美团外卖Android平台化架构演进实践

原创：吴凯 晓飞 海冰 美团技术团队 2018-03-15

总第227篇

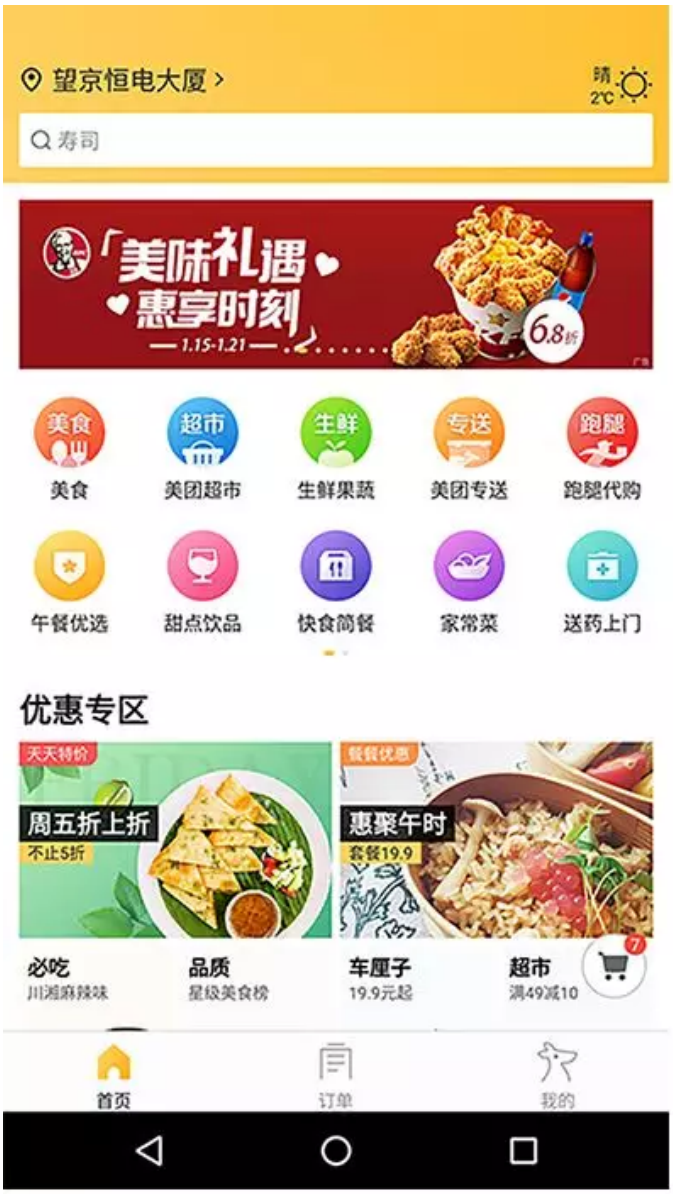
2018年 第19篇

美团外卖自2013年创建以来，业务一直高速发展。目前美团外卖日完成订单量已突破1800万，成为美团点评最重要的业务之一。美团外卖的用户端入口，从单一的外卖独立App，拓展为外卖、美团、点评等多个App入口。美团外卖所承载的业务，也从单一的餐饮业务，发展到餐饮、超市、生鲜、果蔬、药品、鲜花、蛋糕、跑腿等十多个大品类业务。业务的快速发展对客户端架构不断提出新的挑战。

