

iOS 组件化 —— 路由设计思路分析

作者：@一缕殇流化隐半边冰霜

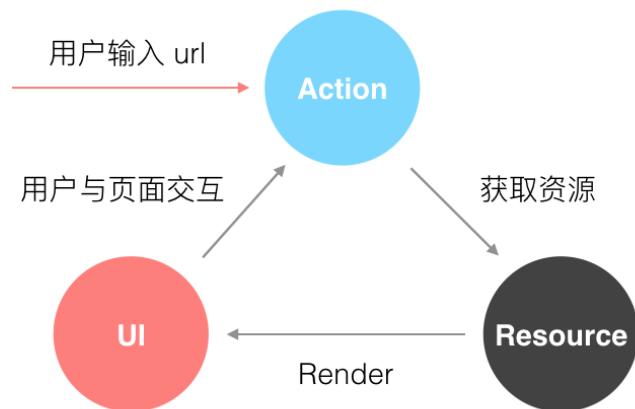
随着用户的需求越来越多，对 App 的用户体验也变的要求越来越高。为了更好的应对各种需求，开发人员从软件工程的角度，将 App 架构由原来简单的 MVC 变成 MVVM，VIPER 等复杂架构。更换适合业务的架构，是为了后期能更好的维护项目。

但是用户依旧不满意，继续对开发人员提出了更多更高的要求，不仅需要高质量的用户体验，还要求快速迭代，最好一天出一个新功能，而且用户还要求不更新就能体验到新功能。为了满足用户需求，于是开发人员就用 H5，ReactNative，Weex 等技术对已有的项目进行改造。项目架构也变得更加的复杂，纵向的会进行分层，网络层，UI 层，数据持久层。每一层横向的也会根据业务进行组件化。尽管这样做了以后会让开发更加有效率，更加好维护，但是如何解耦各层，解耦各个界面和各个组件，降低各个组件之间的耦合度，如何能让整个系统不管多么复杂的情况下都能保持“高内聚，低耦合”的特点？这一系列的问题都摆在开发人员面前，亟待解决。今天就来谈谈解决这个问题的一些思路。

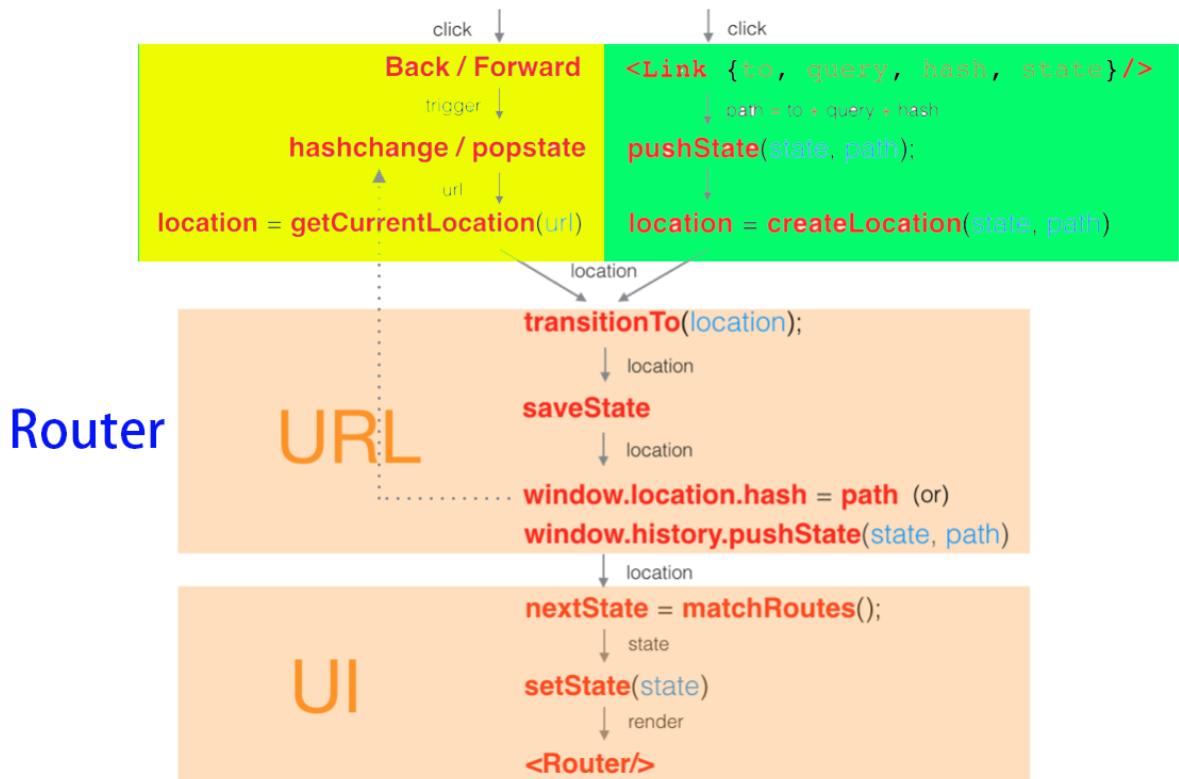
一. 引子

大前端发展这么多年了，相信也一定会遇到相似的问题。近两年 SPA 发展极其迅猛，React 和 Vue 一直处于风口浪尖，那我们就看看他们是如何处理好这一问题的。

在 SPA 单页面应用，路由起到了很关键的作用。路由的作用主要是保证视图和 URL 的同步。在前端的眼里看来，视图是被看成是资源的一种表现。当用户在页面中进行操作时，应用会在若干个交互状态中切换，路由则可以记录下某些重要的状态，比如用户查看一个网站，用户是否登录、在访问网站的哪一个页面。而这些变化同样会被记录在浏览器的历史中，用户可以通过浏览器的前进、后退按钮切换状态。总的来说，用户可以通过手动输入或者与页面进行交互来改变 URL，然后通过同步或者异步的方式向服务端发送请求获取资源，成功后重新绘制 UI，原理如下图所示：



react-router 通过传入的 location 到最终渲染新的 UI，流程如下：



location 的来源有2种，一种是浏览器的回退和前进，另外一种是直接点了一个链接。新的 location 对象后，路由内部的 matchRoutes 方法会匹配出 Route 组件树中与当前 location 对象匹配的一个子集，并且得到了 nextState，在 this.setState(nextState) 时就可以实现重新渲染 Router 组件。

大前端的做法大概是这样的，我们可以把这些思想借鉴到 iOS 这边来。上图中的 Back / Forward 在 iOS 这边很多情况下都可以被 UINavigation 所管理。所以 iOS 的 Router 主要处理绿色的那一块。

二. App 路由能解决哪些问题

既然前端能在 SPA 上解决 URL 和 UI 的同步问题，那这种思想可以在 App 上解决哪些问题呢？

思考如下的问题，平时我们开发中是如何优雅的解决的：

1.3D-Touch 功能或者点击推送消息，要求外部跳转到 App 内部一个很深层次的一个界面。

比如微信的 3D-Touch 可以直接跳转到“我的二维码”。“我的二维码”界面在我的里面的第三级界面。或者再极端一点，产品需求给了更加变态的需求，要求跳转到 App 内部第十层的界面，怎么处理？

2.自家的一系列 App 之间如何相互跳转？

如果自己 App 有几个，相互之间还想相互跳转，怎么处理？

3.如何解除 App 组件之间和 App 页面之间的耦合性？

随着项目越来越复杂，各个组件，各个页面之间的跳转逻辑关联性越来越多，如何能优雅的解除各个组件和页面之间的耦合性？

4.如何能统一 iOS 和 Android 两端的页面跳转逻辑？甚至如何能统一三端的请求资源的方式？

项目里面某些模块会混合 ReactNative，Weex，H5 界面，这些界面还会调用 Native 的界面，以及 Native 的组件。那么，如何能统一 Web 端和 Native 端请求资源的方式？

5.如果使用了动态下发配置文件来配置 App 的跳转逻辑，那么如果做到 iOS 和 Android 两边只要共用一套配置文件？

6.如果 App 出现 bug 了，如何不用 JSPatch，就能做到简单的热修复功能？

比如 App 上线突然遇到了紧急 bug，能否把页面动态降级成 H5，ReactNative，Weex？或者是直接换成一个本地的错误界面？

7.如何在每个组件间调用和页面跳转时都进行埋点统计？每个跳转的地方都手写代码埋点？利用 Runtime AOP？

8.如何在每个组件间调用的过程中，加入调用的逻辑检查，令牌机制，配合灰度进行风控逻辑？

9.如何在 App 任何界面都可以调用同一个界面或者同一个组件？只能在 AppDelegate 里面注册单例来实现？

比如 App 出现问题了，用户可能在任何界面，如何随时随地的让用户强制登出？或者强制都跳转到同一个本地的 error 界面？或者跳转到相应的 H5，ReactNative，Weex 界面？如何让用户在任何界面，随时随地的弹出一个 View？

以上这些问题其实都可以通过在 App 端设计一个路由来解决。那么我们怎么设计一个路由呢？

三. App 之间跳转实现

在谈 App 内部的路由之前，先来谈谈在 iOS 系统间，不同 App 之间是怎么实现跳转的。

1. URL Scheme 方式

iOS 系统是默认支持 URL Scheme 的，具体见[官方文档](#)。

比如说，在 iPhone 的 Safari 浏览器上面输入如下的命令，会自动打开一些 App：

```
// 打开邮箱  
mailto://  
  
// 给110拨打电话  
tel://110
```

在 iOS 9 之前只要在 App 的 info.plist 里面添加 URL types - URL Schemes，如下图：

Key	Type	Value
Bundle name	String	`\${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	VERSION
Bundle creator OS Type code	String	????
CFBundleURLSchemes	String	com.quatanium.ios.homer
▼ URL types	Array	(2 items)
▼ Item 0 (Editor)	Dictionary	(3 items)
Document Role	String	Editor
URL identifier	String	com.quatanium.ios.homer
▼ URL Schemes	Array	(1 item)
Item 0	String	com.ios.Qhomer
► Item 1 (Editor)	Dictionary	(3 items)
Bundle version	String	BUILD

这里就添加了一个 com.ios.Qhomer 的 Scheme。这样就可以在 iPhone 的 Safari 浏览器上面输入：

```
com.ios.Qhomer://
```

就可以直接打开这个 App 了。

关于其他一些常见的 App，可以从 iTunes 里面下载到它的 ipa 文件，解压，显示包内容里面可以找到 info.plist 文件，打开它，在里面就可以相应的 URL Scheme。

```
// 手机QQ
```

```
mqq://
```

```
// 微信
```

```
weixin://
```

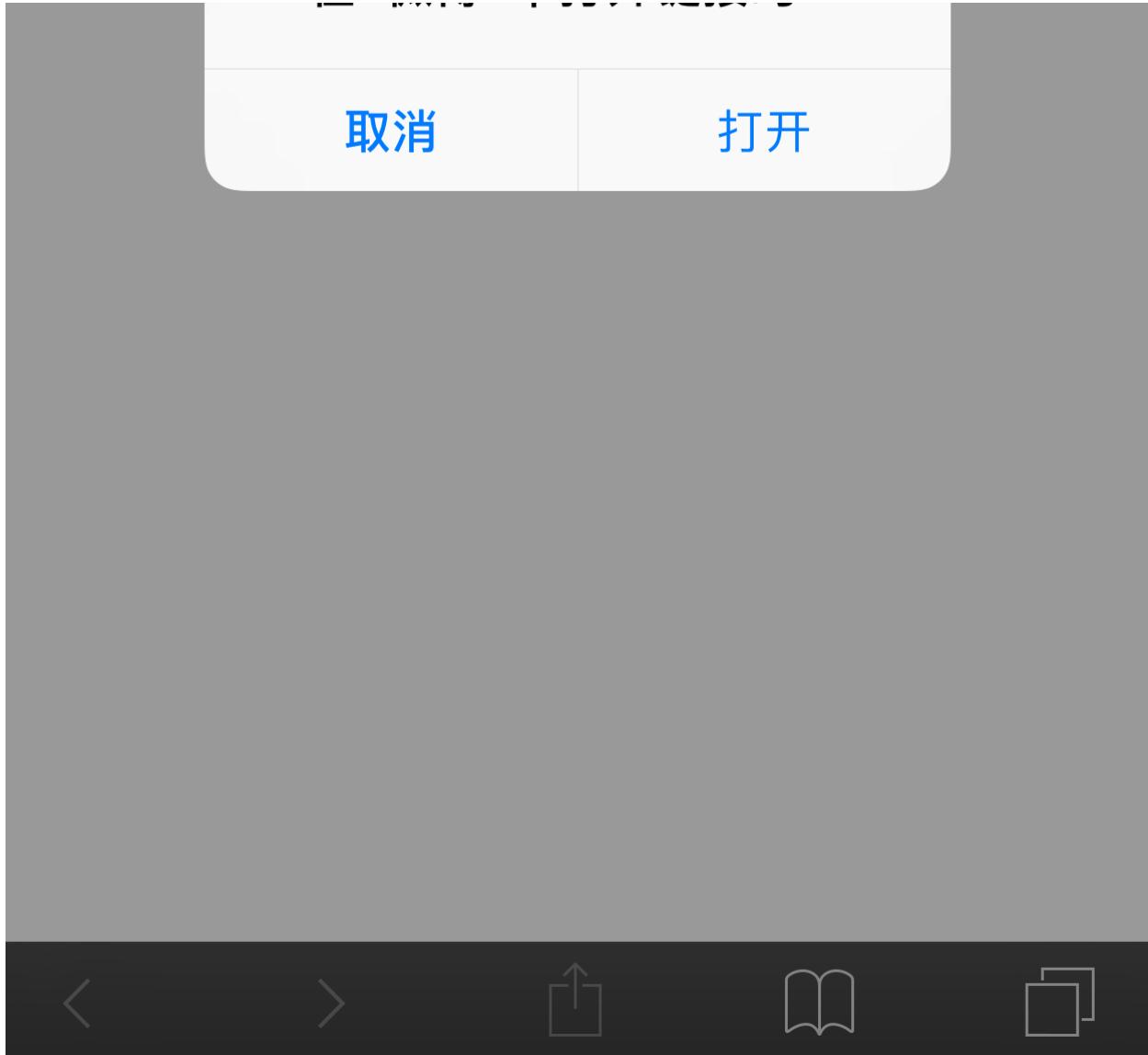
```
// 新浪微博
```

```
sinaweibo://
```

```
// 饿了么
```

```
eleme://
```



