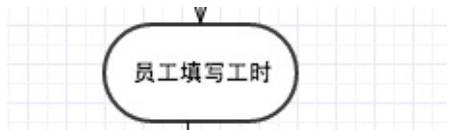
1. 结构建模：表达静态内容；（类图）
2. 行为建模：表达动态内容；（活动图、顺序图、状态机图）
3. 活动图（Activity Diagram）：表达流程的常用UML图
4. 针对行为建模，流程可能是某一个角色通过多个动作来完成某项工作，也可能是多个角色参与，历经多个动作步骤，并完成某项工作。

图详解

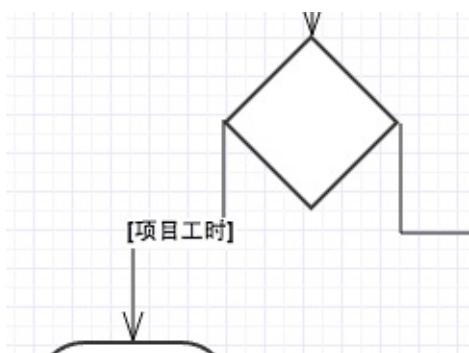
1. 开始状态和结束状态（一个开始态、一个或多个结束态）

- 开始状态（Initial State）: ●
- 结束状态（Final State）: ○

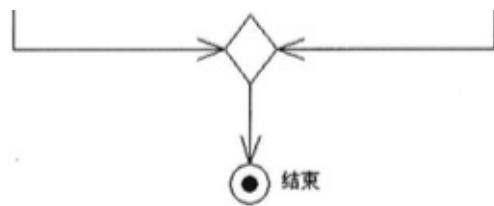
2. 一个活动（Activity即一个步骤）：圆角矩形+主谓宾描述语



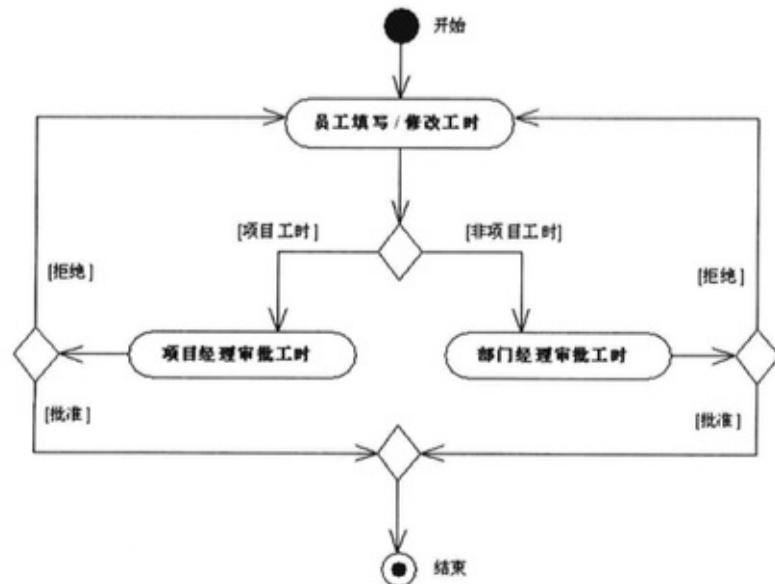
3. 条件判断：分支菱形（每条分支上使用[条件]阐述条件）



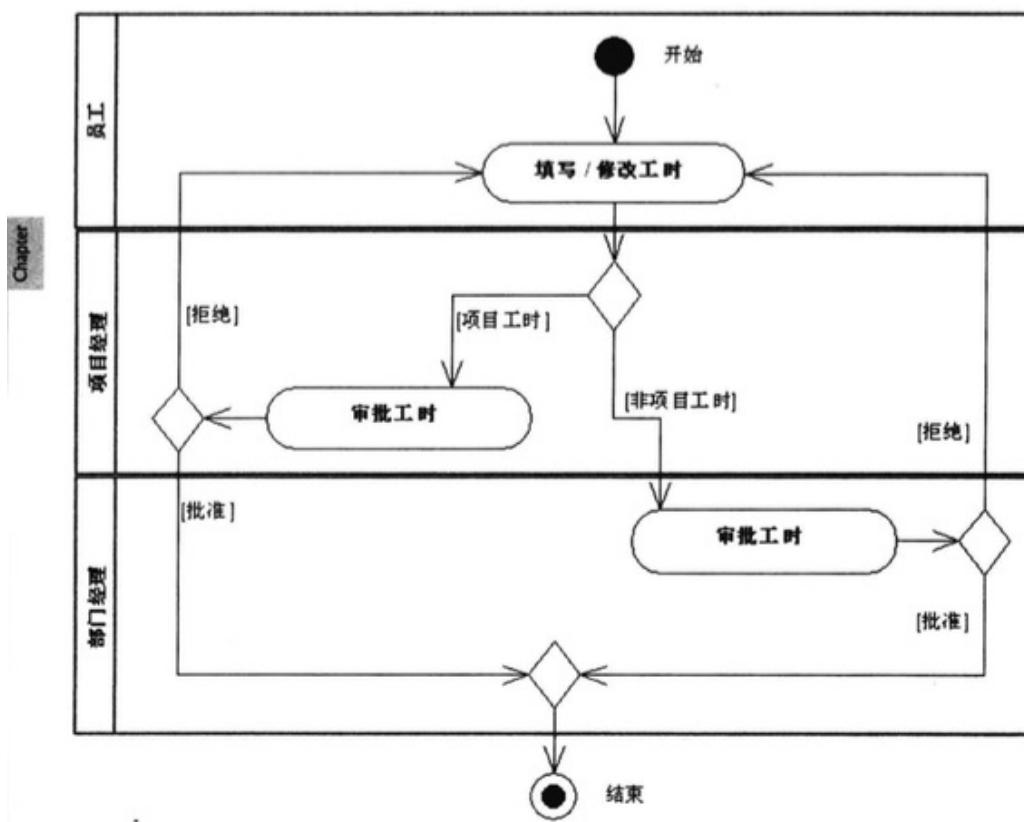
4. 合并分支：合并分支菱形（前面若有分支线路，后面可能合并）



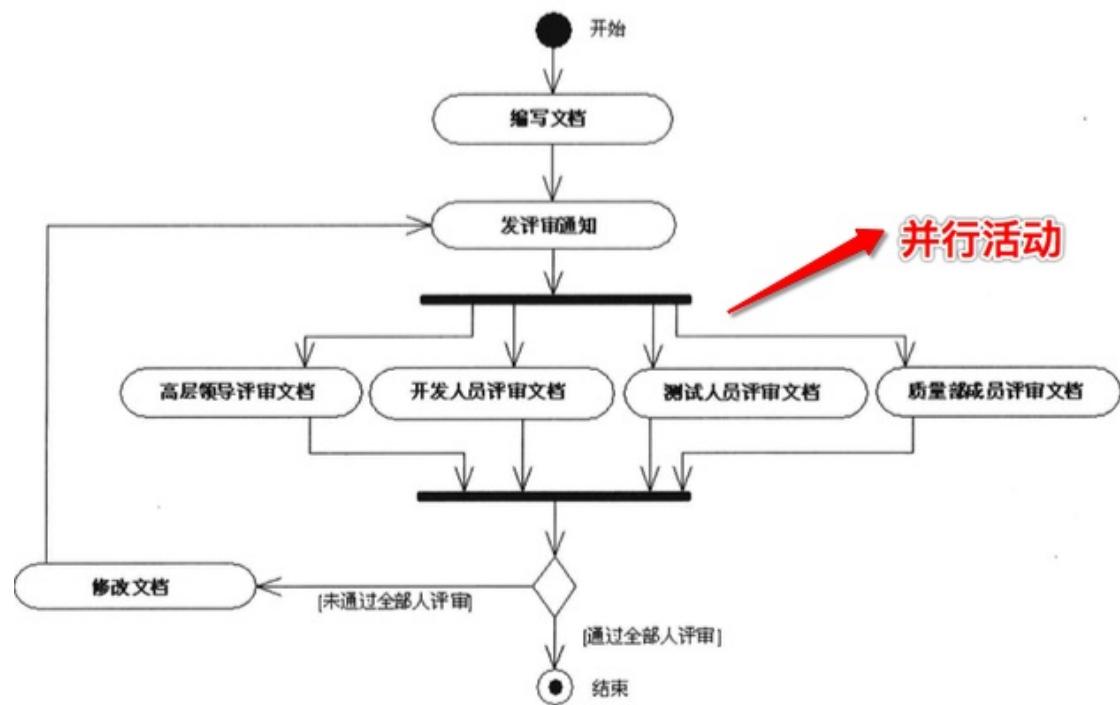
5. 综合上述，给出一个完整示例图解：



6. 池道分区（Swimlanes）：更好的表达Activity的发起者



7. 并行分支：有分支，必须有会合（黑短棒）



文详解

1. 对象流：工作产品的表达

活动图中的矩形框，文字带下划线的即为对象

2. 连接件：活动图的组织

左边：指向另一张图

右边：从此处开始继续活动图

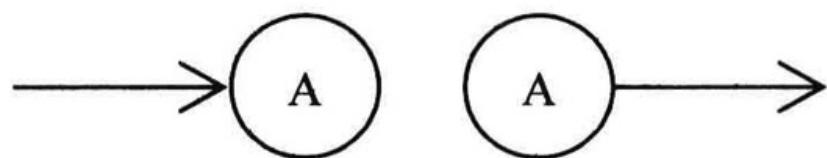


图 4.9 连接件（Connector）

3. 活动的粒度问题

活动和动作：活动最终可细分n个动作；动作是不可再分的步骤。



图 4.10 活动（Activity）与动作（Action）

示例

1. 一个活动图只表示一个事件的流程
2. 一个活动图：目的、角色参与、先绘主干再分支、适当注解

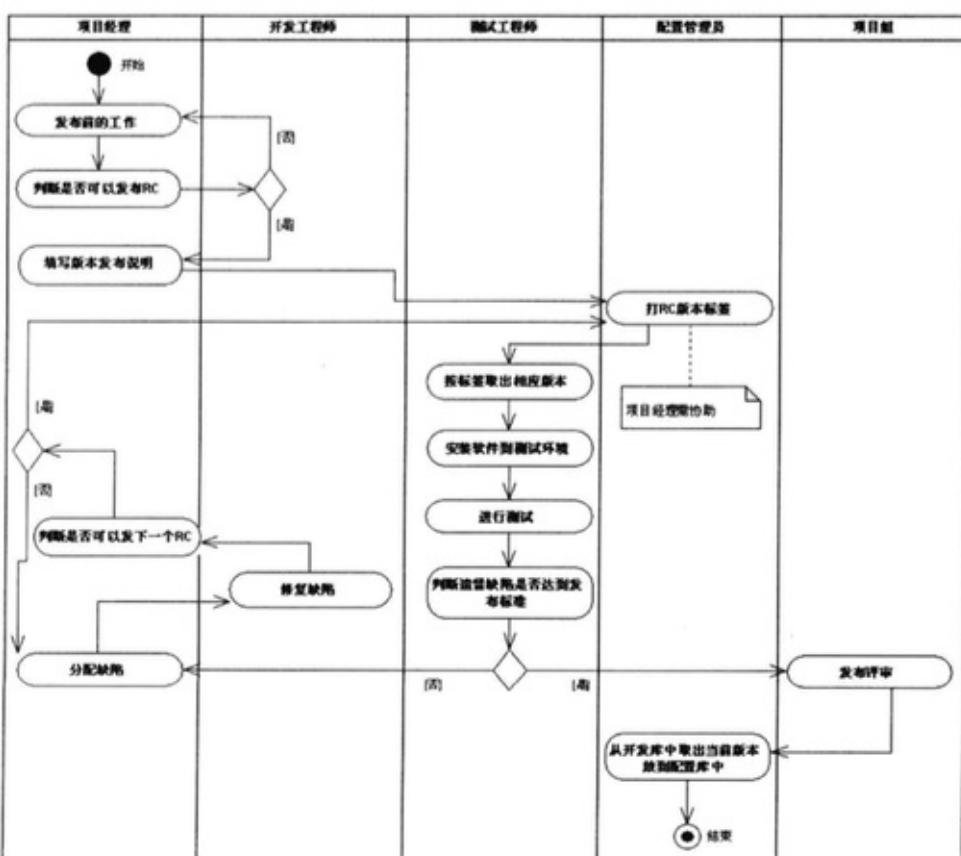
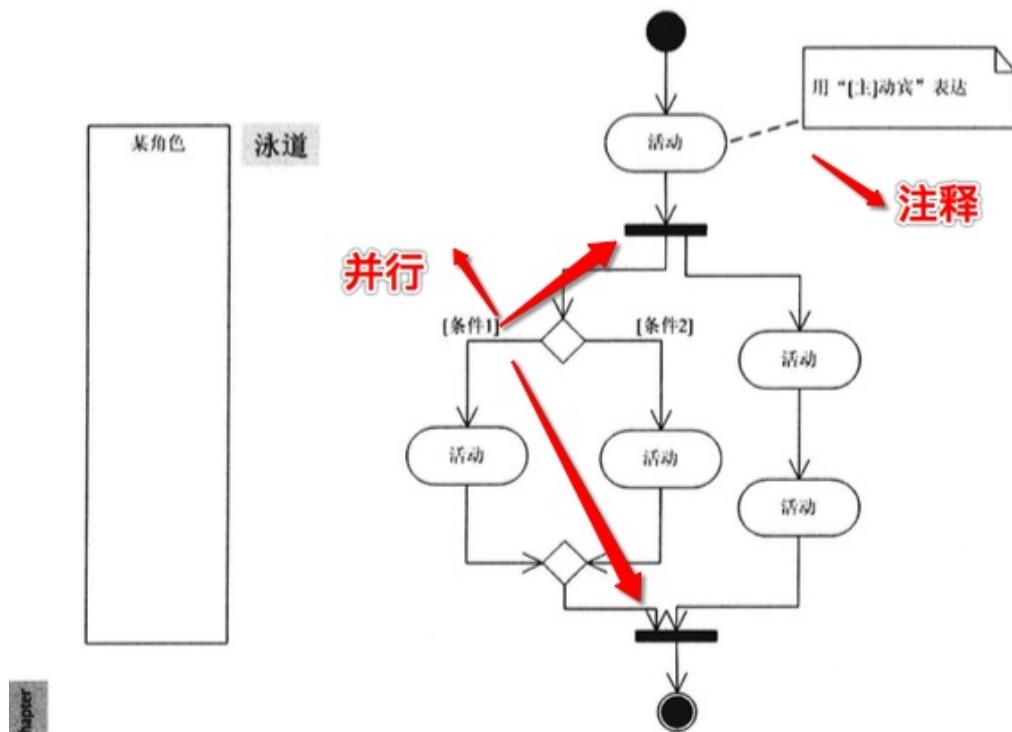


图 4.12 软件版本发布流程

结语

1. 活动图语法



2. 对象流

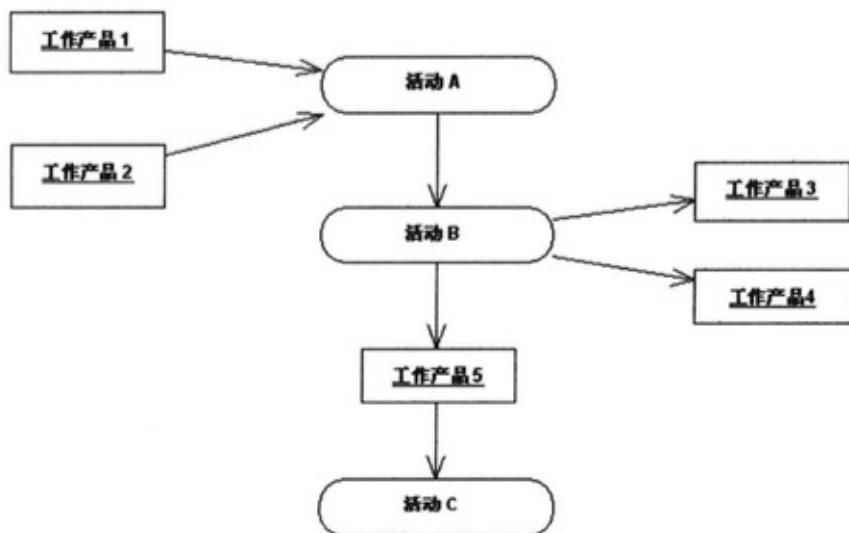


图 4.14 控制流与对象流

3. 步骤

- i. 明确流程的目的
- ii. 流程的角色参与
- iii. 先主干，逐渐添加必要的分支（适当注释）

Copyright © iOS 开发手册小组 all right reserved, powered by Gitbook 修订时间：2018-03-08 10:26:48

- UML之状态机图

- 概念
- 实例
- 总结

UML之状态机图

概念

1. 状态机图：针对某个事务的状态变化来展示流程

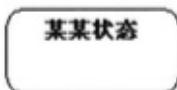
活动图：通过活动先后顺序来展示流程

2. 一个开始状态、1个或多个结束态

i. 开始结束状态图

- **开始状态 (Initial State):** ●
- **结束状态 (Final State):** ○

i. 圆角矩形表示状态



3. 活动图和状态机图的(圆角矩形)区分



图 5.3 活动图、状态机图圆角框的区别

4. 区别对比

表 5.1 活动图、状态机图圆角框的区别

	活动图	状态机图
文字表达	采用主动宾或动宾的表达方式，表示某某做什么事情	一般使用形容词或名词，表示某种状态（注1）
框框的形状	左右两边框全部都是弧线	只有四个角是弧线
注意，在某些画图工具中，这两种框可能区别不大		

实例

1. 请假流程

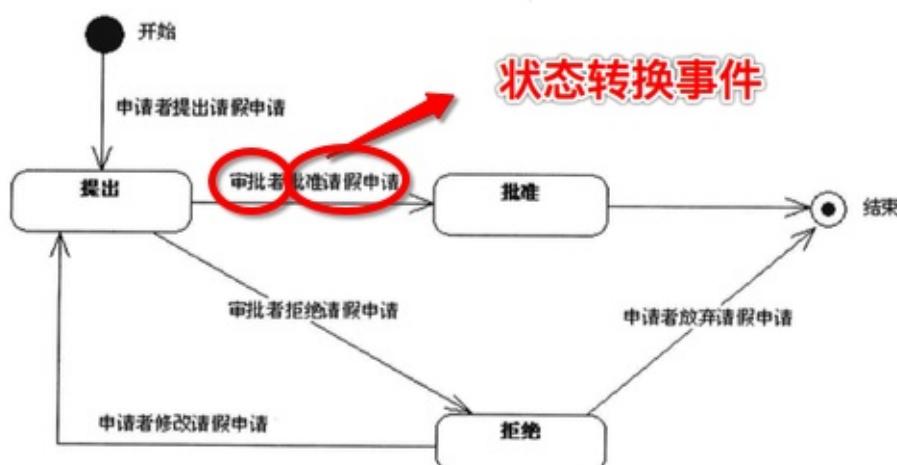


图 5.2 请假流程状态机图

2. 增强版

- i. 状态的确定精确与否，是衡量状态机图好坏的标准。
- ii. 多级审批情况

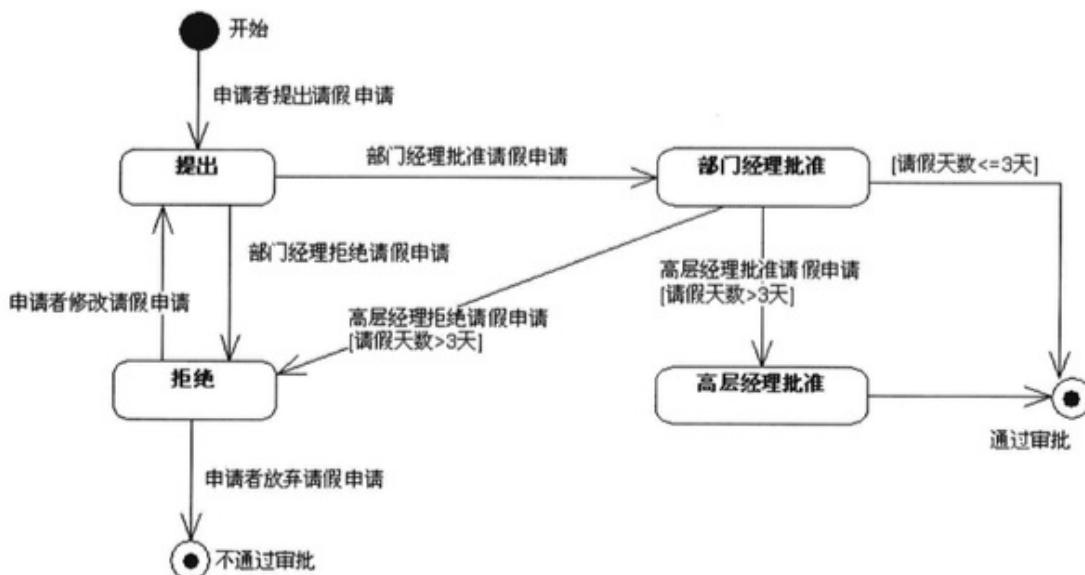


图 5.8 请假申请两级审批状态机图

3. 对应的活动图：

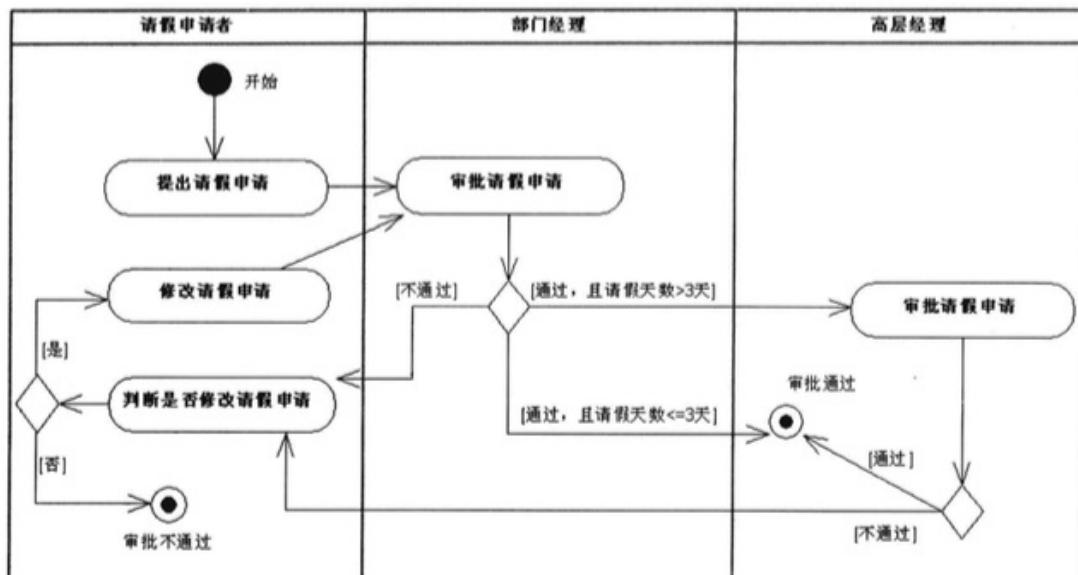


图 5.7 请假申请两级审批活动图

总结

1. 流程的目的
2. 角色参与确定
3. 围绕的事务确定、以及其状态确定
4. 精简状态，转换

Copyright © iOS开发手册小组 all right reserved, powered by Gitbook 修订时间：2018-03-08 10:26:48

设计模式

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 架构

架构

- 项目模块化开发
- MVC
- MVP模式
- MVIP模式
- MVVM

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 项目模块化开发

项目模块化开发

- 模块化开发概述
- 子工程模板搭建
- 模块化开发详情
- 项目部署
- 组件实现细节
- CocoaPods管理子工程
- CocoaPods私有库framework创建和使用

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 目录
- 提要
 - 一、往期项目
 - 二、iOS模块化项目
- 实现
 - 一、子工程模板
 - 二、模块化开发
 - 三、项目部署
 - 略
 - TODO :

