前端之巅

GMTC 全球大前端技术大会

主办方 Geekbang 极客邦科技 InfoQ

GMTC 全球大前端技术大会是由极客邦科技旗下 InfoQ 中国主办的技术盛会，关注前端、移动、AI 应用等多个技术领域，促进全球技术交流，推动国内技术升级。GMTC 为期 4 天，包括两天的会议和两天的培训课，主要面向各行业前端、移动开发、AI 技术感兴趣的中高端技术人员，大会聚焦前沿技术及实践经验，旨在帮助参会者了解大前端 & 移动开发领域的技术趋势与实践案例。

卷首语：2019，如何放大大前端的业务价值？

作者 狼叔

2019 年 GMTC 首次南下深圳，依照往届 GMTC 惯例，每届 GMTC 我们都会出一本迷你书，这次也不例外。这届，我们精心挑选了几个不同方向的文章，希望能为开发者指点迷津。

今年的大前端增速放缓，没有出现很多颠覆性的技术，反而在细分领域厮杀的非常厉害，我想这是好事，意味着前端正在走向成熟。这点，从框架、语言，甚至是前后端分工等都有提效。比如 Flutter，跨端能力进一步增强，通过 2d 引擎上画组件，可以带来更好的灵活性。比如小程序领域，不断的涌现出各种转译实现，Wepy、Taro 等， ReactReconciler 出现之后，出现了 Remax 框架。通过 Remax 把生成的「虚拟 DOM」渲染到视图层，从而做到了使用真正的 React 去构建小程序。比如 React，能讲的新特性并不多，在 Create-React-App 火爆之后，类似的支付宝的 Umi 框架也正在悄然兴起。Umi 是更具有阿里特色的解决方案，它集成了 Antd、dva 等成熟模块，也支持 JS 和 TS，在今年年中还增加服务器端渲染 SSR 相关特性。尤其值得一提的是 Umi UI，在可视化辅助编程领域可谓一个新的突破。但无论怎么看，这些都不算是颠覆性的变革，而是在深度上更精进一步。

在 Node.js 领域，今年新东西也不多，最新已经发布到 v13，lts 是 v12，Egg.js 的生态持续完善，进度也不如前 2 年，成熟之后创新就少了。在很多框架上加入 TS 似乎已经大致正确了。比如自身是基于 TS 的 Nest 框架，比如阿里也开源了基于 Egg 生态的 Midway 框架，整体加入 TS，类型系统和 OOP，对大规模编程来说是非常好的。另外，GraphQL 也有很强的应用落地场景，尤其是 Apollo 项目带来的改变最大，极大的降低了落地成本。

已经用 Rust 重写的 Deno 稳步进展中，没有火起来，但也有很高的关注度，它不会替代 Node.js，而是基于 Node 之上更好的尝试。

你可能会感觉 Node.js 热度不够，但事实很多做 Node.js 的人已经投身到研发模式升级上了。对于今天的 Node.js 来说，会用很容易，但用好很难，比如高可用，性能调优，还是非常有挑战的。我们可以假想一下，流量打网关，网关根据流量来实例化容器，加载 FaaS 运行时环境，然后执行对应函数提供服务。在整个过程中，不许关心服务器和运维工作，不用担心高可用问题，是不是前端可以更加轻松的接入 Node.js。这其实就是当前大厂在前端做的基于 Serverless 的实践，比如基于 FaaS 如何做服务编排、页面渲染、网关等。接入 Serverless 不是目的，目的是让前端能够借助 Serverless 创造更多业务价值。

前端技术趋于成熟，不可否认，这依然是大前端最好的时代，但对前端来说更重要的是证明自己，不是资源，而是可以创造更多的业务价值。在垂直领域深耕可以让大家有更多生存空间，但我更愿意认为 Serverless 可以带来前端研发模式上的颠覆，只有简化前后端开发的难度，才能更好的放大前端的业务价值。最后，引用狼叔常说的一句话送给大家：“少抱怨，多思考，未来更美好。”

蚂蚁财富 Flutter 工程化实践

作者 冉叶兰

Flutter 是 Google 开发的一套全新的跨平台、开源 UI 框架，以便捷快速的开发体验、灵活优雅的布局方式和媲美原生的性能备受好评。但是在 Flutter 业务生态内的落地以及多团队间的协作方面，目前还没有很好的实践。就此 InfoQ 采访了来自阿里巴巴的肖凯老师，希望给正在实践 Flutter 的你一些经验和启发。

Flutter 是 Google 开发的一套全新的跨平台、开源 UI 框架，目前已支持 iOS、Android 和 Web 应用的开发，对于 Desktop 的支持也在进行中。Flutter 以其便捷快速的开发体验，灵活优雅的布局方式和媲美原生的性能为移动开发带来了新的活力。

大多数业务复杂的团队，在 Flutter 业务生态内的落地以及多团队间的协作方面，目前还没有很好的实践。对于 Flutter 的使用，闲鱼团队在业界已经有了不错的实践。

GMTC：首先请介绍下您和所在团队的主要工作。

肖凯：我是蚂蚁金服的一名无线技术专家，负责蚂蚁财富端上的开发工作。先前在网易有道负责有道翻译官的开发工作，2017 年加入蚂蚁金服。目前上线的一些作品有蚂蚁星愿、财富王者和 18 财富日等。另外可以去 AppStore 搜索蚂蚁财富来体验一下，里面包含了我们的 Flutter 实践。

我所在的平台与基础小组，主要负责蚂蚁财富 App 的开发工作，技术栈覆盖了 iOS、Android、H5、Flutter 和 Node.js 等。内部有很多黑科技产出，由于领域特性大部分内容没法对外公开，前期开源过安卓构建系统 freeline：https://github.com/

alibaba/freeline

平台负责人 bang 的个人博客也公开过一部分实践：https://blog.cnbang.net/

GMTC：您先前就职于网易有道，负责有道翻译官等端应用的开发工作，推进了第一代移动开发组件化建设。当时有什么成就？

肖凯：在有道时面临产品多人员少的情况，战略层需要构建产品矩阵覆盖用户。基于这种状况，我们把组件化建设提到最高优先级，采用“三步走”战略：

- 第一步沉淀基础能力、账户以及社交分享等；
- 第二步业务工作流的构建、CI 流程的搭建、业务如何使用和质量怎么保证；
- 第三步业务共建，丰富生态。

对于快速迭代的小规模团队，这套组件化工程方法并不会即时收益，还会增加开发成本，随着有道移动教育产品线全面启动，各种 App 开始立项时这些工作才开始发光发热。

在当时移动开发的作坊时代，这套组件体系让我们以极少的人手（2 人）击败了来自大厂的竞品百度翻译和腾讯翻译君，算是技术价值正名。

GMTC：Flutter 到底有什么魅力，为什么要用 Flutter ？ Flutter 的优缺点又是什么？

肖凯：跨平台是移动开发不断追求的目标。Web 本身是很理想的方案，有很多业界前辈都曾信心满满的使用 Web 作为主技术栈，但是由于性能等问题最后也都败下阵来。后来又出现了 RN 等端渲染方案，但又引入了兼容性等新问题，某些大厂在实践后发现，使用 RN 类的方案，开发成本从 2 端变成了 3 端，于是又开始去 RN 化……这时候谷歌推出了自己的跨平台 UI 框架 Flutter（官方说法：UI 工具包），思路是从精简浏览器来的，从技术方案上就解决了其他方案无法跨越的性能和兼容等问题。底层方案优秀再加上谷歌在社区的影响力，Flutter 就成了移动开发人员无法忽视的存在。

Flutter 也有自己的缺点：

1. 热修复，国内技术圈对热修复的偏执可能会是业务使用 Flutter 的最大障碍。目前官方有提供安卓上基于 CodePush 的热修复方案，但在 iOS 上还是无法做到，官方的态度也并不支持，对于将热修复视为救命稻草的国内大厂是一个很大的障碍。
2. 语言和生态，Dart 语言由谷歌主导开发，于 2011 年 10 月公开，不算新语言，但对大多数开发者而言还是陌生的。语言对于大多数开发者还是一个不小的障碍，小程序发展如火如荼和它使用 JS 有很大的关系。Dart 的生态在 2019 年也还处于刚起步的阶段，对于开发者吸引力不强。
3. 体积，接入 Flutter 会增加 App

体积：iOS 双架构（arm64 和 armv7）15M 左右，Android 单架构约 7M，包的体积对于大厂而言又是选择的条件之一。

4. 版本迭代，目前 Flutter 还处于快速迭代的时期，官方 Release 很快，经常出现发布 Stable 后很快又推出 Hotfix 版本的情况，这就要求技术团队对于新技术要有很强的判断力。

另外，从商业政策上看，Apple 对 Flutter 的态度不明确。近期出现了 Apple 检测二进制包中含有小程序关键字并拒审的情况，Flutter 技术上对于动态化并无限制。不过谷歌官方最新的 Roadmap 中已经移除了动态化支持的内容，从官方的态度上来看，目前是乐观的，使用 Flutter 的技术团队可以持续关注下。

GMTC：您如何看待 Flutter 与小程序生态的关系？

肖凯：小程序主要是迎合了国内的流量垄断的现状，是一次商业上的创新，加上构建在 JS 的丰富生态之上，很快吸引了大量前端开发者。

Flutter 有很多优秀的设计思想，从商业市场、技术语言以及动态化等限制上都可能会导致国内很多厂商通过小程序的方式来使用 Flutter，参考 Flutter 来构建自己的小程序渲染引擎。

但新旧技术更替并不是一蹴而就的，一段时间内 Flutter 和传统的小程序技术方案还是会和谐共存，在不同的业务场景下发挥各自的技术特性。目前已经出现了 IoT 设备上 Flutter 的渲染方案，技术成本和最终性能都是优于传统渲染方案的。

在国内市场环境下，Flutter 最终可能会被吸纳为小程序生态的一种渲染方案或者被 PK 掉。但在某些小厂的垂直 App 和即将出现的小终端上，Flutter 可能会找到自己的完美天堂。

GMTC：蚂蚁财富无线团队在使用 Flutter 时，遇到过什么困难？怎么解决？

肖凯：(1) 混合栈，Flutter 官方推荐的标准使用方式是作为独立应用的 U 框架。对于流量垄断的国内市场，我们面临更多的是混合应用场景，最终我们使用了闲鱼的 FlutterBoost 混合栈方案取得了不错的效果 : https://github.com/alibaba/flutter_boost

(2) 路由，在成熟 App 中接入 Flutter 对于路由方案会有很强的入侵，最后我们通过 Flutter 反向代理路由的方式实现了无侵入的 Flutter 路由方案。

(3) 内存，Flutter 在 iOS 上内存表现并不是很好，最终我们通过复用 engine，修复 engine 自身内存泄露的方式解决了。

(4) 工作流，蚂蚁内部业务团队庞杂，多业务方合作是常态，抽象了工作流 CLI，达到了在 Flutter 上业务隔离，协同开发的目的。

(5) 构建，Flutter 自身的构建和传

统端开发格格不入，最终 iOS 上选择了 BuildPhase，Android 上 Hook 了 Gradle 的构建过程与已有构建系统进行了集成。

(6) 业务埋点，Flutter 生命周期和原生应用有些区别导致某些业务埋点表现与 native 异常，最后通过定制生命周期的方式与原生应用保持了协同。

还有其他很多细小问题，比较通用我就不一一说了。

GMTC：Flutter 未来会有怎样的发展趋势？蚂蚁无线团队下一步计划怎么走？

肖凯：考虑到谷歌未来要推出的 Fuchsia，且 Flutter 是该平台上的标准 UIKit，如果 Fuchsia 能获得商业上的成功，Flutter 无疑会站上移动开发舞台的中央。

