架构师

ARCHITECT

特 刊

大前端

架构师特刊

JUN. 2017

华章科技

Geekbang 极客邦科技

InfoQ

当我们在谈大前端的时候，我们谈的是什么

一篇文章抛砖引玉，探讨大前端的定义。

如何落地和管理一个“大前端”团队?饿了么大前端团队解密

结合饿了么大前端团队的实践，分享如何落地和管理大前端团队。

2017，我们来聊聊 Node.js

直面问题才能有更好的解决方式，Node.js你值得拥有！

大前端公共知识杂谈

想要真正掌握某种客户端开发技术，最好是要了解我们应该掌握那些方面。

Conversational UI：AI时代的人机交互模式

我们应该如何与具有智能的机器交互？

架构师特刊：大前端

本期主编　徐　川

流程编辑　丁晓昀

发行人　霍泰稳

联系我们

提供反馈　feedback@cn.infoq.com

商务合作　sales@cn.infoq.com

内容合作　editors@cn.infoq.com

Geekbang.
极客邦科技

InfoQ | EGO EXTRA GEEKS' ORGANIZATION NETWORKS | StuQ 斯达克学院

INTRODUCTION | 极客邦科技简介 »

极客邦科技 是一家 IT 技术与学习服务综合提供商，旗下运营 EGO 社交网络、InfoQ 技术媒体、StuQ 斯达克学院职业教育三大业务品牌。致力于整合全球优质学习资源，帮助技术人和企业成长。

企业使命	企业愿景	企业价值观
整合全球优质学习资源，帮助技术人和企业成长	打造全球领先的技术人学习和交流的平台	公开透明、诚实正直、每日精进乐于服务、负责守诺、创新敢为

10th 极客邦科技十年：

- 2007 年 3 月　极客邦科技创始人兼 CEO 霍泰稳将 InfoQ 引入中国。
- 2017 年 3 月　极客邦科技十周年，已拥有 EGO（社交网络）、InfoQ（技术媒体）、StuQ 斯达克学院（职业教育）3 条业务线。累计服务国内超过 100 万 的技术人，与超过 300 家企业合作，主办超过 50 场技术大会，走进日美，业务覆盖两岸三地。

- 每天，超过 40 万技术人通过 InfoQ 中国微信公众号了解最新技术趋势与最佳技术实践。
- 每天，超过 100 万技术人与极客邦科技旗下垂直矩阵化的新媒体平台交流互动。
- 每月，超过 1000 人在斯达克学院上学习新课程，掌握实用性 IT 新技能。
- 每年，超过 1 万名中高端 IT 人通过 QCon、ArchSummit 等国际性大会学习最新技术实践，了解最前沿技术趋势，结识业内技术大咖。
- 每年，超过千万的独立账号访问 InfoQ 中国的网站了解国际前沿技术与资讯。
- 每年，EGO 有超过 30 场的学习活动，超过 300 位的国内 CTO、架构师、技术 VP 等高端技术管理者通过 EGO 组织建立紧密联系，分享经验，共同成长。

InfoQ ueue

ABOUT | 关于 InfoQ »

InfoQ 是一家全球性在线新闻 / 社区网站，创立于 2006 年，目前在全球拥有英、法、中、葡、日 5 种语言的站点。InfoQ 中国于 2007 年由极客邦科技创始人兼 CEO 霍泰稳先生引入中国，同年 3 月 28 日，InfoQ 中文站 InfoQ.com.cn 正式上线。每年独立访问用户超过 1000 万人次。

InfoQ 中国以促进软件开发领域知识与创新的传播为使命，面向 5–8 年工作经验的研发团队领导者、CTO、架构师、项目经理、工程总监和高级软件开发者等中高端技术人群，提供中立的、由技术实践主导的技术资讯及技术会议，搭建连接中国技术高端社区与国际主流技术社区的桥梁。

TECHNOSPHERE | InfoQ 覆盖的技术领域 »

大数据　移动　前端　云计算　架构　研发　AI　运维　容器

PRODUCT | InfoQ 产品 »

垂直社群	聊聊架构、移动开发前线、大数据杂谈、细说云计算、前端之巅、高效开发运维
新媒体	InfoQ 微信矩阵公众号、今日头条、百度百家、微博、直播、短视频
直播	大咖说
短视频	二叉树
技术大会	QCon 全球软件开发大会 | ArchSummit 全球架构师峰会 | GMTC 2017 全球移动技术大会 | CNUTCon2017 全球容器技术大会

卷首语

GMTC大会主编
徐川 Amos

我们要埋头做事，也要抬头看路。

随着移动开发与前端技术的融合，不但移动开发的技术栈发生变化，前端领域本身也在进行着激烈的演进，让人目不暇接。从更大的角度看，VR/AR 蓄势待发，AI 成为新的热点，智能助手被视为下一代计算平台，给人以一种即将落后时代的紧迫感。

人的天性就是喜新厌旧，设计的趋势也是一年一变，做图形界面和用户交互的也只能跟上，这是做大前端的宿命。我们只有抬头看路，持续学习，才能保证不掉队。

但总有一些不变的东西，在大前端里，我们发现，有些公共知识，在不同的平台可以复用，因为大家解决的是类似的问题。就比如布局，绝对布局、相对布局、弹性布局、网格布局，无论在哪个平台，变来变去就这么几种，甚至随着 iOS 平台的屏幕尺寸也碎片化了，大家都开始倾向使用弹性布局 Flexbox，无论是 Web 还是 Native。这

些公共知识，就是大前端的知识体系。

当然，不同的平台有自己的特点，比如 Web 平台无需热更新，代码部署上去用户一刷新就生效了。而如果你需要开发 iOS 或 Android 应用，你还需要自己研究实现。这是平台特定的知识，是开发对应平台应用时需要掌握的。

所以，大前端开发者的知识储备 = 公共知识 + 平台特定知识。有了这些知识储备，其实并不用疲于奔命的去学习新知识，去看看业界前沿有哪些新东西，及时更新自己的知识库就行了。

去年的 GMTC 大会关注的是移动开发，今年则是首次以大前端为主题，我们希望跟上时代的脚步，为大家呈现业界最前沿的一些思潮和趋势，今年更为激进一些，则是想推动着大家再往前走两步，能看得更远更清楚一些，这也是我们做事的初心。

由于大前端的发展迅速，以及作者和我的见识所限，这本小书所选的几篇文章，有些过一段时间会过期，有些则存在这样那样的不足，但我希望你能领会它们所要传达的意思，通过思考把它们变成自己的想法，这样能够纠正错误，也能随时更新。

抬头看路，再加上独立思考，我相信大家都能走出自己的一条路。

当我们在谈大前端的时候，我们谈的是什么

作者 徐川

我在筹备GMTC2017大会的时候，走访了美团点评的刘平川老师、滴滴的左志鹏老师、手淘的天施老师，对大前端的问题进行了深入的讨论，在这里，我想用这样一篇文章来抛砖引玉，探讨大前端的定义。希望能看到更多关于它的讨论和分享。

在今天，大前端并不是一个陌生的词汇，我们偶尔会听人谈起它，前些天还看到卓同学写了一篇《大前端时代下App开发者的生存之道》，说明这个词开始成为某种共识了。

但是大前端到底指的是什么？事实上大前端并没有明确的定义，它由国内业界发明，甚至没有对应的英文词汇（如果有，请悄悄告诉孤陋寡闻的笔者）。有人对发明技术词汇不以为然，但我认为国内大前端相关技术发展相较于国外并不差，我们有必要在理论、标准方面也有所建树，这样才能与我们的发展情况相匹配。

三个层面上的大前端

如果分析我们到底在什么时候使用

大前端，大概有三种不同的语境，在每种语境下，大前端的定义都有所不同。

大前端与NodeJS与前后端分离

在前端同学的口中，大前端有时与NodeJS一起提，有时与前后端分离一同提起，事实上，大前端概念也正是由前端同学提出，从这里，我们可以得出大前端的原始定义。

过去几年，前端技术经历了爆发式的发展，这种发展最重要的推动者之一就是NodeJS。NodeJS为前端建立了与系统之间沟通的桥梁，从此前端技术不仅能在服务端大放异彩，还能在本地的前端开发工具与工作流上大展身手，前端从此被解放了，JavaScript统治世界的论调一度甚嚣尘上。

不过，当人们冷静之后，发现NodeJS在服务端并没有太多的优势，再加上NodeJS技术本身发展的一些波折，导致它在服务端的应用并不理想。不过，前端同学还是取得了一些阶段性胜利，其结果就是前后端分离。

在以前前端页面模板由后端生成，导致在页面需要频繁修改的时候效率低下，前后端分离指的是后端只提供接口，前端对页面有完整控制，同时通过中间层将前后端隔开，在这里对数据进行抽取、聚合、分发等操作。这个中间层，通常也是由前端同学负责。

从这个意义上，大前端的原始定义可以称为前端技术的扩大化，包括NodeJS，同时对Web页面有更强的控制权，开发承载更多功能的页面。

大前端与泛GUI交互

移动互联网时代到来之后，移动App成为新的主流，而浏览器的地位则逐渐降低，传统的前端开发遇到尴尬。

当然，前端并未真正遭遇困境，以PhoneGap/Cordova为代表的Hybrid开发，以及内嵌在App中的WebView开发，再加上微信成为主流之后的“微信Web”，前端技术其实在移动端也有很多的使用场景。

但是，当时人们提起移动开发，主要指的还是iOS与Android原生开发技术，这一情况随着React Native的发布得到了改变。其实直到现在，在国内外大规模使用React Native仍然不多，但是它的确能解决原生的跨平台代码复用、动态化等痛点，又避免了之前Hybrid的性能问题，因此受到广泛关注。

随着React Native的加入，前端的技术栈再次扩展。并且React Native让我们发现，其实通过加入一个虚拟视图层（Virtual DOM），逻辑操作和模型部分的代码能够得到很大程度的复用，在已有的实践总结中，大部分React Native代码都得到了80%以上的复用。

虚拟视图层也不仅仅只能用在移动端，在所有通过图形界面进行人机交互的地方都可行，在PC、Web、移动设备甚至还未发明出的未来的种种设备上，只要系统能运行JavaScript引擎，理论上都可以采用类似React Native的开发方案。这种前端技术，当然可以称

为大前端。

如果说前后端分离是前端在纵向上变大，那么这种跨不同的终端的前端技术则可以说是在横向上变大。

大前端团队现状

在实际中，还有一种使用大前端的情况，那就是国内公司的大前端团队 / 部门。据我了解的情况，目前美团点评、饿了么、网易杭研都有叫大前端的团队或者在对外时使用该称谓。

不过，由于之前大前端并没有明确的定义，这些团队的人员构成并不相同，它们都是各个公司在自己对大前端的理解，以及对公司业务的支持需求下设立的。具体情况如下（可能有偏差）：

- 美团点评大前端团队：包括FE、iOS、Android开发，以及一些工程化工作。
- 饿了么大前端团队：以FE为主，包括NodeJS，以及Weex等。
- 网易杭研大前端团队：去年底在网易杭研执行院长汪源的一次分享中，他称网易杭研大前端的技术体系，包含Web前端、PC客户端和移动端。

这些写在新闻稿、印在名片上的文字，是帮助大前端概念落地的重要助力，同时也是最终确定大前端的定义到底是什么的判定依据。当然，目前案例太少，还不足以影响我们对大前端的理解。

为什么说大前端是发展趋势

在客户端开发上，Native 与 HTML5 之争持续快十年，吵了人们都失去兴趣了，从现在来看，并没有谁取代谁，而是有融合的趋势，融合之后的产物就是大前端。

在这里我大胆预言：大前端不仅会成为移动开发与 Web 前端的发展趋势，也将会是未来的显示设备终端的开发技术趋势。

为什么这么说？

终端碎片化

我们已经进入一个终端碎片化的时代，iPhone 第一代发布到今年就整整 10 年，在这 10 年里，我们并没有发现智能手机有被取代的迹象。但是创新仍在继续，于是我们有了智能手表、TV、眼镜、头戴 VR 等等新设备，可以想象这样的设备仍然会继续增多。

这些新设备同时也是新平台，与智能手机类似，可以安装第三方应用，并且，这些平台基本都支持浏览器或内嵌浏览器引擎。虽然有些平台限制使用 Web 技术开发应用，但这只是平台政策原因，只要放开限制，前端技术就能以某种姿态进入，甚至成为主流。

有早期 Hybrid 和后来的 React Native 的探索，在显示终端应用的开发上，前端基本已经成为必备技术。

Serverless

Serverless 中文译为无服务器架

构，是软件架构领域的一个热门概念。这里的无服务并不是说不需要服务器，而是说新的架构取代了传统服务器的概念。Serverless 的代表是 2014 年亚马逊发布的 AWS lamda，后续各大云计算厂商也纷纷跟进。

