

# 架构分析



作者：刘小壮

博客：<https://www.jianshu.com/u/2de707c93dc4>

Github：<https://github.com/DeveloperErenLiu>

## 组件化架构的由来

随着移动互联网的不断发展，很多程序代码量和业务越来越多，现有架构已经不适合公司业务的发展速度了，很多都面临着重构的问题。

在公司项目开发中，如果项目比较小，普通的单工程+MVC架构就可以满足大多数需求了。但是像淘宝、蘑菇街、微信这样的大型项目，原有的单工程架构就不足以满足架构需求了。

就拿淘宝来说，淘宝在13年开启的“All in 无线”战略中，就将阿里系大多数业务都加入到手机淘宝中，使客户端出现了业务的爆发。在这种情况下，单工程架构则已经远远不能满足现有业务需求了。所以在这种情况下，淘宝在13年开启了插件化架构的重构，后来在14年迎来了手机淘宝有史以来最大规模的重构，将项目重构为组件化架构。

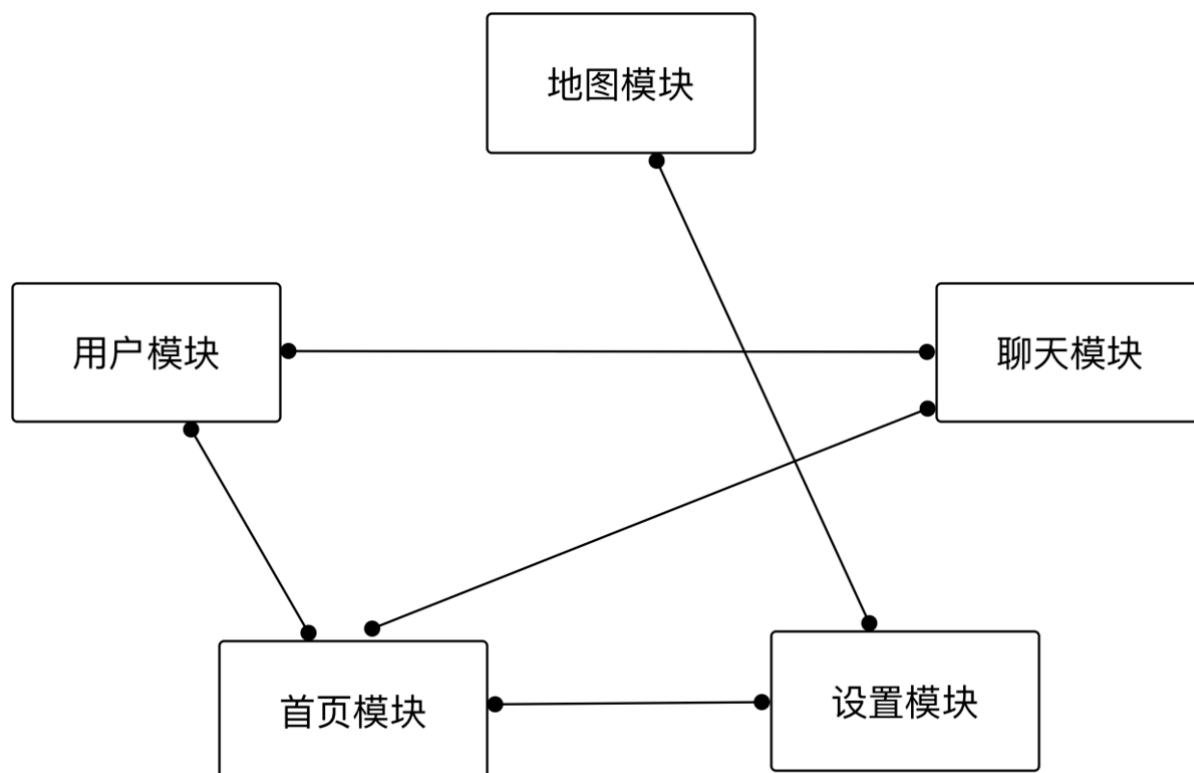
## 蘑菇街的组件化架构

### 原因

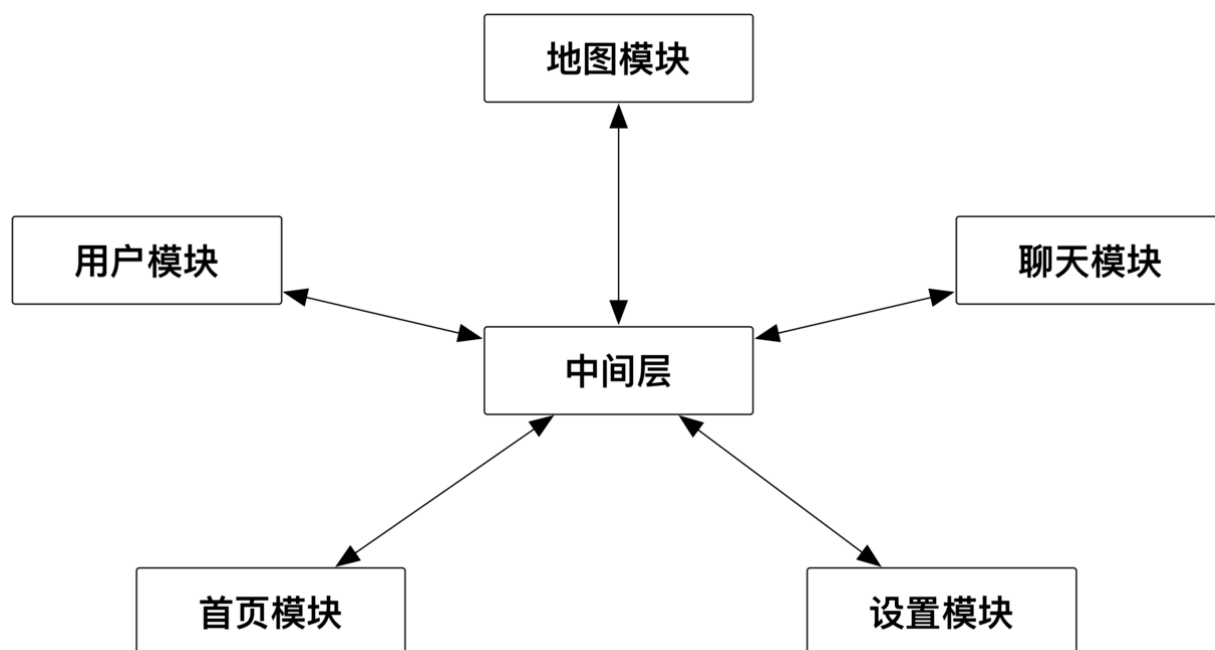
在一个项目越来越大，开发人员越来越多的情况下，项目会遇到很多问题。

- 业务模块间划分不清晰，模块之间耦合度很大，非常难维护。

- 所有模块代码都编写在一个项目中，测试某个模块或功能，需要编译运行整个项目。



为了解决上面的问题，可以考虑加一个中间层来协调各个模块间的调用，所有的模块间的调用都会经过中间层中转。



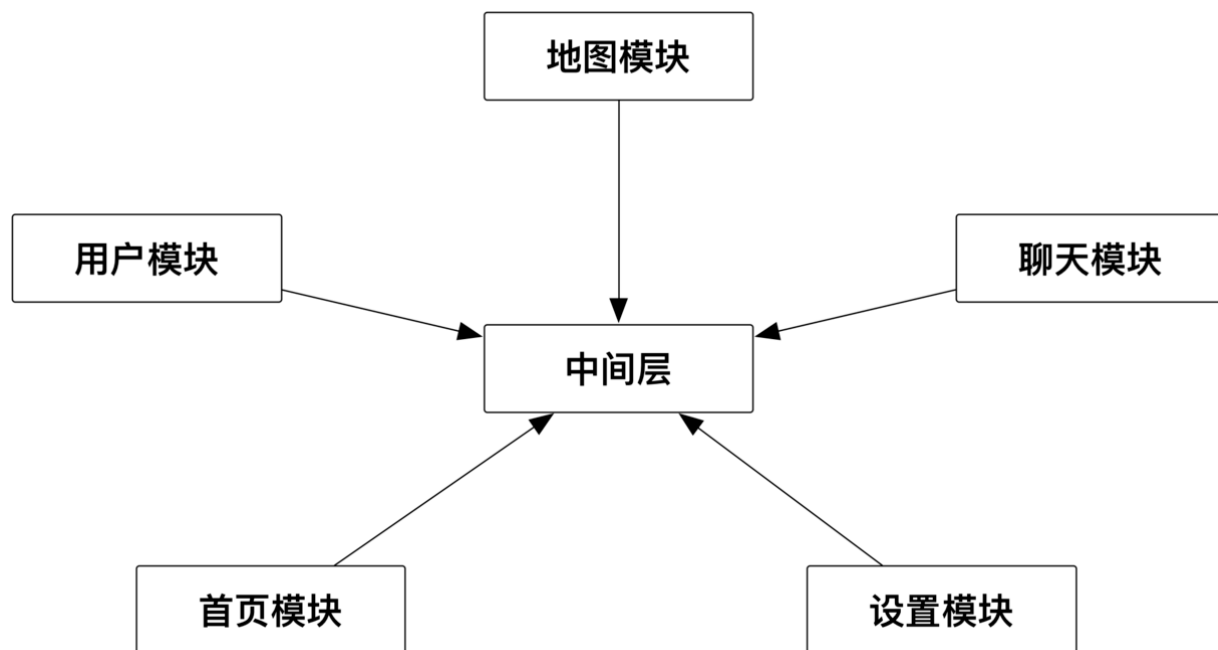
但是发现增加这个中间层后，耦合还是存在的。中间层对被调用模块存在耦合，其他模块也需要耦合中间层才能发起调用。这样还是存在之前的相互耦合的问题，而且本质上比之前更麻烦了。

## 架构改进

所以应该做的是，只让其他模块对中间层产生耦合关系，中间层不对其他模块发生耦合。

对于这个问题，可以采用组件化的架构，将每个模块作为一个组件。并且建立一个主项目，这个主项目负责集成所有组件。这样带来的好处是很多的：

- 业务划分更佳清晰，新人接手更佳容易，可以按组件分配开发任务。
- 项目可维护性更强，提高开发效率。
- 更好排查问题，某个组件出现问题，直接对组件进行处理。
- 开发测试过程中，可以只编译自己那部分代码，不需要编译整个项目代码。
- 方便集成，项目需要哪个模块直接通过 `CocoaPods` 集成即可。

