

架构师 ARCHITECT



人物 | People

褚霸：不要为了开源而开源

解读2014 | Review

解读2014之云计算篇

解读2014之iOS篇

解读2014之Docker篇

专题 | Topic

手机淘宝构架演化实践

从信息安全看未来物联网发展

赵海峰：大数据决定互联网金融未来



Dare to Dream

因为任性，我们成了

2014 年开始的时候，我没有想到会真的完成年初制定的销售额，可是我们做到了；2014 年开始的时候，我没有想到团队能突破 20 人，可是我们也真的做到了；2014 年开始的时候，我也没有想到我们能有一个稳定的管理团队，可是我们也真的有了，而且运转的很好；2014 年开始的时候，我也真的没有想到某客户的 KPI 真的能够完成，可是我们竟然提前完成了……

因为梦想，我们有了

再拔高一点儿，这一切源于我们年初的梦想。每个人都需要有梦想，也都必须有梦想，不论是大梦还是小梦，这样对未来的才能有些盼头。雷军同学不是说过吗，梦想很大很虚，可是万一实现了呢？也正是因为人类的梦想，我们距离理想才越来越近。现在我还清晰地记得霍华德·舒尔茨在《将心注入》中提到的“我不做小梦，我做大梦。谁想要唾手可得的梦呢？”

回望过去，来之不易

回到现实，回望过去的一年，我们取得这样骄人的成绩并不是那么容易。我们为是否能完成客户的 KPI 而焦虑，为没有足够多的编辑、项目经理而发愁，为会议报名人数缓慢上升而挠头……在做事的过程中，不知道损失了多少脑细胞。我也清楚地看到大家做每日分享、周报，以及面对面的沟通中，大家所反馈的问题。

每次收到这些问题的反馈，说实话，有时候我也很沮丧。不过很快就释然了，因为作为管理者，天生就是为解决问题而存在的啊，没有问题，那么也就不需要团队、不需要协作，我们还在这儿干什么呢？“美好”到来之前，都是经过艰辛的精雕细琢的。

回望过去的几年，看看我们团队的人员规模，看看我们 QCon 网站的设计变迁，看看我们会议参会者的规模，看看我们的收入增加情况。

一路走来，我们解决了多少个问题啊，又进步了多少啊。到目前为止，我对我们所做的工作，感到非常骄傲和自豪。我想国内的架构师们是幸福的，因为有我们辛勤耕耘，对他们不离不弃。我想国内的技术社区也是幸福的，因为有我们做标杆，大家可以互相比拼。

这就是我们存在的意义和价值啊，想到这儿，有什么困难是不能被克服呢？

展望未来，仍做大梦

新的一年，我们有什么新的梦想呢？InfoQ 中国团队在接下来的几年会走向何方？在完成对 InfoQ 中国业务的收购之后，在有更多的自由度和决策权之后，我们如何更好地发挥现在这个团队的潜能呢？在过去的一段时间，好像我消失了，其实是钻到各种“沟通”中去了，和团队成员沟通，和顾问沟通，和朋友沟通，等等。基本把我们要做的事情大方向确定下来，那就是打造全球最具影响力的技术人学习和交流平台——极客邦，我们的任务就是让技术人学习和交流更简单。

所以在 2015 年，我们除了坚持通过 InfoQ 为国内的中高端技术人员提供最棒的资讯、最好的会议，还将通过公开课为国内的初级技术人员提供有品质的学习内容，通过搭建有规则的技术人组织为资深技术人员提供社交网络平台。

2015 年，我能想到我们会取得更大的成绩，但我更能想到我们会面对更多的挑战。团队的挑战、业务的挑战、管理的挑战、宣传的挑战……一大波的挑战。但一看到下面这句话，我就觉得所有的挑战都不在话下了，因为任性和坚持早已经在我们团队的身上打下深深的烙印：

我想建立的是这样一个公司：它的长期兴旺基于自己的价值观和指导原则的竞争优势。我想要吸引和雇佣那些为着同一个目标一起工作的人，他们不作窝里斗，却喜欢挑战人家以为不可能达到的目标。我想要创造这样一种企业文化，即个人在其中不仅能得到满足，而且能够得到他人的尊重和羡慕。我不做小梦，我做大梦。

——霍华德·舒尔茨

最后的最后，请大家记住这个名字，因为它极有可能会影响和改变你
的生活，如果你是一个技术人的话：极客邦。

极客邦创始人兼 CEO 霍泰稳

2014 年 12 月 31 日



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

QCon

全球软件开发大会

2015.4.23-25 北京国际会议中心



8
折
优惠

现在报名

(截至3月1日)

- 针对软件开发领域的热点与趋势，议定18个最具实践价值的专题，覆盖更广的开发者人群，为参会者提供更多选择

/可扩展、高可用架构设计/

/新兴大数据处理技术与工具/

/自动化运维/

/云计算平台构建与应用/

/团队建设/

/软件质量/

/敏捷之后，是什么/

/知名移动案例分析/

/挑战全栈开发/

/移动开发最佳实践/

/安全、隐私与风控/

/互联网金融背后的技术架构/

/微服务和响应式框架/

/Java和新锐语言/

/超越JavaScript/

/思考开源/

/基于大数据的机器学习和数据挖掘/

/技术创业/

抢票热线: 010-64738142

会务咨询: qcon@cn.infoq.com

精彩内容策划中，讲师自荐/推荐，线索提供: speakers@cn.infoq.com

更多经典专题 精彩内容 敬请登录: www.qconbeijing.com

目 录

解读 2014 | Review

解读 2014 之云计算篇：有一种态度叫做“拥抱”（上）

解读 2014 之 iOS 篇：拥抱变化

解读 2014 之 Docker 篇：才气、运气、勇气

观点 | Opinion

SDN 落地的实践与思考：带着问题找方案，别管定义啦

Swift 编程语言

专题 | Topic

手机淘宝构架演化实践

手机 QQ 的移动化实践之路

物联网传输协议 MQTT

从信息安全看未来物联网发展

赵海峰：大数据决定互联网金融未来

推荐文章 | Article

为何 Asana 开始用 TypeScript

为什么不要把 ZooKeeper 用于服务发现

Java 8 新特性：全新的 Stream API

人物 | People

褚霸：不要为了开源而开源

解读 2014 之云计算篇：有一种态度叫做“拥抱”（上）

作者 崔康

编者按

2014 年，整个 IT 领域发生了许多深刻而又复杂的变化，InfoQ 策划了“解读 2014”年终技术盘点系列文章，希望能够给读者清晰地梳理出技术领域在这一年的发展变化，回顾过去，继续前行。