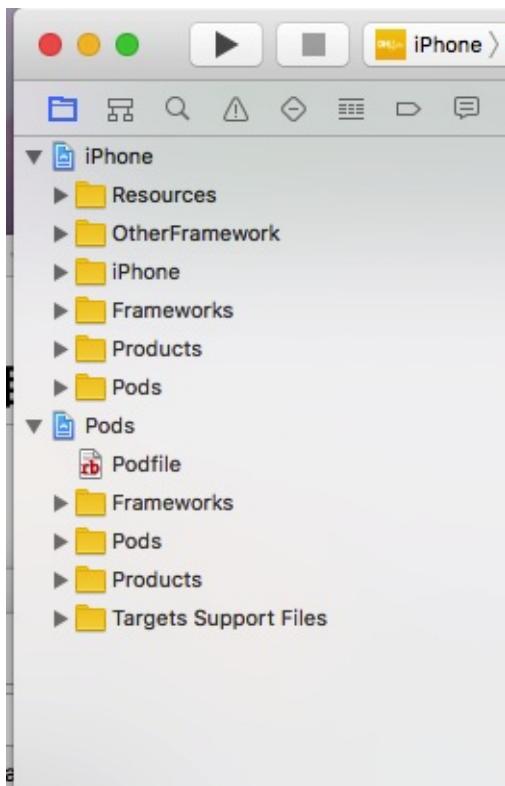
目 录

- 提要
 - 1. 往期项目
 - 2. iOS模块化项目
- 简介
 - 1. 子工程模板
 - 2. 模块化开发
 - 3. 项目部署
- 组件开发细节

提 要

一、往期项目

- 结构图



- 说明

1. 对应的黄色文件即为逻辑分组，逻辑清晰。
 - Group实际路径可以有同名的物理文件隔离，也可以无单一文件隔离。
2. cocapods管理+逻辑分组实现一个项目的开发，即便模块化开发也可以实现。

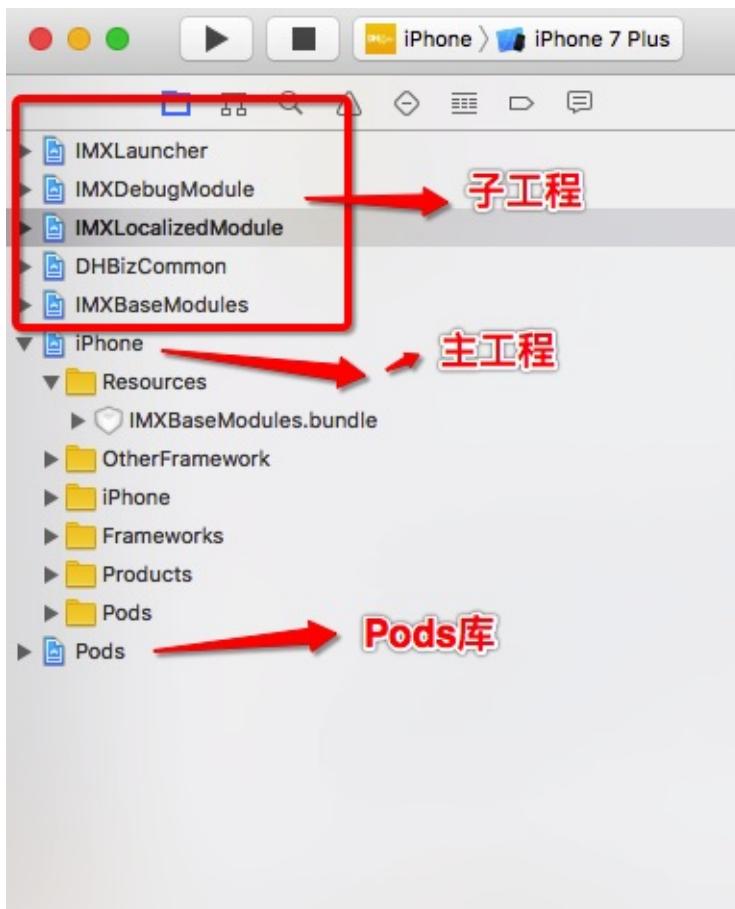
- 缺点

1. 逻辑分组无物理隔离性，模块化开发中，后期很容易造成文件分散，功能划分混乱。
 - 主要原因有：开发人员增加、项目体积骤增、开发工期紧张等

二、iOS模块化项目

模块化项目是什么样子的？

- 结构图（初步方案）



- 说明

1. 项目以 cocoaPods公共库管理 + 子工程 + 主工程 相结合的方式实现。

- 下文 子工程模板 会做概要介绍

- 优势

1. 各组件、模块物理隔离，完美解决往期项目中逻辑分组的缺点。
2. 子工程各司其职，单一独立。甚至包括相互间引用也有独立的路由组件实现，便于后期提取编译为组件、模块库，进而以Pods分发管理等。

- 后期具体完善方案请参见：[APP模块化开发详情](#) 中 进化方案 说明。

- 缺点

1. 开发阶段，编译速度慢
 - 多个子工程编译、链接耗时

实现

一、子工程模板

一个模块里面都有什么？

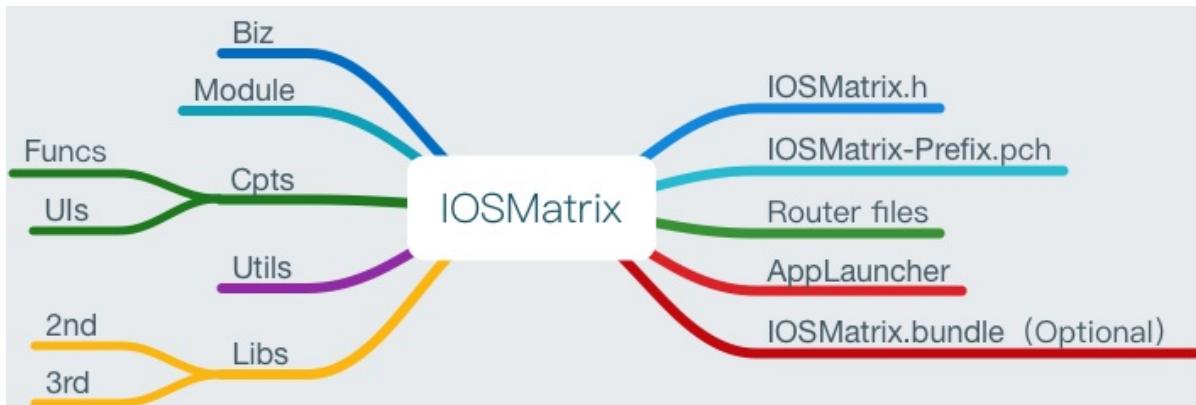
- 即涵盖脚手架结构、子工程自动化生成、链接主工程实施方案等

工程配置细节，参考文档：[iOS子工程模板搭建](#)

- 主工程目录：



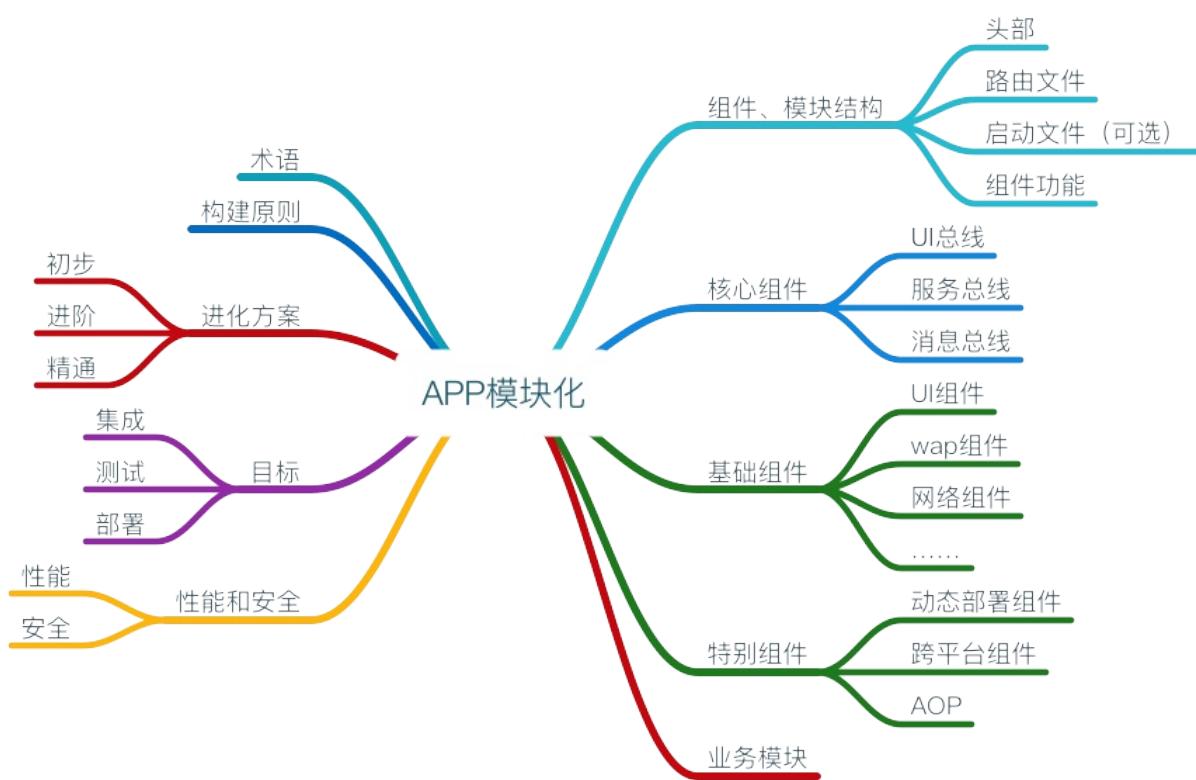
- 子工程目录：



二、模块化开发

组件、模块都有哪些？后期规划是什么？

- 开发图谱



- 最终目标：单一组件、模块，单一产出；优化测试；实现权限管理(GitLab私有化)、实现分支管理；编译集成均实现可视化等。

详情参考文档：[iOS模块化开发详情](#)

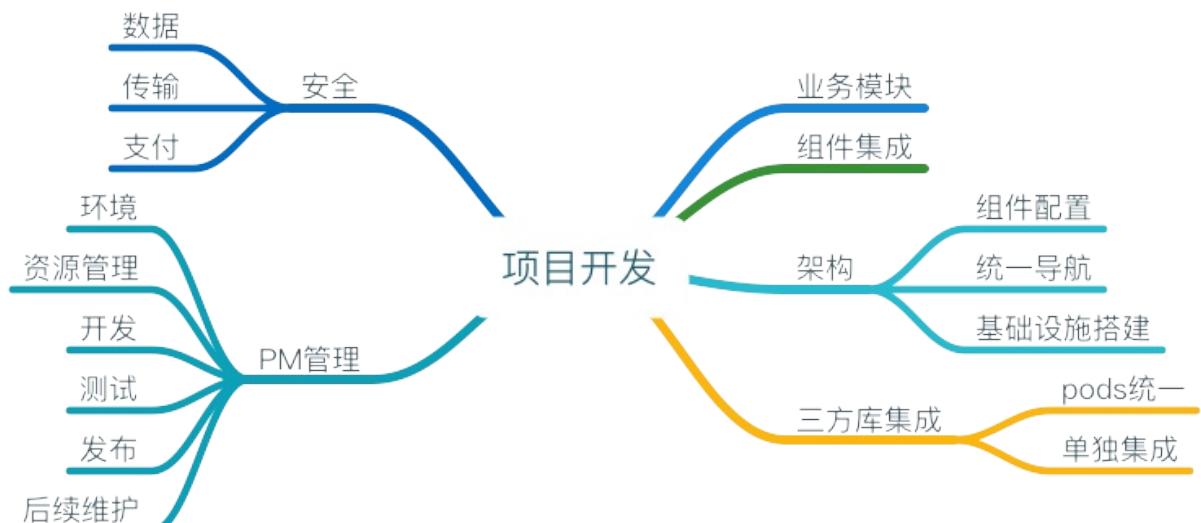
三、项目部署

这是一个项目周期

- APP结构图（初稿）



- 消息组件不应当承担其它的组件通讯工作。
- 路由组件承担 业务<->组件 、 业务<->业务 、 组件<->组件 之间的路由寻址功能。
- 项目图谱



- 详情参考文档：[项目部署](#)

略

1. MVIP模式:后续文档

TODO :

1. 后续业务开发前以及开发中，实施架构文档内容，逐步丰富完善组件

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- iOS子工程模板搭建
 - 一、子工程
 - 二、主工程
 - I 设置
 - II 开发
 - 三、框架
 - I 主工程框架
 - II 子工程框架
 - 四、脚本（需细化）
 - 注：

iOS子工程模板搭建

一、子工程

1. 创建子工程模板，命名为 `IOSMatrix.xcodeproj`。
 - i. 工程内部添加通用的文件组（见 `三、II 子工程框架`）
 - ii. 添加通用 Target Aggregate，命名为 `IOSMatrixAgg`。并添加复制脚本。

二、主工程

- 以新工程为例，老工程重构的话逐步替代即可

I 设置

1. 创建新工程，如命名为 `MainMatrix.xcodeproj`
 - i. Pods 配置文件初始化。
 - ii. 创建 `APPFks` 文件夹，涵盖子工程复制脚本添加进来的静态库，供主工程使用。
 - iii. 创建 `Resources` 文件夹，涵盖自子工程复制添加进来的静 `bundle` 资源文件（脚本未支持，需手动链接：不需复制，资源文件仍在子工程静态库中），供主工程使用。也涵盖主工程的资源文件。
 - iv. 将子工程模块置入 `MainMatrix.xcodeproj` 中，并在主工程目录中引入脚本 `createapp.sh`。
 - v. 工程目录设计（见 `三、I 主工程框架`）。
2. 依据子工程模板+脚本，生成具体子工程，并引入主工程 (`**.xcworkspace`)。

- i. 在终端键入 `sh createapp.sh **name` 即可生成子工程。
- ii. 拖动子工程 `**name.xcodeproj` 到 `**.xcworkspace` 中。
- iii. 子工程同名头文件手动更改内容。

3. 编译子工程

- i. 在 `Target-Build Phases` 中，将其中需要暴露的文件置入 `Public` 选项下。
- ii. 选择 `Scheme`：子项目 `-Generic iOS Device`，`⌘+B` 执行编译，得到 `framework` 库。
- iii. 选择 `Scheme`：子项目 `Target-Generic iOS Device`，`⌘+B` 执行编译，得到 `Appsfks` framework 库。（首次生成，需手工导入工程中）
- iv. `bundle` 文件需 `copy` 至 `framework` 中（若有），主工程链接即可。资源文件仍在子工程静态库中。

II 开发

1. 子工程的开发可以在主项目中开发，也可自己做测试平台单独开发。

该部分属于项目开发范畴，后续会有完整方案

三、框架

I 主工程框架

主工程：属于子工程的调度中心，除了一些必要的初始化工作，应不直接参与各种业务、功能的处理

图谱



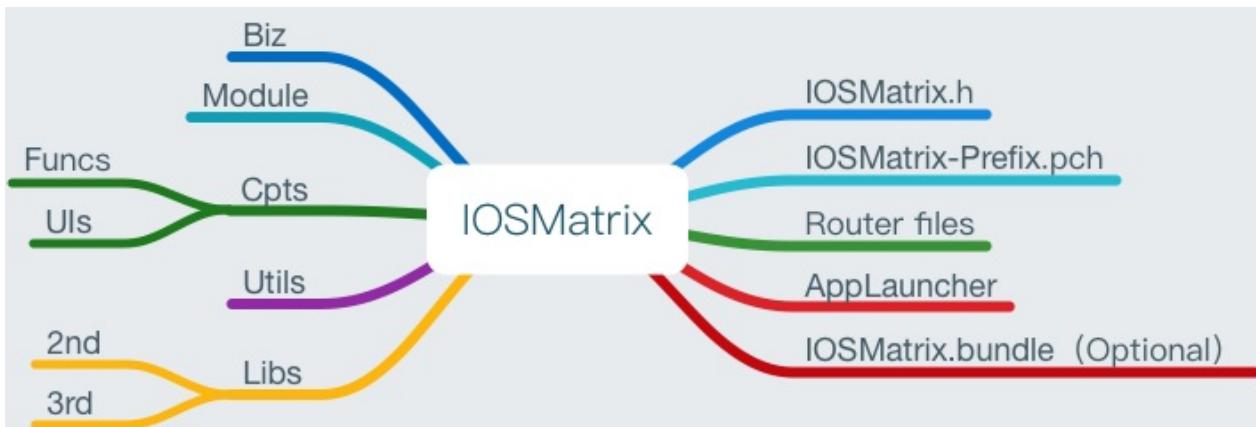
说明

- 右边分支为主工程目录集合；左边分支为其它目录集合。
- 说明顺序自左至右，自上而下。

- 图谱做了路径简化，部分路径、文件做了简化处理。
- *.xcodeproj: 主工程文件
 - 内部含有子工程模板:由于脚本路径问题，暂时将模板置于此处
- *.xcworkspace: pods 管理下的项目启动文件
- module: 功能组件（含有多个）
- component: 业务模块（含有多个）
- Apps: 组件、模块生成的静态库
- Pods: 三方库文件集合
- Resources: 自身资源文件、链接组件模块中的bundle 资源文件
- AppFks: 链接组件模块生成的静态库（即 Apps 中库文件）
- MainMatrix: main.m 文件，主工程文件集合
- frameworks: 系统库文件链接
- 其它：如 pch 文件、Products 文件夹等

II 子工程框架

图谱



说明

- 右边分支为具体文件；左边分支为目录集合。
- 自上至下，自右及左说明
- 与工程同名的h头文件，需要修改宏定义设置。

- import所有需要公开暴露的文件，便于外部引用
- 若不添加该文件，会出现警告 no umbrella header ,故添加之，若使用，需要取消注释。
- 版本信息：

```
FOUNDATION_EXPORT double IOSMatrixVersionNumber;
```

```
FOUNDATION_EXPORT const unsigned char IOSMatrixVersionString[];
```

- **pch文件设置**

- 设置：进入 Target->Build Settings : Precompile Prefix Header = YES ; Prefix Header = \$(SRCROOT)/\$(PROJECT)/\$(PROJECT)-Prefix.pch 即可。
- 功能：子工程内部需要引入的文件、公用宏定义等。

- **Router files**：路由文件,需配合路由组件

- URI寻址方式获取该子工程中任一文件路由，包括弹出方式、参数传递、回调等功能
- 具体路由文件形式根据路由组件不同而有所区别，但功能一致。
- 详细使用方法见Router组件使用说明。

- **AppLauncher**：启动文件，需配合服务组件

- 功能一：hook App启动流程。实现在主工程启动初期，子工程需要初始化的情况。如推送服务。
- 功能二：Service服务。服务化各组件，便于获取和使用。
- 详细使用方法见AppLauncher组件使用说明。

- **IOSMatrix.bundle**（可选）：资源文件集合

- 放置图片、音频等多媒体文件
- 子工程资源文件若只在内部使用，一般使用Asert(图片)或者Resources文件夹即可。
- 资源文件若被外部调用，则需打包成bundle文件，编译会随着framework暴露出去。

- **Biz**：具体的业务模块集合

- 子工程为功能组件时，Biz为空
- MVC+MVIP设计模式运用

- **Module**：功能组件集合

- 子工程为业务模块时，此为空

- **Cpts**：业务模块集合。

- 被多个子工程公用，但尚未或者不足以组件化。
- Funcs：功能型模块
- UIs:视图型模块
- Utils：子工程内部公用的一些工具类文件
- Libs：子工程内使用的库文件
 - 2nd:二方库
 - 3rd:三方库
 - 若需导入系统库，则直接添加至 Build Phases 即可。
 - 若某三方库其它子工程也在使用，则作为公共库在主工程使用Pods管理即可。
附：若子工程需要使用pod对应的库，需在target-build settings中设置Header Search Pths :/Pods/Headers/Public (recursive) 即可。

在Frame search Paths下置.../Apps(non-)、.../Pods(recu)

四、脚本（需细化）

- 以上各部分需要使用完善的脚本来实现自动化，此部分也是后期主要优化点
 - | 参考淘票票脚本实现，在此感谢曾经共事的大神。
 - 诸如权限管理、可视化操作等
 - | 参考京东模块化方案

注：

1. 发布环境更改时，注意测试子工程库文件是否是arm架构。

- | 子工程目前要求支持armv7 arm64，可使用 lipo -info **.framework检测

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- iOS模块化开发详情（初稿）
- 理论
 - 图谱梗概
 - 目录
 - 一、术语
 - 二、构建原则
 - 三、进化方案
 - 四、目标
 - 1. 集成
 - 2. 测试
 - 3. 部署
- 实践
 - 一、组件、模块结构
 - 1. 主要包括：
 - 2. 细节
 - 二、核心组件
 - 1. UI总线：路由策略组件
 - 2. 服务总线：初始化启动策略
 - 3. 消息总线
 - 三、基础组件
 - (1) UI组件
 - (2) Wap容器组件
 - (3) 网络组件
 - (4) Debug组件
 - (5) 数据收集组件（Data Collector）
 - (6) 持久化组件（Data Persistence）
 - (7) 安全组件
 - (8) 扫描组件
 - (9) IM组件
 - (10) push组件
 - (11) 分享组件
 - (12) 设备信息组件
 - (13) 国际化组件
 - (14) 用户体系组件
 - (15) AB test组件
 - (16) Style Kit组件
 - (17) 扩展组件
 - 四、特别组件
 - (1) 动态部署组件
 - (2) 跨平台组件（weex组件或RN组件）

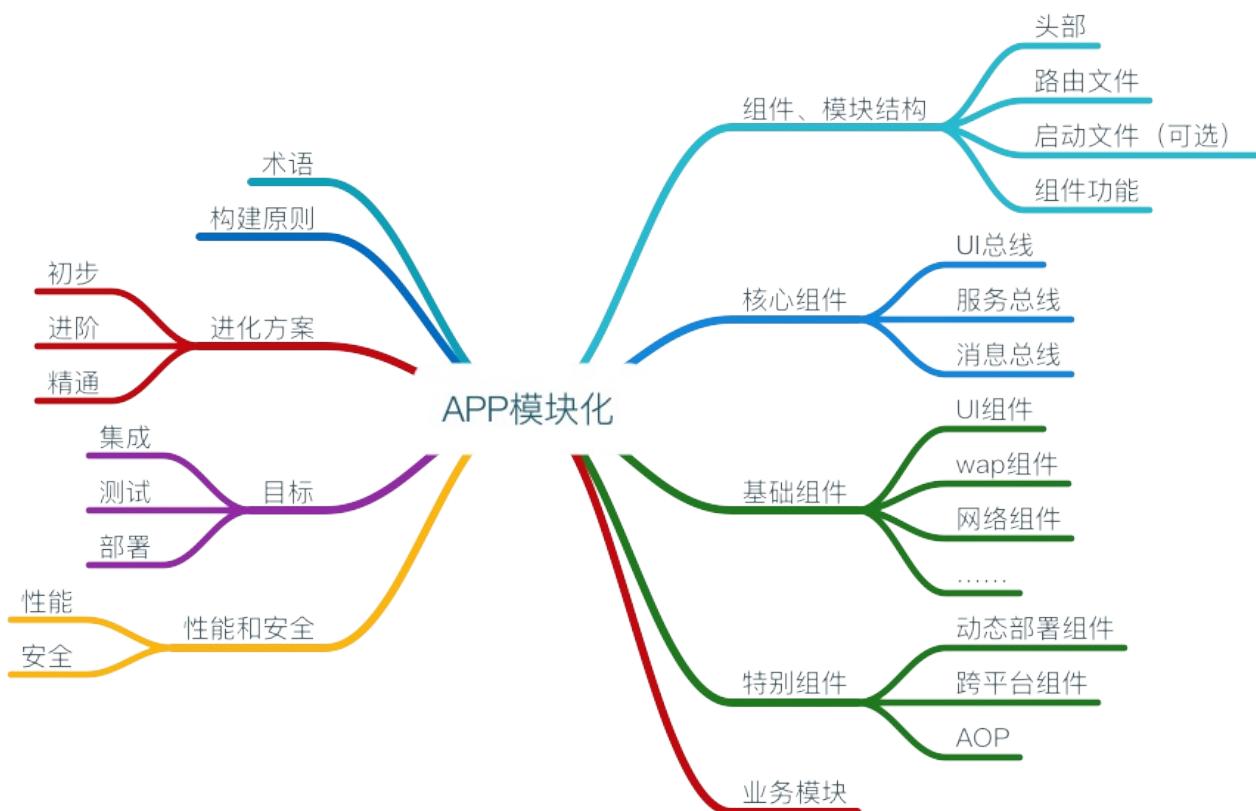
- (3) AOP组件
- 功能汇总
- iOS实现
- 五、业务模块

iOS模块化开发详情（初稿）

理论

- 本文涉及的大多数组件尚在开发中，该文档中只是粗略带过。
- 后续完善功能后，将以单独的组件开发文档体现出来。

图谱梗概



目录

理论

1. 术语

2. 构建原则
3. 进化方案
4. 目标

实践

1. 组件、模块结构
2. 核心组件
3. 基础组件
4. 特别组件
5. 业务模块

一、术语

1. 一切功能皆组件、一切业务皆模块

- 组件：功能块，于业务无关。即公共基础设施；组件之间无依赖，严格隔离，依靠路由互通。
- 模块：业务块，具体项目具体不同；通过路由策略，串接多个业务模块，形成一个完整的业务线。