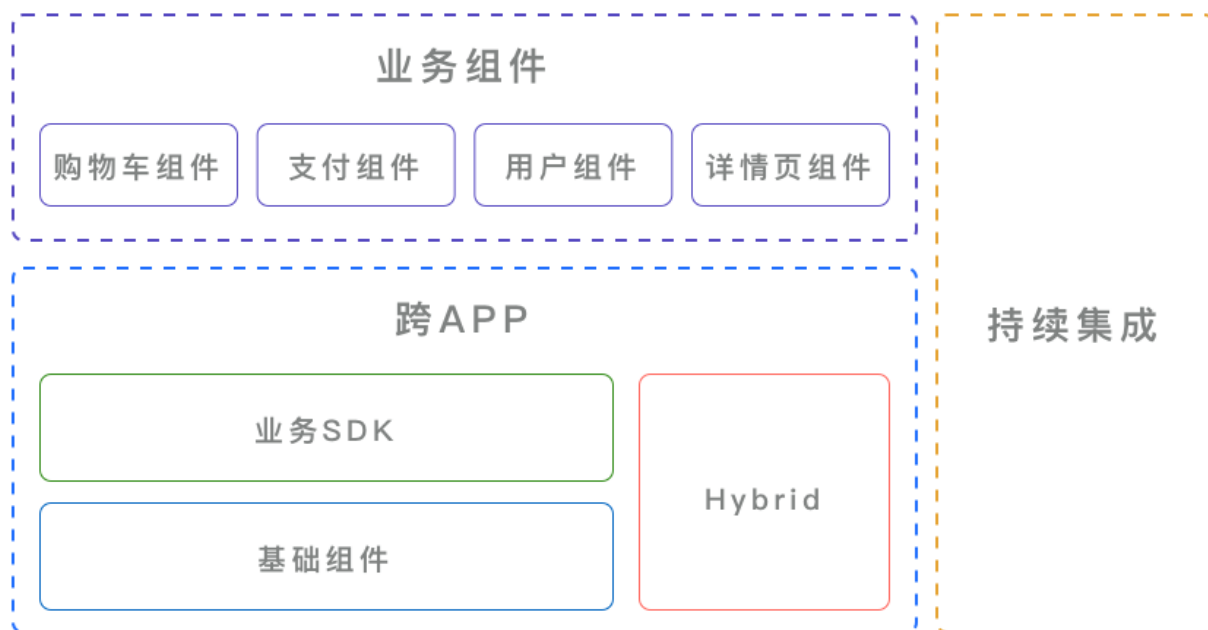


进行组件化开发后，可以把每个组件当做一个独立的app，每个组件甚至可以采取不同的架构，例如分别使用 `MVVM`、`MVC`、`MVCS` 等架构，根据自己的编程习惯做选择。

## MGJRouter方案

蘑菇街通过 `MGJRouter` 实现中间层，由 `MGJRouter` 进行组件间的消息转发，从名字上来说更像是“路由器”。实现方式大致是，在提供服务的组件中提前注册 `block`，然后在调用方组件中通过 `URL` 调用 `block`，下面是调用方式。

### 架构设计



`MGJRouter` 是一个单例对象，在其内部维护着一个“URL -> block”格式的注册表，通过这个注册表来保存服务方注册的 `block`，以及使调用方可以通过 URL 映射出 `block`，并通过 `MGJRouter` 对服务方发起调用。

`MGJRouter` 是所有组件的调度中心，负责所有组件的调用、切换、特殊处理等操作，可以用来处理一切组件间发生的关系。除了原生页面的解析外，还可以根据URL跳转H5页面。

在服务方组件中都对外提供一个 `PublicHeader`，在 `PublicHeader` 中声明当前组件所提供的所有功能，这样其他组件想知道当前组件有什么功能，直接看 `PublicHeader` 即可。每一个 `block` 都对应着一个 `URL`，调用方可以通过 `URL` 对 `block` 发起调用。

```
#ifndef UserCenterPublicHeader_h
#define UserCenterPublicHeader_h

/** 跳转用户登录界面 */
static const NSString * CTBUCUserLogin = @"CTB://UserCenter/UserLogin";
/** 跳转用户注册界面 */
static const NSString * CTBUCUserRegister = @"CTB://UserCenter/UserRegister";
/** 获取用户状态 */
static const NSString * CTBUCUserStatus = @"CTB://UserCenter/UserStatus";

#endif
```

在组件内部实现 `block` 的注册工作，以及 `block` 对外提供服务的代码实现。在注册的时候需要注意注册时机，应该保证调用时 `URL` 对应的 `block` 已经注册。

蘑菇街项目使用 `git` 作为版本控制工具，将每个组件都当做一个独立工程，并建立主项目来集成所有组件。集成方式是在主项目中通过 `CocoaPods` 来集成，将所有组件当做二方库集成到项目中。详细的集成技术点在下面“标准组件化架构设计”章节中会讲到。

## MGJRouter调用

下面代码模拟对详情页的注册、调用，在调用过程中传递 `id` 参数。参数传递可以有两种方式，类似于 **Get** 请求在 `URL` 后面拼接参数，以及通过字典传递参数。下面是注册的示例代码：

```
[MGJRouter registerURLPattern:@"mgj://detail" toHandler:^(NSDictionary
*routerParameters) {
    // 下面可以在拿到参数后，为其他组件提供对应的服务
    NSString uid = routerParameters[@"id"];
}];
```

通过 `openURL:` 方法传入的 `URL` 参数，对详情页已经注册的 `block` 方法发起调用。调用方式类似于 **GET** 请求，`URL` 地址后面拼接参数。

```
[MGJRouter openURL:@"mgj://detail?id=404"];
```

也可以通过字典方式传参，`MGJRouter` 提供了带有字典参数的方法，这样就可以传递非字符串之外的其他类型参数，例如对象类型参数。

```
[MGJRouter openURL:@"mgj://detail" withParam:@{@"id" : @"404"}];
```

## 组件间传值

有的时候组件间调用过程中，需要服务方在完成调用后返回相应的参数。蘑菇街提供了另外的方法，专门来完成这个操作。

```
[MGJRouter registerURLPattern:@"mgj://cart/ordercount"
toObjectHandler:^(NSDictionary *routerParameters){
    return @42;
}];
```

通过下面的方式发起调用，并获取服务方返回的返回值，要做的就是传递正确的 `URL` 和参数即可。

```
NSNumber *orderCount = [MGJRouter objectForURL:@"mgj://cart/ordercount"];
```

## 短链管理

这时候会发现一个问题，在蘑菇街组件化架构中，存在了很多硬编码的 `URL` 和参数。在代码实现过程中 `URL` 编写出错会导致调用失败，而且参数是一个字典类型，调用方不知道服务方需要哪些参数，这些都是个问题。

对于这些数据的管理，蘑菇街开发了一个 `web` 页面，这个 `web` 页面统一来管理所有的 `URL` 和参数，`Android` 和 `iOS` 都使用这一套 `URL`，可以保持统一性。

## 基础组件

在项目中存在很多公共部分的东西，例如封装的网络请求、缓存、数据处理等功能，以及项目中所用到的资源文件。蘑菇街将这些部分也当做组件，划分为基础组件，位于业务组件下层。所有业务组件都使用同一套基础组件，也可以保证公共部分的统一性。

## Protocol方案

### 整体架构



为了解决 MGJRouter 方案中 **URL 硬编码**，以及字典参数类型不明确等问题，蘑菇街在原有组件化方案的基础上推出了 **Protocol** 方案。**Protocol** 方案由两部分组成，进行组件间通信的 **ModuleManager** 类以及 **MGJComponentProtocol** 协议类。

通过中间件 **ModuleManager** 进行消息的调用转发，在 **ModuleManager** 内部维护一张映射表，映射表由之前的 "URL -> block" 变成 "Protocol -> Class"。

在中间件中创建 **MGJComponentProtocol** 文件，服务方组件将可以用来调用的方法都定义在 **Protocol** 中，将所有服务方的 **Protocol** 都分别定义到 **MGJComponentProtocol** 文件中，如果协议比较多也可以分开几个文件定义。这样所有调用方依然是只依赖中间件，不需要依赖除中间件之外的其他组件。

**Protocol** 方案中每个组件需要一个 **MGJModuleImplement**，此类负责实现当前组件对应的协议方法，也就是对外提供服务的实现。在程序开始运行时将自身的 **Class** 注册到 **ModuleManager** 中，并将 **Protocol** 反射为字符串当做 **key**。

**Protocol**方案依然需要提前注册服务，由于 **Protocol** 方案是返回一个 **Class**，并将 **Class** 反射为对象再调用方法，这种方式不会直接调用类的内部逻辑。可以将 **Protocol** 方案的 **Class** 注册，都放在类对应的 **MGJModuleImplement** 中，或者专门建立一个 **RegisterProtocol** 类。

### 示例代码

创建 **MGJUserImpl** 类当做 **User** 组件对外公开的类，并在 **MGJComponentProtocol.h** 中定义 **MGJUserProtocol** 协议，由 **MGJUserImpl** 类实现协议中定义的方法，完成对外提供服务的过程。下面是协议定义：

```
@protocol MGJUserProtocol <NSObject>
- (NSString *)getUserName;
@end
```

**Class** 遵守协议并实现定义的方法，外界通过 **Protocol** 获取的 **Class** 并实例化为对象，调用服务方实现的协议方法。



`ModuleManager` 的协议注册方法，注册时将 `Protocol` 反射为字符串当做存储的 `key`，将实现协议的 `Class` 当做值存储。通过 `Protocol` 取 `Class` 的时候，就是通过 `Protocol` 从 `ModuleManager` 中将 `Class` 映射出来。

```
[ModuleManager registerClass:MGJUserImpl forProtocol:@protocol(MGJUserProtocol)];
```

调用时通过 `Protocol` 从 `ModuleManager` 中映射出注册的 `Class`，将获取到的 `Class` 实例化，并调用 `Class` 实现的协议方法完成服务调用。

```
Class cls = [[ModuleManager sharedInstance]
classForProtocol:@protocol(MGJUserProtocol)];
id userComponent = [[cls alloc] init];
NSString *userName = [userComponent getUserName];
```

## 项目调用流程

蘑菇街是 `MGJRouter` 和 `Protocol` 混用的方式，两种实现的调用方式不同，但大体调用逻辑和实现思路类似。在 `MGJRouter` 不能满足需求或调用不方便时，就可以通过 `Protocol` 的方式调用。

