豆瓣 App 的模块化实践

2016-10-29郭麟

本文为投稿文章，作者郭麟。

原文地址：http://lincode.github.io/Modularity

背景

业务背景

豆瓣在 2014 年聚合了移动端业务，推出了一款叫“豆瓣”的 App。随着豆瓣 App 的发展，豆瓣越来越多的业务线被纳入其中。豆瓣 App 代码量越来越多，功能越来越复杂，体积越来越庞大。为了更从容地应对这种状况，使整个项目更健康，我们实施了模块化。模块化的最终目的是独立出几个业务模块，使得各个业务模块互不干扰，可以独立开发。但其实在当前的豆瓣，豆瓣 App 的开发仍然由是一个团队负责，并没有以业务线划分工程团队。所以，业务模块的独立并不是一个对应公司组织构架上的工程要求。而是我们基于项目发展健康做出的考虑。虽然，整个过程伴随着一些痛苦。但结果还是不错的，最后我们得到了以下好处：

- 一个更清晰的项目结构；
- 一些可以复用的公共组件；
- 几个相互隔离的业务模块。

开源成果

我们在模块化过程中，也产出了一些库和工具：

- FRDIntent，处理页面间跳转的库。
- FRDModuleManager，简单的模块管理库。
- Rexxar，移动混合开发框架。

我们将这些工具开源。一方面，是为了给大家提供一些借鉴的方向；另一方面，也是为了提高项目本身的质量。我们知道还存在不少问题。所以，会悉心接受大家的意见和建议。

工程环境

在描述豆瓣的模块化实践之前，先简单介绍一下豆瓣的移动开发中相关的工作环境。

在 iOS 开发中，我们版本管理工具是 Git。公司所有的项目代码都托管在内网搭建的 github enterprise 上。github enterprise 的使用体验和 github.com 基本一致。这受到了工程师的普遍欢迎。

