特别专题 语言 & 开发

分布式应用无银弹——分布式应用架构核心要素的设计方法探讨

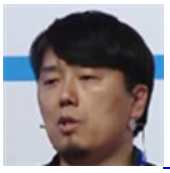
任何有点规模的应用，都无法回避分布式架构，任何分布式架构都没有一招制敌的法宝，路由、负载均衡、调度、并行计算、资源竞争、高可用、高可靠、可伸缩、高性能，这些要素一个也不能少，而每个要素又包含了多种设计选择，每种设计方法都会有妥协、有争取，最终这些选择的组合才能构成一个完整的、有针对性的分布式应用。

浏览所有 语言 & 开发

- 架构 & 设计
  - 架构
  - 企业架构
  - 性能和可伸缩性
  - 设计
  - 案例分析
  - 设计模式
  - 安全

特别专题 架构 & 设计

分布式应用无银弹——分布式应用架构核心要素的设计方法探讨


任何有点规模的应用，都无法回避分布式架构，任何分布式架构都没有一招制敌的法宝，路由、负载均衡、调度、并行计算、资源竞争、高可用、高可靠、可伸缩、高性能，这些要素一个也不能少，而每个要素又包含了多种设计选择，每种设计方法都会有妥协、有争取，最终这些选择的组合才能构成一个完整的、有针对性的分布式应用。

浏览所有 架构 & 设计

- 数据科学
  - 大数据
  - NoSQL
  - 数据库

特别专题 数据科学

InfoQ播客：Cathy O'Neil谈论有害的机器学习算法及审计方法

在《Weapons of Math Destruction》一书中，作者提出了存在一些有害的机器学习算法，这类算法正日益统治我们的社会。此博客是InfoQ就此书内容对作者的访谈，其中涉及了有缺陷的数据问题、数据科学及对算法的审计问题。

浏览所有 数据科学

- 文化 & 方法
  - Agile
  - 领导能力
  - 团队协作
  - 测试
  - 用户体验
  - Scrum
  - 精益

特别专题 文化 & 方法

架构师（2016年9月）


本期主要内容：怎样打造一个分布式数据库，10亿级流数据交互查询，为什么抛弃MySQL选择VoltDB，从单体架构迁移到微服务，8个关键的思考、实践和经验，50天10万行代码，一号专车系统重构细节回顾，京东前端：三级列表页持续架构优化，怎样才能叫高级程序员

浏览所有 文化 & 方法

- DevOps
  - 持续交付
  - 自动化操作
  - 云计算

特别专题 DevOps

iOS App组件化开发实践

作者 曹俊 发布于 2016年9月26日 / 被首富的“一个亿”刷屏？不如定个小目标，先把握住QCon上海的优惠吧！ 讨论

前因

其实我们这个7人iOS开发团队并不适合组件化开发。原因是因为性价比低，需要花很多时间和经历去做这件事，带来的收益并不能彻底改变什么。但是因为有2～3个星期的空档期，并不是很忙；另外是可以用在一个全新的App上。所以决定想尝试下组件化开发。

所谓尝试也就是说：去尝试解决组件化开发当中的一些问题。如果能解决，并且有比较好的解决方案，那就继续下去，否则就放弃。

背景

脱离实际情况去谈方案的选型是不合理的。

相关厂商内容

JVM虚拟化——重新定义 Java 容器热部署资源管理机制

Java 模块化技术演进和对现有应用微服务化的意义

面对JDK9的新变化你应该做哪些准备

通过探针技术，实现Java应用程序自我防护

新Java，新未来

相关赞助商

所以先简单介绍下背景：我们是一家纳斯达克交易所上市的科技企业。我们公司还有好几款App，由不同的几个团队去维护，我们是其中之一。我们这个团队是一个7人的iOS开发小团队。作者本人是小组长。

之前的App已经使用了模块化（CocoaPods）开发，并且已经使用了二进制化方案。App已经在使用自动化集成。

虽然要开发一个新App，但是很多业务和之前的App是一样的或者相似的。

为什么要写这篇博客?

想把整个过程记录下来，方便以后回顾。

我们的思路和解决方案不一定是对的或者是最好的。所以希望大家看了这篇博客之后，能给我们提供很多建议和别的解决方案，让我们可以优化使得这个组件化开发的方案能变得更加好。

技术栈

- gitlab
- gitlab-runner
- CocoaPods
- CocoaPods-Packager
- fir
- 二进制化
- fastlane
- deploymate
- oclint
- Kiwi

成果

使用组件化开发App之后：

- 代码提交更规范，质量提高。体现在测试人员反馈的bug明显减少。
- 编译加快。在都是源码的情况下：原App需要150s左右整个编译完毕，然后开发人员才可以开始调试。而现在组件化之后，某个业务组件只需要10s ~ 20s左右。在依赖二进制化组件的情况下，业务组件编译速度一般低于10s。
- 分工更为明确，从而提升开发效率。
- 灵活，耦合低。
- 结合MVVM。非常细致的单元测试，提高代码质量，保证App稳定性。体现在测试人员反馈的bug明显减少。
- 回滚更方便。我们经常会发生业务或者UI变回之前版本的情况，以前我们都是checkout出之前的代码。而现在组件化了之后，我们只需要使用旧版本的业务组件Pod库，或者在旧版本的基础上再发一个Pod库。
- 新人更容易上手。

对于我来说：

- 更加容易地把控代码质量。
- 更加容易地知道小组成员做了些什么。
- 更加容易地分配工作。
- 更加容易地安排新成员。

解耦

我们的想法是这样的，就算最后做不成组件化开发，把这些应该重用的代码抽出来做成Pod库也没有什么影响。所以优先做了这一步。

哪些东西需要抽成Pod库?

我们之前的App已经使用了模块化（CocoaPods化）开发。我们已经把会在App之间重用的Util、Category、网络层和本地存储等等这些东西抽成了Pod库。还有些一些和业务相关的，比如YTXChart,YTXChartSocket；这些也是在各个App之间重用的。

所以得出一个很简单的结论：要在App之间共享的代码就应该抽成Pod库，把它们作为一个个组件。

我们去仔细查看了原App代码，发现很多东西都需要重用而我们却没有把它们组件化。

为什么没有把这些代码组件化?