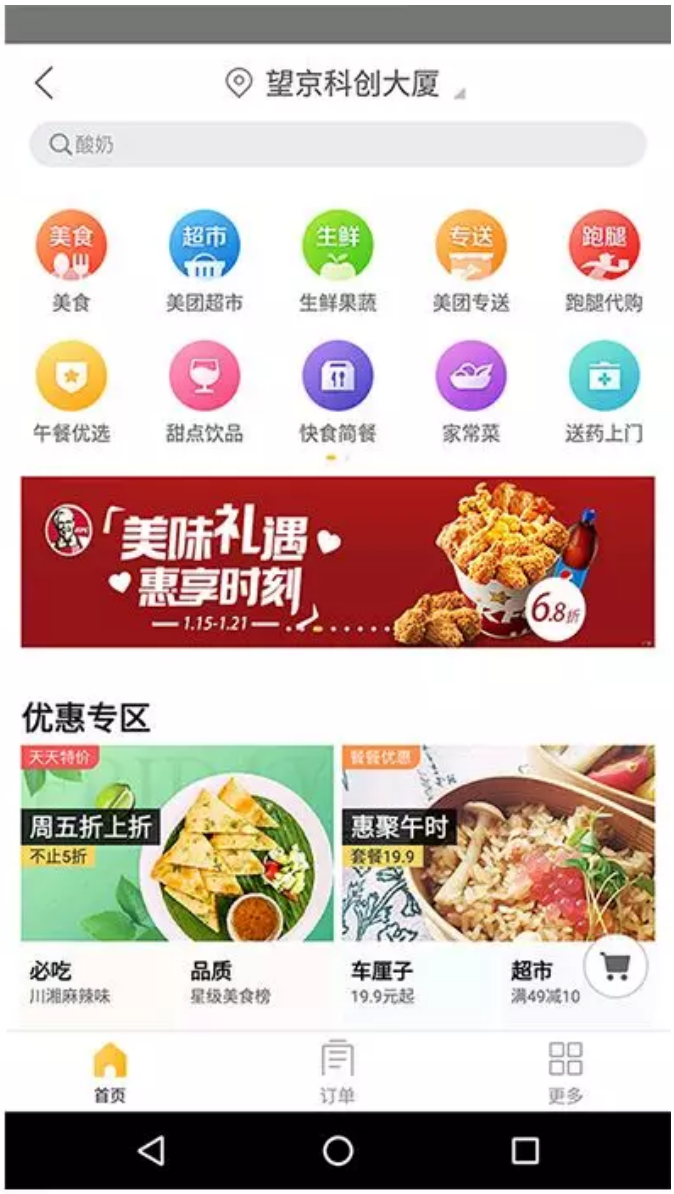
平台化背景

很早之前，外卖作为孵化中的项目只有美团外卖App（下文简称外卖App）一个入口，后来外卖作为一个子频道接入到美团App（下文简称外卖频道），两端业务并行迭代开发。早期为了快速上线，开发同学直接将外卖App的代码拷贝出一份到外卖频道，做了简单的适配就很快接入到美团App了。

早期外卖App和外卖频道由两个团队分别维护，而在随后一段时间里，两端代码体系差异越来越来大。最后演变成了从网络、图片等基础库到UI控件、类的命名等都不尽相同的两套代码。尽管后来两个团队合并到一起，但历史的差异已经形成，为了优先满足业务需求，很长一段时间内，我们只能在两套代码的基础上不断堆积更多的功能。维护两套代码的成本可想而知，而业务的迅猛发展又使得这一问题越发不可忍受。



外卖App首页



外卖频道首页

在我们探索解决两端代码复用的同时，业务的发展又对我们提出新的挑战。随着团队成员扩充了数倍，商超生鲜等垂直品类的拆分，以及异地研发团队的建立，外卖客户端的平台化被提上日程。而在此之前，外卖App和外卖频道基本保持单工程开发，这样的模式显然是无法支持多团队协作开发的。因此，我们需要快速将代码重构为支持平台化的多工程模式，同时还要考虑业务模块的解耦，使得新业务可以拷贝现有的代码快速上线。此外，在实施平台化的过程中，两端代码复用的问题还没有解决，如果两端的代码没有统一而直接做平台化业务拆库，必然会导致问题的复杂化。

在这样的背景下，可以看出我们面临的问题相较于其他平台型App更为特殊和复杂：既要解决外卖业务平台化的问题，又要解决外卖App和外卖频道两端代码复用的问题。

屡次探索

在实施平台化和两端代码复用的道路上并非一帆风顺，很多方案只有在尝试之后才知道问题所在。我们多次遇到这样的情况：设计方案完成后，团队已经全身心投入到开发之中，但是由于业务形态发生变