本文为“解读 2014 之云计算篇”，今年，云计算已经不再像前几年那样火热，产业界对云计算的关注度已经被大数据、物联网等新的名词所超越，但这并不意味着云计算本身影响力的削弱，而是因为“云”已经成为 IT 技术和服务领域的“常态”。产业界对待云计算不再是抱着疑虑和试探的态度，而是越来越务实地接纳它、拥抱它，不断去挖掘云计算中蕴藏的巨大价值。

本文试图通过几个关键字来勾勒出整个云计算领域今年发展的脉络，深入分析这些词背后的意义、InfoQ 对此的报道、对势态的评论等。

生态

“生态”是 2014 年云计算领域说得最多的一个词，各大厂商都逐渐认识到云服务不是一个孤立的产品，传统的软件销售模式已经不再适应互联网的要求。客户购买云服务，不是使用它的某种“功能”，而是基于厂商的平台搭建一个系统，规模可大可小，这种服务使用模式要求客户具备全面的技能，面临的困难也接踵而来。基于此，打造一个完备的生态圈，成为云服务厂商的当务之急。

InfoQ 此前[报道](#)过，阿里云宣布启动“云合计划”，基于开放的 API 以及共享的市场资源，对外招募合作伙伴。目标是在未来三年内招募万家以上的合作伙伴，无论是咨询公司、系统集成商、独立软件商、中小服务商、软件外包与 IT 服务商还是数据中心服务商。事实上，据

我和多个云服务厂商沟通了解，包括 Microsoft Azure UCloud 等都推出了相应的创业者扶植计划，不一而足。

既然是生态系统，那么一定要制定并维持好生态系统中的游戏规则。正如我在某云厂商的恳谈会中所说，这个系统不单单是开发者，还需要包括他背后的就职公司、他开发的产品面向的客户、开发者的合作伙伴等等，当然还有云服务厂商。一个平台想要凝聚力，不外乎从物质和精神两个方面分析。物质方面，开发者能够从这个平台上获取利益，比如这个平台是支持开发者创业的、多语言跨平台的、具备快速部署优势的、适合迭代开发的，生态系统的成功案例多，具备这些特质，会让这个平台的开发者最终受益。从精神方面，要让开发者有参与感，他在开发方面的经验和知识能够通过生态系统分享出去，影响其他人和公司，并且他的意见能够得到及时的反馈，那么这种参与感会吸引开发者。

云厂商的成败关键在于生态圈建设，谁的圈子越大越成熟，谁就越容易活下来。可以预见，在未来两年，厂商的生态圈建设将全面铺开，除了目前已有的创业者支持计划等，更重要的是制定生态圈的生存法则，塑造合作共赢的业务模式，让生态圈里的大大小小的利益相关者都能找到适合自己的定位，并最终盈利。

落地

“落地”这个词在今年更多地用在了全球的云服务提供商。从没有人怀疑中国市场的巨大潜力和利益，云厂商也不例外。从世界范围来看，美国领跑全球云服务市场，云计算产业体系完整，巨头企业加速向全球扩张，据业内数据，目前在全球 TOP 100 的云计算企业中，美国占 84 家。亚马逊占全球 IaaS 市场的 40%、微软占全球 PaaS 市场的 64%，Salesforce 占全球 SaaS 市场的 21%。众多厂商在领跑之后，都把下一个掘金的目标对准了中国，国内的 IT 产业近几年迅猛发展，云计算市场发展势头很好，特别是国内云厂商正在抢占市场份额，让美国公司的落地步伐迈得更紧。

据 InfoQ [报道](#)，微软中国在年初宣布由世纪互联运营的 Microsoft Azure 在中国正式商用，并在年末的时候开放注册，虽然中国版和国外版在某些最新功能上有所延迟，但同步的节奏正在加快。Satya Nadella 在今年 2 月上任微软 CEO 后提出“云优先、移动优先”的口号，Scott Guthrie 则接任 Satya 此前担任的 Cloud and Enterprise 执

行副总裁一职。年末的时候，我采访了 Azure 落地中国的总导演、微软大中华区副总裁兼市场营销及运营总经理严治庆。他指出，作为第一个吃螃蟹的人，国外云厂商需要通过独特的商务和运营模式在中国落地，这既需要和各级政府部门的积极努力，也需要合作伙伴的紧密合作。

除了已经落地的 Azure，还有一个即将落地的大鳄——AWS，作为云服务的开创者，AWS 在国内一直保持的很高的品牌知名度。据我了解，很多有国外业务项目的本土公司已经大量使用了国外的 AWS 服务。面对国内机遇，AWS 着手布局。目前在国内推出了有限预览版，开设了多个数据中心，并在今年推出了中文支持和运营团队，部分企业开始在国内区域上线业务，何时才能真正落地，我们在 2015 年值得期待。

未来 IaaS 和 PaaS 平台的竞争将更加激烈，特别是在国外云厂商进入中国以后。国外公有云进入中国的门槛比较高，受到一些条件的制约。但是中国的市场潜力巨大，包括亚马逊、微软等 IT 巨头都在进入中国。在未来两年，更多的国际云厂商通过与国内公司合营的方式进入中国，其相对成熟稳定的云服务和较高的品牌效应将抢占国内份额。不过，和众多国外产品进入中国面临的问题一样，如何“接地气”是一个巨大的挑战。我们看到，对于耕耘中国多年的国际巨头，在这方面已经很有经验，构建生态圈，同时给予中国的研发团队足够的话语权，问题正在得到解决。