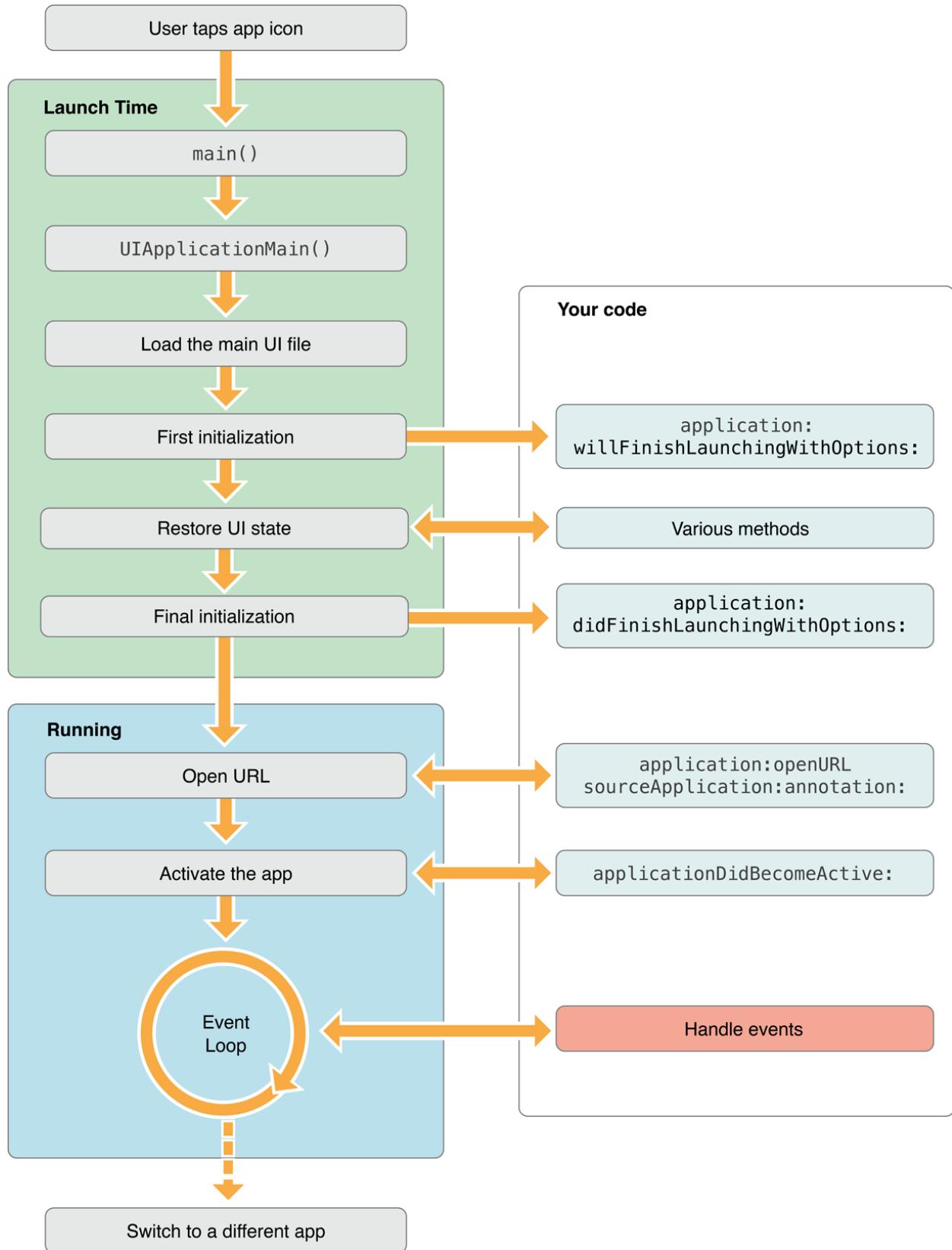


当然了，某些 App 对于调用 URL Scheme 比较敏感，它们不希望其他的 App 随意的就调用自己。

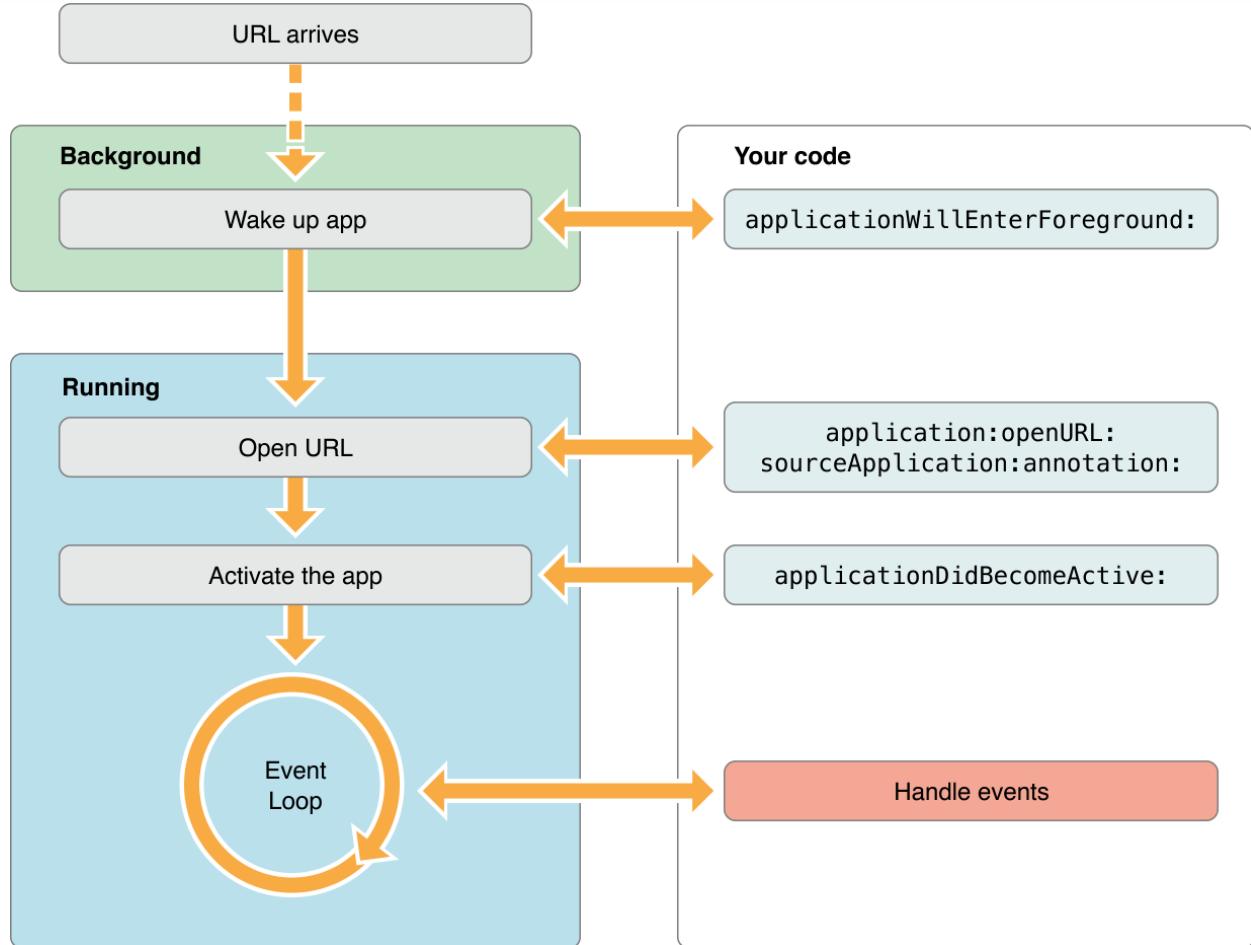
```
- (BOOL)application:(UIApplication *)application
              openURL:(NSURL *)url
            sourceApplication:(NSString *)sourceApplication
               annotation:(id)annotation
{
    NSLog(@"sourceApplication: %@", sourceApplication);
    NSLog(@"URL scheme:%@", [url scheme]);
    NSLog(@"URL query: %@", [url query]);

    if ([sourceApplication isEqualToString:@"com.tencent.weixin"]){
        // 允许打开
        return YES;
    }else{
        return NO;
    }
}
```

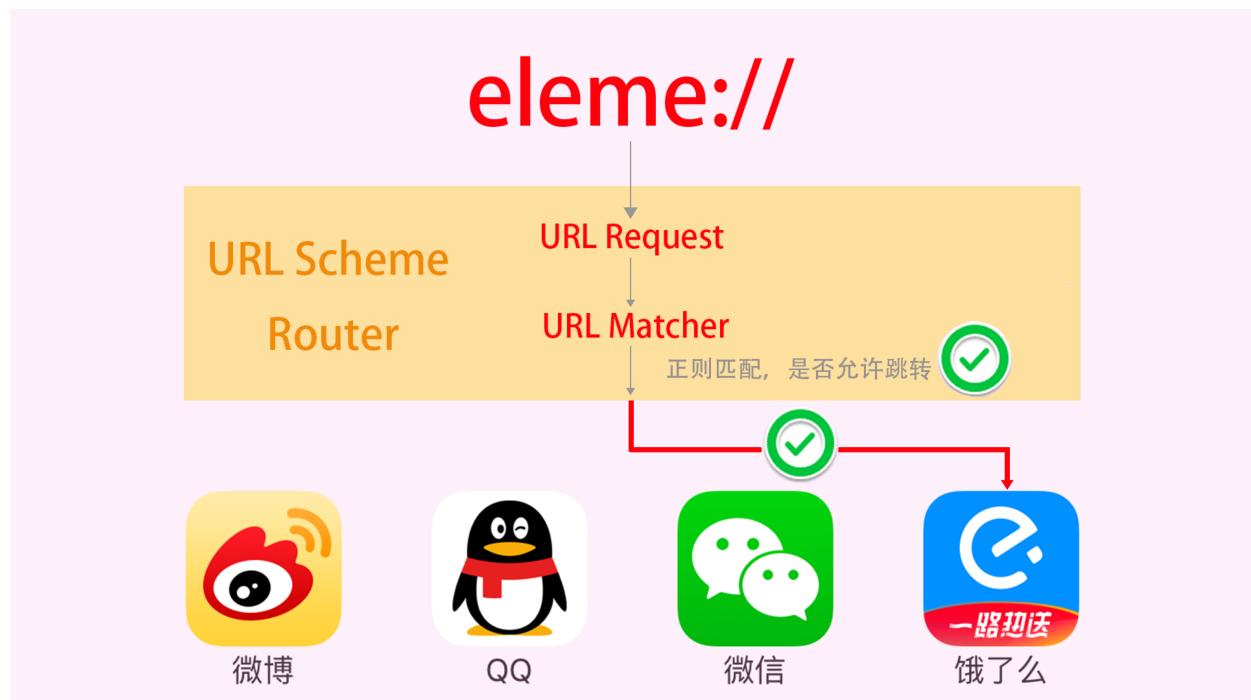
如果待调用的 App 已经运行了，那么它的生命周期如下：



如果待调用的 App 在后台，那么它的生命周期如下：



明白了上面的生命周期之后，我们就可以通过调用[application:openURL:sourceApplication:annotation:](#)这个方法，来阻止一些 App 的随意调用。





如上图，饿了么 App 允许通过 URL Scheme 调用，那么我们可以在 Safari 里面调用到饿了么 App。手机 QQ 不允许调用，我们在 Safari 里面也就没法跳转过去。

关于 App 间的跳转问题，感兴趣的可以查看官方文档[Inter-App Communication](#)。

App 也是可以直接跳转到系统设置的。比如有些需求要求检测用户有没有开启某些系统权限，如果没有开启就弹框提示，点击弹框的按钮直接跳转到系统设置里面对应的设置界面。

[iOS 10 支持通过 URL Scheme 跳转到系统设置](#)

[iOS10跳转系统设置的正确姿势](#)

[关于 iOS 系统功能的 URL 汇总列表](#)

2. Universal Links 方式

虽然在微信内部开网页会禁止所有的 Scheme，但是 iOS 9.0 新增加了一项功能是 Universal Links，使用这个功能可以使我们的 App 通过 HTTP 链接来启动 App。

1. 如果安装过 App，不管在微信里面 http 链接还是在 Safari 浏览器，还是其他第三方浏览器，都可以打开 App。
2. 如果没有安装过 App，就会打开网页。

具体设置需要3步：

1. App 需要开启 Associated Domains 服务，并设置 Domains，注意必须要 applinks: 开头。

Domains	Status
applinks:www.ele.me/home/	ON

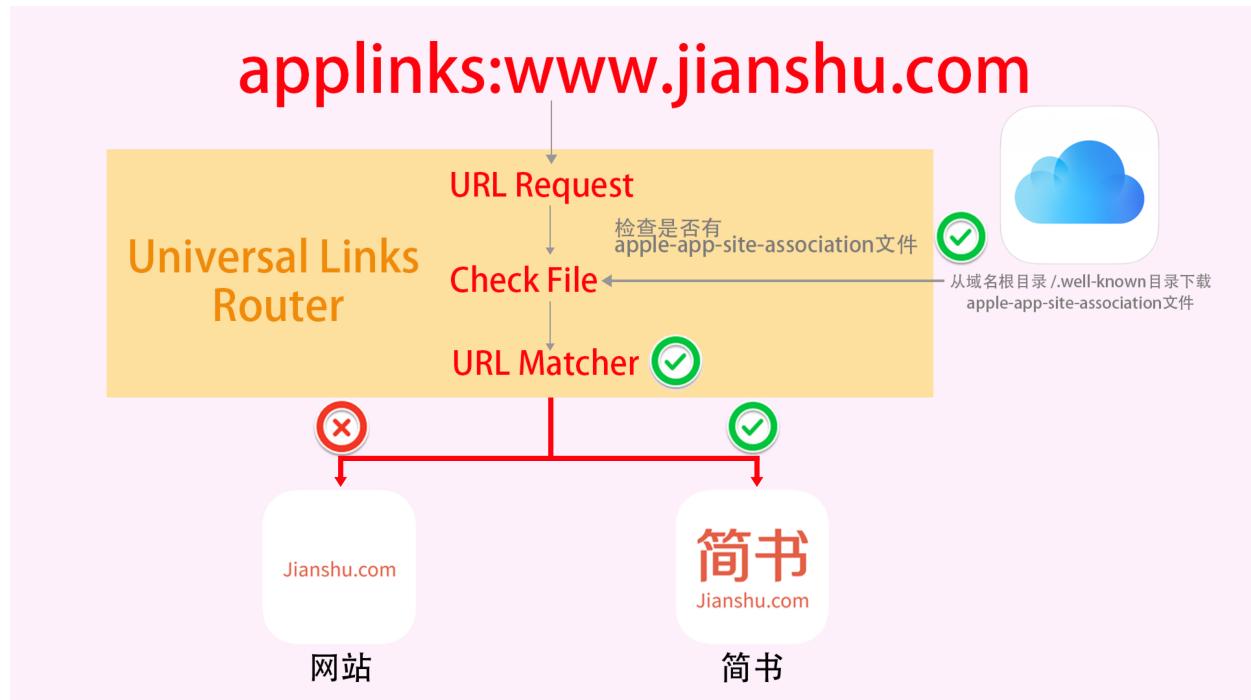
Domains: applinks:www.ele.me/home/

+

Steps: ✓ Add the Associated Domains entitlement to your entitlements file
① Add the Associated Domains feature to your App ID.

2. 域名必须要支持 HTTPS。

3. 上传内容是 Json 格式的文件，文件名为 apple-app-site-association 到自己域名的根目录下，或者 .well-known 目录下。iOS 自动会去读取这个文件。具体的文件内容请查看[官方文档](#)。



如果 App 支持了 Universal Links 方式，那么可以在其他 App 里面直接跳转到我们自己的 App 里面。如下图，点击链接，由于该链接会 Matcher 到我们设置的链接，所以菜单里面会显示用我们的 App 打开。



<http://www.jianshu.com/u/12201cdd5d7a>

在 Safari 中打开

在“简书”中打开

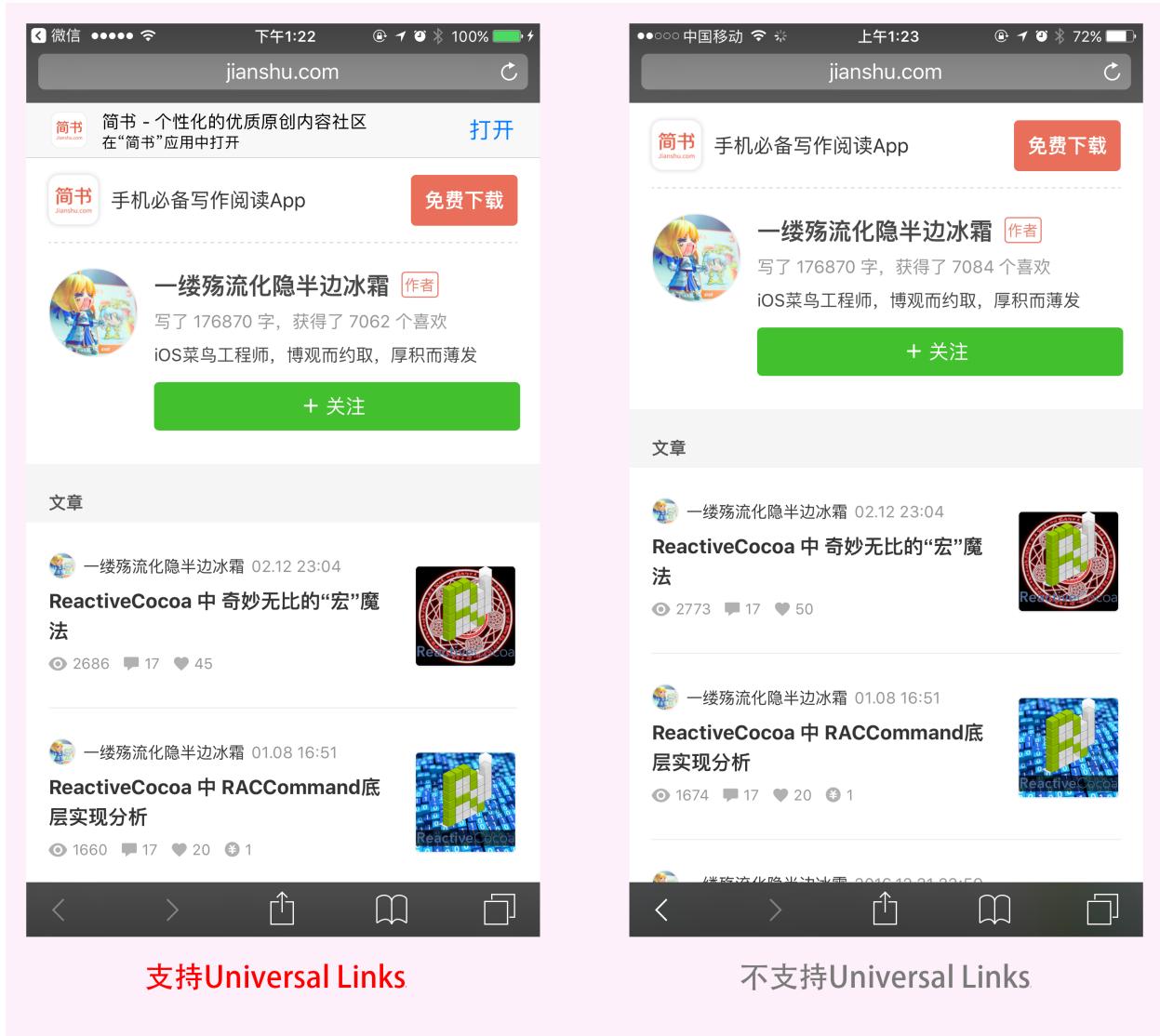
加入阅读列表

拷贝

共享...