化，原有的设计也被迫更改。在不断的探索和实践过程中，我们经历了多个中间阶段。虽然有不少失败的案例，但是也积累了很多架构设计上的宝贵经验，整个团队对业务和架构也有了更深的理解。

搜索库拆分实践

早期美团外卖App和美团外卖频道两个团队的合并，带来的最大痛点是代码复用，而非平台化，而在很长的一段时间内，我们也没有想过从平台化的角度去解决两端代码复用的问题。然而代码复用的一些失败尝试，给后续平台化的架构带来了不少宝贵的经验。当时是怎么解决代码复用问题的呢？我们通过和产品、设计同学的沟通，约定了未来的需求，会从需求内容、交互、样式上，两端尽可能的保持一致。经过多次讨论后，团队发起了两端代码复用的技术方案尝试，我们决定将搜索模块从主工程拆分出来，并实现两端代码复用。然而两端的搜索模块代码底层差异很大，BaseActivity和BaseFragment不统一，UI样式不统一，数据Model不统一，图片、网络、埋点不统一，并且两端发版周期也不一致。针对这些问题的解决方案是：

1. 通过代理屏蔽Activity和Fragment基类不统一的问题；
2. 两端主工程style覆盖搜索库的UI样式；
3. 搜索库使用独立的数据Model，上层去做数据适配；
4. 其他差异通通抛出接口让上层实现；
5. 和PM沟通尽量使产品需求和发版周期一致。

架构大致如图：

虽然搜索库在短期内拆分为独立的工程，并实现了绝大部分的两端代码复用，但是好景不长，仅仅更新过几个版本后，由于需求和版本发布周期的差异，搜索库开始变为两个分支，并且两个分支的差异越来越大，最后代码无法合并而不得不永久维护两个搜索库。搜索库事实上是一次失败的拆分，其中的问题总结起来有三个：

1. 在两端底层差异巨大的情况下自上而下的强行拆分，导致大量实现和适配留在了两端主工程实现，这样的设计层级混乱，边界模糊，并且极大的增加了业务开发的复杂性；
2. 寄希望于两端需求和发版周期完全一致这个想法不切实际，如果在架构上不为两端的差异性预留可伸缩的空间，复用最终是难以持续的；
3. 约定或规范，受限于组织架构和具体执行的个人，不确定性太高。

页面组件化实践

在经历过搜索库的失败拆分后，大家认为目前还不具备实现模块整体拆分和复用的条件，因此我们走向了另一个方向，即实现页面的组件化以达成部分组件复用的目标。页面组件化的设计思路是：

1. 将页面拆分为粒度更小的组件，组件内部除了包含UI实现，还包含数据层和逻辑层；
2. 组件提供个性化配置满足两端差异需求，如果无法满足再通过代理抛到上层处理。

页面组件化是一个良好的设计，但它主要适用于解决Activity巨大化的问题。由于底层差异巨大的情况，使得页面组件化很难实现大规模的复用，复用效率低。另一方面，页面组件化也没有为2端差异性预留可伸缩的空间。

MVP分层复用实践

我们还尝试过运用设计模式解决两端代码复用的问题。想法是将代码分为易变的和稳定的两部分，易变部分在两端上层实现差异化处理，稳定部分可以在下层实现复用。方案的主要设计思路是：

1. 借鉴Clean MVP架构，根据职责将代码拆分为Presenter，Data Repository，Use Case，View，Model等角色；
2. UI、动画、数据请求等逻辑在下层仅保留接口，在上层实现并注入到下层；
3. 对于两端不一致的数据Model，通过转换器适配为下层统一的模型。

架构大致如图：

这是一种灵活、优雅的设计，能够实现部分代码的复用，并能解决两端基础库和UI等差异。这个方案在首页和二级频道页的部分模块使用了一段时间，但是因为学习成本较高等原因推广比较缓慢。另外，这个时期平台化已被提上日程，业务痛点决定了我们必须快速实施模块整体的拆分和复用，而优雅的设计模式并不适合解决这一类问题。即使从复用性的角度来看，这样的设计也会使得业务开发变得更为复杂、调试困难，对于新人来说难以胜任，最终推广落地困难。