1. 在进入程序后，先使用 `MGJRouter` 对服务方组件进行注册。每个 `URL` 对应一个 `block` 的实现，`block` 中的代码就是组件对外提供的服务，调用方可以通过 `URL` 调用这个服务。
2. 调用方通过 `MGJRouter` 调用 `openURL:` 方法，并将被调用代码对应的 `URL` 传入，`MGJRouter` 会根据 `URL` 查找对应的 `block` 实现，从而调用组件的代码进行通信。
3. 调用和注册 `block` 时，`block` 有一个字典用来传递参数。这样的优势就是参数类型和数量理论上是没有限制的，但是需要很多硬编码的 `key` 名在项目中。

## 内存管理

蘑菇街组件化方案有两种，`Protocol` 和 `MGJRouter` 的方式，但都需要进行 `register` 操作。`Protocol` 注册的是 `Class`，`MGJRouter` 注册的是 `Block`，注册表是一个 `NSMutableDictionary` 类型的字典，而字典的拥有者又是一个单例对象，这样会造成内存的常驻。

下面是对两种实现方式内存消耗的分析：

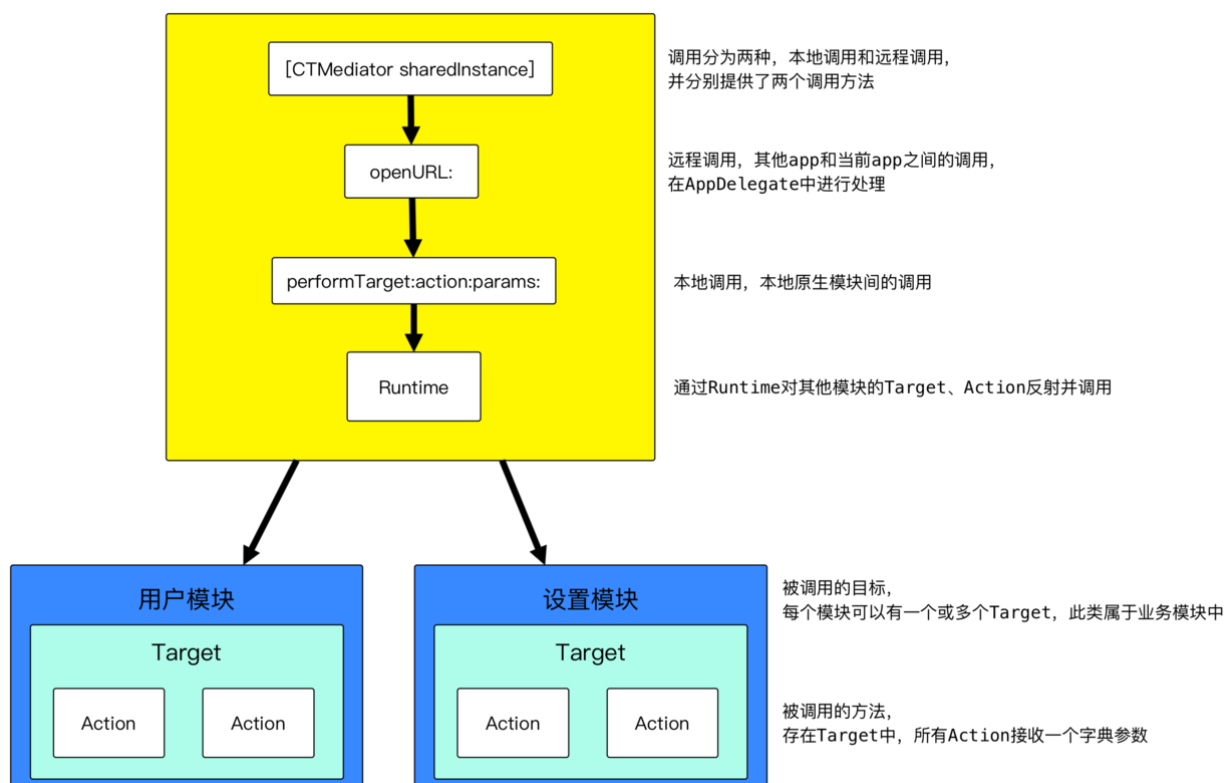
- 首先说一下 `MGJRouter` 方案可能导致的内存问题，由于 `block` 会对代码块内部对象进行持有，如果使用不当很容易造成内存泄漏的问题。  
`block` 自身实际上不会造成很大的内存泄漏，主要是内部引用的变量，所以在使用时就需要注意强引用的问题，并适当使用 `weak` 修饰对应的变量。以及在适当的时候，释放对应的变量。除了对外部变量的引用，在 `block` 代码块内部尽量不要直接创建对象，应该通过方法调用中转一下。
- 对于协议这种实现方式，和 `block` 内存常驻方式差不多。只是将存储的 `block` 对象换成 `Class` 对象。这实际上是存储的类对象，类对象本来就是单例模式，所以不会造成多余内存占用。

## casatwy组件化方案

### 整体架构

**casatwy**组件化方案可以处理两种方式的调用，**远程调用**和**本地调用**，对于两个不同的调用方式分别对应两个接口。

- 远程调用通过 `AppDelegate` 代理方法传递到当前应用后，调用远程接口并在内部做一些处理，处理完成后会在远程接口内部调用本地接口，**以实现本地调用为远程调用服务。**
- 本地调用由 `performTarget:action:params:` 方法负责，**但调用方一般不直接调用 `performTarget:` 方法。**`CTMediator` 会对外提供明确参数和方法名的方法，在方法内部调用 `performTarget:` 方法和参数的转换。

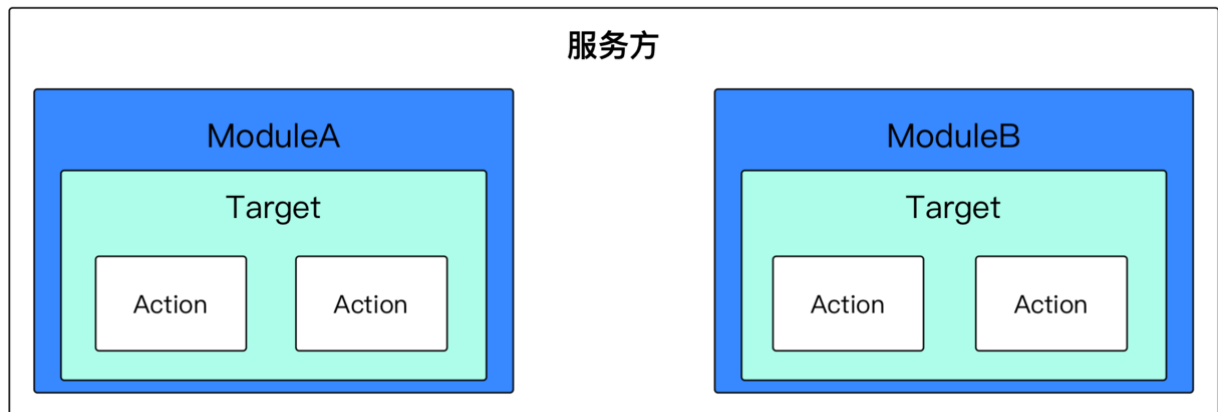
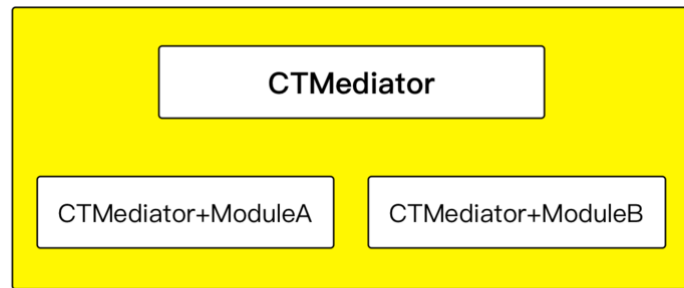


## 架构设计思路

**casatwy**是通过 `CTMediator` 类实现组件化的，在此类中对外提供明确参数类型的接口，接口内部通过 `performTarget` 方法调用服务方组件的 `Target`、`Action`。由于 `CTMediator` 类的调用是通过 `runtime` 主动发现服务的，所以服务方对此类是完全解耦的。

但如果 `CTMediator` 类对外提供的方法都放在此类中，将会对 `CTMediator` 造成极大的负担和代码量。解决方法就是对每个服务方组件创建一个 `CTMediator` 的 `Category`，并将对服务方的 `performTarget` 调用放在对应的 `Category` 中，这些 `Category` 都属于 `CTMediator` 中间件，从而实现了感官上的接口分离。





对于服务方的组件来说，每个组件都提供一个或多个 `Target` 类，在 `Target` 类中声明 `Action` 方法。`Target` 类是当前组件对外提供的一个“服务类”，`Target` 将当前组件中所有的服务都定义在里面，`CTMediator` 通过 `runtime` 主动发现服务。

在 `Target` 中的所有 `Action` 方法，都只有一个字典参数，所以可以传递的参数很灵活，这也是 `casatwy` 提出的去 `Model` 化的概念。在 `Action` 的方法实现中，对传进来的字典参数进行解析，再调用组件内部的类和方法。

## 架构分析

`casatwy` 为我们提供了一个 [Demo](#)，通过这个 `Demo` 可以很好的理解 `casatwy` 的设计思路，下面按照我的理解讲解一下这个 `Demo`。



打开 Demo 后可以看到文件目录非常清楚，在上图中用蓝框框出来的就是中间件部分，红框框出来的就是业务组件部分。我对每个文件夹做了一个简单的注释，包含了其在架构中的职责。

在 CTMediator 中定义远程调用和本地调用的两个方法，其他业务相关的调用由 Category 完成。

```
// 远程App调用入口
- (id)performActionWithUrl:(NSURL *)url completion:(void(^)(NSDictionary
*info))completion;
// 本地组件调用入口
- (id)performTarget:(NSString *)targetName action:(NSString *)actionName params:
(NSDictionary *)params;
```

在 CTMediator 中定义的 ModuleA 的 Category，为其他组件提供了一个获取控制器并跳转的功能，下面是代码实现。由于 casatwy 的方案中使用 performTarget 的方式进行调用，所以涉及到很多硬编码字符串的问题，casatwy 采取定义常量字符串来解决这个问题，这样管理更方便。

```

#import "CTMediator+CTMediatorModuleAActions.h"

NSString * const kCTMediatorTargetA = @"A";
NSString * const kCTMediatorActionNativeFetchDetailViewController =
@"nativeFetchDetailViewController";

@implementation CTMediator (CTMediatorModuleAActions)

- (UIViewController *)CTMediator_viewControllerForDetail {
    UIViewController *viewController = [self performTarget:kCTMediatorTargetA
    action:kCTMediatorActionNativeFetchDetailViewController
    params:@{@"key":@"value"}];

    if ([viewController isKindOfClass:[UIViewController class]]) {
        // view controller 交付出去之后，可以由外界选择是push还是present
        return viewController;
    } else {
        // 这里处理异常场景，具体如何处理取决于产品逻辑
        return [[UIViewController alloc] init];
    }
}
}

