iOS App重构策略

1，代码规范方面

可以参考国际化大公司的代码风格和样式来书写代码，下面是一些比较好的代码书写规范

1.1

Objective-C 规范指南（纽约时报移动开发团队）

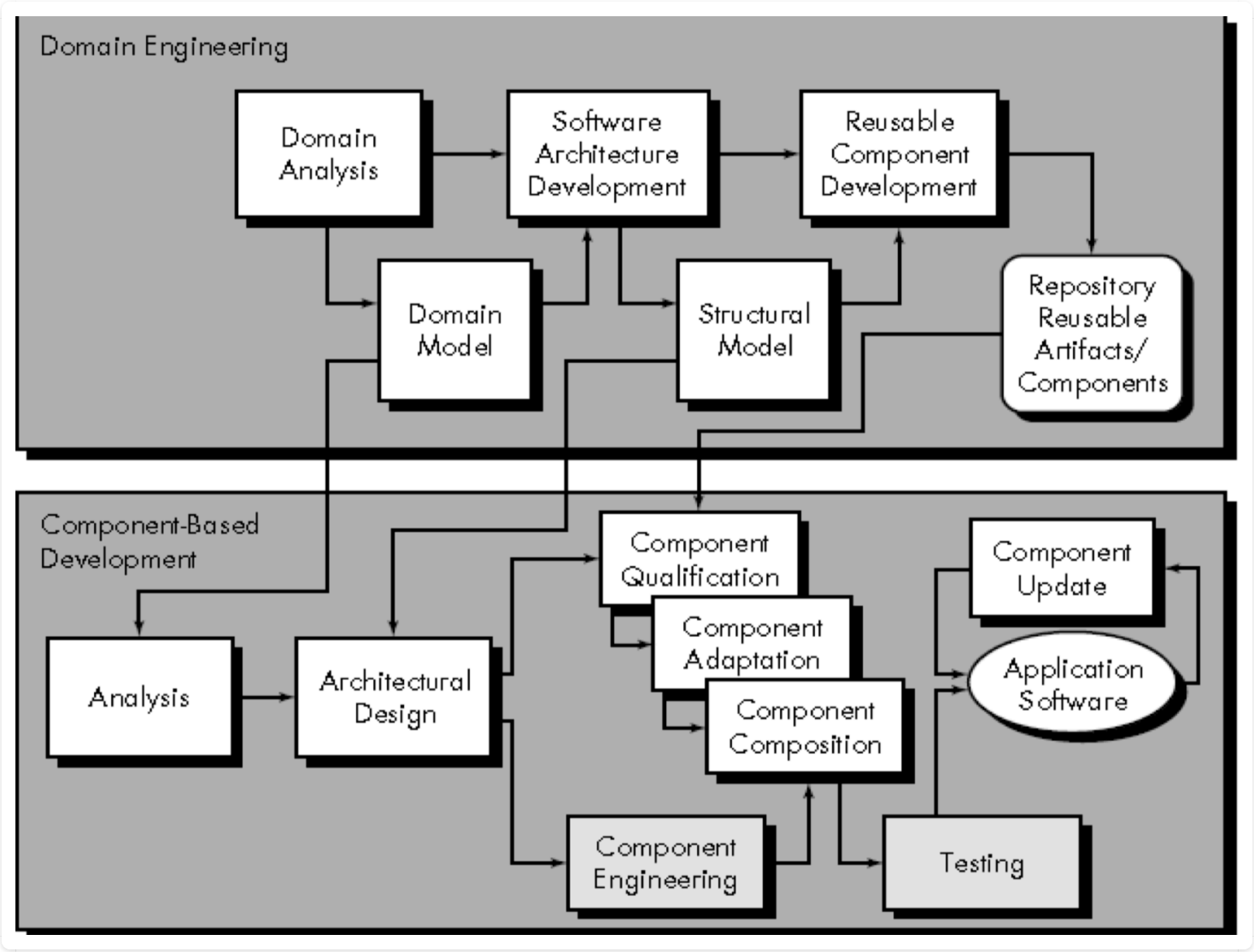
如果感觉我们的不太符合你的口味，可以看看下面的风格指南：

- Google
- GitHub
- Adium
- Sam Soffes
- CocoaDevCentral
- Luke Redpath
- Marcus Zarra