Flutter 社区目前都很活跃，生态方面不用过于担心。从技术人员的角度来看大胆拥抱吧。

在未来 Flutter 除了和传统移动开发技术在手机端持续竞争，还会在新兴小终端应用上深入探索，并且谷歌对 Flutter 的期望就是打造全平台 UI 工具包，随着对 Flutter 挖掘的深入和 5G 时代的来临，可能小终端会成为移动开发新的战场，Flutter 会带来强大的活力。

Flutter 工程化方面我们已经取得了不错的积累，之后计划：

- 进一步拓展能力，可能的话会输出到社区；
- 不仅仅局限在前端，后端和其他终端我们也会继续探索；
- Flutter 在国内还处于推广阶段，关注多，使用少，后面我们将会和社区的交互放在高优先级，通过技术交流和输出证明其在生产环境中的价值，为 Flutter 在国内的推广尽一份力。

美团外卖 Flutter 容器化生态建设实践

作者 冉叶兰

如何支撑整个外卖链路的高可用性？美团外卖的 Flutter 容器化生态建设实践告诉你，其实 Flutter 可以这样用。

Flutter 作为革命性的跨终端解决方案，刚推出就被相关从业者广泛关注。美团外卖终端团队为了更好地平衡团队的开发效率与稳定性，2018 年着手调研并引入 Flutter，先后在用户端和商家端验证可行性，确认了其在中后台业务场景中具有实用价值，且目前已经在美团点评多个部门不同类型中后台业务大规模线上使用，提升了研发效率，在这期间美团外卖团队收获了很多实践经验。

GMTC：请您简单介绍下自己以及团队所负责的工作。

陈航：我是美团外卖终端团队商家业务的技术负责人，也是极客时间《Flutter 核心技术与实战》的作者。我 2015 年加入美团，经历了外卖用户端和商家端发展的多个阶段，期间推动了外卖移动端架构演进、线上运维及终端技术栈融合等相关工作。

我们团队的主要职责是为商家提供稳定可靠的生产经营工具，支撑整个外卖链路的高可用性，保持其稳定发展。团队技术栈整体比较丰富，也相对较新，覆盖了 Android、iOS、React、Flutter、React Native 和 NW 等。

GMTC：您加入美团后，经历了外卖用户端和商家端发展的多个阶段，哪个阶段您的印象最深，为什么？

陈航：不同阶段背后的思考和决策过程可能印象会更深一些。从 2015 年到现在，外卖市场整体发展还是非常快的。在行业高度竞争、业务快速发展的背景下，美团外卖不仅是一个场景化交易业务闭环，更是一个流量平台。所以对于终端团

队而言，如何推进各业务节点复用，在保证核心主业务稳定快速发展的同时，输出成熟的标准化技术平台支持新业务，是我们关注的重点。

在这个方向下，我们把工作目标划分为两个层次和一个支撑体系：

- 第 1 层次：将基础能力及交易链路标准化，以分层形式实现移动端架构升级，推进业务复用；
- 第 2 层次：推进终端技术栈融合及周边工具支撑建设，减少需求交付链路之间的信息和资源损耗；
- 支撑体系：围绕问题的发现、定位和解决，持续建设终端监控、日志和容灾体系，保障业务高可用。

支撑体系和第 1 个层次的工作思路和逻辑及可以参考我们在美团技术博客输出的文章《美团外卖客户端高可用建设体系》《美团外卖Android平台化架构演进实践》《美团外卖 iOS 多端复用的推动、支撑与思考》；而第 2 个层次的工作重心则主要集中在给开发者赋能上，即如何抽象终端能力和研发过程，用尽量完整的技术栈和工具链去隔离各 App、各终端系统及各研发环节差异，向业务开发同学提供统一而标准化的能力，在这其中，跨平台是一个重要的手段。

GMTC：美团外卖终端团队在 2018 年着手调研并引入 Flutter ，为什么在短时间就确定引入？现在进展如何？

陈航：为了解决需求交付链路的损耗，我们一直在持续跟进业界前沿的跨平台解决方案，也先后大规模落地了动态模板和 RN 等技术。这些方案解决了外卖业务强运营场景下的研发效率和发布效率问题，但由于其实现机制所限，在可扩展性、性能、质量这几个维度上和原生存在差距，只能根据业务特点做取舍，因此很难扩展到核心高频链路场景，满足我们高性能 / 高稳定 / 无降级的研发诉求。在 2018 年中，我们对跨平台方案的另一典型代表 Flutter 进行了原理和实践上的探索，并先后在用户端和商家端上分别对上述问题进行了可行性验证，最终确认其高保真、高性能的技术特点在商家端的中后台业务场景中具有实用价值，也推进了多个中等核心页面的落地，节约了研发成本，而我们在美团技术博客输出的文章《Flutter 原理与实践》就是这一时期的典型产出。

为了方便公司内部其他类似场景实践应用，我们基于自身的实践经验，建设了 Flutter 容器化工具链 MTFlutter 。作为源自美团外卖，服务于全公司的容器化方案，目前，MTFlutter 已经在美团点评多个部门不同类型中后台业务大规模线上使用。

GMTC：现在很多大厂都应用了 Flutter ，您认为在 Flutter 应用上，有什么共同点？

陈航：任何技术选型都不是孤立的技术问题，最终都要服务于商业目标。移动

互联网经过十多年的发展，市场驱动的红利已经见顶，企业增长进入了领导力驱动和创新驱动的阶段。这两个阶段的典型特征都是通过组织或创新等手段，以更高的效率提供更低成本的服务，达成更高的市场份额，这与终端技术的发展方向也是类似的。以当下来看，在终端技术永恒的三大主题即效率、质量和性能，大家都面临着相似的问题。而 Flutter 的解决方案抽象高度足够高，从原理和运行机制这两个维度有领先同类产品的表现，因此也是业界目前关注度最高的。

对于新技术不盲从，而是充分理解其原理和最佳实践的应用场景，从而清楚它们在自己团队中可以解决什么具体的问题，也同样重要。

Flutter 有很多优点，但其实也存在比较明显的缺点，由于官方不支持发布期动态化，因此与 Native 类似的，在业务增量的包大小表现上还不尽如人意。技术领域没有银弹，技术选型要结合业务场景合理评估。于美团外卖终端团队而言，我们在性能要求不高和有动态化诉求的强运营业务场景中，会优先选择 RN ；而针对性能、稳定性和多端体验一致要求高并且对动态化无特殊要求的中后台业务场景，我们则优先选择 Flutter。

GMTC：美团外卖 Flutter 容器化生态的研发背景是？

陈航：虽然 Flutter 提供了开发框架、IDE 插件、调试器等相对丰富的开发工具，但从企业级应用迭代工程角度看，还需要对接大量现有移动端基础设施和工具（如大团队协作、CI、原生组件能力、发布运维基建等）才能够支撑起企业级的业务持续交付和线上运维。为了打通 Flutter 标准工作流与美团点评原生技术栈的融合，我们将 Flutter 的研发阶段（初始化、开发 / 调试、构建、测试、发布、集成）进行了抽象，在工作模式层面，将作为原生工程上游的 Flutter 模块开发，抽象为原生依赖产物的工程管理，并提炼出对应的工作流，以可重复、配置化的方式对各个阶段进行统一管理。在这个过程之中，我们联合了公司多个基建团队，对其开发生态做了支持定制，提供了像工程模板、脚手架、基础能力插件、视觉稿代码自动生成、混合开发集成工具、Pub 服务、打包构建、资源托管、监控运维等开发者能力。

GMTC：在美团外卖 Flutter 容器化生态建设中遇到的最大的挑战是什么？怎么解决的？

陈航：前期主要还是在现有终端技术体系之下，确定 Flutter 的技术定位和其他跨平台方案的分工服务边界。在这一时期，我们做了大半年的技术调研和技术储备，并通过大量灰度测试和验证，从原理和实践角度确定了业务在做技术选型的标准，也最终明确了 Flutter 容器化的建设思路和方向。

在真正落地的时候碰到的最多还是一些兼容适配性适配问题。Flutter 官方 SDK 所依赖的工具链及周边基础设施相对较新，而我们实际业务为了兼容低端机和旧操作系统，开发生态往往比较陈旧，在对接公司内的终端基础设施时，遇到了很多兼容问题，比如低端机 CPU 架构、Java 8 语法特性兼容、Gradle API 适配、个别机型引擎加载异常、崩溃等问题。我们内部 Fork 了一套 Flutter SDK ，将这些问题一一修复和完善，并通过一套编译脚本，实现了从源码编译到美团云上传的自动化流程，最终打包成 MTFlutter SDK ，而用户在实际使用时，使用 MTFlutter 脚手架提供的命令，即可一键部署。

GMTC：未来，美团外卖 Flutter 容器化生态建设布局规划上会有新变化吗，下一步规划是什么？

陈航：核心工作思路不会发生变化。我们仍然会在推进业务复用，以及如何减少需求交付链路之间的信息和资源损耗上重点发力。

在业务复用上，我们会在业务组件库、页面模板、开发一体化解决方案上持续投入，提升易用性，从而降低使用成本；而在减少需求交付链路之间的信息和资源损耗上，我们不会孤立地考虑 Flutter 生态，而会更多地思考如何与其他的终端技术融合，比如通过编译期或运行期代码转换的方式去统一上层 DSL ，让开发者可以按需选择底层渲染技术，从而节省技术栈切换所带来的资源损耗。

百度 App 移动端工程能力演进

作者 冉叶兰

近几年，前端工程化越来越多地被提及。到底什么是前端工程化？大前端工程化中的“大”我们又该如何去理解？我们该怎样去搭建一个适合自身的前端工程化工具？在搭建前端工程化工具中会遇到哪些问题？又该如何解决？

InfoQ 在会前采访了郭金老师，带大家了解百度“大前端工程化”的打开方式，看看百度如何落地 Tekes 研发一体化平台，以及百度 App 的架构与工程能力。百度又是如何通过研发流程一体化平台，实现并行开发、快速迭代和高效复用的?