取消

在浏览器里面也是一样的效果，如果是支持了 Universal Links 方式，访问相应的 URL，会有不同的效果。如下图：



以上就是iOS系统中 App 间跳转的二种方式。

从 iOS 系统里面支持的 URL Scheme 方式，我们可以看出，对于一个资源的访问，苹果也是用 URI 的方式来访问的。

统一资源标识符（英语：Uniform Resource Identifier，或URI)是一个用于**标识**某一互联网资源名称的**字符串**。该种标识允许用户对网络中（一般指万维网）的资源通过特定的**协议**进行交互操作。URI的最常见的形式是**统一资源定位符**（URL）。

举个例子：

https	//	username:password	@	example.com	:123	/path/data	? key=value&key2=value2	# fragid1
Scheme		user information		host	port	path	query	fragment

这是一段 URI，每一段都代表了对应的含义。对方接收到了这样一段字符串，按照规则解析出来，就能获取到所有的有用信息。

这个能给我们设计 App 组件间的路由带来一些思路么？如果我们想要定义一个三端（iOS, Android, H5）的统一访问资源的方式，能用 URI 的这种方式实现么？

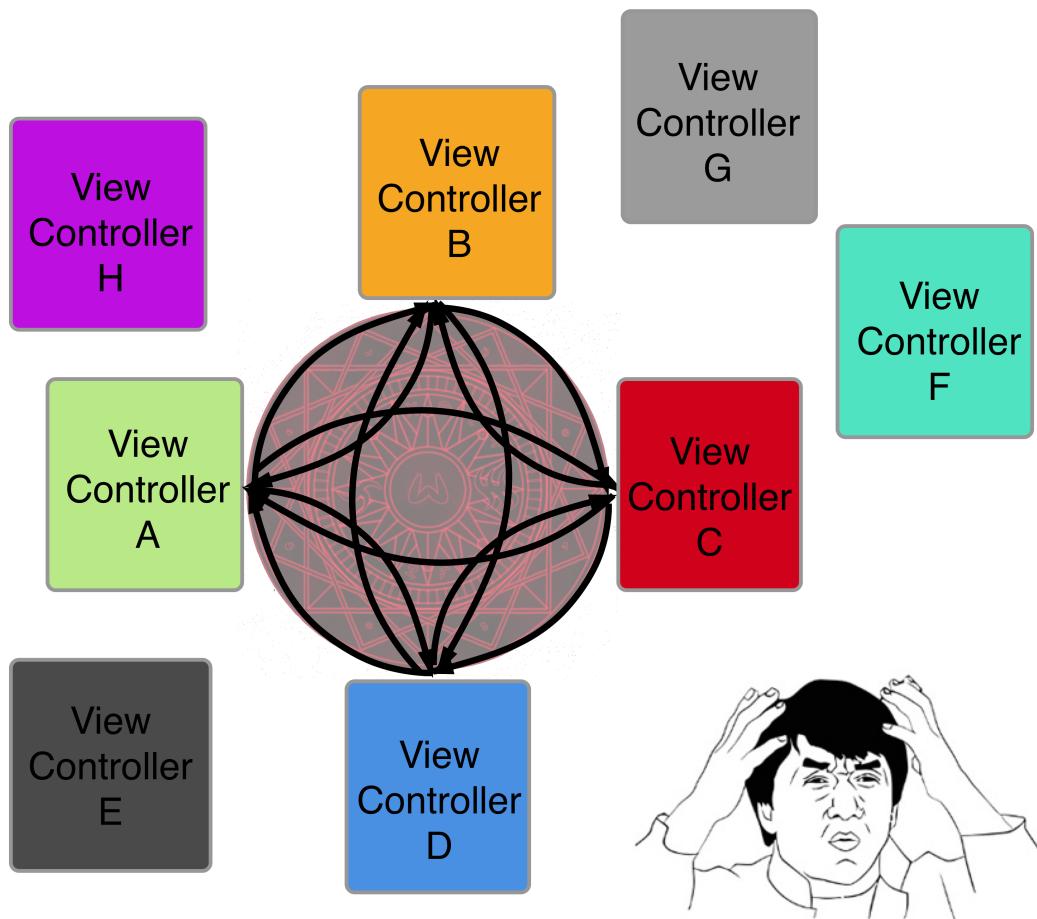
四. App 内组件间路由设计

上一章节中我们介绍了 iOS 系统中，系统是如何帮我们处理 App 间跳转逻辑的。这一章节我们着重讨论一下，App 内部，各个组件之间的路由应该怎么设计。关于 App 内部的路由设计，主要需要解决2个问题：

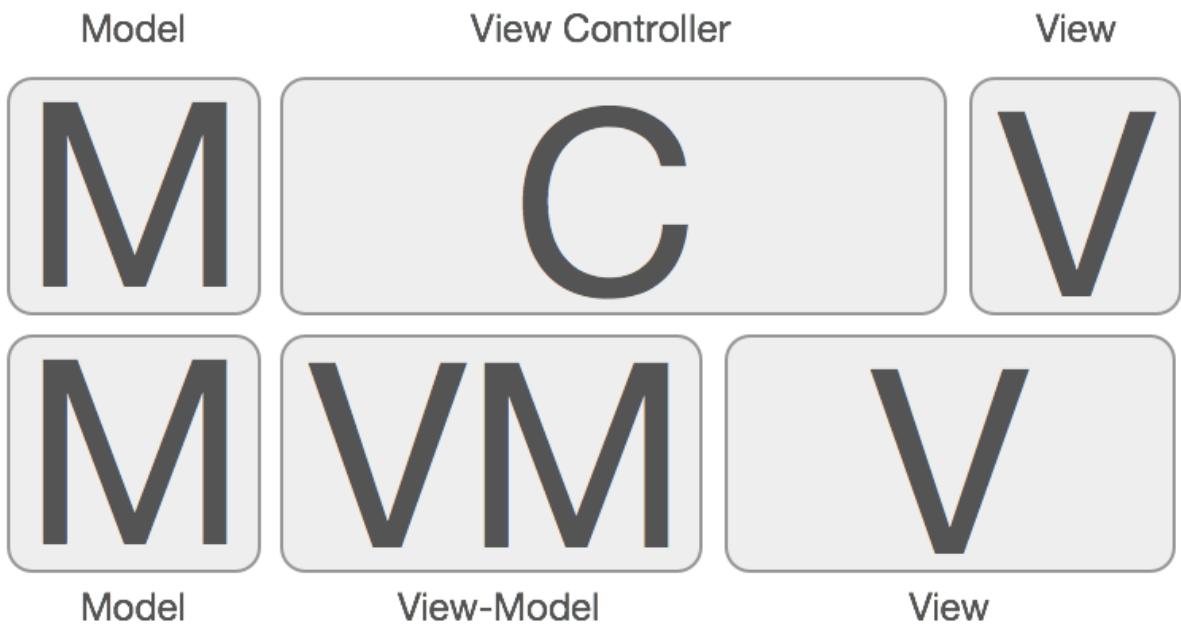
- 1.各个页面和组件之间的跳转问题。
- 2.各个组件之间相互调用。

先来分析一下这两个问题。

1. 关于页面跳转



在 iOS 开发的过程中，经常会遇到以下的场景，点击按钮跳转 Push 到另外一个界面，或者点击一个 cell Present 一个新的ViewController。在 MVC 模式中，一般都是新建一个 VC，然后Push / Present 到下一个 VC。但是在 MVVM 中，会有一些不合适的情况。



众所周知，MVVM 把 MVC 拆成了上图演示的样子，原来 View 对应的与数据相关的代码都移到 ViewModel 中，相应的 C 也变瘦了，演变成了 M-VM-C-V 的结构。这里的 C 里面的代码可以只剩下页面跳转相关的逻辑。如果用代码表示就是下面这样子：

假设一个按钮的执行逻辑都封装成了 command。

```

@weakify(self);
[[[_viewModel.someCommand executionSignals] flatten] subscribeNext:^(id
x) {
    @strongify(self);
    // 跳转逻辑
    [self.navigationController pushViewController:targetViewController
animated:YES];
}];

```

上述的代码本身没啥问题，但是可能会弱化 MVVM 框架的一个重要作用。

MVVM 框架的目的除去解耦以外，还有2个很重要的目的：

1. 代码高复用率
2. 方便进行单元测试

如果需要测试一个业务是否正确，我们只要对 ViewModel 进行单元测试即可。前提是假定我们使用 ReactiveCocoa 进行 UI 绑定的过程是准确无误的。目前绑定是正确的。所以我们只需要单元测试到 ViewModel 即可完成业务逻辑的测试。

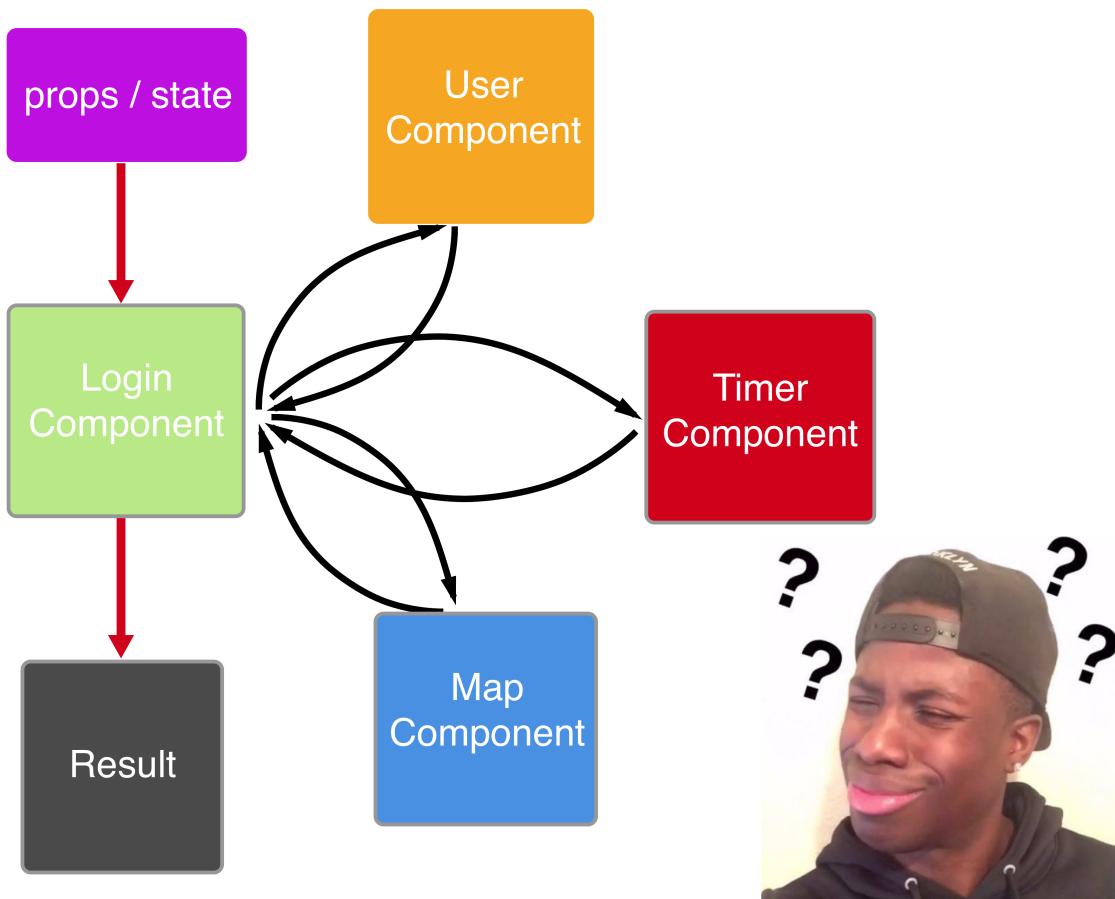
页面跳转也属于业务逻辑，所以应该放在 ViewModel 中一起单元测试，保证业务逻辑测试的覆盖率。

把页面跳转放到 ViewModel 中，有2种做法，第一种就是用路由来实现，第二种由于和路由没有关系，所以这里就不多阐述，有兴趣的可以看[lpd-mvvm-kit](#)这个库关于页面跳转的具体实现。

页面跳转相互的耦合性也就体现出来了：

- 1.由于 pushViewController 或者 presentViewController，后面都需要带一个待操作的 ViewController，那么就必须要引入该类，import 头文件也就引入了耦合性。
- 2.由于跳转这里写死了跳转操作，如果线上一旦出现了 bug，这里是不受我们控制的。
- 3.推送消息或者是 3D-Touch 需求，要求直接跳转到内部第10级界面，那么就需要写一个入口跳转到指定界面。

2. 关于组件间调用



关于组件间的调用，也需要解耦。随着业务越来越复杂，我们封装的组件越来越多，要是封装的粒度拿捏不准，就会出现大量组件之间耦合度高的问题。组件的粒度可以随着业务的调整，不断的调整组件职责的划分。但是组件之间的调用依旧不可避免，相互调用对方组件暴露的接口。如何减少各个组件之间的耦合度，是一个设计优秀的路由的职责所在。

3. 如何设计一个路由

如何设计一个能完美解决上述2个问题的路由，让我们先来看看 GitHub 上优秀开源库的设计思路。以下是我从 Github 上面找的一些路由方案，按照 Star 从高到低排列。依次来分析一下它们各自的设计思路。

(1) [JLRoutes](#) Star 3189

JLRoutes 在整个 Github 上面 Star 最多，那就来从它来分析分析它的具体设计思路。

首先 JLRoutes 是受 URL Scheme 思路的影响。它把所有对资源的请求看成是一个 URI。

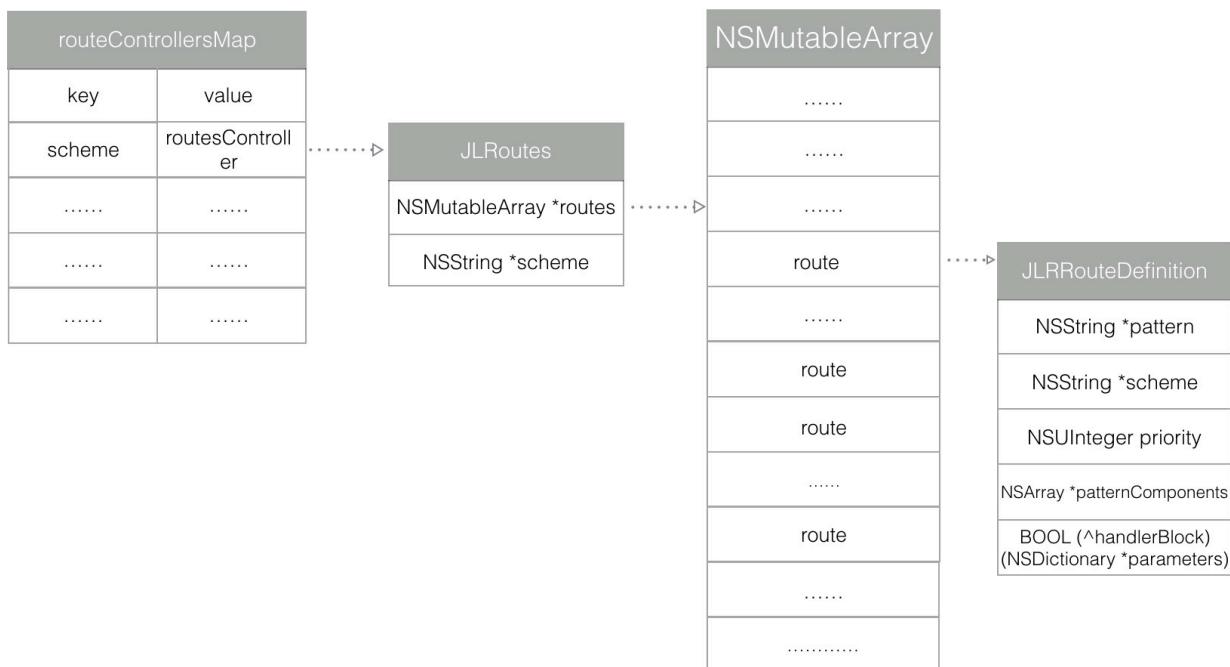
首先来熟悉一下 NSURLConnection 的各个字段：



Note

The URLs employed by the NSURL class are described in [RFC 1808](#), [RFC 1738](#), and [RFC 2732](#).