国外云服务落地中国，一方面是带来竞争，另一方面是繁荣市场。我们应该看到，国内的云计算市场正在处于“落地”的阶段，需要有更多的玩家进来，不论是国外的、大大小小的，一起把这个市场做大，让企业更加了解云计算，拥抱云计算，这样大家才有肉吃。如果只是一两家独大，活力不足，整个云计算产业也带动不起来。

收购

“收购”正在逐渐成为各大公司实现战略布局、抢占先机的快捷手段。我们很久就了解到，制约经济增长与经济活动的，并不是资金的供给不足，而是对资金的需求不大。所以，对很多公司来说，担心的不是没有钱，而是有了钱，应该如何进行有效的利用。在日新月异的互联网时代，一方面云计算产业机会很多，给中小型创业公司提供了

发展和崛起的空间；另一方面，大厂商应对变化的动作受到机制的约束，缓慢而繁琐。收购成为大公司占坑的好办法。

我们来看看本年度比较有名的云计算领域收购新闻：

- 惠普收购云计算软件公司 Eucalyptus
- Docker 收购持续集成服务提供商 Koality
- 爱立信收购云计算软件公司 Apcera 控股权
- EMC 收购云计算创业公司 Cloudscaling
- SAP 收购云计算软件公司 Concur
- 苹果收购云计算公司 Union
- 云计算企业 CallidusCloud 收购 LeadFormix
- 思科收购私人控股 OpenStack 供应商 Metacloud
- 世纪互联收购第一线集团旗下的 DermotHoldingsLimited
- Google 收购云计算工具开发商 Firebase

其中，惠普收购 Eucalyptus 的新闻，InfoQ 曾经[报道](#)过，收购之后，原 Eucalyptus CEO Marten Mickos 将出任惠普 Senior VP、云计算业务部 General Manager，负责 Helion Cloud 的发展并直接汇报给惠普 CEO Meg Whitman。Meg Whitman 表示将投入 10 亿美元打造 HP Helion Cloud 的品牌，而 Helion Cloud 在技术上则跟 OpenStack 项目保持高度一致。

这个名单还可以列很长，不一而足。我们可以看到，收购成为了云计算产业的常态，这也从侧面反映了这个产业正在处于快速发展和变革的阶段。从收购公司的角度分析，收购行为的目的不外乎以下几点：

- **获取人才。**有些收购事件，大企业想要的可能不是这些小公司的产品或者核心技术，而是背后的人才我曾经有机会问过多位公司高管：“你们的核心竞争力是什么”，他们的回答通常会归结为一点：人。看起来挺虚的，不过仔细考虑，不论是产品设计、架构设计，还是运营模式、商业模型、资源运作等，都是人在做，通常顶尖的人才比普通人的效率和效益要高几百上千。有时候，少数顶尖的人才做的决策会决定公司的正确走向。所以收购之后，这些人才可能会转而参与更加重要的工作当中。
- **获取核心技术。**这在收购事件中比较普遍。一项核心技术的研发有时需要比较长的时间和比较成规模的团队，而且还带着一些运气的成本，再加上存在的专利约束。考虑到这些综合性的因素，从头研发一项核心技术，还是收购一项核心技术，对于大公司来说，收购的性价比更高。这种方式可以让公司快速突破技术障

碍，收购之后，核心技术会马上应用到公司的重要产品中，获得技术优势，提高市场竞争力，从而积极影响公司的发展趋势。

- **获取产品**，也就是补充产品线。产品和核心技术还不太一样。有些收购，公司是为了把被收购的产品加入到自己的产品体系中。考虑到云计算产品和服务的多样性，大公司不容易做到一个完整的产品线，对于非核心的业务，通过收购的方式纳入一些辅助的产品，从而帮助自己快速构建一个完善的系统，这种方式值得推荐。当然，整合收购的产品是一项非常具有挑战性的任务，之前也有失败的案例，需要公司投入比较多的人力和物力，才能整合成功。
- **获取客户**，也就是获取市场。这在收购事件中也比较普遍。通常在这种情况下，被收购的公司和收购公司存在竞争关系，两家各占有一部分市场份额，但是产品和服务都有差别。大公司收购小公司之后，市场份额产生叠加，提高了市场占有率。通常有两种发展方向，一种是保持现有的状态，两种不同的产品服务于不同的客户；另一种则是以一种产品为主，逐渐替代另一种产品，从而以统一的方式服务于客户。目前来看，收购之后的结局大部分是后一种。不过，这种变化同样具有很大的挑战性，如何实现两种产品的统一化服务是个难题，在实践中有成功也有失败。
- **防御性收购**。所谓“敌人的敌人就是朋友”。这种情况下，收购公司对被收购公司的人才、核心技术、产品和客户都不感兴趣，大公司通常不希望竞争对手发展壮大，所以对于对手的潜在收购目标都会出手干预，甚至收购过来，要么好好培育，要么慢慢扼杀，总之不能让对手占到便宜。这种单纯的收购方式不常见，一般都是多种收购目的综合作用的结果。

对于被收购公司来说，大多数情况是个好事情，背靠大树好乘凉，市场竞争的激烈让很多创业中小型公司过早夭折，活下来的公司通常都是被收购，极少数才能走到独立发展甚至上市那一步，在当今的 IT 产业中很少见。对于很多公司来说，他们开始创业的时候，最初的目标也许就是被收购。

回到云计算厂商的收购分析上，刚才提到的几种目的在今年的收购中都出现过，不论是哪种目的，这些收购一定是为了公司的战略决策服务的，至于收购的效果，还需要更多时间至少到明年才能看出些眉目，现在下结论就太早了。收购作为大公司的一种标配，为云计算产业的布局起到了重要作用。未来还会有更多的收购出现，直到整个产业进入稳定期，只有几家大玩家的时候。但对于云计算来说，还太早，而且很多细分领域不会出现几家独大的情况。

Docker

“Docker”应该是今年最火爆的技术之一，如果没有听说过，那么你有点 out 了。关于 Docker 在这一年的发展情况，可以单独成一篇稿子了，我们 InfoQ 最年轻的高级编辑、也是长期追踪 Docker 发展的创作者郭蕾同学稍后会发表一篇详尽的 Docker 年终盘点，我就不在这里献丑了。我希望从技术发展史的角度来分析下目前 Docker 所处的发展阶段和历史机遇。