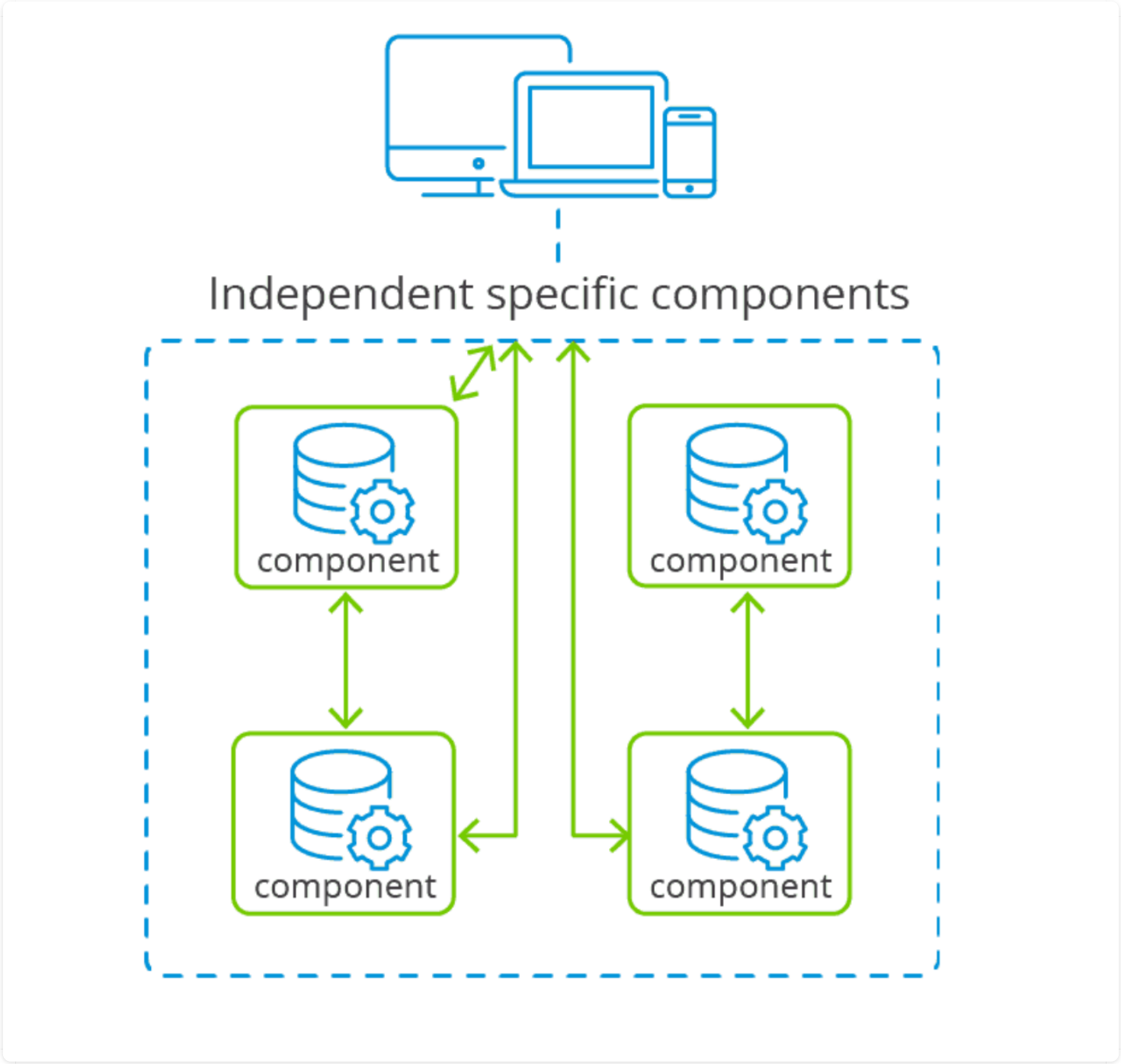
2，重构前使用tool去分析查找问题

- 1，使用AppCode自带inspect code功能分析并对于代码进行对应的优化
- 2，使用codeBeat工具来进行分析，支持Gitlab，github，bitbucket等平台
- 3，使用oclint来提升代码质量，具体教程可以参照此链接