JLRoutes 会传入每个字符串，都按照上面的样子进行切分处理，分别根据 RFC 的标准定义，取到各个 NSURLConnection 的各个字段：



JLRoutes 全局会保存一个 Map，这个 Map 会以 scheme 为 Key，JLRoutes 为 Value。所以在 `routeControllerMap` 里面每个 scheme 都是唯一的。

至于为何有这么多条路由，笔者认为，如果路由按照业务线进行划分的话，每个业务线可能会有不相同的逻辑，即使每个业务里面的组件名字可能相同，但是由于业务线不同，会有不同的路由规则。

举个例子：如果滴滴按照每个城市的打车业务进行组件化拆分，那么每个城市就对应着这里的每个 scheme。每个城市的打车业务都有叫车，付款.....等业务，但是由于每个城市的地方法规不相同，所以这些组件即使名字相同，但是里面的功能也许千差万别。所以这里划分出了多个 route，也可以理解为不同的命名空间。

在每个 JLRoutes 里面都保存了一个数组，这个数组里面保存了每个路由规则 JLRRouteDefinition 里面会保存外部传进来的 block 闭包，pattern，和拆分之后的 pattern。

在每个 JLRoutes 的数组里面，会按照路由的优先级进行排列，优先级高的排列在前面。

```
- (void)_registerRoute:(NSString *)routePattern priority:
(NSUInteger)priority handler:(BOOL (^)(NSDictionary
*parameters))handlerBlock
{
    JLRRouteDefinition *route = [[JLRRouteDefinition alloc]
initWithScheme:self.scheme pattern:routePattern priority:priority
handlerBlock:handlerBlock];

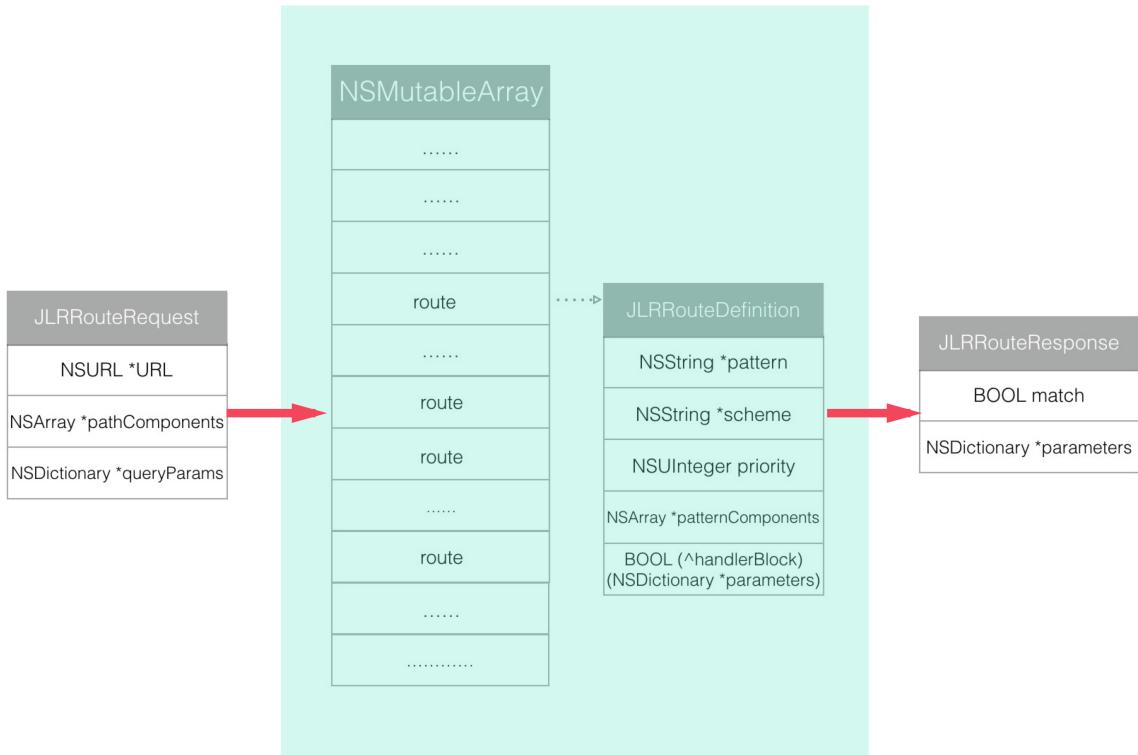
    if (priority == 0 || self.routes.count == 0) {
        [self.routes addObject:route];
    } else {
        NSUInteger index = 0;
        BOOL addedRoute = NO;

        // 找到当前已经存在的一条优先级比当前待插入的路由低的路由
        for (JLRRouteDefinition *existingRoute in [self.routes copy]) {
            if (existingRoute.priority < priority) {
                // 如果找到，就插入数组
                [self.routes insertObject:route atIndex:index];
                addedRoute = YES;
                break;
            }
            index++;
        }

        // 如果没有找到任何一条路由比当前待插入的路由低的路由，或者最后一条路由优先级和当前路由一样，那么就只能插入到最后。
        if (!addedRoute) {
            [self.routes addObject:route];
        }
    }
}
```

由于这个数组里面的路由是一个单调队列，所以查找优先级的时候只用从高往低遍历即可。

具体查找路由的过程如下：



首先根据外部传进来的URL初始化一个 `JLRRouteRequest`, 然后用这个 `JLRRouteRequest` 在当前的路由数组里面依次 request, 每个规则都会生成一个 response, 但是只有符合条件的 response 才会 match, 最后取出匹配的 `JLRRouteResponse` 拿出其字典 parameters 里面对应的参数就可以了。查找和匹配过程中重要的代码如下:

```

- (BOOL)_routeURL:(NSURL *)URL withParameters:(NSDictionary *)parameters
executeRouteBlock:(BOOL)executeRouteBlock
{
    if (!URL) {
        return NO;
    }

    [self _verboseLog:@"Trying to route URL %@", URL];

    BOOL didRoute = NO;
    JLRRouteRequest *request = [[JLRRouteRequest alloc] initWithURL:URL];

    for (JLRRouteDefinition *route in [self.routes copy]) {
        // 检查每一个route, 生成对应的response
        JLRRouteResponse *response = [route routeResponseForRequest:request
decodePlusSymbols:shouldDecodePlusSymbols];
        if (!response.isMatch) {
            continue;
        }

        [self _verboseLog:@"Successfully matched %@", route];
    }
}

```

```

    if (!executeRouteBlock) {
        // 如果我们被要求不允许执行，但是又找了匹配的路由response。
        return YES;
    }

    // 装配最后的参数
    NSMutableDictionary *finalParameters = [NSMutableDictionary
dictionary];
    [finalParameters addEntriesFromDictionary:response.parameters];
    [finalParameters addEntriesFromDictionary:parameters];
    [self _verboseLog:@"Final parameters are %@", finalParameters];

    didRoute = [route callHandlerBlockWithParameters:finalParameters];

    if (didRoute) {
        // 调用Handler成功
        break;
    }
}

if (!didRoute) {
    [self _verboseLog:@"Could not find a matching route"];
}

// 如果在当前路由规则里面没有找到匹配的路由，当前路由不是global 的，并且允许降级到
global里面去查找，那么我们继续在global的路由规则里面去查找。
if (!didRoute && self.shouldFallbackToGlobalRoutes && ![self
_isGlobalRoutesController]) {
    [self _verboseLog:@"Falling back to global routes..."];
    didRoute = [[JLRoutes globalRoutes] _routeURL:URL
withParameters:parameters executeRouteBlock:executeRouteBlock];
}

// 最后，依旧没有找到任何能匹配的，如果有unmatched URL handler，调用这个闭包进行
最后的处理。

if, after everything, we did not route anything and we have an unmatched URL
handler, then call it

if (!didRoute && executeRouteBlock && self.unmatchedURLHandler) {
    [self _verboseLog:@"Falling back to the unmatched URL handler"];
    self.unmatchedURLHandler(self, URL, parameters);
}

return didRoute;
}

```

举个例子：

我们先注册一个 Router，规则如下：

```
[[JLRoutes globalRoutes] addRoute:@"/:object/:action"
handler:^BOOL(NSDictionary *parameters) {
    NSString *object = parameters[@"object"];
    NSString *action = parameters[@"action"];
    // stuff
    return YES;
}];
```

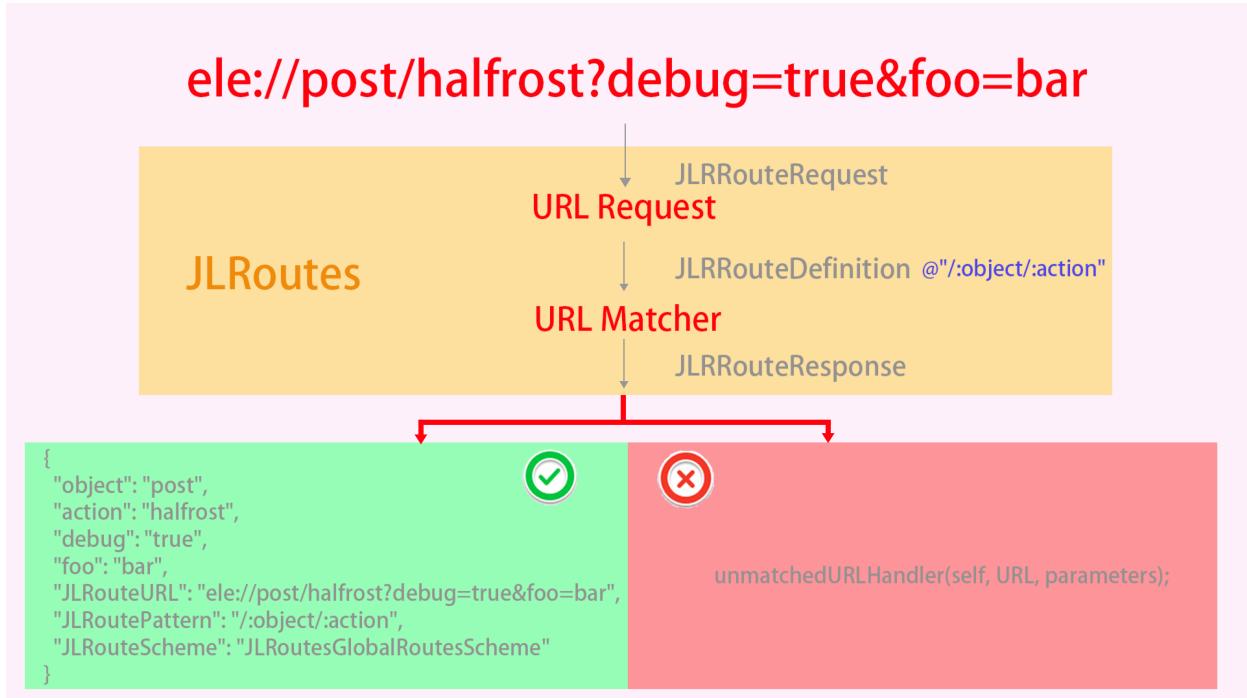
我们传入一个 URL，让 Router 进行处理。

```
NSURL *editPost = [NSURL URLWithString:@"ele://post/halfrost?
debug=true&foo=bar"];
[[UIApplication sharedApplication] openURL:editPost];
```

匹配成功之后，我们会得到下面这样一个字典：

```
{
    "object": "post",
    "action": "halfrost",
    "debug": "true",
    "foo": "bar",
    "JLRouteURL": "ele://post/halfrost?debug=true&foo=bar",
    "JLRoutePattern": "/:object/:action",
    "JLRouteScheme": "JLRoutesGlobalRoutesScheme"
}
```

把上述过程图解出来，见下图：



JLRoutes 还可以支持 Optional 的路由规则，假如定义一条路由规则：

```
/the(/foo/:a)(/bar/:b)
```

JLRoutes 会帮我们默认注册如下4条路由规则：

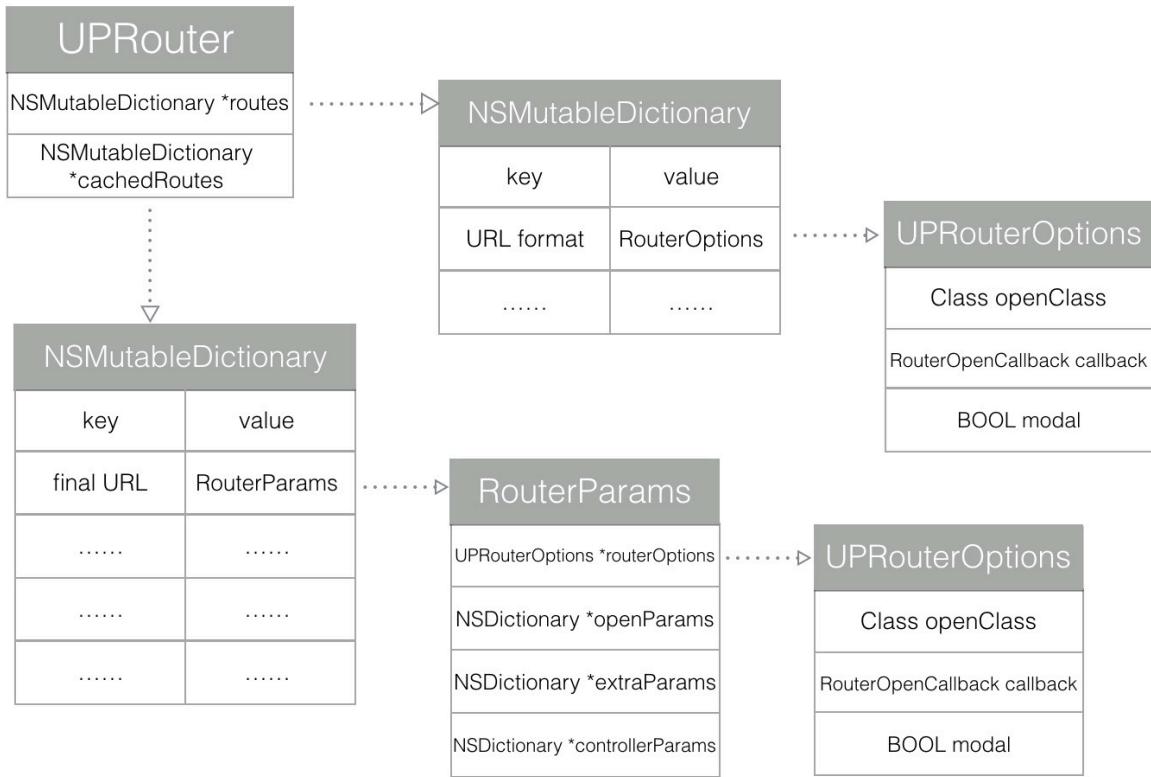
```

/the/foo/:a/bar/:b
/the/foo/:a
/the/bar/:b
/the

```

(2) [routable-ios](#) Star 1415

Routable 路由是用在 in-app native 端的 URL router, 它可以用在 iOS 上也可以用在 [Android](#) 上。



UPRouter 里面保存了2个字典。routes 字典里面存储的 Key 是路由规则，Value 存储的是 UPRouterOptions。cachedRoutes 里面存储的 Key 是最终的 URL，带传参的，Value 存储的是 RouterParams。RouterParams 里面会包含在 routes 匹配的到的 UPRouterOptions，还有额外的打开参数 openParams 和一些额外参数 extraParams。

```

- (RouterParams *)routerParamsForUrl:(NSString *)url extraParams:
(NSDictionary *)extraParams {
    if (!url) {
        //if we wait, caching this as key would throw an exception
        if (_ignoresExceptions) {
            return nil;
        }
        @throw [NSEException exceptionWithName:@"RouteNotFoundException"
                                         reason:[NSString
stringWithFormat:ROUTE_NOT_FOUND_FORMAT, url]
                                         userInfo:nil];
    }

    if ([self.cachedRoutes objectForKey:url] && !extraParams) {
        return [self.cachedRoutes objectForKey:url];
    }

    // 比对url通过/分割之后的参数个数和pathComponents的个数是否一样
}

```

// 比对url通过/分割之后的参数个数和pathComponents的个数是否一样

```

NSArray *givenParts = url.pathComponents;
NSArray *legacyParts = [url componentsSeparatedByString:@"/"];
if ([legacyParts count] != [givenParts count]) {
    NSLog(@"Routable Warning - your URL %@ has empty path components -
this will throw an error in an upcoming release", url);
    givenParts = legacyParts;
}

__block RouterParams *openParams = nil;
[self.routes enumerateKeysAndObjectsUsingBlock:
 ^(NSString *routerUrl, UPRouterOptions *routerOptions, BOOL *stop) {

    NSArray *routerParts = [routerUrl pathComponents];
    if ([routerParts count] == [givenParts count]) {

        NSDictionary *givenParams = [self
paramsForUrlComponents:givenParts routerUrlComponents:routerParts];
        if (givenParams) {
            openParams = [[RouterParams alloc]
initWithRouterOptions:routerOptions openParams:givenParams extraParams:
extraParams];
            *stop = YES;
        }
    }
}];

if (!openParams) {
    if (_ignoresExceptions) {
        return nil;
    }
    @throw [NSEException exceptionWithName:@"RouteNotFoundException"
reason:[NSString
stringWithFormat:ROUTE_NOT_FOUND_FORMAT, url]
userInfo:nil];
}
[self.cachedRoutes setObject:openParams forKey:url];
return openParams;
}

```

这一段代码里面重点在干一件事情，遍历 routes 字典，然后找到参数匹配的字符串，封装成 RouterParams 返回。

```

- (NSDictionary *)paramsForUrlComponents:(NSArray *)givenUrlComponents
routerUrlComponents:(NSArray *)routerUrlComponents {

    __block NSMutableDictionary *params = [NSMutableDictionary dictionary];
    [routerUrlComponents enumerateObjectsUsingBlock:
     ^(NSString *routerComponent, NSUInteger idx, BOOL *stop) {

        NSString *givenComponent = givenUrlComponents[idx];
        if ([routerComponent hasPrefix:@":"]) {
            NSString *key = [routerComponent substringFromIndex:1];
            [params setObject:givenComponent forKey:key];
        }
        else if (![routerComponent isEqualToString:givenComponent]) {
            params = nil;
            *stop = YES;
        }
    }];
    return params;
}