因为当时没想好怎么解耦，举个例子。

有一个类叫做YTXAnalytics。是依赖UMengAnalytics来做统计的。 它的耦合是在于一个方法。这个方法是用来收集信息的。它依赖了User，还依赖了currentServerId这个东西。

```
+ (NSDictionary*)collectEventInfo:(NSString*)event withData:(NSDictionary*)data
{
.......
    return @{
        @"event" : event,
        @"eventType" : @"event",
        @"time" : [[[NSDate date] timeIntervalSince1970InMillionSecond] stringValue],
        @"os" : device.systemName,
        @"osVersion" : device.systemVersion,
        @"device" : device.model,
        @"screen" : screenStr,
        @"network" : [YTXAnalytics networkType],
        @"appVersion" : [AppInfo appVersion],
        @"channel" : [AppInfo marketId],
        @"deviceId" : [ASIdentifierManager sharedManager].advertisingIdentifier.UUIDString,
        @"username" : objectOrNull([YTXUserManager sharedManager].currentUser.username),
        @"userType" : objectOrNull([[YTXUserManager sharedManager].currentUser.userType stringValue]),
        @"company" : [[ServiceProvider sharedServiceProvider].currentServerId stringValue],
        @"ip" : objectOrNull([SSNetworkInfo currentIPAddress]),
        @"data" : jsonStr
    };
}
```

解决方案是，搞了一个block，把获取这些信息的责任丢出来。

```
     [YTXAnalytics sharedAnalytics].analyticsDataBlock = ^ NSDictionary *() {
        return @{
                 @"appVersion" : objectOrNull([PBBasicProviderModule appVersion]),
                 @"channel" : objectOrNull([PBBasicProviderModule marketId]),
                 @"username" : objectOrNull([PBUserManager shared].currentUser.username),
                 @"userType" : objectOrNull([PBUserManager shared].currentUser.userType),
                 @"company" : objectOrNull([PBUserManager shared].currentUser.serverId),
                 @"ip" : objectOrNull([SSNetworkInfo currentIPAddress])
                 };
    };
```

我们的耦合大多数都是这种。解决方案都是弄了一个block，把获取信息的职责丢出来到外面。

我们解耦的方式就是以下几种：

1. 把它依赖的代码先做成一个Pod库，然后转而依赖Pod库。有点像是“依赖下沉”。

2. 使用category的方式把依赖改成组合的方式。
3. 使用一个block或delegate（协议）把这部分职责丢出去。
4. 直接copy代码。copy代码这个事情看起来很不优雅，但是它的好处就是快。对于一些不重要的工具方法，也可以直接copy到内部来用。

初始化

AppDelegate充斥着各种初始化。 比如我们自己的代码。已经只是截取了部分！

```
[self setupScreenShowManager];

//event start
[YTXAnalytics createYtxanalyticsTable];
[YTXAnalytics start];
[YTXAnalytics page:APP_OPEN];
[YTXAnalytics sharedAnalytics].analyticsDataBlock = ^ NSDictionary *() {
    return @{
             @"appVersion" : objectOrNull([AppInfo appVersion]),
             .......
             @"ip" : objectOrNull([SSNetworkInfo currentIPAddress]),
             };
};

[self registerPreloadConfig];
//Migrate UserDefault 转移standardUserDefault到group
[NSUserDefaults migrateOldUserDefaultToGroup];
[ServiceProvider sharedServiceProvider];

[YTXChatManager sharedYTXChatManager];
[ChartSocketManager sharedChartSocketController].delegate = [ChartProvider sharedChartProvider];

//初始化最初的行情集合
[[ChartProvider sharedChartProvider] addMetalList:[ChartSocketManager sharedChartSocketController].quoteList];

//初始化环信信息Manager
[YTXEaseMobManager sharedManager];
```

比如第三方：

```
//注册环信
[self setupEaseMob:application didFinishLaunchingWithOptions:launchOptions];

//Talking Data
[self setupTalkingData];
[self setupAdTalkingData];
[self setupShareSDK];
[self setupUmeng];
[self setupJSPatch];
[self setupAdhocSDK];
[YTXGdtAnalytics communicateWithGdt];//广点通
```

首先这些初始化的东西是会被各个业务组件都用到的。

那我组件化开发的时候，每一个业务组件如何保证我使用这些东西的时候已经初始化过了呢？难道每一个业务组件都初始化一遍？有参数怎么办，能不能使用单例？

但问题是第三方库基本都需要注册一个AppKey，我每一个业务组件里都写一份？那样肯定不好，那我配置在主App里的info.plist里面，每一个业务组件都初始化一下好了，也不会有什么副作用。但这样感觉不优雅，而且有很多重复代码。万一某个AppKey或重要参数改了，那每一个业务组件岂不是都得改了。这样肯定不行。另外一点，那我的业务组件必须依赖主App的内容了。无论是在主App里调试还是把主App的info.plist的相关内容拷贝过来使用。

更关键的是有一些第三方的库需要在application: didFinishLaunchingWithOptions:时初始化。

```
//初始化环信，shareSDK，友盟，Talking Data等
[self setupThirdParty:application didFinishLaunchingWithOptions:launchOptions];
```

有没有更好的办法呢？

首先我写了一个YTXModule。它利用runtime，不需要在AppDelegate中添加任何代码，就可以捕获App生命周期。

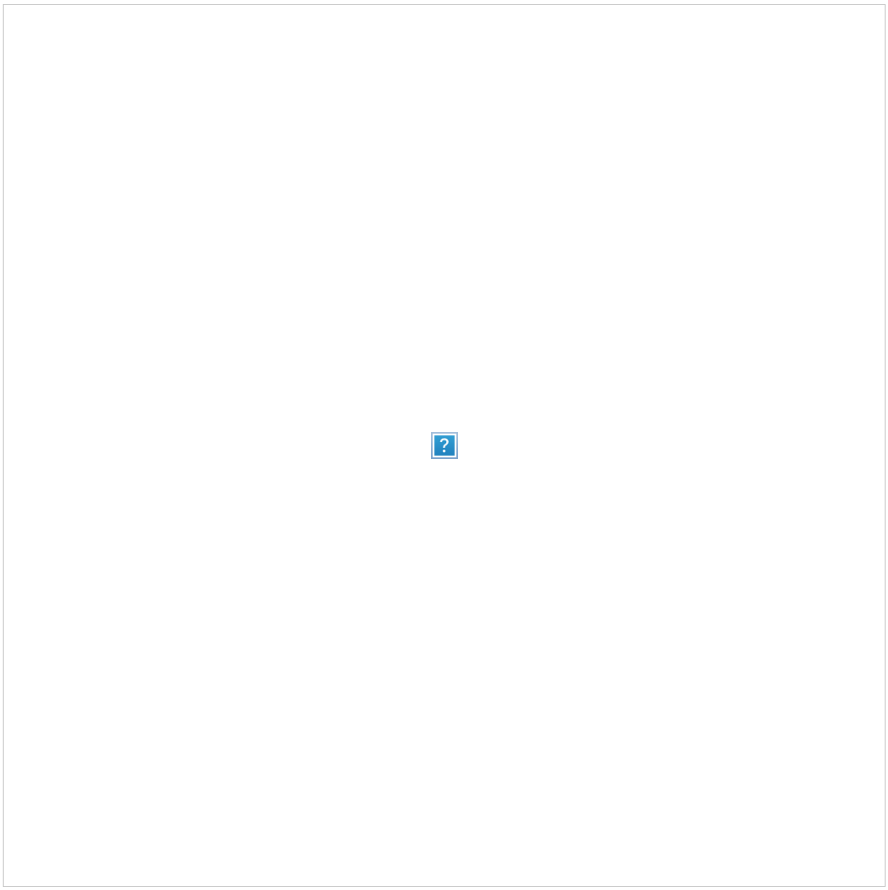
在某个想获得App生命周期的类中的.m中这样使用：

```
YTXMODULE_EXTERN()
{
    //相当于load
    isLoad = YES;
}
+ (BOOL)application:(UIApplication *)application willFinishLaunchingWithOptions:(nullable NSDictionary *)launchOptions
{
    //实现一样的方法名，但是必须是静态方法。
    return YES;
}
```