中间层实践

通过多次实践，我们认识到要实现两端代码复用，基础库的统一是必然的工作，是其他一切工作的基础。否则必然导致复杂和难以维护的设计，最终导致两端复用无法快速推进下去。

计算机界有一句名言：“计算机科学领域的任何问题都可以通过增加一个中间层来解决。”（原始版本出自计算机科学家David Wheeler）我们当然有想过通过中间层设计屏蔽两端的基础库差异。例如网络库，外卖App基于Volley实现，外卖频道基于Retrofit实现。我们曾经在Volley和Retrofit之上封装了一层网络框架，对外暴露统一的接口，上层可以切换底层依赖Volley或是Retrofit。但这个中间层并没有上线，最终我们将两端的网络库统一成了Retrofit。

这里面有多个原因：首先Retrofit本身就是较高层次的封装，并且拥有优雅的设计模式，理论上我们很难封装一套扩展性更强的接口；其次长期来看底层网络框架变更的风险极低，并且适配网络层的各种插件

也是一件费时费力的事情，因此保持网络中间层的性价比极低；此外将两端的网络请求都替换为中间层接口，显然工作量远大于只保留一端的依赖。

通过实践我们认识到，中间层设计是一把双刃剑。如果基础框架本身的扩展性足够强，中间层设计就显得多此一举，甚至丧失了原有框架的良好特性。

平台化实践

好的架构源于不停地衍变，而非设计。对于外卖Android客户端的平台化架构构建也是经历了同样的过程。我们从考虑如何解决代码复用的问题，逐渐的衍变成如何去解决代码复用和平台化的两个问题。而实际上外卖平台化正是解决两端代码复用的一剂良药。我们通过建立外卖平台，将现有的外卖业务降级为一个频道，将外卖业务以aar的形式分别接入到外卖平台和美团平台，这样在解决外卖平台化的同时，代码复用的问题也将得到完美的解决。

平台化架构

经过了整整一年的艰苦奋斗，形成了如图所示的美团外卖Android客户端平台化架构：

从底层到高层依次为平台层、业务层和宿主层。

1. 平台层的内容包括，承载上层的数据通信和页面跳转；提供外卖核心服务，例如商品管理、订单管理、购物车管理等；提供配置管理服务；提供统一的基础设施能力，例如网络、图片、监控、报警、定位、分享、热修、埋点、Crash上报等；提供其他管理能力，例如生命周期管理、组件化等。
2. 业务层的内容包括，外卖业务和垂直业务。
3. 宿主层的内容包括，Waimai App壳和美团外卖频道Waimai-channel壳，这一层用于Application的初始化、dex加载和其他各种必要的组件或基础库的初始化。

在构建平台化架构的过程中，我们遇到这样一个问题，如何长久的维持我们平台化架构的层级边界。试想，如果所有的代码都在一个工程里面开发，通过包名、约定去规范层级边界，任何一个紧急的需求都可能破坏层级边界。维持层级边界的最好办法是什么？我们的经验是工程隔离。平台化的每一层都去做工程隔离，业务层的每个业务都建立自己的工程库，实现工程隔离。同时，配套编译脚本，检查业务库之间是否存在相互依赖关系。工程隔离的好处是显而易见的：

1. 每个工程都可以独立编译、独立打包；
2. 每个工程内部的修改，不会影响其他工程；
3. 业务库工程可以快速拆分出来，集成到其他App中。

但工程隔离带来的另一个问题是，同层间的业务库需要通信怎么办？这时候就需要提供业务库通信框架来解决这个问题。

业务库通信框架

在拆分外卖商家业务库的时候，我们就发这样一个案例：在商家页有一个业务，当发现当前商家是打烊的，就会弹出一个浮层，推荐相似的商家列表，而在我们之前划分的外卖子业务库里面，相似商家列表应该是属于页面库里面的内容。那怎么让商家业务库访问到页面库里面的代码呢。如果我们将商家库去依赖页面库，那我们的层级边界就会被打破，我们的依赖关系也会变得复杂。因此我们需要在架构中提供同层间的通信框架，它去解决不打破层级边界的情况下，完成同层间的通信。