二、构建原则

1. 代码整洁：工程(子工程)依据工程规范、文件依据文件规范、代码细节依据代码规范

TODO:文件级别代码片段完善

2. 单一功能，单一组件；单一业务，单一模块。

- 后续参考着重SOLID原则，提高抽离高度
- 组件模块分解粒度务必最细化，利于后期扩展
- 组件模块中无 Common、Core 集合（后期细化困难），无集中化的管理模块，即去中心化
- TODO:目前子工程有组件集合，后期进行细化提取时，注意依赖、规范问题

3. 关于细分和依赖问题（基于iOS考量）

- 主工程：一般通用三方库，多个组件、模块调用。使用cocoapods或carthage管理较为合适
- 组件细分：单一组件使用到的库手工引入该组件即可。
- 模块细分：基于组件或依赖于其它模块，依赖UI总线（路由策略）进行通信。若不得已使用组件具体功能，可从主工程进行分发。（详见子工程模板创建文档）

4. 业务基于组件

- 重点开发组件，拼接健壮基础架构层，然后打造业务模块
- 不同业务，依赖组件集或有差异。故在进行业务模块开发时，会逐步丰富组件和新增依赖业务模块
- 杜绝横向依赖，不跨层访问组件模块

三、进化方案

1. 初步：物理式隔离各功能块、各业务块。即组件化、模块化开发。

- 工程结构：主工程、子工程、Pods管理、子framework管理

2. 进阶：脚手架创建，模板生成，组件、模块生成，均实现脚本自动化。

- 工程结构：主工程、子工程、Pods管理、子framework管理
- 组件、模块进行打包编译，形成可被初步管理的库文件。如使用Cocapods管理的.a库文件。

1. 精通：组件、模块实现权限管理(GitLab私有化)、实现分支管理；编译集成均实现可视化

- 工程结构：主工程、Pods管理（含子工程）
- TODO:iBiu参考，iBiuL可视化配置，通过组件配置表：组件版本、责任人、格式是否二进制、依赖等，自动组装业务流，进而生成APP
- 以Pods搭建私有库、创建模板。如IOSMatrix模板
- Pods管理组件、模板：源码或以二进制包呈现，依据权限不同而定
- 可视化方案：包括进阶中的功能

四、目标

简化集成；重测试组件模块、轻测试集成封装；部署自动化

1. 集成

1. 标准化产出：单一组件、模块的源码库产出、二进制库文件产出
2. 依据三大总线组件进行集成组装

2. 测试

1. 优化测试，便于开发中的业务回归
2. 对组件、模块独立执行性能测试，进行定量优化

3. 部署

- 实现并行开发，独立调试，易于集成。
-

实践

提要：目前模块化开发处于初步阶段，一些基础组件暂由通用基础模块整合在一起，后续还需拆解细化

- 基础组件目前简单，未进行单独拆分
- 时间关系

一、组件、模块结构

1. 主要包括：

- 头部信息
- 路由文件
- 启动文件(可选)
- 组件、模块功能文件

2. 细节

1. 头部信息

| 信息 | 功能 | demo || :----: | :----: | :----: || 版本号 | 后续完善更新匹配策略，如不匹配
warning | v1.0 || 依赖信息 | 所依赖的组件、模块信息 | 组件一般无依赖 || 其它 | 谁如组
件负责人，建立时间、功能梗概信息 | 无 |

2. 路由文件

| 信息 | 功能 | demo || :----: | :----: | :----: || Target_XX文件 | 组件、模块路由文件 | 无 ||
CTMediator + category | 待完善 | 无 |

3. 启动文件

| 信息 | 功能 | demo || :----: | :----: | :----: || XXAppLauncher | 组件、模块初始化操作文件
| 无 |

4. 组件、模块功能文件

为组件、模块的主要功能，具体分析，此处略

二、核心组件

1. UI总线：路由策略组件

组件模块之间建立统一的交流规范，是打造模块化工程中最基础的组件设施。

功能汇总

1. 跨平台URI寻址方案
2. 降级方案：组件模块若无对应URI，自动以Web容器加载资源（TODO）
3. 分发策略：中心化或者去中心化选择

iOS实现

1. 实现组件：CTMediator
 - APP间跳转
 - APP内部页面跳转
 - 组件、模块间通信、跳转，组织功能实现
 - 中心化分发机制
2. 需要组件、模块单独开辟路由文件：调用和响应（input、output）
3. 功能待完善
 - 跨进程安全问题：需要确认

2. 服务总线：初始化启动策略

模块化内核管理组件。依据主程启动策略，完成工程中组件、模块集的初始化事务

功能汇总

1. 可跨进程，可同步/异步执行
2. 主要事务：容器启动、类加载、核心中间件初始化
3. 全局服务类，诸如：LBS、路由文件加载（与现有路由策略做对比分析，得出最优方案）、App更新服务、Push、埋点，timeAsync等

iOS实现

1. 实现组件：APPLauncher
2. 实现初始启动服务
 - 调度中心：LauncherMgr
 - 子类化IMXBaseLauncher，若有服务和其共同存在，则子类化同时实现service协议
3. 通过service服务模板，执行启动事务时，加入了诸如顺序、优先级、同异步等各方面考量
4. 需要组件、模块单独开辟启动文件
5. 具体实现后续会有单独的文档呈现
6. 注：因为 `+ (void)load` 方法在 `main` 函数之前执行，故只允许挂载组件名称，禁止其它耗时行为。
 - 此处若不进行处理，那么在 `appDidfinshiLoad` 时同样要处理初始化操作，所以都会在 app 启动之前耗时。
 - 故后续需要多次验证此类情景。

3. 消息总线

业务模块间消息通道

功能汇总

1. TODO :

iOS 实现

1. 依据 Android 的 `EventBus` 消息机制，制定 iOS 端消息模块。

| TODO :

三、基础组件

(1) UI 组件

功能汇总

1. 文件级别，制定并遵守文件规范
 - 规范各功能代码放置顺序。如 IDE 中定义代码片段实现
2. MVC 架构分层：`M`：数据层、`V`：展示层、`C`：业务层。

- 所有其它架构模式均是基于MVC的细化
- MVC架构为主，其它派生模式为辅

3. 基类Ctrl定义方案

- 尽量不使用继承，使用AOP等方案补充
- 规范约定

iOS实现

1. 代码片段定义

- code snippet主要是方便统一输出代码块，此处添加了文件级别的代码注释块，便于实现file规范。
- 具体说明见Obj-C规范，code snippet地址：[snippets_tools](#)

2. 架构模式：MVC架构作为基本架构，MVIP架构模式引导开发

- MVIP:增强版的MVP模式，后续会有相关文档

3. UI组件使用基类Ctrl

- 基类主要做通用的独立于项目的设置：如样式等
- 使用AOP+category替代基类效果：如数据收集、导航定义等
- 功能拆分为独立的扩展文件

4. 空页面和等待层

- TODO:

(2) Wap容器组件

功能汇总

1. 导航效果：手势、动画等

- iOS使用最新WKWebView控件，原生实现

2. JS交互功能：需要各端统一方案

3. 路由降级策略

- 即无具体页面路由，可视为wap内容：如错误页面展示

4. 缓存策略、离线机制

5. 安全：交于安全组件

- SPDY、DNS旁路解析等

6. 数据采集、性能监控：交于采集组件、性能组件执行

iOS 实现

1. JavascriptBridge三方库实现JS交互功能

待完善：注入的JS定义，需三端统一协商

(3) 网络组件

功能汇总

1. 离散型网络架构

- 每个网络请求单独处理

2. 图片加载

3. 数据解析

4. 安全：可交于单独的安全组件处理

- 数据安全校验：如签名
- 网络传输安全

5. 网络可达性实时检测等

iOS 实现

1. 目前使用现有组件

2. 后续会使用YTK网络组件库

待完善

(4) Debug 组件

功能汇总

1. 性能分析

2. UI检测

iOS 实现

1. FLEX实现：UI相关检测

2. CPU、内存检测

3. 内存分析(测试)

- 目前使用facebook开源库：FBMemoryProfiler
- 未完善

4. Log工具

- 依据不同环境，控制NSLog功能开关

5. 其它

(5) 数据收集组件 (**Data Collector**)

功能汇总

1. 导入方式：AOP横向切入，业务无感知
2. 具有高隔离性，可以支持多个平台无缝接入

iOS实现

1. TODO :

(6) 持久化组件 (**Data Persistence**)

功能汇总

1. 存储方案实现以及选择
2. 组件架构
3. 性能考量

iOS实现

1. TODO :

(7) 安全组件

功能汇总

1. 网络安全
2. Wap安全

iOS实现

1. TODO :

(8) 扫描组件

1. TODO :

(9) IM组件

1. TODO :

(10) push组件

1. TODO :

(11) 分享组件

1. TODO :

(12) 设备信息组件

1. TODO :

(13) 国际化组件

1. TODO :

(14) 用户体系组件

1. TODO :

(15) AB test组件

1. TODO :

(16) Style Kit组件

功能汇总

1. 字体设置

- 字体适配策略 TODO
- 具体项目的字体配置可以在bizCommon中做这部分功能（一套完整的字体体系应该可以满足APP的要求）
- 目前涵盖常见字体设置
- 其它待完善

2. iconfont功能

3. **iconfont**属于自定义字体。目前要求**iconfont**字体集个数不限

4. 目前使用和工程名相同的**iconfont**字体集，如 `projName.ttf`

i. 使用 (iOS)

- 查找ttf文件，找到具体icon对应的Unicode值，如 `\ue600`。
 - UI控件调整fontsize，使之具有合适赋值iconfont字体的大小
 - UI控件赋值iconfont文本即可。

ii. 导入文件

- 进入网站[iconfont](#),搜索需要的icons，整合为同一项目中，然后下载至本地。
- 下载文件中含有X.ttf的文件，置入主工程，即完成导入的所有icon字体图标（属于资源文件导入范畴），Target设置。
- 在工程项目的info.plist文件中，设置KV值：

```
<key>UIAppFonts</key>
<array>
<string>****.ttf</string>
</array>
```

5. 色值功能

- UIColor类别：实现RGB十六进制定义

6. 其它

(17) 扩展组件

- 主要涵盖**UI**、功能对象扩展功能(如**Array,Map**安全写入等)

功能汇总

1. 安全写入问题

iOS 实现

1. 该部分功能未细化
2. 暂时放置在baseModule组件中
3. TODO :

iOS 实现

1. TODO :

四、特别组件

(1) 动态部署组件

功能汇总

1. 业务不经过审核，即可从server端动态下发

iOS 实现

1. 热更新方案：apple暂时搁浅
2. 可以着重考虑使用跨平台方案替代之

(2) 跨平台组件（weex组件或RN组件）

tag : 特别组件

功能汇总

1. 一次开发，多端共用

iOS 实现

1. weex、RN组件使用
 - TODO : 将其进行组件化

(3) AOP组件

功能汇总

1. TODO :

iOS 实现

1. 简单实现：Aspect三方库

五、业务模块

略，详见工程开发相关文档

Copyright © iOS 开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

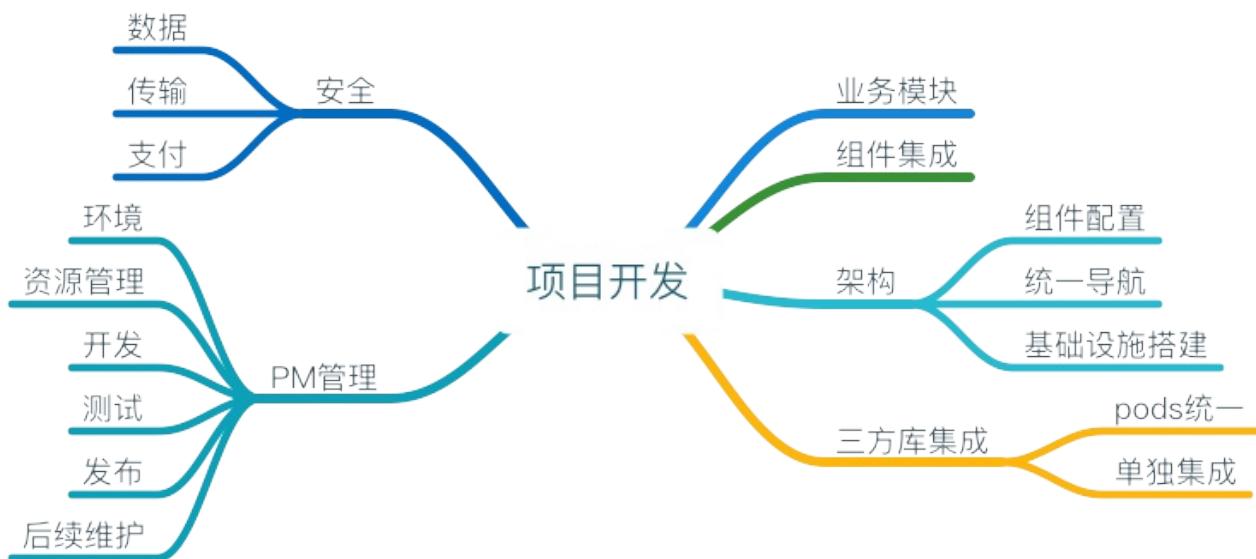
- 项目部署（初稿大纲）
 - App架构
 - 项目图谱
 - 目录
 - 业务模块
 - 模块细分（粒度较大，未完善）
 - 组件集成
 - 架构
 - 安全
 - 三方库
 - PM项目管理
 - 1. 环境
 - 2. 资源管理
 - 3. 开发
 - 4. 测试
 - 5. 发布
 - 6. 后续维护

项目部署（初稿大纲）

App架构



项目图谱



目录

1. 业务模块
2. 组件集成
3. 架构
4. 安全
5. 三方库
6. PM项目管理

业务模块

- 确立项目需求，拆分模块单元。该过程贯穿项目开发始终。

模块细分（粒度较大，未完善）

1. 导航模块：商品导航（Home、category、stores等）
2. 订单模块：商品浏览-下单-付款-等
3. 用户模块：配合用户体系组件打造用户模块，包括profile等

涵盖一系列用户登录注册流程

4. 基于模块开发中，有通用UI或者通用其它功能，也会逐步产出新的组件或者通用业务模块。

组件集成

- 依据业务模块需求，集成组件。在项目开发始终，增减组件以配合业务模块的完善。
- 安全组件配合路由、wap组件外，用户体系也需要该组件，但具体的安全配置，需要单独的安全组件完成。
- 其它TODO：

架构

- 主要涵盖组件配置，项目统一导航，以及一系列基础设置搭建
- 启动：组件、模块启动文件配置
 - 项目初始化需求
- 导航搭建
 - 如基于导航，基于tabbar建立项目基础框架
- 统一文件配置：一般有相关组件完善约定
 - 基于组件，构建项目相关的统一色值文件
 - 基于组件，构建项目相关的统一字体文件
 - 基于组件，构建项目相关的统一基类文件
 - 基于组件，构建项目相关的统一全局宏文件
 - 基于组件，构建项目相关的统一全局持久化文件
 - 其它

安全

- 项目设置安全模块（可选），再配合安全组件，完善具体的安全要求
- 传输安全
- 数据安全
- 支付安全

三方库

PM项目管理

1. 环境

1. IDE：Mac OS、Xcode8.x
2. 版本支持：iOS8及以上
3. 配置：TODO：

2. 资源管理

1. 代码管理：SVN（多用Git）
2. 文档管理：SVN

其中项目checklist根据具体项目不同而不同，故在代码中创建doc文件夹进行管理

3. 开发

1. 开发协同
 - 注意开发过程中，随时变动的需求
2. 开发完成
3. 开发结束
 - CodeReview

4. 测试

- 具体参见测试相关文档
- 测试
 - Xcode性能测试、instruments等方式，建议贯穿整个项目开发始终
- 集成测试
- 混合测试
- 单元测试
- 回归测试
- 其它

5. 发布

1. 项目checklist执行
 - 位于项目doc文档中
2. 内灰
 - 内部测试，定期CI集成测试

3. 外灰

- TestFlight分发（缺点：邮件邀请，且人数有限制）
- 蒲公英
- FIR

4. 发布AppStore

6. 后续维护

1. 重复文件梳理、统一
2. 业务模块梳理
3. 组件、模块重构：app包瘦身等

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

组件实现细节

组件实现请点击进入[开发&进阶--模块化方案](#)章节

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

- [pragma clang diagnostic ignored "```"和#pragma clang diagnostic pop```包裹起来。](#)
 - [TODO :](#)
 - [参考链接 :](#)

提要

前面有几篇文章已经详细介绍了iOS模块化开发。

为了实现**iOS**模块化开发，已经完善了以下必不可少的前置条件（试行）：

1. 子工程和主工程的模板定制，也即脚手架的定义。

比如对文件组织形式进行了统一的规范；添加pch等一系列通用文件，避免了后续子工程和主工程创建中一些不必要的重复工作量。

- [具体详情可参考文章：](#)
- [iOS子工程模板搭建](#)。（其中也包含了主工程的配置细节）

2. 实现单个组件、模块的之间隔离：利用单一功能原则，每一个功能都独立于其它的功能存在，业务模块亦然。

- 组件即功能组件；模块即业务模块。
- 在[iOS模块化开发详情](#)中不仅详细定义了组件、模块，也对如何部署做了严格的规范。

3. 代码规范：在文件维度和代码维度上做了代码规范。

- 在文件内部的代码组织上做了规范。
- 在遵循[Apple](#)代码规范的前提下，做了部分代码优化的总结。
- 详情见[Obj-C代码规范](#)。关于Swift代码规范，目前正在整理中。

进阶方案

在[iOS模块化开发详情](#)中有提到进行模块化开发的三个进阶方案：

1. 初步：物理式隔离各功能块、各业务块。即组件化、模块化开发。
2. 进阶：脚手架创建，模板集成，组件、模块集成，均实现脚本自动化。
3. 精通：组件、模块实现权限管理(Github、GitLab)、实现分支管理；编译集成均实现可视化。

目前已实现初步方案。但也有明显的缺点，如子工程分发执行时，需要更多的重新编译操作，不仅耗时，而且项目集成也不方便。

进阶2实现脚本自动化虽然可提高效率，但相对初步方案来讲，不是目前最大的问题。

所以利用**Github**、**GitLab**实现子工程的集成、权限管理，对于提高开发效率，解决初步方案中存在的弊端，个人感觉其优先级比进阶2要高的。

cocoaPods管理子工程功能组件

Obj-C项目开发中，我们引入和使用三方库，使用技术最多的便是cocoaPods集中式管理。

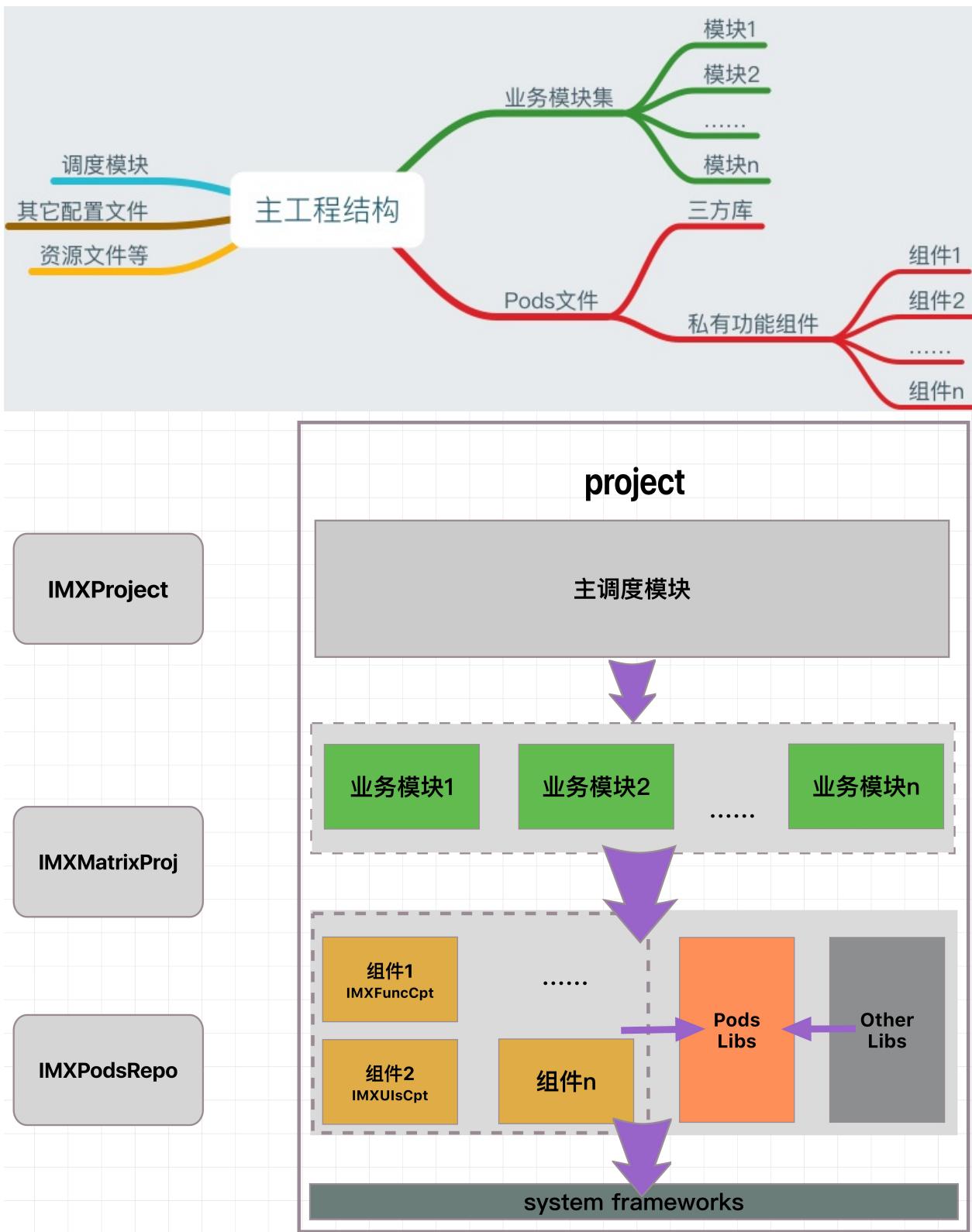
关于Swift项目开发，可以考虑Carthage技术管理子工程，比cocoaPods更为简单些，后续追加。

使用Pods我们不仅可以开发和使用开源的三方库。而且还支持开发和使用私有库，并且通过配置Podspec文件，还能解决私有库开发过程中复杂的依赖问题以及其它预配置操作。

相较于方案1，我们不用自己重复编译子工程和配置依赖等问题。

1. 我们只要创建私有子工程，并配置好podsepc文件，像使用其它开源三方库使用我们自己的子工程就可以了。
2. 本文子工程管理只考虑功能组件。关于业务模块方面，使用私有GitLb库配置，或者使用Git子模块方案也可以。但考虑目前项目开发人员较少，并且没有跨地域开发等问题，所以业务模块直接嵌入在子工程中即可。

1. 图谱预览：



- 针对图谱2中所示的资源，均放置于Github中，列出如下：

1. **IMXProject:** Object-C 主工程模板项目，基于其上派生实用项目。（修改项目名即可）

- git clone之后，需要重置git关联、修改项目名，完善主调度模块等步骤，便完成了一个新的空项目project的初始化。

2. **IMXMatrixProj**:Object-C子工程脚手架。基于其上派生功能组件、业务模块。

- 通过内置脚本生成子工程、修改子工程的若干配置，即可完成初始化配置。
- 配置组件，完善功能后，则交由cocoaPods来管理，这也是本文主要讲述的内容。
- 配置模块，完善功能后，使用Git submodule子模块来管理较为合适，后续完善。

3. **IMXPodsRepo**:私有Repo版本库。

- CocoaPods配置文件：主要记录私有Pods组件。
- 若后续进行Swift项目开发，则使用Carthage或swift package manager技术，该部分会有不同。

4. **IMXFuncCpt**:基础功能组件，即一系列通用功能的组合。

5. **IMXUIsCpt**:控件类组件集合，即包含UIs工具、以及UIs控件。

6. DebugModule组件。（图谱未提及）

7. 待添加.....

