滴滴的组件化实践与优化

| 作者 李贤辉 发布于 2016 年 9 月 8 日

本文根据李贤辉在 2016GMTC 全球移动开发大会上的演讲整理而成。

我叫李贤辉，于 2013 年 5 月份加入滴滴，经历了从早期滴滴 2.0 版本到现在 4.3.8 版本的一系列版本，包括了微信红包补贴大战等各种大事记。

今天分享的主题是滴滴组件化实践与优化。第一部分介绍组件化，是 2015 年 6 月 1 号开始进行的、到 2015 年 12 月 31 号上线，历时七个月的工作。第二部分介绍专项技术，包括地图业务模块解耦、界面解耦的内容。第三部分是滴滴客户端现在面临的问题和思考，以及正在做的事情。

组件化

我先说一下组件化。1 表示一个 Project，滴滴早期只有出租车一个业务，业务也相对简单，那时一个 Project 就可以解决问题。2015 年 6 月，iOS 开发人员约有 70 人，分布在北京、上海、杭州三个城市，有 7 条业务线，如出租车、快车、专车、顺风车等，同时，这个 Project 有 70 万行代码。

在滴滴组件化之前，我们遇到了一些问题。第一，代码冲突多。每一次拉下代码开发功能，开发完成准备提交代码时，往往有其他工程师提交了代码，需要重新拉去代码合并后再提交，一般合并代码的过程需要半个小时以上。即使开发一个很小的功能，也需要在整个工程里做编译和调试，效率较低。第二，迭代速度慢。出租车、快车是滴滴早期的业务线，和公共代码相互耦合，代码冲突较多；等快要发版时，所有的业务线修改都需要全量回归，每次发版都比较晚。第三，版本风险。由于是源码级集成，如果想增加或者撤回功能都有风险，都需要修改很多代码，风险不可控。

2015 年，我们启动了代号为 TheOne 的项目，这个项目有两个目标。第一，提供技术组件和业务组件，技术组件的目标是可以跨 App 使用，业务组件是在乘客端里使用，通过组件化把代码分割在不同的方格里，把模块间的边界划分清楚，也方便未来在方格里做重构。第二，改善迭代体验，协同发版，目前发版的难度大，每个人都非常忙。

以上就是 2015 年 6 月 1 号启动的项目。当时组建了一个五人小分队，成立了平台产品中心。大约在 9 月，我们完成了框架的雏形，然后先接入顺风车来验证框架。10 月，顺风车顺利接入。在 11 月，开始接入了所有的业务线。12 月 31 号，项目正式上线。

以前开发任何功能都需要编译整个工程。我们看一下出租车业务线现在的开发迭代，有网络、日志、还有业务，基本上只需要依赖自己，即出租车自身的业务线以及一些平台的业务，不再需要依赖专车、快车的代码。这里对业务线有强制要求，业务线是强制隔离的。