分层

因为在解决初始化问题的时候，要先设计好层级结构。所以这里突然跳转到分层。

上个图：



我们自己定了几个原则。

- 业务组件之间不能有依赖关系。
- 按照图示不能跨层依赖。
- 所谓弱业务组件就是包含着少部分业务，并且可以在这个App内的各个业务组件之间重用的代码。
- 要依赖YTXModule的组件一定要以Module结尾，而且它一定是个业务组件或是弱业务组件。
- 弱业务组件以App代号开头（比如PB），以Module结尾。例：PBBasicProviderModule。
- 业务组件以App代号开头（比如PB）BusinessModule结尾。例：PBHomePageBusinessModule。

业务组件之间不能有依赖关系，这是公认的的原则。否则就失去了组件化开发的核心价值。

弱业务组件之间也不应当有依赖关系。如果有依赖关系说明你的功能划分不准确。

初始化

我们约定好了层级结构，明确了职责之后。我们就可以跳回初始化的设计了。

创建一个PBBasicProviderModule弱业务组件。

- 它通过依赖YTXModule来捕捉App生命周期。
- 它来负责初始化自己的和第三方的东西。
- 所有业务组件都可以依赖这个弱业务组件。
- 它来保证所有东西一定是是初始化完毕的。
- 它来统一管理。
- 它来暴露一些类和功能给业务组件使用。

反正就是业务组件中依赖PBBasicProviderModule，它保证它里面的所有东西都是好用的。

因为有了PBBasicProviderModule，所以才让我更明确了弱业务组件这个概念。

因为我们懒，如果把PBBasicProvider定义为业务组件。那它和其他业务组件之间的通信就必须通过Bus、Notification或协议等等。

但它又肯定是业务啊。因为那些AppKey肯定是和这个App有关系的，也就是App的相关配置和参数也可以说是业务；我需要初始化设置那些Block依赖User信息、CurrentServerId等等肯定都是业务啊。

那只好搞个弱业务出来啊。因为我不能打破这个原则啊：业务组件之间不能互相依赖。

再进一步分清弱业务组件和业务组件。

业务组件里面基本都有：storyboard、nib、图片等等。弱业务组件里面一般没有。这不是绝对的，但一般情况是这样。

业务组件一般都是App上某一具体业务。比如首页、我、直播、行情详情、XX交易大盘、YY交易大盘、XX交易中盘、资讯、发现等等。而弱业务组件是给这些业务组件提供功能的，自己不直接表现在App上展示。

我们还可以创建一些弱业务组件给业务组件提供功能。当然了，不能够滥用。需要准确划分职责。

最后，代码大概是这样的：

```
@implementation PBBasicProviderModule

YTXMODULE_EXTERN()
{

}

+ (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(nullable NSDictionary *)launchOptions
{
    [self setupThirdParty:application didFinishLaunchingWithOptions:launchOptions];
    [self setupBasic:application didFinishLaunchingWithOptions:launchOptions];

    return YES;
}

+ (void) setupThirdParty:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
        [self setupEaseMob:application didFinishLaunchingWithOptions:launchOptions];
        [self setupTalkingData];
        [self setupAdTalkingData];
        [self setupShareSDK];
        [self setupJSPatch];
        [self setupUmeng];
//        [self setupAdhoc];
    });
}

+ (void) setupBasic:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self registerBasic];

    [self autoIncrementOpenAppCount];

    [self setupScreenShowManager];
```

```
    [self setupYTXAnalytics];

    [self setupRemoteHook];
}

+ (YTXAnalytics) sharedYTXAnalytics
{
    return ......;
}
......
```

设想

这个PBBasicProviderModule简直就是个大杂烩啊，把很多以前写在AppDelegate里的东西都丢在里面了。毫无优雅可言。

的确是这样的，感觉没有更好的办法了。

既然已经这样了。我们可不可以大胆地设想一下：每个开发者开发自己负责的业务组件的时候不需要关心主App。

因为我知道美团的组件化开发必须依赖主App的AppDelegate的一大堆设置和初始化。所以干脆他们就直接在主App中集成调试，他们通过二进制化和去Pod依赖化的方式让主App的构建非常快。

所以我们是不是可以继续污染这个PBBasicProviderModule。不需要在主App项目里的AppDelegate写任何初始化代码？基本或者尽量不在主App里写任何代码？改依赖主App变为依赖这个弱业务组件？

按照这个思路我们搬空了AppDelegate里的所有代码。比如一些初始化App样式的东西、初始化RootViewController等等这些都可以搬到一个新的弱业务组件里。

而业务组件其实根本不需关心这个弱业务组件，开发人员只需要在业务组件中的Example App中的AppDelegate中初始化自己业务组件的RootViewController就好了。

其他的事情交给这个新的弱业务组件就好了。而主App和Example App只要在Podfile中依赖它就好了。

所以最后的设想就是：开发者不会去改主App项目，也不需要知道主App项目。对于开发者来说，主App和业务组件之间是隔绝的。

有一个更大的好处，我只要更换这个弱业务组件，这个业务组件就能马上适配一个新App。这也是某种意义上的解耦。

Debug/Release

谁说不用在主App里的AppDelegate写任何代码的，打脸。。。

我们在对二进制Pod库跑测试的发现，源码能过，二进制(.a)不能过。百思不得其解，然后仔细查看代码，发现是这个宏的锅：

```
#ifdef DEBUG

#endif
```