GMTC：自从有前端工程化这个概念，很快就出现了“大前端工程化”，您是怎么看待“大”前端工程化这个概念呢？

郭金：工程化目标是提升开发效率，保障应用质量。“大”前端工程化是指移动端、前端在项目规模、工程复杂度、快速迭代等相同背景下，对一些共性问题的思考。比如，如何保障多人协同并行开发效率、如何控制质量影响范围、如何进行依赖管理 (包管理)、如何规范研发流程以及在快速迭代下如何控制劣化等。

总的来说，前端、移动端在容器化、组件化 (模块化)、依赖管理 (包管理)、自动化、流程规范等方面都有很多相似性。

针对这些问题我们会产生一些思考 : 既然面临同样的问题，那能不能用同样的解决方案，甚至同样的技术栈。这时候有一些全栈工程师跳出来，他们游走在各种技术角色之间，充分吸收各端技术优势，让各端技术优势互补。

GMTC：大型 App 需要对工程适度拆分来保障并行开发。在工程拆分方面，有什么指导原则？

郭金：

- 定义架构层级，并规范底层组件

不能反向依赖上层组件，越往下层对组件接口稳定性、依赖的合理性以及质量要求就要更高；

- 要有一套机制或框架来保障各组件或模块能做到逻辑、资源、数据各有归属，这个框架就是我们常说的广义组件化框架；
- 沉淀一些各业务复用的通用服务组件，比如账号、ABTest 等组件；
- 基础库的体系化建设，基础库的建设主要是避免交叉依赖，有合理的粒度，做到低成本输出；
- 全面的业务组件化，利用组件化框架提供的容器，分发、依赖注入、数据拆分等框架做到逻辑、资源、数据各有归属，逻辑、接口边界清晰，组件必须有明确的功能定位。

做好上面这些，也就达到壳化的目标。最后，需要有防劣化的机制，这也是最难做到的。百度 App 是通过 EasyBox 工具链和 Tekes 平台发布通知系统来做到这一点的。通过 EasyBox 做到组件间的编译隔离，层级反向依赖防控；通过 Tekes 来做依赖、接口劣化防控。

GMTC：大型 App 与中小型 App 的差异点及复杂度在哪里？

郭金：差异主要体现在大型 App 的团队规模、业务规模 (包含非基础库类的接入业务规模) 比较大，迭代速度快、技术形态多以及有多样性的目标。

大型 App 的关注点会比小型 App 多，最基本的要求是保障并行开发，提升研发效率，以及不同组件间相对能做到故障隔离来保障整体 App 的质量；更进一步是多技术形态下基础能力复用、矩阵产品孵化；还有技术组件对外输出 (包含小程序开源)，这些输出很多不是单一组件对外输出，而是成组对外输出，对外输出的组件不仅要适应百度 App 的架构与环境，也要适应不同的宿主架构和环境；另外就是性能、体积因素的约束，问题快速定位能力的要求。复杂度就体现在这样的背景和多样性的目标上。毫不夸张地说，在大型 App 里做架构和工程能力就是要让大象跳舞。

GMTC：组件化与组件化框架，以及依赖管理工具间的关系是什么？

郭金：组件化是一个体系化的工作，需要通过上面讲到的一系列的工作来一步步落实。

广义的组件化框架指一切具有依赖注入、分发、容器化、数据拆分、资源拆分、组间通信性质的非功能性组件。这些组件的主要目的是保障其他组件逻辑、资源、数据各有组件归属，组件化框架是并行开发的基础保障。

关于依赖管理工具，很多人会混淆依赖管理工具和组件化框架，依赖管理工具在一定程度上能保障组件的逻辑边界，也就是控制开放接口，也能协助组件明确依赖。依赖管理工具更多是构建系统的一部分。

GMTC：工程拆分后，目前的收益和成果有哪些？

郭金：可量化部分的数据，首先是我们矩阵产品组件复用率能达到 55%，矩阵产品同步主线的时间也由原来的 5 人周降低到 1 人周。

不可量化的数据，我觉得价值更大。首先提升了研发效率，研发效率的提升主要体现在几个方面：

- 复杂度内敛，不对外暴露复杂度；
- 并行开发效率，组件化具有分发、容器化等性质，很多集中式管理的逻辑、事件会通过组件化框架分发到各目标组件，减少了组间耦合和交互；
- 体现在复用上，工程拆分后，沉淀了大量的服务组件和基础库组件，也就是有很多可用的轮子；
- 在质量上，逻辑各有归属，加上组件化的分发机制，确保各组件间相对隔离，单个组件的故障范围能内敛到组间内部。

最后，拆分的组件是一个编译单元，是启动速度、体积等指标的量化单位，方便快速定位性能、体积等问题。

GMTC：EasyBox 综合解决方案主要包含哪些部分？ EasyBox 的出现是基于什么样的背景？

郭金：EasyBox 综合解决方案包含三部分: 多仓库管理工具 MGIT、二进制管理、构建系统。多仓库部分主要是团队规模变大，通过代码评审权限分离来保障质量，另外也是多产品线仓库粒度源码复用的需求推动的；二进制管理部分是从编译速度和对外输出稳定的发布版本这两个方面考虑的；最后构建系统支持分组件源码 / 二进制切换模式，这既满足了开发的灵活性，也统一了多方合作的开发模式。

GMTC：多仓库管理为什么不使用 Repo ？

郭金：这个主要是从容错和易用性两个方面考虑的。我们强化了同名分支的原则，来保障多仓库间同步。易用性上操作命令基本和 git 对齐，另外对很多输出做了归类整合，让研发同学能像操作单仓库一样操作多仓库，降低上手难度。

GMTC：对 iOS 来讲，构建系统 (依赖管理) 为什么不使用业内知名的 CocoaPods ？

郭金：CocoaPods 是一个很好的依赖管理工具，虽然 Apple 自己出了 SPM，但是仅支持 SWIFT 包管理，CocoaPods 是 iOS 平台事实上的依赖管理标准。其实，在开始做技术选型的时候，是基于 CocoaPods 修改还是自建我们也很纠结。最后几方面考虑，我们决定自建：首先是编译隔离，CocoaPods 组件间的访问相对比较随意；第二，当时我们也参考了 FaceBook 的 Buck 方案，去工程文件化，早期百度 App 工程文件有 5 ~ 6 万行，且可读性较差，去工程文件化大幅降低了工程文件的合并成本；第三，我们要保障架

构有自底向上的输出复用能力，要做架构层级的反向依赖控制；最后，我们要实现分组件源码 / 二进制一键切换的开发模式。基于这几个原因，即使基于 CocoaPods，也需要大改，最后决定自建，这样我们几个子系统间还能较好地做好分工配合。

我们早期上线的时候，因为一些体验问题，很多同学会有“为什么不用 CocoaPods”的疑问，当我们逐步优化后，问这个问题的同学越来越少，大家也越来越认可 EasyBox 才是最适合我们的解决方案。

GMTC：EasyBox 综合解决方案落地后，收益和成果有哪些？

郭金：可量化的部分，首先，我们把原来的工程拆分为多仓库，并且 iOS 做到了去工程文件化，能做到并行上车 (合并到 master)，并行上车率 46%，上车耗时也降到了原来的 1/3，每次迭代大约节省了 20~30 个小时；其次，实现组件源码 / 二进制的切换的开发模式，编译速度提升了 74.8% ~ 85.8%(不同性能的机器从 16 分 12 秒优化到 2 分 18 秒，或 6 分 30 秒到 1 分 38 秒)；最后，不可量化部分，我们升级了构建系统，让矩阵产品可以组件仓库粒度源码复用，实现了 App 生产线的能力，也提供了架构的灵活度，即降低了架构升级成本。

GMTC：百度 App Tekes 研发一体化平台是基于什么样的背景，此方案带来了哪些变化？百度 App Tekes 研发一体化平台在未来有怎样的规划？

郭金：首先是我们实现组件二进制化后，需要经常发布组件的二进制版本，因为本地发布不安全，并且百度 App 也有对外输出稳定二进制版本的诉求，我们考虑能将这一流程自动化，当然一些 ChangeLog 还是需要手工编辑的；其次是基于组件化劣化防控的目标，我们希望有个平台能把各个组件依赖、接口变更、对外文档的情况展示出来，有劣化的话及时通知对应的研发同事优化；最后是希望能把研发流程体系化，比如及时产出性能、体积、单测报表。

这个方案上线后，我们建立了组件二进制版本规范，矩阵产品有了二进制版本复用的基础，也统一了百度 App 的研发和合作团队研发的开发模式，编译成功率也提升了很多。

Tekes 平台的目标还是把整个研发流程体系化，更好的保障研发效率、矩阵产品孵化能力以及研发质量不劣化。

大前端时代下的热修复平台建设

作者 李晓清 程益

随着移动需求的增加、移动项目的拓展，如果移动端应用出现 Bug 不能及时得到修复，影响的不仅仅是用户体验，还会造成业务上的损失，因此，建立一套完整的热修复平台迫在眉睫。基于此，本文作者所在的搜狗商业应用研发团队构建了一套移动热修复服务中间件平台，本文从系统架构到主要流程对解决方案进行了详细的呈现，无论是 iOS、Android、RN、Flutter 都可以借助这一思想来开发自己的热修复平台。

写在前面

移动应用开发与服务端开发有很大不同，服务端应用如果出现问题，可以通过发布新版本修复，或立即回滚到上一个版本，用户能够立刻感知到这一变化；而移动端应用则不同，即使立刻发布新版本修复了问题，也无法保证所有人都能更新到这个版本，如果用户不升级移动端应用，问题依然得不到修复。

此外，众所周知，iOS 发布 App 需要经过 AppStore 的审核，审核的周期几天甚至一周，虽然近来时间有所缩短，但如果想要快速发布新版本依然避免不了审核；而 Android 发布 App 虽然没有审核的过程，但国内多样化的应用市场，依然影响着新版本的发布。

因此，针对移动端应用的实时更新是非常必要的需求。随着移动需求的增加、移动项目的扩展，建立一套完整的热修复平台日益迫切。

热修复概念

2015 年以来，移动开发领域对热修复技术的讨论和分享越来越多，同时也陆续出

现了一些不同的解决方案。业内普遍共识是把不用重新发布新版本，不更新 App 自身安装包，在用户无感知的情况下，就可以对应用当前版本实现 bug 修复、部分功能修改的技术解决方案称为热修复 HotFix。

对比常规的开发流程而言，热修复的开发流程显得更加灵活方便，优势很多：

- 无需重新发版，实时高效修复 bug；
- 用户无感知修复，无需下载新的应用，代价小；
- 修复成功率高，能把损失降到最低。

实现方式

目前的移动项目中既有使用 iOS/Android 原生技术进行开发，也有使用 React Native/Flutter/Weex 等跨平台和 H5 Hybrid 技术进行开发的。H5 的特性使其不用关注热更新的问题，原生开发和跨平台开发虽然都能实现热更新 / 热修复，但技术实现手段不尽相同，因此热修复平台必须能够提供对上述不同技术方案的支持。热修复本质上是动态化、插件化技术的一种形式，可以理解为动态加载一个插件，在不同平台由于系统底层实现的不同，采用了不同的实现方式。

iOS 系统

用于 iOS 原生应用热修复的第三方技术方案主要有 JSPatch 和 waxPatch/LuaView 等。主要技术原理是用脚本语言编写补丁 patch，下发给客户端，在客户端本地通过 Objective-C Runtime 在运行时进行类名 / 方法名反射，替换相应的类和方法实现。

Android 系统

普遍的修复原理都基于 DEX 分包方案，使用了多 DEX 加载的原理，大致的过程就是：把 BUG 方法修复以后，放到一个单独的 DEX 里，插入到 dexElements 数组的最前面，让虚拟机去加载修复完后的方法。在功能上已经支持类、资源的替换和新增，功能非常强大。

跨平台

目前比较主流的跨平台方案就是 React Native、Weex 和 Flutter 了，RN 和 Weex 原理类似都是通过 JavaScript 语言反射成原生语言代码去执行，是使用 JS 脚本语言来编写的，也就是“即读即运行”。我们在“读”之前将之替换成新版本的脚本，运行时执行的便是新的逻辑了。脚本本质上和图片资源一样，都是可以进行热修复的，所以

热修复的原理也比较简单，只要下发相应的 JS 补丁包就可以了。业内比较成熟的方案有微软的 CodePush。

Flutter 由于出现的比较晚，在 Flutter 1.2.1 中，Google 提供了 ResourceUpdater，用来做包的检查和下载解压，可以理解为官方支持的热修复。许多公司为了使用方便也研发了自己的热修复方案。

