31 | 了解移动App的持续交付生命周期

2018-09-13 王潇俊

31 | 了解移动App的持续交付生命周期
朗读人：王潇俊　14′21″ | 6.58M

你好，我是王潇俊。今天我和你分享的主题是：了解移动 App 的持续交付生命周期。

我已经和你分享完的前 30 个主题里，介绍的都是偏向后端持续交付体系的内容。在服务端持续交付的基础上，我会再用两篇文章，和你聊聊移动 App 的持续交付。

与后端服务相比，移动 App 的出现和工程方面的发展时间都较短，所以大部分持续交付的流程和方法都借鉴了后端服务的持续交付。但是，移动 App 因为其自身的一些特点，比如版本更新要依赖用户更新客户端的行为等，所以移动 App 的持续交付也呈现了一些独有的特点。

同时，移动 App 的持续交付也存在一些痛点。比如，没有主流的分支模型，甚至产生了 Android 开发团队使用 Gerrit 这样一个独特代码管理平台和分支模型的特例；又比如，移动 App 的编译速度，也随着应用越来越大变得越来越慢；再比如，Apple Store 审核慢、热修复困难等问题。

这样总体来看，移动 App 的持续交付体系的搭建完全可以借鉴服务端的持续交付的经验。然后，再针对移动 App 固有的特点，进行改进和优化。

因此，在这个系列我只会通过三篇文章，和你分享移动 App 持续交付体系的特色内容，而对于共性的内容部分，你可以再次回顾一下我在前面所分享的内容。如果你感觉哪里还不太清楚的话，就给我留言我们一起讨论解决吧。

作为移动 App 的持续交付系列的第一篇文章，我会和你一起聊聊移动 App 交付涉及的问题、其中哪些部分与后端服务不太一样，以及如何解决这些问题打造出一套持续交付体系。

代码及依赖管理

首先，和后端服务一样，移动 App 的持续交付也需要先解决代码管理的问题。

我在专栏的第四篇文章《一切的源头，代码分支策略的选择》中，和你分享了各种代码分支策略（比如，Git Flow、GitHub Flow 和 GitLab Flow 这三种最常用的特性分支开发模型）的思路、形式，以及作用。

对于移动 App 来说，业界流行的做法是采用“分支开发，主干发布”的方式，并且采用交付快车的方式进行持续的版本发布。

关于这种代码管理方式，我会在下一篇文章《细谈移动 App 的持续交付流水线（pipline）》中进行详细介绍。

其次，移动 App 的开发已经走向了组件化，所以也需要处理好依赖管理的问题。

移动端的技术栈往往要比统一技术栈的后端服务更复杂，所以在考虑依赖管理时，我们需要多方位地为多种技术栈做好准备。比如：

针对 Android 系统，业界通常使用 Gradle 处理依赖管理的问题。Gradle 是一个与 Maven 类似的项目构建工具。与 Maven 相比，它最大的优点在于使用了以 Groovy 为基础的 DSL 代替了 Maven 基于 XML 实现的配置脚本，使得构建脚本更简洁和直观。

针对 iOS 系统，我们则会使用 CocoaPods 进行依赖管理。它可以将原先庞大的 iOS 项目拆分成多个子项目，并以二进制文件的形式进行库管理，从而实现对 iOS 的依赖管理。另外，这种管理依赖的方式，还可以提高 iOS 的构建速度。

除了以上两个技术栈外，移动 App 还会涉及到 H5、Hybrid 等静态资源的构建、发布和管理。那么同样的，我们也就需要 Nexus、npm 等构建和依赖管理工具的辅助。

可以说，移动 App 的技术仍旧在快速发展中，与后端服务比较成熟和统一的状态相比，我们还要花费更多的精力去适配和学习新的构建和依赖管理工具。

项目信息管理

项目信息管理主要包括版本信息管理和功能信息管理这两大方面。

对于移动 App 的持续交付来说，我们特别需要维护版本的相关信息，并对每个版本进行管理。

对后端服务来说，它只要做到向前兼容，就可以一直以最新版本的形式进行发布；而且，它的发布相对自主，控制权比较大。

但对移动 App 来说，情况则完全不同了：一方面，它很难保证面面俱到的向前兼容性；另一方面，它的发布控制权也没那么自主，要受到应用商店、渠道市场和用户自主更新等多方面因素的影响。