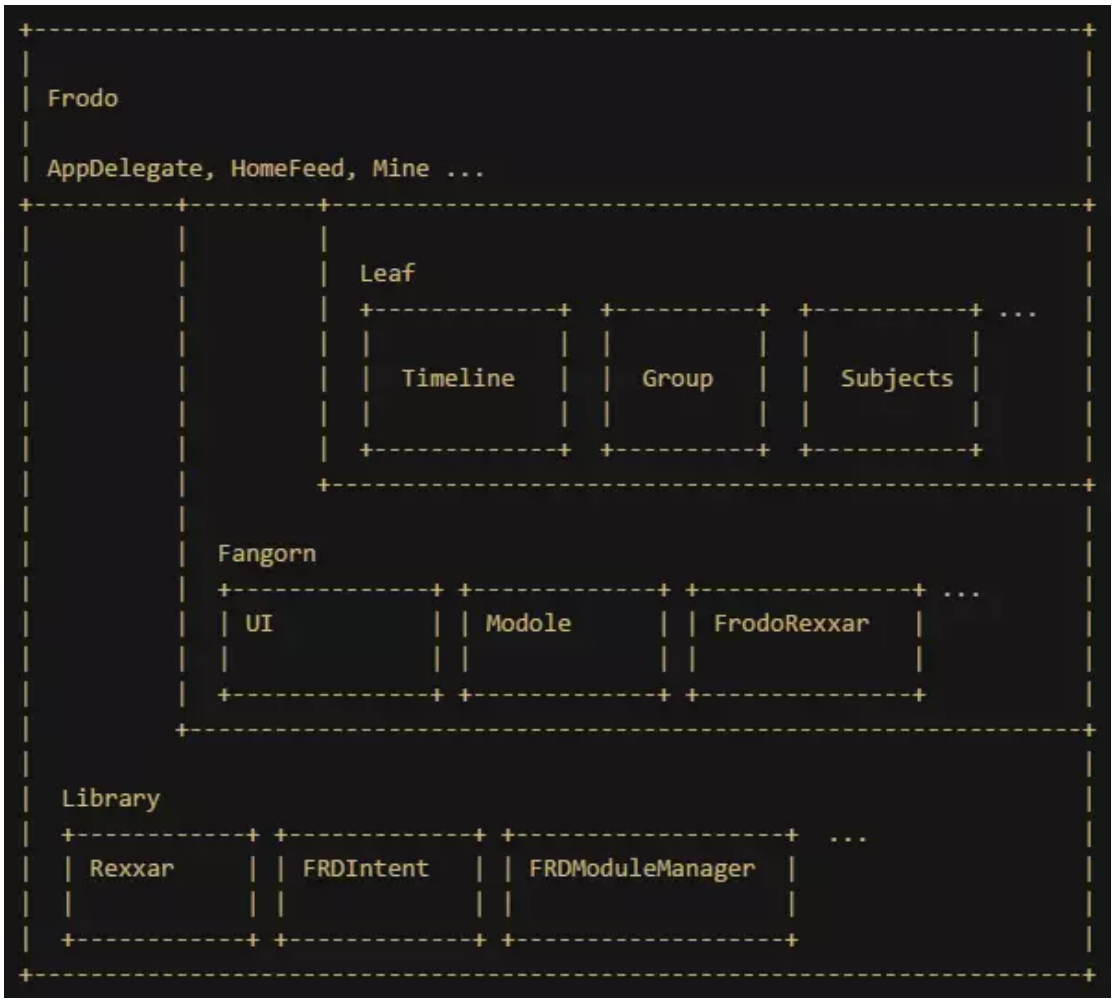
我们管理项目依赖的工具是 Cocoapods。

Git 和 Cocoapods 这两项现在基本上业界的事实标准。本文中，也只有这两个工具和模块化过程有关。

模块化的实施

项目结构

我们首先刻画了模块化之后最终我们希望得到的项目结构。然后向着这个目标，一步步靠近。



Frodo，是豆瓣 App 的项目名，豆瓣有使用魔戒人物命名项目的传统。

- Leaf，需要独立的业务模块在没有拆分之前都放在 Frodo 的 Leaf 文件夹下。
- Timeline，Group，Subjects，是三个拆分出来模块。
- Fangorn，是 Frodo 的公共底层库的项目名。包括了业务相关公共 UI 组件，数据 Model，以及例如 FrodoRexxar 这样的对外部库的封装。
- Library，是我们自己拆分出去的库，或者第三方库。后面会介绍到的 Rexxar，FRDIntent, FRDModuleManager 都属于 Library。

实施步骤

模块化是一个浩大的工程，对于项目有着重大影响。我们在确定目标之后，接着制定了有一个详细的计划。然后按计划一步步实施。

我们把模块化分为四个步骤，下面会分别介绍每个步骤。

文件夹隔离

我们首先需要改变项目的文件组织结构。之前项目简单地以 View / Controller / Model 划分各类文件。现在改为首先按模块划分，每个模块再划分出自己 View / Controller / Model / Network 等文件夹。 比如，广播 Timeline 有自己的 View / Controller / Model 文件夹；小组 Group 也有自己的 View / Controller / Model 文件夹。虽然改为按模块组织项目的文件结构，但此时，所有模块仍然还在一个仓库里。这其实只是做到了文件夹隔离，代码并没有被真正隔离。我们会查看各个文件的 #import 部分，减少业务模块间的相互依赖。几个业务模块都用到的文件，则会沉入到公共层。

这个阶段是没法做到彻底隔离的。因为，理清依赖的过程有赖于程序员自己查看代码。文件位置看上去是分隔开了，但是由于还在一个仓库里，代码依赖肯定没有办法完全处理干净。不过，没有关系，文件夹隔离只是一个过渡阶段。需要这样一个过渡阶段的原因是，产品开发不会因为我们要重构项目就停止。文件夹隔离避免直接拆分一次性集中付出很长时间解决编译错误。而使得团队可以将处理依赖的时间分摊到每个版本中去。一个业务模块在经历文件夹隔离阶段之后，正真需要拆出去时所需要处理的未解决的依赖应该已经所剩不多了。拆分为独立产品时的时间压力就小了很多。