3，采用组件化开发的模式进行项目开发



当今应用程序开发领域的一个共同趋势是微服务和基于组件的体系结构的模式，如果操作正确，它将为您提供最大的灵活性和可伸缩性。事实上，mendix公司将Mendix应用程序构建为云原生和容器化的，为创建复杂的应用程序创建了一个理想的生态系统，具有适当的组件化级别和随时间增长应用程序的灵活性。随着市场上云计算的采用(以及整个云生态系统的Mendix)，组织可以很容易地利用这些现代模式和基于组件的体系结构，其复杂性要小得多。



组件化利弊：

优点

- 1，组件可独立运行，提高的代码的复用性，组件化的颗粒度越细，可复用度就越高。
- 2，当组件库的数量足够庞大时，项目只需要组合组件即可完成大部分的开发工作。
- 3，组件化后项目的代码结构更加清晰，追踪问题、修复bug、增加需求更方便
- 4,不同业务组件相互独立，明确团队开发的业务边界，增加团队协作效率 ### 缺点
- 1，增加开发人员的学习成本
- 2，增加了代码的冗余，组件化颗粒度越细，中间代码越多

- 3，增加了项目的复杂度，复杂度越高越容易出问题

除了组件化方案外，可以采取后台流行的微服务设计模式来构建项目，或者面向服务的设计模式来设定，对于业务划分要明确。

4，软件架构

1，问题：目前公司项目多以MVC的架构模式为主，MVC存在很多问题，主要问题在于以下：

- 1，臃肿的ViewController，viewController的责任过多，做了很多事，不符合Single responsibility原则
- 2，代码复用性差
- 3，可测试性差 随着项目迭代周期越久，代码维护的成本就越高，到一定程度后很难维护。

2， 替代方案：

MVVM+ReactiveObjc的架构模式
优势明显：代码复用性较高，业务和UI隔离，可测试性强（针对某些业务逻辑可进行对应的单元测试）
实现思路也是一种基于服务的思路进行，viewModel和model、viewController实现双路绑定，而具体的业务则以protocol来定义业务协议，在impl中完成对应的方法，将对应的service绑定到对应的viewModel上。

MVVM主要体现在业务和UI隔离，主要利于团队协作开发，UI与业务隔离，会提升开发效率，而且可测试性强，更利于早期发现bugs并修正

当然还有其他的架构可选，但是Ele架构，Viper架构，但是Ele架构还未正式在生产环境中被人实践过，而Viper的架构过于复杂，对于团队人员的要求高，因此不太适合。
关于为什么要写软件测试？可以看下面的链接为什么你要写测试用例
iOS架构

5，软件性能

1，app启动时间 为什么要优化app启动时间呢？App启动时间带来的用户体验建议应用程序应该在发布动画时间内发布，以便在用户看来是即时发布的。发布动画的时间在iPhone上是400毫秒，在iPad上是500毫秒。你应该把发射时间定在接近这个时间。优化app启动时间，app启动的前世今生，使用timer profile优化app启动时间