所以，在移动 App 的持续交付中，我们需要管理好每个版本的相关信息。

另外，为了提高移动 App 的构建和研发效率，我们会把整个项目拆分多个子项目，而主要的拆分依据就是功能模块。也就是说，除了从技术角度来看，移动 App 的持续交付会存在依赖管理的内容外，从项目角度来看，也常常会存在功能依赖和功能集成的需要。所以，为了项目的协调和沟通，我们需要重点管理每个功能的信息。

可见，做好项目信息管理在移动 App 的持续交付中尤为重要，而在后端服务的持续交付中却没那么受重视了，这也是移动 App 的持续交付体系与服务端的一大不同点。以携程或美团点评为例，它们都各自研发了 MCD 或 MCI 平台，以求更好地管理项目信息。

静态代码检查

静态代码检查的内容，就和后端服务比较相似了。为了提高移动端代码的质量，业界也陆续提供了不少的静态代码检查方案。比如：

- Clang Static Analyzer，被 Xcode 集成，但其缺乏代码风格的检查，可配置性也比较差；
- OCLint，其检查规则更多，也更易于被定制；
- Infer，是 Facebook 提供的一款静态检查工具，具有大规模代码扫描效率高、支持增量检查等特点。

我们也可以很方便地把这些静态检查工具集成到移动 App 的持续交付当中去。基本做法，你可以参考我在第 25 篇文章《代码静态检查实践》最后分享的 Sonar 代码静态检查的实例的内容。

构建管理

移动 App 构建管理的大体流程，我们可以借鉴后端服务的做法，即：通过代码变更，触发自动的持续集成。集成过程基本遵循：拉取代码、静态检查、编译构建、自动化测试，以及打包分发的标准过程。

移动 App 和后端服务的持续交付体系，在构建管理上的不同点，主要体现在以下三个方面：

1. 你需要准备 Android 和 iOS 两套构建环境，而且 iOS 的构建环境还需要一套独立的管理方案。因为，iOS 的构建环境，你不能直接使用 Linux 虚拟机处理，而是要采用 Apple 公司的专

用设备。

2. 在整个构建过程中，你还要考虑证书的管理，不同的版本或使用场景需要使用不同的证书。如果证书比较多的话，还会涉及到管理的逻辑问题，很多组织都会选择自行开发证书管理服务。

3. 为了解决组件依赖的问题，你需要特别准备独立的中央组件仓库，并用缓存等机制加快依赖组件下载的速度。其实，这一点会和后端服务比较相像。

发布管理

移动 App 的发布管理，和后端服务相比，相差就比较大了。

首先，移动 App 无法做到强制更新，决定权在终端用户。移动 App 的发布，你所能控制的只是将新版本发布到市场而已，而最终是否更新新版本，使得新版本的功能起效，则完全取决于用户。这与后端服务强制更新的做法完全不同。

其次，移动 App 在正式发布到市场前，会进行时间比较长的内测或公测。这些测试会使用类似 Fabric Beta 或者 TestFlight 这样的 Beta 测试平台，使部分用户优先使用，完成灰度测试；或者在公司内部搭建一个虚拟市场，利用内部资源优先完成内测。而且，这个测试周期往往都比较长，其中也会迭代多个版本。

最后，移动 App 的分发渠道比较多样。还可能会利用一些特殊的渠道进行发布。为了应对不同的渠道的需求，比如标准渠道版本，控制部分内容，一些字样的显示等等。在完成基本的构建和打包之后，还需要做一些额外的配置替换、增删改查的动作。比如，更新渠道配置和说明等。

以上这些因素，就决定了移动 App 与后端服务的发布管理完全不同。关于移动 App 的发布，我会在下一篇文章《细谈移动 App 的持续交付流水线（pipeline）》中进行详细介绍。