存在的问题

由于系统的差异性，不同平台有着不同的解决方案，它们的原理各有不同，适用场景各异，到底采用哪种方案，是开发者比较头疼的问题。如针对 iOS 平台的 JSPatch、滴滴 DynamicCocoa、阿里聚划算 LuaView 等，Android 平台的 QQ 空间补丁方案、阿里 AndFix 和 Sophix 以及微信 Tinker 等等，当然也可根据技术实现原理自研。

不管使用哪种热修复技术，我们都需要后台服务的支持，不然就无法实现补丁的分发。由于不同技术团队选用的技术方案也不一样，导致存在以下几个问题：

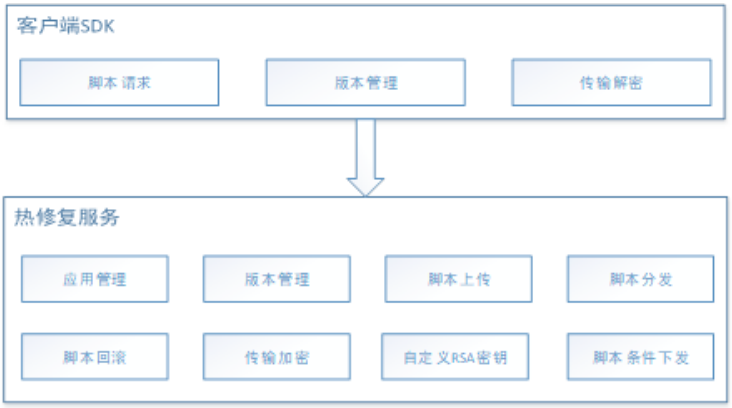
- 针对不同客户端平台需要单独开发不同的补丁上传下载后台；
- 如果更换热修复技术方案，后台也需要做调整；
- 客户端接入逻辑复杂。客户端接入不同的第三方 SDK，需要进行代码适配；
- 缺乏数据监控和统计。修复是否成功无法得到相应的数据反馈；
- 如果管理多个 App，需手动管理版本、渠道等多种复杂工作，增加出错隐患。

解决方案

虽然客户端由于平台的差异性选择的热修复技术不同，但服务端相对而言整体流程和处理策略是统一的，因此，热修复需要一个后台服务来上传、管理和分发修复脚本；同时，也要提供针对不同客户端的 SDK，封装向平台请求脚本、传输解密、版本管理等功能。基于以上几点考虑，我们构建了一套移动热修复服务中间件平台，主要功能包括：

- 对 iOS、Android 原生技术及 React Native 技术开发的移动应用提供热修复服务；
- 提供不同平台的客户端 SDK；
- 提供分发平台，方便复补丁的上传和维护；
- 有补丁的版本控制功能，可以进行更新、回滚操作等；
- 支持补丁文件全量下发和按条件下发；
- 补丁分发时进行加密传输，保证安全性；
- 支持数据统计。

系统架构



热修复服务平台

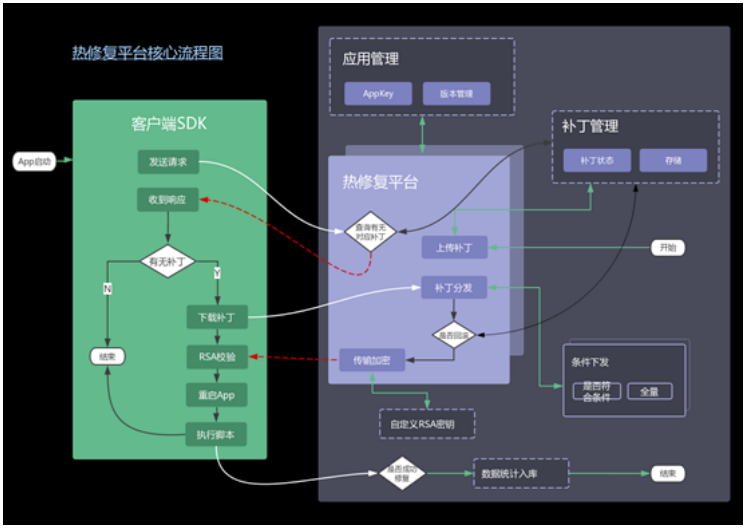
提供应用管理、应用版本和补丁的管理、修复补丁的上传和分发、补丁异常时的回滚、传输过程中的加密传输、自定义加密的 RSA 密钥、按条件下发等功能。

客户端 SDK

提供了向后端请求补丁、补丁的版本管理、传输后的解密等功能。

主要流程

热修复涉及的主要模块是热修复服务平台和客户端 SDK，核心流程如下图所示：



通常情况的热修复流程如下：

1. 首先需要在热修复平台创建应用，设置版本号等信息。一般该操作和应用发布、消息推送等功能一起在移动开发平台统一管理；
2. 添加一个补丁脚本，并选择所属应用名、系统和版本，设置下发条件规则等信息；
3. App 启动同时 SDK 也启动，并发送查询请求给服务端，请求携带 App 标识、App 版本号等参数；
4. 服务端根据收到的请求和条件下发规则判断下发哪个修复脚本。如果该 App 在该版本中存在修复脚本，则返回当前生效的脚本信息和地址；
5. 客户端 SDK 根据接口返回数据判断进行下载、回滚或不进行任何操作；如果未返回任何信息，说明不存在修复脚本需要下发，此时客户端 SDK 不进行任何操作；
6. 客户端 SDK 对下载后的补丁脚本进行 RSA 校验，并执行脚本；
7. 补丁脚本执行成功或者失败等 log 信息上传至后台，以供数据分析使用。

应用管理

热修复是针对 App 的某个版本进行修复，因此发补丁前必须创建相应的应用和版本。应用创建后会分配一个随机的 appKey，客户端 SDK 与服务端交互时必须携带 appKey，用于标识应用。

补丁管理

补丁列表

补丁列表显示历史上发布的补丁信息，可以根据应用名称和版本号进行查询。

大前端时代下的热修复平台建设

相关说明如下：

- 添加新补丁：添加补丁的入口；
- 应用名称：显示应用名称，来自于创建应用时的输入；

- 平台：显示应用所属平台，即 iOS/Android/React Native；
- 应用版本号：应用的版本号，来自于添加应用版本时的输入；
- 补丁版本号：补丁的版本号，该应用在该版本内的顺序递增；
- 补丁大小：补丁文件的大小；
- 补丁描述：来自于添加补丁时的输入；
- 补丁状态：补丁的生效 / 失效状态，同一应用同一版本内，只有一个生效状态的补丁；
- 下发状态：补丁的下发状态，有全量下发和条件下发两种；
- 更新事件：该补丁的最近一次操作事件；
- 显示内容：显示补丁的内容，只能显示 iOS 补丁的内容；
- 全量下发：将补丁的下发状态改为全量下发；
- 条件下发：将补丁的下发状态改为条件下发。

添加补丁

在发布一个修复补丁时，要将其上传至分发平台，并选择所属应用名和版本。上传后，分发平台存储补丁并存储补丁的相关信息后，由于一个版本内可能存在多个修复补丁，因此新上传的补丁会标记为生效，其他的历史补丁标记为失效状态。

存储

服务端由多台机器构成，需要使用统一的文件存储，不能使用本地的文件系统。

方案一：使用 DFS

将补丁文件存储到 DFS 中，在 MySQL 中记录应用、版本，以及 DFS 唯一标识的关系，并提供下载接口，用于 SDK 请求。

方案二：使用 CDN

将补丁文件上传至 CDN，客户端 SDK 下载时直接访问 CDN 的链接。CDN 支持高

并发，且访问速度有保障。

考虑到 CDN 有代码暴露的风险，倾向于选择使用 DFS。

补丁下发

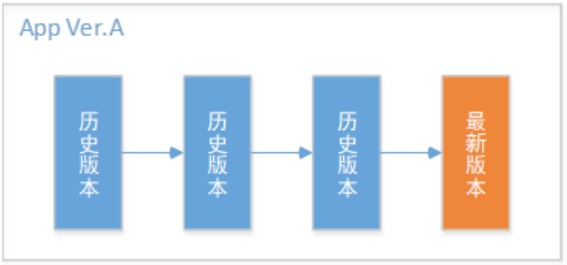
一个 App 可能发布了多个版本，一个版本内又可能发布过多个修复补丁。在向 SDK 下发补丁时，应该下发哪个补丁呢?

下发时遵循如下约定：首先，补丁是基于某个 App 版本的，App 不能跨版本请求补丁；其次，App 一个版本内的多个补丁，只下发最新生效的一个。这是由于 iOS 和 Android 的热修复原理决定的。

iOS 的热修复是通过运行时用补丁中的 JavaScript 代码动态替换 Objective-C 代码实现的，这就造成无法用补丁将 App 的版本整体升级。因此，在发布下一个版本时必须把上一个版本的补丁中的 JS 代码在新版本中用 OC 再实现一遍。例如，当 App 处于版本 A 时某个方法 foo 出现了问题，在补丁 A1 中用 JS 对 foo 方法进行了修复，在发布下一个 App 的版本 B 时，B 中 foo 方法必须用 OC 重写一遍。Android 的热修复是基于基准包，用修复后的新包与基准包 diff 后生成的补丁，客户端再加载补丁实现修复，这也要求在发布下一个 App 版本的时候，必须含有补丁中的内容。

此外，如果在一个版本内下发多个补丁的话，比如版本 A 中，发现了一个 bug，发布了一个修复补丁 A1；之后，发现了另一个 bug，再发布一个补丁 A2。如果 A1 和 A2 的内容彼此无关，那么就要求客户端 SDK 要加载多个补丁文件，当补丁之间存在依赖关系时，更需要控制加载的顺序，这无疑增加了复杂度，而且无法回滚到特定版本；对于 Android 而言，由于加载的补丁是基于基准包 diff 后的包，也做不到加载多个补丁。因此，针对同一版本内有多个补丁的情况，只下发最新的一个生效补丁。

不同 App 版本不能跨版本下发补丁

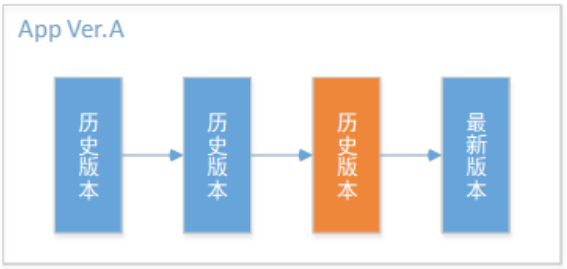


同一个 App 版本内多个补丁时只下发最后生效的版本

回滚

如果所发布的补丁存在问题，这会造成客户端 APP 本身出现异常，甚至应用闪退、完全不可用。针对这种情况，有两种方案，第一是再发布一个新的补丁，补丁中包括修正了的正确代码。另一种情况是，有可能错误难以定位或修正时间太长，根本来不及发布新补丁，那么必须及时将错误补丁回滚。