一项技术的成功因素其实和谈恋爱一样：在正确的时间、正确的地点遇到正确的人，也就是古语所说的“天时、地利、人和”。回首整个 IT 发展史，我们很难讲某项技术好还是不好，却可以说这个技术是不是顺应了时代的发展需求。一项技术要想获得较好的发展，通常需要满足以下条件：

- 该技术相比之前的解决方案有创新之处，在本质上有所改善——以 Docker 为例，创新分很多种，发明创造是一种，而把现有技术应用到另一个领域解决问题，也是一种创新，Docker 就是后一种，而且解决问题的效果比较明显。
- 应用该技术的领域进入快速发展阶段或者积重难返、亟需取得突破——随着云计算领域的发展，开发者面临着环境管理复杂、软件配置管理复杂等问题，Docker 提供了一种在安全、可重复的环境中自动部署软件的方式，它的出现为基于云计算平台发布产品提供了崭新的方式，这也得益于云计算的发展需求。
- 该技术门槛不高，社区和开发者乐于学习——对于云计算领域的底层工程师来说，Docker 的学习曲线比较友好，没有高深的数学算法，再加上是开源项目，社区参与度高，学习资料和论坛多，所以关注 Docker 的人基数比较大，群众基础好。
- 有厂商愿意推动该技术的发展——刚才提到，Docker 的出现为基于云计算平台发布产品提供了崭新的方式，这对厂商来说是吸引开发者的亮点之一，所以 Docker 一经推出，就吸引了 IBM、Google、RedHat 等业界知名公司的关注和技术支持。

Docker 满足了上述条件，这也是如今火爆的原因。那么，它现在发展处于哪个阶段呢？我们应该如何把握它的发展脉络，是应该全情投入，还是继续观望。我们在这里做一个简单分析。一项技术的发展周期，通常分为以下阶段：

- **播种期**——技术概念刚刚提出，原型正在制作，没有任何评估和分析，只停留在纸上谈兵。市场份额为 0。
- **孕育期**——有社区和厂商开始关注该技术，并投入资源研究和推动技术发展，讨论其有效性和可行性，培养生态系统，处于起步阶段。
- **成长期**——在技术方面明确之后，应用技术的产品和服务快速发展和推出，整个社区和众多厂商推动该技术发展，并获得客户的广泛认可。
- **成熟期**——技术研究、应用、评价、路线图都已经非常明确。社区对技术已经完全了解，更多关注应用，厂商的产品已经成熟，并逐渐发现新的问题。市场份额保持稳定。
- **衰落期**——技术逐渐无法满足产业发展的需求，新的技术被提上日程，老技术逐渐收到冷落，市场份额严重下降。

如果按照以上的标准来分析 Docker，我们可以认为，它还处于孕育期末尾、成长期开始阶段，社区和厂商开始关注，并投入了较多资源，在业内也产生了一定的声音，也出现了部分应用场景，但大规模的应用还没有推广开来，需要时间的积累。我曾经和国内多家云厂商的专家聊过，他们都很关注 Docker 技术，认为是云计算的一个助力点，但目前都还没有实际应用。一方面是等待技术和社区更加成熟，另一方面也需要投入资源实现技术转型。

由此可见，Docker 的未来趋势已经很明朗，尽早关注和拥抱 Docker 是正确的选择。按照以前的 IT 发展规律来看，规则的制定者要比追随者更有主动权和竞争优势。所以，目前阶段的 Docker 给了厂商和开发社区留下了“上船”的时间和空间，越往后越被动。

InfoQ 一直尝试站在技术浪潮的前沿，让国内的架构师、开发者了解和借鉴整个技术社区的成果。就在几年前，Node.js 刚刚开始萌芽和孕育时，我就策划了“[深入浅出 Node.js](#)”专栏和迷你书，在国内较早地推动该技术的发展和普及，现在 Node.js 已经进入了成长期，我们也为自己曾经做过的努力感到欣慰。现在轮到 Docker 了，在同事郭蕾的积极推动下，InfoQ 已经成为国内 Docker 技术传播推广的最前沿阵地，“[深入浅出 Docker](#)”、“Docker 源码分析”、“Docker 周报”等热门栏目不断推出，让国内社区和厂商有一个比较全面深入的了解。前几天，郭蕾告诉我，InfoQ 挖掘的 Docker 作者都已经成为国内各个 Docker 社区和会议的带头人了，这是好事，这是我们的价值。在 2015 年，我们还要做更多。

上篇结束，感谢大家的耐心阅读，后续内容将稍后推出。本文由 InfoQ 总编辑崔康执笔，编辑部成员均有贡献。

解读 2014 之 iOS 篇：拥抱变化

作者 唐巧

对于 iOS 技术领域，2014 年是变化巨大的一年。在硬件上，由于苹果发布了更大尺寸的 iPhone 6 和 iPhone 6 plus，使得 iOS 设备的屏幕适配工作也多了起来。这还不包括全新的 Apple Watch 带来的开发上的变化。在软件上，Swift 语言的推出惊呆了整个业界，从 Objective-C 切换成另一个全新语言的代价是巨大的，但我们也看到了它长远的好处以及苹果对此的决心。除了软硬件外，编程语言、中国区、开发社区在 2014 年也有着巨大变化，我们试着用几个关键词来勾勒出 iOS 技术领域在 2014 年所经历的变化。