DEBUG在编译阶段就已经决定了。二进制化的时候已经编译完成了。 而我们的代码中充满着#ifdef DEBUG 就这样这样。那怎么办，这是二进制化的锅。但是我们的二进制化已经形成了标准，大家都自觉会这么做，怎么解决这个问题呢。

解决方案是：

创建了一个PBEnvironmentProvider。大家都去依赖它。

然后原来判断宏的代码改成这样：

```
if([PBEnvironmentProvider testing])
{
//...
}
```

在主App的AppDelegate中这样：

```
#if DEBUG && TESTING
//PBEnvironmentProvider提供的宏
```

```
CONFIG_ENVIRONMENT_TESTING
#endif
```

原理是：如果AppDelegate有某个方法（CONFIG_ENVIRONMENT_TESTING宏会提供这个方法），[PBEnvironmentProvider testing]得到的结果就是YES。

为什么要写在主App里呢？其实也可以丢在PBBasicProviderModule里面，提供一个方法啊。

因为主App的AppDelegate.m是源码，未经编译。另外注意TESTING这个宏。我们可以在xcode设置里加一个macro参数TESTING，并且修改为0的情况下，能够生成一个实际是DEBUG的App但里面内容却是线上的内容。

这个需求是来自于我们经常需要紧急通过xcode直接build一个app到手机上以解决或确认线上的问题。

虽然打脸了，但是也还好，以后也不用改了。再说这个是特殊需求。除了这个之外，主App没有其他代码了。

业务组件间通信

我们解决了初始化和解耦的问题。接下来只要解决组件间通信的问题就好了。

然后我找了几个第三方库，选用了MGJRouter。本来直接依赖它就好了。

后来觉得都使用Block的方式会导致这样的代码，全部堆在了一个方法里：

```
+ (void) setupRouter
{
......
[MGJRouter registerURLPattern:@"mgj://foo/a" toHandler:^(NSDictionary *routerParameters) {
    NSLog(@"routerParameterUserInfo:%@", routerParameters[MGJRouterParameterUserInfo]);
}];
[MGJRouter registerURLPattern:@"mgj://foo/b" toHandler:^(NSDictionary *routerParameters) {
    NSLog(@"routerParameterUserInfo:%@", routerParameters[MGJRouterParameterUserInfo]);
}];
......
}
```

这样感觉很不爽。那我干脆就把MGJRouter代码复制了下来，把Block改成了@selector。并且把它直接加入了YTXModule里面。并且使用了宏，让结果看起来优雅些。代码看起来是这样的：

```
//在某个类的.m里，其实并不需要继承YTXModule也可以使用该功能
YTXMODULE_EXTERN_ROUTER_OBJECT_METHOD(@"object1")
{
    YTXMODULE_EXAPAND_PARAMETERS(parameters)
    NSLog(@"%@ %@", userInfo, completion);
    isCallRouterObjectMacro2 = YES;
    return @"我是个类型";
}

YTXMODULE_EXTERN_ROUTER_METHOD(@"YTX://QUERY/:query")
{
    YTXMODULE_EXAPAND_PARAMETERS(parameters)
    NSLog(@"%@ %@", userInfo, completion);
    testQueryStringQueryValue = parameters[@"query"];;
    testQueryStringNameValue = parameters[@"name"];
    testQueryStringAgeValue = parameters[@"age"];
}
```

调用的时候看起来是这样的：

```
 [YTXModule openURL:@"YTX://QUERY/query?age=18&name=CJ" withUserInfo:@{@"Test":@1} completion:nil];

 NSString * testObject2 = [YTXModule objectForURL:@"object1" withUserInfo:@{@"Test":@2}];
```

通信问题解决了。其实页面跳转问题也解决了。

页面跳转

页面跳转解决方案与业务组件之间通信问题是一样的。

但是需要注意的是，你一个业务组件内部的页面跳转也请使用URL+Router的方式跳转，而不要自己直接pushViewController。

这样的好处是：如果将来某些内部跳转页面需要给其他业务组件调用，你就不需要再注册个URL了。因为本来就有。

是否去Model化

去Model化主要体现在业务组件间通信，要不要传一个Model过去(传过去的Dictionary中的某个键是Model)。

如果去Model化，这个业务组件的开发者如何确定Dictionary里面有哪些内容分别是什么类型呢？那需要有个地方传播这些信息，比如写在头文件，wiki等等。

如果不去Model化的话，就需要把这个Model做成Pod库。两个业务组件都去依赖它。

最后决定不去Model。因为实际上有一些Model就是在各个业务组件之间公用的（比如User），所以肯定就会有Model做成Pod库。我们可以把它做成重Model，Model里可以带网络请求和本地存储的方法。唯一不能避免的问题是，两个业务组件的开发者都有可能去改这个Model的Pod库。

信息的披露

跳转的页面需要传哪些参数？ 业务组件之间传递数据时候本质的载体是什么？

不同业务开发者如何知晓这些信息。

使用去Model化和不使用去Model化，我们都有各自的方案。

去Model化，则披露头文件，在头文件里面写详细的注释。

如果不去Model化，则就看Model就可以了。如有特殊情况，那也是文档写在头文件内。

总结的话：信息披露的方式就是把注释文档写在头文件内。

组件的生命周期

业务组件的生命周期和App一样。它本身就是个类，只暴露类方法，不存在需要实例，所以其实不存在生命周期这个概念。而它可以使用类方法创建很多ViewController，ViewController的生命周期由App管理。哪怕这些ViewController之间需要通信，你也可以使用Bus/YTXModule/协议等等方式来做，而不应该让业务组件这个类来负责他们之间的通信；也不应该自己持有ViewController；这样增加了耦合。

弱业务组件的生命周期由创建它的对象来管理。按需创建和ARC自动释放。

基础功能组件和第三方的生命周期由创建它的对象来管理。按需创建和ARC自动释放。

版本规范

我们自己定的规则。

所有Pod库都只依赖到minor

```
"~> 2.3"
```

主App中精确依赖到patch

```
"2.3.1"
```

主App中的业务组件版本号的Main.Minor要和主App版本保持一致。

参考：Semantic Versioning RubyGems Versioning Policies

二进制化

二进制化我认为是必须的，能够加快开发速度。

而我使用的这个二进制方案

有个坑就是在gitlab-runner上在二进制和源码切换时，经常需要pod cache clean --all，test/lint/publish才能成功。而每次pod cache clean --all之后CocoaPods会去重新下载相关的pod库，增加了时间和不必要的开销。