```

下面是 `ModuleA` 组件中提供的服务，被定义在 `Target_A` 类中，这些服务可以被 `CTMediator` 通过 `runtime` 的方式调用，这个过程就叫做发现服务。

在 `Target_A` 中对传递的参数做了处理，以及内部的业务逻辑实现。方法是发生在 `ModuleA` 内部的，这样就可以保证组件内部的业务不受外部影响，对内部业务没有侵入性。

```

- (UIViewController *)Action_nativeFetchDetailViewController:(NSDictionary *)params
{
    // 对传过来的字典参数进行解析，并调用ModuleA内部的代码
    DemoModuleADetailViewController *viewController =
[[DemoModuleADetailViewController alloc] init];
    viewController.valueLabel.text = params[@"key"];
    return viewController;
}

```

## 命名规范

在大型项目中代码量比较大，需要避免命名冲突的问题。对于这个问题 **casatwy** 采取的是加前缀的方式，从 **casatwy** 的 `Demo` 中也可以看出，其组件 `ModuleA` 的 `Target` 命名为 `Target_A`，可以区分各个组件的 `Target`。被调用的 `Action` 命名为 `Action_nativeFetchDetailViewController:`，可以区分组件内的方法与对外提供的方法。

**casatwy** 将类和方法的命名，都统一按照其功能做区分当做前缀，这样很好的将组件相关和组件内部代码进行了划分。

## 结果分析

### Protocol

从我调研和使用的结果来说，并不推荐使用 Protocol 方案。首先 Protocol 方案的代码量就比 MGJRouter 方案的多，调用和注册代码量很大，调用起来并不是很方便。

本质上来说 Protocol 方案是通过类对象实例一个变量，并调用变量的方法，并没有真正意义上的改变组件之间的交互方案，但 MGJRouter 的方案却通过 URL Router 的方式改变和统一了组件间调用方式。

并且 Protocol 没有对 Remote Router 的支持，不能直接处理来自 Push 的调用，在灵活性上就不如 MGJRouter 的方案。

### CTMediator

我并不推荐 CTMediator 方案，这套方案实际上是一套很臃肿的方案。虽然为 CTMediator 提供了很多 Category，但实际上组件间的调用逻辑都耦合在了中间件中。同样，和 Protocol 方案存在一个相同的问题，就是调用代码量很大，使用起来并不方便。

在 CTMediator 方案中存在很多硬编码的问题，例如 target、action 以及参数名都是硬编码在中间件中的，这种调用方式并不灵活直接。

但 casatwy 提出了去 Model 化的想法，我觉得这在组件化中传参来说，是非常灵活的，这点我比较认同。相对于 MGJRouter 的话，也采用了去 Model 化的传参方式，而不是直接传递模型对象。组件化传参并不适用传模型对象，但组件内部还是可以使用 Model 的。

### MGJRouter

MGJRouter 方案是一套非常轻量级的方案，其中间件代码总共也就两百行以内，非常简洁。在调用时直接通过 URL 调用，调用起来很简单，我推荐使用这套方案作为组件化架构的中间件。

MGJRouter 最强大的一点在于，统一了远程调用和本地调用。这就使得可以通过 Push 的方式，进行任何允许的组件间调用，对项目运营是有帮助的。

这三套方案都实现了组件间的解耦，MGJRouter 和 Protocol 都是调用方对中间件的耦合，CTMediator 是中间件对组件的耦合，都是单向耦合。

### 接口类

在三套方案中，服务方组件都对外提供一个 PublicHeader 或 Target，在文件中统一定义对外提供的服务，组件间通信的实现代码大多数都在里面。

但三套实现方案实现方式并不同，蘑菇街的两套方案都需要注册操作，无论是 Block 还是 Protocol 都需要注册后才可以提供服务。而 casatwy 的方案则不需要，直接通过 runtime 调用。

## 架构设计

### 组件化架构设计

在上面文章中提到了casatwy方案的CTMediator，蘑菇街方案的MGJRouter和ModuleManager，之后将统称为中间件，下面让我们设计一套组件化架构。

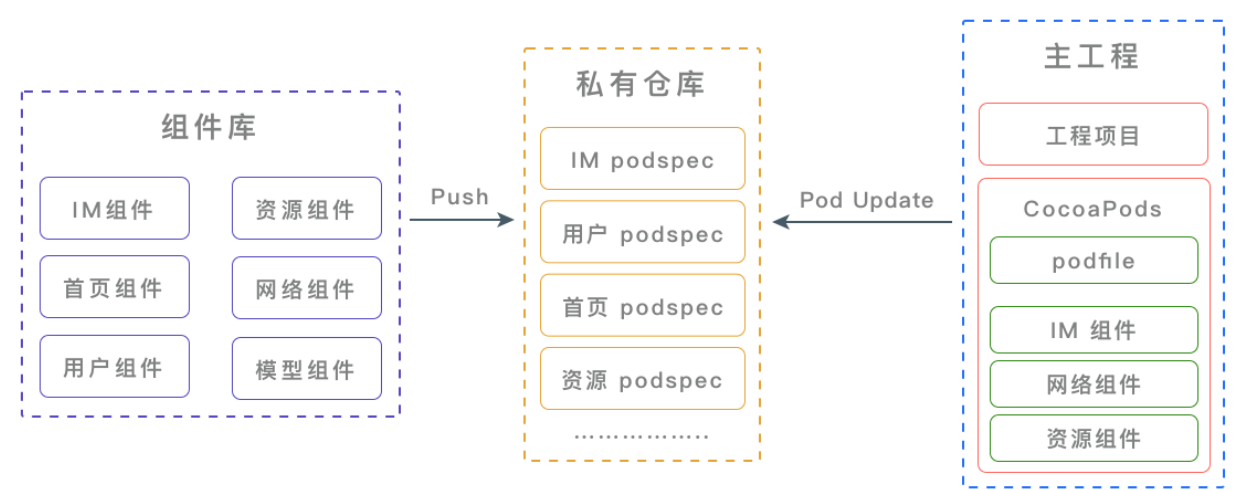
## 整体架构

组件化架构中，需要一个主工程，主工程负责集成所有组件。每个组件都是一个单独的工程，创建不同的git私有仓库来管理，每个组件都有对应的开发人员负责开发。开发人员只需要关注与其相关组件的代码，不用考虑其他组件，这样来新人也好上手。

组件的划分需要注意组件粒度，粒度根据业务可大可小。组件划分可以将每个业务模块都划分为组件，对于网络、数据库等基础模块，也应该划分到组件中。项目中会用到很多资源文件、配置文件等，也应该划分到对应的组件中，避免重复的资源文件。项目实现完全的组件化。

每个组件都需要对外提供调用，在对外公开的类或组件内部，注册对应的URL。组件处理中间件调用的代码应该对其他代码无侵入，只负责对传递过来的数据进行解析和组件内调用的功能。

## 组件集成



每个组件都是一个单独的工程，在组件开发完成后上传到git仓库。主工程通过Cocoapods集成各个组件，集成和更新组件时只需要pod update即可。这样就是把每个组件当做第三方来管理，管理起来非常方便。

Cocoapods可以控制每个组件的版本，例如在主项目中回滚某个组件到特定版本，就可以通过修改podfile文件实现。选择Cocoapods主要因为其本身功能很强大，可以很方便的集成整个项目，也有利于代码的复用。通过这种集成方式，可以很好的避免在传统项目中代码冲突的问题。

## 集成方式

对于组件化架构的集成方式，我在看完bang的博客后专门请教了一下bang。根据在微博上和bang的聊天以及其他博客中的学习，在主项目中集成组件主要分为两种方式——源码和framework，但都是通过CocoaPods来集成。

无论是用CocoaPods管理源码，还是直接管理framework，集成方式都是一样的，都是直接进行pod update等CocoaPods操作。

这两种组件集成方案，实践中也是各有利弊。直接在主工程中集成代码文件，可以看到其内部实现源码，方便在主工程中进行调试。集成 `framework` 的方式，可以加快编译速度，而且对每个组件的代码有很好的保密性。如果公司对代码安全比较看重，可以考虑 `framework` 的形式。

例如手机QQ或者支付宝这样的大型程序，一般都会采取 `framework` 的形式。而且一般这样的大公司，都会有自己的组件库，这个组件库往往可以代表一个大的功能或业务组件，直接添加项目中就可以使用。关于组件化库在后面讲淘宝组件化架构的时候会提到。

## 资源文件

对于项目中图片的集成，可以把图片当做一个单独的组件，组件中只存在图片文件，没有任何代码。图片可以使用 `Bundle` 和 `image assets` 进行管理，如果是 `Bundle` 就针对不同业务模块建立不同的 `Bundle`，如果是 `image assets`，就按照不同的模块分类建立不同的 `assets`，将所有资源放在同一个组件内。

`Bundle` 和 `image assets` 两者相比，我还是更推荐用 `assets` 的方式，因为 `assets` 自身提供很多功能(例如设置图片拉伸范围)，而且在打包之后图片会被打包在 `.cer` 文件中，不会被看到。(现在也可以通过工具对 `.cer` 文件进行解析，获取里面的图片)

使用 `Cocoapods`，所有的资源文件都放置在一个 `podspec` 中，主工程可以直接引用这个 `podspec`，假设此 `podspec` 名为：`Assets`，而这个 `Assets` 的 `podspec` 里面配置信息可以写为：

```
s.resources = "Assets/Assets.xcassets/ ** / *.{png}"
```

主工程则直接在 `podfile` 文件中加入：

```
pod 'Assets', :path => '../MainProject/Assets' (这种写法是访问本地的，可以换成git)
```

这样即可在主工程直接访问到 `Assets` 中的资源文件（不局限图片，`sqlite`、`js`、`html` 亦可，在 `s.resources` 设置好配置信息即可）了。