```

上面这段函数，第一个参数是外部传进来 URL 带有各个入参的分割数组。第二个参数是路由规则分割开的数组。routerComponent 由于规定：号后面才是参数，所以 routerComponent 的第1个位置就是对应的参数名。params 字典里面以参数名为 Key，参数为 Value。

```

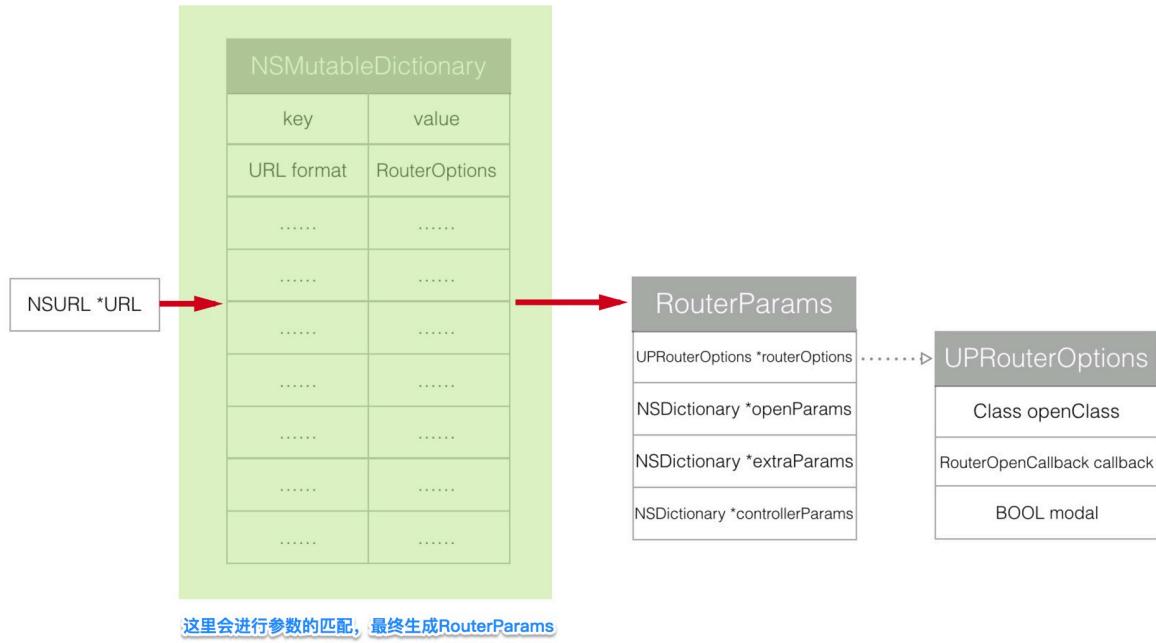
NSDictionary *givenParams = [self paramsForUrlComponents:givenParts
routerUrlComponents:routerParts];
if (givenParams) {
    openParams = [[RouterParams alloc]
initWithRouterOptions:routerOptions openParams:givenParams extraParams:
extraParams];
    *stop = YES;
}

```

最后通过 RouterParams 的初始化方法，把路由规则对应的 UPRouterOptions，上一步封装好的参数字典 givenParams，还有 routerParamsForUrl: extraParams: 方法的第二个入参，这3个参数作为初始化参数，生成了一个 RouterParams。

```
[self.cachedRoutes setObject:openParams forKey:url];
```

最后一步 self.cachedRoutes 的字典里面 Key 为带参数的 URL, Value 是 RouterParams。



最后将匹配封装出来的 RouterParams 转换成对应的 Controller。

```

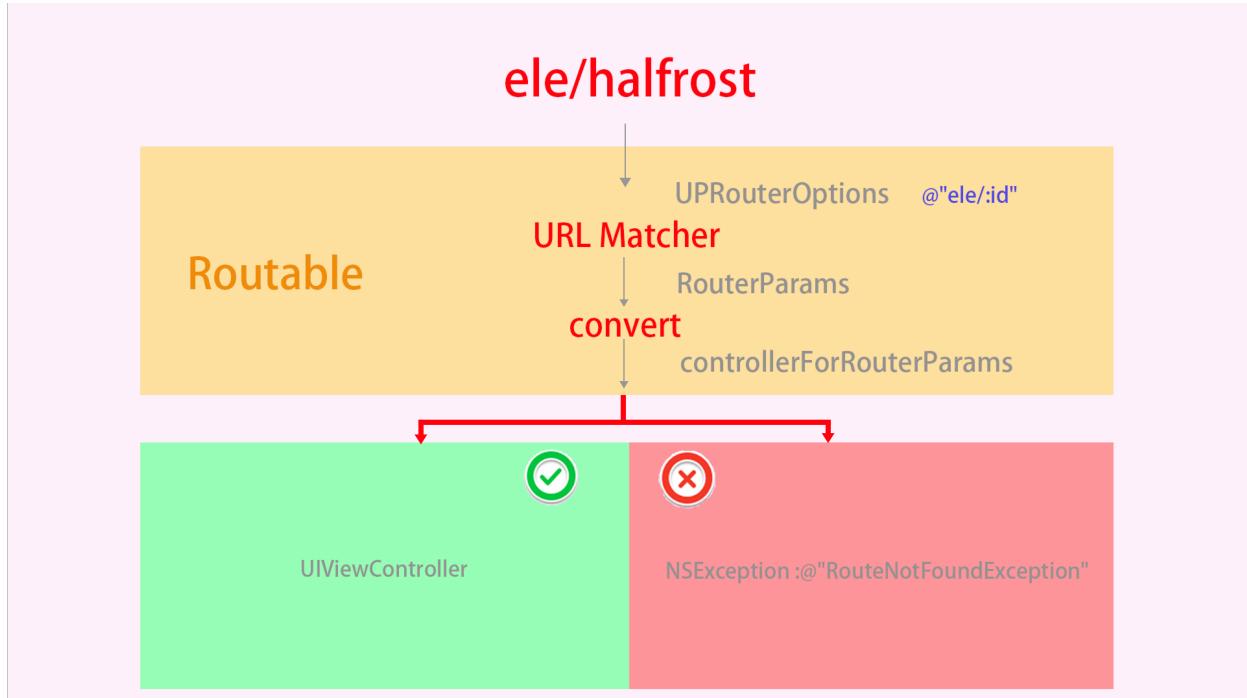
- (UIViewController *)controllerForRouterParams:(RouterParams *)params {
    SEL CONTROLLER_CLASS_SELECTOR =
sel_registerName("allocWithRouterParams:");
    SEL CONTROLLER_SELECTOR = sel_registerName("initWithRouterParams:");
    UIViewController *controller = nil;
    Class controllerClass = params.routerOptions.openClass;
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
    if ([controllerClass respondsToSelector:CONTROLLER_CLASS_SELECTOR]) {
        controller = [controllerClass
performSelector:CONTROLLER_CLASS_SELECTOR withObject:[params
controllerParams]];
    }
    else if ([params.routerOptions.openClass
instancesRespondToSelector:CONTROLLER_SELECTOR]) {
        controller = [[params.routerOptions.openClass alloc]
performSelector:CONTROLLER_SELECTOR withObject:[params controllerParams]];
    }
#pragma clang diagnostic pop
    if (!controller) {
        if (_ignoresExceptions) {
            return controller;
        }
        @throw [NSError exceptionWithName:@"RoutableInitializerNotFound"
reason:[NSString
stringWithFormat:INVALID_CONTROLLER_FORMAT,
NSStringFromClass(controllerClass),
NSStringFromSelector(CONTROLLER_CLASS_SELECTOR),
NSStringFromSelector(CONTROLLER_SELECTOR)]
userInfo:nil];
    }

    controller.modalTransitionStyle = params.routerOptions.transitionStyle;
    controller.modalPresentationStyle =
params.routerOptions.presentationStyle;
    return controller;
}

```

如果 Controller 是一个类，那么就调用 allocWithRouterParams: 方法去初始化。如果 Controller 已经是一个实例了，那么就调用 initWithRouterParams: 方法去初始化。

将 Routable 的大致流程图解如下：



(3) [HHRouter Star 1277](#)

这是布丁动画的一个 Router，灵感来自于 [ABRouter](#) 和 [Routable iOS](#)。

先来看看 HHRouter 的 Api。它提供的方法非常清晰。

ViewController 提供了2个方法。map是用来设置路由规则，matchController 是用来匹配路由规则的，匹配争取之后返回对应的 UIViewController。

```

- (void)map:(NSString *)route toControllerClass:(Class)controllerClass;
- (UIViewController *)matchController:(NSString *)route;
  
```

block 闭包提供了三个方法，map 也是设置路由规则，matchBlock: 是用来匹配路由，找到指定的 block，但是不会调用该 block。callBlock: 是找到指定的 block，找到以后就立即调用。

```

- (void)map:(NSString *)route toBlock:(HHRouterBlock)block;
- (HHRouterBlock)matchBlock:(NSString *)route;
- (id)callBlock:(NSString *)route;
  
```

matchBlock: 和 callBlock: 的区别就在于前者不会自动调用闭包。所以 matchBlock: 方法找到对应的 block 之后，如果想调用，需要手动调用一次。

除去上面这些方法，HHRouter 还为我们提供了一个特殊的方法。

```
- (HHRouteType)canRoute:(NSString *)route;
```

这个方法就是用来找到执行路由规则对应的 RouteType，RouteType 总共就3种：

```
typedef NS_ENUM (NSInteger, HHRouteType) {
    HHRouteTypeNone = 0,
    HHRouteTypeViewController = 1,
    HHRouteTypeBlock = 2
};
```

再来看看 HHRouter 是如何管理路由规则的。整个 HHRouter 就是由一个 NSMutableDictionary *routes 控制的。

```
@interface HHRouter ()
@property (strong, nonatomic) NSMutableDictionary *routes;
@end
```

HHRouter

NSMutableDictionary *routes

别看只有这一个看似“简单”的字典数据结构，但是 HHRouter 路由设计的还是很精妙的。

```

- (void)map:(NSString *)route toBlock:(HHRouterBlock)block
{
    NSMutableDictionary *subRoutes = [self subRoutesToRoute:route];
    subRoutes[@"_" ] = [block copy];
}

- (void)map:(NSString *)route toControllerClass:(Class)controllerClass
{
    NSMutableDictionary *subRoutes = [self subRoutesToRoute:route];
    subRoutes[@"_" ] = controllerClass;
}

```

上面两个方法分别是 block 闭包和 ViewController 设置路由规则调用的方法实体。不管是 ViewController 还是 block 闭包，设置规则的时候都会调用 subRoutesToRoute: 方法。

```

- (NSMutableDictionary *)subRoutesToRoute:(NSString *)route
{
    NSArray *pathComponents = [self pathComponentsFromRoute:route];

    NSInteger index = 0;
    NSMutableDictionary *subRoutes = self.routes;

    while (index < pathComponents.count) {
        NSString *pathComponent = pathComponents[index];
        if (![subRoutes objectForKey:pathComponent]) {
            subRoutes[pathComponent] = [[NSMutableDictionary alloc] init];
        }
        subRoutes = subRoutes[pathComponent];
        index++;
    }

    return subRoutes;
}

```

上面这段函数就是来构造路由匹配规则的字典。

举个例子：

```
[[HHRouter shared] map:@"/user/:userId/"
    toControllerClass:[UserViewController class]];
[[HHRouter shared] map:@"/story/:storyId/"
    toControllerClass:[StoryViewController class]];
[[HHRouter shared] map:@"/user/:userId/story/?a=0"
    toControllerClass:[StoryListViewController class]];
```

设置3条规则以后，按照上面构造路由匹配规则的字典的方法，该路由规则字典就会变成这个样子：

```
{
    story = {
        ":storyId" = {
            "_" = StoryViewController;
        };
    };
    user = {
        ":userId" = {
            "_" = UserViewController;
            story = {
                "_" = StoryListViewController;
            };
        };
    };
}
```

路由规则字典生成之后，等到匹配的时候就会遍历这个字典。

假设这时候有一条路由过来：

```
[[[HHRouter shared] matchController:@":hhrouter20://user/1/" ] class],
```

HHRouter 对这条路由的处理方式是先匹配前面的 scheme，如果连 scheme 都不正确的话，会直接导致后面匹配失败。

然后再进行路由匹配，最后生成的参数字典如下：

```
{
    "controller_class" = UserViewController;
    route = "/user/1/";
    userId = 1;
}
```

具体的路由参数匹配的函数在

```
- (NSDictionary *)paramsInRoute:(NSString *)route
```

这个方法里面实现的。这个方法就是按照路由匹配规则，把传进来的URL的参数都一一解析出来，带?号的也都会解析成字典。这个方法没什么难度，就不在赘述了。

ViewController 的字典里面默认还会加上2项：

```
"controller_class" =  
route =
```

route 里面都会保存传过来的完整的 URL。

如果传进来的路由后面带访问字符串呢？那我们再来看看：

```
[[HHRouter shared] matchController:@"/user/1/?a=b&c=d"]
```

那么解析出所有的参数字典会是下面的样子：

```
{  
    a = b;  
    c = d;  
    "controller_class" = UserViewController;  
    route = "/user/1/?a=b&c=d";  
    userId = 1;  
}
```

同理，如果是一个 block 闭包的情况呢？

还是先添加一条 block 闭包的路由规则：

```
[[HHRouter shared] map:@"/user/add/"  
    toBlock:^id(NSDictionary* params) {  
    }];
```

这条规则对应的会生成一个路由规则的字典。

```

{
    story = {
        ":storyId" = {
            "_" = StoryViewController;
        };
    };
    user = {
        ":userId" = {
            "_" = UserViewController;
            story = {
                "_" = StoryListViewController;
            };
        };
        add = {
            "_" = "<__NSMallocBlock__: 0x600000240480>";
        };
    };
}

```

注意 "_" 后面跟着是一个 block。

匹配 block 闭包的方式有两种。

```

// 1. 第一种方式匹配到对应的block之后，还需要手动调用一次闭包。
HHRouterBlock block = [[HHRouter shared] matchBlock:@"/user/add/?a=1&b=2"];
block(nil);

// 2. 第二种方式匹配block之后自动会调用改闭包。
[[HHRouter shared] callBlock:@"/user/add/?a=1&b=2"];

```

匹配出来的参数字典是如下：

```

{
    a = 1;
    b = 2;
    block = "<__NSMallocBlock__: 0x600000056b90>";
    route = "/user/add/?a=1&b=2";
}

```

block 的字典里面会默认加上下面这2项：

```
block =  
route =
```

route 里面都会保存传过来的完整的 URL。

生成的参数字典最终会被绑定到 ViewController 的 Associated Object 关联对象上。

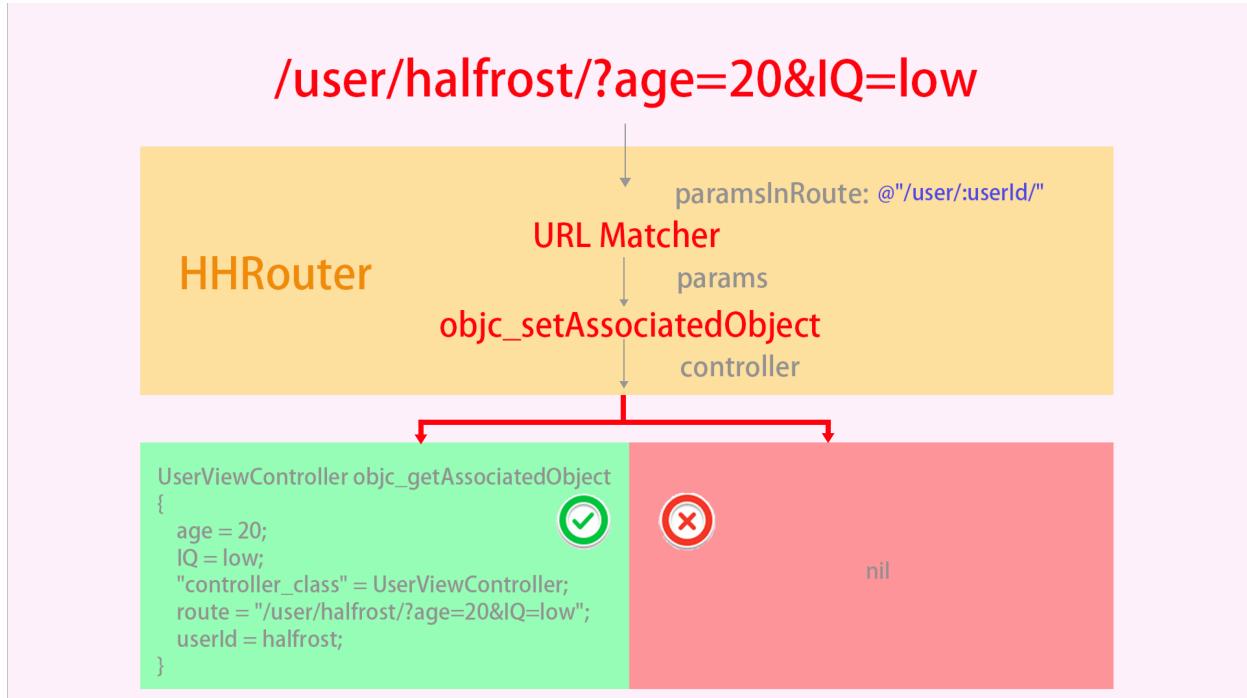
```
- (void)setParams:(NSDictionary *)paramsDictionary  
{  
    objc_setAssociatedObject(self, &kAssociatedParamsObjectKey,  
    paramsDictionary, OBJC_ASSOCIATION_RETAIN_NONATOMIC);  
  
}  
  
- (NSDictionary *)params  
{  
    return objc_getAssociatedObject(self, &kAssociatedParamsObjectKey);  
}
```