我们现在通过podspec中增加preserve_paths和执行download_zip.sh解决了cache的问题。原理是让pod cache既有源码又有二进制.a。具体可以看ytx-pod-template项目中的Name.podspec和download_zip.sh。

二进制化还得注意宏的问题。小心使用宏，尤其是#ifdef。避免源码和二进制代码运行的结果不一样。

集成调试

集成调试很简单。每一个业务组件在自己的Example App中调试。

这个业务组件的podspec只要写清楚自己依赖的库有哪些。剩下的其他业务组件应该写在Example App的Podfile里面。

依赖的Pod库都是二进制的。如有问题可以装源码(IS_SOURCE=1 pod install)来调试。

开发人员其实只需要关心自己的业务组件，这个业务组件是自洽的。

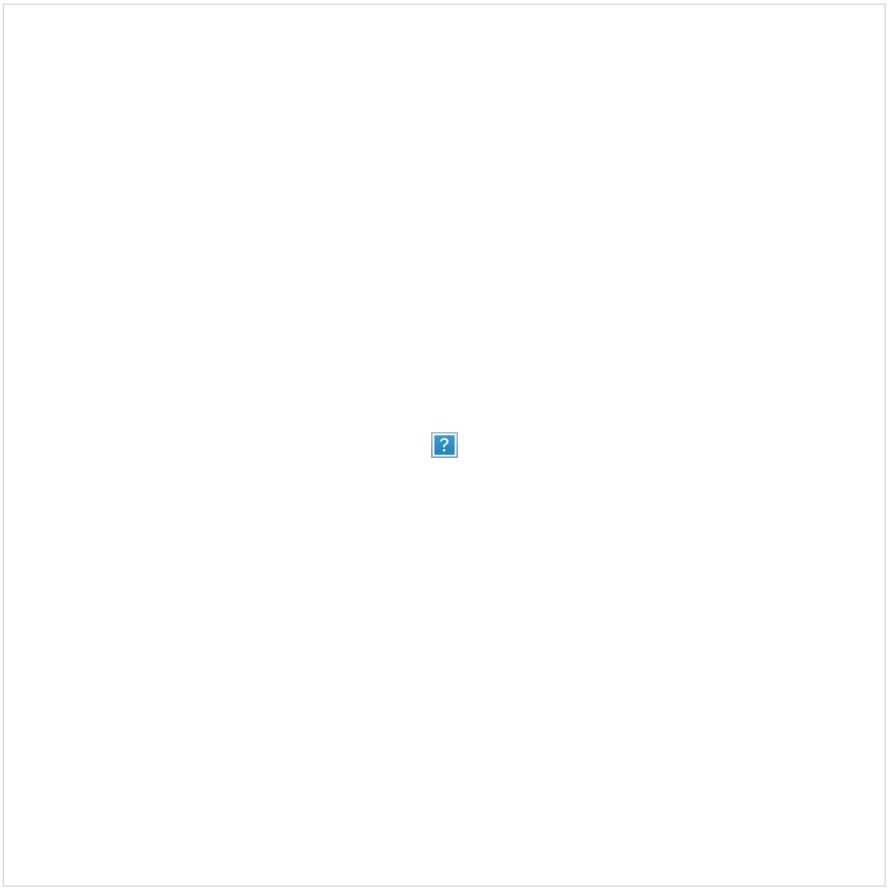
公共库谁来维护的问题

这个问题在我们这种小Team不存在。没有仔细地去想过。但是只要做好代码准入（Test/Lint/Code Review）和权限管理就应该不会存在大的问题。

单元测试

单元测试我们用的是Kiwi。 结合MVVM模式，对每一个业务组件的ViewModel都进行单元测试。每次push代码，gitlab-runner都会自动跑测试。一旦开发人员发现测试挂了就能够及时找到问题。也可以很容易的追溯哪次提交把测试跑挂了。

这也是我们团队的强制要求。没有测试，测试写的不好，测试挂了，直接拒绝merge request。



lint

对每一个组件进行lint再发布，保证了正确性。这也是一步强制要求。

lint的时候能够发现很多问题。通常情况下不允许warning出现的。如果不能避免（比如第三方）请用--allow-warnings。

```
pod lib lint --sources=$SOURCES --verbose --fail-fast --use-libraries
```

统一的网络服务和本地存储方式

这个就很简单。把这两个部分抽象成几个Pod库供所有业务组件使用就好了。 我们这边分别是三个Pod库：

- YTXRequest
- YTXRestfulModel
- NSUserDefault+YTX

其他一些内容

ignore了主App中的Podfile.lock尽量避免冲突。

主App Archive的时候要使用源码，而不是二进制。

后期可以使用oclint和deploymate检查代码。

使用fastlane match去维护开发证书。

一些需要从plist或者json读取配置的Pod库模块，要注意读出来的内容最好要加一个namespace。namespace可以是这个业务组件的名字。

业务组件读取资源文件的区别

```
#从main bundle中取。如果图片希望在storyboard中被找到，使用这种方式。
s.resource = ["#{s.name}/Assets/**"]

#只是希望在我这个业务组件的bundle内使用的plist。作为配置文件。这是官方推荐方式。
s.resource_bundles = {
  "{s.name}/" => ["{s.name}/Assets/config.plist"]
}
```

持续集成

原来的App就是持续集成的。想当然的，我们希望新的组件化开发的App也能够持续集成。

Podfile应该是这样的：这里面出现的全是私有Pod库。

```
pod 'YTXRequest', '2.0.1'
pod 'YTXUtilCategory', '1.6.0'

pod 'PBBasicProviderModule', '0.2.1'
pod 'PBBasicChartAndSocketModule', '0.3.1'
pod 'PBBasicAppInitModule', '0.5.1'
...

pod 'PBBasicHomepageBusinessModule', '1.2.15'
pod 'PBBasicMeBusinessModule', '1.2.10'
pod 'PBBasicLiveBusinessModule', '1.2.1'
pod 'PBBasicChartBusinessModule', '1.2.6'
pod 'PBBasicTradeBusinessModule', '1.2.7'
...
```

如果Pod依赖的东西特别特别多，比如100多个。另外又必须依赖主App做集成调试。 你也可以用这种方案：把你所有的Pod库的依赖都展开写到主App的Podfile中。而发布Pod库时podspec中不带任何的依赖的。这样就避免了pod install的时候解析依赖特别耗时的问题。