按照上一节中的下发策略，服务端只会下发当前生效的补丁，因此服务端在回滚的时候只需要简单地将目标补丁标记为生效即可。



传输安全

由于下发的补丁会改变客户端应用的行为，如果被人攻击替换代码，会造成很大危害，因此必须考虑传输过程中的安全性。

针对这一问题，设计了如下 3 个解决方案：

方案一：对称加密

若要让补丁在传输的过程中不会轻易被中间人截获替换，很容易想到的方式就是对补丁进行加密，可以用 zip 的加密压缩，也可以用 AES 等加密算法。

优点：实现非常简单。

缺点：是安全性低，容易被破解。因为密钥是要保存在客户端的，只要客户端被

人拿去反编译，把密码字段找出来，就完成破解了。

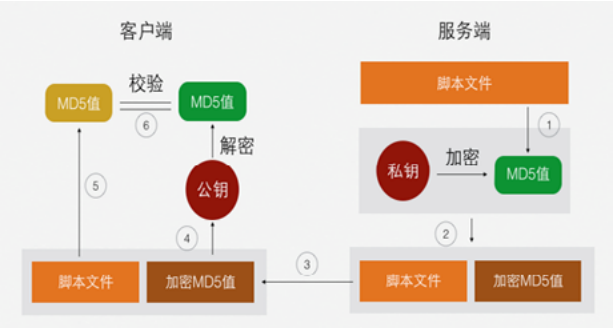
方案二：HTTPS

第二个方案是直接使用 HTTPS 传输。

优点：安全性高，只要使用正确，证书在服务端未泄露，就不会被破解。

缺点：部署麻烦，需要服务器支持 HTTPS，门槛较高。另外客户端需要做好 HTTPS 的证书验证（有些使用者可能会漏掉这个验证，导致安全性大降）。如果服务器本来就支持 HTTPS，使用这种方案也是一种不错的选择。

方案三：RSA 校验



这种方式属于数字签名，用了跟 HTTPS 一样的非对称加密，只是简化了，把非对称加密只用于校验文件，而不解决传输过程中数据内容泄露的问题，而我们的目的只是防止传输过程中数据被篡改，对于数据内容泄露并不是太在意。整个校验过程如下：

- 服务端计算出补丁文件的 MD5 值，作为这个文件的数字签名；
- 服务端通过私钥加密第 1 步算出的 MD5 值，得到一个加密后的 MD5 值；
- 把补丁文件和加密后的 MD5 值一起下发给客户端；
- 客户端拿到加密后的 MD5 值，通过保存在客户端的公钥解密；
- 客户端计算补丁文件的 MD5 值；
- 对比第 4/5 步的两个 MD5 值（分别是客户端和服务端计算出来的 MD5 值），若相等则通过校验。

只要通过校验，就能确保补丁在传输的过程中没有被篡改，因为第三方若要篡改补丁文件，必须计算出新的补丁文件 MD5 并用私钥加密，客户端公钥才能解密出这个 MD5 值，而在服务端未泄露的情况下第三方是拿不到私钥的。

优点：非对称加密能够有效解决传输中被篡改的问题。

缺点：数据内容可能会泄露，其实在传输过程中不泄露，保存在本地同样会泄露，

若对此在意，可以对补丁文件再加一层简单的对称加密。

自定义 RSA 密钥

分发平台使用一套默认的公钥 / 私钥进行补丁传输过程中的加密 / 解密，如果对安全性要求更高，可以在上传补丁时设置自定义的 RSA 私钥。

条件下发

很多时候在发布一个补丁时，需要在小范围内进行验证，比如特定某个 iOS 版本或者特定某个用户；在验证通过后再进行全网用户的下发。这时可以用到条件下发。

分发平台在发布补丁时可以选择使用条件下发，除上传补丁外，还可以填写条件语句，只有满足条件的设备才会执行修复补丁。条件语句由 key/value/ 运算符组成。条件语句的规则与代码中的条件表达式一致，支持“==、!=、>、<、>=、<=、&&、||”等运算符。如：

iOS>9.0 或者 userId == 200758 && role == 1

当补丁的下发状态处于条件下发，且条件语句与 SDK 上报参数中的条件一致时，才会将补丁发送给该 SDK。

计算条件表达式时，如果通过字符串解析和替换的处理等方式，开发繁琐且实现时不够优雅。可以使用 EL 表达式引擎解决这一问题。常见的表达式引擎有 Apache Commons 中的 JEXL（Java Expression Language）、fast-fel 等，甚至 Java 1.6 后自带的 JavaScript 脚本引擎也可以完成这个工作。在综合考量性能和易用性后选择了 JEXL 表达式引擎（测试样例见附录 1）。JEXL 除了支持基本的算术表达式外，也支持在表达式中访问对象的属性、访问数组和集合、调用 Java 方法等特性，对于表达式的使用有很强的扩展性。

下面是一个 JEXL 的例子：

```
JexlEngine jexl = new JexlBuilder().create();

String expression = "userId == 200758 && role.startsWith('manager')";
JexlExpression e = jexl.createExpression(expression);

JexlContext ctx = new MapContext();
ctx.set("userId", 200758);
ctx.set("role", "managerXXX");

Object obj = e.evaluate(ctx);
```

数据统计

提供分日、分 App、分版本的补丁分发数据统计功能。

SDK 设计

iOS/Android 和各种跨平台方案只需实现接口的查询和 patch 包的下载即可，再根据所采用的热修复库实现对应平台的热修复功能。

查询接口

请求参数	/query.do ?appKey=xxx &appVersion=xxx &K1=V1（条件参数） &K2=V2（条件参数） …

返回数据	{
	"success": true,
	"data": {
	"appKey": "xxx",
	"appVersion": "1.0.0",
	"patchVersion": 4
	},
	"errors": []
	}

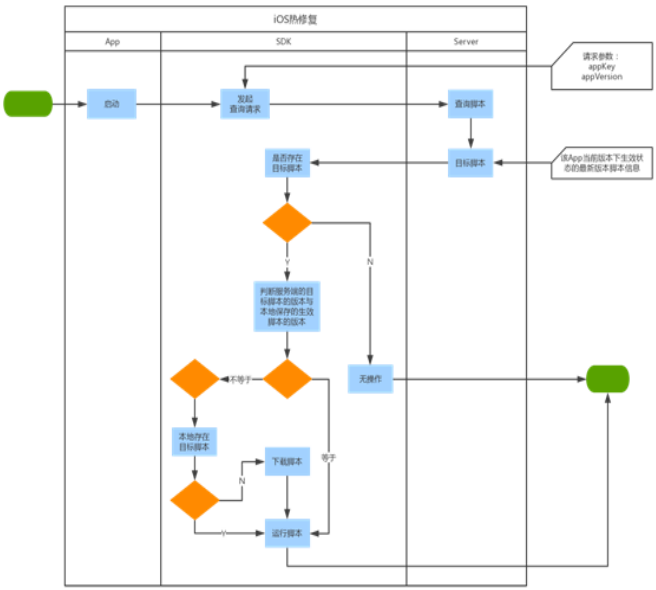
下载接口

请求参数	/download.do ?appKey=xxx &appVersion=xxx &patchVersion=xxx
返回数据	数据流

以 iOS 为例：

客户端 SDK 启动时会发请求询问服务端，根据服务端返回数据进行相应处理。客户端 SDK 会保存下载到的修复脚本，避免重复下载造成的流量损失。具体流程如下。

1. App 启动同时 SDK 也启动，并发送查询请求给服务端，请求携带 App 标识、App 版本号等参数；
2. 服务端根据收到的请求判断下发哪个修复脚本。如果该 App 在该版本中存在修复脚本，则返回当前生效的脚本信息和地址。客户端 SDK 根据接口返回数据



判断进行下载、回滚或不进行任何操作；

3. 如果未返回任何信息，说明不存在修复脚本需要下发，此时客户端 SDK 不进行任何操作；
4. 如果返回的脚本信息中，脚本的版本号等于本地生效脚本的版本，说明客户端保存的脚本已是最新的，SDK 直接执行本地保存的脚本；
5. 如果返回的脚本信息中，脚本的版本号不等于本地生效脚本的版本，说明服务端有新的修复脚本发布，或发生了回滚操作，客户端 SDK 判断本地是否存在该版本的脚本，存在时直接执行本地脚本；不存在时发起下载请求获取脚本，并在本地缓存，然后执行。

争议

2017 年 3 月，众多 iOS 开发者收到苹果警告邮件，称其 App 违规使用动态方法，责令限时整改。这封邮件引起了开发者的恐慌，最后发现问题集中出现在两个热更新工具 Rollout 和 JSPatch 上。由于 JSPatch 在 iOS 业内的高覆盖率，这个事件影响几乎波及到了国内所有在 AppStore 上线的 App。

这次警告事件无疑是对 iOS 平台 Native 动态化是一次严重打击，其影响甚至可能波及到 Android 平台，毕竟 Google 也是禁止加载远程代码的，并且执行更为严格，只是管不到中国的 Android 开发而已。

在安卓平台，虽然谷歌没有能力像苹果一样干涉国内的开发，但插件化技术从另一方面遭遇了困境。这一困境就是安卓新版本以及国内各种魔改 ROM 对于底层的改动。安卓插件化技术依赖部分底层方法以及私有 API，而这些在新版本里是很有可能改动的，一旦修改了，插件化就会失效甚至出错。国内各大手机厂商的系统也喜欢对底层进行修改，它们的修改甚至都不会公开告知，因此兼容问题是插件化技术遇到的最大挑战。

2018 年发布的 Android 9.0，甚至要求开发者不得使用私有 API，少了这些 API，安卓开发被重新关回笼子里，还能玩的黑科技大大减少，无意之中竟然取得了和苹果警告类似的效果。

在苹果“热修复门”事件之后，iOS 动态化的工具都转入地下发展，关于这方面的研究和分享也急剧减少，甚至连整个 iOS 技术的分享也变少了。虽然最初通过代码混淆等可以骗过苹果审核，但是很快也被禁止了，滴滴声称的 DynamicCocoa 也迟迟没有开源，QQ 甚至开发了一个自己的中间语言 OCScript，还开发了一个自己的虚拟机 OCSVM 去执行它。

虽然使用企业证书发布的 App 不受应用市场的监管影响，但是整个行业对于热修复技术的研究和讨论也越来越少了，大家不得不寻找新的技术突破点，这种氛围也间接促进了跨平台技术的推广。尤其是 Flutter 发布之后，相较于以前的 RN 等跨平台技术拥有了更流畅的用户体验，许多大厂也开始积极使用。相信在不久的将来移动开发会形成原生开发与跨平台开发并驾齐驱的态势。

结束语

本文介绍了一种基于第三方或自研的热修复客户端技术，但又不强依赖特定服务的通用热修复中间件管理平台，可以实现安全、稳定、可靠的热修复补丁上传、分发、版本管理等功能，并提供完善的数据统计。在实际应用中，可以结合团队自身技术栈打造成更加通用的热修复管理平台。

开发一个高质量的前端组件，这些一定要知道

作者 郭美青

2009 年 11 月 8 日，在欧洲 JSConf 大会上，Ryan Dahl 第一次正式向业界宣布了 Node.js 的面世，使 JS 语言书写后端应用程序成为了可能。在随后的几年里，Node.js 受到了 JavaScript 社区的狂热追捧，前端行业也因此进入了一个全新的工程化和全栈时代。回顾历史，总会让人心潮澎湃。在这股浪潮中，有无数的人和项目在这座丰碑中刻下了自己的名字：React、Vue、Yeoman、RequireJS、Backbone、Antd、Webpack、Mocha、Eslint 等等。在这些知名项目的熠熠光辉下，我们可能会忽略为 Node.js 生态的繁荣之下建立不世之功的 NPM，它才是当之无愧的肱骨重臣。