2. Pods子工程创建

创建Pods管理的子工程，本文以IMXUIsCpt组件为例。我们需要完成以下功能：

1. 完成Repo版本库创建。
2. 以子工程模板组件开发IMXUIsCpt，并提交Github。
3. Repo版本库和子工程库关联。
4. 附加操作（可选）：Pods子模块、Framework生成。
5. 使用案例展示

1. 完成Repo版本库创建：

Pods私有库开发只要创建一份Repo版本库即可。如之前创建了Repo库，此步骤可以忽略。

1. 创建CocoaPods repo 私有版本仓库：

- 在Github上创建一公开仓库，按步骤生成即可。
- 举例：<https://github.com/PanZhow/IMXPodsRepo.git>

2. 将私有版本库添加至本地Repo：

- pod repo add IMXRepo git@github.com:PanZhow/IMXPodsRepo.git
- 检测Finder中repo库，是否新增repo记录成功：~/.cocoapods/repos，会看到IMXRepo文件夹，即添加成功。

2. 以子工程模板组件开发IMXUIsCpt例，并提交Github。

1. 创建私有代码库：

- 在Github、GitLab或者私有Git server上创建一仓库：IMXUIsCpt，按步骤生成即可。（<https://github.com/PanZhou/IMXFuncCpt.git>）
- git clone置本地后，通过github上子工程模板 IMXMatrixProj，创建IMXUIsCpt子工程项目。并copy置该仓库中。

2. 处理子工程相关配置：具体参考[iOS子工程模板搭建](#)。

3. pods文件.swift-version生成和配置

- i. touch .swift-version
- ii. open .swift-version
- iii. 输入：echo "3.0" > .swift-version

1. pods文件podspec生成和配置：

- cd 项目目录，执行：pod spec create IMXFuncCptPodspec
- .podspec详细配置：见[官网](#)或附录1，并以此修改此文件。
- 检测：pod lib lint；pod lib lint --private；pod lib lint --allow-warnings；pod spec lint（推荐：本地和远程验证）
- 提示 IMXUIsCpt passed validation 时，表明通过。

2. 添加功能代码，资源文件等。并提交Github远程。

3. Repo版本库和子工程库关联：

1. 代码库push至远程git库：注意tag和版本一致。
2. 描述文件podspec同步至Repo版本库中：

- pod repo push IMXRepo IMXFuncCpt.podspec
- Repo版本库push至远程仓库。（pods自动上传）

3. 验证：

在终端键入：pod search IMXFuncCpt，搜索之。

4. Pods子模块、Framework模块生成：

1. Pods子模块创建原因：

- i. 由于很多功能组件体积较小，功能较简单，故可以将多个组件组织在一起，形成一个大的功能组件集合。（如IMXUIsCpt中的IMXStyleKit）
- ii. 由于功能组件尚在开发中，还不成熟，故可以临时先放置在一个功能组件集合中。后续开发成熟，可以抽离出去，形成一个独立的pods私有库。（如IMXUIsCpt中的IMXTabBarKit）
- iii. 步骤简述：模块化分组->配置podspec，添加子模块(public&private TARGET配置)->继续添加子模块.....
- iv. 检测可用性-> 提交代码至远程仓库-> 同步Repo库

2. Framework子模块创建原因：

- i. 某些组件不打算公开源码，仅以h头文件+framework呈现。
- ii. 业务组件使用该模式管理，仅以h头文件+framework呈现。
- iii. 步骤：见Pods私有库framework创建和使用。
- iv. 检测可用性-> 提交代码至远程仓库-> 同步Repo库

5. 使用案例展示

1. 私有pods库使用，需要在Podfile中添加私有repo库：

```
source 'https://github.com/PanZhow/IMXPodsRepo.git'
```

2. 使用。如其它pods公共三方库资源。

附录：

1. 附录1：podspec部分字段诠释

```

Pod::Spec.new do |s|
  s.name      = "IMXUIIsCpt" # 项目名称
  s.version   = "0.0.1"        # 版本号 与 你仓库的 标签号 对应
  s.license   = "MIT"         # 开源证书
  s.summary   = "私人pod代码" # 项目简介

  s.homepage  = "https://github.com/PanZhow/IMXPodsRepo.git" # 仓库的主页
  s.source    = { :git => "https://github.com/PanZhow/IMXPodsRepo.git", :tag => "#{s.version}" }#你的仓库地址，不能用SSH地址
  s.source_files = "MyAdditions/*..{h,m}" # 你代码的位置， BYPhoneNumTF/*..{h,m} 表示 BYP
  honeNumTF 文件夹下所有的.h和.m文件
  s.requires_arc = true # 是否启用ARC
  s.platform   = :ios, "7.0" #平台及支持的最低版本
  # s.frameworks = "UIKit", "Foundation" #支持的框架
  # s.dependency = "AFNetworking" # 依赖库

  # User
  s.author     = { "BY" => "2331838272@qq.com" } # 作者信息
  s.social_media_url = "http://****" # 个人主页

end

```

2. 附录2：Pods子模块创建创建

```

# Launcher Cpt
s.subspec 'Launcher' do |lcr|
  lcr.source_files  = 'IMXFuncCpt/Libs/2nd/Launcher/*..{h,m}'
  lcr.public_header_files = [
    'IMXFuncCpt/Libs/2nd/Launcher/*..{h}'
  ]
end

```

3. 附录3：Pods子模块创建时注意事项

当私有库中引用了私有库依赖的情况，那么在验证和push repo时均要如下指明source，否则pod会默认从官方repo查询：

- i. pod lib lint --

sources='https://github.com/PanZhow/IMXPodsRepo.git,https://github.com/Co

coaPods/Specs'
- ii. pod repo push 本地repo名 podspec名 --sources='私有仓库repo地

址,https://github.com/CocoaPods/Specs'

1. 附录4：关于repo的相关命令

- i. repo展示：pod repo list
- ii. repo删除：pod repo remove **

2. 附录5：出现警告问题的处理

若出现警告问题，最好不要使用 `pod lib lint --allow-warnings` 忽略警告，解决方案：

- i. 发生警告的代码片段以````#pragma clang diagnostic push`

**pragma clang diagnostic ignored
"***"和#pragma clang diagnostic
pop```包裹起来。**

- ii. 在`podspec`中将该模块设置为 `s.compiler_flags = '***'` 即可。

TODO：

1. Pods库多个子模块中含有尚未成熟的子项目，后续需要单独抽离，并作为独立的pods库来管理，甚至开源为公开pods库。

参考链接：

1. [cocoapods系列教程---模块化设计](#)
2. [Cocoapods使用私有库中遇到的坑](#)
3. [pods私有仓库创建](#)
4. [pods公共仓库创建方法](#)

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

目标

本文是文章 [CocoaPods管理子工程](#) 的续篇，主要目的是将Pods私有库打造成framework静态库来使用。

实现方案

1. 步骤回顾：

1. 创建私有 **pods repos** 版本库。
2. 存放**framework**源码的项目。
 1. 源码项目。（不强制要求是模板项目）
 2. **podpec**文件定义：

- 版本定义：注意一定要与tag保持一致。
- source code位置
- public header file 公开文件。见附录1。
- 调用的依赖的 framework 等。

3. 关联之。

- `pod repo push IMXRepo IMXFuncCpt.podspec`

2. 资源转换为**framework**：

1. 安装[CocoaPods Packager](#)，并按照提示安装。
2. 使用:在资源文件目录下执行：`pod package **.podspec`
3. 可以看到在当前目录下生成了**framework**文件夹：

- 更改主**podspec**文件：将生成的**podsepc**部分信息替换，见附录2
- commit 文件夹,注意tag更新。
- `pod update`即可完成使用**framework**的更新。

附录

1. 附录1：

```
s.source_files = 'IMXFuncCpt/Libs/2nd/Launcher/*.{h,m}'  
# s.exclude_files = "Classes/Exclude"  
  
s.public_header_files = [  
    'IMXFuncCpt/Libs/2nd/Launcher/*.{h}'  
]
```

1. 附录2：

```
//注释或删除掉附录1中的内容  
s.preserve_paths = 'IMXFuncCpt-1.0.0/ios/IMXFuncCpt.framework'  
s.ios.deployment_target = '8.0'  
s.ios.vendored_framework = 'IMXFuncCpt-1.0.0/ios/IMXFuncCpt.framework'
```

参考链接：

1. [CocoaPods Framework创建](#)
2. [Pods官网](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间：2018-03-08 10:26:48

架构

- [MVC](#)
- [MVP](#)
- [MVVM](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- **MVP详解**
 - 前情提要
 - 结构
 - 整体俯瞰
 - 模块详解
 - 功能总结
 - 实例代码：
 - 参考

MVP详解

- MVP模式增强版请见文档：[MVIP模式](#)。
 - MVIP模式基于MVP模式的基础上，增加了Interactor层，细化了Presenter和View之间的联动，即细化Data和UI之间的绑定处理。
 - 本文理论部分和MVIP模式一致，可以作为参考。

前情提要

目前iOS架构级别的设计模式，常见于文章的大致有：MVC、MVCS、MVP、MVVM，VIPER这几种。但常用主要是MVC、MVP和MVVM。

1. 关于MVC，Xcode默认创建一个界面模块时，即为该模式。躲都躲不掉的亲密模式，所以即便要使用MVP或者MVVM模式的项目，肯定是以MVC作为入口的。
2. 其次是MVVM，目前不使用它主要固于3个原因：>

其一：它主要是将MVC中C的代码全量到VM中了，并没有真正做到解耦和[单一功能原则](#)。

其二：VM中双向绑定的使用技术，如RC、RxSwift是需要学习成本的；对于团队协作来讲亦然。

其三：Uh，还没想出来第三点。

3. 主角登场：MVP。

详见下文，主要从结构构成和功能两大块来讲述。

结构

整体俯瞰

1. 入口：在Xcode新建一个MVC界面，在**ViewController中添加MVP架构的模块。

前面讲到MVC是Xcode固有的架构，所有入口没办法只能以此为准。此处的 ViewController负责View Container周期、其它V（MVP）的接入，并监听与MVP相关的事件。

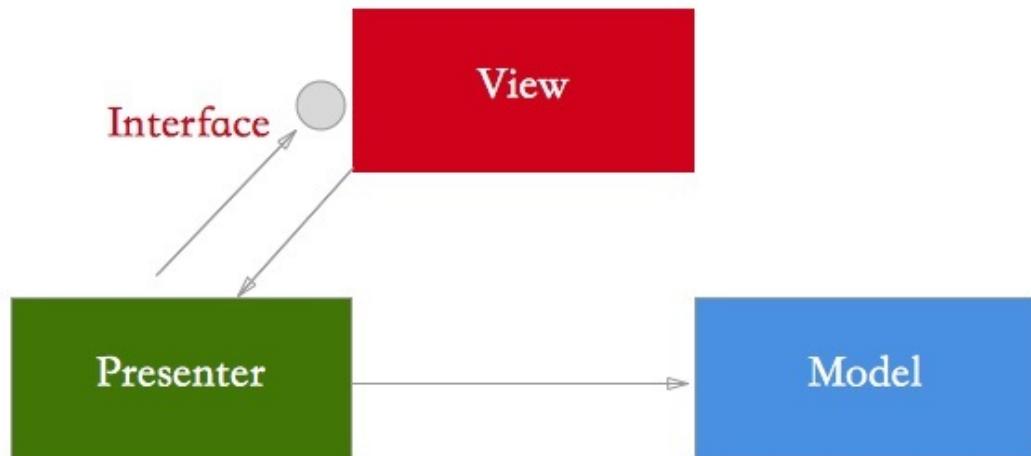
2. 以View为中心：MVP中，以View为交互中心，Model和Presenter辅之。

- i. 以View为中心，将其添加于MVC中。
- ii. 初始化所属Presenter，获取相关Model
- iii. Presenter间接更新View

类似于NativeScript，加载XML(View)时，会查询同名的js和css文件(逻辑和布局文件)，然后加载之。

个人认为以View为中心，结构明朗，另外尽量在Presenter中不涉及View模块。

3. 图示（注意图示中箭头对应上述黑体 所属 、 相关 、 间接 ）



模块详解

一、Model层

1. 由Entity实体、Service网络请求、cache缓存机制组成

- i. Entity: 即单独的数据模型，负责模块化网络、缓存中的数据。
 - ii. Service: 可选。通过网络层获取数据的服务。（具体网络请求由底层模块负责，此处只需实现请求响应即可）
 - iii. Cache: 可选。完善的缓存机制，如memory、File System、FMDB等。往往和网络请求配合使用。（具体的缓存机制由底层缓存模块负责，此处只需实现功能即可）
2. Code组织：Entity由Model类组织，而网络请求和缓存可以使用对应的category实现。
(缓存可以在网络请求失败时使用，请求成功时保存)

```
//model示例
struct DemoModel {
let firstName: String
let lastName: String
let email: String
let age: String
}
```

```
extension DemoModel {
static func requestUsers(block:@escaping ([DemoModel]?, _ error: Error?) -> Void)
{
//网络请求实现+缓存运用
//回调
block(models, nil)
}
}
```

二、View层

1. 由View和ViewController组成。
 - i. View: 简单的View控件；ViewController：视图以及视图逻辑，所以都属于View层的一部分
 - ii. View层直接持有一个Presenter层负责业务逻辑
2. 负责用户Action响应和Data展示

- i. Action响应：直接调用所属Presenter，执行对应业务逻辑。如数据请求、按钮事件等。
- ii. Data展示：在Presenter层含有View层的间接引用，获取Model后，会调用View层，执行data渲染。而最为重要的：执行具体View层渲染的，是View自身。

关于View层的render，会涉及到Model元素的一一对应问题，这和MVP模式中View层和Model层不直接通讯的理念有点出入，但没办法。而且两者并没有实质的逻辑联动，只是视图展示。

在Swift中可以做View层的扩展，将此视图逻辑与view隔离开；OC中使用category实现renderView，作为View层和Model层一次暧昧的接触。

附：Data展示这块，其实有两种方案，一种如上所述。另一种则是在Presenter层实施具体的渲染工作：优点是View层减轻了负重，业务逻辑和View&Model绑定都放在了Presenter层；缺点是Presenter成了ViewModel，变得臃肿了，而且前面也提到了尽量不要在Presenter层导入UIKit模块。

3. code组织：即系统或者自定义View；UIViewController子类

- i. 继承 PZViewProtocol 协议，实现View渲染的协议或者block
- ii. 定义所属Presenter，负责业务逻辑
- iii. 部分示例：

```
class ViewController: UIViewController, PZViewProtocol{
    var demoPresenter: DemoPresenter?
    //实例化Presenter
}
```

```
``` //通过delegate实现渲染；也可使用回调方案 extension ViewController{ func
renderView(model: Any) -> Void{
```

```
let retModel = model as! Array<Any>
demoPresenter?.demoModels = retModel;
tableView?.reloadData()
}
```

```
}
```

## 三、Presenter层

### 1. 弱引用或者间接引用View层，直接引用Model层（service、Cache或者Model）

- i. View层传输Action事件，P予以响应
- ii. P通过Model层得到的数据，通知View层，渲染之。

2. Code组织：P主要做的事情有2件，一个是承 view 启 Model；另一个则是处理业务逻辑。

```
//业务逻辑
func requestModels() -> Void {
 viewDelegate?.loading!()
 DemoModel.requestUsers { (response,error) in
 if let ret = response{
 self.viewDelegate?.finishLoading!()
 self.viewDelegate?.renderView!(model: ret)
 }else{
 self.viewDelegate?.endLoading!(error: error!)
 }
 }
}
```

## 功能总结

行文至此，下面总结下我所理解的MVP：

### 1. Model层：

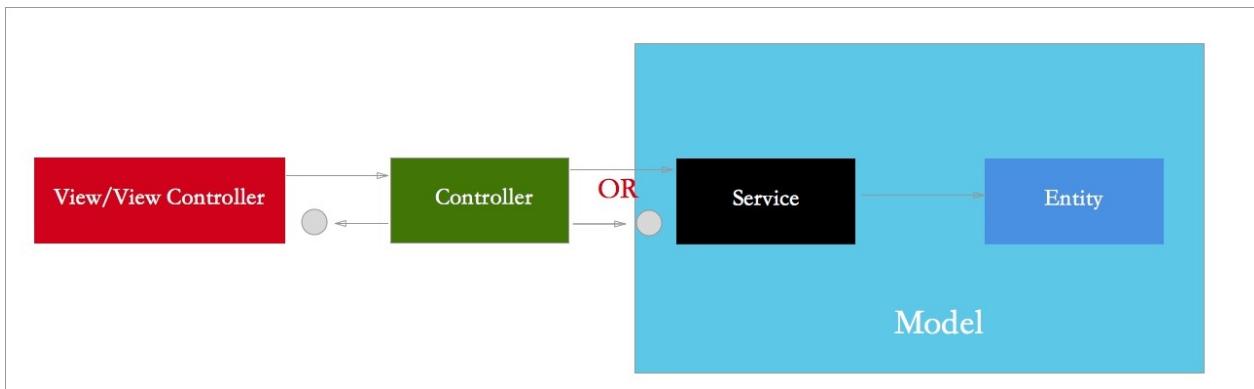
- i. 由Service、Cache与Entity组成。Service&Cache为P层提供网络与本地数据服务，即处理网络请求、数据库、文件等操作。Entity为实体类，负责定义数据的模型。
- ii. Presenter直接引用Model层

### 1. View层：

- i. 包括View、ViewController（含视图逻辑）两个模块。
- ii. View层是MVP的中心，P和M均为其服务。
- iii. View层直接引用所属Presenter层，执行Action和渲染逻辑。View将界面的响应处理移交给Presenter，而Presenter调用Model进行处理，最后Presenter将Model处理完毕的数据通过Interface的形式递交给View做相应的改变。
- iv. View层不与Model层交互，但在Data展示时需要做一下引用，非逻辑处理。  
(可以使用delegate形式，将Model绑定隔离开；也可以使用block形式，但隔离性略有降低)

### 2. Presenter层：

- i. 负责业务逻辑，关联View层和Model层（可根据项目特点选择直接/接口引用）
- ii. P间接引用View的方式，通知View层的变更：可以是delegate，也可以是block形式



## 实例代码：

代码请移步[github](#)

附：诚如课本中的课后习题一样，其复杂度和示例完全不是一个量级。所以demo只是简单的讲解，后续随着开发使用的深入，逐步丰富使用方案。

## 参考

1. <https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52#.h8i2v6evy>
2. <http://catchzeng.com/2016/01/26/iOS%E7%94%A8%E8%A2%AB%E8%AF%AF%E8%A7%A3%E7%9A%84MVC%E9%87%8D%E6%9E%84%E4%BB%A3%E7%A0%81/>
3. <https://martinfowler.com/eaaDev/uiArchs.html>

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- MVIP模式
  - 0、提要
- 一、功能详解
  - 1.0 流程图解
  - 1.1、View
    - 1. Render View
    - 2. Order 外放
  - 1.2、Interactor
    - Order : action + render data
  - 1.3、Presenter
    - 响应Interactor
  - 1.4、Model
- 二、功能实现
  - 2.0 类图
  - 2.1、View
  - 2.2、Interactor
  - 2.3、Presenter
  - 2.4、Model

## MVIP模式

### 0、提要

- MVIP模式属于MVP的一种尝试。即在View和Presenter之间添加了一个交互层（Interactor），主要来处理它们之间的绑定操作。
- 源码Demo请移步:<https://github.com/PanZhou/MVIP>
- 理论性的东西请参考上一篇文章MVP模式

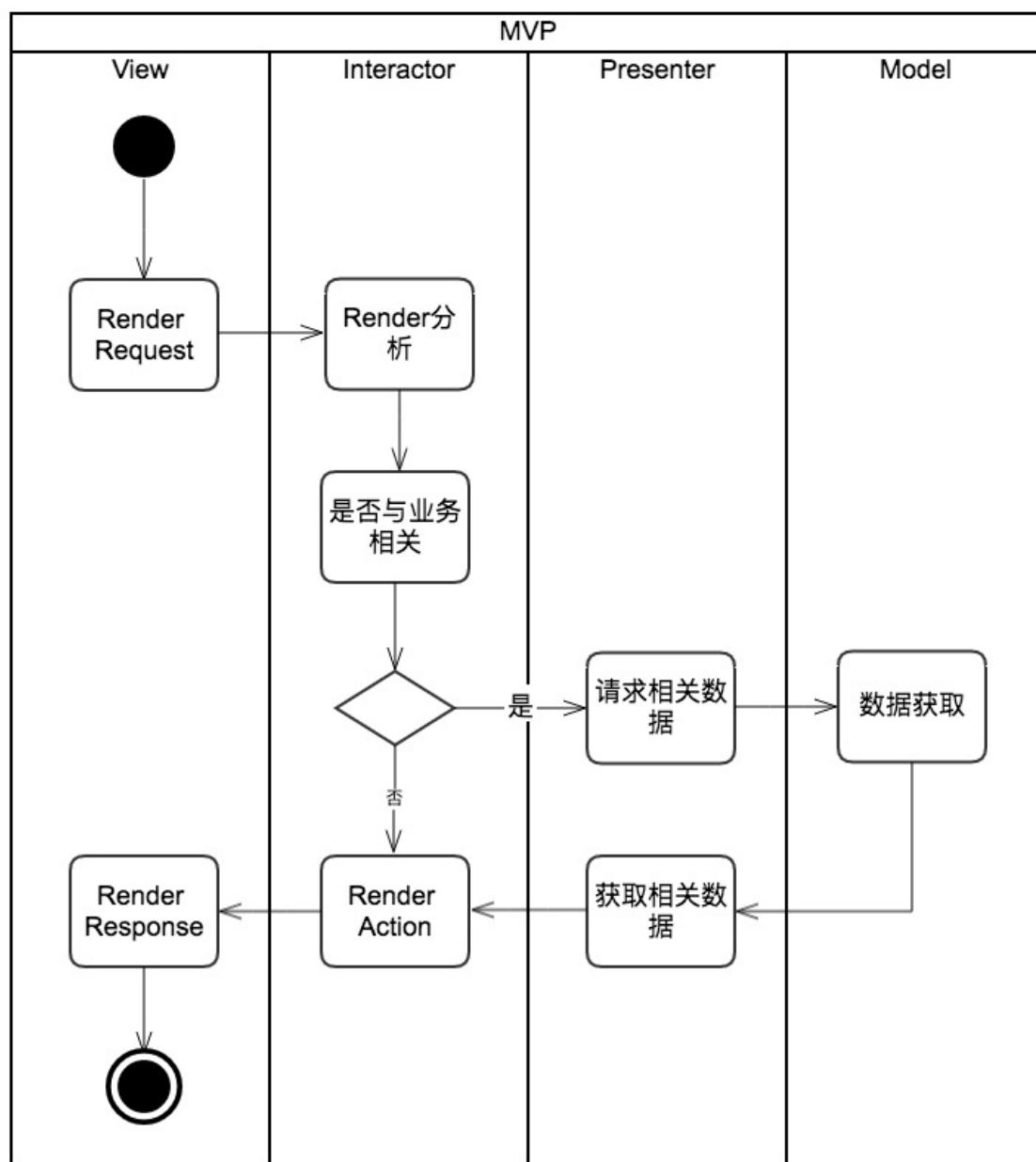
MVP详解中的Demo可以忽略，与本文的实现有改动。即增加了中间层Interactor。

- TODO：关于MVP、MVIP，目前问题还是在View和Model之间的数据绑定上，即双向绑定问题未完全解决，未能做最简化联动处理。
  - 后续需要改进的也在于此处。

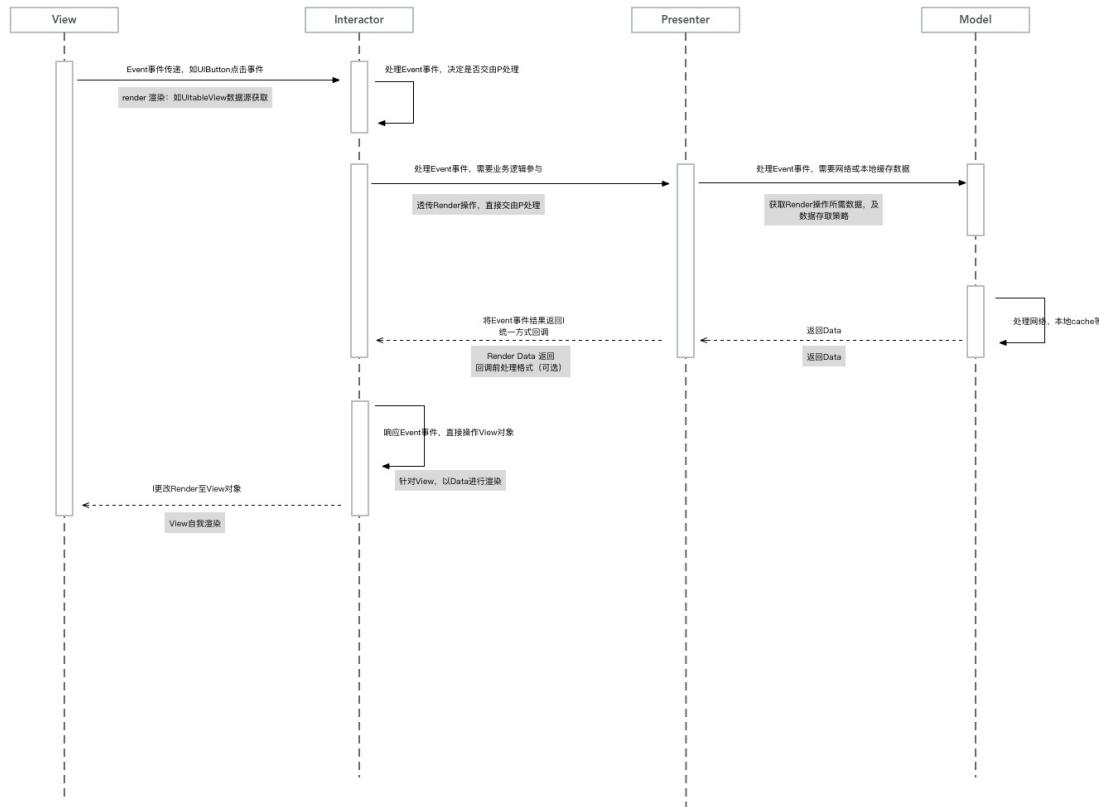
### 一、功能详解

#### 1.0 流程图解

## 1. 活动图



## 2. 时序流程图



## 1.1、View

### 1. Render View

具体视图 Render：如UIButton、UITextView、UITableView（含协议实现）添加

### 2. Order 外放

Action、render data 具体实现交于 Interactor：如UIButton Action、UITableView 数据获取等。

## 1.2、Interactor

### Order : action + render data

1. action：如UIButton的Action具体实现、UITableView的点击事件
2. render data：与Data+View相关，如UITextView的占位符、UITableView的数据回执

## 1.3、Presenter

## 响应 Interactor

1. action : 需要集合业务逻辑的功能
2. ret data : 具体的数据请求工作

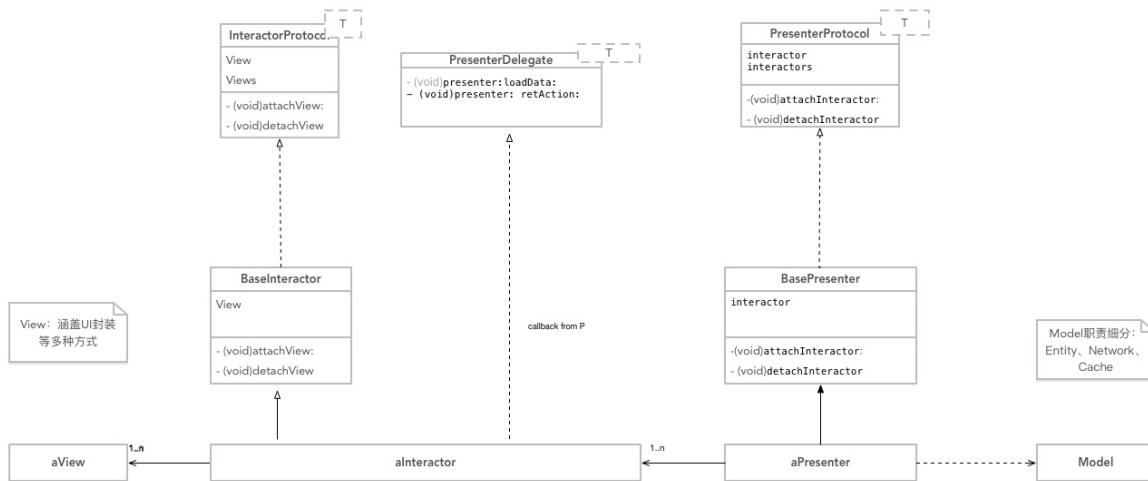
数据来源 : Model层

## 1.4、Model

- 涵盖 Entity、Net、Cache 等功能

## 二、功能实现

### 2.0 类图



## 2.1、View

1. 创建 Interactor，并执行初始化
2. Interactor挂载View，即弱引用View
3. Action、Render data 行为注入。

具体行为实现在 Interactor 中

## 2.2、Interactor

1. 初始化：创建Presenter，并初始化之
2. Presenter挂载Interactor，即弱引用Interactor。
3. Presenter实现代理
4. action、render data实现：响应View

是否需要业务逻辑部分协助，是具体情况而定。如render data一般获取需要Presenter协助。

5. Presenter通过业务处理，将结果回执。
  - i. 通过委托方式返回。
  - ii. 回执结果，可能进一步展示在View上，此时可能会调动View。

## 2.3、Presenter

1. 初始化工作。
2. 业务逻辑执行，响应自Interactor的事件。
3. 回执结果（代理方式、Block）

## 2.4、Model

- 主要分为Entity、Net文件、Cache文件。
  - Entity: 可能会继承某些json解析的基类，或继承其协议
  - Net文件：基于不同的网络库，可能形式不同
  - Cache文件：若网络库含有cache功能，可能和Net文件一体；也可以单独设置cache策略

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

## 架构

- [MVC](#)
- [MVP](#)
- [MVVM](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- QA
- 

- [QA](#)

# QA

- [性能优化](#)
- [测试](#)
- [其它](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 性能优化

## 性能优化

- 启动优化
- 耗电优化
- 内存优化
- 网络优化
- 优化工具

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 测试

## 测试

- 单测
- 自动化测试
- 机型测试（分布调研）
- 其它（AB测试等）

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 其它

## 其它

- [CodeReview](#)
- [安全扫描](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

- [CodeReview](#)

# CodeReview

- [代码规范度](#)
- [安全等各方面](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

- 安全扫描

## 安全扫描

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 代码管理

## 代码管理

- IDE
- 包管理
- 版本控制

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- [IDE](#)

# IDE

- Xcode

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 包管理

## 包管理

- CocoaPods
- macOS之Carthage使用
- Swift Package Manager

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 二、Cartfile文件构成
  - 1. Cartfile
    - origin :
    - version要求
    - 示例
  - 2. Cartfile.private
  - 3. Cartfile.resolved
  - 4. Carthage/Build
  - 5. arthage/Checkouts
  - 6. ~/Library/Caches/org.carthage.CarthageKit
  - 7. Binary Project Specification

## 0、说明

- Carthage之于CocoaPods，在于去中心化，更多主动管理三方库以及二方库。
- 对于后续模块化开发，十分方便。（即将开发的独立模块置于自己的服务上）
- 本文是 Carthage GitHub文档翻译 + 后续实际开发经验 的整合，旨在方便理解和使用。

## 参考链接

- Github地址：<https://github.com/Carthage/Carthage>
- 资源Trending索引：<https://github.com/trending?l=swift>

## 一、作业流程：

1. 创建Cartfile：项目中所有到的所有frameworks均罗列于此。
2. 运行Carthage，获取和构建上述罗列的frameworks。
3. 拖拽构建完成的 .framework 二进制文件至项目中。
4. 注：Carthage仅支持动态库，且使用范围：ios8及以上

## 安装Carthage

1. 使用Homebrew：