对于终端开发者，并不需要太深入的了解 Serverless，因此这里不过多介绍，只需要知道它被认为是云计算发展的趋势之一。

Serverless 与大前端的关系则在于，Serverless 需要更强大的前端，在《Serverless Architectures on AWS》一书中介绍了 Serverless 的五大原则，当中有这样一条：

Create thicker, more powerful front ends

因此，从软件架构的发展趋势来看，前端会越来越“大”，在整个系统中的重要性也会提升。

大前端的代表技术

说了这么多，到底哪些是大前端的代表技术？从业务上来说，我认为终端开发、网关设计、接口设计、桌面端的工程化都可以算是大前端的业务范畴。具体的技术，则是基于 HTML5、NodeJS 的通用技术，以及各平台的专有技术。从现阶段来说，还需要掌握一些代表性的框架、平台等。

React系与Vue系：两大前端生态

前端框架目前有三架马车，除了 Angular 之外，React 与 Vue 都已经形成各自的生态体系。

生态的意义就是覆盖全面，几乎没有短板，React 和 Vue 已经覆盖了目前主流的系统平台，并且可以用 React Native、Weex 等框架进行原生开发，相较于其它技术有很强的优势。

PWA：开放的理想

PWA 是 Google 力推的技术，对于前端开发者来说，它代表着标准化的努力和开放的理想。虽然从目前来看，它还达不到实用阶段，但从我近期获得的一些信息表明，这项技术还是有很大潜力的，也有不少的支持者。今年可能就会有更多的实践案例涌现出来。

小程序：Super App指向的另一种未来

微信小程序正式发布之后，与它未发布之前的火热形成了反比，在市场上几乎没有声音了。但这只是产品策略的原因，并不是技术带来的问题。事实上微信小程序的用户体验很不错，表明了这项技术的应用潜力。

小程序更多的还是为我们带来了一种可能性，超级 App 成为 PC 时代浏览器的精神继承者，成为新的操作系统。

大前端带来的影响

最后来说说大前端带来的影响。

新的移动开发技术栈

移动设备作为主流的终端设备，其

> “纯原生的移动开发的道路会越来越窄，整个移动开发的技术栈必须要做一个大的改变。”

应用开发技术也应该是大前端最关注的技术。在以前移动开发的技术栈以原生开发为主，但以后恐怕做移动开发需要同时掌握前端技术才行。卓同学的文章也表达了这个意思。

在今年 1 月份的 WeexConf 上，天施老师分享的一段话让我颇有感触，他的大意是移动端经过 10 年演化，创新变缓，移动开发正走在标准化的道路上，所以 Weex 会遵循 W3C 的规范。而我有更深一层的理解：移动开发之前很多组件都需要靠自研，但随着大厂更多的开源，我们在基础组件 / 框架上的自研需求会越来越少，移动开发没有 W3C 这样的规范，但会有基于开源的事实标准。

所以，我认为纯原生的移动开发的道路会越来越窄，整个移动开发的技术栈必须要做一个大的改变。

新职业：大前端工程师

随着大前端的概念逐渐深入人心，会带来什么？我认为会出现新的职业：大前端工程师。

它与以前的 Web 前端的区别是，大前端将做更多的终端开发、工程化等工作，而不仅仅只是开发 Web 页面。大前端工程师将能搞定所有端上的开发。与充满争议的全栈工程师相比，它更具可操作性。

并且，大前端工程师将会是一个拥有强大生命力的职业，因为显示终端设备的生命力会很强，毕竟人类的信息获取有 80% 以上是通过视觉，无论 Amazon Echo 这样的语音交互设备如何演化，显示终端都会有一席之地，大前端也因而不会失业。

结语

本篇内容肯定有偏颇的地方，我所说的也不可能都是对的。希望它能引起你的思考，那么本文就达到目的了。

如何落地和管理一个“大前端”团队？饿了么大前端团队解密

作者 韩婷

最近，“大前端”这个词被频繁提及，很多团队也在重新思考“大前端团队”和“移动团队 + 前端团队”这两种模式的优劣。而在大家还在热火朝天地讨论概念的时候，饿了么大前端团队已经茁壮成长，有了很多先人一步的实践了。InfoQ 特别邀请了饿了么大前端部门负责人林建锋，请他结合饿了么大前端团队的实践，向大家分享如何落地和管理一个大前端团队。

前言

平时大家会叫我小鱼或 Sofish，尴尬的是只有屈指可数的同行知道我真名叫林建锋。曾经，为了逃离数学，大学我选了法学这个专业；而毕业前，又为了逃离职业性的“辩论”选择了不用说太多话的前端开发，至此踏上了程序员的不归路。

这段程序员的不归路从实习开始，到远赴杭州支付宝，而后以第一个前端工程师的身份百姓网，再之后选择创业，最后加入饿了么并定居上海。目前作为饿了么大前端部门负责人，我和一群小伙伴在努力把饿了么变得更好，顺路立志成为业界顶尖团队。

一、饿了么大前端团队的定位

1.为什么叫“大前端”团队

大前端这个词最早是因为在阿里内部有很多前端开发人员既写前端又写 Java 的 Velocity 模板而得，而我们部门成为之初所承担的内容不仅仅是前端，还包含 CDN 和 Nginx 层，所以取名“大前端”。时至今日，大家所说的大前端已经包含了前端、Node、Native-Like (Hybrid / Weex / RN)，甚至包括 Native App 开发。

2.我所理解的“大前端”

在我看来，“大前端”是一种变化形态多过于固定的职责范围，是“前端”职责范围的延伸，是对这个社会分工因为能力范围和因此所达到效率提升的一种进化形态。给大家分享个小道消息：CTO 曾多次开玩笑说 —— 你们负责的已经不仅仅是前端，要不就改名“大全栈”吧。这部门的名字很霸气但同时也太高调，所有并没有接受 BOSS 的提议，而是继续沿用“大前端”这个部门名。

3.饿了么大前端团队的职责

如上面所说的，除了纯 iOS / Android App 的开发，其他都是我们团队职责所在，同时我们还负责公司 HTTP API 层，有一些自己运维的系统。

从分工来说来，目前我们有架构与机动组负责框架、规范、工具的生产；Node 研发团队负责公司 Node 业务的基础设施和业务支撑；多个业务前端团队支持不同的 BU 前端。这里值得一提的是，架构与机动组会对每个业务团队至少派出一名架构师长期、深入地了解业务团队所遇到的问题并反馈到架构团队，并在解决方案提出后协助推动。

在具体职能分工之外，各团队也有以项目而组织起来的虚拟团队，比如由我们部门负责的“游戏中心系统”就由 Node 研发团队和架构与机场组中的成员组成；又如小程序团队；亦如发起一次由“93 后”独立组织的招聘；专栏编辑团队、官微小分队、对内对外分享会小分队，等等。除了大家看到的开源产品，内部的所有项目都是“内部开源”，特别鼓励大家提 Pull Request 和相互 Code Review。这些与部门所创建的文化息息相关，且如你可能猜到的，大多合作都是一旦有人提出，即自发组队。

二、饿了么大前端团队如何看待和落地新技术

我们是如何看待新技术的？在面试前端 Leader 候选人的时候，我通常会问一个问题：你如何看待技术债，有没有办法可以避免？几乎任何程序员都讨厌还技术债，所以才会有那句“挖坑一时爽，填坑火葬场”。因为痛苦才会非常值得我们去思考和解决。技术债从某种程序上代表着过时的技术，而新技术也将在未来某个时刻变成新的“技术债”。饿了么大前端是如何回答这个问题的？就是我们对新技术的看法。

我 2014 年加入饿了么，那会 PC 和 Mobile 都还是后端渲染的模式，使用 Bootstrap 和 jQuery。我进去的第一件事是用 Angular.js 重写移动网站，并且前 / 后端分离，提倡后端即服务。在高速发展期利用移动网站支撑了当时 10 倍增长的业务；第二件事是重构 PC 站，把 web2 升级到 web3（Code Name），同样是前后端分离，到 14 年底 15 年初，基本实现完全分离。重构一方面是提高前端协作的效率，一方面是提升两部各自的掌握力 —— 只要 API 约定一致，内部封装可以自己随时变更（提升）。在此之后，我们的方向一直是比较激进的技术模式 —— 新业务可以用任何框架，大家自由选择；旧业务只要负责团队（人）有能力升级，那就鼓励用最新的。由于后端已经完全分离出去，所以从掌握力大大提升，加上这种受鼓励的激进技术模式，Angular.js 1.x 这种当年的新技术在日渐变成技术债的今天，也已经几乎全被重写成 Vue.js 和 React.js。

当然，也像今天大家能看到的，当大家都还在转发关于 PWA 文章的时候，我们已经和 Google 合作并把 PWA 上线；开源的项目大多是内部成熟应用的项目，而开源的产品亦让我们成为 Vue.js World Ranking 最高的团队；Weex 方面，我们是除阿里内部团队外最早上线的大型用户。这些看起来快速和无止追求新技术的背后，其实并没有大家想的那么了不起，仅仅是因为团队文化本身就鼓励利用新技术解决问题。

如果一定要拿 Vue.js 来举例，可能和你想象不一样的是，不用“落地”，仅仅是因为有人说了一句“WOW，Vue.js 写起来好简洁啊，大家要不要一起来试试？”。然后，一个团队，两个团队……几乎所有团队都开始应用，几乎所有新项目都在用。一位 IBM 的朋友告诉我，他申请在项目试用 Vue.js，上级说在半年后试用，结果半年后又被推翻了。所以你看，在合适的文化土壤里新事物就是一种常态，如果做一个项目用什么技术还要上级主管确定“能不能做”，那本身就不是一件简单的事。

我们对于技术选型通常来说要求是 —— 是否提升饿了么运行的效率或者团队开发的效率？是否能 hold 住？有没有人负责到底？符合这样的条件，就会被推动。当大家都在说 HTTPS 是好东西的时候，我们已经推动全网上线，诸如此类 —— Webp、Https、Vue.js / React.js / Angular.js、Weex、PWA 都是如此。比如大家可能去年就关注到我们在上线 HTTP/2，而今天饿了么大前端内部已经做过 HTTP/2 Server Push 的实验，可以想象在不久的将来，线上应用将会大面应用。这就是我们的选型和落地模式。

三、饿了么大前端团队的特色：散养

特色？如果只用两个字来回答就是 —— 散养。但仅仅这样描述大家会一

脸问号。外部对我们的评价是：“新技术跟进好快啊”、“怎么又又又出去玩了”、“下班很早”、“好多大牛啊”、“开源的东西获得好多星星啊”，诸如此类，但这不是我们要特地我呈现的，只是一种表像，或者说是一种副产品。

一个团队的风格与创始人有很大的关系，比如喜欢愉懒的我会更多考虑自动化；又如有洁癖所以会有代码规划和 Code Review；还有大家看到的爱玩，所以会经常有团建、下午茶这样的文化；但另一方面，我并不想让团队仅仅是和我一样有大家喜欢的，同时充满缺点，而希望是不断尝试的，兼容并包的，让每个人的闪光点都成为集体中有特色标志的。所以我有自己的一套，就是前端所说的“散养”式管理，说起来可能会很大，重点说几点：

1. **招聘最好的人**。最好的人不是业界里的所有明星，更重要的是能从某方面给带队带来提升的人。这些人通常自驱能力强，只要有一个方向就能推动事件的发生和发展。加入的人会被要求不要以学习姿态加入这个团队，而是以加入会让这个团队会让其变得更好为姿态，成长就会成为副产品。
2. **鼓励创造结果而不是追求上班时间**。如果我们的目标是页面加载时间不要超过 800ms，那么目标就是 800ms 而不是上 12 个小时的班。
3. **营造环境**。我们有最好的人，我们追求结果而不是追求上班时间，我们鼓励主动和主人公意识，我们创新以打破规则 ，我们声明所有人为自己而生为用户工作而非老板，我们会包个酒包或找个海岛玩到天亮。有很多东西是要刻意去营造的，创新土壤，主动的意识，热爱生活的文化，鼓励什么就会聚集/培养一群什么样的人。

因为这样的管理方式，通常大部分事情都会被内部很好地解决，而我也得到更多的时间去思考如何做和决定做什么；团队也因为成员不断成长闪耀不同的光芒而变得更好。如果以官话来说，就是我们要发现一种“可持续发展”的模式。这种模式目前运行的不错，无论是业务上的，还是团队文化本身，抑或是加入成员的成长，都是让人高兴的。但，更好的方式仍在探索，如果说只分享一个点的话，那就是千万别用“管”的方式，而是“理”顺，就会顺理成章。

至于是不是盲目追求新技术。上面我们已经谈过技术选型的要求，最最重要的也是最最根本的问题“是否进升饿了么运行的效率或者团队开发的效率？”，我相信如果大家能回答好这个问题，就解决了“盲目”追求的问题。

最后说一句，无论是管理上、技术上、生活上，预留一定的空间和自由度，一定会带来惊喜。具体就不解释了，大家自行理解，或许某天我们遇到可以用这个话题开场，就开聊起来了。