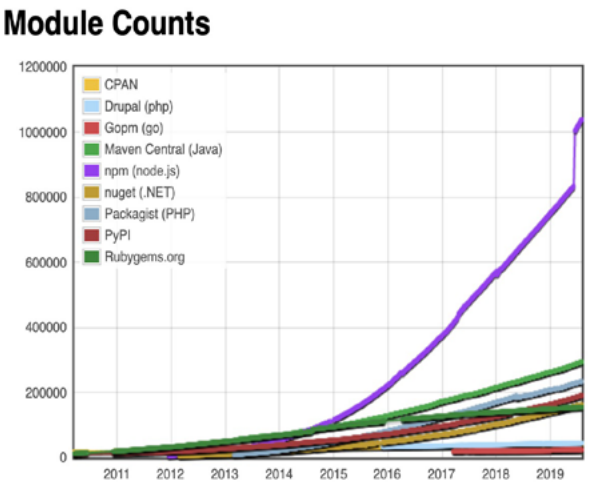
NPM 生于 2010 年 1 月，它从出生就背负了让 Node.js 社区更加繁荣的使命。NPM 致力于让 JS 程序员能够更加方便地发布、分享 Node.js 类库和源码，并且简化模块的安装、更新和卸载的体验。

从今天（2019 年）这个时间节点来看，NPM 无论从知名度、模块数量、社区的话题数量来看，都算得上是一骑绝尘，将其他语言的模块仓库远远甩在了后面。

NPM 的生态既已如此成熟，按说开发者对于 NPM 包的发布和维护应该非常熟悉才是，但事实真的是这样吗？环顾身边的 FE，没有发过任何 NPM 包的同学大有人在，已经发过包的同学也有相当一部分并未考虑过如何才算规范、高质量地发布一个包。

如今 NPM 的模块数量已上升至 100W，在这样一个 JavaScript 组件化开发时代，除了能找到好用的组件，我们自然也需要了解如何才能成为创造这个时代的一员。而第一步就是要知道并掌握如何规范地、负责任地发布一个 NPM 包。

这就是本文接下来的主要内容。



数据来源 : moudlecounts

1. 组件化思考

发布人生中第一个 NPM 组件虽然只是在终端命令行中潇洒地敲下 npm publish，静等成功通知即可，但这从 0 到 1 的跨越却并非易事。这个行为背后的始作俑者是开发者大脑中开始萌发组件化思维方式，开始去思考何为组件？为什么要发布组件？这些更深层次的问题。

组件的存在的终极意义是为了复用，一个组件只要具备了被复用的条件，并且开始被复用，那么它的价值才开始产生。组件复用的次数越高、被传播的越广，其价值就越大。而要实现组件的价值最大化，需要考虑以下几点：

1. 我要写一个什么组件？组件提供什么样的能力？
2. 组件的适用范围是什么？某个具体业务系统内还是整个团队、公司或者社区？
3. 组件的生产过程是否规范、健壮和值得信赖？
4. 组件如何被开发者发现和认识？

以上四点中，前两点是生产组件必须要思考的问题；第四点是组件如何推广运营的问题，这是另外一个话题，本文不展开探讨；第三点是开发者的基本素养，它决定了开发者如何看待这个组件，也间接暴露了开发者的素养和可信赖程度。

2. 组件开发的最佳姿势

一个优秀的组件除了拥有解决问题的价值，还应该具备以下三个特点：

1. 生产和交付的规范性；

2. 优秀的质量和可靠性；

3. 较高的可用性。

只有三者都能满足才可以称其为优秀组件，否则会给使用者带来各种各样的困惑：经常出 Bug、坑很多、不稳定、文档太简单、不敢用等等。

2.1 规范性

2.1.1 目录结构

事实上，社区并没有一个官方的或者所有人都认同的目录结构规范，但从耳熟能详的知名项目中进行统计和分析，可以得出一个社区优秀开发者达成非官方共识的一个目录结构清单：

```
├── test //测试相关
├── scripts //自定义的脚本
├── docs //文档，通常文档较多，有多个md文档
├── examples //可以运行的示例代码
├── packages //要发布的npm包，一般用在一个仓库要发多个npm包的场景
├── dist|build //代码分发的目录
├── src|lib //源码目录
├── bin //命令行脚本入口文件
├── website|site //官方网站相关代码，譬如antd、react
├── benchmarks //性能测试相关
├── types|typings// typescript的类型文件
├── Readme.md //仓库介绍或者组件文档
└── index.js //入口文件
```

以上目录清单是一个比较完整的清单，大多数组件只需根据自己的需求选择性地使用一部分即可。一份几乎适用于所有组件的最小目录结构清单如下：

```
├── test //测试相关
├── src|lib //源码目录
├── Readme.md //仓库介绍或者组件文档
└── index.js //入口文件
```

2.1.2 配置文件

这里的配置文件主要指的是各种工程化工具所依赖的本地化的配置文件，以及在 GitHub 上开源所需要声明的一些文件。一份比较全的配置文件清单如下：

```
├── .circleci          //目录。circleci持续集成相关文件
├── .github            //目录。github扩展配置文件存放目录
|   ├── CONTRIBUTING.md
|   └── ...
```

```
├── .babelrc.js          // babel编译配置
├── .editorconfig        //跨编辑器的代码风格统一
├── .eslintignore        //忽略eslint检测的文件清单
├── .eslintrc.js         // eslint配置
├── .gitignore           // git忽略清单
├── .npmignore           // npm忽略清单
├── .travis.yml          // travis持续集成配置文件
├── .npmrc               // npm配置文件
├── .prettierrc.json     // prettier代码美化插件的配置
├── .gitpod.yml          // gitpod云端IDE的配置文件
├── .codecov.yml         // codecov测试覆盖率配置文件
├── LICENSE //开源协议声明
├── CODE_OF_CONDUCT.md   //贡献者行为准则
└── ... //其他更多配置
```

以上配置可以根据组件的实际情况、适用范围来进行删减。一份在各种场景都比较通用的清单如下：

```
├── .babelrc.js          // babel编译配置
├── .editorconfig        //跨编辑器的代码风格统一
├── .eslintignore        //忽略eslint检测的文件清单
├── .eslintrc.js         // eslint配置
├── .gitignore           // git忽略清单
├── .npmignore           // npm忽略清单
├── LICENSE //开源协议声明
└── ... //其他更多配置
```

上述清单移除了只有在 GitHub 上才用得到的配置，只关注仓库管理、发包管理、静态检查和编译这些基础性的配置，适用于团队内部、企业私有环境的组件开发。如果要在 GitHub 上维护，则还需要从大清单中继续挑选更多的基础配置，以便可以使用 GitHub 的众多强大功能。

2.1.3 package.json

如果说 NPM 官方给出了一个发包规范的话，那么这个规范就是 package.json 文件，这是发包时唯一不可或缺的文件。一个最精简的 package.json 文件是执行 npm init 生成的这个版本：

```
{
  "name": "npm-speci-test", //组件名
  "version": "1.0.0", //组件当前版本
  "description": "", //组件的一句话描述
  "main": "index.js", //组件的入口文件
  "scripts": { //工程化脚本，使用npm run xx来执行
    "test": "echo \"Error: no test specified\" && exit 1"
```

```
  },
  "author": "", //组件的作者
  "license": "ISC"          //组件的协议
}
```

有这样一个版本的 package.json 文件，我们就可以直接在该目录下直接执行 npm publish 发布操作了，如果 name 的名称在 npm 仓库中尚未被占用的话，就可以看到发包成功的反馈了：

```
$ npm publish
+ npm-speci-test@1.0.0
```

但光有这些基础信息肯定是不够的，作为一个规范的组件，我们还需要考虑：

1. 我的代码托管在什么位置了；
2. 别人可以在仓库里通过哪些关键词找到组件；
3. 组件的运行依赖有哪些；
4. 组件的开发依赖有哪些；
5. 如果是命令行工具，入口文件是哪个；
6. 组件支持哪些 node 版本、操作系统等。

一份比较通用的 package.json 文件内容如下：

```
{
  "name": "@scope/xxxx",
  "version": "1.0.0",
  "description": "description:xxx",
  "keywords": "keyword1, keyword2,...",
  "main": "./dist/index.js",
  "bin": {},
  "scripts": {
    "lint": "eslint --ext ./src/",
    "test": "npm run lint & istanbul cover _mocha -- test/ --no-timeouts",
    "build": "npm run lint & npm run test & gulp"
  },
  "repository": {
    "type": "git",
    "url": "http://github.com/xxx.git"
  },
  "author": {
      "name": "someone",
      "email": "someone@gmail.com",
      "url": "http://someone.com"
  },
  "license": "MIT",
```

```
  "dependencies": {},
  "devDependencies": {
    "eslint": "^5.2.0",
    "eslint-plugin-babel": "^5.1.0",
    "gulp": "^3.9.1",
    "gulp-rimraf": "^0.2.0",
    "istanbul": "^0.4.5",
    "mocha": "^5.2.0"
  },
  "engines": {
    "node": ">=8.0"
  }
}
```

- name 属性要考虑的是组件是否为 public 还是 private，如果是 public，要先确认该名称是否已经被占用，如果没有占用，为了稳妥起见，可以先发一个空白的版本；如果是 private，则需要加上 @scope 前缀，同样也需要确认名称是否已被占用。
- version 属性必须要符合 semver 规范，简单理解就是：
  - 第一个版本一般建议用 1.0.0；
  - 如果当前版本有破坏性变更，无法向前兼容，则考虑升第一位；
  - 如果有新特性、新接口，但可以向前兼容，则考虑升第二位；
  - 如果只是 bug 修复，文档修改等不影响兼容性的变更，则考虑升第三位。
- keywords 会影响在仓库中进行检索的结果。
- main 入口文件的位置最好可以固定下来，如果组件需要构建，建议统一设置为 ./dist/index.js，如果不需要构建，可以指定为根目录下的 index.js。
- scriptsscripts 通常会包含两部分：通用脚本和自定义脚本。无论是个人还是团队，都应该为通用脚本建立规范，避免过于随意的命名 scripts；自定义脚本则可以灵活定制，比如：
  - 通用 scripts：start、lint、test、build；
  - 自定义 scripts：copy、clean、doc 等。
- repository 属性无论在私有环境还是公共环境，都应该加上，以便通过组件可以定位到源码仓库。
- author 如果是一个人负责的组件，用 author，多个人就用 contributors。

更详细的 package.json 文件规范可以参见 npm-package.json。

2.1.4 开发流程

很多同学在开发组件时都会使用 master 分支直接进行开发，觉得差不多可以发版

了就直接手动执行一下 npm publish，然后下一个版本，继续在 master 上搞。

这样做是非常不规范的，会存在很多问题，譬如：

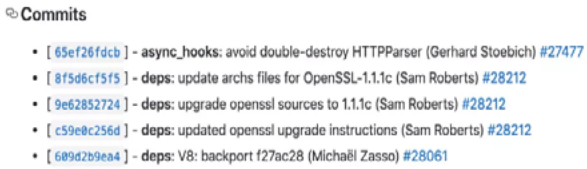
1. 正在开发一个比较大的版本，此时当前线上版本发现一个重要 bug 需要紧急修复;

2. 没有为每一个发布的版本指定唯一的 tag 标签以便回溯。

git 的 workflow 有很多种，各有适合的场景和优缺点。开发组件大多数时候是个人行为，偶尔是 team 行为，所以不太适合用比较复杂的流程。个人观点是，如果是在 GitHub 上维护的开源组件，则参照 GitHub 流程；如果是个人或者公司内私有环境，只要能保障并行多个版本，并且每一个发布的版本可回溯即可，可以在 GitHub 流程上精简一下，不区分 feature 和 hotfix，统一采用分支开发，用 master 作为线上分支和预发分支，开发分支要发版需要预先合并到 master 上，然后在 master 上 review 和单测后直接发布，并打 tag 标签，省略掉 pull request 的流程。

2.1.5 commit && changelog

一个组件从开发到发布通常会经历很多次的代码 commit，如果能在一开始就了解 git commit 的 message 书写规范，并通过工具辅助以便低成本地完成规范的实践落地，则会为组件的问题回溯、了解版本变更明细带来很大的好处。我们可能都见过 Node.js 项目的 changelog 文件：


Commits

- [65ef26fdcb] - **async_hooks:** avoid double-destroy HTTPParser (Gerhard Stoebich) #27477
- [8f5d6cf5f5] - **deps:** update archs files for OpenSSL-1.1.1c (Sam Roberts) #28212
- [9e62852724] - **deps:** upgrade openssl sources to 1.1.1c (Sam Roberts) #28212
- [c59e0c256d] - **deps:** updated openssl upgrade instructions (Sam Roberts) #28212
- [609d2b9ea4] - **deps:** V8: backport f27ac28 (Michaël Zasso) #28061

非常规范地将当前版本的所有关键 Commit 记录全部展示出来，每一条 commit 记录的信息也非常完整，包含了：commit 的 hash 链接、修改范围、修改描述以及修改人和 pull request 地址。试想一下，如果前期 commit 阶段没有很好的规范和工具来约束，手工完成这个工作需要花多长时间才能搞定呢?