2，内存 优化

- 1，iOS不使用交换文件，但支持虚拟内存。如果应用程序在内存中保存了大量数据，以便进行随机访问，那么您希望将其组织为mapfile，而不是使用malloc()将其加载到RAM中。最简单的方法是调用NSData initWithContentsOfMappedFile:
- 2， 当你在没有明确分配的情况下实例化NSString这样的对象时，避免堆叠自动释放的对象——通常直到你的应用退出。这种技术的广泛使用可能会导致RAM中的大量垃圾。使用NSString initWithContentsOfFile:所以你可以稍后释放它而不是NSString stringWithContentsOfFile:。同样的规则也适用于UIImage imageNamed: -这不推荐用于图像加载。
- 3， 处理内存警告时卸载不必要的资源。即使你不能卸载所有UIViewControllers的所有东西调用[super didReceiveMemoryWarning]。默认情况下，这将释放一些资源，比如非前端视图上的UI控件。如果无法处理这个事件，iOS可能会认为你的应用程序应该被杀死。
- 4，考虑到动画视图转换动画的有限使用，比如flip transition动画会在执行时导致RAM使用量激增。这个特性非常简洁，应该在许多情况下使用，但它可能会在重载的多任务环境中触发内存警告。我们特别强烈建议避免对OpenGL视图进行动画化。
- 5，在设备上测试你的内存占用，使用仪器进行测试。最有用的工具是分配、泄漏和活动监视器。在模拟器上测试在大多数情况下是不相关的，因为它的内存占用倾向于完全不同。测试之后，您就可以计算出应用程序的每个部分使用了多少RAM，瓶颈在哪里，以及如何进行优化。Note: 有用的链接App内存来龙去脉 ## 3，网络
- 1，优化DNS查找时间，一般而言初始化连接的时候便是DNS查找，如果App是网络重量化产品DNS查找时间会减慢它。查找时间是一个基本的DNS服务器配置函数，最后的链接是一个可追溯到目标IP地址的路由函数，使用CDNs来最小化延迟是一个最佳实践

```
; <<>> DiG 9.10.6 <<>> www.sina.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4921
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.sina.com.                  IN      A

;; ANSWER SECTION:
www.sina.com.           28      IN      CNAME   us.sina.com.cn.
us.sina.com.cn.         28      IN      CNAME   wwwus.sina.com.
wwwus.sina.com.         28      IN      A       66.102.251.33

;; Query time: 315 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Mon Sep 10 06:19:48 CST 2018
;; MSG SIZE  rcvd: 105
```

尽量减少应用程序使用的唯一域的数量。由于路由通常的工作方式，多个域是不可避免的。最可能的情况是，您将需要每一个为以下: a,身份管理(登录、注销、配置文件) b,数据服务(API端点) c,CDN(图像和其他静态工件)可能需要其他域(例如，服务视频、上传仪表数据、特定于子组件的数据服务、服务广告，甚至特定于国家的地理定位)。如果子域名数量上升到两位数，就会引起担忧。

- 2,SSL握手时间实践 a，最小化App连接数量 b，请求完成后不要关闭HTTP/S连接，增加header connection 使得keep alive c，使用域名分片 这允许你可以使用相同的socket即使连接是针对多个主机
- 3，网络类型实践 a，使可变的网络是可用的，例如3G，4G，wifi b，在失败的情况下，在一个随机的指数增长的延迟之后重新尝试。例如第一次失败后，1s后再去尝试connect，第二次失败后2s后再去连接，不过要记得设置最大自动尝试链接次数 c，在强迫刷新之间建立一个最短的时间 当用户请求显式刷新时，不要立即触发请求。”相反，检查请求是否已经挂起，或者上一次尝试的时间间隔是否小于阈值。如果是，不要发送请求。 d，使用reachability来发现网络上的变化对于下个工作状态 使用可达性发现网络状态的任何变化。使用指示器向用户显示任何不可用之处。毕竟，没有互联网接入并不是你的错。通过让用户知道潜在的连接问题，你可以避免被指责到你的应用上。 e，不要缓存状态 总是使用网络敏感任务的最新值，要么通过回调来知道何时触发请求，要么在发出请求之前进行显式检查 f，基于网络类型去下载content 例如如果你有一个图片去展示，不要总是去下载原图或者高清图片。总是基于设备类型去下载合适的图 g，在WiFi网络上，预取你认为用户在一段时间后需要的内容。随后使用此缓存内容。喜欢突发下载，使用后关闭网络电台。这将有助于节省电池。 h，如果适用，当网络可用时，支持离线同步存储。通常，网络缓存就足够了。但如果你需要更多结构化数据，使用本地文件或CoreData总是首选

4，掉帧和响应速度 使用CADisplaylink来检测drop frame

深入分析
提升App响应速度策略

5，tableView的滑动流畅

优化tableView滑动流畅策略

6 开发tips

6.1，使用面向service的方式来轻量化AppDelegate，首先创建一个面向service的AppDelegate类，它持有一系列服务，例如push notification，崩溃日志收集服务

```
// SOAppDelegate.h
#import <UIKit/UIKit.h>
@interface SOAppDelegate : UIResponder <UIApplicationDelegate>
// The list of services that will be integrated into our application lifecycle
- (NSArray *)services;
// Main window reference
@property (strong, nonatomic) UIWindow *window;
@end
```