四、大前端模式的利弊

“大前端”模式的特点前面已有提及，即是对前端本身的一种能力范围延伸。移动开发团队，我这里指包含移动网页、Native-Like、Native App 这样的团队，如携程；纯大前端的团队如美团和饿了么北京研发中心，只要是客户端的无论是网页还是 App 都在单一个团队；饿了么不仅有大前端，还有各 BU 的 Native App 团队，甚至还有专注于移动基础框架的公共的移动技术团队。

不同的分工模式，其利弊通常与公司状态、团队本身所创造的价格有很大的关联；虽然大家都是“离用户最近的工程师”，但没有公式可抄。

就拿饿了么大前端来说，最开始是因为业务的快速发展，除了 C 端部门，其他新成立的部门前端工程师极度紧缺，为了资源的高效配置，才成立了大前端这个部门。这是当时公司业务的状态。

创始人和 CTO 曾问我：“你觉得如果今天合了，几时会分？”当时，我的想法是，如果一个业务前端团队发展到 10 人左右，在负责好本身的业务外，能建立且不断升级自己的技术基础设施又具备有自己的文化，是一个分的时机。时至两年后的今日，我已不再是这样的想法，并且新成立的 BU 更愿意把人放到大前端。

这是为什么呢？我们从以下几点分析：

1. 如果一个大团队，并不能提便利的基础设施，不能创建自由且充满可能的文化，不能持续提升自己并帮助成员提升其自身水平。也就是大家经常谈到的技术追求、归属感、成就感。那么，打散到业务组可能

更灵活。所以前端业务团队的分与合归根到底在于 —— 大团队是否创造比小团队更高的价值。

2. 大多数人高估了“前端+后端+产品”坐在一起的效果，认为这样就能完美解决问题。很多时候，对于程序员来说更少的打扰，更多的思考（比如坐内部电梯去找某个人太慢了，就会开始思考是不是有必要去找某个人）过后的沟通，是更高效的沟通。
3. 划分框架、机动与业务团队，一方面基础设施共享（框架工具）有更多重用，人员调度可以省不少资源（小团队需求少的团队可以合并支持）且又有专注于业务的团队，似乎是最前非常不错的划分方式，而在 10 个人的团队中是很难做到的。

由此，我们可以看出，一方面是业务影响，另一方面也依赖团队本身产生的价值，今天我们要分还是合，其利弊计算出现在决定做出之后，带领团队的人能否利用“合”的优势去产生出更大的价值。这个问题的答案就是利与弊的答案。

五、业界大前端团队的现状

我们团队每年都会以个人或者团队名义邀请多位前端业界大牛来内部交流，也会组织内、外部交流会，这通常是几个电话和微信就搞定的事，总体交流还是比较多的。即使没有专门的会议，也会偶尔相互通通气。

我没有具体统计过业界现状，但从我这边交流过的团队来说，现在很多团队都是大前端方向，或向这个方向发展的比较多。有少部分像携程、腾讯这种体量本身就很大职能划分也比较明确的公司，做的事可能是“大前端”但分开仍是比较偏向于 JS+HTML+CSS 这样的纯前端模式。

这里也期望读者所在的团队，如有新的实践和想法我们可以偶尔探讨。

六、如何落地大前端团队？

前文也有提到，要不要组建大前端团队，一方面是看业务是否有需求，另一方面是看合能否比分开带来更大的价值。

除非人极少，通常来说业务大多数情况都会随公司的发展变得越分越开，而价值则主要是关于人和文化。目标不是为了合而合，或者追随业界模式，而应该着眼于是否优化了公司的人才架构从而优化业务。

如果一定要从具体实施点上来说，这里说两点：

- 框架团队和业务团队应该同时设立，并且框架与业务不能相互脱离，具体可以参考饿了么大前端的模式 —— 相互渗透；
- 在大职能团队下，尽可能以业务划分团队，不要以职能划分团队；反之亦然。如果分析我们到底在什么时候使用

2017，我们来聊聊 Node.js

作者 桑世龙

版本帝？

Chrome 浏览器已经蹦到 57 版本了，是名副其实的版本帝，作为兄弟的 Node.js 也一样，1.0 之前等了 6 年，而从 1.0 到 8.0，只用了 2 年时间，这世界到底怎么了？

我们就数一下（见图 1）：

- 从v0.1到0.12用了6年；
- 2015-01-14发布了v1.0.0版本（io.js）；
- 2.x（io.js）；
- 3.x（io.js）；
- 2015年09月Node.js基金会已发布Node.js V4.0版 与io.js合并后的第一个版本；
- 2015年10月Node.jsv4.2.0将是首个lts长期支持版本；
- 2016年底发布到4.2.4 && 5.4.0；
- 2016年3月20日v4.4.0 LTS（长期支持版本）和v5.9.0 Stable（稳定版本）；
- 2016 年底 v6.0 支持95%以上的es6特性 ， v7.0通过flag支持async函数，99%的es6特性；
- 2017年2月发布v7.6版本，可以不通过flag使用async函数。

整体来说趋于稳定：

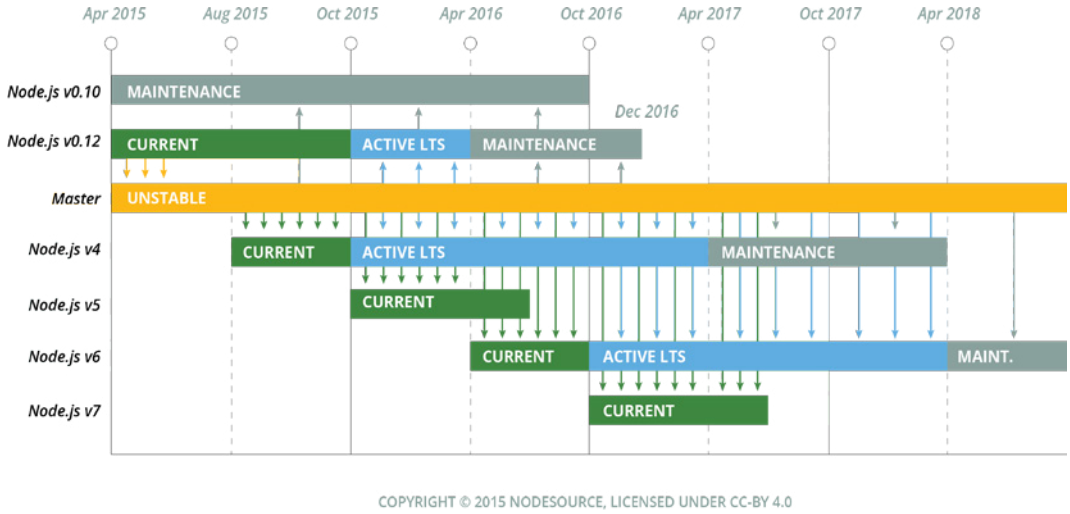

图 1

- 成立了Node.js基金会，能够让Node.js在未来有更好的开源社区支持；
- 发布了LTS版本，意味着api稳定；
- 快速发版本，很多人吐槽这个，其实换个角度看，这也是社区活跃的一个体现，但如果大家真的看CHANGELOG，其实都是小改进，而且是边边角角的改进，也就是说Node.js的core（核心）已经非常稳定了，可以大规模使用。

已无性能优势？

Node.js 在 2009 年横空出世，可以说是纯异步获得高性能的功劳。所有语言几乎没有能够和它相比的，比如 Java、PHP、Ruby 都被啪啪的打脸。但是山一程，水一程，福祸相依，因为性能太出众，导致很多语言、编程模型上有更多探索，比如 go 语言产生、php 里的 swolo 和 vm 改进等，大家似乎都以不支持异步为耻辱。后来的故事大家都知道了，性能都提到非常高，c10 问题已经没人再考虑，只是大家实现早晚而产生的性能差距而已。

编程语言的性能趋于一样的极限，所以剩下的选择，只有喜好

那么在这种情况下，Node.js 还有优势么？

- 实现成本：Node.js除了异步流程控制稍复杂外，其他的都非常简单，比如写法，你可以面向过程、面向对象、函数式，根据自己的解决选择就好了。不要因为它现在变化快，就觉得自己跟不上潮流。尤其是后端程序员转Node.js几乎是2周以内的成本，某些语言光熟悉语法习惯也不止2周吧？
- 调优成本：Node.js即使不优化，它的性能也非常好，如果优化，也比

其他语言更简单

- 学习成本：是否必须用，如果是必须要用，那就少学一样是一样，人生有限，不能都花在写hello world上。我想问，大前端离得开js么？

误读：Node.js已无性能优势，它现在最强大的是基于npm的生态

上面是成本上的比较，其实大家把关注点都转移到基于 npm 的生态上，截止 2017 年 2 月，在 npm 上有超过 45 万个模块，秒杀无数。npm 是所有的开源的包管理里最强大的，我们说更了不起的 Node.js，其实 npm 居功甚伟，后面会有独立的章节进行阐述（见图 2：来自 www.modulecounts.com 的各个包管理模块梳理的比较）。

npm 生态是 Node 的优势不假，可是说“Node.js 没有性能优势”真的对么？这其实就是误读，Node.js 的性能依然很好呀，而且它有 npm 极其强大的生态，可谓性能与生态双剑合璧，你说你死不死？

异步和回调地狱？

天生异步，败也异步，成也异步

正因为异步导致了 API 设计方式只能采用 error-first 风格的回调，于是大家硬生生的把 callback 写成了 callback hell。于是各种黑粉就冒出来，无非是一些浅尝辄止之辈。但也正因为回调地狱是最差实践，所以大家才不得不求变，于是 thunk、promise 等纷沓而至。虽然 Promise/A+ 不完美，但对于解决回调地狱是足够的了。而且随着 ES6 等规范实现，引入 generator、co 等，

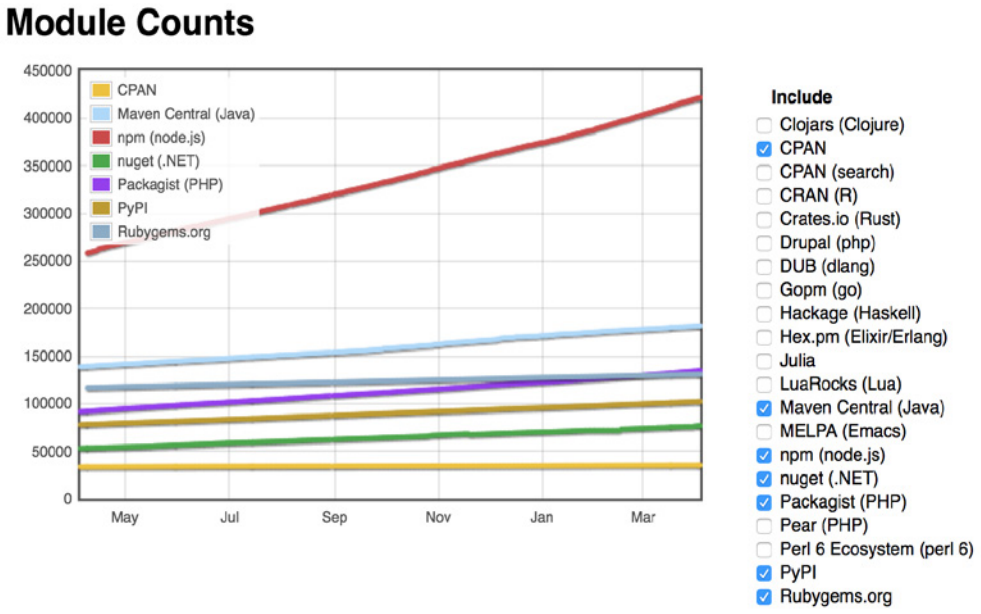

图 2

名称	说明
callback	Node.js API天生就是这样的
thunk	参数的求值策略
promise	最开始是Promise/A+规范，随后成为ES6标准
generator	ES6种的生成器，用于计算，但tj想用做流程控制
co	generator用起来非常麻烦，故而tj写了co这个generator生成器，用法更简单
async函数	原本计划进入es7规范，结果差一点，但好在v8实现了，所以node 7就可以使用，无须等es7规范落地

表 1

让异步越来越近于同步。当 async 函数落地的时候，Node 已经站在了同 C#、Python 一样的高度上，大家还有什么理由黑呢？

本小节先科普一下异步流程里的各种概念，后面会有独立章节进行详细讲解（见表 1）。

有时，将一件事儿做到极致，也许能有另一种天地

应用场景

MEAN 是一个 Javascript 平台的现代 Web 开发框架总称，它是 MongoDB + Express +AngularJS + NodeJS 四个框架的第一个字母组合。它与传统 LAMP 一样是一种全套开发工具的简称。在 2014 和 2015 年喜欢讲这个，并且还有 MEAN.js 等框架，但今天已经过时，Node.js 有了更多的应用场景。