汇总同层间通信的场景，大致上可以划分为：页面的跳转、基本数据类型的传递（包括可序列化的共有类对象的传递）、模块内部自定义方法和类的调用。针对上述情况，在我们的架构里面提供了二种平级间的通信方式：scheme路由和美团自建的ServiceLoaders sdk。scheme路由本质上是利用Android的scheme原理进行通信，ServiceLoader本质上是利用的Java反射机制进行通信。

scheme路由的调用如图所示：

最终效果：所有业务页面的跳转，都需要通过平台层的scheme路由去分发。通过scheme路由，所有业务都得到解耦，不再需要相互依赖而可以实现页面的跳转和基本数据类型的传递。

serviceloader的调用如图所示：

提供方和使用方通过平台层的一个接口作为双方交互的约束。使用方通过平台层的ServiceLoader完成提供方的实现对象获取。这种方式可以解决模块内部自定义方法和类的调用，例如我们之前提到了商家库需要调用页面库代码的问题就可以通过ServiceLoader解决。

外卖内核模块设计

在实践的过程中，我们也遇到业务本身上就不好划分层级边界的业务。大家可以从美团外卖三层架构图上，看出外卖业务库，像商家、订单等，是和外卖的垂类业务库是同级的。而实际上外卖业务的子业务是否应该和垂类业务保持同层是一个目前无法确定的事情。

目前，外卖接入的垂类业务商超业务，是隶属于外卖业务的子频道，它依然依赖着外卖的核心model、核心服务，包括商品管理、订单管理、购物车管理等，因此目前它和外卖业务的商家、订单这样的子业务库同层是没有问题的。但随着商超业务的发展，商超业务未来可能会建设自己的商品管理、订单管理、购物车管理的服务，那么到时商超业务就会上升到和外卖业务一样同层的业务。这时候，外卖核心管理服务，处在平台层，就会导致架构的层级边界变得不再清晰。

我们的解决办法是通过设计一个属于外卖业务的内核模块来适应未来的变化，内核模块的设计如图：

1. 内圈为基础模型类，这些模型类构成了外卖核心业务（从门店→点菜→购物车→订单）的基础；
2. 中间圈为依赖基础模型类构建的基础服务（CRUD）；
3. 最外圈为外卖的各维度业务，向内依赖基础模型圈和外卖基础服务圈。

如果未来确定外卖平台需要接入更多和外卖平级的业务，且最内圈都完全不一样，我们将把外卖内核模块上移，在外卖业务子库下建立对内核模块的依赖；如果未来只是有更多的外卖子业务的接入，那就继续保留我们现在的架构；如果未来接入的业务基础模型类一样，但自己的业务服务需要分化，那么我们将对保留内核模块最核心的内圈，并抽象出服务层由外卖和商超上层自己实现真正的服务。

业务库拆分

在拆分业务库的时候，我们面临着这样的问题：业务之间的关系是较为复杂的，如何去拆分业务库，才是较为合理的呢？一开始我们准备根据外卖业务核心流程：页面→商家→下单，去拆分外卖业务。但是随着外卖子频道业务的快速发展，子频道业务也建立了自己的研发团队，在页面、商家、下单等环节，也开始建立自己的页面。如果我们仍然按照外卖下单的流程去拆分库，那在同一个库之间，就会有外卖团队和外卖子频道团队共同开发的情况，这样职责边界很不清晰，在实际的开发过程中，肯定会出现理不清的情况。

我们都知道软件工程领域有所谓的康威定律：

> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. - Melvin Conway(1967)