各个脚本都在这个ytx-pod-template。先从.gitlab-ci.yml看起。

我们持续集成的工具是gitlab runner。

持续集成的整个流程是：

第一步：

使用template创建Pod。像这样：

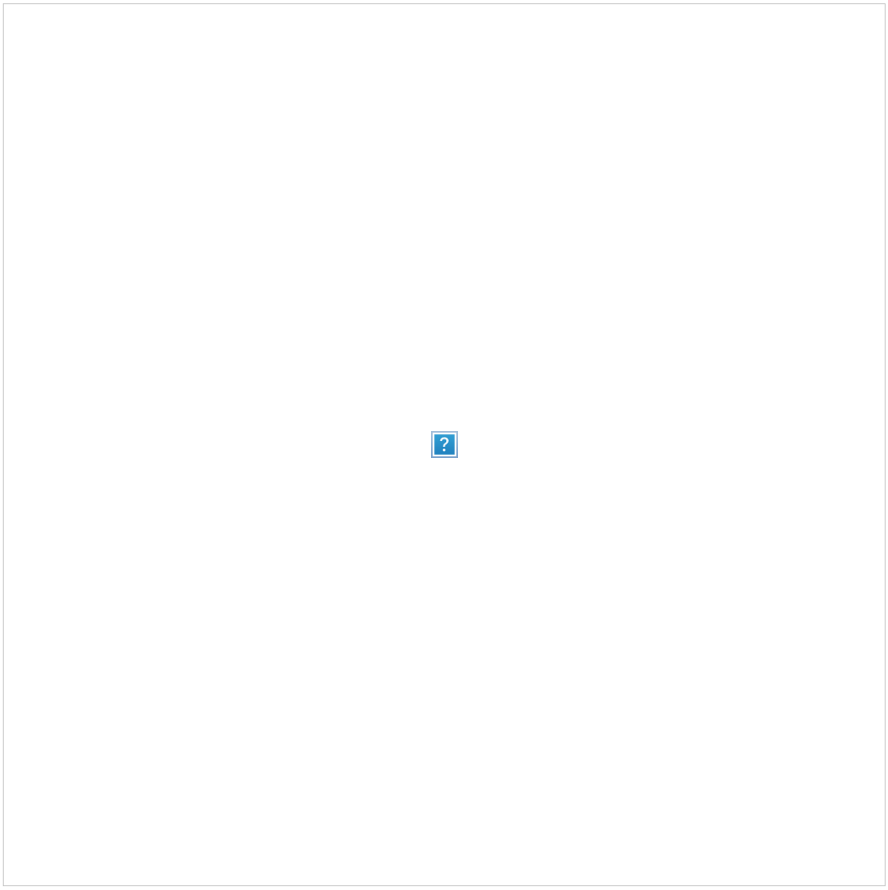
```
pod lib create <Pod库名称> --template-url="http://gitlab.baidao.com/pods/ytx-pod-template"
```

第二步：

创建dev分支。用来开发。

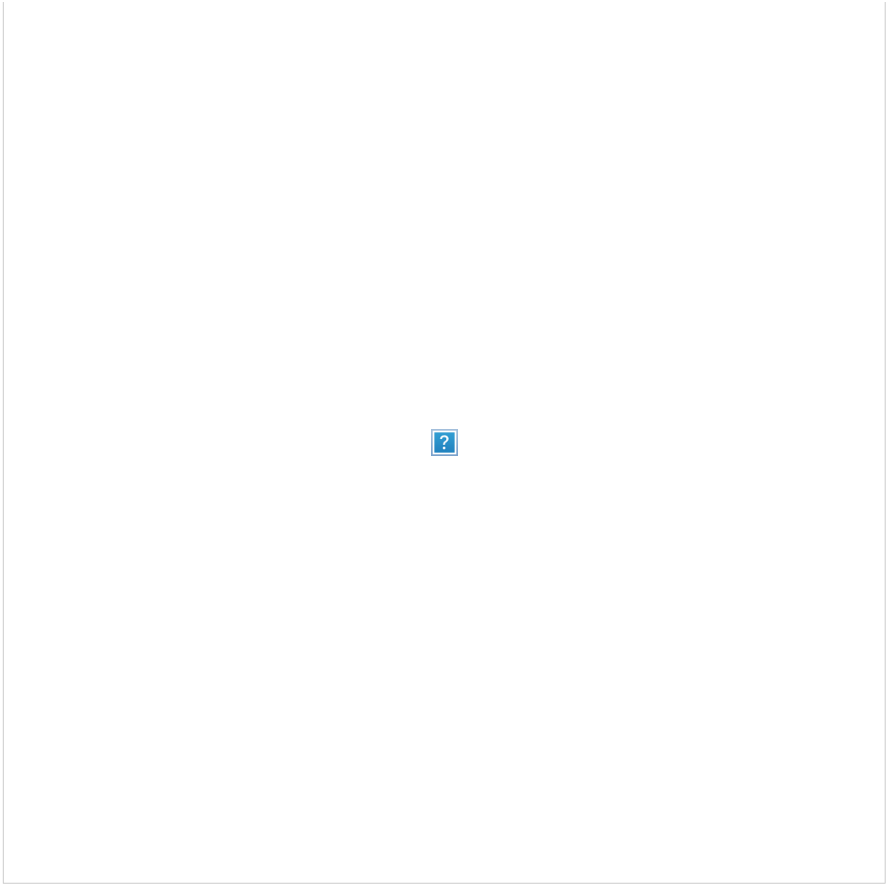
第三步：

每次push dev的时候会触发runner自动跑Stage: Init Lint(中的test)