《Node.js in action》一书里说，Node 所针对的应用程序有一个专门的简称：DIRT。它表示数据密集型实时（data-intensive real-time）程序。因为 Node 自身在 I/O 上非常轻量，它善于将数据从一个管道混排或代理到另一个管道上，这能在处理大量请求时持有很多开放的连接，并且只占用一小部分内存。它的设计目标是保证响应能力，跟浏览器一样。

这话不假，但在今天来看，DIRT 还是范围小了。其实 DIRT 本质上说的 I/O 处理的都算，但随着大前端的发展，Node.js 已经不再只是 I/O 处理相关，而是更加的“Node”！

这里给出Node.js的若干使用场景：

- 网站（如express/koa等）
- IM即时聊天(socket.io)
- API（移动端，pc，h5）
- HTTP Proxy（淘宝、Qunar、腾讯、百度都有）
- 前端构建工具(grunt/gulp/bower/webpack/fis3…)
- 写操作系统（NodeOS）
- 跨平台打包工具（PC端的electron、nw.js，比如钉钉PC客户

端、微信小程序IDE、微信客户端，移动的cordova，即老的Phonegap，还有更加有名的一站式开发框架ionicframework）
- 命令行工具（比如cordova、shell.js）
- 反向代理（比如anyproxy，node-http-proxy）
- 编辑器Atom、VSCode等

可以说目前大家能够看到的、用到的软件都有Node.js身影，当下最流行的软件写法也大都是基于Node.js的，比如PC客户端luin/medis采用electron打包，写法采用React+Redux。我自己一直的实践的【Node全栈】，也正是基于这种趋势而形成的。在未来，Node.js的应用场景会更加的广泛。更多参见sindresorhus/awesome-nodejs。

Web框架

演进时间线大致如下：
- 2010年tj写的Express；
- 2011年Derby.js开始开发，8月5日，WalmartLabs的一位成员Eran Hammer提交了Hapi的第一次commit。Hapi原本是Postmile的一部分，并且最开始是基于Express构建的。后来它发展成自己自己的框架；
- 2012年1月21日，专注于rest api的restify发布1.0版本，同构的meteor开始投入开发，最像rails的sails也开始了开发；
- 2013年tj开始玩generator，编写co这个generator执行器，并开始了Koa。2013 年下半年李成银开始ThinkJS，参考ThinkPHP；
- 2014年，4月9日，express发布4.0，进入4.x时代持续到今天，MEAN.js开始随着MEAN架构的提出开始开发，意图大一统，另外total.js开始，最像PHP’s Laravel 或Python’s Django 或 ASP.NET MVC的框架；
- 2015年8月22日， 下一代Web框架Koa发布1.0，可以在node 0.12下面，通过co + generator实现同步逻辑，那时候co还是基于thunkfy的，2015.10.30 ThinkJS发布了Es2015+ 特性开发的v 2.0版本；
- 2016 年 09 月，蚂蚁金服的eggjs，在 JSConf China 2016 上亮相并宣布开源；
- 2017年2月，下一代Web框架Koa发布2.0。

我们可以根据框架的特性进行分类（见表2）。

对于框架选型：
- 业务场景、特点，不必为了什么而什么，避免本末倒置
- 自身团队能力、喜好，有时候技术选型决定团队氛围的，需要平衡激进与稳定
- 出现问题的时候，有人能Cover的住，Node.js虽然8年历史，但模块

框架名称	特性	点评
Express	简单、实用，路由中间件等五脏俱全	最著名的Web框架
Derby.js && Meteor	同构	前后端都放到一起，模糊了开发便捷，看上去更简单，实际上上对开发来说要求更高
Sails、Total	面向其他语言，Ruby、PHP等	借鉴业界优秀实现，也是Node.js成熟的一个标志
MEAN.js	面向架构	类似于脚手架，又期望同构，结果只是蹭了热点
Hapi和Restfy	面向Api && 微服务	移动互联网时代Api的作用被放大，故而独立分类。尤其是对于微服务开发更是利器
ThinkJS	面向新特性	借鉴ThinkPHP，并慢慢走出自己的一条路，对于Async函数等新特性支持，无出其右
Koa	专注于异步流程改进	下一代Web框架

表 2

完善程度良莠不齐，如果不慎踩到一个坑里，需要团队在无外力的情况能够搞定，否则会影响进度

个人学习求新，企业架构求稳，无非喜好与场景而已。

我猜大家能够想到的场景，大约如下：

- 前端工具，比如gulp、grunt、webpack等；
- 服务器，做类似于Java、PHP的事儿。

如果只是做这些，和 Java、PHP 等就没啥区别了。如果再冠上更了不起的 Node.js，就有点名不符实了。所以这里我稍加整理，看看和大家想的是否一样：

技术栈演进

自从 ES 2015（俗称 ES 6）在 Node.js 落地之后，整个 Node.js 开发都发生了翻天覆地的变化。自从 0.10 开始，Node.js 就逐渐的加入了 ES 6 特性，比如 0.12 就可以使用 generator，才导致寻求异步流程控制的 tj 写出了 co 这个著名的模块，继而诞生了 Koa 框架。但是在 4.0 之前，一直都是要通过 flag 才能开启 generator 支持，故而 Koa 1.0 迟迟未发布，在 Node 4.0 发布才发布的 Koa 1.0。

2015 年，成熟的传统，而 2016 年，变革开始。

核心变更：ES 语法支持：

- 使用Node.js 4.x或5.x里的ES6特性，如果想玩更高级的，可以使用Babel编译支持ES7特性，或者TypeScript。
- 合理使用standard 或者 xo 代码风格约定。
- 适当的引入ES 6语法，只要Node.js SDK支持的，都可以使用。
- 需要大家重视OO（面向对象）写法的学习和使用，虽然ES 6的OO机制不健全，但这是大方向，以后会一直增强。OO对于大型软件开发更好。这其实也是我看好typescript的原因。

对比一下变革前后的技术栈选型，希望读者能够从中感受到其中的变化（见表 3）。

预处理器

前端预处理可分 3 种：

- 模板引擎
- css预处理器
- js友好语言

这些都离不开 Node.js 的支持，对于前端工程师来说，使用 Node.js 来实现这些是最方便不过的（见表 3）。

跨平台

跨平台指的是 PC 端、移动端、Web/H5（见表 4）。

构建工具

说起构建工具，大概会想到 make、ant、rake、gradle 等，其实 Node.js

名称	实现	描述
模板引擎	art\mustache\ejs\hbs\jade…	上百种之多，自定义默认，编译成html，继而完成更多操作
css预处理器	less\sass\scss\rework\postcss	自定义语法规则，编译成css
js友好语言	coffeescript、typescript	自定义语法规则、编译成js

平台	实现	点评
Web/H5	纯前端	不必解释
PC客户端	nw.js和electron	尤其是atom和vscode编辑器最为著名，像钉钉PC端，微信客户端、微信小程序IDE等都是这样的，通过web技术来打包成PC客户端
移动端	cordova（旧称PhoneGap），基于cordova的ionicframework	这种采用h5开发，打包成ipa或apk的应用，称为Hybrid开发（混搭），通过webview实现所谓的跨平台，应用的还是非常广泛的

表 3、表 4

里有更多实现（见表 5）

构建工具都不会特别复杂，所以 Node.js 世界里有非常多的实现，还有人写过 node 版本的 make 呢，玩的很嗨。

HTTP Proxy

- 请求代理
- SSR && PWA
- Api Proxy

1）请求代理

对于 http 请求复杂定制的时候，你是需要让 Node.js 来帮你的，比如为了兼容一个历史遗留需求，在访问某个 CSS 的时候必须提供 HEADER 才可以，如果放到静态 server 或 cdn 上是做不到的。

2）SSR && PWA

SSR 是服务器端渲染，PWA 是渐进式 Web 应用，都是今年最火的技术。如果大家用过，一定对 Node.js 不陌生。比如 React、Vuejs 都是 Node.js 实现的 ssr。至于 pwa 的 service-worker 也是 Node.js 实现的。那么为啥不用其他语言实现呢？不是其他语言不能实现，而是使用 Node.js 简单、方便、学习成本低，轻松获得高性能，如果用其他语言，我至少还得装环境

3）Api Proxy

产品需要应变，后端不好变，一变就要设计到数据库、存储等，可能引发事故。而在前端相对更容易，前端只负责组装服务，而非真正对数据库进行变动，所以只要服务 api 粒度合适，在前端来处理是更好的。

Api的问题

- 一个页面的Api非常多。
- 跨域，Api转发。
- Api返回的数据对前端不友好，后端讨厌（应付）前端，几种api都懒得根据ui/ue去定制，能偷懒就偷懒
- 需求决定Api，Api不一定给的及时。

所以，在前端渲染之余，加一层 Api Proxy 是非常必要的。淘宝早起曾

名称	介绍	点评
jake	基于coffeescript的大概都熟悉这个，和make、rake类似	经典传统
grunt	dsl风格的早期著名框架	配置非常麻烦
gulp	流式构建，不会产生中间文件，利用Stream机制，处理大文件和内存有优势，配置简单，只有懂点js就能搞定	grunt的替代品
webpack + npm scripts	说是构建工具有点过，但二者组合勉强算吧，loader和plugin机制还是非常强大的	流行而已

表 5

公开过一张架构图，在今天看来，依然不过时（见图 3）。

- 左侧半边，浏览器和Node.js Server通信可以有多种协议，HTML、RESTfull、BigPipe、Comet、Socket等，已经足够我们完成任何想做的事儿了。
- 右侧半边，是Node.js实现的WebServer，Node服务分了2个部分。
  - 常规的Http服务，即大块部分二；
  - ModelProxy指的是根据Server端的服务，组成并转化成自身的Model层。磨蹭用于为Http服务提供更好的接口。

这里的 Model Proxy 其实就是我们所说的 Api Proxy，这张图里只是说了结果，把聚合的服务转成模型，继而为 HTTP 服务提供 Api。

下面我们再深化一下 Api Proxy 的概念（见图 4）。

- 这里的Node Proxy做了2件事儿，Api和渲染辅助。
- 前端的异步ajax请求，可以直接访问Api。

如果是直接渲染或者 bigpipe 等协议的，需要在服务器端组装 api，然后再返回给浏览器。

所以 Api 后面还有一个服务组装，在微服务架构流行的今天，这种服务组装放到 Node Proxy 里的好处尤其明显。既可以提高前端开发效率，又可以让后端更加专注于服务开发。甚至如果前端团队足够大，可以在前端建一个 Api 小组，专门做服务集成的事儿。

Api服务

说完了 Proxy，我们再看看利益问

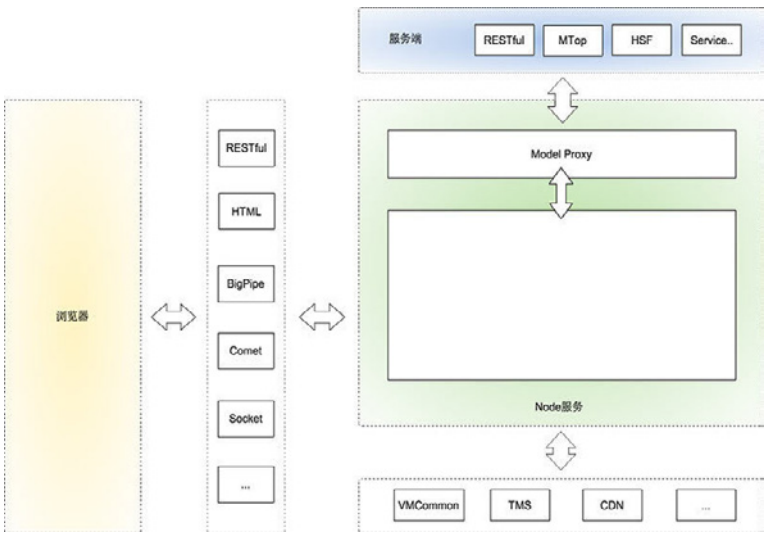

图 3

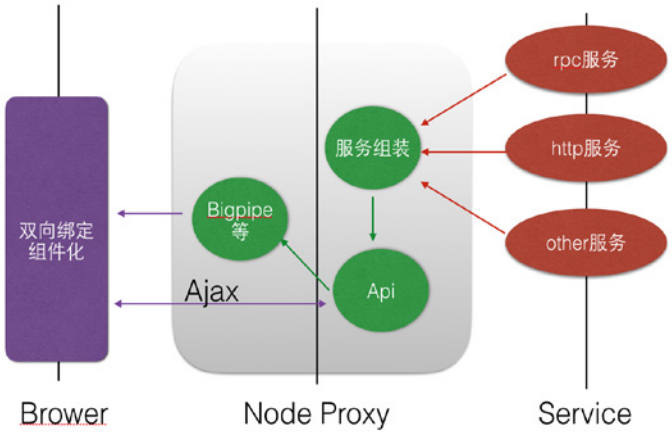

图 4

题。Node.js 向后端延伸，必然会触动后端开发的利益。那么 Proxy 层的事儿，前后端矛盾的交界处，后端不想变，前端又求变，那么长此以往，Api 接口会变得越来越恶心。后端是愿意把 Api 的事儿叫前端的，对后端来说，只要你不动我的数据库和服务就可以。