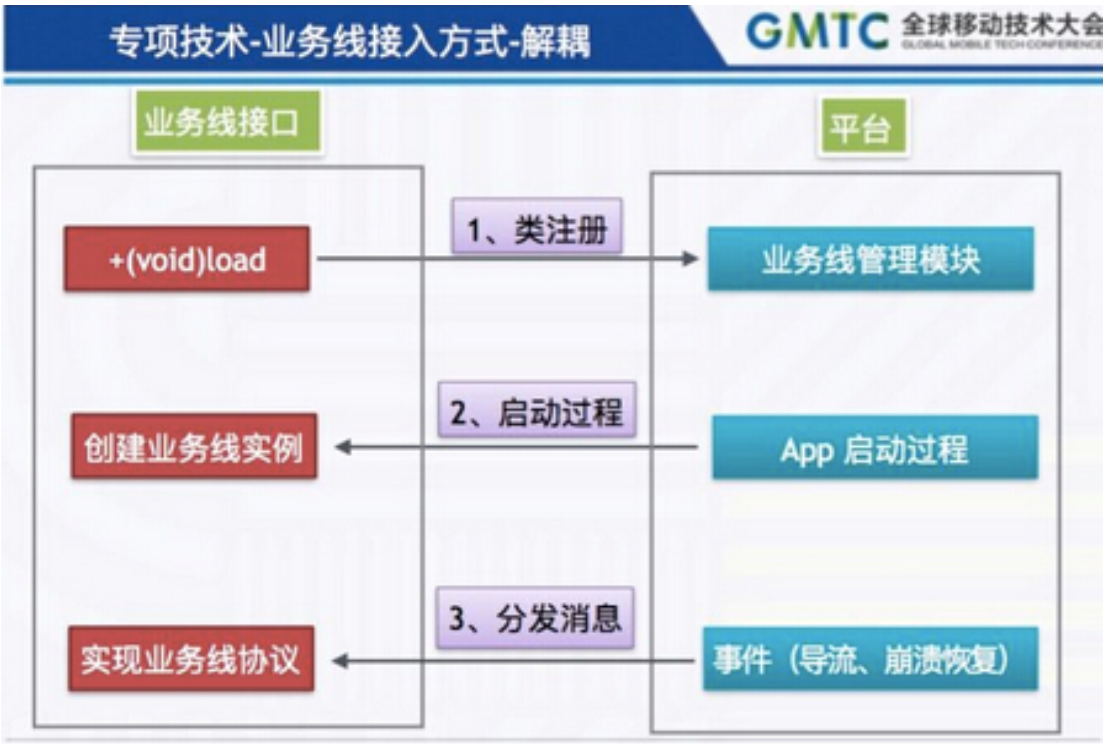
再看一下代码库。出租车业务线拥有独立的代码仓库，公共代码库放在平台上。然后是业务线的集成。业务线或者 SDK 是在独立的项目中开发测试的，然后进行打包，变为滴滴出行整体 App，整体 App 包含快车、专车等业务线。

那么，现在是不是解决了之前的痛点？可以从三个方面来看。首先是开发体验方面。开发人员按不同项目组成团队，负责不同代码仓库，一个代码仓库也就十几个人，一个项目组只开发一个组件，冲突就比以前少了，通常不需要解决冲突。其次是迭代速度方面。集成更加简单了，业务线只需更新自己的 tag，不需要用源码去合并。第三是版本风险方面。因为我们完全用 tag 的方式集成，更新或者回归 tag 都很简单，可能只修改一个数字就好