第四步：

1.准备发布Pod库。修改podspec的版本号，打上相应tag。 2.使用merge_request.sh向master提交一个merge request。

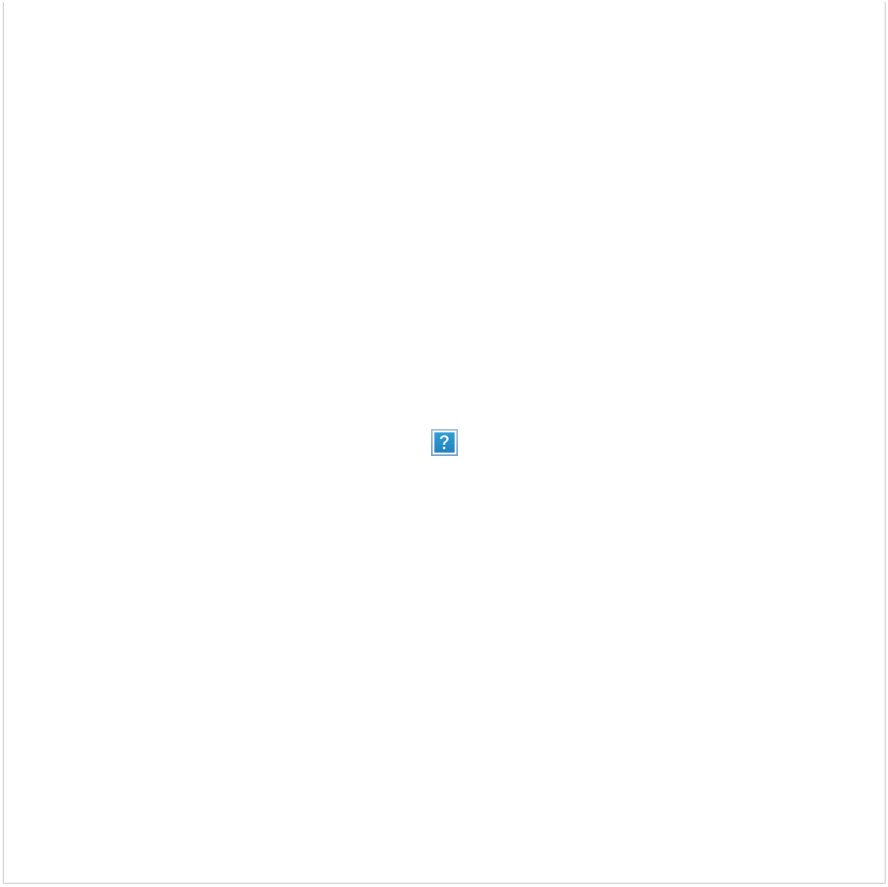


第五步：

1.其他有权限开发者code review之后，接受merge request。 2.master合并这个merge request 3.master触发runner自动跑Stage: Init Package Lint ReleasePod UpdateApp

第六步：

如果第五步正确。主App的dev分支会收到一个merge request，里面的内容是修改Podfile。 图中内容出现了AFNetworking等是因为这个时候在做测试。



第七步：

主App触发runner，会构建一个ipa自动上传到fir。

Init

- 初始化一些环境。
- 打印一些信息。

Package

- 二进制化打包成.a

Lint

- Pod lib lint。二进制和源码都lint。
- 测试。
- 以后考虑加入oclint和deploymate。

ReleasePod

- 把相关文件zip后，传到静态服务器库。以提供二进制化下载包。
- pod repo push。发布该Pod库。

ReleasePod的时候不允许Pod库出现警告。

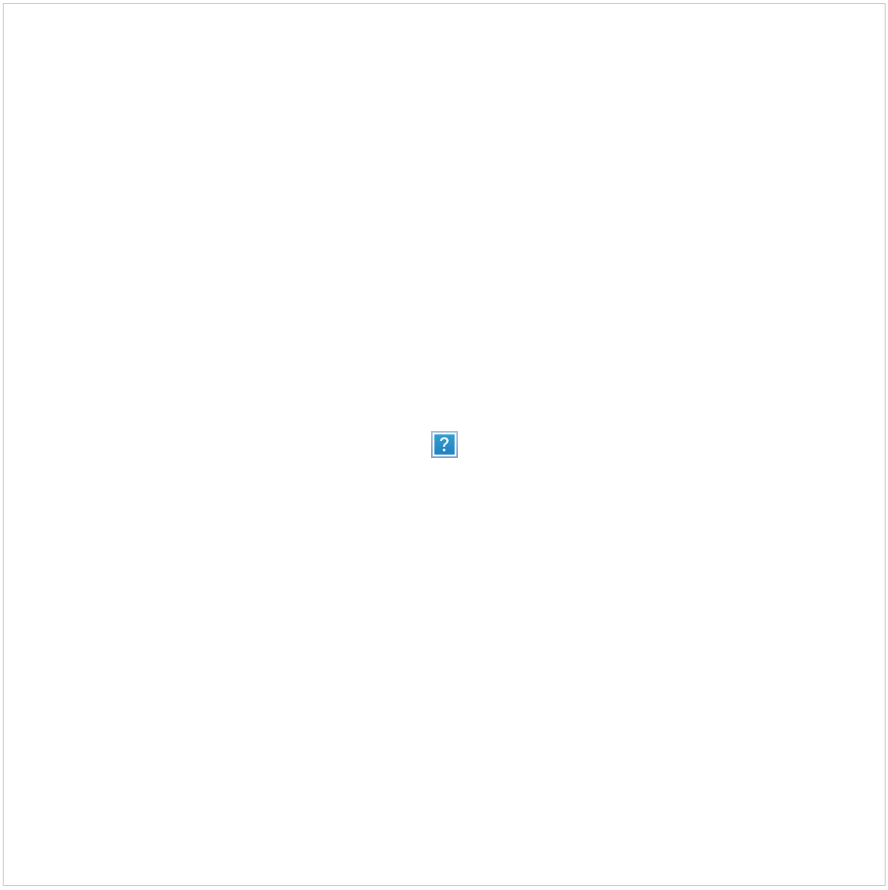
UpdateApp

- 下载App代码
- 修改Podfile文件。如果匹配到pod库文件名则修改，否则添加。
- 生成一个merge request到主App的dev分支。

关于**gitlab runner**。

stage这个功能非常的厉害。强烈推荐。

每一个stage可以跑在不同的runner上。每一个stage失败了可以单独retry。而某一个stage里面的任务可以并行执行：(test和lint就是并行的)



感谢徐川对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至editors@cn.infoq.com。也欢迎大家通过新浪微博（@InfoQ，@丁晓昀），微信（微信号：InfoQChina）关注我们。

【ArchSummit北京2016】七月深圳，是怎样的技术触动技术的心弦？不管是共享经济下各巨头同台竞技还是大数据上中外大牛各显神通，一切都将在12月北京更进一步：立足经典，双十一技术竞技、高可用架构、大数据、移动等，精彩依旧；拥抱热点，视频直播、新闻资讯、Fin-Tech等，引爆架构前沿。更精彩的技术尽在ArchSummit北京2016。

- 领域
- 架构 & 设计
- 语言 & 开发
- 专栏
- 基于组件的架构
- 开发
- APP
- 架构
- 移动
- 操作系统
- iOS

相关内容

阿里热修复技术，减少在App开发中踩坑

iOS 开发周报：Apple CEO 库克称 Mac 新品即将发布

iOS 开发周报：Apple 发布 iPhone 7 / 7 Plus 、Apple Watch 2 等新品

Facebook iOS App技术演化十年之路

iOS 开发周报：Apple 将于 9 月 7 号举行秋季发布会

您好，朋友!