```
$ brew update
$ brew install carthage
删除老版路径：/Library/Frameworks/CarthageKit.framework
```

## 添加**frameworks**到项目中

### 1. 创建Cartfile，并罗列frameworks

```
$ cd ..path
$ touch Cartfile
$ open -a Xcode Cartfile (optional)
add origin to file
```

### 2. 执行运行命令：`carthage update --platform ios`

下载依赖库至 `Carthage/Checkout` 目录，并编译每个依赖库或下载编译好的库

更新亦是用该命令或者单独更新某一个库：`carthage update Box Result`

### 3. 引入Framework：

- i. 在Mac开发中：在项目Targets的 General tab-> Embedded Binaries，拖拽编译好的依赖库framework导入即可。

编译好的库位于 `Carthage/Build` 路径中。

- ii. 在iOS中：在项目Targets的 General tab-> “Linked Frameworks and Libraries”，拖拽编译好的依赖库framework导入。

追加步骤1：在targets中点击+，选择New Run Script Phase，选择/bin/sh,添加脚本：`/usr/local/bin/carthage copy-frameworks`

追加步骤2：在1中脚本下Input Files添加如下 `$(SRCROOT)/Carthage/Build/iOS/Result.framework` 语句

追加步骤3：在1中脚本下Output Files添加如下 `$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/Result.framework` 语句

### 4. 可选。添加debug 文件

在项目Targets的Buildling phases-> + -> new copy files Phases,选择Destination为Products Direction，Subpath为相应依赖库framework的DYSM文件路径

---

## 二、Cartfile文件构成

### 1. Cartfile

- 文件由2部分构成：origin、Version requirement

## origin：

- 目前支持源：GitHub repositories、Git repositories、以https提供的二进制framework。

- GitHub:

```
github "ReactiveCocoa/ReactiveCocoa" # GitHub.com
github "https://enterprise.local/ghe/desktop/git-error-translations" # GitHub
Enterprise
```

- Git:

```
git "https://enterprise.local/desktop/git-error-translations2.git"
```

- 二进制库：

```
binary "https://my.domain.com/release/MyFramework.json"
```

## version要求

- 针对具体framework版本要求
- `>= 1.0` : 最低版本为1.0
- `~> 1.0` : 需兼容版本1.0（可以高于1.0，但必须兼容1.0）
- `== 1.0` : 严格控制为版本1.0
- `some-branch-or-tag-or-commit` : 针对Git要求，不支持二进制库

## 示例

```

Require version 2.3.1 or later
github "ReactiveCocoa/ReactiveCocoa" >= 2.3.1

Require version 1.x
github "Mantle/Mantle" ~> 1.0 # (1.0 or later, but less than 2.0)

Require exactly version 0.4.1
github "jspahrsummers/libextobjc" == 0.4.1

Use the latest version
github "jspahrsummers/xccconfigs"

Use the branch
github "jspahrsummers/xccconfigs" "branch"

Use a project from GitHub Enterprise
github "https://enterprise.local/ghe/desktop/git-error-translations"

Use a project from any arbitrary server, on the "development" branch
git "https://enterprise.local/desktop/git-error-translations2.git" "development"

Use a local project
git "file:///directory/to/project" "branch"

A binary only framework
binary "https://my.domain.com/release/MyFramework.json" ~> 2.3

```

## 2. Cartfile.private

- framework需要使用的依赖库，对于主项目来讲，非必须引入的部分，即可选的依赖库。
  - 使用场景：如一些测试库：在debug阶段需要，在发布阶段需要去除。

## 3. Cartfile.resolved

- 执行 carthage update 后，在 cartfile 同级别中将产生此目录：罗列所有依赖库和版本信息

## 4. Carthage/Build

- 执行 carthage update 后产生，涵盖：二进制库和debug信息

## 5. arthage/Checkouts

- 执行 `carthage checkout` 后产生于 `application` 目录中，涵盖：依赖库的源码文件，`carthage/Checkouts` 目录用于后续 `carthage build` 命令。
- todo : submodules

## 6. ~/Library/Caches/org.carthage.CarthageKit

- 自动创建，涵盖 Git 下载的依赖库的源数据，多个项目公用。
- 在下次调用 `carthage checkout`，将重新生成

## 7. Binary Project Specification

- 二进制文件
- 例：

```
{
 "1.0": "https://my.domain.com/release/1.0.0/framework.zip", "1.0.1":
 "https://my.domain.com/release/1.0.1/framework.zip"
}
```

Copyright © iOS 开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 版本控制

## 版本控制

- Mac\_For\_Git\_Server 简易配置
- SVN

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

## 一、目标

- 在没有远程Git服务器的情况下，又需要多人协作开发项目。我们可以临时使用自己的Mac电脑搭建简易Git服务器。
- 使用本电脑新账户作为Git服务器，并不能保证数据安全问题，所以只是权宜之计。

## 二、创建**Git**服务器

**server**端（即**Mac**）建立新账户，始终后台开启。

1. 新建Giter账户
2. \$ sudo chmod u+w /etc/sudoers
3. \$ sudo vi /etc/sudoers

找到这一行 "root ALL=(ALL) ALL"，在下面添加 "AccountName ALL=(ALL) ALL"  
(这里是 git ALL=(ALL) ALL)并保存。

## 三、**Git**仓库建立及关联

**client**端(设置访问**Git**服务器的权限用户，设置一次即可)

1. \$ ssh git@yourComputerName.local mkdir .ssh # 登陆 server 并创建 .ssh 文件夹
2. \$ scp ~/.ssh/id\_rsa.pub giter@pan.local:.ssh/authorized\_keys  
| cat ~/.ssh/id\_rsa.pub

远程**server**：远程仓库建立

1. \$ su Giter
2. cd desktop
3. mkdir git\_repo.git
4. cd git\_repo.git
5. git init --bare

本地仓库建立及关联远程仓库

1. \$ mkdir file
2. \$ git init

3. \$ git add .
4. \$ git commit -m "init"
5. \$ git remote add origin Giter@pan.local:/Users/giter/Desktop/git\_repo.git
6. git push

## 四、其它

1. 设置中远程登录设为enable

1. 去除.git链接仓库

```
find . -name ".git" | xargs rm -Rf
```

2. 获取本地RSA子串：

```
cat ~/.ssh/id_rsa.pub
```

```
生成：ssh-keygen -t rsa
```

Copyright © iOS 开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 打包&发布

## 打包&发布

- 帐号相关
- 持续集成
- 优化
- 灰度分发&发布

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 帐号相关

## 帐号相关

- 申请帐号
- IAP
- iAd

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 持续集成

## 持续集成

- Jenkins
- fastlane

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48

- 优化

## 优化

- app瘦身

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 灰度分发&发布

## 灰度分发&发布

- DMG打包流程
- 发布前CheckList-通用版

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

# 一、可视化导出DMG文件

步骤：

1. 打开 磁盘工具，并新建一空白映像，如图所示



2. 在弹出的选项框中：命名文件并保存目录、分区改为CD/DVD模式,读写磁盘映像，其它默认即可。



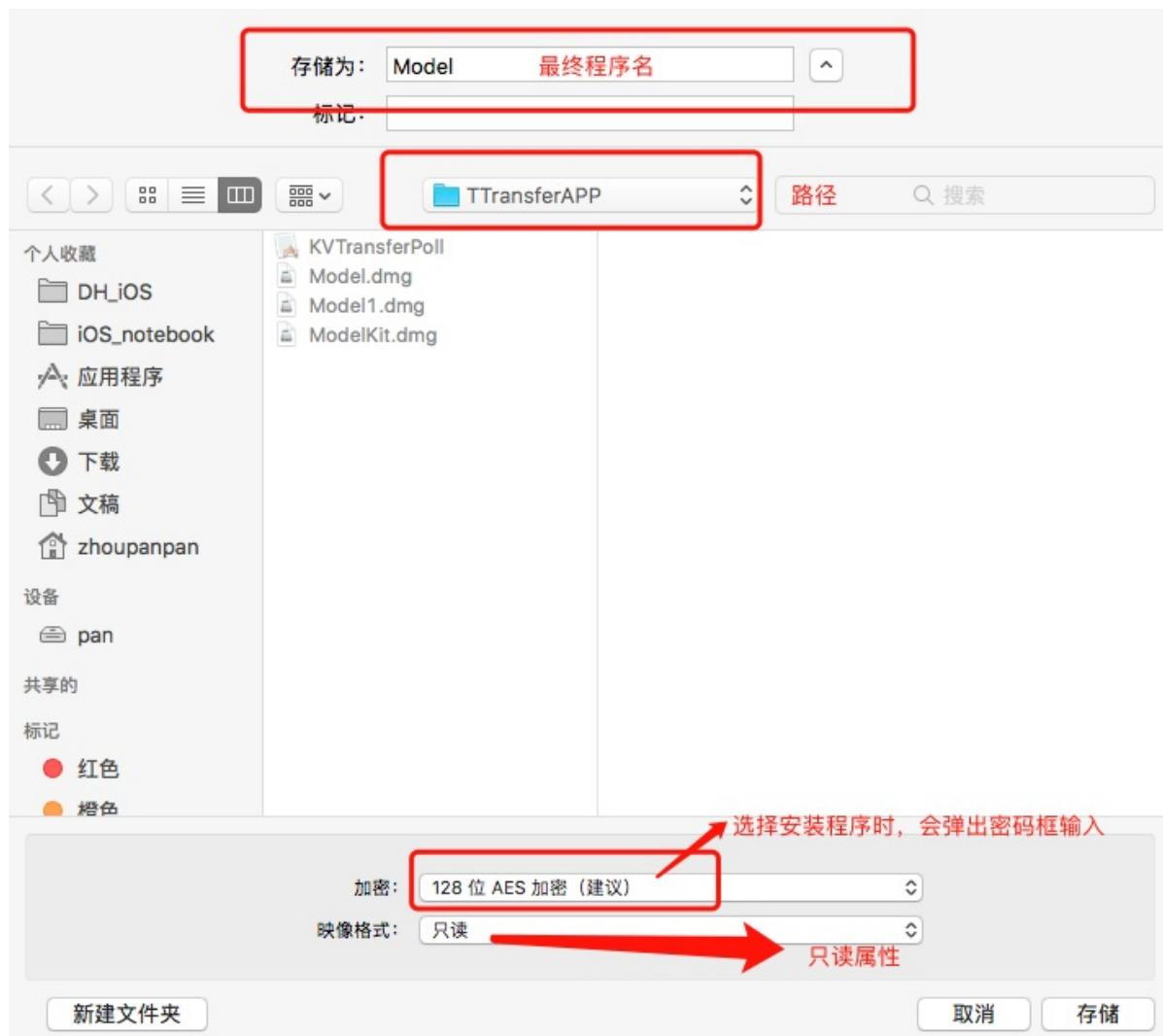
3. 点击上图所示的 Model1.dmg，会在桌面展示 未命名 程序，点击之后显示空白。将项目生成的 \*.app 拖拽至内部、添加 应用程序 文件夹路径、添加背景图片，并设置如下：



4. 最终呈现效果大致如下 (QQ示例) :



5. 设置完成后，再次打开 磁盘工具，选择 映像 -> 转换，选择要转换并且已修改好的 Model1.dmg，配置如下图，然后导出新的DMG文件。



## 注意问题

1. 使用的 \*.app 注意需是release版本。
2. 背景图片资源隐藏配置： mv background.jpg .background.jpg.jpg
3. 添加 应用程序 文件夹替身：

```
ln -s /Applications /Volumes/Model1/Applications
```

4. 添加<软件许可协议>，参考：

- i. <http://www.owsiak.org/adding-license-to-a-dmg-file-in-5-minutes-or-less/>
- ii. <https://justinyan.me/post/1715>

## 二、脚本化方式导出**DMG**文件

- 待添加

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

1. https开启，环境切换是否复原为线上。
2. 应用内Debug模块是否关闭并移除。
  - main()函数处有侵入代码，需要清除或者注释掉。
3. 临时资源：如图片、html是否完全替换。
4. 版本Version设置是否正确；buildVersion设置是否正确。
5. app名称，图标icon，启动页是否正确。
6. 网络模块中version参数是否设置正确。
7. info.list配置：
  - Build Active Architecture Only -> No
  - App Transport Security Settings 网络配置是否正确。
  - 证书配置等。
  - bundle id 是否匹配。
8. 查看项目中静态分析的警告条目，是否是逻辑性问题引起。
9. 上线前私有API检测。
10. 官网版本文案、图片是否更新为匹配项。

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- 规范

## 规范

- 工程规范
- 代码规范
- UED规范
- CodeReview规范

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 工程规范

## 工程规范

- 脚手架规范（crash、数据统计等）
- 机型屏幕适配规范

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 代码规范

## 代码规范

- Obj-C规范
- Swift规范1：[linkedin](#)代码规范译文
- Swift规范2：[raywenderlich](#)代码规范译文

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

- I、代码规范
  - 一、命名规范
  - 二、格式规范
  - 三、语法规范
- II、文件规范
  - 文件
  - 结构
  - 代码布局
- III、TODO :

## 提要

1. 所有规范（试行）基于苹果官方[编码准则](#)。若有冲突之处，以官方为主。

## 目录

I、代码规范

II、文件规范

III、TODO:

## I、代码规范

### 一、命名规范

- 变量和属性：遵循小驼峰规则，且 \* 紧跟变量

1. NSString \*userName;
2. @property (nonatomic, strong) NSString \*nickName;
3. 临时变量禁止使用下划线开头！因为Default Synthesis生成的成员变量就是下划线打头的，避免可能存在的冲突！
4. 关于NSString的定义：明显预知的类型可以不加后缀，如userName可以预判是String类型；但不可预判的必须加，如URL string和URL对象需要添加后缀。
5. 关于全局类的属性，变量。定义时需要无歧义描述，如数组定义：cells，cellArray。

- 属性的访问：需使用点语法执行

```
self.userName = @"john";
```

- 方法的访问：需使用方括号访问

```
count = [self.userName length];
```

- 类名：遵循大驼峰式命名法，需要带上前缀。

```
@interface UITextView
```

视图类：`**View`；控制器类：`**Controller`；实体模型：`**Model` 或者 `**Entity`。依此类推之。

类似的，对于Protocol、Struct、const命名规则，与之相同。

- 方法名：的每一部分都必须遵循小驼峰命名法；若参数过长，则每个参数占用一行且以`:`对齐。（依据Xcode提示即可）

```
- (NSRange)rangeOfString:(NSString *)aString options:
(NSStringCompareOptions)mask;
```

方法名详细参考：[Cocoa Style for Objective-C: Part II](#)

特例：方法名以特殊大写字母开头。如PDFXXX、TIFFXX等。

行为方法：以动词开头，宜使用诸如`can`、`should`、`will`等情态动词，避免使用`do`、`does`。

返回值方法：避免使用`get`开头。

- 构造方法：返回实例类型为`instancetype`而不再是`id`类型。

便于编译器正确推断结果类型

- 函数：其实就是所谓的C全局函数，一般采用`前缀+名字`的方式来命名，其中名字部分采用大驼峰式命名法。

1. `NSString *NSHomeDirectory(void);`
2. `NSString NSHomeDirectoryForUser(NSString userName);`

- 常量：使用静态变量的方式来定义数值或者字符串常量，而不要使用预编译的方式。好处是前者存在类型检查，而后者只是简单的文本替换。

```
NSString * const kMyConstVar = @"myConstVar";
```

- 枚举：使用苹果提供的宏来定义枚举，`NS_ENUM`用来定义普通的枚举，`NS_OPTIONS`用来定义`bitmask`。

```
typedef NS_ENUM(NSUInteger, UIViewAnimationTransition) {
 UIViewAnimationTransitionNone,
 UIViewAnimationTransitionFlipFromLeft,
 ...
};
```

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
 UIViewAutoresizingNone = 0,
 UIViewAutoresizingFlexibleLeftMargin = 1 << 0,
 UIViewAutoresizingFlexibleWidth = 1 << 1,
 ...
};
```

- **Notification:** 通知必须以字符串常量的方式存在。命名遵循如下组成结构 [Name of associated class] + [Did | Will] + [UniquePartOfName] + Notification。

```
//TBHDXXX.h
//如果通知名在外部需要使用，则需要在头文件中进行extern声明
extern NSString * const TBHDXXXDidReceivedNotification;
```

```
//TBHDXXX.m
NSString * const TBHDXXXDidReceivedNotification = @"TBHDXXXDidReceivedNotification";
```

- **Exceptions:** 和上面的通知的命令方式很相似，遵循以下模式：

```
[Prefix] + [UniquePartOfName] + Exception
```

1. NSRangeException
2. NSInternalInconsistencyException

- 其它常量变量：遵循 [Prefix] + [UniquePartOfName] 的命名准则。

```
FOUNDATION_EXPORT NSString * const NSURLErrorDomain;
```

- **User Defaults:** 中的 key 值命名。

- 关于警告的处理：工具类、方法不使用的，可以使用注释或者 DEPRECATED 等宏定义做警告处理，后续逐步丢弃掉。

- 关于常量相关的区分细节：

1. `#define PI @"3.14":` 宏定义。预处理器文本替换，无类型检查；占用代码段地址，大量宏会导致二进制文件过大。
2. `NSString *PI = @"3.14":` 变量。共享内存空间；编译阶段进行类型检查。
3. `static const NSString *PI = @"3.14":` 局部常量。共享内存空间；用 `static` 修饰后，不能提供外界访问。
4. `const NSString PI = @"3.14":` 全局常量1。共享内存空间；"`\PI`"不能被修改，"`PI`"能被修改；无论定义在哪个文件夹，外部均能访问。（需要调用时声明 `extern NSString *PI`）
5. `NSString const PI = @"3.14":` 全局常量2。共享内存空间；"`\PI`"不能被修改，"`PI`"能被修改，同全局常量1。
6. `NSString const PI = @"3.14":` 全局常量3。共享内存空间；"`PI`"不能被修改，"`*PI`"能被修改。