下面是它的实现

```
// SOAppdelegate.m
#import "SOAppDelegate.h"
@implementation SOAppDelegate
// By default, we'll have no services in the delegate
- (NSArray *)services {
return nil;
}
#pragma mark - UIApplication Lifecycle
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions
id<UIApplicationDelegate> service;
// Loop through the current services and proxy the delegate call
for(service in self.services){
if ([service respondsToSelector:@selector(application:didFinishLaunchi
[service application:application didFinishLaunchingWithOptions:lau
} }
return YES; }
// U Can do it more here
```

创建一个push notification service

```
// PushNotificationService.h
#import <Foundation/Foundation.h>
@interace PushNotificationService : NSObject <UIApplicationDelegate>
+ (instancetype)sharedInstance;
@end
```

对于push notification服务而言，它只做了二个事情，注册通知（app启动后）和增加push notification handler

```
// PushNotificationService.m
#import "PushNotificationService.h"
@implementation PushNotificationService
// Singleton initializer
+ (instancetype)sharedInstance {
static id _sharedInstance = nil;
static dispatch_once_t _onceToken;
dispatch_once(&_onceToken, ^{
_sharedInstance = [[[self class] alloc] init];
});
return _sharedInstance;
}
// Tap into the application launch sequence and register for remote
// notifications.
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions
// Register for remote notifications
[application registerForRemoteNotificationTypes:
UIRemoteNotificationTypeBadge |
UIRemoteNotificationTypeSound |
UIRemoteNotificationTypeAlert];
// If we launched from a remote notification, we'll pass the handler
// to the notification handler method.
if (launchOptions[UIApplicationLaunchOptionsRemoteNotificationKey]) {
[self application:application didReceiveRemoteNotification:launchOptio
}
return YES;
```

最后一步就是替换AppDelegate.h用如下的代码

```
// AppDelegate.h
#import "SOAppDelegate.h"
@interface AppDelegate : SOAppDelegate
@end
```

实现很简单

```
// AppDelegate.m
#import "PushNotificationService.h"
@implementation AppDelegate
// The services are only initialized once, via their singleton accessors
- (NSArray *)services {
static NSArray * _services;
static dispatch_once_t _onceTokenServices;
dispatch_once(&_onceTokenServices, ^{
_services = @[[PushNotificationService sharedInstance]];
});
return _services;
}
```

6.2，尽可能使得VC轻量化

6.3，不要在ViewController中写动画逻辑，采用一个独立的动画类来替代，例如viewcontroller的 transition也是

6.4，不要写自定义init方法 因为如果你的视图控制器被重新指定为XIB或storyboard，init方法永远不会被调用

6.5，ViewController响应UI相关的事件来自OS，例如 屏幕旋转或者低内存，这或许会触发view的再布局

6.6， 不要用代码在视图控制器中手工创建UI。不要在视图控制器中实现所有UI、xib或者storyboard来创建，或者将其分组件在view中写

6.7，隔离dataSource和Delegate从ViewController中抽离

6.8 推荐创建baseViewController携带一些普遍的setup，然后让其他VC继承它。

6.9 使用categories创建ViewController重用的代码

6.10，抽离页面跳转逻辑从ViewController

6.11 使用IBAnalyzer在不运行应用程序或编写单元测试的情况下找到常见的xib和storyboard相关问题IBAnalyzer

6.12，很多时候我们都会用宏，宏有很强大的作用，因此被广为传颂

作为一个iOS经常使用block，为了打破循环引用，可能很多时候我们都会这样做
距离在我们的视图控制器中有这样的一个模型,模型在block内持有了viewcontroller，因此这就是循环引用。