文件夹隔离也为团队提供了一个转变开发方式的缓冲期。业务模块的划分首先需要在全体组员间达成共识。从此，大家开始有了模块化的意识。这表现为，新增文件会被放入对应的模块之中；code review 时会提出不应该引用其他业务模块的要求和建议。文件夹隔离使得组员逐步适应模块化的思维，后续的产品功能也被归入到对应的模块之中。

文件夹隔离，使产品开发和模块化得以并行推进。在不影响产品开发进度的情况下，较从容地推进模块化。

抽象出业务无关的库

我们同时也鼓励将一些业务无关的代码抽象成一个个独立的库。这类库应该是与产品无关，与业务无关的。这就意味着，它们在一定时期内可以保持稳定，不会随着产品和业务的频繁变化而变化。将这些底层逻辑抽象出来，拆分成一个个独立的库，有不少好处：

- 每个库都有了自己清晰的边界。未拆分之前各个库的代码混在项目中，就存在相互干扰的可能性。独立出库使得项目代码中有了更多的隔离，项目质量得到了提升；
- 拆分独立的库使得复用成为了可能，我们可以在新项目中使用它们，甚至将其开源，供其他开发者使用。FRDIntent, FRDModuleManager, Rexxar 都是这种情况；
- 为拆分出公共底层模块 Fangorn 打下了基础。这是因为，拆分出去的模块，必须先处理好它的依赖，它将只能依赖已经拆分出去的组件和第三方库。

拆分出公共底层模块

在 Frodo 中，公共底层模块叫 Fangorn。Fangorn 包括了业务模块所需要的一些公共代码，但是要么是和业务关系较大，要么就是还没到可以抽象成一个库的程度。

在拆分独立出很多业务无关的组件和库之后。我们仍然有一部分代码是公共的，为多个业务模块所使用的，但却和业务有一定的关系。这部分代码由于和业务相关，拆分出去也没有复用的可能。但却是拆分业务模块的前提条件。将 Fangorn 独立为一个仓库之后，我们才能着手业务模块的拆分。

在拆分公共模块时，有两种方法：

- 一种是，将 Fangorn 划分为一个个子模块。将这些子模块一个个拆分出去；
- 一种是，将 Fangorn 作为一个整体先摘出来。Fangorn 内部各子模块之间的依赖关系先按文件夹隔离的方式运作一段时间。如果，发现一个子模块确实可以拆出去了就拆出去。

第一种方式更优雅一些。如果完成，各个业务模块可以选择自己需要的依赖，而不是将 Fangorn 作为一个整体全部依赖。但是也更困难一些。因为，这需要花很长时间理清楚 Fangorn 各个子模块之间的依赖关系。

第二种方式更简单一些。将 Fangorn 作为一个整体摘出来，就只要处理好 Fangorn 和外部其他模块的依赖关系。这个工作量就小很多了。

为了早一点将业务模块独立，我们选择了第二种方式。

业务模块独立

在 Frodo 中，独立的业务模块在文件夹隔离阶段都放在一个叫 Leaf 的文件夹下。我们的目标是拆出广播 Timeline，小组 Group，条目 Subjects 三个模块。

这一部分的工作是解除业务模块之间的依赖。使得这三个模块都只依赖 Fangorn，拆分出去的库，和第三方库。最终的目的是，这三个业务模块独立，并拆分到单独的库中。在经历了相当长时间的文件夹隔离，拆分出不少业务无关的库，并独立了 Fangorn 之后，我们开始按由容易到困难的顺序拆分各个业务模块。第一个待拆分的业务模块是广播 Timeline，而后会是小组 Group，再接着会是条目 Subjects。

最后我们为这三个业务模块都单独建立了可以运行的应用 Demo 项目。这样，它们就可以真正独立开发地开发运行。我们的开发流程就变为：先在这三个模块库中开发新的产品功能，使用模块自有的 Demo 查看结果。完成之后在主项目 Frodo 中升级三个业务模块库的版本，在 Frodo 中验证测试集成效果。

FRDIntent: View controller 间的解耦

在 iOS 项目，存在大量页面跳转。但 iOS 系统并不存在像 Android 的 Intent 一样的统一的页面跳转方法。在 iOS 中，处理页面跳转，需要依赖代表跳转目的页面的类，需要知道它的初始化方法。这样各个页面就需要相互依赖。这种情况对解除耦合，拆分模块很不利。为了解决这个问题，我们做了一个专门处理 iOS 中页面跳转的库：FRDIntent，并将其开源。