## 优点

- 组件化开发可以很好的提升代码复用性，组件可以直接拿到其他项目中使用，这个优点在下面淘宝架构中会着重讲一下。
- 对于调试工作，可以放在每个组件中完成。单独的业务组件可以直接提交给测试使用，这样测试起来也比较方便。最后组件开发完成并测试通过后，再将所有组件更新到主项目，提交给测试进行集成测试即可。
- 通过这样的组件划分，组件的开发进度不会受其他业务的影响，可以多个组件并行开发。组件间的通信都交给中间件来进行，需要通信的类只需要接触中间件，而中间件不需要耦合其他组件，这就实现了组件间的解耦。中间件负责处理所有组件之间的调度，在所有组件之间起到控制核心的作用。
- 组件化框架清晰的划分了不同模块，从整体架构上来约束开发人员进行组件化开发，实现了组件间的物理隔离。组件化架构在各个模块之间天然形成了一道屏障，避免某个开发人员偷懒直接引用头文件，产生组件间的耦合，破坏整体架构。
- 使用组件化架构进行开发时，因为每个人都负责自己的组件，代码提交也只提交自己负责模块的仓库，所以代码冲突的问题会变得很少。
- 假设以后某个业务发生大的改变，需要对相关代码进行重构，可以在单个组件内进行重构。组件

化架构降低了重构的风险，保证了代码的健壮性。

## 架构分析

在 `MGJRouter` 方案中，是通过调用 `OpenURL:` 方法并传入 `URL` 来发起调用的。鉴于 `URL` 协议名等固定格式，可以通过判断协议名的方式，使用配置表控制 `H5` 和 `native` 的切换，配置表可以从后台更新，只需要将协议名更改一下即可。

```
mgj://detail?id=123456
http://www.mogujie.com/detail?id=123456
```

假设现在线上的 `native` 组件出现严重 `bug`，在后台将配置文件中原有的本地 `URL` 换成 `H5` 的 `URL`，并更新客户端配置文件。

在调用 `MGJRouter` 时传入这个 `H5` 的 `URL` 即可完成切换，`MGJRouter` 判断如果传进来的是一个 `H5` 的 `URL` 就直接跳转 `webView`。而且 `URL` 可以传递参数给 `MGJRouter`，只需要 `MGJRouter` 内部做参数截取即可。

使用组件化架构开发，组件间的通信都是有成本的。所以尽量将业务封装在组件内部，对外只提供简单的接口。即“高内聚、低耦合”原则。

把握好组件划分粒度的细化程度，太细则项目过于分散，太大则项目组件臃肿。但是项目都是从小到大的一个发展过程，所以不断进行重构是掌握这个组件的细化程度最好的方式。

## 注意点

如果通过 `framework` 等二进制形式，将组件集成到主项目中，需要注意预编译指令的使用。因为预编译指令在打包 `framework` 的时候，就已经在组件二进制代码中打包好，到主项目中的时候预编译指令其实已经不再起作用了，而是已经在打包时按照预编译指令编码为固定二进制。

## 我公司架构

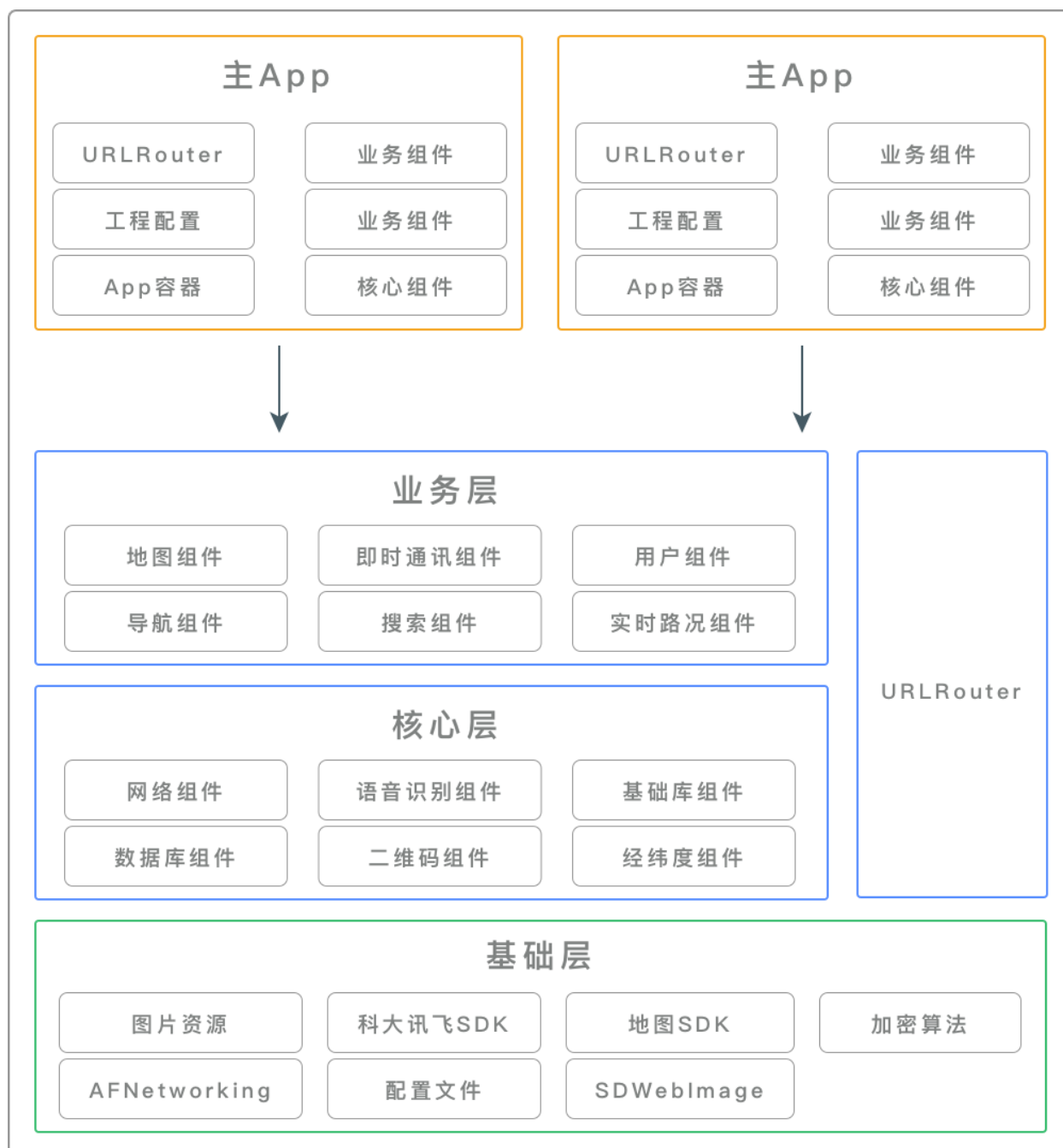
对于项目架构来说，一定要建立于业务之上来设计架构。不同的项目业务不同，组件化方案的设计也会不同，应该设计最适合公司业务架构。

## 架构设计

我公司项目是一个地图导航应用，业务层之下的核心模块和基础模块占比较大，涉及到地图SDK、算路、语音等模块。且基础模块相对比较独立，对外提供了很多调用接口。由此可以看出，公司项目是一个重逻辑的项目，不像电商等 `App` 偏展示。

项目整体的架构设计是：层级架构+组件化架构，对于具体的实现细节会在下面详细讲解。采取这种结构混合的方式进行整体架构，对于组件的管理和层级划分比较有利，符合公司业务需求。





在设计架构时，我们将整个项目都拆分为组件，组件化程度相当高。用到哪个组件就在工程中通过 `Podfile` 进行集成，并通过 `URLRouter` 统一所有组件间的通信。

组件化架构是项目的整体框架，而对于框架中每个业务模块的实现，可以是任意方式的架构，`MVVM`、`MVC`、`MVCS` 等都是可以的，只要通过 `MGJRouter` 将组件间的通信方式统一即可。

## 分层架构

组件化架构在物理结构上来说是不分层次的，只有组件与组件之间的划分关系。但是在组件化架构的基础上，应该根据项目和业务设计自己的层次架构，这套层次架构可以用来区分组件所处的层次及职责，所以我们设计了层级架构+组件化架构的整体架构。

我公司项目最开始设计的是三层架构：业务层 -> 核心层 (`high` + `low`) -> 基础层，其中核心层又分为 `high` 和 `low` 两部分。但是这种架构会造成核心层过重，基础层过轻的问题，这种并不适合组件化架构。

在三层架构中会发现，low 层并没有耦合业务逻辑，在同层级中是比较独立的，职责较为单一和基础。我们对 low 层下沉到基础层中，并和基础层进行合并。所以架构被重新分为三层架构：业务层 -> 核心层 -> 基础层。之前基础层大多是资源文件和配置文件，在项目中的存在感并不高。

在分层架构中，需要注意只能上层对下层依赖，下层对上层不能有依赖，下层中不要包含上层业务逻辑。对于项目中存在的公共资源和代码，应该将其下沉到下层中。

## 职责划分

在三层架构中，业务层负责处理上层业务，将不同业务划分到相应组件中，例如 IM 组件、导航组件、用户组件等。业务层的组件间关系比较复杂，会涉及到组件间业务的通信，以及业务层组件对下层组件的引用。

核心层位于业务层下方，为业务层提供业务支持，如网络、语音识别等组件应该划分到核心层。核心层应该尽量减少组件间的依赖，将依赖降到最小。核心层有时相互之间也需要支持，例如经纬度组件需要网络组件提供网络请求的支持，这种是不可避免的。

其他比较基础的模块，都放在基础层当做基础组件。例如 AFN、地图 SDK、加密算法等，这些组件都比较独立且不掺杂任何业务逻辑，职责更加单一，相对于核心层更底层。可以包含第三方库、资源文件、配置文件、基础库等几大类，基础层组件相互之间不应该产生任何依赖。

在设计各个组件时，应该遵循“高内聚，低耦合”的设计规范，组件的调用应该简单且直接，减少调用方的其他处理。对于核心层和基础层的划分，可以以是否涉及业务、是否涉及同级组件间通信、是否经常改动为参照点。如果符合这几点则放在核心层，不符合则放在基础层。

## 集成方式

新建一个项目后，首先将配置文件、URLRouter、App 容器等集成到主工程中，做一些基础的项目配置，随后集成需要的组件即可。项目被整体拆分为组件化架构后，应用对所有组件的集成方式都是一样的，通过 Podfile 将需要的组件集成到项目中。通过组件化的方式，使得开发新项目速度变得非常快。

在集成业务层和核心层组件后，组件间的通信都是由 URLRouter 进行通信，项目中不允许直接依赖组件源码。而基础层组件则在集成后直接依赖，例如资源文件和配置文件，这些都是直接在主工程或组件中使用的。第三方库则是通过核心层的业务封装，封装后由 URLRouter 进行通信，但核心层也是直接依赖第三方库源码的。