## 二、格式规范

- 空格：合理使用空格，增加代码的可读性。

1. `NSArray *names = @[@"a", @"b"];` ✓
2. `NSArray *names = @[@"a",@"b"];` ✗
3. `Dictionary NSDictionary *errorCodeMap = @{@"0":@"网络错误"};` ✓  
`NSDictionary *errorCodeMap = @{@"0":@"网络错误"};` ✗
4. 二元/三元运算符(Dot运算符例外,一元运算符以及强制类型转换无需空格)  
`if(a && b) {}` ✓   `if(a&b) {}` ✗
5. `for` 循环中也应使用空格隔开 `for (unsigned int i = 0; i < 10; i++)...` ✓   `for (unsigned int i = 0;i < 10;i++)...` ✗
6. 方法声明中的“-”和“+”后面必须留空格 `- (id)initWithCoder:(NSCoder *)aDecoder;` ✓  
`- (id)initWithCoder:(NSCoder *)aDecoder;` ✗
7. 方法调用的时候，`receiver` 和 `method` 之间必须要有空格 `[[NSUserDefaults standardUserDefaults] synchronize];` ✓  
`[[NSUserDefaults standardUserDefaults synchronize];` ✗
8. 代码块的起始的 `{` 前要留一空格，`else` 前和 `}` 也要留一空格 `if {...} else {...}`
9. 注释内容和注释标志要有一个空格: `// momo`
10. 逻辑比较复杂的功能修改，需要添加注释：时间，详细内容等。

- 回车换行：

1. 代码之间不需要多余的回车，不超过2行回车。
2. 不同方法体添加一行回车即可。
3. 方法体内不同逻辑块之间加一行回车即可。

- {}大括号:左大括号不要另起一行，右大括号必须新起一行。即使只有一条语句。

```
if(ok) {
 return success;
}
```

- ?:条件运算符:严禁嵌套使用，否则代码的可读性会下降。

1. a ? a : b (此种情况也可写成 a ?: b ) ✓
2. a ? (b > c ? b : c) : d ✗

- 注释:注释本身也存在其相关语法，可以参考苹果的[HeaderDoc User Guide](#)。

1. 不要进行大片的代码注释，直接删除它们。

- 函数注释:针对文件或者函数进行详细解释格式。

基本涵盖：名称、描述、入参、出参、返回值

- **Option + command + /**: 函数注释 以及h文件中类的注释。方法体、函数体内部业务逻辑较复杂的需要单独添加注释。

### 三、语法规范

- NSArray, NSDictionary, NSNumber 能够使用新的文法的地方就使用新的文法:

1. NSNumber \*price = @1000;
2. NSArray \*names = @[@"a", @"b"];
3. NSDictionary \*errorCodeMap = @{@"0": @"网络错误", @"1": @"连接超时"};

- **CGRect函数**: 使用[CGRect函数](#)来获取CGRect结构体中的数据,因为CGRect函数在计算结果前会对变量进行检查，比如对长宽进行检查，是否为正。

```
CGFloat height = CGRectGetHeight(frame); ✓ CGFloat height =
frame.size.height; ✗
```

- 单例模式:统一使用 `dispatch_once` 的方式，这种方式简单，性能好，线程安全。

```
+ (MyClass *)sharedInstance{
 // Static local predicate must be initialized to 0
 static MyClass *sharedInstance = nil;
 static dispatch_once_t onceToken;
 dispatch_once(&onceToken, ^{
 sharedInstance = [[MyClass alloc] init];
 // Do any other initialisation stuff here
 });
 return sharedInstance;
}
```

## II、文件规范

### 文件

- **Utility**和**Helper**:一些工具或者辅助类文件名的后缀。

`Utility` 或 `Util` :一般在使用的时候无需实例化，直接使用其类方法，是无状的。  
(因为属于Common类文件，所以不推荐使用)

`Helper` 或 `Builder` :可以实例化，有状态的

- 导入:头文件中能够不import (使用class声明即可) 的时候就严谨import。

- **h文件**:布局方案详细

1. 头文件{}外，空1行书写属性，如果需要分类区别，各类别之间空1行
2. 属性下面空1行开始写方法，如果需要分类区别，各类别之间空1行
3. 方法完成后，空1行@end
4. 如果需要声明protocol，空2行接着写。通常protocol写在@end后面，但是声明在 @interface之前

- **m文件**:布局方案详细

1. 文件说明与头文件包含(#import)之间空1行
2. 头文件包含(#import)之间，如果需要分类区别，各类别之间空1行
3. 方法与方法之间空1行

### 结构

- 代码行数:一行最大为100列
- 函数分组:使用类似 `#pragma mark --- <* text *>---` 方式进行分组

#### 分组详情:

- 该分组模式业已提交**code snippet**，便于统一处理文件。

- 具体导入使用方法参考Github地址中 `README`说明。
- github地址：[snippets\\_tools](#)

根据 `函数分组` 方案，文件内部组织形式为：

1. `- (void)dealloc{}`
2. `#pragma mark --- public ---`

对外公开的方法，即在h文件中声明的方法

### 3. #pragma mark --- life cycle ---

- (void)viewDidLoad{}

一系列声明周期方法集合

### 1. #pragma mark --- delegate ---

一系列 自定义/系统 代理方法

### 2. #pragma mark --- event ---

如UIButton点击事件

### 3. #pragma mark --- private ---

相对 public 而言，只在m文件中使用的自定义方法

### 4. #pragma mark --- getter & setter ---

一系列 属性 定义方法

## 代码布局

- 不想暴露于头文件的属性，可以使用类扩展(Class Extension，也即匿名Category)的方式进行声明。

```
//PZMyClass.m文件中
@interface PZMyClass ()

@property (nonatomic, strong) UIView *myView;

@end
```

## III 、TODO :

后续需要继续完善的部分（红色分支）：



Copyright © iOS 开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- [linkedin规范](#)
  - [1. 代码格式](#)
  - [2. 命名](#)
  - [3. 代码规范](#)
  - [4. 文档、注释](#)

# linkedin规范

原文链接：[linkedin规范](#)

确保已阅读：[Apple's API Design Guidelines.](#)

该指南针对于Swift4.0、更新于2018-02-14

## 1. 代码格式

**1.1** 针对**tabs** 使用**4个空格**。

**1.2** 强制约定每行最多字符数为**160**，避免一行过长。（**Xcode->Preferences->Text Editing->Page guide at column: 160**）

**1.3** 确保每个文件的结尾处有一空行。

**1.4** 确保在任何地方都无额外多余的空格出现。（**Xcode->Preferences->Text Editing->Automatically trim trailing whitespace + Including whitespace-only lines**）

**1.5** 不要在新的一行放置花括号，我们使用**1TBS style**。

```
class SomeClass {
 func someMethod() {
 if x == y {
 /* ... */
 } else if x == z {
 /* ... */
 } else {
 /* ... */
 }
 }

 /* ... */
}
```

**1.6** 针对属性、常量、变量、字典的**key**、**protocol**、父类等进行类型书写时，不要在冒号前

添加空格。在其后添加空格。

```
// 类型定义
let pirateViewController: PirateViewController

// dictionary syntax (注意是左对齐而非冒号对齐)
let ninjaDictionary: [String: AnyObject] = [
 "fightLikeDairyFarmer": false,
 "disgusting": true
]

// 函数声明
func myFunction<T, U: SomeProtocol>(firstArgument: U, secondArgument: T) where T.RelatedType == U {
 /* ... */
}

// calling a function
someFunction(someArgument: "Kitten")

// superclasses
class PirateViewController: UIViewController {
 /* ... */
}

// protocols
extension PirateViewController: UITableViewDataSource {
 /* ... */
}
```

**1.7** 一般而言，逗号后面应该有空格。

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

**1.8** 二元运算符前后应均有一个空格。如`+`、`==`、`->`。

**1.9** 在`(`之后和`)`之前不应有空格。

```
let myValue = 20 + (30 / 2) * 3
if 1 + 1 == 3 {
 fatalError("The universe is broken.")
}
func pancake(with syrup: Syrup) -> Pancake {
 /* ... */
}
```

**1.10** 遵循**Xcode**推荐的缩进风格。（即当按下**CTRL-I**时，你的代码无变化）

当声明一个跨越多行的函数时，倾向于使用**Xcode**（从7.3版本开始）默认的语法。

```
// Xcode indentation for a function declaration that spans multiple lines
func myFunctionWithManyParameters(parameterOne: String,
 parameterTwo: String,
 parameterThree: String) {
 // Xcode indents to here for this kind of statement
 print("\(parameterOne) \(parameterTwo) \(parameterThree)")
}

// Xcode indentation for a multi-line `if` statement
if myFirstValue > (mySecondValue + myThirdValue)
 && myFourthValue == .someEnumValue {

 // Xcode indents to here for this kind of statement
 print("Hello, World!")
}
```

**1.11** 调用具有许多参数的函数时，将每个参数置于单独的行中，并使用一个额外的缩进。

```
someFunctionWithManyArguments(
 firstArgument: "Hello, I am a string",
 secondArgument: resultFromSomeFunction(),
 thirdArgument: someOtherLocalProperty)
```

**1.12** 当处理一个足够大的隐式数组或字典，并保证将其分解成多行时，将【和】视为方法中的大括号{}来处理。if语句、方法中的闭包等，类似处理。

```
someFunctionWithABunchOfArguments(
 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)
 })
```

**1.13** 优先使用本地常量或其他缓解技术，以尽可能避免多行谓词出现。

```

// PREFERRED
let firstCondition = x == firstReallyReallyLongPredicateFunction()
let secondCondition = y == secondReallyReallyLongPredicateFunction()
let thirdCondition = z == thirdReallyReallyLongPredicateFunction()
if firstCondition && secondCondition && thirdCondition {
 // do something
}

// 不推荐
if x == firstReallyReallyLongPredicateFunction()
 && y == secondReallyReallyLongPredicateFunction()
 && z == thirdReallyReallyLongPredicateFunction() {
 // do something
}

```

## 2. 命名

**2.1** 在Swift中不需要Objective-C风格的前缀。（如命名**GuybrushThreepwood**而非**LIGuybrushThreepwood**）

**2.2** 使用**PascalCase**拼写法，来命名类型名称。（如**struct**、**enum**、**class**、**typedef**、**associatedtype**等）

帕斯卡拼写法：一种计算机编程中的变量命名方法。它主要的特点是将描述变量作用所有单词的首字母大写，然后直接连接起来，单词之间没有连接符。

**2.3** 使用**camelCase**（初始小写字母）拼写法，命名函数，方法，属性，常量，变量，参数名称，枚举类型等。

**2.4** 处理缩略词或首字母纯大写的缩略词时，均原样使用该缩略词即可。但有个例外：若该词是在一个名字的开始，需要以小写字母开头 - 在这种情况下，使用全部小写的缩写词。

```

// "HTML" is at the start of a constant name, so we use lowercase "html"
let htmlBodyContent: String = "<p>Hello, World!</p>"
// Prefer using ID to Id
let profileID: Int = 1
// Prefer URLFinder to UrlFinder
class URLFinder {
 /* ... */
}

```

**2.5** 所有常量都应该是静态的，独立的。所有常量置于**class**、**struct**、**enum**的醒目处，且它们含有众多的常量，应依照相同的前缀、后缀或者用途将所属常量分组。

```
// PREFERRED
class MyClassName {
 // MARK: - Constants
 static let buttonPadding: CGFloat = 20.0
 static let indianaPi = 3

 static let shared = MyClassName()
}

// 不推荐
class MyClassName {
 // Don't use `k`-prefix
 static let kButtonPadding: CGFloat = 20.0

 // Don't namespace constants
 enum Constant {
 static let indianaPi = 3
 }
}
```

## 2.6 对于泛型（**generics**）和关联（**associated**）类型，使用**PascalCase**描述泛型。

- 如果这个词与它所符合的协议或它的子类冲突，可以将 Type 后缀附加到泛型、关联类型名称后面。

```
class SomeClass<Model> { /* ... */ }
protocol Modelable {
 associatedtype Model
}
protocol Sequence {
 associatedtype IteratorType: Iterator
}
```

## 2.7 命名具有可描述性和无歧义性。

```
// PREFERRED
class RoundAnimatingButton: UIButton { /* ... */ }

// NOT PREFERRED
class CustomButton: UIButton { /* ... */ }
```

## 2.8 不可简化，缩短单词，或单一字符来命名。

```
// PREFERRED
class RoundAnimatingButton: UIButton {
 let animationDuration: NSTimeInterval

 func startAnimating() {
 let firstSubview = subviews.first
 }

}

// 不推荐
class RoundAnimating: UIButton {
 let aniDur: NSTimeInterval

 func srtAnmating() {
 let v = subviews.first
 }

}
```

## 2.9 如果不是很明显，则在常量或变量名称中包含类型信息。

```
class ConnectionTableViewCell: UITableViewCell {
 let personImageView: UIImageView

 let animationDuration: TimeInterval

 // it is ok not to include string in the ivar name here because it's obvious
 // that it's a string from the property name
 let firstName: String

 // though not preferred, it is OK to use `Controller` instead of `ViewController`
 let popupController: UIViewController
 let popupViewController: UIViewController

 // when working with a subclass of `UIViewController` such as a table view
 // controller, collection view controller, split view controller, etc.,
 // fully indicate the type in the name.
 let popupTableViewController: UITableViewController

 // when working with outlets, make sure to specify the outlet type in the
 // property name.
 @IBOutlet weak var submitButton: UIButton!
 @IBOutlet weak var emailTextField: UITextField!
 @IBOutlet weak var nameLabel: UILabel!

}
```

## 2.10 函数命名：函数名、参数名需具有可描述性，可理解其作用。

## 2.11 协议命名,依据协议作用分以下情况：

1. 使用名词命名：描述什么做什么。如CollectionView。
2. 使用后缀 able 、 ible 或 ing ：描述具备什么样的能力。如Equatable。
3. 视具体使用场景：其它情况。如使用后缀 Protocol 等。

```
// here, the name is a noun that describes what the protocol does
protocol TableViewSectionProvider {
 func rowHeight(at row: Int) -> CGFloat
 var numberOfRows: Int { get }
 /* ... */
}

// here, the protocol is a capability, and we name it appropriately
protocol Loggable {
 func logCurrentState()
 /* ... */
}

// suppose we have an `InputTextView` class, but we also want a protocol
// to generalize some of the functionality - it might be appropriate to
// use the `Protocol` suffix here
protocol InputTextViewProtocol {
 func sendTrackingEvent()
 func inputText() -> String
 /* ... */
}
```

## 3. 代码规范

### 3.1 通用

**3.1.1** 任何场景下，只要允许，尽可能使用**let**，而非**var**。

**3.1.2** 当从一个集合转换到另一个集合时，优先考虑**map**、**filter**、**reduce**等的组合。

1. 确保在使用这些方法时避免使用有副作用的闭包

```

// PREFERRED
let stringOfInts = [1, 2, 3].flatMap { String($0) }
// ["1", "2", "3"]

// 不推荐
var stringOfInts: [String] = []
for integer in [1, 2, 3] {
 stringOfInts.append(String(integer))
}

// PREFERRED
let evenNumbers = [4, 8, 15, 16, 23, 42].filter { $0 % 2 == 0 }
// [4, 8, 16, 42]

// 不推荐
var evenNumbers: [Int] = []
for integer in [4, 8, 15, 16, 23, 42] {
 if integer % 2 == 0 {
 evenNumbers.append(integer)
 }
}

```

**3.1.3** 若果可以通过推断得出常量、变量的类型，那就不用显式声明类型方式。

**3.1.4** 若函数返回值有多个，则使用元祖（**Tuple**）而非**inout**参数返回。

1. 若返回值不易解读，最好使用标记元祖来清晰表达返回值作用。
2. 若多次使用某个元祖，可以考虑使用别名 `typealias`
3. 若返回元祖中含有 $>=3$ 个元素，则考虑使用 `struct` 或 `class` 替代方案。

```

func pirateName() -> (firstName: String, lastName: String) {
 return ("Guybrush", "Threepwood")
}

let name = pirateName()
let firstName = name.firstName
let lastName = name.lastName

```

**3.1.5** 使用**delegate**、**protocol**时，注意循环引用问题。一般而言，此类属性命名，需要声明为**weak**。

**3.1.6** 在**escaping**闭包中直接调用**self**，注意循环引用问题。使用**capture list**解决。

```

myFunctionWithEscapingClosure() { [weak self] (error) -> Void in
 // you can do this

 self?.doSomething()

 // or you can do this

 guard let strongSelf = self else {
 return
 }

 strongSelf.doSomething()
}

```

### 3.1.7 不要使用可标记的Breaks。

### 3.1.8 不要在控制流谓词上添加括号。

```

// PREFERRED
if x == y {
 /* ... */
}

// 不推荐
if (x == y) {
 /* ... */
}

```

### 3.1.9 在enum枚举中：避免写出enum完整方式，尽可能使用简写方式。

```

// PREFERRED
imageView.setImageWithURL(url, type: .person)

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

```

### 3.1.10 在类方法中：避免使用缩写方式，因为从类方法中推断上下文通常比较困难。

```

// PREFERRED
imageView.backgroundColor = UIColor.white

// NOT PREFERRED
imageView.backgroundColor = .white

```

### 3.1.11 除非必要，不使用self.。

### 3.1.12 书写方法，是否考虑其后续被重写。

1. 若后续重写，则不需要考量。
2. 若后续不重写，需使其标记为 `final`。（有助于优化编译时间）
3. 注：在 Library 中和在主项目中使用 `final`，意义不同。

### 3.1.13 使用诸如 `else`、`catch` 等语法时，其后跟随 `block` 块。遵循 **1TBS style**。

```
if someBoolean {
 // do something
} else {
 // do something else
}

do {
 let fileContents = try readFile("filename.txt")
} catch {
 print(error)
}
```

### 3.1.14 定义类函数或类属性时，推荐使用 `static` 而非 `class`。

- 仅当在子类中重写类方法或属性时，使用 `class`。当然此时更推荐使用 `protocol` 技术作为替代方案。

### 3.1.15 若函数无参数，无歧义，且有返回值。可以考虑使用计算属性来实现。

demo TODO:

## 3.2 访问权限修饰符

### 3.2.1 权限修饰符写在最前：

```
// PREFERRED
private static let myPrivateNumber: Int
// NOT PREFERRED
static private let myPrivateNumber: Int
```

### 3.2.2 修饰符需与要修饰语句在同一行，不应隔行展示：

```
// PREFERRED
open class Pirate {
 /* ... */
}

// NOT PREFERRED
open
class Pirate {
 /* ... */
}
```

**3.2.3** 一般情况下，不必显式的添加`internal`修饰符，因它是访问权限默认值。

**3.2.4** 若某属性需在单元测试中可访问，则更改修饰符为`internal`，并使用`@testable import ModuleName`。

若属性本应修饰为`private`，但为了单元测试，更改为`internal`。记得添加文本注释解释原因。如下使用`warning`标记语法阐述之：

```
/**
This property defines the pirate's name.
- warning: Not `private` for `@testable`.
*/
let pirateName = "LeChuck"
```

**3.2.5** 若可能，尽量使用`private`而非`fileprivate`。

**3.2.6** 当需在`public`和`open`修饰符中选择其一时：

1. 使用`open`：当需要在给定模块`Module`外部子类化时。
2. 否则使用`public`。

注：`open`、`public`和`internal`修饰符使用`@testable import` 测试时，均可子类化。所以测试目的并不是使用`open`的原因。

一般而言，在Lib库中时，使用`open`自由度较高的修饰符；在代码库中的模块中，偏向于使用较为保守的修饰符，如在app内部，同步更改较为容易。（翻译不精确，可参考原文）

### 3.3 自定义运算符

- 建议为自定义运算符创建命名函数
- 如果你想引入一个自定义的操作符，确保你有一个很好的理由，解释为什么要引入一个新的操作符到全局范围，而不是使用其他的构造。

- 你可以覆盖现有的运算符来支持新的类型（特别是`==`）。但是，您的新定义必须保留运算符的语义。例如，`==`必须总是测试相等并返回一个布尔值。

## 3.4 Switch 语句和 enum

**3.4.1** 使用**switch**语句时，若检测的是有限集合分支，则不需添加**default case**，代之以：将未使用的案例放在底部，并使用**break**关键字来防止执行。

**3.4.2** 因在**swift**中每个**case**后面是隐式具有**break**功能的，若无必要，无需再次手动追加**break**关键字。

**3.4.3** **case**语句应按照**Swift**默认标准与**switch**语句本身保持一致。

**3.4.4** 当定义的**case** 具有关联值，确保这个值也被恰当地标记。（如**case hunger(hungerLevel: Int)**而非**case hunger(Int)**）

```
enum Problem {
 case attitude
 case hair
 case hunger(hungerLevel: Int)
}

func handleProblem(problem: Problem) {
 switch problem {
 case .attitude:
 print("At least I don't have a hair problem.")
 case .hair:
 print("Your barber didn't know when to stop.")
 case .hunger(let hungerLevel):
 print("The hunger level is \(hungerLevel).")
 }
}
```

**3.4.5** 若可能，尽量使用值列表展示（如 **case 1,2,3**），而非**fallthrough**。

**3.4.6** 若你有一个**default case**, 不应被适配。建议使用**error throw**来完善。（其它类似技术均可，如**asserting**）

```
func handleDigit(_ digit: Int) throws {
 switch digit {
 case 0, 1, 2, 3, 4, 5, 6, 7, 8, 9:
 print("Yes, \(digit) is a digit!")
 default:
 throw Error(message: "The given number was not a digit.")
 }
}
```

## 3.5 Optionals

**3.5.1** 唯一一次你应该使用隐式解包选项**optional**是@IBOutlets。在其他情况下，最好使用非可选**non-optional**或常规的**ptional**可选属性。是的，在某些情况下，除非您可以“保证”使用该属性时永远不会为**nil**，但最好是安全一致。不要使用强制解包。

**3.5.2** 不要使用**as!**、**try!**。

**3.5.3** 如果不打算实际使用存储在可选项中的值，但是需要确定该值是否为零，则建议显式检查该值是否为**nil**，而不是使用**if let**语法。

```
// PREFERRED
if someOptional != nil {
 // do something
}

// NOT PREFERRED
if let _ = someOptional {
 // do something
}
```

**3.5.4** 不要使用**unowned**。你可以认为**unowned**等价于一个隐含**unwrapped**的**weak**属性（虽然**unowned**完全忽略引用计数，性能上有轻微的改进）。既然我们不想有隐含的解包，我们同样不想要**unowned**修饰的属性。

```
// PREFERRED
weak var parentViewController: UIViewController?

// NOT PREFERRED
weak var parentViewController: UIViewController!
unowned var parentViewController: UIViewController
```

**3.5.5** 当**unwrapping optionals**时，在适当的场景下使用相同的名称来表示常量或变量。

```
guard let myValue = myValue else {
 return
}
```

## 3.6 Protocols

当实现某协议时，有两种代码组织方式：

1. 使用 `// MARK:` 注释来隔离协议实现与其它代码段。
2. 在 `class` struct` 实现代码外添加扩展**extension**来实现协议。（但仍在同一个文件中）

注：当使用扩展时，扩展中的方法不能被子类覆盖重写，这会使测试变得困难。如果这是一个常见的用例，那么坚持使用 方法#1 来保持一致性可能会更好。否则，使用 方法#2 可以更清楚地分离关注点。

即使在使用 方法#2 时，无论如何也要添加 // MARK: 语句，以便在Xcode的 method / property / class / 等中更易读。列表UI。

## 3.7 Properties

### 3.7.1 若创建一个只读的，计算的属性，提供的**getter**方法，无需**get {}**。

```
var computedProperty: String {
 if someBool {
 return "I'm a mighty pirate!"
 }
 return "I'm selling these fine leather jackets."
}
```

### 3.7.2 当使用**get {}**, **set {}**, **willSet**, **didSet**, 注意**blocks**区块缩进。

### 3.7.3 尽管可以为**willSet**、**didSet**和**set**创建新值或旧值的自定义名称，但推荐使用默认提供的标准**newValue** / **oldValue**标识符。

```
var storedProperty: String = "I'm selling these fine leather jackets." {
 willSet {
 print("will set to \(newValue)")
 }
 didSet {
 print("did set from \(oldValue) to \(storedProperty)")
 }
}

var computedProperty: String {
 get {
 if someBool {
 return "I'm a mighty pirate!"
 }
 return storedProperty
 }
 set {
 storedProperty = newValue
 }
}
```