但是 Node.js 能不能做这部分呢？答案是能的 ，这个是和 Java、PHP 类似的，一般是和数据库连接到一起，处理带有业务逻辑的。目前国内大部分都是以 Java、PHP 等为主，所以要想吃到这部分并不容易。

- 小公司，创业公司，新孵化的项目更倾向于Node.js，简单，快速，高效；
- 微服务架构下的某些服务，使用Node.js开发，是比较合理的。

国内这部分一直没有做的很好，所以 Node.js 在大公司还没有很好的被应用，安全问题、生态问题、历史遗留问题等，还有很多人对 Node.js 的误解。

- 单线程很脆弱，这是事实，但单线程不等于不能多核并发，而且你还有集群呢。
- 运维，其实很简单，比其他语言之简单，日志采集、监控也非常简单
- 模块稳定性，对于Mongodb、MySQL、Redis等还是相当不错，但其他的数据库支持可能没那么好。
- 安全问题。

这些对于提供 Api 服务来说已经足够了。

其他

详见表 6。

Async函数与Promise

- Async函数是趋势，Chrome 52. v8 5.1已经支持Async函数(https://github.com/nodejs/CTC/issues/7)

用途	说明	前景
爬虫	抢了不少Python的份额，整体来说简单，实用	看涨
命令行工具	写工具、提高效率，node+npm真是无出其右	看涨
微服务与RPC	Node做纯后端不好做，但在新项目和微服务架构下，必有一席之地	看涨
微信公众号开发	已经火了2年多了，尤其是付费阅读领域，还会继续火下去，gitchat就是实用Node.js做的，而且还在招人	看涨
反向代理	Node.js可以作为nginx这样的反向代理，虽然线上我们很少这样做，但它确确实实可以这样做。比如node-http-proxy和anyproxy等，其实使用Node.js做这种请求转发是非常简单的	看涨

表 6

了，Node.js 7.0+支持还会远么？

- Async和Generator函数里都支持promise，所以promise是必须会的。
- Generator和yield异常强大，不过不会成为主流，所以学会基本用法和promise就好了，没必要所有的都必须会。
- co作为Generator执行器是不错的，它更好的是当做Promise 包装器，通过Generator支持yieldable，最后返回Promise，是不是有点无耻？

我整理了一张图，更直观一些（见图 5）。

- 红色代表Promise，是使用最多的，无论async还是generator都可用；
- 蓝色是Generator，过度货；
- 绿色是Async函数，趋势。

结论：Promise 是必须会的，那你为什么不顺势而为呢？

推荐：使用 Async 函数 + Promise 组合，如图 6 所示。

实践

合理的结合 Promise 和 Async 函数是可以非常高效的，但也要因场景而异：

- Promise更容易做promisefyAll（比如使用bluebird）；
- Async函数无法批量操作。

那么，在常见的 Web 应用里，我们总结的实践是，dao 层使用 Promise 比较好，而 service 层，使用 Async/Await 更好。

dao 层使用 Promise：

- crud
- 单一模型的方法多
- 库自身支持Promise

这种用 promisefyAll 基本几行代码就够了，一般单一模型的操作，不会特别复杂，应变的需求基本不大。

而 service 层一般是多个 Model 组合操作，多模型操作就可以拆分成多个小的操作，然后使用 Await 来组合，看

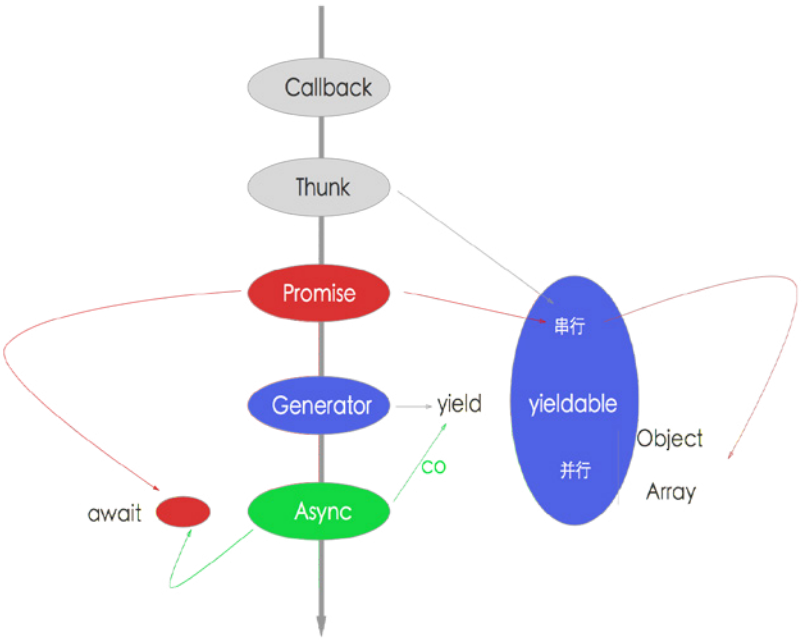

图 5

起来会更加清晰，另外对需求应变也是非常容易的。

ES.next

Node.js + ES.next = ♥

Flow && TypeScript

Type Systems Will Make You a Better JavaScript Developer.

ES6模块

现在ES6自带了模块标准，也是JS第一次支持module（之前的CommonJS、AMD、CMD都不算），但目前的所有Node.js版本都没有支持，目前只能用用Traceur、BabelJS，或者TypeScript把ES6代码转化为兼容ES5版本的js代码，ES6模块新特性非常吸引人，下面简要说明。

ES6模块的目标是创建一个同时兼容CommonJS和AMD的格式，语法更加紧凑，通过编译时加载，使得编译时就能确定模块的依赖关系，效率要比CommonJS模块的加载方式高。而对于异步加载和配置模块加载方面，则借鉴AMD规范，其效率、灵活程度都远远好于CommonJS写法。

- 语法更紧凑。
- 结构更适于静态编译（比如静态类型检查，优化等）。
- 对于循环引用支持更好。

ES6模块标准只有2部分，它的用法更简单，你根本不需要关注实现细节：

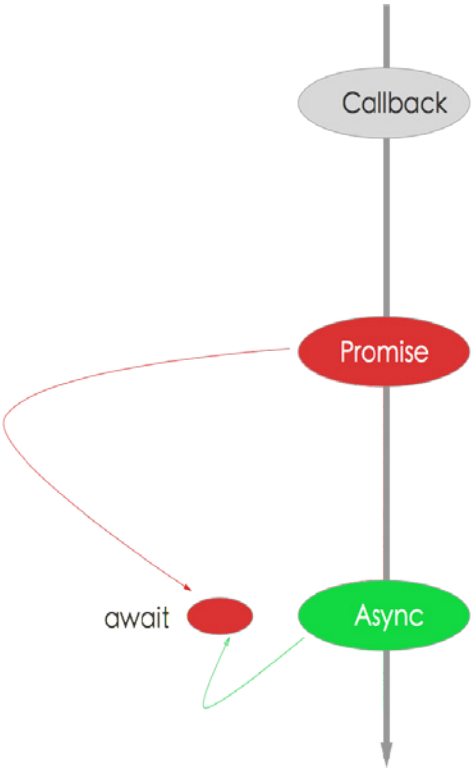

图 6

- 声明式语法：模块导入import、导出export，没有require了；
- 程式化加载API：可以配置模块是如何加载，以及按需加载。

多模块管理器：Lerna

A tool for managing JavaScript projects with multiple packages.

https://lernajs.io/

在设计框架的时候，经常做的事儿是进行模块拆分，继而提供插件或集成机制，这样是非常好的做法。但问题也随之而来，当你的模块模块非常多时，你该如何管理你的模块呢？

- 法1：每个模块都建立独立的仓库
- 法2：所有模块都放到1个仓库里

法 1 虽然看起来干净，但模块多时，依赖安装，不同版本兼容等，会导致模块间依赖混乱，出现非常多的重复依赖，极其容易造成版本问题。这时法 2 就显得更加有效，对于测试，代码管理，发布等，都可以做到更好的支持。

Lerna 就是基于这种初衷而产生的专门用于管理 Node.js 多模块的工具，当然，前提是你有很多模块需要管理。

你可以通过 npm 全局模块来安装 Lerna，官方推荐直接使用 Lerna 2.x 版本。

更好的NPM替代品：Yarn

Yarn是开源JavaScript包管理器，由于npm在扩展内部使用时遇到了大小、性能和安全等问题，Facebook携手来自Exponent、Google和Tilde的工程师，在大型JavaScript框架上打造和测试了Yarn，以便其尽可能适用于多人开发。Yarn承诺比各大流行npm包的安装更可靠，且速度更快。根据你所选的工作包的不同，Yarn可以将安装时间从数分钟减少至几秒钟。Yarn还兼容npm注册表，但包安装方法有所区别。其使用了lockfiles和一个决定性安装算法，能够为参与一个项目的所有用户维持相同的节点模块（node_modules）目录结构，有助于减少难以追踪的bug和在多台机器上复制。

Yarn还致力于让安装更快速可靠，支持缓存下载的每一个包和并行操作，允许在没有互联网连接的情况下安装（如果此前有安装过的话）。此外，Yarn承诺同时兼容npm和Bower工作流，让你限制安装模块的授权许可。

2016年10月份，Yarn在横空出世不到一周的时间里，github上的star数已经过万，可以看出大厂及社区的活跃度，以及解决问题的诚意，大概无出其右了！

替换的原因：

在Facebook的大规模npm都工作的不太好；

npm拖慢了公司的ci工作流；

对一个检查所有的模块也是相当低效的；

npm被设计为是不确定性的，而Facebook工程师需要为他们的DevOps工作流提供一直和可依赖的系统。

与hack npm限制的做法相反，Facebook编写了Yarn：

- Yarn的本地缓存文件做的更好；
- Yarn可以并行它的一些操作，这加速了对新模块的安装处理；
- Yarn使用lockfiles，并用确定的算法来创建一个所有跨机器上都一样的文件。

出于安全考虑，在安装进程里，Yarn不允许编写包的开发者去执行其他代码。

> Yarn, which promises to even give developers that don’t work at Facebook’s scale a major performance boost, still uses the npm registry and is essentially a drop-in replacement for the npm client.

很多人说和ruby的gem机制类似，都生成lockfile。确实是一个很不错的改进，在速度上有很大改进，配置cnpm等国内源来用，还是相当爽的。

友好语言

- 过气的Coffeescript，不多说
- Babel：also an ES6 to ES5 transpiler that’s growing in popularity possibly because it also supports React’s JSX

syntax. As of today it supports the most ES6 features at a somewhat respectable 73%.

- TypeScript：a typed superset of JavaScript that not only compiles ES6 to ES5 (or even ES3) but also supports optional variable typing. TypeScript only supports 53% of ES6 features.

总结

坦诚的力量是无穷的。

Node.js 是为异步而生的，它自己把复杂的事儿做了（高并发，低延时），交给用户的只是有点难用的 Callback 写法。也正是坦诚的将异步回调暴露出来，才有更好的流程控制方面的演进。也正是这些演进，让 Node.js 从 DIRT（数据敏感实时应用）扩展到更多的应用场景，今天的 Node.js 已经不只是能写后端的 JavaScript，已经涵盖了所有涉及到开发的各个方面，而 Node 全栈更是热门种的热门。

直面问题才能有更好的解决方式，Node.js 你值得拥有！

大前端公共知识杂谈

作者 王下邀月熊

近年来，随着移动化联网浪潮的汹涌而来与浏览器性能的提升，iOS、Android、Web 等前端开发技术各领风骚，大前端的概念也日渐成为某种共识。

其中特别是 Web 开发的领域，以单页应用为代表的富客户端应用迅速流行，各种框架理念争妍斗艳，百花竞放。Web 技术的蓬勃发展也催生了一系列跨端混合开发技术，希望能够结合 Web 的开发便捷性与原生应用的高性能性；其中以 Cordova、PWA 为代表的方向致力于为 Web 应用尽可能添加原生体验，而 以 NativeScript、ReactNative、Weex 为代表的利用 Web 技术或者理念开发原生应用。

平心而论，无论哪一种开发领域或者技术，他们本质上都是进行图形用户界面（GUI）应用程序的开发，面对的问题、思考的方式、架构的设计很大程度上仍然可以回溯到当年以 MFC、Swing、WPF 为主导的桌面应用程序开发时代，其术不同而道相似。

任何的前端开发学习中，我们都需要掌握基本的编程语言语法与接口；譬如在 Android 开发中使用的 Java 或者 Kotlin，在 iOS 开发中使用的 Objective-C 或者 Swift，在 Web 开发中使用的 JavaScript、HTML 与 CSS 等。