运营管理

移动 App 发布之后，还有一件比较重要的事项，那就是对每个版本的运营管理。

这里讲的运营主要是指，追踪、分析和调优这个版本发布的表现和反馈。我们运营时，主要关注的内容包括：奔溃报告、区域分析、用户分析、系统资源消耗、流量消耗、响应时长、包体大小、系统监控，以及预警等。

通常，我们也会对比版本之间的这些运营指标，以判断应用是变好了，还是变坏了。

热修复

后端服务修复 Bug 的方式，一般是：发现 Bug 后，可以立刻再开发一个新版本，然后通过正常的完整发布进行修复。

而移动 App 的发布需要用户安装才能起效，这就决定了它不能采用后端服务修复 Bug 的方式。因为，这会要求用户在很短的时间内重新安装客户端，这样的用户体验相当糟糕。

但是，我们也无法避免 Bug。所以，对移动 App 来说，我们就要通过特定的热修复技术，做到在用户不重新安装客户端的前提下，就可以修复 Bug。这也就是我所说的热修复。

关于热修复，比如Android 系统，主要的方式就是以下两步：

1. 下发补丁（内含修复好的 class）到用户手机；
2. App 通过类加载器，调用补丁中的类。

其实现原理，主要是利用了 Android 的类加载机制，即从 DexPathList 对象的 Element 数组中获取对应的类进行加载，而获取的方式则是遍历。也就是说，我们只需要把修复的类放置在这个 Element 数组的第一位就可以保证加载到新的类了，而此时有 Bug 的类其实还是存在的，只是不会被加载到而已。

当然技术发展到今天，我们已经无需重复造轮子了，完全可以利用一些大厂开放的方案和平台完成热修复。比如，百川的 hHotFix、美团 Robust、手机 QQ 空间、微信 Tinker，都是很好的方案。

iOS 系统方面，Apple 公司一直对热修复抓的比较严。但是，从 iOS7 之后，iOS 系统引入了 JavaScriptCore，这样就可以在 Objective-C 和 JavaScript 之间传递值或对象了，从而使得创建混合对象成为了可能。因此，业界产生了一些成熟的热修复方案。比如：

1. Rollout.io、JSPatch、DynamicCocoa
   这三个方案，只针对 iOS 的热更新。目前，Rollout.io 和 JSPatch 已经实现了平台化，脚本语言用的都是 JavaScript。Rollout.io 除了支持 OC 的热更新外，还支持 Swift。
   DynamiCocoa 源自滴滴，目前还没开源，所以我也没怎么体验过。但是，它号称可以通过 OC 编码，自动转换成 JavaScript 脚本，这对编码来说好处多多。
2. React Native、Weex
   这两个方案，都是跨平台的热更新方案。其中，React Native 是由 Facebook 开发的， Weex 是由阿里开发的。就我个人的体验来说，Weex 从语法上更贴近编程思路，而且还实现了平台化，使用起来更加便捷。
3. Wax 、Hybrid
   这两个方案，比较特殊。其中，Wax 采用的脚本语言是 Lua 而不是 JavaScript，所以比较适用于游戏；而 Hybrid 主要面向 H5，Hybrid App 已经被证明不是好的方案，所以用户越来越少了。

总结

今天我主要和你分享了移动 App 的持续交付生命周期的几个主要部分，包括代码及依赖管理、项目信息管理、静态代码检查、构建管理、发布管理、运营管理，以及热修复。

然后，我分享了相比于后端服务，移动 App 的持续交付体系有哪些不同的地方。比如，项目信息管理、运营管理和热修复，在移动 App 的交付过程中被提到了更重要的位置；而其他几个主要过程，代码、构建、发布这三部分都因为移动 App 开发的特性与后端服务相比有所区别，这些区别也是我要在下一篇文章中和你重点分享的内容。

思考题

对于移动 App 的交付来说，版本和信息管理非常重要。你所在的公司是如何管理这些信息的，有哪些可以优化的可能吗？

感谢你的收听，欢迎你给我留言。

版权归极客邦科技所有，未经许可不得转载

精选留言

九脉一谷 0

公司内部搭建了一套移动app管理平台，所有的版本发布，行为分析，接口动态配置，参数配置等等都实现了平台化管理。

2018-09-13