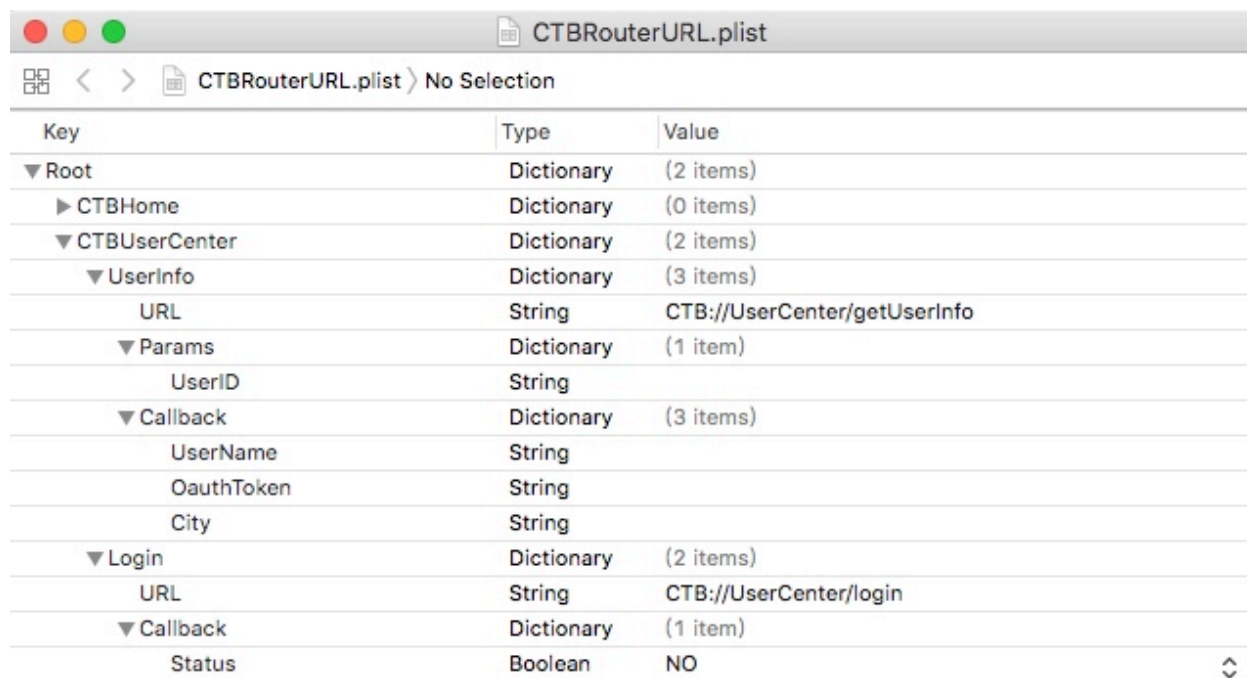
组件的集成方式有两种，源码和 framework 的形式，我们使用 framework 的方式集成。因为一般都是项目比较大才用组件化的，但大型项目都会存在编译时间的问题，如果通过 framework 则会大大减少编译时间，可以节省开发人员的时间。

## 组件间通信

对于组件间通信，我们采用的 MGJRouter 方案。因为 MGJRouter 现在已经很稳定了，而且可以满足蘑菇街这样量级的 App 需求，证明是很好的，没必要自己写一套再慢慢踩坑。

MGJRouter 的好处在于，其调用方式很灵活，通过 MGJRouter 注册并在 block 中处理回调，通过 URL 直接调用或者 URL+Params 字典的方式进行调用。由于通过 URL 拼接参数或 Params 字典传值，所以其参数类型没有数量限定，传递比较灵活。在通过 openURL: 调用后，可以在 completionBlock 中处理完成逻辑。

MGJRouter 有个问题在于，在编写组件间通信的代码时，会涉及到大量的 Hardcode。对于 Hardcode 的问题，蘑菇街开发了一套后台系统，将所有的 Router 需要的 URL 和参数名，都定义到这套系统中。我们维护了一个 Plist 表，内部按不同组件进行划分，包含 URL 和传参名以及回调参数。



Key	Type	Value
▼ Root	Dictionary	(2 items)
▶ CTBHome	Dictionary	(0 items)
▼ CTBUserCenter	Dictionary	(2 items)
▼ UserInfo	Dictionary	(3 items)
URL	String	CTB://UserCenter/getUserInfo
▼ Params	Dictionary	(1 item)
UserID	String	
▼ Callback	Dictionary	(3 items)
UserName	String	
OauthToken	String	
City	String	
▼ Login	Dictionary	(2 items)
URL	String	CTB://UserCenter/login
▼ Callback	Dictionary	(1 item)
Status	Boolean	NO

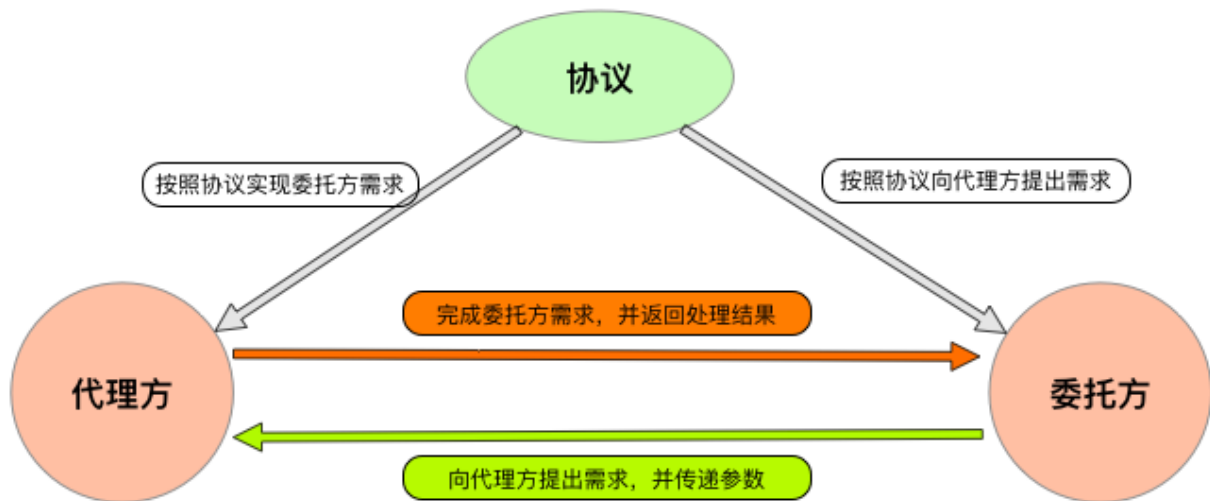
## 路由层安全

组件化架构需要注意路由层的安全问题。MGJRouter 方案可以处理本地及远程的 OpenURL 调用，如果是程序内组件间的 OpenURL 调用，则不需要进行校验。而跨应用的 OpenURL 调用，则需要进行合法性检查。这是为了防止第三方伪造进行 OpenURL 调用，所以对应用外调起的 OpenURL 进行的合法性检查，例如其他应用调起、服务器 Remote Push 等。

在合法性检查的设计上，每个从应用外调起的合法 URL 都会带有一个 token，在本地会对 token 进行校验。这种方式的优势在于，没有网络请求的限制和延时。

## 代理方法

在项目中经常会用到代理模式传值，代理模式在 iOS 中主要分为三部分，协议、代理方、委托方三部分。



但如果使用组件化架构的话，会涉及到组件与组件间的代理传值，代理方需要设置为委托方的 `delegate`，但组件间是不可以直接产生耦合的。对于这种跨组件的代理情况，我们直接将代理方的对象通过 `MGJRouter` 以参数的形式传给另一个组件，在另一个组件中进行代理设置。

```
HomeViewController *homeVC = [[HomeViewController alloc] init];
NSDictionary *params = @{@"CTBUserCenterLoginDelegateKey" : homeVC};
[MGJRouter openURL:@"CTB://UserCenter/UserLogin" withUserInfo:params
completion:nil];

[MGJRouter registerURLPattern:@"CTB://UserCenter/UserLogin"
toHandler:^(NSDictionary *routerParameters) {
    UIViewController *homeVC = routerParameters[CTBUserCenterLoginDelegateKey];
    LoginViewController *loginVC = [[LoginViewController alloc] init];
    loginVC.delegate = homeVC;
}];
```

协议的定义放在委托方组件的 `PublicHeader.h` 中，代理方组件只引用这个 `PublicHeader.h` 文件，不耦合委托方内部代码。为了避免定义的代理方法中出现耦合的情况，方法中不能出现和组件内部业务有关的对象，只能传递系统的类。如果涉及到交互的情况，则通过协议方法的返回值进行。

### 组件传参

`MGJRouter` 可以在 `openURL:` 时传入一个 `NSDictionary` 参数，在接触 `RAC` 之后，我在想是不是可以把 `NSDictionary` 参数变为 `RACSignal` 参数，直接传一个信号过去。

注册 `MGJRouter`：

```

RACSignal *signal = [RACSignal createSignal:^(RACDisposable *(id<RACSubscriber>
subscriber) {
    [subscriber sendNext:@"刘小壮"];
    return [RACDisposable disposableWithBlock:^(
        NSLog(@"disposable");
    )];
}]];

[MGJRouter registerURLPattern:@"CTB://UserCenter/getUserInfo" withSignal:signal];

```

调用 `MGJRouter` :

```

RACSignal *signal = [MGJRouter openURL:@"CTB://UserCenter/getUserInfo"];
[signal subscribeNext:^(NSString *userName) {
    NSLog(@"userName %@", userName);
}]];

```

这种方式是可行的。使用 `RACSignal` 方式优点在于，相对于直接传字典过去更加灵活，并且具备 `RAC` 的诸多特性。但缺点也不少，信号控制不好乱用的话也很容易挖坑，是否使用还是看团队情况了。

## 常量定义

在项目中经常会定义一些常量，例如通知名、常量字符串等，这些常量一般都和所属组件有很强的关系，不好单独拆出来放到其他组件。但是这些变量数量并不是很多，而且不是每个组件中都有。

所以，我们将这些变量都声明在 `PublicHeader.h` 文件中，其他组件只能引用 `PublicHeader.h` 文件，不能引用组件内部业务代码，这样就规避掉了组件间耦合的问题。