Github 地址：https://github.com/douban/FRDIntent

FRDIntent 有两个部分：FRDIntent/Intent 用于解决应用内的页面跳转；FRDIntent/URLRoutes，用于解决外部应用对本应用的页面调用。在使用了 FRDIntent 之后，我们很好地解除了 view controller 之间的耦合。并且为内部调用和外部调用提供了一套清晰统一的解决方案。解决了 iOS 项目中了很大一部分耦合问题：view controller 之间的耦合。为我们顺利推进模块化奠定了坚实基础。

FRDIntent/Intent

FRDIntent/Intent 是一个消息传递对象，用于启动 UIViewController。可以认为它是对 Android 系统中的 Intent 的模仿。当然，FRDIntent/Intent 做了极度简化。这是因为 FRDIntent/Intent 的使用场景更为简单：只处理应用内的 view controller 间跳转。

直接使用 iOS 系统方法完成各 view controller 之间的跳转，各 view controller 代码会耦合得很紧。跳转时，一个 view controller 需要知道下一个 view controller 是如何创建的各种细节。这造成了 view controller 之间的依赖。使用 FRDIntent/Intent 传递 view controller 跳转信息，可以解除 view controller 之间的代码耦合。

FRDIntent/Intent 有如下优势：

- 充分解耦。调用者和被调用者完全隔离，调用者只需要依赖协议：FRDIntentReceivable。一个 UIViewControlller 符合该协议即可被启动。

- 对于“启动一个页面，并从该页面获取结果”这种较普遍的需求提供了一个通用的解决方案。具体查看方法：startControllerForResult。这是对 Android 中 startActivityForResult 的模仿和简化。
- 支持自定义转场动画。
- 支持传递复杂数据对象。

FRDIntent/URLRoutes

FRDIntent/URLRoutes 是一个 URL Router。通过 FRDIntent/URLRoutes 可以用 URL 调起一个注册过的 block。

iOS 系统为各个应用间的相互调用提供了一种基于 URL 的处理方案。即应用可以声明自己可以处理某些有特定 scheme 和 host 的 URL。其他应用就可以通过调用这些 URL 而跳转到该应用的某些页面。

FRDIntent/URLRoutes 是为了让 iOS 系统中这种基于 URL 的应用间调用的处理更为简单。所以 FRDIntent/URLRoutes 和社区已经存在的诸多 URL Routers 的功能和目的差别不大。FRDIntent 再次造了轮子是为了使 FRDIntent/URLRoutes 可以和 FRDIntent/Intent 配合一起解决应用内和应用外的页面调用。

FRDIntent/Intent 和 FRDIntent/URLRoutes

FRDIntent/URLRoutes 和 FRDIntent/Intent 可以配合使用。Intent 处理内部页面跳转；URLRoutes 负责来自外部的页面调用。在 FRDIntent/URLRoutes 的实现中，FRDIntent/URLRoutes 只是起了暴露外部调用入口，接收外部调用的作用。在应用内，仍然是通过 FRDIntent/Intent 启动 view controller。

这么做在隔离了外部调用和内部调用的同时，统一了外部调用和内部调用的实现方法。外部调用最终使用内部调用落地，是自然地复用代码的结果。隔离外部和内部调用则会带来以下这些好处：

- iOS 系统提供的通过 URL 调用另外一个应用功能本身就是使用在应用之间的。iOS 系统中应用之间的隔离是清晰而明确的，通过 URL 在应用之间传递信息是合适的。但是，如果应用内部调用也使用 URL 传递信息，就会带来诸多限制。Intent 更适合内部调用的场景。通过 Intent，可以传递复杂数据对象，可以很容易地自定义转场动画。这些在 URL 方案中都很难做到。

- 区分了外部调用和内部调用，我们就可以选择是否要将一个内部调用给暴露外部使用。这就避免了在 URL 的方案中，无法区分内部调用和外部调用，将本应只给内部使用的调用也暴露给应用外部了这种问题。