硬件

苹果在 2014 年推出了 iPhone 6 和 iPhone 6 plus，iOS 设备也进入了大屏时代！虽然包括笔者在内的许多朋友都认为：“Plus 实在太大了”，但是从销售量看，iPhone 6 plus 在初期占据了 [6成](#) 的销售量。我本人在实际使用 iPhone 6 plus 几周后，也很快适应了它的屏幕尺寸，现在反而不习惯使用了两年的 iPhone 4s 了。在开发上，iPhone6 的推出使得 iOS 开发者也开始像 Android 开发者那样，为适配工作担忧。与 Android 开发者不同，iOS 开发在过去多年都是用“绝对定位”的方式来放置各种 UI 控件，使用 Autoresizing Mask 就可以轻松搞定 iPhone 4 和 iPhone 5 屏幕高度的差异问题。但现在不一样了，iOS 开发者需要学习使用 AutoLayout，通过各种 Constraint 来调整 UI 控件，但是很多界面是需要用代码来编写的，所以这方面的开发工作会比较辛苦，好在有 [Masonry](#) 这一类的开源库来辅助我们做界面开发工作。

iPhone 6 plus 引入的特有的 3x 的图片问题使 UI 设计师的工作量也大大的增加了。另外，以前由于 Android 设备其实太过于分裂，无法对每种屏幕分辨率都做到精致地设计，所以设计师大多只是用“凑合用”的思想来实施 Android 应用的适配。这一点从 Android 大屏手机上的应用呈现就能看出来。而现在同样的方式不太可能被严格的苹果以及挑剔的 iPhone 用户所接受，设计师很可能需要为 iPhone 6 plus 做专门的大屏设计，才能获得用户的芳心。

Apple Watch 的推出让适配工作真正地进入了 Hard 模式。优秀的开发者需要在工程中做好代码地复用，才能让底层的逻辑能够同样运行在 Apple Watch 中。不过这方面的工作还只是猜想，我们得等到 Apple Watch 真正面世的时候，才能真正了解到相关的开发工作有多么不同。

软件

苹果在 2014 年的全球开发者大会（World Wide Developer Conference，简称 WWDC）上并没有发布任何新的硬件产品，但展示出其在软件上的各种功能改进和整合上的巨大努力。苹果推出了新的 OS X 10.10 操作系统 Yosemite，它吸收了从 iOS 7 开始的扁平化的设计风格，很多改进的细节都可以看到 iOS 系统对于 OS X 的影响。Yosemite 最吸引我的是 handoff 功能。handoff 功能将 Mac 电脑和 iPhone、iPad 的功能进行了深度整合，当你拥有这三个设备时，它们每一个都可以接打电话和收发短信，你在它们任何一个设备上编辑的文档，也可以实时同步到另一个设备上。handoff 并不是苹果的应用独有的功能，苹果将其相关的 API 都开放给了开发者，所以开发者也可以完全开发出支持 handoff 功能的应用。

相对于 Yosemite 带来的整合方面的改进，新版 iOS 8 的改进的重点则是开放。苹果开放了输入法接口，允许第三方提供输入法。苹果也开放了 Touch ID，允许第三方应用获得指纹信息。HomeKit 则为智能硬件开放了不少可以互操作的接口。另外，iOS 8 还新增了 4000 个新的 API，开放了大量底层 API 接口供应用调用。所以对于普通用户而言，iOS 8 新增的可见功能可能是有限的，但是其开放的大量 API 和新 SDK 释放出了巨大的想象空间，相信会产生更多有趣的应用。

对于一个硬件来说，相关的软件是非常重要的，但是苹果却将它的操作系统和 iWork 办公套件完全免费，这显示出苹果对于软件生态圈不同的理解。而微软至今最大的两大收入还是来自它的 Office 和 Windows 系列操作系统。在微软还在发愁他的新版 Wiindows 10 怎么卖出去时，苹果早已不玩这种 1.0 的商业模式游戏了。

Swift

苹果的 Objective-C 语言发明于与 C++ 同时期的上世纪 80 年代初，虽然苹果对其进行了很多次改进，但这终究像给应用打补丁一样，不是

特别舒服。这次 Swift 的推出，苹果试图用全新的编程语言来提高 OS X 和 iOS 程序员的开发效率。单从语言特性上来看，Swift 吸收了众多现代编程语言的优秀特性，例如类型推断（Type inference）、范型（Generic）、闭包（Closure）、命名空间（namespace），元组（tuple）等，整体语法上也更加简洁。我们也了解到 Swift 是苹果从 4 年前就开始筹划中的事情，可见苹果在长远规划上对于 Swift 语言的重视，相信未来 Swift 会接替更多 Objective-C 原本的地位，所有 iOS 开发者都应该花时间学习这门全新的语言。

苹果这次推出 Swift 也吸引了大量的讨论，其中比较不正确的观点是认为 iOS 开发的门槛因此而降低了。从我和身边的同行对 Swift 的学习和讨论来看，Swift 并不是一门可以简单上手的语言。并且在未来一到两年内，iOS 开发必然会经历同时使用 Objective-C 和 Swift 的过渡阶段。对于 iOS 开发者，除了必须掌握以前的 Objective-C 语言的知识，还需要学习并不简单的 Swift，可见 iOS 开发的门槛不但没有降低，反而提高了不少。

中国

苹果是重视中国的，它在大中华区的销售贡献了约 15% 的全球营收。但是从某些方面看，苹果还是不够重视中国的。很多中国用户抱怨的问题，苹果都一直没有解决。我们来看看苹果对于中国区的用户和开发者，还有哪些需要改进的地方。

刚刚提到的 handoff 功能依赖于苹果的 iCloud，但是苹果的 iCloud 存取速度在中国却是非常糟糕的。笔者试图打开一个通过 iCloud 共享的约为 6M 的视频，但是等了 3 分钟才打开。与此同时，国内的各种网盘服务却能提供非常大的免费容量以及非常快的访问速度。

中国的垃圾短信和骚扰电话一直是非常严重的，而 iMessage 免费发送的特点给了垃圾短信发送提供了很好的平台。苹果到现在都没有推出相关的有效避免垃圾短信的办法，只提供了一个用于反馈垃圾短信的邮箱，整个反馈流程也非常繁琐，需要提供发送者邮件、内容以及相关截图。笔者反馈了数十次之后，也没有得到过任何回应。对于骚扰电话，苹果只提供了事后将该电话号码加入黑名单的功能，这使得用户只能是接了骚扰电话后，才能做相应的手工处理。这些问题从技术上要解决没有任何难度，看看 Android 平台就可以看到这方面的问题