了。从现实情况看，我们现在发布新版本，一般在晚上九点前就能完全准备好，而且下午两点之后不允许改代码。

专项技术

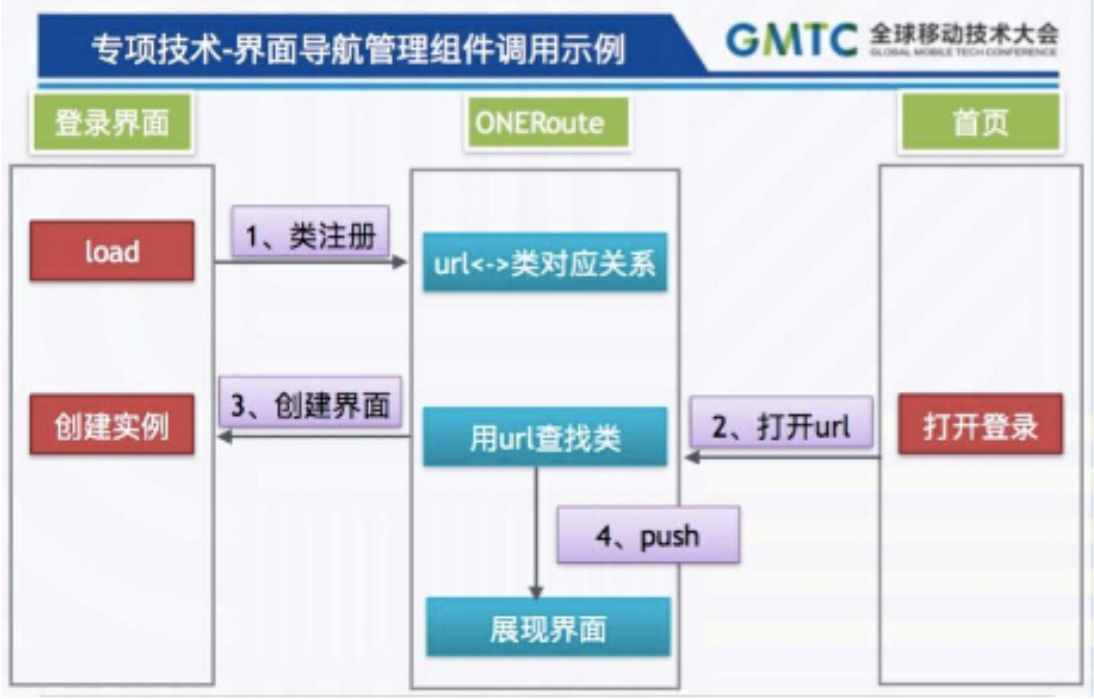
接下来讲讲专项技术，包括业务线接入、页面结构、地图模块等内容。



如果要接入出租车业务线，应该怎么办？出租车业务线接口类先注册到一个业务线的管理模块，这个过程会在 didFinishLaunching 之前，在 didFinishLaunching 之后，业务线管理模块会把注册过的业务线全部初始化，并且加载一遍。之后遇到的系统层面的事件，例如订单恢复，即一个订单没有结束，App 被 Kill 后，重新打开时，系统会提示你进入哪条业务线，平台会获取该事件，分发给业务线，之后业务线会根据不同情况跳转到等待应答或者等待接驾页面。通过这个方式可以把业务线接入进来。

首先，滴滴出行的业务结构，包含出租车、专车等，所有业务线都没包含在 Project 里。想一想，如果首页在 Project 里，如果要集成出租车业务线或者快车业务线，会遇到很大麻烦，所以首页必须在组件里。第二，每个首页都是子页面，当我们切换首页的时候就相当于切换子页面。第三，业务线的首页只需要从 UIViewController 派生就够了。这种方式对业务线来讲，学习成本偏高，因为需要学习平台特有的属性，对业务的开发更加不利。所以最终采取的方式是，业务线只要从系统的 UIViewController 派生，当切换业务线时，框架层面会设置背景为透明，这样就简化了业务线的开发。

从界面来看，地图、导航都不在最前面。我们采用的处理方式是，在后边有一个 contentView，我们知道在所有的点击事件中，后面的 view 会更早获取这些点击事件，会处理 hitTest 的方法，判断它所在的区域。如果区域在最顶部，我们会传递给最上面的组件，如果在业务线选择的部分，我们会传递给业务线选择，如果是在地图区域，我们会传递给地图。通过这种方式，既让业务线简单了，也实现了业务的一些功能，简化了业务线的开发。



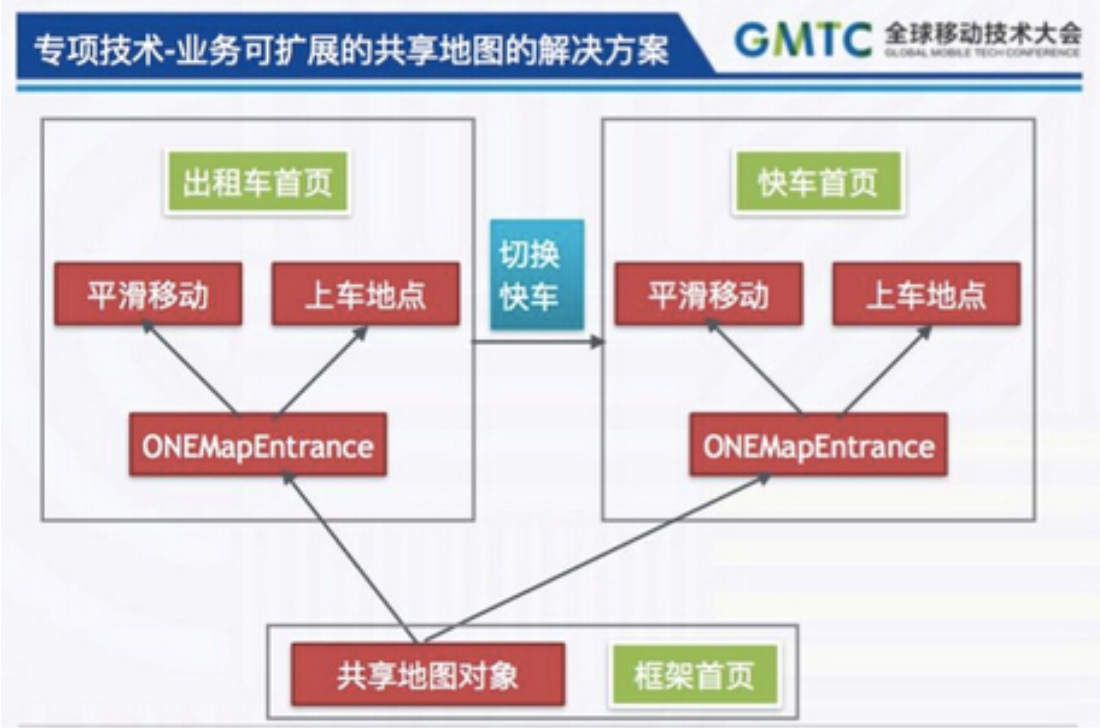
这里要解释的是，在首页怎么打开登录界面。首先，登录界面要在界面导航里注册，然后在 ONERoute 里，建立 URL 和类的对应关系。于是，在首页打开登录界面的时候，会查找对应的类，由 ONERoute 找到对应的对象。