翻译成中文的大概意思是：设计系统的组织，其产生的设计等同于组织之内、组织之间的沟通结构。

在康威定理的指导下：我们认为技术架构应该反映出团队的组织结构，同时，组织结构的变迁，也应该导致技术架构的演进。美团外卖平台下包含外卖业务和垂直品类业务，对于在我们团队中已经有了组织结构，优先组织结构，去拆出独立的业务库，方便子业务库的同学内部沟通协作，减少他们跨组织沟通的成本。同时，我们将负责外卖业务的大团队，再进一步细化成页面小组、商家小组和订单小组，由这些小组的同学去在外卖业务下完成更细维度的外卖子业务库拆分。根据组织结构划分的业务库，天然的存在业务边界，每个同学都会按照自己业务的目标去继续完善自己的业务库。这样的拆库对内是高内聚，对外是低耦合的，有效的降低了内外沟通协作的成本。

工程内代码隔离

在实现工程隔离之后，我们发现工程内部的代码还是可以相互引用的。工程内部如果也不能实现代码的隔离，那么工程内部的边界就是模糊的。我们希望工程内至少能够实现页面级别的代码隔离，因为Activity是组成一个App的页面单元，围绕这个Activity，通常会有大量的代码及资源文件，我们希望这些代码和资源文件是被集中管理的。

通常我们想到的做法是以module工程为单位的相互隔离，但在module是相对比较重的一个约束，难道每个Activity都要建一个module吗？这样代码结构会变得很复杂，而且针对一些大的业务体，又会形成巨大化的module。

那我们又想到规范代码，用包名去人为约定，但靠包名约束的代码，边界模糊，时不时的紧急需求，就把包名约定打破了，而且资源文件的摆放也是任意的，迁移成本高。

那怎么去解决工程内部的边界问题呢？《微信的模块化架构重构实践》一文中提到了一个重要的概念p(pins)工程，p工程可谓是工程内约束代码边界的重要法宝。通过在Gradle里面配置sourceSets，就可以改变工程内的代码结构目录，完成代码的隔离，配置示例：

```
sourceSets {
    main {
        def dirs = ['p_widget', 'p_theme',
                    'p_shop', 'p_shopcart',
                    'p_submit_order','p_multperson','p_again_order',
                    'p_location', 'p_log','p_ugc','p_im','p_share']
        dirs.each { dir ->
            java.srcDir("src/$dir/java")
            res.srcDir("src/$dir/res")
        }
    }
}
```

效果如图所示：

从图上可以可以看出，这个业务库被以页面为单元拆分成了多个p工程，每个p工程的边界都是清楚的，实现了工程内的代码隔离。工程内代码隔离带来的好处显而易见：

1. p工程实现了最小粒度的代码边界约束；
2. 工程内模块职责清晰；
3. 业务模块可以被快速的拆分出来。

代码复用

p工程满足了工程内代码隔离的需求，但是别忘了，我们每个模块在外卖两个终端上（外卖App&美团App）上可能存在差异，如果能在模块内部实现两端差异，我们的目标才算达成。基于上述考虑，我们想到了使用Gradle提供的productFlavors来实现两端的差异化。为此，我们需要定义两个flavor：wm和mt。

```
productFlavors {
     wm {}
     mt {}
}
```

但是，这样生成的p工程是并列的，也就是说，各个p工程中所有的差异化代码都需要被存放在这两个flavor对应的SourceSet下，这岂不是跟模块间代码隔离的理念相违背？理想的结构是在p工程内部进行flavor划分，由p工程内部包容差异化，继续改成Gradle脚本如下：

```
productFlavors {
    wm {}
    mt {}
}
sourceSets {
    def dirs = ['p_restaurant', 'p_goods_detail', 'p_comment', 'p_compose_order',
                'p_shopping_cart', 'p_base', 'p_product_set']
    main {
        manifest.srcFile 'src/p_restaurant/main/AndroidManifest.xml'
 dirs.each { dir ->
            java.srcDir("src/${dir}/main/java")
            res.srcDir("src/${dir}/main/res")
        }
    }
    wm {
        dirs.each { dir ->
            java.srcDir("src/${dir}/wm/java")
```

```
                res.srcDir("src/${dir}/wm/res")
            }
        }
        mt {
            dirs.each { dir ->
                java.srcDir("src/${dir}/mt/java")
                res.srcDir("src/${dir}/mt/res")
            }
        }
    }
```