编程语言的学习中我们往往关注于语法基础、数据结构、功能调用、泛型编程、元编程等内容，譬如如何声明表达式、如何理解作用域与闭包、如何进行基本的流程控制与异常处理、如何实践面向对象编程、如何进行网络请求通信等等。

接下来我们就需要了解如何构建基础的界面，譬如利用 HTML 与 CSS 绘制简单 Web 页面、利用代码创建并使用简单的 Activity、利用 StoryBoard 快速构建界面原型等等。

然后我们需要去学习使用常见的系统功能，譬如如何进行网络交互，如何访问远端的 RESTful 接口以获取需要的数据、如何读取本地文件或者利用 SharedPreference、localStorage、CoreData 来存取数据、如何进行组件间或者应用间信息交互等内容。

到这里我们已经能够进行基础的界面开发，并且为其增添必要的特性，不过在真实的项目中我们往往还会用到很多的组件或者插件，iOS 或者 Android 中为我们提供了丰富的 SDK，譬如 UITableView 或者 RecycleView 可以帮助我们快速构建高性能列表组件，Android 5.0 之后默认的 Material Design 也是非常优秀的界面样式设计指南；而 Web 开发中我们往往需要引入第三方模式库，譬如著名的 BootStrap、React Material UI、Vue element 都为我们提供了很多预置的样式组件，react-virtualized 也为我们提供了高性能的类似于 ListView 这样的部分项渲染机制。

然后我们需要将应用真实地发布给用户使用，我们需要考虑很多工程实践的问题，譬如如何进行测试与调试、如何进行性能优化并且在生产环境下完成应用状态跟踪、热更新等操作、如何统一开发团队的代码风格与约定等等；这里 Web 因为其特性而自带了热更新的功能，而在 Android 或者 iOS 我们则可以利用插件化技术或者 JSPatch 来实现热更新。

Java 与 Swift 都是强类型语言，其能够在编译阶段帮开发者排查问题减少潜在风险；而我们也可以使用 TypeScript 或者 Flow 为 JavaScript 添加静态类型检测的特性，在 VSCode 等现代编辑器中同样可以达到类似于 Android Studio、XCode 中的即时检查与提示的功能。

最后，随着应用功能的增加、代码库的扩展，我们需要考虑整体的应用架构与工程化的问题；在应用架构中我们往往需要考虑模块化、组件化以及状态管理等多个方面，选择合适的 MVC、MVP、MVVM、Flux、VIPER 等不同的架构模式来引导应用中的代码组织与职责分割；我们也需要考虑选择合适的构建与部署工具来简化或者自动化应用发布流程，在 Android 开发中我们会选择 Gradle 及其自带的多模块特性来管理依赖与分割代码，而在 Web 中我们可以使用 Webpack、Rollup 等工具来自

动处理依赖并且进行构建，iOS 中我们也可以选择 CoocaPods。

到这里我们会发现虽然具体的代码实现、使用的技术不同，但是 Android、iOS 以及 Web 乃至于 React Native 等开发中，我们需要解决的问题、能够用到的架构设计模式都是可以相互借鉴的。

在我们从某个领域迁移到其他领域时，我们能很方便地知道应该学习些什么，不同的技术、工具他们的职责是什么，应该选择怎样的架构或者设计模式。古语云，欲穷千里目，更上一层楼，我们想要真正掌握某种客户端开发技术，最好是要了解我们应该掌握那些方面。

编程语言

编程语言的学习是我们进入软件世界的基础阶梯，著名的 Code Complete 一书中提到：Program into Your Language, Not in it.

我们不应该将自己的编程思维局限于掌握的语言提供的那些特性或者概念，而是能够理解这些语法特性背后能提供的抽象功能与原理，从而能够根据自己想要达到的目标选择最合适的编程语言。

而从另一个角度来看，无论哪一门编程语言的学习也是具有极大的共性，从严谨而又被诟病过度冗余的 Java 到需要用游标卡尺的 Python，从挣扎着一路向前的 JavaScript 到含着金汤匙出生的 Swift、Rust，我们都能够发现其中的相通与互相借鉴之处。

语法基础

任何一门编程语言的学习都需要从基本的表达式（Expression）语法开始学习，我们需要了解如何去声明与使用变量、如何为这些变量赋值、如何使用运算符进行简单的变量操作等等。

在很多语言之中都有所谓的传值还是传引用的思量，譬如 Java 与 JavaScript 本质上就是 Pass-by-Value 的语言，只不过会将复杂对象的引用值传递给目标变量。这个特性又引发了所谓浅复制与深复制、如何进行复合类型深拷贝等等需要注意的技术点。

除此之外，作用域与闭包也是很多语言学习中重点讨论的内容，在 JavaScript 与 Python 的学习中我们就会经常讨论如何利用闭包来保存外部变量，或者在循环中避免闭包带来的意外变量值。

表达式是一门编程语言语法基础的重要组成部分，接下来我们就需要去学习流程控制与异常处理、函数定义与调用、类与对象、输入输出流、模块等内容。

流程控制的典型代表就是分支选择与循环，譬如不同的语言都为我们提供了基础 for 循环或者更方便地 for-in 循环，而在 JavaScript 中我们还可以使用 forEach 与 for-of 循环，Java 8 之后我们也可以基于 Stream API 中的 forEach 编写声明式地循环执行体，而 Python 中的列表推导也可以看做便

捷的循环实现方式。

异常处理也是各个编程语言的重要组成部分，合理的异常处理有助于增强应用的鲁棒性；不过很多时候会出现滥用异常的情况，我们只是一层一层地抛出而并未真正地去处理或者利用这些异常。Java 中将异常分为了受控异常与不受控异常这两类，虽然 JavaScript 等语言中并未在数据类型中有所区分，但是却可以引入这种分类方式来进行不同的异常处理；有时候 Let it Crash 也是不错的设计模式。

Eric Elliott 曾在博文中提及，软件开发实际就是 Function Composition 与 DataStructure Design；函数或者方法是软件系统的重要基石与组成。我们需要了解如何去定义函数，包括匿名函数以及 Lambda 表达式等；尽管 Java 中的 Lambda 表达式是对于 FunctionalInterface 的实现，但是鉴于其表现形式我们也可以将其划归到函数这个知识类别中。

接下来我们需要了解如何定义与传入函数参数，在 C 这样的语言中我们会去关心指针传递的不同姿势；而在 JavaScript 中我们常常会关心如何设置默认参数，无论是使用对象解构还是可选参数，都各有利弊。

Objective-C 与 Swift 中提供的外部参数就是不错的函数自描述，Java 或 Python 中提供的不定参数也能够帮我们更灵活地定义参数，在 JavaScript 中我们则可以通过扩展操作符实现类似的效果。

然后我们就需要去考虑如何调用函数，最典型就是就是 JavaScript 中函数调用的四种方式，我们还需要去关心调用时函数内部的 this 指针指向。

而装饰器或者注解能帮我们更好地组织代码，以类似于高阶函数的方式如洋葱圈般一层一层地剥离与抽象业务逻辑。最后在函数这部分我们还需要关心下迭代器与生成器，它们是不错的异步实现模式或者流数据构建工具。

近几年随着前端富客户端应用的迅猛发展与服务端并发编程的深入应用，函数式编程以及 Haskell 这样的函数式编程语言也是引领风骚。

尽管面向对象编程也有着很多其他被人诟病的地方，但是在大型复杂业务逻辑的应用开发中我们还是会倾向使用面向对象编程的范式；这就要求我们对于类与对象的基本语法有所掌握。

我们首先要去了解如何定义类，定义类的属性、方法以及使用访问修饰符等方式进行访问控制。其次我们需要了解如何从类中实例化出对象，如何在具体的语言中实践单例模式等。

然后我们就需要去了解面向对象的继承与多态的特性，应该如何实现类继承，子类与父类在静态属性、静态方法、类属性、构造函数上的调用顺序是怎样的；以及如何利用纯虚函数、抽象类、接口、协议这些不同的关键字在具体语言中实现多态与约定。

最后我们还需要去关注下语言是否

支持内部类，譬如 Java 就分为了静态内部类、成员内部类、局部内部类与匿名内部类这四种不同的分类。

在整个语法基础部分的最后，我们还需要去了解下输入输出流与模块化相关的知识，譬如 Java 9 中即将推出 JPMS 模块化系统，而 JavaScript 的模块化标准则历经了 CommonJS、AMD、UMD、ES6 Modules 等多轮变迁。

数据结构与功能

语法基础是我们掌握某门编程语言的敲门砖，而学习内建的数据结构与功能语法则是能够用该语言进行实际应用开发的重要前提。

在数据结构的学习中我们首先要对该语言内建的数据类型有所概览，我们要了解如何进行常见的类型与值判断以及类型间转换；譬如如何进行引用与值的等价性判断、如何进行动态类型检查、如何对复合对象的常用属性进行判断等等。

很多编程语言中会将数据类型划分为原始类型（Primitive）与复合类型（Composite），不过这里为了保证通用性还是将学习复杂度较低的数据类型划归到基本类型中。常见的基本类型囊括了数值类型、空类型、布尔类型、可选类型(Optional)以及枚举类型(Enum)等等。

学习数值类型的时候我们还需要了解如何进行随机数生成、如何进行常见的科学计算，这也是基础的数值理论算法的重要组成。JavaScript 中提供了 undefined 与 null 两个关键字，二者都可以认为是空类型不过又有所区别；而可选类型则能够帮我们更好地处理可能为空地对象，避免很多的运行时错误。

接下来我们就要将目光投注于字符串类型上，我们需要了解如何创建、删除、复制、替换某个字符串或者其他内容；很多语言也提供了模板字符串或者格式化字符串的方式来创建新字符串。

我们还需要知道如何对字符串进行索引遍历，如何对字符串进行常见的类型编码以及如何实践模式匹配。模式匹配中最直接的方式就是使用正则表达式，这也是我们应用开发中经常会使用到的技术点。

除此之外我们还需要关注字符串校验、以及如何进行高效模糊搜索等等内容；我们也可以学习使用 KMP、Sunday 等常见的模式匹配算法来处理搜索问题。

然后我们需要学习常见的时间与日期处理方式，了解如何时间戳、时区、RFC2822、ISO8601 这些基础的时间与日期相关的概念，了解如何从时间戳或者时间字符串中解析出当前编程语言支持的时间与日期对象。

我们还需要了解时区转换、时间比较以及如何格式化地展示时间等内容，有时候我们还需要利用日历等对象进行事件的增减以及偏移计算。

接下来就是非常重要的集合类型，无论哪种编程语言都会提供类似于

Array、List、Set、Dictionary、Map 等相关的数据结构实现，而我们也就需要去了解这些常见集合类型中的增删复替以及索引遍历这些基础操作以及每个集合的特点；譬如对于序列类型我们要能熟练使用 map、reduce、filter、sort 这些常见的变换进行序列变换与生成。

进阶而言的话我们可以多了解下这些数据结构的底层算法实现，譬如 Java 8 中对于 HashMap 的链表 / 红黑树实现，或者 V8 中是如何利用 Hidden Class 进行快速索引的。

接下来的话我们可以对于像 Java 中 SteamAPI 或者各种语言的 Immutable 对象的实现方式有所了解，还有就是常见的 JSON、XML、CSV 这些类型的序列化与反序列化操作库也是实际开发中经常用到的。

接下来我们就需要对语言提供的常用外部功能相关的 API 或者语法有所掌握，主要也是分为存储、网络与系统进程这三个部分。在存储部分我们需要掌握如何与 MySQL、Redis、Mongodb 等关系型或者非关系型数据库进行数据交互，掌握如何对文件系统进行如文件寻址、文件监控等操作，并且还需要能够使用一些譬如 Java 堆外存储这样的应用内缓存来存放数据。

而网络部分我们应该掌握如何利用 HTTP 客户端进行网络交互、如何使用相对底层的 Socket 套接字建立 TCP 连接、或者使用语言内置的一些远程调用框架与远端服务进行交互。最后我们需要对如何利用语言进行系统进程操作有所了解，本部分笔者认为最重要的当属并发编程相关知识。

在而今服务器性能不断提升、处理的数据量越来越多的情况下，我们不可避免地需要使用并发操作来提高应用吞

吐量。并发编程领域我们应该去学习如何使用线程、线程池或者协程来实现并发，如何利用锁、事务等方式进行并发控制并保持数据一致性，如何使用回调、Promise、Generator、Async/Await 等异步编程模式。

除此之外，我们还需要对切面编程、系统调用以及本地跨语言调用有所了解。

工程实践与进阶

编程语言初学阶段的最后我们需要了解下工程实践以及一些偏原理与底层实现的进阶内容。

首先开发者应当对具体编程语言中如何实现 S.O.L.I.D 编程原则与数十种设计模式有所了解，当然也不能邯郸学步只求形似，而是能够根据业务功能需求灵活地选择适用的范式。