被非常完美地解决了。但是苹果却不愿意做相关的努力，这只能说明它还不够重视中国用户。

苹果的 App Store 中国区也是比较混乱的，各种刷榜的应用充斥着排行榜的前几名。苹果在这方面应该是做了很多努力的，但是终究还是做得不够。从很多应用的评论中，我们都能看出刷榜的痕迹。由于这方面直接和收入相关，所以我相信苹果还是会尽力改进的。App Store 还有一个问题是支付不太方便，今年苹果和银联合作，不但推出 1 元的应用，而且允许用户使用银联卡来付费，极大地方便了用户购买应用。

社区、会议和第三方服务

国外的 iOS 开发社区今年依然发展红火，比较引人注目的是 objc.io。objc.io 每一期的质量都非常高，所以由知名开发者王巍 (onevcat) 组织的翻译团队会将每一期文章都翻译成中文。国内的博客和社区发展相比国外还是比较慢，参与分享 iOS 技术的独立博客相比以往多了很多，但整体质量还有待提高。

移动开发也实实在在地成为了一个重要的技术领域，QCon 和 ArchSummit 每一届都会有专门的移动专题，可见这方面聚焦了开发者大量的关注。

2014 年国内外都涌现出了大量服务于移动开发者的第三方服务。这些第三方服务作为生态圈的一个重要组成部分，将会方便开发者聚焦核心功能，而不用担心基础设施的搭建。

展望

展望 2015 年，笔者认为：

- 苹果的 iPhone 将会继续占据高端机市场，但千元左右的 Android 机将会占据其它所有的用户，最终 Android 的用户量将远远大于 iPhone 的用户量，但由于 iPhone 用户的优质属性（付费意愿高），所以各大应用将会在这两大平台都会投入足够的开发力量。如果有人能够比较好的解决代码在 iOS 和 Android 平台的复用性问题，将会很大程度上改变现在的开发模式。

- Swift 语言将仍然处于发展阶段，Objective-C 依旧会是 iOS 程序开发的主要语言。更多的开发者将会使用混合开发的方式，同时使用 Objective-C 和 Swift 来开发应用。
- 国内 iOS 开发社区还会进一步发展，会出现更多的移动开发相关的开源基础设施或第三方服务。
- 苹果很可能会继续不那么重视中国区用户。iCloud 服务依然糟糕，iMessage 垃圾短信依然泛滥，由于害怕“侵犯用户隐私”，骚扰电话还是没有很好的解决办法。但苹果在大中华区的收入会继续增加。
- 由于 Objective-C 和 Swift 相比 Java 语言没有那么普及，Mac 电脑也没有 Windows 电脑普及，这造成 iOS 学习门槛较高。优秀的 iOS 开发者依然是稀缺资源，相比同等的 Android 开发者，iOS 开发者可以获得更多的收入。

解读 2014 之 Docker 篇：才气、勇气、运气

作者 郭蕾

编者按

2014 年，整个 IT 领域发生了许多深刻而又复杂的变化，InfoQ 策划了“解读 2014”年终技术盘点系列文章，希望能够给读者清晰地梳理出技术领域在这一年的发展变化，回顾过去，继续前行。

毫无疑问，[Docker](#) 已经成为 2014 年最热门的技术之一，它被爱好者冠以云计算新星、下一代虚拟机等称号，可见大家对其的期望之高。2014 年，Docker 的发展可谓是一路凯歌，从年初的 B 轮融资到年末的 DockerCon 欧洲大会，Docker 在这一年里顺风顺水，就连微软、谷歌、AWS 这样的巨头也敬它三分。

InfoQ 从 2014 年 5 月开始重点关注 Docker，先后策划了[深入浅出](#)、[源码解读](#)、[CoreOS 实战](#)、[Docker 周报](#)等系列专栏，共产出了百余篇新闻与文章，同时还建有两个千人的 QQ 群以供读者学习交流。在这一年里，作为负责 Docker 专栏的技术编辑，我密切关注了 Docker 的每一次融资、收购、合作，也亲身见证了一个技术的发展之路。本文作为 Docker 专栏的年终总结，分析了过去一年 Docker 的发展、模式、应用，并结合自己的理解展望了 2015 年 Docker 的发展以及与它相关的技术变革。

另本文的标题来自热播无胸版电视剧《武媚娘传奇》张公公之口。

发展

2014 年是 Docker 的起步之年，这一年里 Docker 又是融资，又是收购，又是办大会，又是与巨头合作，迅速获得了资本以及云厂商的认可。记得在 6 月的一段时间里，每天早上起来只要一打开朋友圈，就会看到关于 Docker 的爆炸性新闻，想想也是醉了。先撇开 Docker 的优势不谈，我们来回顾下过去一年 Docker 的几个重要的发展节点。

- 1) **Docker 1.0 发布。** 6月10日，[Docker 团队宣布发布 Docker 1.0 版本](#)，从第一个版本到 1.0 版本的发布，Docker 大约经历了 15 个月的时间，共收到了超过 460 位贡献者的 8741 条改进建议。1.0 版本的发布也就意味着 Docker 可用于生产环境，从发布特性中可以看到，Docker 当时重点在关注稳定性、扩展性、兼容性等方面的问题，并没有提及安全事宜。同时从数据中可以看到社区对 Docker 的贡献。
- 2) **Docker 出售 dotCloud。** 8月4日，[Docker 宣布出售其平台即业务服务 dotCloud](#)，并称接下来将专注于容器业务。CEO Ben 表示他们已经看到 Docker 的快速增长，希望把所有的精力和资源都集中在这项业务上。从这件事情上可以看出 Docker 公司的决心，把成败都押在 Docker 上，何尝不是一种魄力。当然，那时 Docker 有足够多的用户，他们也很有信心。
- 3) **微软全面拥抱 Docker。** 10月，[微软宣布下一个版本的 Windows Server 将支持 Docker](#)，同时还发布了 Windows 10 客户端的技术预览版。Windows Server 主要是面向企业市场，显然，微软已经意识到容器技术的发展趋势以及其对企业的意义。比较有意思的是，微软早在之前就开发了自己的容器技术（Drawbridge），而 Windows Server 却选择支持 Docker，可见 Docker 的势头之猛。
- 4) **AWS 加码押注 Docker。** 在 AWS re:Invent 2014 大会上，[AWS 推出了高性能容器管理服务 EC2 Container 服务](#)，用户可以在 AWS 上使用容器轻松地运行和管理分布式应用。AWS 是云计算的老大哥，在其 EC2 上早就支持 Docker，这次的容器服务瞄准的是集群服务，也是第一家将 Docker 应用于集群服务的公有云，AWS 果然是 IaaS 的带路人和领头羊。
- 5) **Docker 发布新的跨容器的分布式应用编排服务。** 12月4日，[Docker 宣布发布跨容器的分布式应用编排服务](#)，编排服务可以帮助开发者创建并管理新一代的可移植的分布式应用程序。之前 Docker 关注的都是单个或者少量的容器，所以我们也经常看到官方用词“Engine”，而编排服务的发布意味着 Docker 正式进军集群服务，彻底平台化。通过三个新的编排服务（[Machine](#)、[Swarm](#)、[Compose](#)），用户可以迅速构建 Docker 集群环境并部署应用，至此，Docker 为 2014 年画上了圆满的句号。