您需要 注册一个InfoQ账号 或者 登录 才能进行评论。在您完成注册后还需要进行一些设置。

获得来自InfoQ的更多体验。

告诉我们您的想法

请输入主题 信息

允许的HTML标签: a,b,br,blockquote,i,li,pre,u,ul,p

当有人回复此评论时请E-mail通知我

发送信息

社区评论 Watch Thread

关闭

by

发布于

- 查看
- 回复
- 回到顶部

关闭

引用原消息

主题 您的回复

允许的HTML标签: a,b,br,blockquote,i,li,pre,u,ul,p

当有人回复此评论时请E-mail通知我

发送信息 取消

关闭

主题 您的回复

允许的HTML标签: a,b,br,blockquote,i,li,pre,u,ul,p

当有人回复此评论时请E-mail通知我

取消

关闭

OK

相关内容

- iOS 开发周报：Apple CEO 库克称 Mac 新品即将发布 2016年9月20日

- iOS 开发周报：Apple 发布 iPhone 7 / 7 Plus 、Apple Watch 2 等新品 2016年9月18日
- iOS 开发周报：Apple 将于 9 月 7 号举行秋季发布会 2016年9月6日
- Visual Studio Code现在支持iOS Web应用调试了 2016年9月5日
- 因为那里有光——高效开发运维微信号开篇记 2016年9月2日
- iOS 开发周报：Apple 推送 iOS 9.3.5 ，修复严重安全漏洞 2016年8月30日
- Facebook iOS App技术演化十年之路 2016年9月9日 
- 同程App 混合开发实践 2016年8月31日 
- iOS交互式动画详解（下）：iOS 10 的新变化 2016年7月29日 
- iOS交互式动画详解（上）：iOS 10以下的实现 2016年7月25日 
- 网易NeteaseAPM iOS SDK技术实现分享 2016年5月24日 


ArchSummit
全球架构师峰会 2016
[北京站]
2016.12.2-3 / 国际会议中心
主办方 Geekbang InfoQ

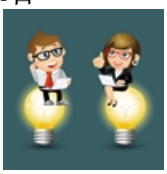
相关内容

- 从视觉到App：网易有钱iOS项目切图与适配实践 2016年5月24日 

- 从Android到Swift iOS开发：语言与框架对比 2016年5月23日 

- iOS遗留系统重构实践 2016年5月5日 

- iOS瘦身之删除FrameWork中无用mach-O文件 2016年4月30日 
- LuaView高性能、动态化、跨平台应用开发引擎—聚划算动态化之路 2016年5月11日 
- Swift 3来了！ 2016年9月26日
- Storyboard 优雅使用规范 2016年7月26日 
- 

- CocoaPods组件半滑二进制化解决方案 2016年7月25日 
- 网易漫画Swift混编实践 2016年4月28日 
- JSPatch开源经验分享 2016年4月7日
- JVM堆内存监测的一种方式，性能调优依旧任重道远 2016年9月19日


amazon web services
Geekbang.
InfoQ
InfoQ在线课堂
解析AWS上的大数据架构模式
时间 2016年9月20日（周二）20:00–21:00
JAVA

相关内容

- Golang中依赖管理的灾难 2016年8月16日 
- 理解质量和可靠性 2016年6月14日 

- InfoQ播客：Cathy O'Neil谈论有害的机器学习算法及审计方法 2016年9月28日
- 巨建华：巨头与传统金融战争方始，FinTech还须技术路径破局 2016年9月28日
- JavaOne 2016主旨演讲畅谈Java近期及远期规划 2016年9月28日
- Consul 0.7新增原子键/值更新和ACL复制特性，提升了协议健壮性 2016年9月28日
- HyperGrid发布了将应用程序迁移到容器的平台 2016年9月28日
- YouTube推荐算法原理 2016年9月28日
- 物联网接入服务架构设计 2016年9月28日 
- 分布式应用无银弹——分布式应用架构核心要素的设计方法探讨 2016年9月28日 
- Kafka和DistributedLog技术对比 2016年9月28日 

InfoQ每周精要

通过个性化定制的新闻邮件、RSS Feeds和InfoQ业界邮件通知，保持您对感兴趣的社区内容的时刻关注。

您的邮箱 订阅

语言 & 开发

巨建华：巨头与传统金融战争方始，FinTech还须技术路径破局

JavaOne 2016主旨演讲畅谈Java近期及远期规划

YouTube推荐算法原理

架构 & 设计

巨建华：巨头与传统金融战争方始，FinTech还须技术路径破局

YouTube推荐算法原理

分布式应用无银弹——分布式应用架构核心要素的设计方法探讨

文化 & 方法

重构和代码异味——通往更整洁的代码

反思领导力：做平凡的事，获得非凡的成果

拒绝重写敏捷宣言

数据科学

InfoQ播客：Cathy O'Neil谈论有害的机器学习算法及审计方法

YouTube推荐算法原理

象SaaS一样用亚马逊Kinesis Analytics做大数据分析

DevOps

HyperGrid发布了将应用程序迁移到容器的平台

Consul 0.7新增原子键/值更新和ACL复制特性，提升了协议健壮性

JavaOne 2016主旨演讲畅谈Java近期及远期规划

InfoQ每周精要

通过个性化定制的新闻邮件、RSS Feeds和InfoQ业界邮件通知，保持您对感兴趣的社区内容的时刻关注。

- InfoQ官方微博
- InfoQ官方微信
- 社区新闻和热点

特别专题

- 活动大本营
- 月刊：《架构师》
- AWS专区
- 百度技术沙龙专区
- 腾讯云专区
- 蝴蝶沙龙
- 信息无障碍参考文档
- 极光推送
- 又拍云专区
- 中国应用性能管理技术大会

定制您感兴趣的技术领域

- [x] 语言 & 开发
- [x] 架构 & 设计
- [x] 数据科学
- [x] 文化 & 方法
- [x] DevOps

这会影响您在主页和RSS订阅中看到的内容。点击“偏好设置”可选择更多精彩定制内容。

提供反馈 feedback@cn.infoq.com
错误报告 bugs@cn.infoq.com
商务合作 sales@cn.infoq.com
内容合作 editors@cn.infoq.com
Marketing marketing@infoq.com