- （void）setupModel {
Model *model = [Model new];
model.dataChnaged = NSString *title {
self.label.text = title;
};
self.model = model;
}
当然我们可以采用如下的代码来消除
- （void）setupModel {
Model *model = [Model new];
__weak typeof(self) weakself = self;
model.dataChanged = NSString *title {
weakself.lable.text = title;
};
self.model = model'
}
我们在weak和strong之间跳舞，当然这样做也无可厚非，但是它容易出错。当新特性被引入并且块的定义变大时，最终会有人在其中使用self。我们不会注意到它什么时候发生的 — — 编译器只在最简单的情况下起作用。这是weakify和strong宏派上用场的一部分。

6.13，使用中间者模式来解耦页面跳转，或者采用coordinator的设计模式来优化

在iOS中我们都熟知MVC.controller是必不可少的，各个控制器之间都有交互，彼此紧密依存。尤其是在后续中加入更多视图控制器的时候，情况会变得很糟糕。
1，从一个视图控制器到另外一个视图的迁移
一般而言，我们点击navigationBar后则替换为另外一个视图控制器，这是一般的做法，看起来没有什么问题，因为能正常工作，但是对于App程序设计而言不是特别好，如果视图与控制器之间彼此依存，应用程序就不能更好的扩展，如果修改视图的变化的方式，也势必要修改对应的代码，依存关系不仅限于控制器，也包括了按钮，按钮间接地与某些视图控制器发生了关联，如果应用程序变大变复杂，依存关系将难以控制，随之而来的是大量难以理解的视图迁移逻辑。这时候我们需要一种机制去减少不同控制器与按钮间的交互，以降低整体结构的耦合，提高复用性和扩展性。像视图控制器只有的协调控制器会有助于此，但其作用不是协调视图和模型，它要协调不同视图控制器，已完成正确的视图迁移。
2，使用中介者模式来协调视图迁移
中介者模式指的是用一个对象来封装一组对象之间的交互逻辑。中介者通过避免对象间显式的相互引用来增进不同对象间的松耦合，因此对象之间的交互可以集中在一个地方控制，对象之间的依存关系减少。

```
//  CoordinatorController.h
//  VetNXKit
//
//  Created by Marc Zhao on 2017/11/26.
//  Copyright © 2017年 VetNX Cooporation. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "FirstViewController.h"
#import "SecondViewController.h"

@interface CoordinatorController : NSObject
{
    @private
    FirstViewController *firstViewController;
    SecondViewController *secondViewController;



}
@property (nonatomic,readonly) FirstViewController *firstViewController;
@property (nonatomic,readonly) SecondViewController *secondViewController;
+ (CoordinatorController *)sharedInstance;
- (IBAction)requestViewChangeByObject:(id)object;

@end
```

6.14，尽可能的把警告当成错误来对待，尽早地消除bug

这里面有2种类型的警告要特殊处理
编译的警告
隐式的属性合成：因为属性现在已经自动合成了，所以这不是错误，除非明确说要手动合成属性
未使用的参数/函数/变量：在写代码的时候这个很烦人，因为显然代码还没有完成可以考虑只是非debug编译启用这个选项

6.15 启用完整的ASLR

ASLR指的是地址空间布局随机化，确保了内存中程序的结构和数据被加载到虚拟地址空间中不可预测的位置，这会使得代码执行时逆向更加困难
为了能做相应的黑盒测试或者正确开启应用程序的ASLR，可以使用二进制工具检查
otool

6.16 Clang和静态分析

一般静态分析Xcode都自带的，主要进行分析如下四种问题：
1，逻辑错误：访问空指针或者未初始化的变量
2，内存管理错误：如内存泄露
3，声明错误：从未使用过的变量
4，API调用错误：未包含使用的库等

6.17,使用watchdog来监控系统活动

安装watchdog
`pip install watchdog`
watchdog自带了一个便于使用的命令行工具- watchmodo，打开终端跳到模拟器目录使用watchmedo来监控模拟器目录书中所有文件的变化