这样做可以达到三个目标。第一，使用便捷。常规的处理需要获取 VC 当前的 Navigation，然后进行 push 或者 present，而在 UIViewController 里面，还需要知道 UIViewController 到底是 push 还是 present 出来的，这两种情况要分别做处理，分别对应 pop 和 dismiss 操作。通过 ONERoute 这样一些组件，只需调用 ONERoute 的一个 pop 动作就够了，非常省心。第二，支持 H5 打开 Native 界面。第三，页面间解耦。在首页，平台只需要知道具体的业务线首页的 URL，而不需要知道具体的类。

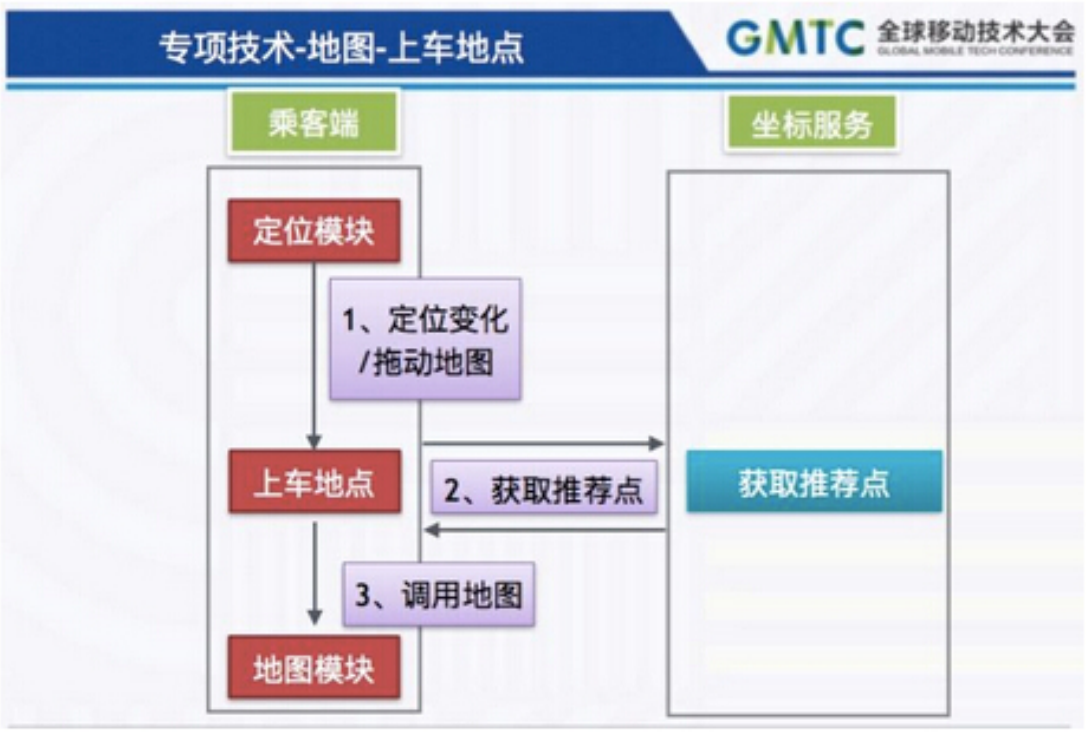
上面讲的内容，好多公司也有对应的技术。而接下来讲的内容，是滴滴独有的技术——地图如何解耦。