模式

正如热播电视剧《武媚娘传奇》中所说，一个才人的上位需有才气、勇气、运气，Docker 在这一年里能有如此快的发展也与此道理大同小异。才气不多说，从 Docker 一出现大家就很认可它带来的颠覆性的优

势，现在流行打标签，如果要给 Docker 写标签，我认为应该有几个词：容器、虚拟化、轻量、可移植、分布式。谈到勇气，不得不提 Docker 的创始人 Solomon (dotCloud 的创始人)，在 PaaS 市场举步维艰的情况下，他敢于将自己的核心引擎开源，并让团队的核心成员参与开源项目，以及后来直接把 dotCloud 卖掉，把全部精力都投入到 Docker 的开发上，可谓魄力十足。运气又名天时，近几年，DevOps、微服务、云计算等技术理念如日中天，而 Docker 可以全部和这些技术集成，并且都能为之一颤。

从[官方公布的数据](#)来看，截止到 2014 年 11 月，Docker 的贡献者已经超过 700 人，与去年相比增长 52%；Pull Request 数量为 5200 个，增长 37%；GitHub 上相关的项目已达 18000 个，增长 177%；相关的仓库数量已达 65000 个，增长 348%；Docker 的下载量有 6700 万之多，增长 2336%。

一年的时间里，Docker 的生态系统发展迅猛，知名的云计算公司以及软件、操作系统、系统集成厂商、配置管理软件、大数据厂商以及开源软件都在向 Docker 靠拢，不管是在哪个领域，Docker 都在“Doing the old thing the new way”。Docker 相关的生态圈可以参考下图，图片来自 DockerCon 欧洲，需要注意的是图中没有列出国内的云计算公司。



应用

前面提到过，Docker 可以和很多的概念联系到一起，生态圈如此之大，那可以做的事情肯定也很多。但是，现在有多少公司/开发者在使用 Docker 了？这个问题真是得打个大大的问号，从平时的采访以及 QQ 群中读者的问题来看，绝大部分人对 Docker 的使用还停留在观望状态，并且将 Docker 应用于生产环境的公司少之又少。记得刚开始追踪 Docker 的时候，我把英文站的一篇[新闻](#)中描述 Docker 发展的词主观的翻译为了“吹捧”，我认为媒体、社区、厂商对 Docker 炒作的声音远大于需求驱动，看似火热而又令人心动的变革其实并不如外表看起来的那么红火。不过，从一个[技术的发展角度来看](#)，如此情况也在情理之中。

Docker 的应用和它本身的优势密切相关，系统集成厂商 [Flux7 曾总结过 Docker 的一些应用场景](#)，虽然 Docker 目前的实际应用人数并不多，但也不乏最佳实践。开发方面，开发者可以使用 Docker 搭建开发环境，借助 Docker 可移植的特性，开发者可以将自己的环境分享给开发以及相关的测试同事，省去了因为环境搭建而耗费的时间。同时，由于 Docker 可以快速创建并启动一个或多个容器，所以它可以和 Jenkins 一起来进行持续集成（CI），相关的开源项目有 [Drone](#)、[Strider](#)。

很少有人提及 Docker 对 SaaS 带来的影响，从普通用户的角度来看，Docker 可以为他们解决很多苦恼的技术问题。举个例子，之前用户想卖东西，可能就是入驻淘宝这样的平台，一是需要依靠大平台的流量，二是自己根本无法搞定一套电商程序的安装以及维护。有了 Docker 后，用户拿到的就是镜像，通过几个来回的命令就可以完成安装。所以我认为假以时日，类似 Wordpress、Drupal 这样的开源软件都会通过镜像的方式交付给用户，用户也会在选择 SaaS 平台还是自己搭建方面有更多的主动权。同样，得益于 Docker，SaaS 平台也许会向 PaaS 过渡。

PaaS 方面，Docker 已经扎根新一代的 PaaS，如果把 Azure、Heroku 等公有的 PaaS 看为第一代，允许用户自建 PaaS 的 Cloud Foundry 和 OpenShift 就应该是第二代，那以 Docker 为首的平台就应该是第三代，主要代表有 [Deis](#)、[Flynn](#)、[Tsuru](#)。目前第三代 PaaS 也已经成熟，其中 Deis 和 Flynn 都已经发布 1.0 版本，借助 Docker，这些平台可以占用更小的资源。部署方面，新一代 PaaS 平台也可以借助 Docker 实

现从开发环境到 PaaS 平台的无缝迁移，可谓从里到外，Docker 都是得力帮手。

IaaS 方面，巨头 AWS 已经推出基于 Docker 的容器服务，老二 Azure 也已经全面支持 Docker，并和官方建立了合作关系。放眼国内，阿里云、腾讯云、UCloud、青云等公司都已经支持 Docker，更进一步的支持看似也没有。同时，新一代的基于 Docker 的 IaaS 也在紧锣旗鼓的设计中（此处省去很多创业公司），Docker 可以提高资源利用率，降低云成本，所以也有很提出了 CaaS（容器即服务）的概念。没有人能够知道在这样的时代 Docker 到底会带来什么样的变革，所以面对如此火热的技术，IaaS 厂商扑上去也容易理解。