这个绑定的过程是在 match 匹配完成的时候进行的。

```
- (UIViewController *)matchController:(NSString *)route  
{  
    NSDictionary *params = [self paramsInRoute:route];  
    Class controllerClass = params[@"controller_class"];  
  
    UIViewController *viewController = [[controllerClass alloc] init];  
  
    if ([viewController respondsToSelector:@selector(setParams:)]) {  
        [viewController performSelector:@selector(setParams:)  
            withObject:[params copy]];  
    }  
    return viewController;  
}
```

最终得到的 ViewController 也是我们想要的。相应的参数都在它绑定的 params 属性的字典里面。

将上述过程图解出来，如下：



(4) [MGJRouter Star 633](#)

这是蘑菇街的一个路由的方法。

这个库的由来：

JLRoutes 的问题主要在于查找 URL 的实现不够高效，通过遍历而不是匹配。还有就是功能偏多。

HHRouter 的 URL 查找是基于匹配，所以会更高效，MGJRouter 也是采用的这种方法，但它跟 ViewController 绑定地过于紧密，一定程度上降低了灵活性。

于是就有了 MGJRouter。

从数据结构来看，MGJRouter 还是和 HHRouter 一模一样的。

```

@interface MGJRouter ()
@property (nonatomic) NSMutableDictionary *routes;
@end
  
```

MGJRouter

NSMutableDictionary *routes

那么我们就来看看它对 HHRouter 做了哪些优化改进。

1. MGJRouter 支持 openURL 时，可以传一些 userinfo 过去

```
[MGJRouter openURL:@"mgj://category/travel" userInfo:@{@"user_id": @1900} completion:nil];
```

这个对比 HHRouter，仅仅只是写法上的一个语法糖，在 HHRouter 中虽然不支持带字典的参数，但是在 URL 后面可以用 URL Query Parameter 来弥补。

```
if (parameters) {
    MGJRouterHandler handler = parameters[@"block"];
    if (completion) {
        parameters[MGJRouterParameterCompletion] = completion;
    }
    if (userInfo) {
        parameters[MGJRouterParameterUserInfo] = userInfo;
    }
    if (handler) {
        [parameters removeObjectForKey:@"block"];
        handler(parameters);
    }
}
```

MGJRouter 对 userInfo 的处理是直接把它封装到 Key = MGJRouterParameterUserInfo 对应的 Value 里面。

2. 支持中文的 URL

```
[parameters enumerateKeysAndObjectsUsingBlock:^(id key, NSString *obj,
BOOL *stop) {
    if ([obj isKindOfClass:[NSString class]]) {
        parameters[key] = [obj
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    }
}];
```

这里就是需要注意一下编码。

3. 定义一个全局的 URL Pattern 作为 Fallback

这一点是模仿的 JLRoutes 的匹配不到会自动降级到 global 的思想。

```
if (parameters) {
    MGJRouterHandler handler = parameters[@"block"];
    if (handler) {
        [parameters removeObjectForKey:@"block"];
        handler(parameters);
    }
}
```

parameters 字典里面会先存储下一个路由规则，存在 block 闭包中，在匹配的时候会取出这个 handler，降级匹配到这个闭包中，进行最终的处理。

4. 当 OpenURL 结束时，可以执行 Completion Block

在 MGJRouter 里面，作者对原来的 HHRouter 字典里面存储的路由规则的结构进行了改造。

```
NSString *const MGJRouterParameterURL = @"MGJRouterParameterURL";
NSString *const MGJRouterParameterCompletion =
@"MGJRouterParameterCompletion";
NSString *const MGJRouterParameterUserInfo = @"MGJRouterParameterUserInfo";
```

这3个 key 会分别保存一些信息：

MGJRouterParameterURL 保存的传进来的完整的 URL 信息。

MGJRouterParameterCompletion 保存的是 completion 闭包。

MGJRouterParameterUserInfo 保存的是 UserInfo 字典。

举个例子：

```
[MGJRouter registerURLPattern:@"ele://name/:name"
toHandler:^(NSDictionary *routerParameters) {
    void (^completion)(NSString *) =
    routerParameters[MGJRouterParameterCompletion];
    if (completion) {
        completion(@"完成了");
    }
}];

[MGJRouter openURL:@"ele://name/halfrost/?age=20"
withUserInfo:@{@"user_id": @1900} completion:^(id result) {
    NSLog(@"%@",result);
}];
```

上面的 URL 会匹配成功，那么生成的参数字典结构如下：

```
{
    MGJRouterParameterCompletion = "<__NSGlobalBlock__: 0x107ffe680>";
    MGJRouterParameterURL = "ele://name/halfrost/?age=20";
    MGJRouterParameterUserInfo = {
        "user_id" = 1900;
    };
    age = 20;
    block = "<__NSMallocBlock__: 0x608000252120>";
    name = halfrost;
}
```

5. 可以统一管理 URL

这个功能非常有用。

URL 的处理一不小心，就容易散落在项目的各个角落，不容易管理。比如注册时的 pattern 是 mgj://beauty/:id，然后 open 时就是 mgj://beauty/123，这样到时候 url 有改动，处理起来就会很麻烦，不好统一管理。

所以 MGJRouter 提供了一个类方法来处理这个问题。

```

#define TEMPLATE_URL @"qq://name/:name"

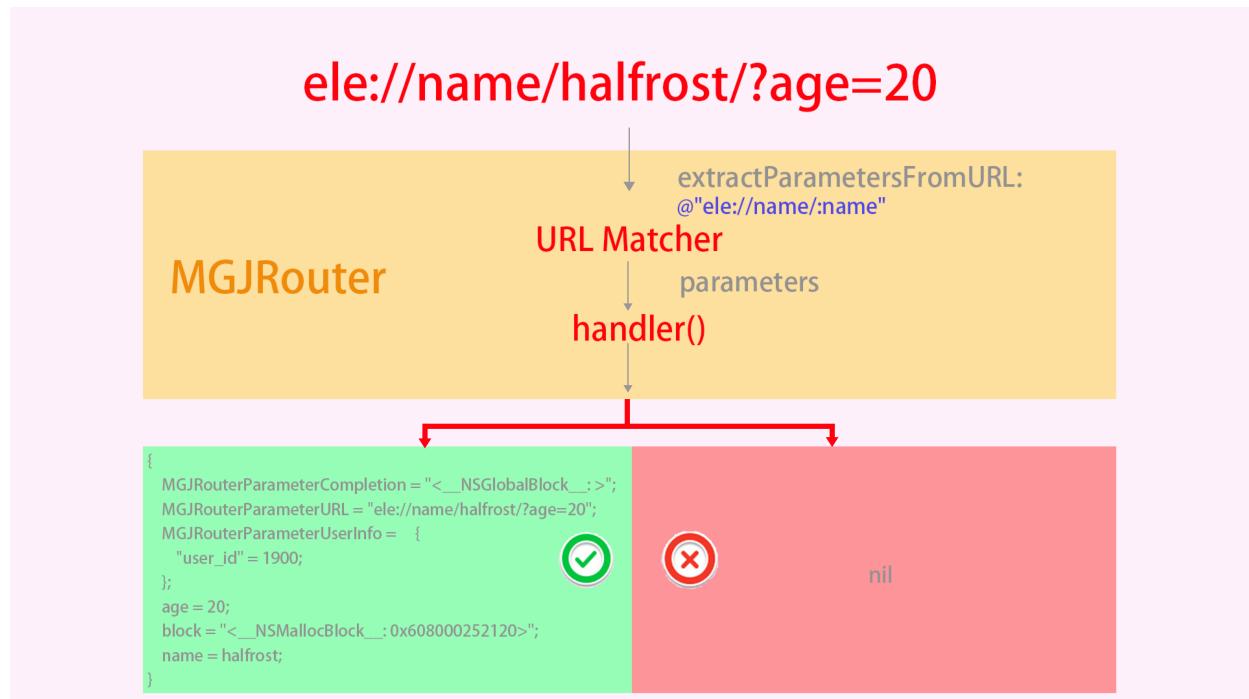
[MGJRouter registerURLPattern:TEMPLATE_URL toHandler:^(NSDictionary *routerParameters) {
    NSLog(@"routerParameters[name]:%@", routerParameters[@"name"]); // halfrost
}];

[MGJRouter openURL:[MGJRouter generateURLWithPattern:TEMPLATE_URL parameters:@[@"halfrost"]]];
}

```

generateURLWithPattern: 函数会对我们定义的宏里面的所有的:进行替换，替换成后面的字符串数组，依次赋值。

将上述过程图解出来，如下：

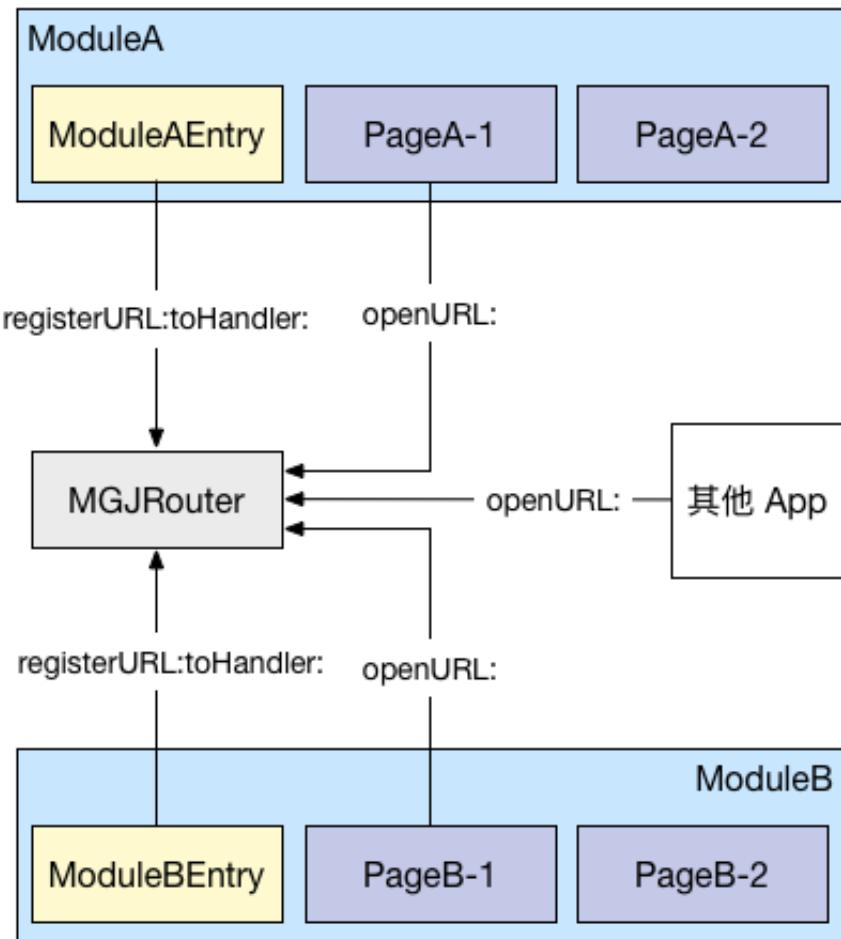


蘑菇街为了区分开页面间调用和组件间调用，于是想出了一种新的方法。用 Protocol 的方法来进行组件间的调用。

每个组件之间都有一个 Entry，这个 Entry，主要做了三件事：

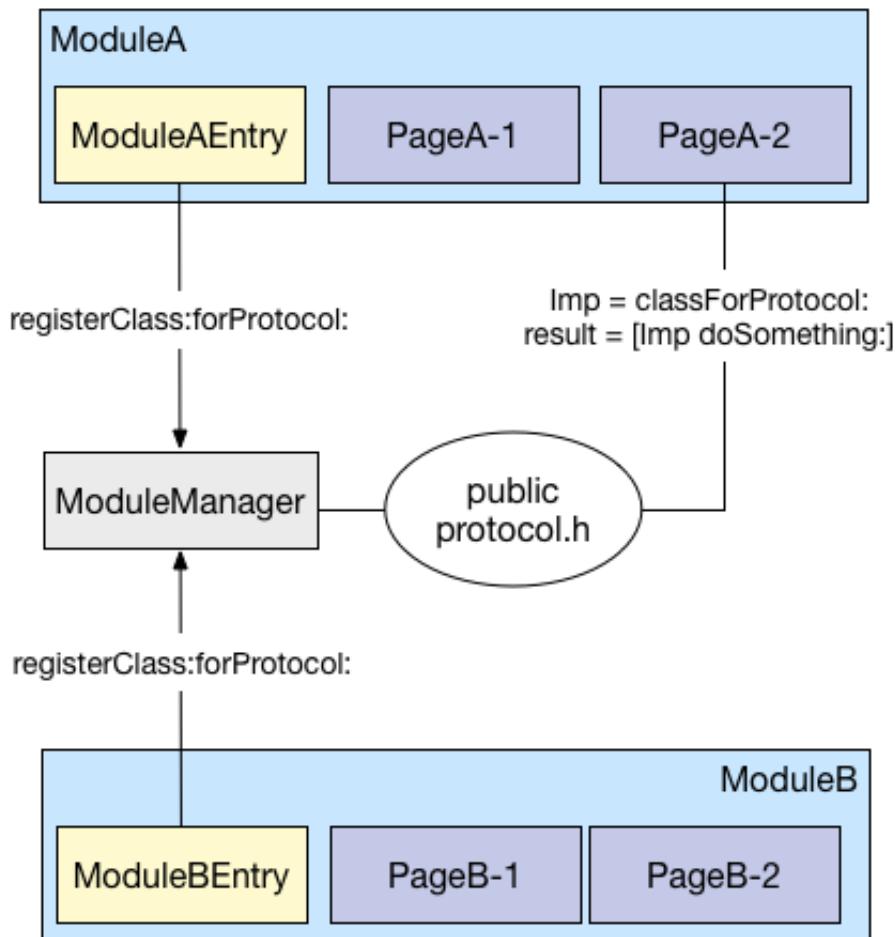
1. 注册这个组件关心的 URL
2. 注册这个组件能够被调用的方法/属性
3. 在 App 生命周期的不同阶段做不同的响应

页面间的 openURL 调用就是如下的样子：



每个组件间都会向 MGJRouter 注册，组件间相互调用或者是其他的 App 都可以通过 openURL: 方法打开一个界面或者调用一个组件。

在组件间的调用，蘑菇街采用了 Protocol 的方式。



[ModuleManager registerClass:ClassA forProtocol:ProtocolA] 的结果就是在 MM 内部维护的 dict 里新加了一个映射关系。

[ModuleManager classForProtocol:ProtocolA] 的返回结果就是之前在 MM 内部 dict 里 protocol 对应的 class，使用方不需要关心这个 class 是个什么东东，反正实现了 ProtocolA 协议，拿来用就行。

这里需要有一个公共的地方来容纳这些 public protocol，也就是图中的 PublicProtocol.h。

我猜测，大概实现可能是下面的样子：

```

@interface ModuleProtocolManager : NSObject

+ (void)registServiceProvide:(id)provide forProtocol:(Protocol*)protocol;
+ (id)serviceProvideForProtocol:(Protocol *)protocol;

@end

```

然后这个是一个单例，在里面注册各个协议：

```

@interface ModuleProtocolManager ()

@property (nonatomic, strong) NSMutableDictionary *serviceProvideSource;
@end

@implementation ModuleProtocolManager

+ (ModuleProtocolManager *)sharedInstance
{
    static ModuleProtocolManager * instance;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[self alloc] init];
    });
    return instance;
}

- (instancetype)init
{
    self = [super init];
    if (self) {
        _serviceProvideSource = [[NSMutableDictionary alloc] init];
    }
    return self;
}

+ (void)registServiceProvide:(id)provide forProtocol:(Protocol*)protocol
{
    if (provide == nil || protocol == nil)
        return;
    [[self sharedInstance].serviceProvideSource setObject:provide
forKey:NSStrringFromProtocol(protocol)];
}

+ (id)serviceProvideForProtocol:(Protocol *)protocol
{
    return [[self sharedInstance].serviceProvideSource
objectForKey:NSStrringFromProtocol(protocol)];
}

```

在 ModuleProtocolManager 中用一个字典保存每个注册的 protocol。现在再来猜猜 ModuleEntry 的实现。

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

@protocol DetailModuleEntryProtocol <NSObject>

@required;
- (UIViewController *)detailViewControllerWithId:(NSString*)Id Name:
(NSString *)name;
@end
```

然后每个模块内都有一个和暴露到外面的协议相连接的“接头”。

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

@interface DetailModuleEntry : NSObject
@end
```

在它的实现中，需要引入3个外部文件，一个是 ModuleProtocolManager，一个是 DetailModuleEntryProtocol，最后一个是在模块内跳转或者调用的组件或者页面。

```
#import "DetailModuleEntry.h"

#import <DetailModuleEntryProtocol/DetailModuleEntryProtocol.h>
#import <ModuleProtocolManager/ModuleProtocolManager.h>
#import "DetailViewController.h"

@interface DetailModuleEntry()<DetailModuleEntryProtocol>

@end

@implementation DetailModuleEntry

+ (void)load
{
    [ModuleProtocolManager registServiceProvide:[[self alloc] init]
forProtocol:@protocol(DetailModuleEntryProtocol)];
}

- (UIViewController *)detailViewControllerWithId:(NSString*)Id Name:
(NSString *)name
{
    DetailViewController *detailVC = [[DetailViewController alloc]
initWithId:id Name:name];
    return detailVC;
}

@end
```