### 3.7.4 单例属性声明如下：

```
class PirateManager {
 static let shared = PirateManager()

 /* ... */
}
```

## 3.8 Closures 闭包

**3.8.1** 如果参数的类型是明显的，可以省略类型名称，但是显式的也是可以的。有时通过添加明确的细节，有时通过重复的部分，增强可读性 - 使用你最好的判定方案。

```
// omitting the type
doSomethingWithClosure() { response in
 print(response)
}

// explicit type
doSomethingWithClosure() { response: NSURLResponse in
 print(response)
}

// using shorthand in a map statement
[1, 2, 3].flatMap { String($0) }
```

**3.8.2** 如果指定一个闭包为某个确定的类型时，不需要用圆括号包装它，除非它是必需的。（例如，如果该类型是可选的，或者闭包在另一个闭包中）。

总是将闭包内的参数封装在一组圆括号中:use () 指示没有参数，并使用Void来表示没有返回。

```
let completionBlock: (Bool) -> Void = { (success) in
 print("Success? \(success)")
}

let completionBlock: () -> Void = {
 print("Completed!")
}

//需要以 () 包装的类型
let completionBlock: ((() -> Void)? = nil
```

**3.8.3** 如果没有太多的水平溢出，尽量将参数名称列表保持在与封闭的左括号相同的行上。

确保一行少于160个字符

**3.8.4** 使用尾随闭包语法，除非闭包的含义在没有参数名称的情况下不明显（例如，如果方法有成功和失败闭包的参数）。

```
// trailing closure
doSomething(1.0) { (parameter1) in
 print("Parameter 1 is \(parameter1)")
}

// no trailing closure
doSomething(1.0, success: { (parameter1) in
 print("Success with \(parameter1)")
}, failure: { (parameter1) in
 print("Failure with \(parameter1)")
})
```

## 3.9 Arrays

### 3.9.1 访问数组方式：

1. 通常，避免直接使用下标访问数组。
2. 如果可能的话，使用访问器，如 `.first` 或 `.last`，这是可选的，不会崩溃。
3. 在可能的情况下，优先使用 `for item in items` 语法，而不是 `for i in 0 ..< items.count` 语法。
4. 如果您需要直接访问数组下标，请确保进行适当的边界检查。您可以  
在 `items.enumerated()` 中使用 `for (index, value)` 来获取索引和值。

### 3.9.2 数组操作：

1. 切勿使用 `+ =` 或 `+` 运算符将元素追加/连接到数组。代之以，使  
用 `.append()` 或 `.append(contentsOf:)`，因为这些在Swift的当前状态下性能更高（至  
少就编译而言）。
2. 如果你正在声明一个基于其他数组的数组，并希望保持它不可变，使用 `let myNewArray = [arr1, arr2].joined()`，而非 `myNewArray = arr1 + arr2`。

## 3.10 Error Handling

1. 假设函数`myFunction`应该返回一个字符串，但是，在某些时候它可能会出现错误。一个  
常用的方法是让这个函数返回一个可选的`String?` 如果出现问题，我们将返回`nil`。

```

func readFile(named filename: String) -> String? {
 guard let file = openFile(named: filename) else {
 return nil
 }

 let fileContents = file.read()
 file.close()
 return fileContents
}
//
func printSomeFile() {
 let filename = "somefile.txt"
 guard let fileContents = readFile(named: filename) else {
 print("Unable to open file \(filename).")
 return
 }
 print(fileContents)
}

```

2. 1的替代方案：适当场景使用Swift的try / catch行为来了解失败的原因。

```

//struct定义
struct Error: Swift.Error {
 public let file: StaticString
 public let function: StaticString
 public let line: UInt
 public let message: String

 public init(message: String, file: StaticString = #file, function: StaticString = #function, line: UInt = #line) {
 self.file = file
 self.function = function
 self.line = line
 self.message = message
 }
}

```

```
//应用
func readFile(named filename: String) throws -> String {
 guard let file = openFile(named: filename) else {
 throw Error(message: "Unable to open file named \(filename).")
 }

 let fileContents = file.read()
 file.close()
 return fileContents
}

//
func printSomeFile() {
 do {
 let fileContents = try readFile(named: filename)
 print(fileContents)
 } catch {
 print(error)
 }
}
```

有一些例外，更倾向于使用方案1而不是方案2。当结果在语义上可能为nil而不是在检索结果时出错的情况下，返回 optional 值更有意义，而不是使用错误处理。

一般而言，假如方法可以失败，并且失败的原因并不能使用返回值类型明显的描述出来，则使用方案2更为合适。

## 3.11 使用guard语法

**3.11.1** 一般来说，我们倾向于使用“早期**return**”策略，而不是在**if**语法嵌套代码。对这个用例使用**guard**语句通常是有帮助的，且可以提高代码的可读性。

```
// PREFERRED
func eatDoughnut(at index: Int) {
 guard index >= 0 && index < doughnuts.count else {
 // return early because the index is out of bounds
 return
 }

 let doughnut = doughnuts[index]
 eat(doughnut)
}

// NOT PREFERRED
func eatDoughnut(at index: Int) {
 if index >= 0 && index < doughnuts.count {
 let doughnut = doughnuts[index]
 eat(doughnut)
 }
}
```

### 3.11.2 当展开optionals时，首选guard语句而不是if语句来减少代码中嵌套缩进量。

```
// PREFERRED
guard let monkeyIsland = monkeyIsland else {
 return
}
bookVacation(on: monkeyIsland)
bragAboutVacation(at: monkeyIsland)

// NOT PREFERRED
if let monkeyIsland = monkeyIsland {
 bookVacation(on: monkeyIsland)
 bragAboutVacation(at: monkeyIsland)
}

// EVEN LESS PREFERRED
if monkeyIsland == nil {
 return
}
bookVacation(on: monkeyIsland!)
bragAboutVacation(at: monkeyIsland!)
```

### 3.11.3 当不涉及展开optionals时，决定是使用if语句还是使用guard语句，最重要的是要注意代码的可读性。

这里有很多可能的情况，比如依赖于两个不同的布尔值；一个涉及多重比较的复杂逻辑语句等。所以一般来说，用你最好的判断来编写可读性高的代码。如果你不确定用 `guard` 还是 `if`，或者无法判定哪个可读性更高，或者看起来可读性相同，则推荐使用 `guard`。

```
// an `if` statement is readable here
if operationFailed {
 return
}

// a `guard` statement is readable here
guard isSuccessful else {
 return
}

// double negative logic like this can get hard to read - i.e. don't do this
guard !operationFailed else {
 return
}
```

### 3.11.4 如果在两种不同状态之间进行选择，则使用**if**语句更有意义而不是**guard**语句。

```
// PREFERRED
if isFriendly {
 print("Hello, nice to meet you!")
} else {
 print("You have the manners of a beggar.")
}

// NOT PREFERRED
guard isFriendly else {
 print("You have the manners of a beggar.")
 return
}

print("Hello, nice to meet you!")
```

### 3.11.5 当前上下文中，当且仅当失败时退出流程，应使用**guard**。

下面是一个使用两个**if**语句更有意义，而不是使用两个 **guard** 的例子。我们有两个不相关的条件，不应该相互阻塞。

```
if let monkeyIsland = monkeyIsland {
 bookVacation(onIsland: monkeyIsland)
}

if let woodchuck = woodchuck, canChuckWood(woodchuck) {
 woodchuck.chuckWood()
```

### 3.11.6 通常，我们可能遇到一种情况：即需要使用**guard**语法来展开多个**optionals**。

一般来说，如果处理每个解包的失败结果是相同的（例如，一个 `return` , `break` , `continue` , `throw` , `@noescape` ），则将解包含并为单个 `guard` 语法。

```
// combined because we just return
guard let thingOne = thingOne,
 let thingTwo = thingTwo,
 let thingThree = thingThree else {
 return
}

// separate statements because we handle a specific error in each case
guard let thingOne = thingOne else {
 throw Error(message: "Unwrapping thingOne failed.")
}

guard let thingTwo = thingTwo else {
 throw Error(message: "Unwrapping thingTwo failed.")
}

guard let thingThree = thingThree else {
 throw Error(message: "Unwrapping thingThree failed.")
}
```

### 3.11.7 使用`guard`语法不要单行展示。

```
// PREFERRED
guard let thingOne = thingOne else {
 return
}

// NOT PREFERRED
guard let thingOne = thingOne else { return }
```

## 4. 文档、注释

### 4.1 文档

1. 如果一个函数比一个简单的O (1) 操作更复杂，那么通常来说，应该考虑为该函数添加一个`doc`文档注释，因为可能有一些信息无法通过方法签名轻易的获取到。
2. 如果函数的实现有什么奇怪的方式，无论技术上是否有趣、棘手、不明显等，这都应该添加`doc`记录。
3. 复杂的类、结构、枚举、协议和属性等均应为其添加文档。
4. 所有公开的函数、类、属性、常量、结构、枚举、协议等，也应为其添加文档。（他们的签名/名称不能显式的描述其意义/功能）

5. 务必查看[苹果文档](#)中有关Swift注释标记中提供的全部功能。

**4.1.1 160个字符的列限制。(如代码)**

**4.1.2** 即使文档注释占用一行，也使用注释块(*/\* \*/*)。

**4.1.3** 不要在每个附加行加上\*。

**4.1.4** 使用新的-**parameter**语法，而不是旧的：**param**：语法（确保使用小写**parameter**，而不是大写**Parameter**）。

按住 `option` 键，点击你写的方法，使用快速帮助检测注释是否正确。

```
class Human {
 /**
 * This method feeds a certain food to a person.

 * - parameter food: The food you want to be eaten.
 * - parameter person: The person who should eat the food.
 * - returns: True if the food was eaten by the person; false otherwise.
 */
 func feed(_ food: Food, to person: Human) -> Bool {
 // ...
 }
}
```

**4.1.5** 如果你要**doc**记录一个方法的参数/返回/抛出，即使一些文档最终有些重复（最好的文档看起来也不完善）。有时候，如果只有一个参数需要证明文件，那么只需在说明中提到它就可以了。

**4.1.6** 对于复杂的类，推荐用一些可能的例子来描述类的用法。

注：`markdown` 语法在Swift注释文档中是有效的。换行符、列表因此是适合的。

```

/**
Feature Support

This class does some awesome things. It supports:

- Feature 1
- Feature 2
- Feature 3

Examples

Here is an example use case indented by four spaces because that indicates a
code block:

let myAwesomeThing = MyAwesomeClass()
myAwesomeThing.makeMoney()

Warnings

There are some things you should be careful of:

1. Thing one
2. Thing two
3. Thing three
*/
class MyAwesomeClass {
 /* ... */
}

```

#### 4.1.7 设计代码注释部分，使用` `

```

/**
This does something with a `UIViewController`, perchance.
- warning: Make sure that `someValue` is `true` before running this function.
*/
func myFunction() {
 /* ... */
}

```

#### 4.1.8 书写**doc**文档时，倾向于简介明了。

## 4.2 其它注释指南：

### 4.2.1 //之后，注释之前，添加一个空格。

### 4.2.2 总是注释自身所在的行。

### 4.2.3 当使用**// MARK**时，无论如何，在注释之后添加一换行符。

```
class Pirate {

 // MARK: - instance properties

 private let pirateName: String

 // MARK: - initialization

 init() {
 /* ... */
 }

}
```

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48

# raywenderlich.com官方Swift规范

以Swift3为基准更新

该规范指南可能不同于你所看到过的，因为它将着重聚焦于代码打印和web展示的可读性。

尽管我们有很多作者合作开发教程，但创建该规范，保证了我们书籍、教程、start代码的美观和一致性。

我们的目标即清晰、一致、简洁。

## 正确性

1. 努力保证代码编译无警告。该条规范影响了众多样式规范，诸如使用 `#selector` 而非字符串别名。

## 命名

具有可描述性的、统一的命名方式可提高软件编码易读性。建议使用 [API Design Guidelines](#) 中列出的Swift代码规范。一些要点罗列如下：

1. 努力做到：代码无处不清晰可读。
2. 代码的清晰可读优先级大于简洁。
3. 使用驼峰命名法。
4. 类型（协议）使用大写命名方案（大写驼峰）、小写用于其它情境。
5. 涵盖所需单词而去掉不需要的单词。（额）
6. 基于功能命名，而非类型。
7. 有时会补偿弱类型信息。
8. 流利使用
9. 工厂方法以 `make` 开头。
10. 方法命名凸现其功能：
  - i. 动态方法命名辅以 `ed`、`ing` 规则。（非变异版non-mutating）
  - ii. 名词方法命名辅以 `formX` 规则。（变异版mutating）
  - iii. 布尔类型需如断言，具有可读性。
  - iv. 协议用以描述事务的，应以名词修饰。
  - v. 协议用以描述能力功能的，应以 `-able`、`-ible` 等修饰。
11. 使用不会让专家感到意外的术语或混淆初学者。
12. 一般情况下，禁用缩写。

13. 使用名称先例。
14. 优先使用方法和属性，而非函数。（preferring methods and properties to free functions）
15. 外壳首字母缩写词和首字母缩写词统一向上或向下。
16. 给共享相同含义的方法提供相同的基本名称。
17. 避免返回类型的重载。
18. 选择可以充当文档的参数名称。
19. 标记闭包和元组参数。
20. 充分利用默认参数。

## Prose

在prose区提及方法时，明确性是至关重要的。要引用方法名称，请使用最简单的表单方式。

1. 书写无参方法名。如：接下来，你需要调用方法：`addTarget`。
2. 书写有参并只含参数名的方法名。如：接下来，你需要调用方法：`addTarget(_:action:)`。
3. 书写完整方法名，含参数名和类型。如：接下来，你需要调用方法：`addTarget(_: Any?, action: Selector?)`。

上述3种方式，若用在 `UIGestureRecognizer` 时，推荐使用方法1。

附：你可以使用Xcode具备的jump bar功能查看具有参数的方法。

```
C AppDelegate
P window
M application(_: didFinishLaunchingWithOptions:)
M applicationWillResignActive(_:)
M applicationDidEnterBackground(_:)
M applicationWillEnterForeground(_:)
M applicationDidBecomeActive(_:)
M applicationWillTerminate(_:)
```

## 类前缀

Swift中的类型由包含它们的模块自动进行命名空间配置，不需要人为添加诸如RW之类的类前缀。如果来自不同模块的两个名称发生冲突，则可以通过在类型名称前加上模块名称来消除歧义。但是，只有在可能出现混淆时才指定模块名称，这种情况应该很少见。

```
import SomeModule

let myClass = MyModule.UsefulClass()
```

# Delegates 委托代理

在创建自定义委托方法时，未命名的第一个参数应该是委托源。（UIKit包含了很多这样的例子。）

```
//推荐
func namePickerView(_ namePickerView: NamePickerView, didSelectName name: String)
func namePickerViewShouldReload(_ namePickerView: NamePickerView) -> Bool
```

```
//不推荐
func didSelectName(namePicker: NamePickerController, name: String)
func namePickerShouldReload() -> Bool
```

## 使用上下文推断类型功能

使用编译器推断上下文的功能来编写较短，清晰的代码。（更多：类型推断Type Inference）

```
//Preferred
let selector = #selector(viewDidLoad)
view.backgroundColor = .red
let toView = context.view(forKey: .to)
let view = UIView(frame: .zero)
```

```
//Not Preferred
let selector = #selector(ViewController.viewDidLoad)
view.backgroundColor = UIColor.red
let toView = context.view(forKey: UITransitionContextViewKey.to)
let view = UIView(frame: CGRect.zero)
```

## 泛型

泛型类型参数应该是可描述性的，大写驼峰命名。当类型名称没有有意义的关系或角色时，请使用传统的单个大写字母，例如T，U或V.

```
//Preferred

struct Stack<Element> { ... }
func write<Target: OutputStream>(to target: inout Target)
func swap<T>(_ a: inout T, _ b: inout T)
```

```
//Not Preferred

struct Stack<T> { ... }

func write<target: OutputStream>(to target: inout target)
func swap<Thing>(_ a: inout Thing, _ b: inout Thing)
```

## 语言

使用美式英语拼写来匹配Apple的API。

```
//Preferred

let color = "red"
```

```
//Not Preferred

let colour = "red"
```

## 代码组织

使用扩展（extensions）将代码组织到逻辑功能块中。每个扩展应该用 `// MARK` 标记： - 注释以保持组织良好。

## 协议一致性

特别是，在向模型添加协议时，最好为协议方法添加一个单独的扩展（extensions）。这将相关的协议方法分组在一起，并且可以方便将某协议相关的方法添加到类中。

由于编译器不允许您在派生类中重新声明协议一致性，因此并不总是需要复制基类的协议扩展组。如果派生类是一个最终类，并且需覆盖少数几个方法，则尤其如此。何时保留扩展组由作者决定。

对于UIKit视图控制器，考虑将生命周期，自定义访问器和IBAction分组在单独的类扩展中。

```
//Preferred

class MyViewController: UIViewController {
 // class stuff here
}

// MARK: - UITableViewDataSource
extension MyViewController: UITableViewDataSource {
 // table view data source methods
}

// MARK: - UIScrollViewDelegate
extension MyViewController: UIScrollViewDelegate {
 // scroll view delegate methods
}
```

```
//Not Preferred

class MyViewController: UIViewController, UITableViewDataSource, UIScrollViewDelegate
{
 // all methods
}
```

## 无用的代码

未使用（死）的代码，包括Xcode模板代码和占位符注释应该被删除。一个例外情况是，当您的教程或书籍指示用户使用注释代码时。

与实现仅仅调用超类的教程没有直接关联的方法也应该被删除。这包括任何空的/未使用的UIApplicationDelegate方法。

```
//Preferred

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
 return Database.contacts.count
}
```

```
//Not Preferred

override func didReceiveMemoryWarning() {
 super.didReceiveMemoryWarning()
 // Dispose of any resources that can be recreated.
}

override func numberOfSections(in tableView: UITableView) -> Int {
 // #warning Incomplete implementation, return the number of sections
 return 1
}

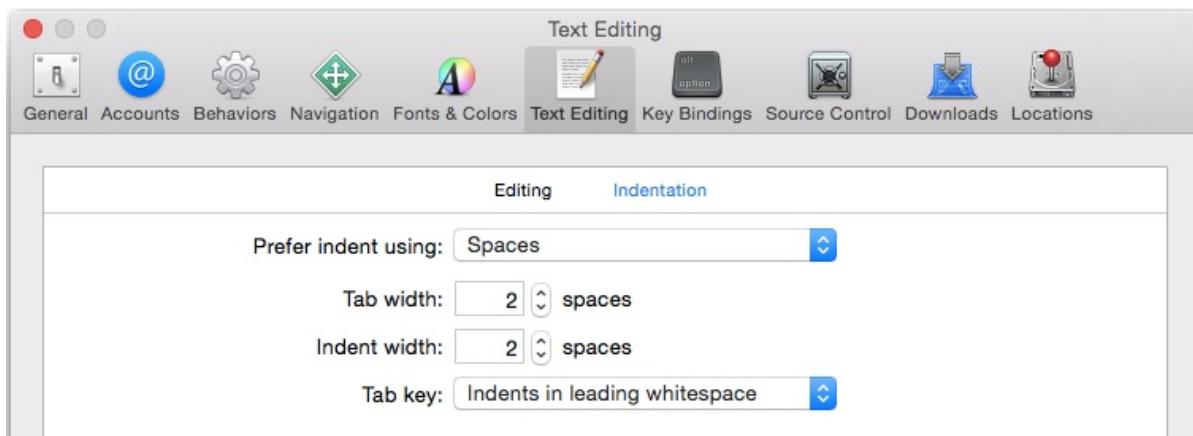
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
 // #warning Incomplete implementation, return the number of rows
 return Database.contacts.count
}
```

## 最小化引入模块

保持导入模块最小化。例如，在导入Foundation时就不要导入UIKit。（不解？？？）

## 空格

- 缩进使用2个空格，而不是制表符来节省空间并有助于防止换行。请确保在Xcode中和项目设置中设置此首选项，如下所示：



- 方法大括号和其他大括号（if / else / switch / while等）始终与语句在同一行打开，但在新行上关闭。
- 提示：可以通过选择一些代码（或`⌘+A`选择全部），然后`ctrl+I`（或菜单中的`Editor \ Structure \ Re-Indent`）来重新缩进。一些Xcode模板代码将会有4个空格的标签硬编码，所以这是解决这个问题的好方法。

```
//Preferred

if user.isHappy {
 // Do something
} else {
 // Do something else
}
```

```
//Not Preferred

if user.isHappy
{
 // Do something
}
else {
 // Do something else
}
```

- 方法之间应该有一条空白线条，以帮助进行视觉清晰度和组织;方法中的空格应该分离功能，但在方法中有太多的部分通常意味着您应该将其重构为若干方法。
- `:` 的左侧总是没有空间,右侧有一个空格。例外是三元操作符`? :`，未命名参数`(_:)`的`#selector`语法，空字典`[:]`。

```
//Preferred

class TestDatabase: Database {
 var data: [String: CGFloat] = ["A": 1.2, "B": 3.2]
}
```

```
//Not Preferred

class TestDatabase : Database {
 var data :[String:CGFloat] = ["A" : 1.2, "B":3.2]
}
```

- 一行代码应包裹大约70个字符。硬性限制是故意不指定的。
- 避免在行末尾空白。
- 在每个文件的末尾添加一个换行符。

## 注释

需要时，使用注释来解释为什么特定的代码会执行某些操作。注释必须保持最新或删除。

避免注释块代码内嵌于代码内，因为代码应尽可能具备可读性。例外情况：这不适用于用于生成文档的注释。

## 类和结构体

### 如何在两者中进行选择，使用哪一个？

- 请记住，结构体具有价值语义(值)。将结构体用于没有身份的事物。

包含[a, b, c]的数组与另一个包含[a, b, c]的数组完全相同，并且它们完全可以互换。无论您使用第一个数组还是第二个数组都没关系，因为它们表示完全相同的东西。这就是为什么数组是结构体的原因。

- 类具有引用语义（址）。对具有身份标识或特定生命周期的事物使用类。

你会将一个人建模为一个类，因为两个人对象是两个不同的东西。仅仅因为两个人有相同的名字和出生日期，并不意味着他们是同一个人。但是这个人的出生日期可以是一个结构体，因为1950年3月3日的日期与1950年3月3日的任何其他日期对象相同。日期本身没有身份。

有时候，应该是结构体，但需要符合AnyObject或者将已经相关模型建模为类了（NSDate，NSSet）。尽可能严格遵循以下这些准则。

### 示例：

这是一个风格良好的类定义：

```

class Circle: Shape {
 var x: Int, y: Int
 var radius: Double
 var diameter: Double {
 get {
 return radius * 2
 }
 set {
 radius = newValue / 2
 }
 }

 init(x: Int, y: Int, radius: Double) {
 self.x = x
 self.y = y
 self.radius = radius
 }

 convenience init(x: Int, y: Int, diameter: Double) {
 self.init(x: x, y: y, radius: diameter / 2)
 }

 override func area() -> Double {
 return Double.pi * radius * radius
 }
}

extension Circle: CustomStringConvertible {
 var description: String {
 return "center = \(centerString) area = \(area())"
 }
 private var centerString: String {
 return "(\(x), \(y))"
 }
}

```

上述代码演示了以下样式准则：

1. 指定属性，变量，常量，参数声明和其他语句的类型；在冒号后面有空格但不在之前，例如 `x: Int` 和 `Circle : Shape`。
2. 如果共享一个共同的目的/上下文，则可以在一行上定义多个变量和结构体。如 `x`、`y` 圆心坐标。
3. 缩进 `getter` 和 `setter` 定义和属性观察者。
4. 无需显式添加默认值，如 `internal`。同样，重写方法时不要重复访问修饰符。
5. 在扩展中组织额外的功能（例如打印输出）。
6. 一些非共享的实现细节需要隐藏，可使用 `private` 修饰符来修饰之。例如在扩展中的 `centerString`。

## self 的使用

为了简洁起见，避免使用 `self`，因为 Swift 不需要 `self`，即可访问对象的属性或调用其方法。

仅在编译器要求时才使用 `self`（在 `@escaping` 闭包中，或者在初始化程序中用于消除参数中的属性歧义）。换句话说，如果它没有自我编译，那就省略它。

## 计算属性

为简明起见，如果计算属性是只读的，则省略 `get` 子句。`get` 子句仅在提供 `set` 子句时才是必需的。

```
//Preferred
var diameter: Double {
 return radius * 2
}
```

```
//Not Preferred
var diameter: Double {
 get {
 return radius * 2
 }
}
```