目前社区使用最广泛的 commit message 规范是：Conventional Commits，由 Angular Commit 规范演变而来，并且配备了非常全的工具：从 git commit 命令行工具 commitizen，到自动生成 Changelog 文件，以及 commitlint 规范验证工具，覆盖非常全面。

2.2 质量和可维护性

开发组件的出发点是为了复用，其价值也体现在最大程度的复用上。团队内部的组件可能会在整个团队的多个系统间复用；公司内部通用的组件，可以为整个公司带来开发成本的降低；像 react、antd 这样的优秀开源组件，则会为整个社区和行业带来重大的价值。

组件是否可以放心使用，一个最简单直接的评判标准就是其质量的好坏。质量的好坏，除了上手试用以外，一般会通过几个方面来形成判断：

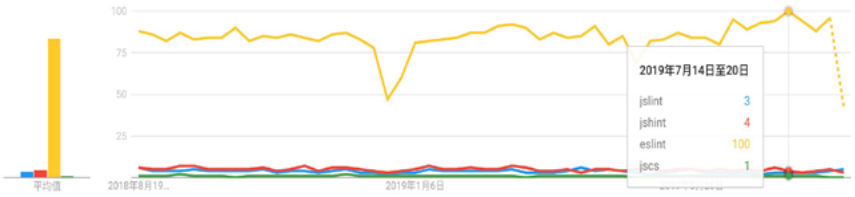
1. 是否有高覆盖率的单元测试用例?
2. 源码是否有规范的编码风格和语法检查?
3. 源码是否使用了类型系统?

这些都直接决定了开发者对这个组件的评价。试想一下，如果开发了一个公共组件，没有规范的开发流程和编码风格检查，也没有单元测试，随手就发布了带 Bug 的版本。此时用户第一次安装使用时就报错，这会让开发者对组件产生强烈的不信任感，甚至这种不信任感会波及到作者本身。

因此，一个规范且合格的组件，至少要在保障组件的质量上做两件事情：1）引入 JavaScript 代码检查工具来规范代码风格和降低出错概率；2）引入单元测试框架，对组件进行必要的单元测试。此外，类型系统（TypeScript）的加入，会帮助组件提高代码质量和可维护性，是组件开发时的推荐选择。

2.2.1 JavaScript 检查工具

JavaScript 语言第一个检查工具是由前端大神 Douglas Crockford 在 2002 年发布的 JSLint，在后续前端行业高速发展的十几年间逐渐演变出了 JSHint 和 ESLint 两个检查工具。关于这三个工具的演变历史，可以参考尚春同学在知乎发表的一篇文章：《JS Linter 进化史》。本文不再赘述，我们可以通过 Google trends 来简单了解一下这三个工具的热度，这里还加上了一个 JSCS 的对比：



可以看到，在过去一年内，全球范围内用户在 Google 搜索这些关键词的热度情况，这个图和身处在前端行业的感受是一致的。因此在 JavaScript 检查工具的选择上，可以毫不犹豫地选择 ESLint。

实际使用 ESLint 时有几点需要考虑：

1. 无论团队还是个人，都需要就配置规范达成认知和共识，以便可以将配置沉淀下来，作为通用的脚手架和规范。

2. 对于不同的组件类型，譬如 react 或者 vue，各有自己独特的语法，需要特定的 ESLint 插件才可以支持，而和框架无关的组件，譬如 lodash，则不需要这些插件。因此如何对配置进行分类和抽象，以便沉淀多套配置规范，而不必每次开发组件都需要重新对配置进行调整和修正。一个比较常规的做法是把组件按照应用的端（浏览器、Node、通用、Electron、…）和运行时依赖的框架（React、VUE、Angular 等）来进行配置的组合。

3. 借助 IDE 的插件来实现自动修复以便提高效率。

4. 如果是团队共同的规范，还需要形成一套规范变更的流程，以便组员对规范有争议时，可以有固定的渠道去讨论、决议并最终落实到规范中。

5. 引入了 ESLint，还需要考虑是否将 ESLint 加入到验收流程中，以及如何加入验收流程。

2.2.2 单元测试和覆盖率

一直以来对于业务类的项目要不要写单测这个问题，个人的选择是可以不写。互联网倡导敏捷开发，快速迭代上线试错，需求变化太快，而为前端代码写单测本身的成本可能并不亚于代码本身。

但是组件的情况就完全不同了，组件是一组边界清晰、效果可预期的接口和能力的集合。而且和业务类代码相比，组件更具备通用性，也就是说不太会随着业务的变更而变更。并且组件的升级通常会对依赖组件的系统造成潜在影响，每一个版本的发布都理应对功能进行详尽的回归测试，以保障发布版本的质量。由于组件的测试通常依靠开发者自己保障，不会有专业的 QA 资源配备，因此单元测试就是最好的解决方案了。

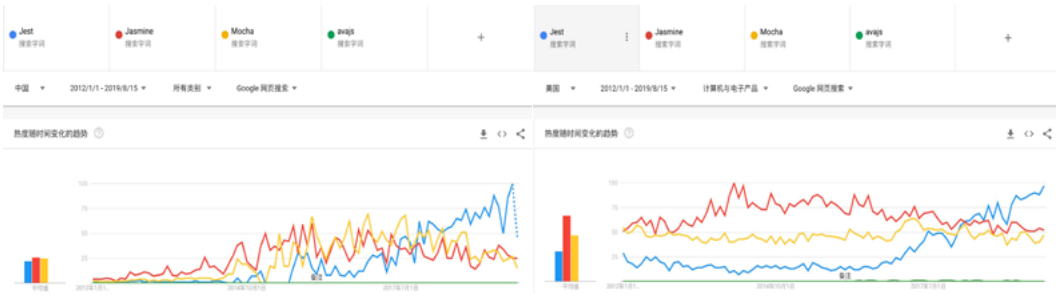
JavaScript 的单元测试解决方案非常之多，呈百花齐放、百家争鸣的态势，耳熟能详的譬如：Jasmine、Mocha、Jest、AVA、Tape 等，每一个测试框架都有其独特的设计，有些是开箱即用的全套解决方案，有些自身很简约，还需要配合其他库一起使用。

事实上，这些框架并无绝对的好坏，如何选择完全取决于个人和团队的喜好。这有一篇测试框架评测的文章，不妨一读：《JavaScript unit testing frameworks: Comparing Jasmine, Mocha, AVA, Tape and Jest [2018]》。

另外，我们依然可以通过 GitHub 上的 star 数和 Google trends 上的搜索量来略窥流行趋势一二。

测试框架	Github stars
Jasmine	14.5k
Jest	27k
Mocha	18.3k
ava	16.7k
tape	5.1k

Google trends 在中国 / 美国的数据：



可以看出 Jest 从 2014 年发布以来，增长势头是最猛的，并在短短 3 年内超过了其他老牌对手，成为目前最炙手可热的 Test Framwork。

除了测试框架选型以外，还有一个比较重要的指标要关注，就是测试覆盖率。推荐使用 nyc, 很多同学可能还用过一个名字比较特殊的库：istanbul。这两个库之前的渊源可以看这个 Issue 了解一下。

2.2.3 类型系统

如今的 JavaScript 已经不是原来那个在浏览器写写动效和交互的愣头小子了，它已经在 Web、Server、Desktop、App、IoT 等众多场景中证明了自己的价值，证明了自己可以被用来解决复杂的问题。事实上，JavaScript 正是通过将众多优秀的高质量组件、框架进行有机组合来提供这种能力的。

但是值得深思的是，JavaScript 采用了动态弱类型的设计，过于灵活的类型转换往往会带来一些不好的事情。试想这样的场景：

1. 调用一个组件的 API 函数，却不清楚这个函数的参数类型，只能自己去撸代码；

2. 对一个组件重要函数的参数做了优化重构，却无法评估影响面。

这些问题在强类型语言中有很好的解决方案，很多可能的错误会在编译期就被发现，很多改动的影响也会第一时间就被 IDE 告警。

事实上，越来越多的知名组件库已经开始引入强类型系统来辅助提高代码的质量和可维护性，比如 Vue.js、Angular、Yarn、Jest 等等。如果你想让自己具备类型思维，让组件具备更好的质量和可维护性，可以考虑把类型系统加到组件的脚手架中去。

目前可选的为 JavaScript 增加强类型检查的解决方案有 FaceBook 的 Flow 和 Microsoft 的 TypeScript，从当下的流行趋势来看，TypeScript 是绝对的首选。

如果想系统、深入地学习 TypeScript 又不想自己苦逼地撸官方文档，强烈推荐学习搜狗高级架构师梁宵在极客时间上的 TypeScript 课程《TypeScript 开发实战》。

2.3 可用性

组件的可用性，主要指的是从组件的使用者角度来看待组件的使用体验：

组件的文档是否完善且易于阅读?

组件暴露的 API 是否有详细且规范的输入输出描述?

是否有可以直接运行或者借鉴的 Demo ?

文档是否有考虑国际化?