而在团队开发中我们往往还需要统一团队内的样式指南，包括代码风格约定中常见的命名约定、文档与注释约定、项目与模块的目录架构以及语法检查规范等。

接下来我们还需要对语言或者常用开发工具的调试方式有所了解，掌握基本的单步调试等技巧，并且能够为代码编写合适的单元测试用例。工程实践方面的最后则是要求我们对代码性能优化所有了解，尽量避免反模式。

进阶内容的话则相对更加地抽象或者需要花费更多的精力去学习，其中包括泛型编程、元编程、函数式编程、响应式编程、内存管理、数据结构与算法等几个部分。

泛型编程与元编程中的反射、代码生成、依赖注入等还是属于语言本身提供的语法特性之一，而函数式编程与响应式编程则偏向于实际应用开发中有所偏爱的开发范式。

即使 Java 这样纯粹的面向对象的语言，当我们借鉴纯函数、不可变对象、高阶函数、Monad 等函数式编程中常见的名词时，也能为代码优化开辟新的思路。响应式编程是非常不错的异步编程范式，这里我们还需要注意下并发编程与异步编程之间的差异。

而内存管理则有助于我们理解编程语言运行地底层机制，譬如对于 JVM 或者 V8 的内存结构、内存分配、垃圾回收机制有所了解的话能够反过来有助于我们编写高性能地应用程序，并且对于线上应用错误的调试也能更加得心应手。

界面基础

用户界面是前端应用程序的核心组成部分，而我们涉足前端开发的第一步往往也就是从简单的界面搭建开始。

我们可能是在 Android 中编写简单的基于 XML 布局的 Activity，在 iOS 中利用 StoryBoard 快速构建导航界面，或者在 Web 中使用某个框架实现 TODOList。

界面开发最基础的部分就是布局与定位，无论在何端开发中我们往往都会使用相对布局、绝对布局、弹性布局、网格布局等布局方式；并且面向多尺寸的屏幕我们往往也需要进行响应式布局的考虑，从横竖屏响应式切换到不同分辨率下的布局与尺寸的调整，都是为了给予用户较好的使用体验。

而了解了布局与定位之后，我们往往就需要来学习如何使用基本的界面容器，譬如常见的滚动视图、导航视图、页卡视图与伸缩视图。Android 与 iOS 往往也为我们对这些基本容器进行了较好地封装，而 Web 中则往往需要我们自己动手去实现相应功能。

譬如在滚动视图中，我们需要去提供常见的滚动事件控制，典型的有如何在不同环境下保证平滑滚动体验、如何设置优美的滚动条、如何设置滚动监听等等。

除此之外，我们往往还需要针对列表或者长阅读界面封装一些高级事件响应，譬如上拉加载、下拉刷新以及无限滚动时需要的滚动触发规则实现。作为最常见的用户交互方式之一，无论是在移动端还是桌面端，我们也都需要实现一些优美的动画；譬如视差滚动就可以给用户带来不一样的视觉感受，而像 Swiper 这样的整页滚动则是很好地产品展示或者讲演页的交互方式。

在基础的界面容器使用中我们已经接触了一些用户交互的监听与响应的实现，接下来我们则是需要深入全面地了解用户交互相关内容。最基础的我们需要了解常用事件与手势操作，了解如何进行事件监听与绑定、如何捕获事件并且进行分发、如何进行缩放、拖拽、摇晃等复杂手势动作地监听与识别、如何响应键盘事件并且进行响应处理。

除此之外，笔者将音视频录制与播放，指纹、计步器等传感器的使用，本地通知与远程推送等内容也都归纳于了

用户交互这个分类下。在 Android 与 iOS 开发中相信对于这些 API 的使用并不会陌生，而随着 HTML5 的流行以及现代浏览器的发展，相信未来 Web 应用也会越来越多地添加这些与系统层面进行交互地功能。

我们在本部分还需要了解下动画与变换、绘图及数据可视化等相关内容。常见的动画引擎包含了属性控制与帧动画两种方式，前者更趋向于命令式编程而后者则适用于声明式编程；除了了解这些基础的语法，我们还需要对常用的动画进行收集与汇总，以便在项目开发中能够灵活应用。

而随着大数据时代的到来，数据可视化相关应用也成了前端开发常见的任务之一。在这个部分，我们需要对 SVG、Canvas、WebGL 等相关绘画基础有所了解，能够运用 D3.js 或者其他类似的库进行简单图形绘制。并且我们要能够利用 ECharts 等优秀的外部绘图库进行散点图、折线图、流程图等常见类型图表进行绘制。

最后，地图以及相关技术也是我们需要去了解的，作为开发者我们要能够基于百度地图等第三方 API 或者 SDK 开发导航、地理位置信息可视化等相关的功能。

系统功能

与界面基础相对的就是常见的系统功能以及 API 的使用语法，其主要分为系统与进程、数据存储以及网络交互这三个部分。

进程与存储

在开发多界面应用程序或者利用 Service、ServiceWorker 等方式启动后台线程时，我们就需要考虑如何进行组件间通信；譬如在 Android 开发中我们可以利用 Otto 等库以消息总线的方式在 Activity、Fragment、Service 等组件之间传递消息。

而在 Android 或者 iOS 开发中我们也常常需要考虑并发编程，可能会涉及到如何利用 Thread、GCD 等方式实现多线程并行、如何利用 RxJava 等响应式扩展优化异步编程模型、如何利用锁等同步方式进行并发控制等等内容。

有时候我们也需要去更多地了解系统服务相关的内容，特别是在 Android 或者桌面应用程序开发时，我们需要考虑如何实现守护进程以协调并且保障各个组件的正常运行。

在系统与进程部分的最后，我们还需要去接触些系统辅助相关的功能实现，譬如如何进行运行环境检测、如何利用 DeepLink 进行 APP 之间跳转、如何进行应用的权限管理等等。

接下来我们讨论下数据存储部分应该掌握哪些内容，最简单的就是类似于 SharedPreference、NSUserDefaults、localStorage 这样的键值类存储；复杂一点的情况我们可能会利用到 SQLite 或者 IndexedDB 这样的简化关系型或者文档型数据库，有时候 Realm 这样

的第三方解决方案也是不错的选择。

很多时候我们还需要了解如何控制缓存或者剪贴板中的内容，以及如何对文件系统进行基本的操作，譬如读写配置文件与资源文件、浏览列举文件系统中的文件并且根据不同的文件类型选用不同的处理方式。

网络交互

而网络交互部分更多地关注如何与服务端或者第三方系统进行交互，实际上对于如何在需求动态变化的情况下较好地协调服务端与客户端对于接口的定义是很多项目开发的痛点。

不过从基础使用的角度，我们首先需要了解如何利用网络客户端进行基于 HTTP 或 HTTPS 的网络请求。这部分我们需要了解如何构造、分析、编码 URI，如何管理请求头、设置请求方法与请求参数，如何同步、异步或者并发地执行请求，如何进行响应解析，如何进行复杂的请求管理等等内容。

除了这些，我们还要能够利用基础的 Socket 进行通信，这样有助于我们理解通信网络与 TCP/IP 实现原理；我们往往还需要关心如何利用 WebSocket 等技术实现推送与长连接功能，如何进行远程与本地方法调用等等。

除了这三个偏功能实现的知识点，我们还可以尝试去了解下系统的底层设计原理。譬如在 Android 开发中我们可以尝试去了解 Dalvik 虚拟机的工作原理，使用 Xposed 或类似工具进行系统层面的一些操作；对于 Web 开发而言我们可以去更多地关注浏览器工作原理，了解现代浏览器的运行机制等等内容。

界面插件

在掌握了如何构建基本的界面并且为应用添加必须的功能之后，我们就需要去尝试进行应用项目开发。每个应用可以按照用户交互地逻辑切分为多个独立界面，而每个界面的开发中我们往往又需要编写导航、菜单、列表、表单等等可重复使用的界面插件。

实际上前端开发中最核心的工作之一就是界面插件的开发，好的开发者能够在项目开发中沉淀出可复用的界面插件库；这类可复用的界面插件往往会独立于具体的业务逻辑，其分类自然也应按照显示或者交互逻辑本身，而不应该受制于不同的业务场景。

笔者习惯地会将界面插件区分为指示器（Indicator）、输入器（Picker）、列表与表单（TableGrid）、对话框（Dialog）、画廊（Galley）、WebView 等几个部分。

指示器与输入器

指示器与输入器算是两个宽泛的界面插件分类，最常见的指示器当属文本显示类别的插件，譬如标签。标签多用于表单中的输入域描述、用户引导等场景，而除了文字标签之外我们也会使用图标或者所谓的 Tags。

除此之外我们还会关注于 MarkDown 等富文本的展示、如何针对不同屏幕对页面进行排版与字体设置、如何针对不同地区的用户进行国际化切换、如何为文本添加合适的动画等等方面。

在应用开发中我们也会添加专门的介绍或引导页，一方面引导用户使用，另一方面也可以进行后台资源请求与处理；譬如我们往往会在应用启动时设置闪屏页（Splash），记得最早在 Uber 见到以短视频为背景的闪屏页很有耳目一新的感觉。

除此之外，我们常见的指示器还包括了进度指示与时间指示这两种。在进行数据请求或者数据处理等需要用户等待的场景中，我们往往会给用户以进度条方式地友好反馈，这种进度条就是典型地进度指示。常用的进度条设计有线性进度条、圆形进度条或者固定在页首或者页尾的进度条，有些设计中我们也会以背景投射地方式反馈当前进度，这种方式可能更具有视觉冲击力。

除了进度条之外，无限循环的加载效果、分页器或者步骤跟踪显示器也是常见的进度指示的表现形式之一。

所谓的时间指示即譬如界面上放置的拟物时钟或者电子时钟、常见于社交媒体上的时间轴或者日历效果以及倒计数效果等。

输入器的典型代表则为按钮与文本输入，譬如我们除了常见的 Primary、Secondary 按钮之外，我们可能还会用到悬浮按钮、可扩展的按钮或者在喜欢与点赞时用到的具有一定动画效果的按钮。

文本输入系列的插件中，除了常见的文本框或者富文本编辑器，有时我们也需要去编写具有自动补全或者类似于密码、勾选之类的特殊格式的输入框。

选择器也是我们常用到的输入器之一，譬如开关、单选按钮、勾选按钮、分段输入以及常用于两个列表互选的左右穿梭器等等。除了这些，搜索、菜单、解锁界面也是归属于输入器这个类别中。

列表、画廊与对话框

在这两个大类之外算得上最常用的插件的当属列表、网格与表单这个系列的控件；基本上每个应用都会包含列表或者网格布局，对于海量数据的列表渲染也是前端常见的挑战之一。

Android 中内置的 RecycleView 与 iOS 中内置的 UITableView 都为我们提供了不错的懒加载、局部渲染的功能，而 Web 中我们往往需要自己定制或者寻求第三方库的帮助。

对于列表的交互也是常见问题之一，除了允许用户正常的点击，我们还需要添加左滑右滑时的反馈、可伸缩或者允许排序、拖拽的方式进行交互，有时候还需要为了列表项添加进出时的转场动画，以这种微互动增加整个界面的友好性。

最后我们来聊聊画廊与对话框，画廊最典型的插件就是提供图片或者

视频预览的走马灯效果的轮播插件，笔者也是将图片加载、呈现、处理相关的插件划分到了画廊这一系列插件中。而在端开发中我们常常需要对相册或者缓存中的图片进行浏览，或者将图片以瀑布流的方式呈现给用户，这种性质的插件也应归属到画廊这一类中。对话框的分类则稍显的有些生硬，譬如 ActionSheet、HUD 是系统提供的消息提示性质的插件，这种弹出与显示层自然会划归到对话框这个系列的组件中。

而在 Web 中我们常常需要自定义的模态对话框、覆盖层也属于对话框系列，有时候我们还需要考虑如何为对话框提供拖拽支持，或者在对话框显示和消失之际添加转场动画。

工程化与应用架构

前面我们讨论了开发某个前端应用所需要的必备技能，而在需要持续交付的团队项目开发中，我们还需要考虑很多工程实践相关的方法与技巧。命令式编程到声明式编程的变化，将更多地功能性工作交于框架处理，而开发人员更加地专注于业务逻辑的实现。

工程实践

代码调试是每个程序员都掌握的技能，不过如何较好地调试代码以快速定位错误所在却并不是那么容易。在开发中我们常常需要热加载、增量编译等相关技术来避免过长的等待，而单步调试则能够帮助我们梳理代码逻辑、循序渐进地发现问题所在。

可能 iOS、Android 的开发人员更习惯使用单步调试，而在 Web 或者 Node.js 开发中我们也应适当地多使用 Chrome 等工具进行代码的单步调试；有时候单步调试也是不错的浏览分析第三方源代码库的方式。另一方面，日志无论在开发环境还是生产环境中都能够帮我们记录应用运行状态等信息。