至此基于 Protocol 的方案就完成了。如果需要调用某个组件或者跳转某个页面，只要先从 ModuleProtocolManager 的字典里面根据对应的 ModuleEntryProtocol 找到对应的 DetailModuleEntry，找到了 DetailModuleEntry 就是找到了组件或者页面的“入口”了。再把参数传进去即可。

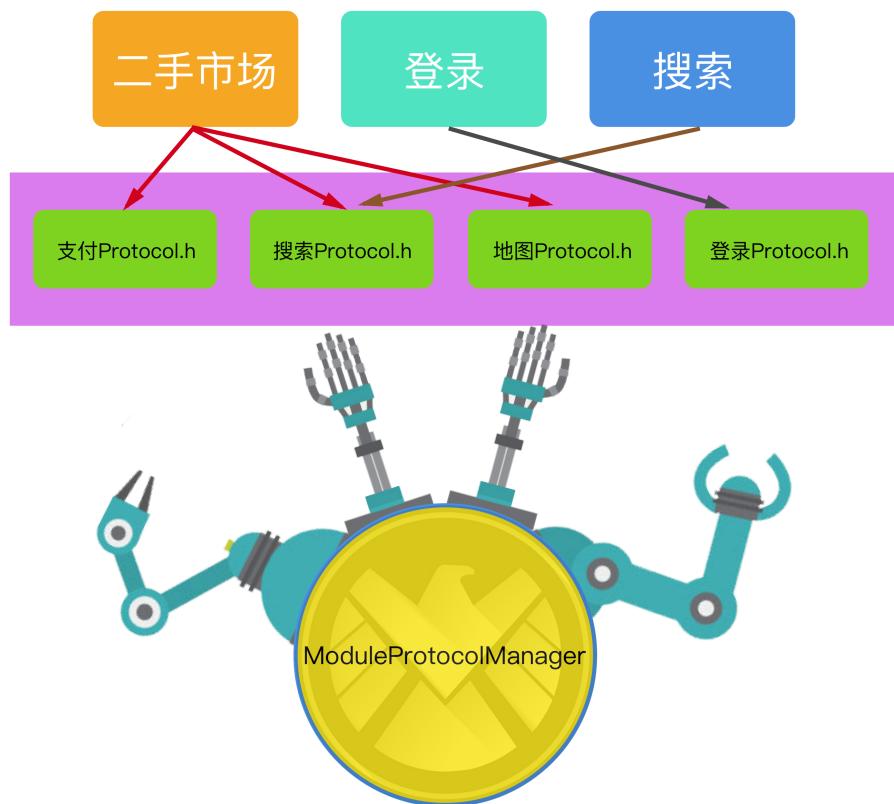
```

- (void)clickDetailButton:(UIButton *)button
{
    id< DetailModuleEntryProtocol > DetailModuleEntry =
    [ModuleProtocolManager
    serviceProvideForProtocol:@protocol(DetailModuleEntryProtocol)];
    UIViewController *detailVC = [DetailModuleEntry
    detailViewControllerWithId:@"详情界面" Name:@"我的购物车"];
    [self.navigationController pushViewController:detailVC animated:YES];
}

```

这样就可以调用到组件或者界面了。

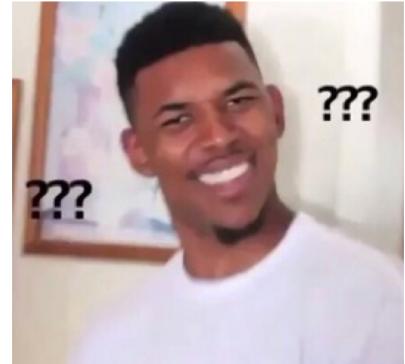
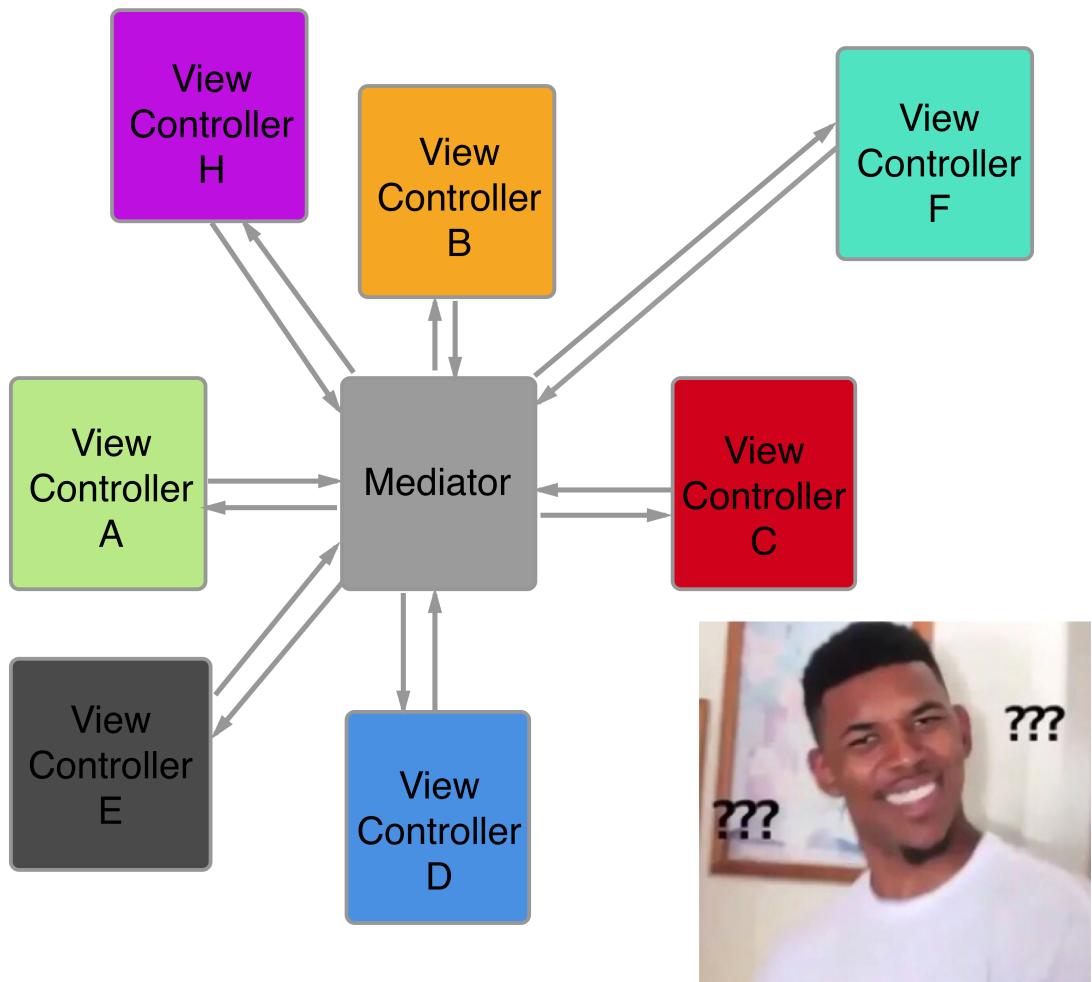
如果组件之间有相同的接口，那么还可以进一步的把这些接口都抽离出来。这些抽离出来的接口变成“元接口”，它们是可以足够支撑起整个组件一层的。



(5) [CTMediator](#) Star 803

再来说说 @casatwy 的方案，这方案是基于 Mediator 的。

传统的中间人 Mediator 的模式是这样的：



这种模式每个页面或者组件都会依赖中间者，各个组件之间互相不再依赖，组件间调用只依赖中间者 Mediator，Mediator 还是会依赖其他组件。那么这是最终方案了么？

看看 @casatwy 是怎么继续优化的。

主要思想是利用了 Target-Action 简单粗暴的思想，利用 Runtime 解决解耦的问题。

```

- (id)performTarget:(NSString *)targetName action:(NSString *)actionName
params:(NSDictionary *)params shouldCacheTarget:(BOOL)shouldCacheTarget
{

    NSString *targetClassName = [NSString stringWithFormat:@"Target_%@",
targetName];
    NSString *actionString = [NSString stringWithFormat:@"Action_%@:", actionName];
    Class targetClass;

    NSObject *target = self.cachedTarget[targetClassName];
    if (target == nil) {
        targetClass = NSClassFromString(targetClassName);
        target = [[targetClass alloc] init];
    }
}

```

```

SEL action = NSSelectorFromString(actionString);

if (target == nil) {
    // 这里是处理无响应请求的地方之一，这个demo做得比较简单，如果没有可以响应的
    target，就直接return了。实际开发过程中是可以事先给一个固定的target专门用于在这个时候顶
    上，然后处理这种请求的
    return nil;
}

if (shouldCacheTarget) {
    self.cachedTarget[targetClassName] = target;
}

if ([target respondsToSelector:action]) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
    return [target performSelector:action withObject:params];
#pragma clang diagnostic pop
} else {
    // 可能target是Swift对象
    actionString = [NSString stringWithFormat:@"Action_%@WithParams:",
actionName];
    action = NSSelectorFromString(actionString);
    if ([target respondsToSelector:action]) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
        return [target performSelector:action withObject:params];
#pragma clang diagnostic pop
    } else {
        // 这里是处理无响应请求的地方，如果无响应，则尝试调用对应target的notFound
方法统一处理
        SEL action = NSSelectorFromString(@"notFound:");
        if ([target respondsToSelector:action]) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
            return [target performSelector:action withObject:params];
#pragma clang diagnostic pop
        } else {
            // 这里也是处理无响应请求的地方，在notFound都没有的时候，这个demo是直
接return了。实际开发过程中，可以用前面提到的固定的target顶上的。
            [self.cachedTarget removeObjectForKey:targetClassName];
            return nil;
        }
    }
}
}

```

targetName 就是调用接口的 Object, actionName 就是调用方法的 SEL, params 是参数, shouldCacheTarget 代表是否需要缓存, 如果需要缓存就把 target 存起来, Key 是 targetClassString, Value 是 target。

通过这种方式进行改造的, 外面调用的方法都很统一, 都是调用 performTarget: action: params: shouldCacheTarget:。第三个参数是一个字典, 这个字典里面可以传很多参数, 只要 Key-Value 写好就可以了。处理错误的方式也统一在一个地方了, target 没有, 或者是 target 无法响应相应的方法, 都可以在 Mediator 这里进行统一出错处理。

但是在实际开发过程中, 不管是界面调用, 组件间调用, 在 Mediator 中需要定义很多方法。于是作者又想出了建议我们用 Category 的方法, 对 Mediator 的所有方法进行拆分, 这样就就可以不会导致 Mediator 这个类过于庞大了。

```

- (UIViewController *)CTMediator_viewControllerForDetail
{
    UIViewController *viewController = [self
performTarget:kCTMediatorTargetA

action:kCTMediatorActionNativeFetchDetailViewController

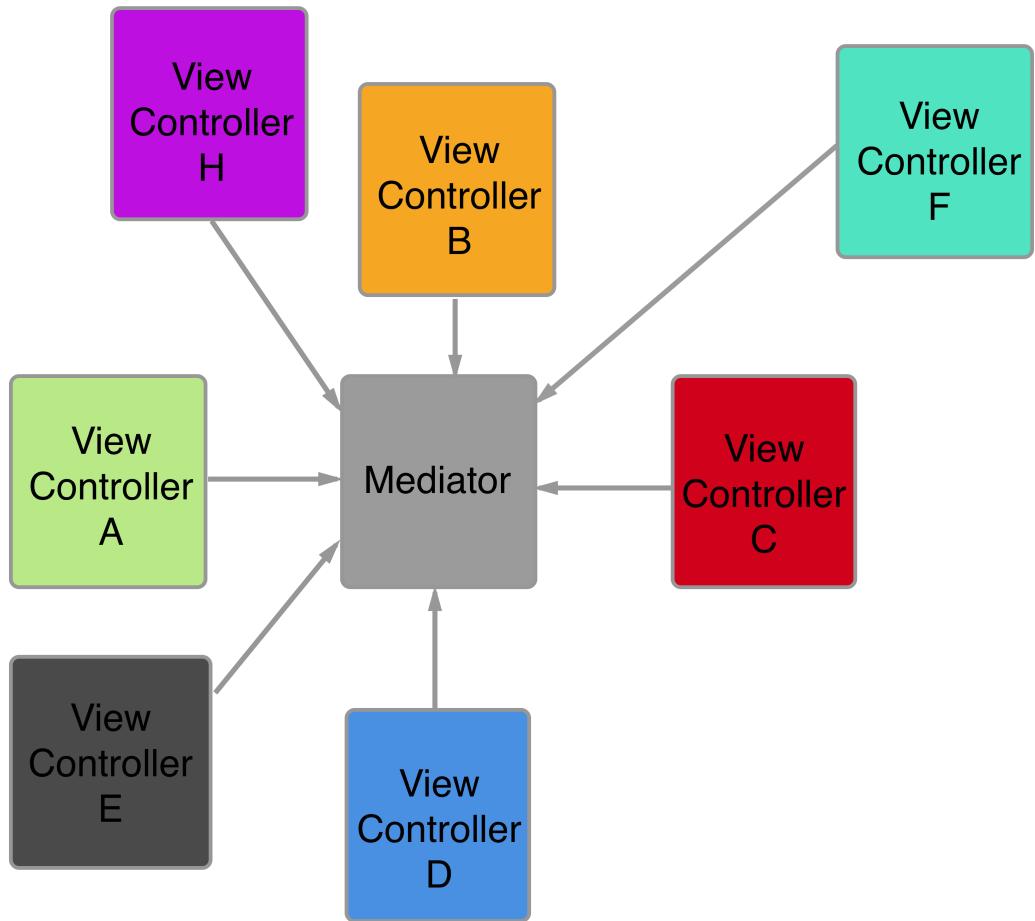
params:@{@"key": @"value"}
                                shouldCacheTarget:NO
];
    if ([viewController isKindOfClass:[UIViewController class]]) {
        // view controller 交付出去之后，可以由外界选择是push还是present
        return viewController;
    } else {
        // 这里处理异常场景，具体如何处理取决于产品
        return [[UIViewController alloc] init];
    }
}

- (void)CTMediator_presentImage:(UIImage *)image
{
    if (image) {
        [self performTarget:kCTMediatorTargetA
                        action:kCTMediatorActionNativePresentImage
                        params:@{@"image":image}
                                shouldCacheTarget:NO];
    } else {
        // 这里处理image为nil的场景，如何处理取决于产品
        [self performTarget:kCTMediatorTargetA
                        action:kCTMediatorActionNativeNoImage
                        params:@{@"image": [UIImage imageNamed:@"noImage"]}
                                shouldCacheTarget:NO];
    }
}

```

把这些具体的方法一个个的都写在 Category 里面就好了，调用的方式都非常的一致，都是调用 performTarget: action: params: shouldCacheTarget: 方法。

最终去掉了中间者 Mediator 对组件的依赖，各个组件之间互相不再依赖，组件间调用只依赖中间者 Mediator，Mediator 不依赖其他任何组件。



(6) 一些并没有开源的方案

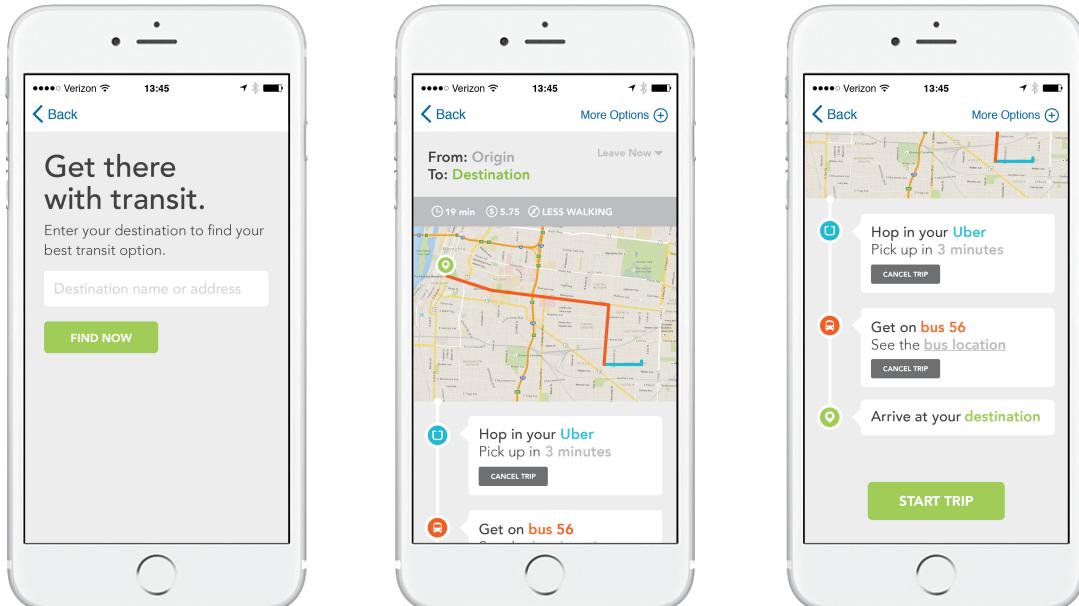
除了上面开源的路由方案，还有一些并没有开源的设计精美的方案。这里可以和大家一起分析交流一下。