## H5和Native通信

在项目中经常会用到 `H5` 页面，如果能通过点击 `H5` 页面调起原生页面，这样的话 `Native` 和 `H5` 的融合会更好。所以我们设计了一套 `H5` 和 `Native` 交互的方案，这套方案可以使用 `URLRouter` 的方式调起原生页面，实现方式也很简单，并且这套方案和 `H5` 原本的跳转逻辑并不冲突。

通过 `iOS` 自带 `UIWebView` 创建一个 `H5` 页面后，`H5` 可以通过调用下面的 `JS` 函数和 `Native` 通信。调用时可以传入新的 `URL`，这个 `URL` 可以设置为 `URLRouter` 的 `URL`。

```

window.location.href = 'CTB://UserCenter/UserLogin?
userName=lxz&WeChatID=1z2046703959';

```

通过 `JS` 刷新 `H5` 页面时，会调用下面的代理方法。如果方法返回 `YES`，则会根据 `URL` 协议进行跳转。

```

- (BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest
*)request navigationType:(UIWebViewNavigationType)navigationType;

```

跳转时系统会判断通信协议，如果是 HTTP 等标准协议，则会在当前页面进行刷新。如果跳转协议在 URL Scheme 中注册，则会通过系统 openURL: 的方式调用到 AppDelegate 的系统代理方法中，在代理方法中调用 URLRouter，则可以通过 H5 页面唤起原生页面。

## AppService

在应用启动过程中，通常会做一些初始化操作。有些初始化操作是运行程序所需要的，例如崩溃统计、建立服务器的长连接等。或有的组件会对初始化操作有依赖关系，例如网络组件依赖 requestToken 等。

对于应用启动时的初始化操作，应该创建一个 AppService 来统一管理启动操作，将初始化操作都放在里面，包含创建根控制器等。其中有的初始化操作需要尽快执行，有的并不需要立即执行，可以根据不同操作设定优先级，来管理所有初始化操作。

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

typedef NS_ENUM(NSUInteger, CTBAppServicePriority) {
    CTBAppServicePriorityLow,
    CTBAppServicePriorityDefault,
    CTBAppServicePriorityHigh,
};

@interface CTBAppService : NSObject
+ (instancetype)appService;
- (void)registerService:(dispatch_block_t)serviceBlock
    priority:(CTBAppServicePriority)priority;
@end
```

## Model层设计

项目中存在很多的模型定义，那组件化后这些模型应该定义在哪呢？

casatwy对模型类的观点是去 Model 化，简单来说就是用字典代替 Model 存储数据。这对于组件化架构来说，是解决组件之间数据传递的一个很好的方法。但是去 Model 的方式，会存在大量的字段读取代码，使用起来远没有模型类方便。

因为模型类是关乎业务的，理论上必须放在业务层也就是业务组件这一层。但是要把模型对象从一个组件中当做参数传递到另一个组件中，模型类放在调用方和被调方的哪个组件都不太合适，而且有可能不只两个组件使用到这个模型对象。这样的话在其他组件使用模型对象，必然会造成引用和耦合。

如果在用到这个模型对象的所有组件中，都分别维护一份相同的模型类，或者各自维护不同结构的模型类，这样之后业务发生改变模型类就会很麻烦，这是不可取的。

### 设计方案

如果将所有模型类单独拉出来，定义一个模型组件呢？

这个看起来比较可行，将这个定义模型的组件下沉到基础层，模型组件不包含业务，只声明模型对象的类。如果将原来各个组件的模型类定义都拉出来，单独放在一个组件中，可以将原有各组件的 Model 层变得很轻量，这样对整个项目架构来说也是有好处的。



在通过 Router 进行组件间调用时，通过字典进行传值，这种方式比较灵活。在组件内部使用 Model 层时，还是用模型组件中定义的 Model 类。Model 层建议还是用 Model 对象的形式比较方便，不建议整体使用去 Model 化的设计。在接收到其他组件传递过来的字典参数时，可以通过 Model 类提供的初始化方法，或其他转 Model 框架将字典转为 Model 对象。

```
@interface CTBStoreWelfareListModel : NSObject
/**
 * 自定义初始化方法
 */
- (instancetype)initWithDict:(NSDictionary *)dict;
@end
```

我公司持久化方案用的是 CoreData，所有模型的定义都在 CoreData 组件中，则不需要再单独创建一个模型组件。

## 动态化构想

我公司项目是一个常规的地图类项目，首页和百度、高德等主流地图导航 App 一样，有很多添加在地图上的控件。有的版本会添加控件上去，而有的版本会删除控件，与之对应的功能也会被隐藏。

所以，有次和组里小伙伴们开会的时候就在考虑，能不能在服务器下发代码对首页进行布局！这样就可以对首页进行动态布局，例如有活动的时候在指定时间显示某个控件，这样可以避免 App Store 审核慢的问题。又或者线上某个模块出现问题，可以紧急下架出问题的模块。

对于这个问题，我们设计了一套动态配置方案，这套方案可以对整个 App 进行配置。

### 配置表设计

对于动态配置的问题，我们简单设计了一个配置表，初期打算在首页上先进行试水，以后可能会布置到更多的页面上。这样应用程序各模块的入口，都可以通过配置表来控制，并且通过 Router 控制页面间跳转，灵活性非常大。

在第一次安装程序时使用内置的配置表，之后每次都用服务器来替换本地的配置表，这样就可以实现动态配置应用。下面是一个简单设计的配置数据，JSON 中配置的是首页的配置信息，用来模拟服务器下发的数据，真正服务器下发的字段会比这个多很多。