## Final

将 `类` 或 `类成员` 标记为教程中的 `Final` 选项可能会分散主题，因此不是必需的。尽管如此，有时使用 `Final` 选项可以澄清你的意图，并且是值得的。在下面的示例中，`Box` 具有特定用途，其派生类中的定制不是意图所需的。因此将 `Box` 标记为 `Final`。

```
// 使用此Box类将任何泛型类型转换为引用类型。
final class Box<T> {
 let value: T
 init(_ value: T) {
 self.value = value
 }
}
```

## 函数声明

在一行上保持最短函数声明，包括左括号：

```
func reticulateSplines(spline: [Double]) -> Bool {
 // reticulate code goes here
}
```

对于具有长签名的函数，在适当的点添加换行符并在随后的行上添加一个额外的缩进：

```
func reticulateSplines(spline: [Double], adjustmentFactor: Double,
 translateConstant: Int, comment: String) -> Bool {
 // reticulate code goes here
}
```

## 闭包表达式

- 只有在参数列表末尾有一个闭包表达式的参数时才使用尾随闭包语法。给出闭包参数的描述性名称。

```
//Preferred
UIView.animate(withDuration: 1.0) {
 self.myView.alpha = 0
}

UIView.animate(withDuration: 1.0, animations: {
 self.myView.alpha = 0
}, completion: { finished in
 self.myView.removeFromSuperview()
})
```

```
//Not Preferred
UIView.animate(withDuration: 1.0, animations: {
 self.myView.alpha = 0
})

UIView.animate(withDuration: 1.0, animations: {
 self.myView.alpha = 0
}) { f in
 self.myView.removeFromSuperview()
}
```

- 对于上下文清晰的单表达式闭包，使用隐式返回

```
attendeeList.sort { a, b in
 a > b
}
```

1. 使用尾随闭包的链接方法应该清晰易读并且在上下文中易于阅读。关于间距，换行符以及何时使用命名与匿名参数的决定由作者决定。例：

```
let value = numbers.map { $0 * 2 }.filter { $0 % 3 == 0 }.index(of: 90)

let value = numbers
 .map {$0 * 2}
 .filter {$0 > 50}
 .map {$0 + 10}
```

## 类型

若可以，始终优先使用Swift的自带类型。Swift提供了与Objective-C的桥接方案，因此您仍然可以根据需要使用全套方法。

```
//Preferred
let width = 120.0 // Double
let widthString = (width as NSNumber).stringValue // String

//Not Preferred
let width: NSNumber = 120.0 // NSNumber
let widthString: NSString = width.stringValue // NSString
```

在Sprite Kit代码中，使用`CGFloat`，以资避免太多转换使代码更加简洁。

## 常量

常量使用`let`关键字定义，变量使用`var`关键字来定义。如果变量的值不会改变，请始终使用`let`而不是`var`。

提示：一个好的方法是使用`let`来定义所有的东西，如果编译器发出抱怨，将它改为`var`！

您可以使用类型定义常量，而不是在该类型的实例上定义常量。要将类型属性声明为常量，只需使用`static let`即可。以这种方式声明的类型属性通常优于全局常量，因为它们更容易与实例属性区分开来。例：

```
//Preferred
enum Math {
 static let e = 2.718281828459045235360287
 static let root2 = 1.41421356237309504880168872
}

let hypotenuse = side * Math.root2
```

备注：使用无case枚举的优点是它不会意外地被实例化并作为纯名称空间运行。

```
//Not Preferred
let e = 2.718281828459045235360287 // pollutes global namespace
let root2 = 1.41421356237309504880168872

let hypotenuse = side * root2 // what is root2?
```

## 静态方法和变量类型属性

静态方法和类型属性的工作方式与全局函数和全局变量类似，应该谨慎使用。当功能范围限于特定类型或需要与Objective-C进行互操作时，它们非常有用。

## Optionals

将变量和函数返回类型声明为可选的`?`，那么返回为`nil`可以接受的了。

只有当您知道的变量会在使用前进行初始化，才能使用用`!`声明的隐式解包类型进行对象的解包。例如将在`viewDidLoad`中设置的子视图。

当访问一个可选值`?`时，如果该值只被访问一次，或者链中有很多可选项`?`，则使用可选的链接：

```
self.textContainer?.textLabel?.setNeedsDisplay()
```

如果打开一次并执行多个操作更方便，请使用可选绑定：

```
if let textContainer = self.textContainer {
 // do many things with textContainer
}
```

命名可选变量和属性时，避免将它们命名为`optionalString`或`mayView`，因为它们的可选项已经在类型声明中。

对于可选绑定，在适当的时候使用原始名称，而不要使用像unwrappedView或actualLabel这样的名称。

```
//Preferred
var subview: UIView?
var volume: Double?

// later on...
if let subview = subview, let volume = volume {
 // do something with unwrapped subview and volume
}
```

```
//Not Preferred
var optionalSubview: UIView?
var volume: Double?

if let unwrappedSubview = optionalSubview {
 if let realVolume = volume {
 // do something with unwrappedSubview and realVolume
 }
}
```

## 延迟初始化

考虑使用延迟初始化对对象生命周期进行更精细的纹理控制。对于延迟加载视图的UIViewController尤其如此。您可以立即调用{}()的闭包或调用私有工厂方法。例：

```
lazy var locationManager: CLLocationManager = self.makeLocationManager()

private func makeLocationManager() -> CLLocationManager {
 let manager = CLLocationManager()
 manager.desiredAccuracy = kCLLocationAccuracyBest
 manager.delegate = self
 manager.requestAlwaysAuthorization()
 return manager
}
```

备注：

- [unowned self] 此处不是必需的，没有造成循环引用问题。
- Location manager有一个弹出UI的副作用，要求用户获得许可。所以细粒度控制在这里很有意义。

## 类型推断

优先使用紧凑的代码，并让编译器推断单个实例类型（常量的或变量的）。类型推断也适用于小型（非空）数组和字典。如果需要，请指定特定类型，例如`CGFloat`或`Int16`。

```
//Preferred
let message = "Click the button"
let currentBounds = computeViewBounds()
var names = ["Mic", "Sam", "Christine"]
let maximumWidth: CGFloat = 106.5
```

```
//Not Preferred
let message: String = "Click the button"
let currentBounds: CGRect = computeViewBounds()
let names = [String]()
```

## 为空数组和词典键入类型注释

对于空数组和字典，请使用类型注释。（对于分配给大型多行文字的数组或字典，请使用类型注释。）

```
//Preferred:
var names: [String] = []
var lookup: [String: Int] = [:]
```

```
//Not Preferred
var names = [String]()
var lookup = [String: Int]()
```

备注：遵循本指南意味着选择描述性名称比以前更重要。

## 语法糖

在整个泛型语法上使用类型声明的快捷版本。

```
//Preferred
var deviceModels: [String]
var employees: [Int: String]
var faxNumber: Int?
```

```
//Not Preferred
var deviceModels: Array<String>
var employees: Dictionary<Int, String>
var faxNumber: Optional<Int>
```

## 函数VS方法

自由函数，不附加到类或类型，应该谨慎使用。如果可能，倾向于使用方法而不是自由函数。这有助于可读性和可发现性。

自由函数在与任何特定类型或实例不相关时都是最合适的选择。

```
//Preferred

let sorted = items.mergeSorted() // easily discoverable
rocket.launch() // acts on the model
```

```
//Not Preferred

let sorted = mergeSort(items) // hard to discover
launch(&rocket)
```

Free Function Exceptions

```
let tuples = zip(a, b) // feels natural as a free function (symmetry)
let value = max(x, y, z) // another free function that feels natural
```

## 内存管理

代码（甚至非生产，教程演示代码）不应产生引用循环。分析您的对象并使用 `weak` 和 `unowned` 来防止强引用产生。或者，使用值类型（结构，枚举）完全防止循环问题产生。

## 延长对象的生存期

使用 `[weak self]` 和 `guardSelf = self else {return}` 延长对象的生命周期。`[weak self]` 比 `[unowned self]` 更受青睐，在`self`超越闭包并不是很明显的情况下，显式延长寿命优于可选的展开。

```
//Preferred

resource.request().onComplete { [weak self] response in
 guard let strongSelf = self else {
 return
 }
 let model = strongSelf.updateModel(response)
 strongSelf.updateUI(model)
}
```

```
//Not Preferred

// might crash if self is released before response returns
resource.request().onComplete { [unowned self] response in
 let model = self.updateModel(response)
 self.updateUI(model)
}
```

```
//Not Preferred

// 在更新模型和更新UI之间可能会发生释放
resource.request().onComplete { [weak self] response in
 let model = self?.updateModel(response)
 self?.updateUI(model)
}
```

## Access Control

教程中的所有的访问控制会分散主题内容并且不是必需的。但是，适当使用 `private` 和 `fileprivate` 可以增加清晰度并促进封装。如果可能的话，更倾向于使用 `private` 而非 `fileprivate`。

使用扩展可能需要您使用 `fileprivate`。

只有在需要完整的访问控制规范时才明确地使用 `open`，`public` 和 `internal`。

使用访问控制作为主要属性说明符。在访问控制之前唯一可添加的指定符是：`static`，或者属性：`@IBAction`、`@IBOutlet`、`@discardableResult`。

```
//Preferred:

private let message = "Great Scott!"

class TimeMachine {
 fileprivate dynamic lazy var fluxCapacitor = FluxCapacitor()
}
```

```
//Not Preferred:

fileprivate let message = "Great Scott!"

class TimeMachine {
 lazy dynamic fileprivate var fluxCapacitor = FluxCapacitor()
}
```

## Control Flow

优先使用 `for-in`，而非 `for`、`while` 样式的循环。

```
//Preferred:

for _ in 0..<3 {
 print("Hello three times")
}

for (index, person) in attendeeList.enumerated() {
 print("\(person) is at position #\(index)")
}

for index in stride(from: 0, to: items.count, by: 2) {
 print(index)
}

for index in (0...3).reversed() {
 print(index)
}
```

```
//Not Preferred:

var i = 0
while i < 3 {
 print("Hello three times")
 i += 1
}

var i = 0
while i < attendeeList.count {
 let person = attendeeList[i]
 print("\(person) is at position #\(i)")
 i += 1
}
```

## Golden Path

使用条件编码时，代码的左边距应该是“golden”或“happy”的路径。也就是说，不要嵌套if语句。多个返回语句都可以。`guard` 声明是为此而构建的。

```
//Preferred:

func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {

 guard let context = context else {
 throw FFTError.noContext
 }
 guard let inputData = inputData else {
 throw FFTError.noInputData
 }

 // use context and input to compute the frequencies
 return frequencies
}
```

```
//Not Preferred:

func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {

 if let context = context {
 if let inputData = inputData {
 // use context and input to compute the frequencies

 return frequencies
 } else {
 throw FFTError.noInputData
 }
 } else {
 throw FFTError.noContext
 }
}
```

如果有多个选项可以通过 `guard let` 或 `if let` 来解包，则尽可能使用复合版本尽量减少嵌套。例：

```
//Preferred:

guard let number1 = number1,
 let number2 = number2,
 let number3 = number3 else {
 fatalError("impossible")
}
// do something with numbers
```

```
//Not Preferred:

if let number1 = number1 {
 if let number2 = number2 {
 if let number3 = number3 {
 // do something with numbers
 } else {
 fatalError("impossible")
 }
 } else {
 fatalError("impossible")
 }
} else {
 fatalError("impossible")
}
```

## Failing Guards

Guard语句需要以某种方式退出。通常，这应该是简单的一行语句，如`return`，`throw`，`break`，`continue`和`fatalError()`。应该避免使用大的代码块。如果多个退出点需要清理代码，请考虑使用`defer`代码块以避免清理代码重复。

## 分号

Swift在代码中的每个语句后均不强制要求分号。只有在您希望将多条语句合并到一行时，才需要它们。

不要在一行上用分号分隔写多个语句。

```
//Preferred:

let swift = "not a scripting language"
```

```
//Not Preferred:

let swift = "not a scripting language";
```

备注：Swift与JavaScript非常不同，（JavaScript）忽略分号通常被认为是不安全的。

## 括号

条件语句周围的括号不是必需的，应该省略。

```
//Preferred:

if name == "Hello" {
 print("World")
}
```

```
//Not Preferred:

if (name == "Hello") {
 print("World")
}
```

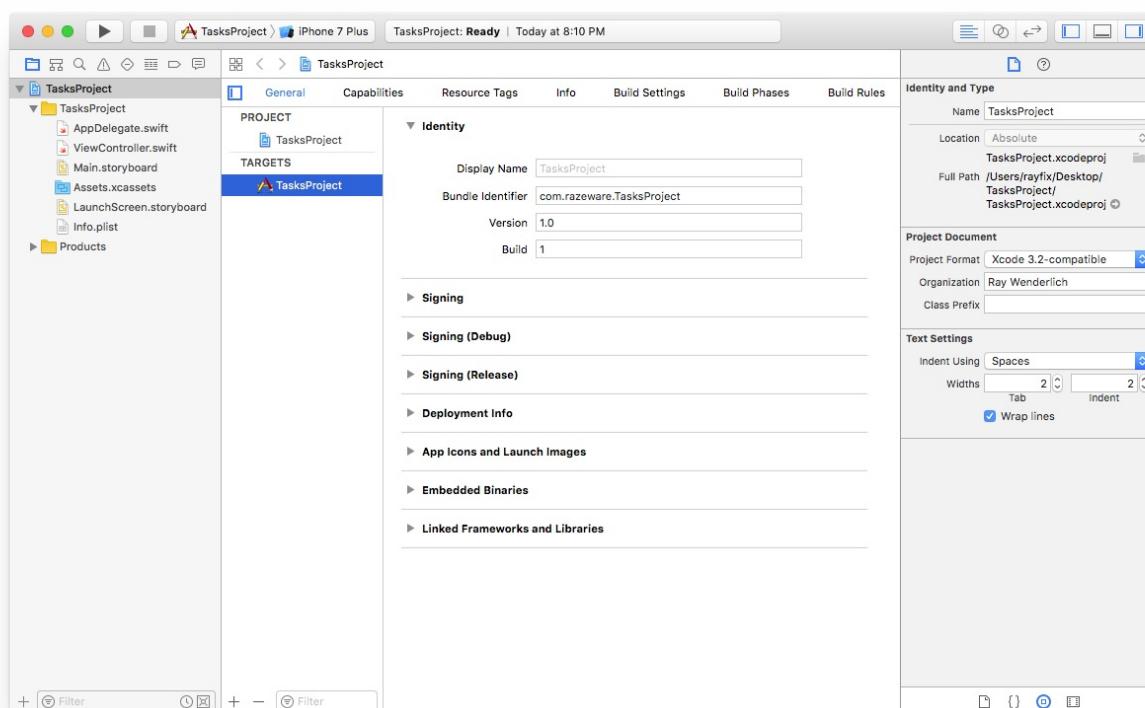
在较大的表达式中，可选的括号有时会使代码更清晰。

```
//Preferred:

let playerMark = (player == current ? "X" : "0")
```

## Organization and Bundle Identifier（作为参考）

如果涉及Xcode项目，应将Organization设置为Ray Wenderlich，并将Bundle Identifier设置为com.razeware.TutorialName，其中TutorialName是教程项目的名称。



## Copyright 声明（作为参考）

以下版权声明应包含在每个源文件的顶部：

```

/// Copyright (c) 2018 Razeware LLC
///
/// Permission is hereby granted, free of charge, to any person obtaining a copy
/// of this software and associated documentation files (the "Software"), to deal
/// in the Software without restriction, including without limitation the rights
/// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
/// copies of the Software, and to permit persons to whom the Software is
/// furnished to do so, subject to the following conditions:
///
/// The above copyright notice and this permission notice shall be included in
/// all copies or substantial portions of the Software.
///
/// Notwithstanding the foregoing, you may not use, copy, modify, merge, publish,
/// distribute, sublicense, create a derivative work, and/or sell copies of the
/// Software in any work that is designed, intended, or marketed for pedagogical or
/// instructional purposes related to programming, coding, application development,
/// or information technology. Permission for such use, copying, modification,
/// merger, publication, distribution, sublicensing, creation of derivative works,
/// or sale is expressly withheld.
///
/// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
/// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
/// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
/// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
/// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
/// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
/// THE SOFTWARE.

```

## 笑脸（参考）

笑脸是raywenderlich.com网站非常突出的风格特征！正确的微笑表示编码主题的巨大快乐和兴奋是非常重要的。因为它代表了使用ASCII艺术可以捕捉到的最大笑容，所以使用了方括号。一个右括号）创造一个半心半意的微笑，因此不是优选的。

## 参考链接：

- [The Swift API Design Guidelines](#)
- [The Swift Programming Language](#)
- [Using Swift with Cocoa and Objective-C](#)
- [Swift Standard Library Reference](#)

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook修订时间：2018-03-08 10:26:48



- [UED规范](#)

# UED规范

Copyright © iOS开发手册小组 all right reserved · powered by Gitbook 修订时间：2018-03-08 10:26:48

- [CodeReview规范](#)
  - [review重点总结](#)

# CodeReview规范

目前CodeReview规范仍未成型，但review执行重点均放在如下几方面，后续会逐步完善。

## review重点总结

1. 业务逻辑
2. 代码冗余检测
3. 代码注释
4. 内存泄露
5. 代码规范
6. 其它

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 产品管理

## 产品管理

- 项目把控
- PM
- UED

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 项目把控

## 项目把控

- Team ambition、Scrum
- checklist，项目复盘

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- PM

## PM

- 术语解读

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

- 术语解读
  - 目录
  - 一、SMART原则
    - 1.0 图解
    - 1.1 概念
    - 1.2 其它
  - 二、5W1H分析法
    - 2.0 图解
    - 2.1 表格
    - 2.2 概念
    - 2.3 示例

## 术语解读

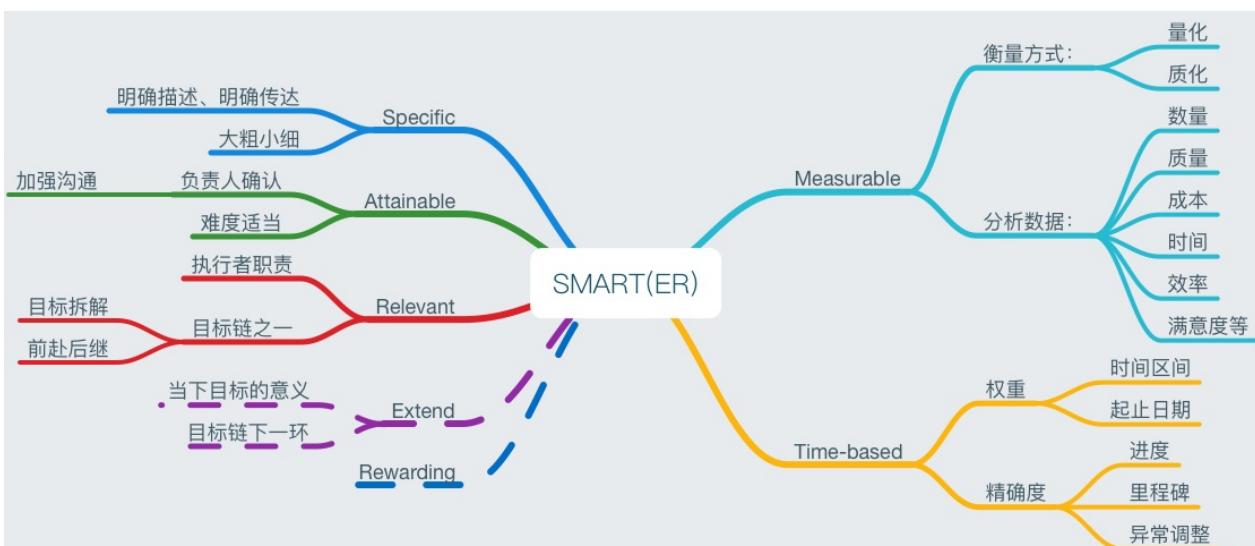
- 描述事务，尽可能以最简单扼要的方式阐述出来，以下是自己做的总结，无示例，自己也在学习中。

## 目录

1. SMART原则
2. 5W1H分析法

## 一、SMART原则

### 1.0 图解



## 1.1 概念

### 1. S : Specific

- 目标需用 明确的 语言表达清晰。
- 目标需能 明确的 传达至相关成员。
- 大粗小细：大目标，粒度大些；小目标，粒度要小。

### 2. M : Measurable

- 目标需是可以以 量化 或 质化 衡量的。
- 一般伴有分析数据。如数量、质量、成本、时间、满意度

### 3. A : Attainable

- 目标需与直接负责人共同确认，确保是可以 实现的 。
- 加强沟通。

### 4. R : Relevant

- 目标是与职责直接 相关联 的。
- 目标不是孤岛，它的实现也是具有目的性的。

### 5. T : Time-based

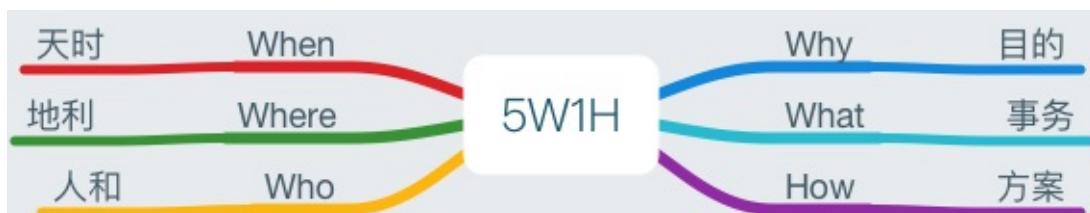
- 根据目标的权重，确定其时间区间，起止日期。
- 目标进度、里程碑、异常变更调整等是其属性。
- 时间把控的精确度和经验成正比

## 1.2 其它

1. 目标管理 首次出现在管理学大师Peter Drucker的《管理实践》一书中。
2. SMART目的：有效进行成员组织和目标的制定和控制。

## 二、5W1H分析法

### 2.0 图解



### 2.1 表格

5H1W	诠释	为什么	能否改善	怎么改善
Why(目的)	什么目的	为什么是这种目的(目的之目的)	有无更优目的	更优目的是什么
事务(What)	事务=操作流程+对象	事务之于目的，不可或缺的原因	之于目的，是否还有更高效的替代事务	更优事务具体是什么
地点(Where)	地利环境	地利之利，利在哪方面	是否还有更优之地点	更优之地，利在哪方面
时间(When)	天时：起止时间，时间区间	天时，选择的原因	是否有更优的时间选择	更优时间，如何选择
责任者(Who)	人和	选择的人员，优势在哪	是否还有更合适人选	如何确定更合适人员
How	如何做	这么做，优势在哪	是否有更优方案	更优方案是什么

- 附：
- What之于Why：事务必须是目的之最精简，最不可或缺。
  - 两者需来回验证
- 天时、地利、人和：在What之后选择
- 基于Why，What以及其它因素确定之后，确定How的具体方案。

## 2.2 概念

1. 5Ws and 1H：六何分析法。
2. 对选定项目、工序、操作:都要从原因Why、对象What、地点Where、时间When、人员Who、方法How六个方面提出并进行思考。

## 2.3 示例

- 列提纲
- what:写什么（选材）
- why:写的目的，欲表达的东西（立意）
- how:怎么写（立意如何表达；选材如何架构：详略，细节运用等）

如何架构已有的What，来达到Why的表达



- [UED](#)

## UED

- 工具使用：PaintCode、Sketch

Copyright © iOS开发手册小组 all right reserved，powered by Gitbook 修订时间：2018-03-08 10:26:48

- 其它
- 

- 其它

## 其它

- [Gitbook初探](#)

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook 修订时间 : 2018-03-08 10:26:48

## 1. 安装Homebrew

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

### 1. 安装nodejs

```
brew update
brew install node
```

#### 1. gitbook 命令

```
npm install gitbook-cli -g
```

#### 2. 初始化 : `gitbook init`

- i. 创建 `README.md` 、 `SUMMARY.md` 文件（若无）
- ii. 根据 `SUMMARY.md` 目录结构生成子目录。（细化部分仍需手工来做）

#### 1. 本地测试 : `gitbook serve`

- i. `gitbook build` 生成gitbook静态网页
- ii. 实时预览 : `localhost:4000`

#### 2. 插件 : 见链接

##### 1. 参考链接 :

- i. <http://www.chengweiyang.cn/gitbook/basic-usage/README.html>
- ii. <http://www.tuicool.com/articles/zee2ui>

Copyright © iOS开发手册小组 all right reserved , powered by Gitbook修订时间 : 2018-03-08 10:26:48