这个方案是 Uber 骑手App 的一个方案。

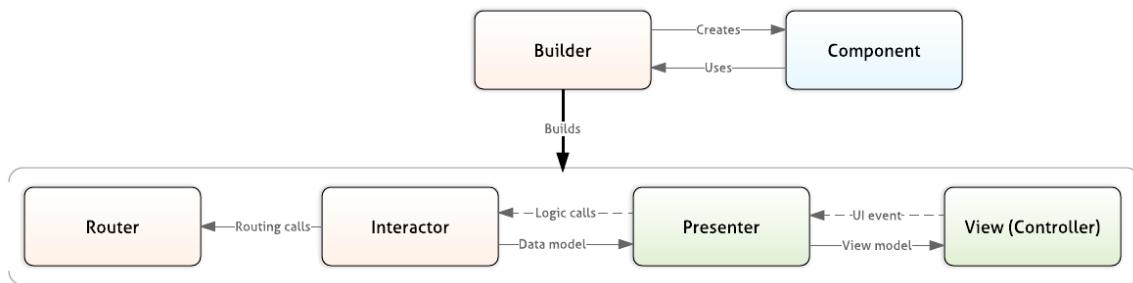
Uber 在发现 MVC 的一些弊端之后：比如动辄上万行巨胖无比的 VC，无法进行单元测试等缺点后，于是考虑把架构换成 VIPER。但是 VIPER 也有一定的弊端。因为它的 iOS 特定的结构，意味着 iOS 必须为 Android 做出一些妥协的权衡。以视图为驱动的应用程序逻辑，代表应用程序状态由视图驱动，整个应用程序都锁定在视图树上。由操作应用程序状态所关联的业务逻辑的改变，就必须经过 Presenter。因此会暴露业务逻辑。最终导致了视图树和业务树进行了紧紧的耦合。这样想实现一个紧紧只有业务逻辑的 Node 节点或者紧紧只有视图逻辑的Node节点就非常的困难了。

通过改进 VIPER 架构，吸收其优秀的特点，改进其缺点，就形成了 Uber 骑手App 的全新架构——Riblets(肋骨)。



在这个新的架构中，即使是相似的逻辑也会被区分成很小很小，相互独立，可以单独进行测试的组件。每个组件都有非常明确的用途。使用这些一小块一小块的 Riblets(肋骨)，最终把整个 App 拼接成一颗 Riblets(肋骨)树。

通过抽象，一个 Riblets(肋骨) 被定义成一下6个更小的组件，这些组件各自有各自的职责。通过一个 Riblets(肋骨) 进一步的抽象业务逻辑和视图逻辑。



一个 Riblets(肋骨) 被设计成这样，那和之前的 VIPER 和 MVC 有什么区别呢？最大的区别在路由上面。

Riblets(肋骨) 内的 Router 不再是视图逻辑驱动的，现在变成了业务逻辑驱动。这一重大改变就导致了整个 App 不再是由表现形式驱动，现在变成了由数据流驱动。

每一个 Riblet 都是由一个路由 Router，一个关联器 Interactor，一个构造器 Builder 和它们相关的组件构成的。所以它的命名 (Router - Interactor - Builder, Rib) 也由此得来。当然还可以有可选的展示器 Presenter 和视图 View。路由 Router 和关联器 Interactor 处理业务逻辑，展示器 Presenter 和视图 View 处理视图逻辑。

重点分析一下 Riblet 里面路由的职责。

1. 路由的职责

在整个 App 的结构树中，路由的职责是用来关联和取消关联其他子 Riblet 的。至于决定是由关联器 Interactor 传递过来的。在状态转换过程中，关联和取消关联子 Riblet 的时候，路由也会影响到关联器 Interactor 的生命周期。路由只包含2个业务逻辑：

1. 提供关联和取消关联其他路由的方法。
2. 在多个孩子之间决定最终状态的状态转换逻辑。

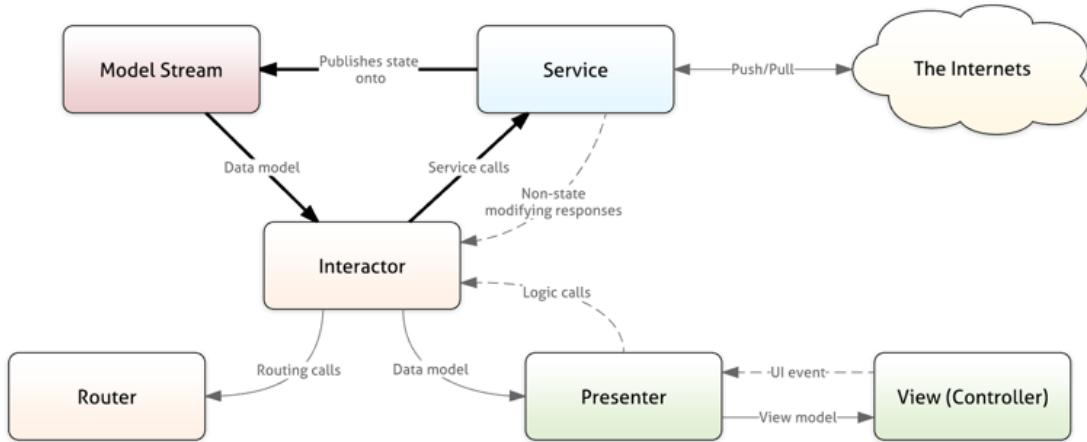
2. 拼装

每一个 Riblets 只有一对 Router 路由和 Interactor 关联器。但是它们可以有多对视图。Riblets 只处理业务逻辑，不处理视图相关的部分。Riblets 可以拥有单一的视图（一个 Presenter 展示器和一个 View 视图），也可以拥有多个视图（一个 Presenter 展示器和多个 View 视图，或者多个 Presenter 展示器和多个 View 视图），甚至也可能没有视图（没有 Presenter 展示器也没有 View 视图）。这种设计可以有助于业务逻辑树的构建，也可以和视图树做到很好的分离。

举个例子，骑手的 Riblet 是一个没有视图的 Riblet，它用来检查当前用户是否有一个激活的路线。如果骑手确定了路线，那么这个 Riblet 就会关联到路线的 Riblet 上面。路线的 Riblet 会在地图上显示出路线图。如果没有确定路线，骑手的 Riblet 就会被关联到请求的 Riblet 上。请求的 Riblet 会在屏幕上显示等待被呼叫。像骑手的 Riblet 这样没有任何视图逻辑的 Riblet，它分开了业务逻辑，在驱动 App 和支撑模块化架构起了重大作用。

3. Riblets 是如何工作的

Riblet 中的数据流



在这个新的架构中，数据流动是单向的。Data 数据流从 service 服务流到 Model Stream 生成 Model 流。Model 流再从 Model Stream 流动到 Interactor 关联器。Interactor 关联器，scheduler 调度器，远程推送都可以想 Service 触发变化来引起 Model Stream 的改动。Model Stream 生成不可改动的 models。这个强制的要求就导致关联器只能通过 Service 层改变 App 的状态。

举两个例子：

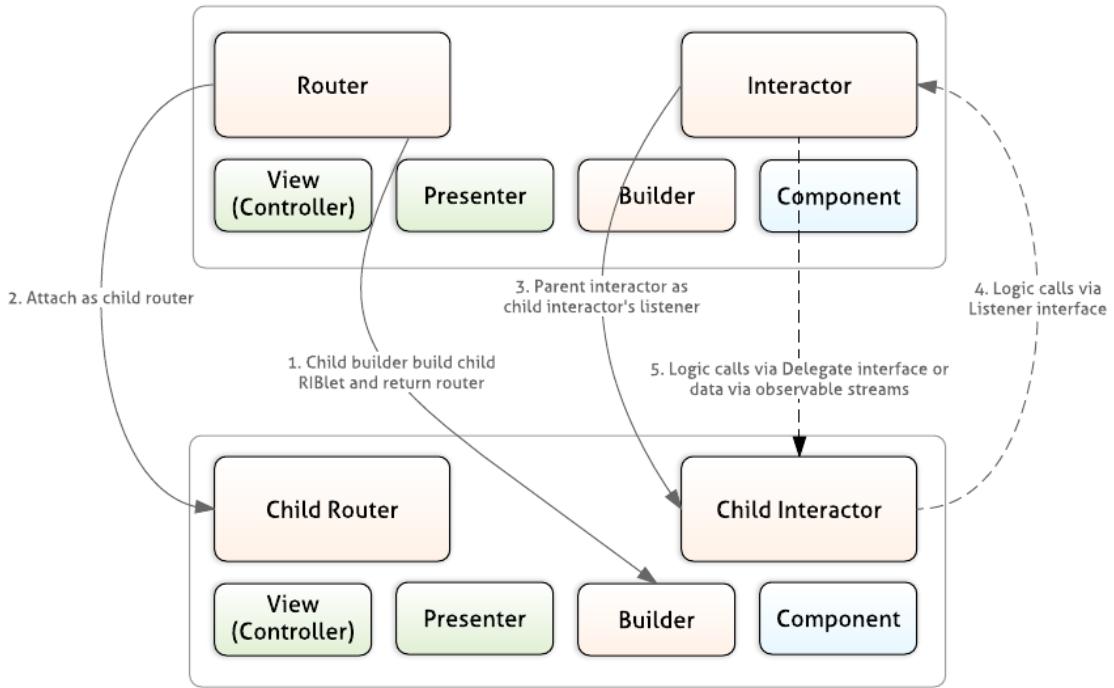
1. 数据从后台到视图 View 上

一个状态的改变，引起服务器后台触发推送到 App。数据就被 Push 到 App，然后生成不可变的数据流。关联器收到 model 之后，把它传递给展示器 Presenter。展示器 Presenter 把 model 转换成 view model 传递给视图 View。

2. 数据从视图到服务器后台

当用户点击了一个按钮，比如登录按钮。视图 View 就会触发 UI 事件传递给展示器 Presenter。展示器 Presenter 调用关联器 Interactor 登录方法。关联器 Interactor 又会调用 Service call 的实际登录方法。请求网络之后会把数据 pull 到后台服务器。

Riblet 间的数据流



当一个关联器 Interactor 在处理业务逻辑的工程中，需要调用其他 RIBLET 的事件的时候，关联器 Interactor 需要和子关联器 Interactor 进行关联。见上图5个步骤。

如果调用方法是从子调用父类，父类的 Interactor 的接口通常被定义成监听者 listener。如果调用方法是从父类调用到子类，那么子类的接口通常是一个 delegate，实现父类的一些 Protocol。

在 RIBLET 的方案中，路由 Router 仅仅只是用来维护一个树型关系，而关联器 Interactor 才担当的是用来决定触发组件间的逻辑跳转的角色。

五. 各个方案优缺点



经过上面的分析，可以发现，路由的设计思路是从 URLRoute ->Protocol-class ->Target-Action 一步一步的深入的过程。这也是逐渐深入本质的过程。

1. URLRoute 注册方案的优缺点

首先 URLRoute 也许是借鉴前端 Router 和系统 App 内跳转的方式想出来的方法。它通过 URL 来请求资源。不管是 H5, RN, Weex, iOS 界面或者组件请求资源的方式就都统一了。URL 里面也会带上参数，这样调用什么界面或者组件都可以。所以这种方式是最容易，也是最先可以想到的。

URLRoute 的优点很多，最大的优点就是服务器可以动态的控制页面跳转，可以统一处理页面出问题之后的错误处理，可以统一三端，iOS, Android, H5 / RN / Weex 的请求方式。

但是这种方式也需要看不同公司的需求。如果公司里面已经完成了服务器端动态下发的脚手架工具，前端也完成了 Native 端如果出现错误了，可以随时替换相同业务界面的需求，那么这个时候可能选择 URLRoute 的几率会更大。

但是如果公司里面 H5 没有做相关出现问题后能替换的界面，H5 开发人员觉得这是给他们增添负担。如果公司也没有完成服务器动态下发路由规则的那套系统，那么公司可能就不会采用 URLRoute 的方式。因为 URLRoute 带来的少量动态性，公司是可以用 JSPatch 来做到。线上出现 bug 了，可以立即用 JSPatch 修掉，而不采用 URLRoute 做。

所以选择 URLRoute 这种方案，也要看公司的发展情况和人员分配，技术选型方面。

URLRoute 方案也是存在一些缺点的，首先 URL 的 map 规则是需要注册的，它们会在 load 方法里面写。写在 load 方法里面是会影响 App 启动速度的。

其次是大量的硬编码。URL 链接里面关于组件和页面的名字都是硬编码，参数也都是硬编码。而且每个 URL 参数字段都必须要一个文档进行维护，这个对于业务开发人员也是一个负担。而且 URL 短连接散落在整个 App 四处，维护起来实在有点麻烦，虽然蘑菇街想到了用宏统一管理这些链接，但是还是解决不了硬编码的问题。

真正一个好的路由是在无形当中服务整个 App 的，是一个无感知的过程，从这一点来说，略有点缺失。

最后一个缺点是，对于传递 NSObject 的参数，URL 是不够友好的，它最多是传递一个字典。

2. Protocol-Class 注册方案的优缺点

Protocol-Class 方案的优点，这个方案没有硬编码。

Protocol-Class 方案也是存在一些缺点的，每个 Protocol 都要向 ModuleManager 进行注册。

这种方案 ModuleEntry 是同时需要依赖 ModuleManager 和组件里面的页面或者组件两者的。当然 ModuleEntry 也是会依赖 ModuleEntryProtocol 的，但是这个依赖是可以去掉的，比如用 Runtime 的方法 NSProtocolFromString，加上硬编码是可以去掉对 Protocol 的依赖的。但是考虑到硬编码的方式对出现 bug，后期维护都是不友好的，所以对 Protocol 的依赖还是不要去除。

最后一个缺点是组件方法的调用是分散在各处的，没有统一的入口，也就没法做组件不存在时或者出现错误时的统一处理。

3. Target-Action 方案的优缺点

Target-Action 方案的优点，充分的利用 Runtime 的特性，无需注册这一步。Target-Action 方案只有存在组件依赖 Mediator 这一层依赖关系。在 Mediator 中维护针对 Mediator 的 Category，每个 category 对应一个 Target，Category 中的方法对应 Action 场景。Target-Action 方案也统一了所有组件间调用入口。

Target-Action 方案也能有一定的安全保证，它对 url 中进行 Native 前缀进行验证。

Target-Action 方案的缺点，Target-Action 在 Category 中将常规参数打包成字典，在 Target 处再把字典拆包成常规参数，这就造成了一部分的硬编码。

4. 组件如何拆分？

这个问题其实应该是在打算实施组件化之前就应该考虑的问题。为何还要放在这里说呢？因为组件的拆分每个公司都有属于自己的拆分方案，按照业务线拆？按照最细小的业务功能模块拆？还是按照一个完成的功能进行拆分？这个就牵扯到了拆分粗细度的问题了。组件拆分的粗细度就会直接关系到未来路由需要解耦的程度。

假设，把登录的所有流程封装成一个组件，由于登录里面会涉及到多个页面，那么这些页面都会被打包在一个组件里面。那么其他模块需要调用登录状态的时候，这时候就需要用到登录组件暴露在外面可以获取登录状态的接口。那么这个时候就可以考虑把这些接口写到 Protocol 里面，暴露给外面使用。或者用 Target-Action 的方法。这种把一个功能全部都划分成登录组件的话，划分粒度就稍微粗一点。

如果仅仅把登录状态的细小功能划分成一个元组件，那么外面想获取登录状态就直接调用这个组件就好。这种划分的粒度就非常细了。这样就会导致组件个数巨多。

所以在进行拆分组件的时候，也许当时业务并不复杂的时候，拆分成组件，相互耦合也不大。但是随着业务不管变化，之前划分的组件间耦合性越来越大，于是就会考虑继续把之前的组件再进行拆分。也许有些业务砍掉了，之前一些小的组件也许还会被组合到一起。总之，在业务没有完全固定下来之前，组件的划分可能一直进行时。

六. 最好的方案

关于架构，我觉得抛开业务谈架构是没有意义的。因为架构是为了业务服务的，空谈架构只是一种理想的状态。所以没有最好的方案，只有最适合的方案。

最适合自己的公司业务的方案才是最好的方案。分而治之，针对不同业务选择不同的方案才是最优的解决方案。如果非要笼统的采用一种方案，不同业务之间需要同一种方案，需要妥协牺牲的东西太多就不好了。

希望本文能抛砖引玉，帮助大家选择出最适合自家业务的路由方案。当然肯定会有更加优秀的方案，希望大家能多多指点我。

参考资料

[在现有工程中实施基于CTMediator的组件化方案](#)

[iOS应用架构谈 组件化方案](#)

[蘑菇街 App 的组件化之路](#)

[蘑菇街 App 的组件化之路·续](#)

[ENGINEERING THE ARCHITECTURE BEHIND UBER'S NEW RIDER APP](#)