接下来我们还要了解应用开发周期中不同阶段使用的单元测试、集成测试以及端到端测试的具体的实现方式，在团队协同开发中统一代码风格与约定，能够利用多种方式对应用进行性能优化，以及在发布到生产环境之后能够混淆加密、进行应用更新以及应用状态跟踪。

应用架构

所谓架构二字，核心即是对于对于富客户端的代码组织 / 职责划分，从具体的代码分割的角度，即是功能的模块化、界面的组件化、应用状态管理这三个方面。

纵览这十年内的架构模式变迁，大概可以分为 MV* 与 Unidirectional 两大类，而 Clean Architecture 则是以严格的层次划分独辟蹊径。

从笔者的认知来看，从 MVC 到 MVP 的变迁完成了对于 View 与 Model 的解耦合，改进了职责分配与可测试性。而从 MVP 到 MVVM，添加了 View 与 ViewModel 之间的数据绑定，使得 View 完全的无状态化。最后，整个从

MV* 到 Unidirectional 的变迁即是采用了消息队列式的数据流驱动的架构，并且以 Redux 为代表的方案将原本 MV* 中碎片化的状态管理变为了统一的状态管理，保证了状态的有序性与可回溯性。

实际上从 MVC、MVP 到 MVVM，一直围绕的核心问题就是如何分割 ViewLogic 与 View，即如何将负责界面展示的代码与负责业务逻辑的代码进行分割。

所谓分久必合，合久必分，从笔者自我审视的角度，发现很有趣的一点。Android 与 iOS 中都是从早期的用代码进行组件添加与布局到专门的 XML/Nib/StoryBoard 文件进行布局，Android 中的 Annotation/DataBinding、iOS 中的 IBOutlet 更加地保证了 View 与 ViewLogic 的分割，这一点也是从元素操作到以数据流驱动的变迁，我们不需要再去编写大量的 findViewById。而 Web 的趋势正好有点相反，无论是 WebComponent 还是 ReactiveComponent 都是将 ViewLogic 与 View 置于一起，特别是 JSX 的语法将 JavaScript 与 HTML 混搭，颇有几分当年 PHP/JSP 与 HTML 混搭的风味。

从代码组织的角度来看，项目的构建工具与依赖管理工具会深刻地影响到代码组织，这一点在功能的模块化中尤其显著。譬如笔者对于 Android/Java 构建工具的使用变迁经历了从 Eclipse 到 Maven 再到 Gradle，笔者会将不同功能逻辑的代码封装到不同的相对独立的子项目中，这样就保证了子项目与主项目之间的一定隔离，方便了测试与代码维护。

同样的，在 Web 开发中从 AMD/CMD 规范到标准的 ES6 模块与 Webpack 编译打包，也使得代码能够按照功能尽可能地解耦分割与避免冗余编码。而另一方面，依赖管理工具也极大地方便我们使用第三方的代码与发布自定义的依赖项，譬如 Web 中的 NPM 与 Bower，iOS 中的 CocoaPods 都是十分优秀的依赖发布与管理工具，使我们不需要去关心第三方依赖的具体实现细节即能够透明地引入使用。

因此选择合适的项目构建工具与依赖管理工具也是好的 GUI 架构模式的重要因素之一。不过从应用程序架构的角度看，无论我们使用怎样的构建工具，都可以实现或者遵循某种架构模式，笔者认为二者之间也并没有必然的因果关系。而组件即是应用中用户交互界面的部分组成，组件可以通过组合封装成更高级的组件。

组件可以被放入层次化的结构中，即可以是其他组件的父组件也可以是其他组件的子组件。根据上述的组件定义，笔者认为像 Activity 或者 UIViewController 都不能算是组件，而像 ListView 或者 UITableView 可以看做典型的组件。 我们强调的是界面组件的 Composable&Reusable，即可

组合性与可重用性。

当我们一开始接触到 Android 或者 iOS 时，因为本身 SDK 的完善度与规范度较高，我们能够很多使用封装程度较高的组件；凡事都有双面性，这种较高程度的封装与规范统一的 API 方便了我们的开发，但是也限制了我们自定义的能力。

同样的，因为 SDK 的限制，真正意义上可复用 / 组合的组件也是不多，譬如你不能将两个 ListView 再组合成一个新的 ListView。在 React 中有所谓的 controller-view 的概念，即意味着某个 React 组件同时担负起 MVC 中 Controller 与 View 的责任，也就是 JSX 这种将负责 ViewLogic 的 JavaScript 代码与负责模板的 HTML 混编的方式。

界面的组件化还包括一个重要的点就是路由，譬如 Android 中的 AndRouter、iOS 中的 JLRoutes 都是集中式路由的解决方案，不过集中式路由在 Android 或者 iOS 中并没有大规模推广。iOS 中的 StoryBoard 倒是类似于一种集中式路由的方案，不过更偏向于以 UI 设计为核心。笔者认为这一点可能是因为 Android 或者 iOS 本身所有的代码都是存放于客户端本身，而 Web 中较传统的多页应用方式还需要用户跳转页面重新加载，而后在单页流行之后即不存在页面级别的跳转，因此在 Web 单页应用中集中式路由较为流行而 Android、iOS 中反而不流行。

所谓可变的与不可预测的状态时软件开发中的万恶之源，我们尽可能地希望组件的无状态性，那么整个应用中的状态管理应该尽量地放置在所谓 High-Order Component 或者 Smart Component 中。在 React 以及 Flux 的概念流行之后，Stateless Component 的概念深入人心，不过其实对于 MVVM 中的 View，也是无状态的 View。通过双向数据绑定将界面上的某个元素与 ViewModel 中的变量相关联，笔者认为很类似于 HOC 模式中的 Container 与 Component 之间的关联。随着应用的界面与功能的扩展，状态管理会变得愈发混乱。

因为时间和篇幅的原因，还有更多的内容不能详细展开，可以到这里查看脑图版本：https://www.processon.com/view/link/5858fa8fe4b0db9f2e0f548e

Conversational UI：AI时代的人机交互模式

作者 徐川

2017 年，AI 绝对是热点中的热点，各大互联网公司都将 AI 作为自己的战略重心，有些甚至是战略核心地位。吴恩达说：『人工智能是新电能』，Google 已在去年从 Mobile first 转型到 AI first，AI 成为热点是毫无疑问的了。

那么问题来了，我们应该如何与具有智能的机器交互？

答案就是 Conversational UI（对话式交互）。

在个人计算机诞生不久的蛮荒时期，人们与机器是通过命令行来交互，叫做 Command UI，其实 Conversational UI 光从表面来看与它很像，也是输入 - 输出的循环，不过用现在我们熟悉的 IM 的 UI 对其进行包装，但是深入下去，它们之间有很大的不同（见图 1）。

如果你关注近两年的 WWDC 和 Google IO，会发现 Siri 和 Google Assistant 的出场率极高，这种智能助手正是 Conversational UI 的代表之一。在这种现场演示的场景下，的确是这种对话式的人机互动更为自然。

图 1

Conversational UI的特点

自然的交流

Conversational UI达成的目标是，让机器和人能用自然语言形成会话，因此会话过程的设计也需要向真实的反应靠拢。综合起来，有以下几个特点：

回合制。这是是Conversational UI与IM不同的一点，IM上面一个人可以连续发言，但当与AI交谈时，AI必须回应人的每一次输入，否则人会怀疑AI是不是坏掉了。

延续性。AI在面对人的输入时需要给出恰当的回应，并维持会话，而不是单纯的问答。

主动推荐。现在的AI通过对人的历史习惯的学习，可以一定程度预测人的接下来的行为，因此可以做到主动推荐操作，避免重复输入。

每个消息都是一个应用

Conversational UI并不是现实中人们聊天，你一句我一句，现代的IM UI已经发展出多种消息形式混合，Conversational UI也继承了这个特点，无论输入输出都可具有多种形式。像在微信里，我们除了文字、语音、图像、视频外，还可输入位置、红包、名片等，而小程序更将这个范围无限拓宽。

在消息中的小程序也正是未来的发展趋势，无论Google、苹果、Facebook都在这个方向上发力。这意味着每条消息都可能成为一个应用，可以进行操作，具有输入输出（见图2）。

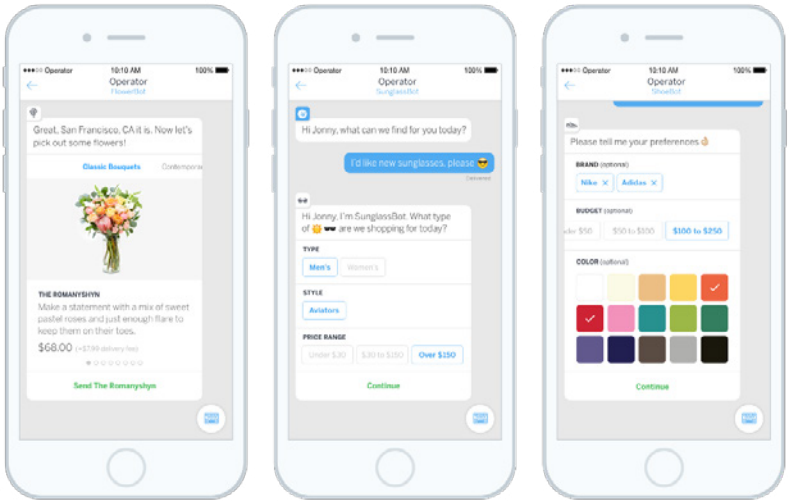

图 2

Hands-Free与无界面

与每条消息成为一个应用相对的，Conversational UI 也可以完全无界面，其中的代表就是智能音箱。当输入是语音或者我们的肢体动作时，我们还可以实现 Hands-Free（非接触），比如 Google Home 上可以实现非接触打电话。

智能音箱听着名字挺土，但事实上它代表着未来的交互方式之一。在科幻电影中我们常见隐形于室内的智能电脑，通过语音与人交互，完成各种任务，智能音箱使这些变成了现实。2014 年底亚马逊发布了 Echo 智能音箱，它被视为过去几年互联网最重要的发明，很多人认为这将是下一个计算平台，亚马逊的智能助手 Alexa 上面现在有上万种功能，让人想起早期的 App Store.

无界面的一个基础是随时待命，在智能音箱上这并不成问题，连 Google 和苹果都为自己的智能助手加上了这个功能，可以认为，这也将是 Conversational UI 的标配。

Conversational UI背后的技术

Conversational UI 背后的技术难点并不在于 UI 界面的开发，而在于语言识别和聊天机器人的智能程度。

语言识别使用的技术叫 NLP，全称自然语言处理，NLP 在过去几年取得了相当大的突破，语言识别的精度达到 90% 以上，甚至能从文字中提炼观点和感情。

智能 ChatBots 是另一项技术难点，在理解了人类的语言之后，如何自然的对话，实施操作，这背后需要复杂的框

架和大量的训练，这也是国外互联网巨头的研发重点。

对于一般规模的公司来说，完全自主研发有些得不偿失，因为这些都是平台，各大平台也都提供了 SDK，如 SiriKit、Google Assistant SDK，开发者只要去使用就行了。

小结

使用 Conversational UI 最自然的无疑是 IM 应用，但其它的应用其实也可以使用。凡是需要与 AI 交互的地方，都可以使用 Conversational UI。

智能助手平台以及 Conversational UI 仍然在快速演化，比如 Google Lens 是 Google 新推出的图像搜索功能，它为语音助手加上了图片识别功能。未来，智能助手或其它 ChatBots 将可以全方位的更好的理解我们，并执行我们的意图，通过将服务更深入的集成到智能平台里，我们有机会给用户提升更好的服务，提供次时代的用户体验。

延伸阅读

- https://www.youtube.com/watch?v=ulAKc2gezfU
- https://www.slideshare.net/MatthieuVaragnat/conversational-interfaces
- https://alistapart.com/article/all-talk-and-no-buttons-the-conversational-ui
- https://alistapart.com/article/designing-the-conversational-ui
- https://medium.com/the-layer/the-future-of-conversational-ui-belongs-to-hybrid-interfaces-8a228de0bdb5
- https://medium.com/swlh/conversational-ui-principles-complete-process-of-designing-a-website-chatbot-d0c2a5fee376
- https://developers.google.com/actions/design/

InfoQ

你应该加入InfoQ全职编辑团队的3大理由

可刷脸
蹭饭蹭会

分享就是
最好的广告

跟大牛们混的多了
想不牛都难

InfoQ社区志愿者永久招募中！

6大招聘职位：

1. 强力的**技术翻译**
2. 喜欢四处组织参与技术活动的**形象大使**
3. 在任意IT技术领域信息灵通的**线索发现者**
4. 深入了解任意IT技术领域的**专业内容把关人**
5. 擅长记笔记的**新闻撰写者**
6. 知道如何问好问题的**采访者**

我们是InfoQ编辑，我们是信息的罗宾汉。

现在就发邮件给editors@cn.infoq.com，告诉我们你的专长和意向，我们会将你培养成为一名好编辑：）

扫一扫关注InfoQ

Geekbang 极客邦科技 | InfoQ