7 使用Xcode自动UI测试框架和Appuim来实现自动化测试

为什么要进行自动化测试？提升工作效率以及更好的发现bugs并提升产品质量而已。技术栈上可以使appium+
Appium介绍
Appium开始教程

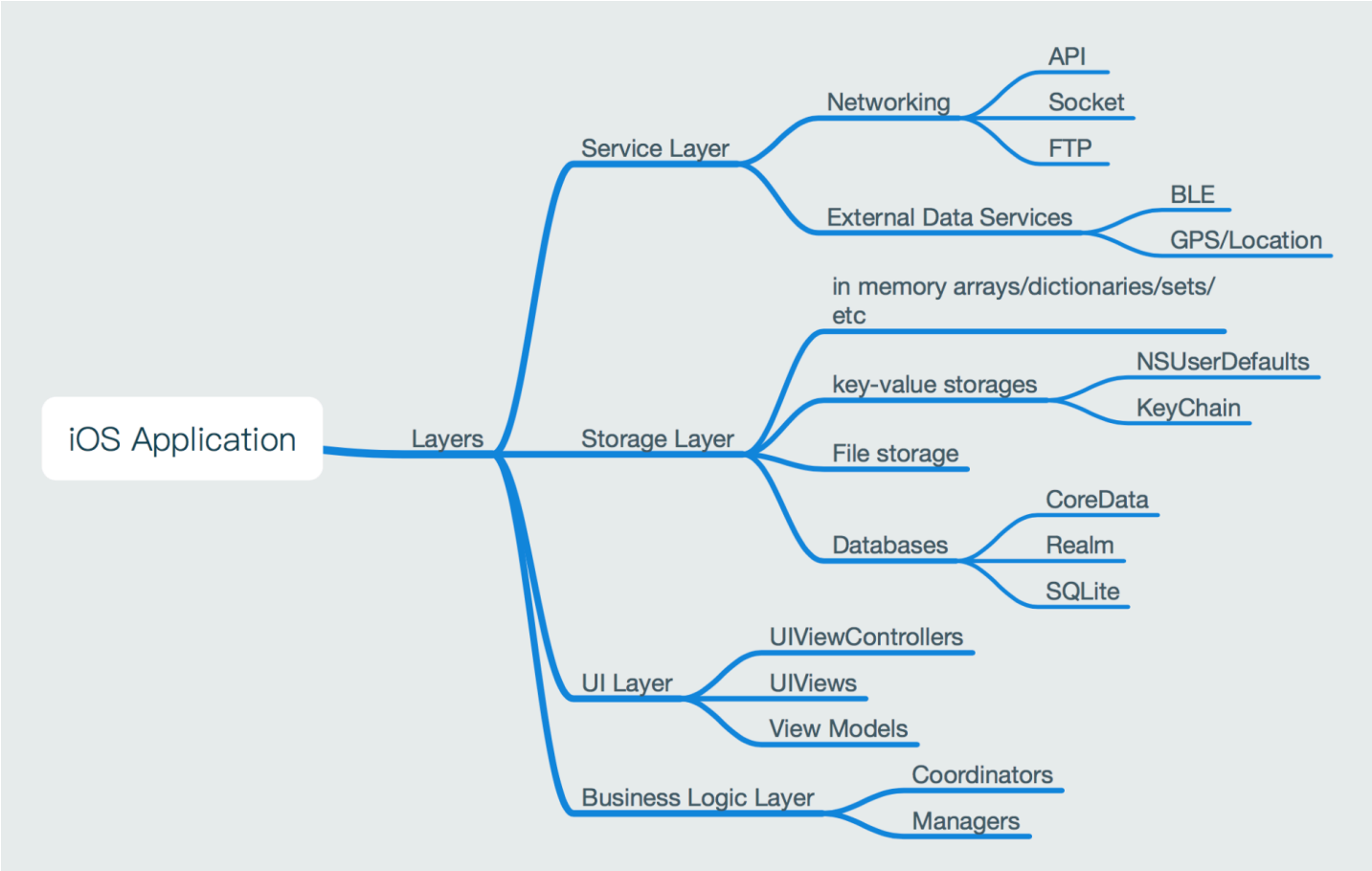
8 Code review

codeReview最佳实践
使用jenkins+Gerrit 进行code review
jenkins可以实现自动部署，构建，自动化测试，单元测试各种功能。
jenkins+gerrit实现自动化code review策略

9 使用fastelane来实现自动化构建和release app

fastlane介绍
fastlane如何使用

10 App使用分层的策略来进行对应的开发



----------Marc Zhao 2021.7.26 ----------