最终工程结构变成如下：

通过p工程和flavor的灵活应用，我们最终将业务库配置成以p工程为维度的模块单元，并在p工程内部兼容两端的共性及差异，代码复用被很好的解决了。同时，两端差异的问题是归属在p工程内部自己处理的，并没有建立中间层，或将差异抛给上层壳工程去完成，这样的设计遵守了边界清晰，向下依赖的原

则。

但是，工程内隔离也存在与工程隔离一样的问题：同层级p工程需要通信怎么办？我们在拆分商家库的时候，就面临这这样的问题，商品活动页和商品详情页，可以根据页面维度，去拆分成2个p工程，这两个页面都会用到同一个商品样式的item。如何让同层间商品活动页p工程和商品详情页p工程访问到商品样式item呢？在实际拆库的实践中，我们逐渐的探索出三级工程结构。三级工程结构不仅可以解决工程内p工程通信的问题，而且可以保持架构的灵活性。

三级工程结构

三级工程结构，指的是工程→module→p工程的三级结构。我们可以将任何一个非常复杂的业务工程内部划分成若干个独立单元的module工程，同时独立单元的module工程，我们可以继续去划分它内部的独立p工程。因为module是具备编译时的代码隔离的，边界是不容易被打破的，它可以随时升级为一个工程。需要通信的p工程依赖module的主目录，base目录，通过base目录实现通信。工程和module具有编译上隔离代码的能力，p工程具有最小约束代码边界的能力，这样的设计可以使得工程内边界清晰，向下依赖。设计如图所示：

三级工程结构的最大好处就是，每级都可按照需要灵活的升级或降级，这样灵活的升降级，可以随时适应团队组织结构的变化，保持架构拆分合并的灵活性，从而动态的满足了康威定理。

工程化建设

平台化一个直观的结果就是产生了很多子库，如何对这些子库进行有效的工程化管理将是一个影响团队研发效率的问题。目前为止，我们从以下两个方面做了改进。

一键切源码

主工程集成业务库时，有两种依赖模式：aar依赖和源码依赖。默认是aar依赖，但是在平时开发时，经常需要从aar依赖切换到源码依赖，比如新需求开发、bugfix及排查问题等。正常情况我们需要在各个工程的build.中将compile aar手动改为compile project，如果业务库也需要依赖平台库源码，也要做类似的操作。如下图所示：

这样手动操作会带来两个问题：

1. build.gradle改动频繁，如果开发人员不小心push上去了，将会造成各种冲突。
2. 当业务库越来越多时，这种改动的成本就越来越大了。

鉴于这种需求具备通用性，我们开发了一个Gradle插件，通过主工程的一个配置文件（被git ignore），可一键切换至源码依赖。例如需要源码依赖商家库，那么只需要在主工程中将该库的源码依赖开关打开即可。商家库还依赖平台库，默认也是aar依赖，如果想改成源码依赖，也只需把开关打开即可。

一键打包

业务库增多以后，构建流程也变得复杂起来，我们交付的产物有两种：外卖App的apk和外卖频道的aar。外卖App的情况会简单一些，在Jenkins上关联各个业务库指定分支的源码，直接打包即可。而外卖频道的情况则比较复杂，因为受到美团平台的一些限制，频道打包不能直接关联各个业务库的源码，只能依赖aar。按照传统做法，需要逐个打业务库的aar，然后统一在频道工程中集成，最后再打频道aar，这样效率实在太低。为此，我们改进了频道的打包流程。如下图所示：

先打平台库aar，打完后自动提PR到各个业务库去修改平台库的版本号，接着再逐个触发业务库去打aar，业务库打完aar之后再自动提PR到频道主库去修改业务库的版本号，等全部业务库aar打完后最后再自动触发打频道主库的aar，至此一键打包完毕。

平台化总结

从搜索库拆分的第一次尝试算起，外卖Android客户端在架构上的持续探索和实践已经经历了2年多的时间。起初为了解决两端代码复用的问题，我们尝试过自上而下的强行拆分和复用，但很快就暴露出层次混乱、边界模糊带来的问题，并且认识到如果不能提供两端差异化的解决方案，代码复用是很难持续的。后来我们又尝试过运用设计模式约束边界，先实现解耦再进行复用，但在推广落地过程中认识到复杂的设计很难快速推进下去。