```

{
  "status": 200,
  "viewList": [
    {
      "className": "UIButton",
      "frame": {
        "originX": 10,
        "originY": 10,
        "sizeWidth": 50,
        "sizeHeight": 30
      },
      "normalImageURL": "http://image/normal.com",
      "highlightedImageURL": "http://image/highlighted.com",
      "normalText": "text",
      "textColor": "#FFFFFF",
      "routerURL": "CTB://search/**"
    }
  ]
}

```

对于服务器返回的数据，我们会创建一套解析器，这个解析器用来将 JSON 解析并“转换”为标准的 UIKit 控件。点击后的事件都通过 Router 进行跳转，所以首页的灵活性和 Router 的使用程度成正比。

这套方案类似于 React Native 的方案，从服务器下发页面展示效果，但没有 React Native 功能那么全。相对而言是一个轻量级的配置方案，主要用于页面配置。

### 资源动态配置

除了页面的配置之外，我们发现地图类 App 一般都存在 ipa 过大的问题，这样在下载时很消耗流量以及时间。所以我们就在想能不能把资源也做到动态配置，在用户运行程序的时候再加载资源文件包。

我们想通过配置表的方式，将图片资源文件都放到服务器上，图片的 URL 也随配置表一起从服务器获取。在使用时请求图片并缓存到本地，成为真正的网络 APP。在此基础上设计缓存机制，定期清理本地的图片缓存，减少用户磁盘占用。

## 滴滴组件化架构

之前看过滴滴 iOS 负责人李贤辉的[技术分享](#)，分享的是滴滴 iOS 客户端的架构发展历程，下面简单总结一下。

### 发展历程

滴滴在最开始的时候架构较混乱。然后在 2.0 时期重构为 MVC 架构，使项目划分更加清晰。在 3.0 时期上线了新的业务线，这时开始采用游戏开发中的状态机机制，暂时可以满足现有业务。

然而在后期不断上线顺风车、代驾、巴士等多条业务线的情况下，现有架构变得非常臃肿，代码耦合严重。从而在 2015 年开始了代号为 “The One” 的方案，这套方案就是滴滴的组件化方案。

### 架构设计

滴滴的组件化方案，和蘑菇街方案类似，将项目拆分为各个组件，通过 `CocoaPods` 来集成和管理各个组件。项目被拆分为业务部分和技术部分，业务部分包括专车、拼车、巴士等组件，使用一个 `pod` 管理。技术部分则分为登录分享、网络、缓存这样的一些基础组件，分别使用不同的 `pod` 管理。

组件间通信通过 `ONERouter` 中间件进行通信，`ONERouter` 类似于 `MGJRouter`，担负起协调和调用各个组件的作用。组件间通信通过 `OpenURL` 方法，来进行对应的调用。`ONERouter` 内部保存一份 `Class-URL` 的映射表，通过 `URL` 找到 `Class` 并发起调用，`Class` 的注册放在 `+load` 方法中进行。

滴滴在业务组件内部使用 `MVVM+MVCS` 混合的架构，两种架构都是 `MVC` 的衍生版本。其中 `MVCS` 中的 `Store` 负责数据相关逻辑，例如订单状态、地址管理等数据处理。通过 `MVVM` 中的 `VM` 给控制器瘦身，最后 `Controller` 的代码量就很少了。

## 滴滴首页分析

滴滴文章中说道首页只能有一个地图实例，这在很多地图导航相关应用中都是这样做的。滴滴首页主控制器持有导航栏和地图，每个业务线首页控制器都添加在主控制器上，并且业务线控制器背景都设置为透明，将透明部分响应事件传递到下面的地图中，只响应属于自己的响应事件。

由主控制器来切换各个业务线首页，切换页面后根据不同的业务线来更新地图数据。

## 淘宝组件化架构

本章节源自于宗心在阿里技术沙龙上的一次[技术分享](#)

### 架构发展

淘宝 `iOS` 客户端初期是单工程的普通项目，但随着业务的飞速发展，现有架构并不能承载越来越多的业务需求，导致代码间耦合很严重。后期开发团队对其不断进行重构，将项目重构为组件化架构，淘宝 `iOS` 和 `Android` 两个平台，除了某个平台特有的一些特性或某些方案不便实施之外，大体架构都是差不多的。

#### 发展历程

1. 刚开始是普通的单工程项目，以传统的 `MVC` 架构进行开发。随着业务不断的增加，导致项目非常臃肿、耦合严重。
2. **2013年**淘宝开启**"all in 无线"**计划，计划将淘宝变为一个大的平台，将阿里系大多数业务都集成到这个平台上，造成了业务的大爆发。  
淘宝开始实行插件化架构，将每个业务模块划分为一个子工程，将组件以 `framework` 二方库的形式集成到主工程。但这种方式并没有做到真正的拆分，还是在一个工程中使用 `git` 进行 `merge`，这样还会造成合并冲突、不好回退等问题。
3. 迎来淘宝移动端有史以来最大的重构，将其重构为组件化架构。将每个模块当做一个组件，每个组件都是一个单独的项目，并且将组件打包成 `framework`。主工程通过 `podfile` 集成所有组件的 `framework`，实现业务之间真正的隔离，通过 `CocoaPods` 实现组件化架构。

### 架构优势

淘宝是使用 `git` 来做源码管理的，在插件化架构时需要尽可能避免 `merge` 操作，否则在大团队中协作成本是很大的。而使用 `CocoaPods` 进行组件化开发，则避免了这个问题。

在 CocoaPods 中可以通过 podfile 很好的配置各个组件，包括组件的增加和删除，以及控制某个组件的版本。使用 CocoaPods 的原因，很大程度是为了解决大型项目中，代码管理工具 merge 代码导致的冲突。并且可以通过配置 podfile 文件，轻松配置项目。

每个组件工程有两个 target，一个负责编译当前组件和运行调试，另一个负责打包 framework。先在组件工程做测试，测试完成后再集成到主工程中集成测试。

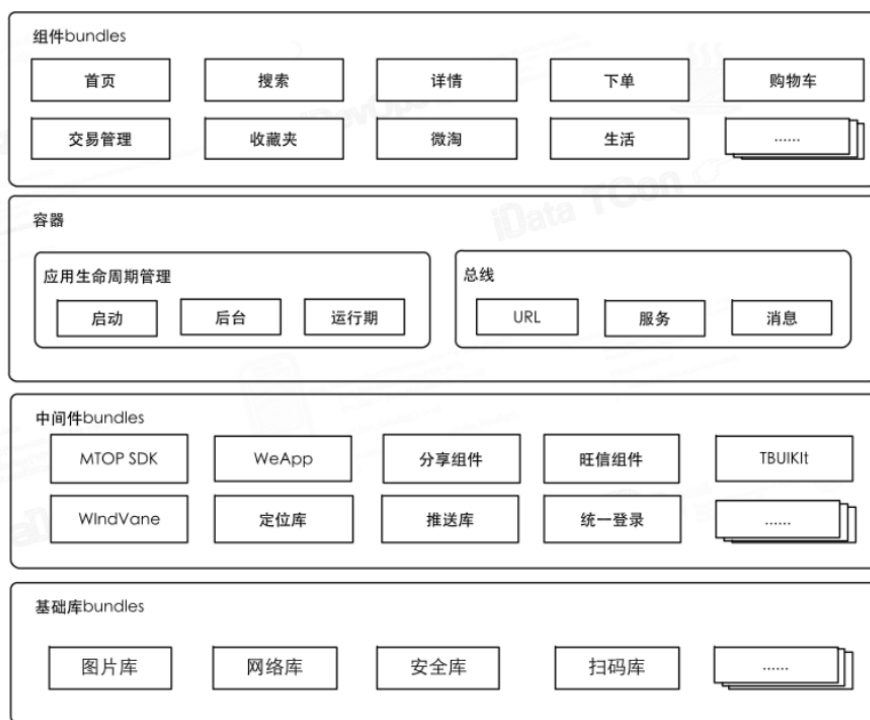
每个组件都是一个独立 app，可以独立开发、测试，使得业务组件更加独立，所有组件可以并行开发。下层为上层提供能满足需求的底层库，保证上层业务层可以正常开发，并将底层库封装成 framework 集成到主工程中。

使用 CocoaPods 进行组件集成的好处在于，在集成测试自己组件时，可以直接在本地主工程中，通过 podfile 使用当前组件源码，可以直接进行集成测试，不需要提交到服务器仓库。

## 淘宝四层架构

# 指导思想

- 分而治之
  - 并行开发
- 一切皆组件
  - BundleApp



淘宝架构的核心思想是一切皆组件，将工程中所有代码都抽象为组件。

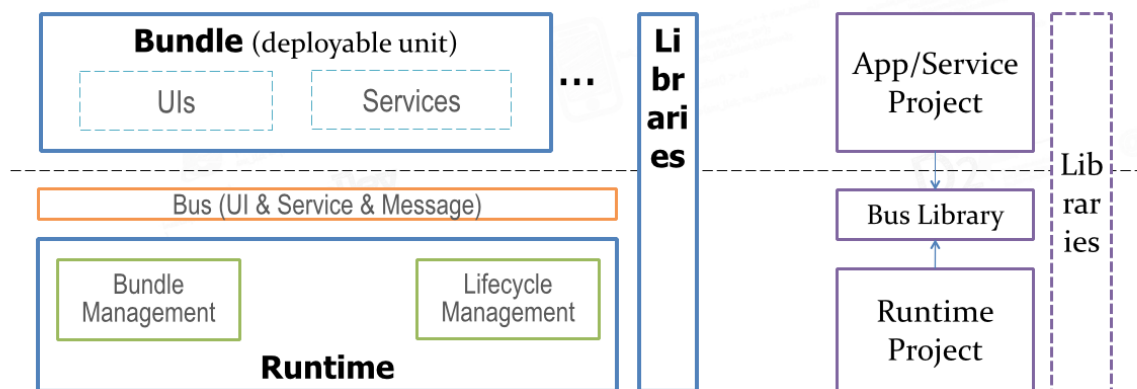
淘宝架构主要分为四层，最上层是组件 Bundle (业务组件)，依次往下是容器(核心层)，中间件 Bundle (功能封装)，基础库 Bundle (底层库)。容器层为整个架构的核心，负责组件间的调度和消息派发。

## 总线设计

总线设计：URL 路由+服务+消息。统一所有组件的通信标准，各个业务间通过总线进行通信。

# 解除耦合，制定标准

- 总线
  - URL总线（跨平台统一URL寻址方式）：三平台统一URL，自动降级，中心分发（支持hook）
  - 服务总线：根据服务接口提供稳定服务
  - 消息总线：中心分发，按需加载
- 开发透明
  - 只需要遵守规则，不关心底层/其他业务实现



## URL总线

通过 URL 总线对三端进行了统一，一个 URL 可以调起 iOS、Android、前端三个平台，产品运营和服务器只需要下发一套 URL 即可调用对应的组件。

URL 路由可以发起请求也可以接受返回值，和 MGJRouter 差不多。URL 路由请求可以被解析就直接拿来使用，如果不能被解析就跳转 H5 页面。这样就完成了一个对不存在组件调用的兼容，使用户手中比较老的版本依然可以显示新的组件。

服务提供一些公共服务，由服务方组件负责实现，通过 Protocol 进行调用。

## 消息总线

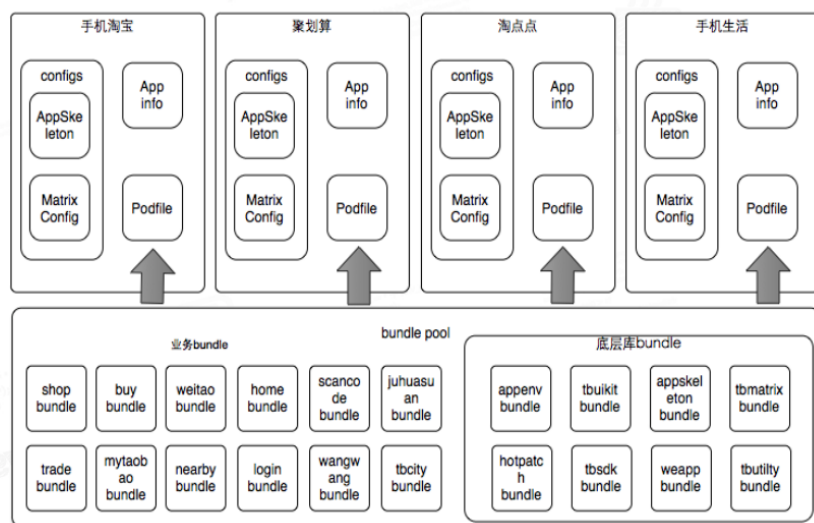
应用通过消息总线进行事件的中心分发，类似于 iOS 的通知机制。例如客户端前后台切换，则可以通过消息总线分发到接收消息的组件。因为通过 URLRouter 只是一对一的进行消息派发和调度，如果多次注册同一个 URL，则会被覆盖掉。

## Bundle App

# 减少新业务接入/移除成本

bundleApp

- 标准化
  - 统一的通信调用标准，bundle间互通的基础
  - 无法回避的瘦身问题
- 灵活性
  - Bundle自由组装（淘宝生活，码上淘）
  - 中间件基础库自由引入



在组件化架构的基础上，淘宝提出 **Bundle App** 的概念，可以通过已有组件，进行简单配置后就可以组成一个新的 **app** 出来。解决了多个应用业务复用的问题，防止重复开发同一业务或功能。

**Bundle** 即 **App**，容器即 **OS**，所有 **Bundle App** 被集成到 **OS** 上，使每个组件的开发就像 **app** 开发一样简单。这样就做到了从巨型 **app** 回归普通 **app** 的轻盈，使大型项目的开发问题彻底得到了解决。

## 总结

### 留个小思考

到目前为止组件化架构文章就写完了，文章确实挺长的，看到这里真是辛苦你了 😊。下面留个小思考，把下面字符串复制到微信输入框随便发给一个好友，然后点击下面链接大概也能猜到微信的组件化方案。

weixin://dl/profile

## 总结

各位可以来我博客评论区讨论，可以讨论文中提到的技术细节，也可以讨论自己公司架构所遇到的问题，或自己独到的见解等等。无论是不是架构师或新入行的 **iOS** 开发，欢迎各位以一个讨论技术的心态来讨论。在评论区你的问题可以被其他人看到，这样可能会给其他人带来一些启发。

[我的博客地址](#)

**Demo** 地址：蘑菇街和 **casatwy** 组件化方案，其 **Github** 上都给出了 **Demo**，这里就贴出其 **Github** 地址了。

[蘑菇街-MGJRouter](#)

[casatwy-CTMediator](#)



好多朋友在看完这篇文章后，都问有没有 Demo。其实架构是思想上的东西，重点还是理解架构思想。文章中对思想的概述已经很全面了，用多个项目的例子来描述组件化架构。就算提供了 Demo，也没法把 Demo 套在其他工程上用，因为并不一定适合所在的工程。

后来想了一下，我把组件化架构的集成方式，简单写了个 Demo，这样可以解决很多人在架构集成上的问题。我把 Demo 放在我 Github 上了，用 Coding 的服务器来模拟我公司私有服务器，直接拿 MGJRouter 来当 Demo 工程中的 Router。下面是 Demo 地址，麻烦各位记得点个 star 😊。

## 组件化架构集成Demo

如果你觉得不错，请把PDF帮忙转到其他群里，或者你的朋友，让更多的人了解组件化架构，衷心感谢！😊



Github地址：<https://github.com/DeveloperErenLiu/ComponentArchitectureBook>