  FRDModuleManager：为 AppDelegate 减负
  在开发的过程中，我们发现项目中实现了 UIApplicationDelegate 协议的 AppDelegate 变得越来越臃肿。为了使 AppDelegate 更为健康，各模块可以更容易地获知应用的生命周期事件。我们开发了一个简单的模块管理小工具：FRDModuleManager。这个小工具异常简单，只有一个 .m 文件。我们也将其开源了。

  Github 地址：https://github.com/lincode/FRDModuleManager

  FRDModuleManager 是一个简单的 iOS 模块管理工具。

  FRDModuleManager 可以减小 AppDelegate 的代码量，把很多职责拆分至各个模块中去。使得 AppDelegate 会变得容易维护。

如果你发现自己项目中实现了 UIApplicationDelegate 协议的 AppDelegate 变得越来越臃肿，你可能会需要这样一个类似的小工具；或者如果你的项目实施了组件化或者模块化，你需要为各个模块在 UIApplicationDelegate 定义的各个方法中留下钩子（hook），以便模块可以知晓整个应用的生命周期，你也可能会需要这样一个小工具，以便更好地管理模块在 UIApplicationDelegate 协议各个方法中留下的钩子。

FRDModuleManager 可以使得留在 AppDelegate 的钩子方法被统一管理。实现了 UIApplicationDelegate 协议的 AppDelegate 是我知晓应用生命周期的重要途径。如果某个模块需要在应用启动时初始化，那么我们就需要在 AppDelegate 的 application:didFinishLaunchingWithOptions: 调用一个该模块的初始化方法。模块多了，调用的初始化方法也会增多。最后，AppDelegate 会越来越臃肿。FRDModuleManager 提供了一个统一的接口，让各模块知晓应用的生命周期。这样将使 AppDelegate 得以简化。

严格来说，AppDelegate 除了通知应用生命周期之外就不应该担负其他的职责。对 AppDelegate 最常见的一种不太好的用法是，把全局变量挂在 AppDelegate 上。这样就获得了一个应用内可以使用的全局变量。如果你需要对项目实施模块化的话，挂了太多全局变量的 AppDelegate 将会成为一个棘手的麻烦。因为，这样的 AppDelegate 成为了一个依赖中心点。它依赖了很多模块，这一点还不算是一个问题。但是，由于对全局变量的访问需要通过 AppDelegate，这就意味着很多模块也同时依赖着 AppDelegate，这就是一个

大问题了。这是因为，AppDelegate 可以依赖要拆分出去的模块；但反过来，要拆分出去的模块却不能依赖 AppDelegate。

这个问题，首先要将全局变量从 AppDelegate 上剔除。各个类应该自己直接提供对其的全局访问方法，最简单的实现方法是将类实现为单例。变量也可以挂在一个能提供全局访问的对象上。当然，这个对象不应该是 AppDelegate。

其次，对于 AppDelegate 仅应承担的责任：提供应用生命周期变化的通知。就可以通过使用 FRDModuleManager 更优雅地解决。

这两步之后，AppDelegate 的依赖问题可以很好地解决，使得 AppDelegate 不再是项目的依赖中心点。

Rexxar：混合开发

豆瓣在混合开发方面做了不少实践工作。我们将这个过程的主要产出：Rexxar 开源了。这篇文章（http://lincode.github.io/Rexxar-OpenSource）详细介绍了 Rexxar。

我们推进混合开发的主要目的是提高工程效率。但同时也有一个副产物：由于使用 Rexxar 实现的页面是使用 Web 技术实现整个业务，所以，在效果上，实现 Rexxar 页面的 Web 代码和项目其余的 Native 代码是完全隔离的。我们在业务层使用的前端框架是 React。由于 React 本身组件化的特性，在前端代码内部，项目代码的模块化也做得不错。