2.3.1 文档

一个好的组件文档至少应该具备以下内容结构：

```
一句话描述组件是什么，解决什么问题
# Usage
//如何安装和使用，提供简单并且一目了然的示例
# API文档
```

```
//提供规范且详细的API接口文档，包括示例代码或者示例链接
#补充信息，譬如兼容性描述等
//如果是浏览器端组件，最好补充一下兼容性的支持情况；如果是Node端组件，也需要描述一下支持的Node.js版本范围
# ChangeLog
//描述各个版本的重要变更内容以及commit链接
#贡献、联系作者、License等
//如果组件希望他人一起参与贡献，需要有一个参与贡献的指南；除此之外，最好再提供一个可以直接联系上作者的方式
```

很多优秀的开发者可以很好地驾驭代码，但对如何写好一份组件文档却有些苦恼，这是因为代码是给自己看的，文档是给用户看的，这两种思维方式之间存在天然的差异。写文档时，需要换位思考，甚至可以把用户当小白，尽可能为小白考虑的多一些，如此可以提高文档的可读性，降低上手难度和使用的挫败感。

2.3.2 DEMO

对一个组件而言，Demo 的重要性不言而喻，还记得 Node.js 那个经典的几行代码创建一个 http server 的招牌式 demo 吗？可以说它几乎成为了 Node.js 的招牌和广告。

组件的 Demo 和文档都是为了可用性负责，但应该互有侧重，相得益彰。文档侧重于介绍关键信息、Demo 侧重于交付具体应用场景中的用法。

对于比较小的组件，这两者可以合二为一；对于 demo 代码量较多，且有多种使用方式和场景的情况，建议在 examples 目录下为每一种场景写一个可以直接运行的 Demo。

3. 结语

组件是开发者创造的产品，在这个产品的生命周期中，第一次发布只是一个开始而已。如何让更多用户关注到，并且成为它的忠实用户，乃至参与贡献才是接下来要重点解决的问题。关于这个话题，本文就点到为止了，欢迎大家在下面留言分享自己在组件推广方面的经验和技巧。

因本人能力的局限性，文中难免有解读不正确之处，盼望大家可以交流指正，笔者 GitHub 博客地址： https://github.com/shanggqm/blog

作者介绍：

郭美青，搜狗营销事业部高级前端架构师。2013 年加入搜狗后，一直负责搜狗商业广告变现业务的前端开发和技术体系建设，目前是竞价前端开发团队的技术负责人。

Kotlin 为跨端开发带来哪些影响？

作者 王莹

Kotlin 是一种在 Java 虚拟机上运行的静态类型编程语言，它可以被编译成为 JavaScript 源代码。虽然与 Java 语法并不兼容，但 Kotlin 可以和 Java 代码相互运作，并可以重复使用现有的 Java 类库。为了解 Kotlin 的更多相关信息，InfoQ 采访了 Kotlin 编译器团队成员 Svetlana Isakova，并希望通过本文为你带来一些关于 Kotlin 方面的启示。

InfoQ：为什么要开发 Kotlin 这门语言？开发一门语言需要考虑哪些因素？

Svetlana Isakova：主要的考量因素，包括这种语言由谁使用，他们又为何使用。在 Kotlin 刚刚诞生之时，人们强烈需要一种能够解决 Java 痛点（例如语法冗长以及可空性）等问题的面向 Java 平台的现代语言。Kotlin 很好地满足了这一需求，因此很多人现在倾向于选择 Kotlin 替代 Java。

如今，Kotlin 试图解决社区的另一大痛点：利用一种语言在不同平台之间共享代码。这种 Kotlin/ 多平台技术仍处于试验阶段，但通过观察有多少人在生产当中使用这项技术，我们得以确认这种需求相当强烈，我们也希望能够为此提供一套良好的实用解决方案。

另一项因素在于，编程语言需要拥有一家强大的公司作为后盾。JetBrains 是一家在全球范围内享有盛誉的成功企业，拥有丰富的相关专业知识。为多种不同语言（包括 Java、C#、Python、Ruby 以及 Go 等）创建 IDE，而 Kotlin 目前正是 JetBrains 的高优先级项目。目前，JetBrains 内部有一支 70 多人的团队，正在社区的帮助下推动 Kotlin 及其 IDE 支持工作。

InfoQ：在今年的 Google I/O 大会上，谷歌宣布 Kotlin 是安卓开发的首选语言，

请您从 Kotlin 的语言特性上来说一下这是为什么？您认为谷歌这么做的原因有哪些？

Svetlana Isakova：这项公告旨在鼓励开发人员转向 Kotlin，同时也使得谷歌公司能够专注于优先为 Kotlin 开发面向 Android 库的新型 API。Kotlin 语言中的大部分功能都是为了与 Java 实现轻松互操作而设计的。但除此之外，通过运用 Kotlin 中的特定功能（例如带有接收者的 lambda 表达式以及运算符重载等 DSL 创建功能，以及协程等），Kotlin 优先型 API 将带来更强大的表达能力与功能水平。

InfoQ：Kotlin 的目标是实现多平台可用，这对前端开发者实现跨端开发带来了哪些影响和变化？

Svetlana Isakova：最大的变化当然是在多种平台之间共享代码！这是开发者社区长期以来的持续需求，具体涵盖移动开发（在 Android 与 iOS 应用程序之间共享代码）以及 Web 开发（在后端与前端之间共享代码）。

Kotlin 多平台方案与其它现有解决方案（例如 Flutter、React Native 或者 Xamarin）之间最大的区别，在于前者可以改变需要共享的部件。我们并不需要对整个应用程序进行共享，有时候只需要保证其中部分组件——例如 UI 逻辑——具有平台中立性即可。Kotlin 方案提供理想的灵活性，所共享逻辑的百分比可以随时间变化，并根据您的实际需求调整。以往，如果有人最初选择 React Native，但之后决定转向 Kotlin 及 Swift 等原生语言，则需要从零开始重写整个应用程序。但使用 Kotlin/ 多平台则不会出现这样的问题：其中某些部分已经使用 Kotlin 及 Swift 编写，且具体需要共享的部分可以轻松完成修改。

再来看 Web 开发，在这方面多平台 Kotlin 项目主要针对那些需要同时编写前端与后端代码，但又更希望只使用同一种语言的开发者。在不同语言的上下文之间来回切换非常麻烦，Kotlin 则很好地解决了这个问题。

InfoQ：跨端框架 Flutter 使开发者可以使用 Dart 语言开发安卓 App，这会对 Kotlin 造成冲击吗？您怎么看待这件事呢？

Svetlana Isakova：对我来说，Flutter 仍将在其小众市场中得到应用，其中 React Native 是最主要的用例：大家能够在短时间内创建出小巧简单，但又外观漂亮的应用程序。此外，当应用程序的大小不太重要，且没有后续开发计划的情况时（例如会务类应用程序），那么熟悉 Web 框架但又不怎么了解 Kotlin 或者 Swift 的开发者，确实更适合选择 Flutter。

但在我看来，如果大家关注应用程序的逻辑，那么不妨仅在 UI 部分使用 Flutter，并选择 Kotlin/ 多平台直接编写业

务逻辑部分。接下来，大家可以轻松从 Flutter 迁移至 Swift 及 Kotlin 等原生语言。例如，当初创企业获得成功，那么接下来对应用程序的使用需求就会不断增长。

我觉得 Kotlin 并不会受到 Dart 的太多影响，因为 Kotlin 是一种更丰富且表达能力更强的语言。不过在 Android 平台上，我们构建 UI（特别是 Jetpack Compose）的方式确实受到了 Flutter 的影响。Flutter 为移动开发带来新的标准，例如用于 UI 组件的热重载与响应式声明方法等。我希望 Flutter 能够影响并最终改善整个 Android 平台。这其实也是技术快速发展，以及不同技术之间相互影响的一大明证，我觉得是好事。

InfoQ：最近，Kotlin 1.3.50 发布了，请您讲下新版本主要有哪些新特性？

Svetlana Isakova：这次的小版本更新带来了一系列质量与工具层面的改进，同时也在语言和库当中提供不少预览性质的实验性功能。这一次，最大的改进来自 Java 到 Kotlin 转换器（我们的目标是最大限度减少使用转换器之后，需要修复的红色提示代码量，这里推荐大家亲自试一试），以及新的周期与时间测量 API。Kotlin/JS 正在积极开发当中；新版本还提供 Dukat 预览，这款新的自动转换器能够将 TypeScript 声明文件转换为 Kotlin 外部声明。Kotlin/Native 代码现在可以使用特殊插件进行调试了。感兴趣的朋友可以在我们的博文当中了解更多详情。

InfoQ：Kotlin 计划添加哪些利于开发者开发的新库？

Svetlana Isakova：我们并没有打算所有的库都由 JetBrains 实现。很多库都是由社区负责实现与支持的。但在这里，我可以强调 Kotlin 团队正在进行的几个研究方向：

用于支持多种不同格式的多平台序列化库。由于其不使用 reflection，因而速度很快。

持续开发，代表基于挂起的响应式流（reactive stream）的流（Flow），目前已经加入稳定版本。套用于向 Kotlin 标准库内添加不可变持久集合的原型方案。

InfoQ：Kotlin 1.4 版本有什么计划？大约在什么时间发布？

Svetlana Isakova：1.4 版本将在明年发布。其中没什么特别的惊喜。大家可以关注新功能的后续进展，它们基本上都已经在本次小版本升级中提供实验预览版。我们正在努力改善整体用户体验，确保编译器与其它工具顺畅运作。很明显，MPP 是个需要优先解决的问题。

InfoQ：Kotlin/Native 的后续计划有哪些？您是否有意引入 self-host 机制？另外，JetBrains 是否有计划推广 Kotlin/Native ？

Svetlana Isakova：我们目前的关注重点，仍然是开发出完整而且稳定

的 Kotlin/ 多平台方案。其次，则是提高 Kotlin/Native 的编译性能。

Self-host（也就是在一种语言中使用另一种语言编写代码）目前还不是优先事务，因为除了作为“一般性概念验证”之外，这种功能不会给最终用户带来明显的助益。Kotlin/Native 代码可以复用 Kotlin 编译器的代码，其主要由 Kotlin/JVM 编写而成。

虽然尚处于预览版本，但很多人已经开始使用 Kotlin/Native 以及 Kotlin/ 多平台。当然，我们希望看到越来越多的用户在未来的实际场景当中享受两者带来的助益。

InfoQ：请问 Kotlin 在 WebAssembly 支持方面有什么规划？我看到 Kotlin 的版本库最近提交了将 WebAssembly 直接作为编译器后端的代码。

Svetlana Isakova：你观察得很细！长久以来，我们一直打算让 Kotlin/Native 支持 WebAssembly 编译。现在，我们打算直接从 Kotlin/JS 编译器中实现 WebAssembly，而不再使用 LLVM 作为转换载体。这套方案能够大大简化整个开发流程。到目前为止，我还没法做出太多承诺，但我们确实在积极进行探索。

InfoQ: Kotlin 在今年 7 月份的 O'Reilly OSCON（O'Reilly 开源软件大会）上获得年度突围项目奖。作为 Kotlin 的布道师，您有什么感想？

Svetlana Isakova：这是一项重要的荣誉，我对此感到非常高兴。如果没有背后强大的社区，这一切根本不可能实现。我要向各位成员表示由衷感谢！

InfoQ：为了推广 Kotlin，Kotlin 中文社区都做了哪些努力呢？

贾彦伟：Kotlin 中文社区目前维护有与官网风格一致的中文站、中文博客、中文论坛。其中，Kotlin 中文站是官方英文站的中文翻译版，很多开发者通过中文站来学习 Kotlin。目前 Kotlin 中文站每月有 120 万次用户访问。社区另一个影响力很大的是 Kotlin 公众号，会经常发布高质量的社区原创文章，目前已被 1 万多用户订阅。中文社区在知乎、掘金等技术社区也有开设账号，同步社区优质内容。除了这些线上资源外，Kotlin 中文社区还在北京、成都等地多次举办线下活动。

采访嘉宾介绍：

Svetlana Isakova：Svetlana Isakova 最初是 Kotlin 编译器团队成员，现在她是 JetBrains 技术布道师。她教授 Kotlin 语言，并在世界各地的会议上演讲，她也是《Kotlin in Action》一书的合著者。

贾彦伟：多年互联网研发从业者，关注编程、架构、技术管理、培训、开源等领域。Kotlin 中文站与 Ktor 中文站负责人。曾在 JetBrains 北京开发者日做过技术分享。