第一，滴滴首页只有一份地图，专车、快车互相切换的时候，App 运行非常流畅。第二是内存的考虑，如果是多份地图，内存会相应增加，这里会遇到一个挑战，就是地图只有一个回调。

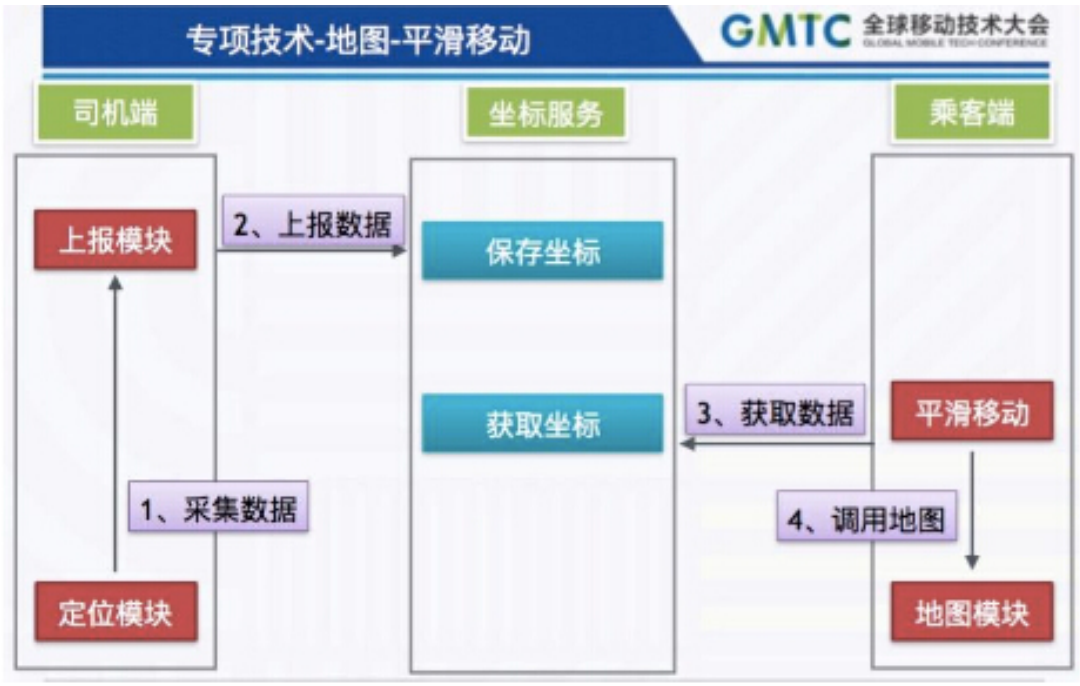
我们是如何应对的？例如，这里显示了一个常规的平滑移动和上车地点，这个时候是在出租车的首页。让共享地图的对象回调到 MapEntrance，MapEntrance 是地图自身的一个封装，在 MapEntrance 里面会回调给平滑移动、上车地点这些模块，当然前提是在出租车首页要分别创建这些模块，并且绑定在 MapEntrance 里，这样在收到回调的时候才可以通知到具体的业务线。切换到快车时，首先切断共享地图对象跟出租车的关系，然后连接到快车的首页，快车也有个 MapEntrance 对象，也有平滑移动和上车地点。



通过这种地图解耦的方案，首先做到了业务隔离。例如，平滑移动和上车地点两个模块是完全隔离的，相互之间完全不知道，这样当编写平滑移动和上车地点代码的时候，代码的难度也比较低。第二是切换清楚。当从出租车切换到快车的时候，会清理掉出租车。这里的 ONEMapEntrance 有点类似于画布，如果不用就全部清理掉。第三是底层更新。上车地点或者平滑移动不知道地图底层是什么样的，切换起来就非常便捷，这些模块根本不需要做



看看大家用得比较多的上车地点这个功能。乘客端拥有定位模块和地图模块，当定位变化或者拖动地图的时候，会触发上车地点的请求，获得当前的推荐点，展示在地图上。这里面还会有很多的技术细节，这里就不展开讲了。