所以，从效果上看，混合开发在我们实施模块化的过程中也起了作用。使用 Rexxar 开发的页面除了 Rexxar 这个库，不依赖于项目的其他 Native 代码。

普通对象间的解耦

在处理页面跳转时使用 FRDIntent，已经解决了大部分的模块间调用问题。但这并没有解决所有的耦合问题，项目中模块间除了页面跳转之外，还会有其他对象间的相互依赖。除了跳转到模块内某个页面这种接口之外，一个模块还需暴露某些对象或者数据的话，我们怎么处理这种依赖呢？例如，要暴露一个 UI 组件，或者一个计算结果。用具体的广播 Timeline 模块举例：如果项目的其他模块需要展示一条广播，或者要知道某用户发了多少条广播这种计算结果。这些当然已经在广播模块里实现过了。那么，我们如何提供个一条广播这个 UI 组件，和用户发了多少条广播这个计算结果呢？我们首先采用了两类稍显简单粗糙的方法：

- 如果它们真的完全一样，而且多个模块中用到了。那我们就将其沉入公共底层模块 Fangorn。这样各个模块都依赖公共底层模块，而不用相互依赖；
- 如果它们并不完全一样，我们就可能会拷贝一份代码。为了防止冲突，会重新命名类。由于实施了模块化，一定程度的代码冗余是应该付出的代价，并可以忍受的。

除此之外，可能的解决方案是类似于 Java 社区中常用到的依赖注入容器。例如，Spring 就是一个著名的依赖注入容器。引入依赖注入容器之后，可以动态地创建对象，并为对象注入依赖。而所有依赖都在容器中注册，业务对象只需

要依赖接口，而无需依赖具体的类型。这是一种通用的解耦方法。适用于几乎所有的依赖管理场景。

但在我们的实践中，考虑到避免复杂性，并没有引入一个依赖注入容器。使用依赖注入容器意味着项目中很多对象的创建都需要交托给容器。这对于整个项目的代码运作方式做了很大的改变。对于这类基础性的改变，我们都会比较慎重。而且我们对于在移动平台中，找到或者自己实现这样的一个坚实的基础容器也并不乐观。

遇到的问题

Swift

我们在项目中使用了不少 Swift。这其中遇到了一些问题：

- 现阶段 Swift 本身并不稳定。Swift 2 和 Swift 3 较前一版本都有较大变化。虽然，有 Xcode 提供的自动转换工具，但仍然需要投入精力检查和测试。在一个大型项目中，安全和稳定一般都是首要要求。引入一种处于发展过程中并不稳定的语言对于项目质量是一种威胁。但是，Swift 的情况稍有不同。对于 iOS 开发，我们并没有太多选择。只有 Objective-C 和 Swift，而可以确定的是 Swift 会是未来，Objective-C 将会是历史。那么，我们其实只能选择何时，以何种方式切换到 Swift 而已。我们的经验是先小块实验，再谨慎推进。每次升级现有代码的 Swift 版本都一小块一小块做，不一次性转换所有的代码。

- 使用包含 Swift 代码的库对 iOS 系统版本有要求。如果要使用一个包含了 Swift 代码的库，需要以动态库的形式引入。动态库对 iOS 的系统版本有要求：iOS 8 以及以上版本。我们在 iOS 10 发布时，放弃了对 iOS 7 的支持。这使得我们可以使用包含 Swift 代码的库。如果项目对于低版本 iOS 的支持有要求，那么现阶段就不能使用包含 Swift 代码的库。通过 Cocoapods 将库以动态库形式使用的方法很简单：开启 Cocoapods 的 !use_framework 标识即可。

  Swift 在工程效率上确实优于 Objective-C。和 Objective-C 相比，Swift 可以用更少的代码，更清晰的方式完成相同的功能。当然，混合使用 Swift 和 Objective-C 存在一定的工程成本。所以，这里就需要权衡：是保持简单，只使用 Objective-C 呢？还是忍受一定的不便，使用一些 Swift，带来效率上的提升呢？

  我们在项目中使用 Swift 的体会是：有快乐，当然也伴随着一些不便。总体而言，不便都可以克服。

总结

相信我们在整个模块化实践过程中取得的经验，对于一个成长中的移动项目应该会有某些借鉴意义。我们产出的一些开源库和工具，也可能会对大家有帮助，或者启发意义。也欢迎大家提出反馈和建议，帮助我们改进和提高。