在平台化开始的时候，团队已经形成了设计简单、边界清晰的架构理念。我们将整体结构划分为宿主层、业务层、平台层，并严格约束层次间的依赖关系。在业务模块拆分的过程中，我们借鉴微信的工程结构方案，按照三级工程结构划分业务边界，实现灵活的代码隔离，并降低了后续模块迁出和迁入成本，使得架构动态满足康威定律。

在两端代码复用的问题上，我们认识到要实现可持续的代码复用，必须自下向上的逐步统一两端底层的基础依赖，同时又能容易的支持两端上层业务的差异化处理。使用Flavor管理两端的差异代码，尽量减少向上依赖，在具体实施时应用之前积累的解耦设计的经验，从而满足了架构的可伸缩性。

没有一个方案能获得每个人的赞同。在平台化的实施过程中，团队成员多次对方案选型发生过针锋相对的讨论。这时我们会抛开技术方案，回到问题本身，去重新审视业务的痛点，列出要解决的问题，再回过头来看哪一个方案能够解决问题。虽然我们并不常常这么做，但某些时刻也会强制决策和实施，遇到问题再复盘和调整。

任何一种设计理念都有其适用场景。我们在不断关注业内一些优秀的架构和设计理念，以及公司内部美团App、点评App团队的平台化实践经验，学习和借鉴了许多优秀的设计思想，但也由于盲目滥用踩过不少坑。我们认识到架构的选择正如其他技术问题一样，应该是面向问题的，而不是面向技术本身。架构的演进必须在理论和实践中交替前行，脱离了其中一个谈论架构，都将是个悲剧。

展望

平台化之后，各业务团队的协作关系和开发流程都发生了很大转变。在如何提升平台支持能力，如何保持架构的稳定性，如何使得各业务进一步解耦等问题上，我们又将面对新的问题和挑战。其中有三个问题是亟待我们解决的：

1. 要确保在长期的业务迭代中架构不被破坏，除了流程规范之外，还需要在本地编译、远程提交、代码合并、打包提测等各个阶段建立更健全的检查工具来约束，而目前这些工具链还并不完善。
2. 插件化架构是平台型App集成的最好方式，不仅使得子业务具备动态发布的能力，还可以解决令人头疼的编译速度问题。目前美团平台已经在部分业务上较好的实现了插件化集成，外卖正在跟进。
3. 统一页面级开发的标准化框架，可以解决代码的可维护性、可测试性，和更细粒度的可复用性，并且有利于各种自动化方案的实施。目前我们正在部分业务尝试，后续会持续推进。

参考资料

1. MVP + Clean Architecture
2. 58同城沈剑：好的架构源于不停地衍变，而非设计
3. 每个架构师都应该研究下康威定理
4. 微服务架构的理论基础 - 康威定律
5. 架构的本质是管理复杂性，微服务本身也是架构演化的结果
6. 微信Android模块化架构重构实践
7. 配置构建变体
8. 美团App 插件化实践

作者简介

吴凯，美团点评技术专家。2016年加入美团点评，目前负责外卖用户端Android团队，主要致力于外卖Android平台化业务支持和技术建设。

晓飞，美团点评资深工程师。2015年加入原美团，是外卖Android的早期开发者之一，目前作为外卖Android App负责人，主要负责平台和业务架构。

海冰，美团点评高级工程师。2015年加入原美团，曾支持开店宝等B端业务，目前作为外卖Android App主力开发，负责商家容器模块，及平台化相关推进工作。

---------- END ----------

招聘信息

> 美团外卖长期招聘Android、iOS、FE 高级/资深工程师和技术专家，base 北京、上海、成都，欢迎有兴趣的同学投递简历到wukai05#meituan.com。

也许你还想看：

大众点评App的短视频耗电量优化实战

美团外卖前端可视化界面组装平台 —— 乐高

Android动态日志系统Holmes