然后，平滑移动涉及司机端和乘客端的交互。司机端会采集当前的定位信息，即经纬度、方向、仰角等信息，并将这些信息上报到坐标服务。当乘客端获取某些司机的时候，会通过 socket 和 Http 两种方式，获取到一系列的坐标，然后把这些点传递给地图模块，最后让小车动起来。

接下来讲一下滴滴在灰度方面的一些实践。滴滴的灰度分三个维度。第一个纬度包括内部员工和外部用户。第二个纬度是根据城市灰度不同，例如北京、上海的滴滴功能方面都有一定的差异。第三个纬度，也是使用比较多的，即百分比灰度，以一定比例放开，在逐步放大过程中观察功能是否正常。

我一般会在三种场景下使用灰度的开关。第一是新功能 3D Touch。第二是功能改版，包括侧边栏改版和登录改版。第三是技术风险模块，例如新日志的模块，经历了若干版本后才最终稳定的。

先看一下侧边栏模块的灰度实践。图中左边是新版的，右边是旧版的。这里有两个风险。第一，产品设计是不是 OK？用户会不会认可？第二，如果用户不认可，怎么办？能不能完全回到旧版的侧边栏？我们的处理方法是，完全保留老的侧边栏，同时写一套新的，它们是两个完全独立的 VC，没有任何共享的东西。然后点击按纽的时候，我们的 apollo 开关会检测到底用新版还是旧版，如果新版有 bug，可以完全返回到旧版。

刚才讲的侧边栏的使用，都是针对有缓存的情况。如果针对网络的无缓存灰度，会遇到更大的挑战。

曾经设计一个流量统计的功能，要去 Hook 底层的东西，但这种有可能导致风险的模块，我们不会缓存它的开关，而是每次去服务器上获取最新的开关，然后再真正打开这个功能，否则会默认不打开。通过这种方式可以完全控制风险。如果模块有问题，服务器就会把这个过程关闭，而你要做的就只是，再重启一次 APP，然后就解决问题了。

第二种是全流程控制，保证开关关闭时，所有修改都不生效。

第三种是及时清理，在灰度完成之后，不再需要旧版的代码，就可以把旧版代码完全清除掉，当然也要考虑它的风险。

问题与思考

前面分享了我们做的一些事情，并总结了一些经验。接下来从四个方面讲解，我们现在面临的问题，以及对这些问题的思考。

第一，瘦身。

滴滴的 App 现在已经超过 100MB 的量级，未来还会有更多的功能和业务线。我们现在只实现了一部分功能，还有很多事情没有做，需要用短期、中期、长期的策略去完成。我们会对 App 进行压缩：处理无用函数；把图片资源缓存到服务器上，只有在使用的时候再下载下来；规定图片格式；还会探索深水区，例如 ProtoBuf 瘦身，在 socket 的时候，整个通信协议用的是 ProtoBuf，能不能简化这些 get/set 的方法，通过动态化的方式，使它不占用那么大的体积；等等。

还会更多使用组件化，并且让组件化更多地推广起来，让不同的业务线使用相同的组件。如果把这些事情做好，整个包体积不会有大量的增长。还有未来长期的计划，让插件动态化。

第二，性能方面的问题就是启动速度。

首先，要分析启动耗时，看函数哪里出了问题，甚至发出一个网络请求，回调到主线的只有会继续卡，也要分析出来。其次，分析出耗时之后，在每个业务线内部进行优化，例如，优化业务线的耗时，优化业务线目前的架构方法，业务线要分析发出的网络请求，并回到主线层的处理。在系统平台的层面，每次只加载一条业务线，如果不是这条业务线的内容，就先不加载。

第三，降低崩溃。

第一是要监控、要持续关注。我们内部有一套自己的系统，定期报警，当发现这些严重崩溃的时候，要用热修复的方法去减少崩溃，降低对用户的影响。第二是分类。我们会根据这些符号表和崩溃栈，将崩溃分散到不同模块，找到对应的负责人来跟进。第三是定期总结，每周总结一次，团队内会定期分享。

最后，模块优化。

我们现在发现，业务线与所有的 SDK 都在一起，只要更新了 SDK，大家会感觉是件很重大的事情，需要做适配。所以，这里要做两件事情，第一是下沉，把业务依赖的内容下沉，第二是上升，对业务没有影响的模块要求快。