2015 年

11 月底，[CoreOS 发布了自己的容器引擎 Rocket](#)，Docker 有了名义上的第一个竞争对手。新的一年，容器之争才刚刚开始，其实微软、谷歌都有自己的容器技术，亚马逊应该也有。如果容器会带来历史性的变革，那容器的战争是早晚的事。CoreOS 发布 Rocket 时称 Docker 已经忘记初心，并且开始冲击生态系统中的其它软件，于是便另起炉灶。现在过去一个多月了，还在 Stage 0 阶段，在社区方面获得的支持远不如 Docker 刚开始的时候，所以我并不看好。单从引擎这一点来看，类似 Rocket 的其它容器引擎并不是没有机会，因为 Docker 也有自己的问题（比如安全），新的一年，更是着重考验它们逼格的时候。

DockerCon 欧洲上，Docker 发布新的跨容器的分布式应用编排服务，旨在简化分布式应用的部署。分布式应用可以保证服务的稳定性和可扩展性，Docker 重新定义了分布式应用之道：我们可以根据需求随意收缩分布式应用节点。之前，Docker 教你怎么玩好一个容器，而现在 Docker 会教你如何玩好成千上万个容器，并在这众多容器上部署你的应用。当你在几百台服务器上部署几万个容器的时候，[Kubernetes](#)、[Mesos](#) 这些调度框架的作用就显现出来了，所以分布式应用相关的开源软件也是新一年的风向标。

回到国内，2015 年 Docker 会逐渐落地，经过去年一年的发展，Docker 已经相对稳定，研究和使用的人越来越多，并且中文资料也开始多了起来。现有的云计算厂商应该会重点关注 Docker，并可能在其底层架

构中使用 Docker，与 Docker 相关的 IaaS 和 PaaS 也会迎来一个创业小高潮，相关的开发者服务商也会提供对 Docker 的支持。

也许是看武媚娘看多了，每每去思考 Docker 发展的时候，我都会想 Docker 又何尝不是下一个“武媚娘”了？如果是，那“[武代李唐](#)”的预言会不会再次上演？这一次李唐们会如何做？好戏还在后头。

SDN 落地的实践与思考：带着问题找方案，别管定义啦

作者 张卫峰

编者按

本文系 InfoQ 中文站对盛科网络软件总监张卫峰的约稿。作者从自身做过的方案和所了解的业界情况为出发点，经过仔细思考，得出本文中对 SDN 当前现状的判断，所以文中难免涉及到作者所在公司之方案、产品，出于参考价值的考虑并未进行删节。若您就这个话题也有希望分享的内容，欢迎[向我们投稿！](#)

半年之前[写过一篇文章](#)讨论 SDN 的本质，当时就说这会是一个系列文章，后面还要写一下 SDN 的落地。这一等就是半年多，其间有朋友问我为什么还不写——不是我不想写，是有些问题还没想清楚。直到现在我也不敢说我完全想清楚了，但是毕竟有了更多的实践，实践过程中不停地思考，加上不断有媒体朋友找我约稿，我想还是写一写吧。这篇文章可以看作是对我三年 SDN 工作历程的一个总结和反思。我在这篇文章会回顾一下人们对 SDN 定义的争议，SDN 的落地实践，阻碍 SDN 落地的一些障碍，给出一些对 SDN 落地的建议和看法。

SDN 的定义回顾

现在大多数人对 SDN 的定义是控制跟转发分离+开放的编程接口，包括 Gartner 也是这样的定义。Gartner 的数据中心云计算行业分析总监曾绍清告诉我，他们认为思科的 ACI 不是 SDN，因为 ACI 并非是控制和转发分离，它只是把策略管理的功能分离到了控制器上，控制协议（OSPF、BGP 等）仍然运行在交换机上。但是我曾经在国外著名的通信技术媒体 lightreading 上看到他们综合一些专家的意见给出的定义，该定义里面只提到了开放的可编程接口以及由此带来的业务敏捷

性。就我个人的观点来看，我更倾向于 lightreading 的定义（具体请参阅[我的第一篇文章](#)）。不过，正如[青云 CEO Richard 接受 InfoQ 采访](#)的时候说过的，每个人对 SDN 都有不同的定义，这个并不重要，我深以为然，重要的是 SDN 带来的价值。所以，以后大家不要把精力浪费在讨论 SDN 的定义上吧。

SDN 在网络虚拟化中的应用

总有人说没看到 SDN 有落地案例，但是你去看一些国外专业的咨询机构，总能看到他们的报告中，SDN 的市场份额在逐年增加，且趋势向好。是有人在撒谎吗？No。

咨询报告中说的 SDN 市场份额在增加，主要是强调 SDN 现在最大的一个应用场景——网络虚拟化中的应用。很多人说没看到 SDN 有落地案例，那是因为他们潜意识里面只把控制器+硬件 SDN 交换机的应用认为是 SDN 应用，云平台+虚拟交换机他们认为不是 SDN。而事实上，以 VMware 的 NSX 为代表的网络虚拟化的应用早已经是被广泛认可的 SDN 的典型落地案例。

目前看到的基于 SDN 的网络虚拟化解决方案有以下三种：

- 1) 纯软件方式，以 VMware 的 NSX 为代表。除了 NSX，还有 Juniper 的 Contrail、Midokura 的 MidoNet 以及 Vyatta、Nuage、Plumgrid 等公司的商业网络虚拟化方案。这些公司的实现方式都不太一样，但是都在不同程度上用到了 SDN 技术。有的只是把一些策略管理的东西放在控制器上，转发表项还是由虚拟交换机自己来生成，而有的则是控制器来下发转发表项。而目前影响最广泛的 OpenStack 的网络组件 Neutron，则两种方式都支持，Neutron 更是一种标准的 SDN 架构。由于本文的目的不是介绍技术细节，所以这里就不深入展开来讲了。
- 2) 硬件方式，以思科的 ACI 为代表，即将网络虚拟化在硬件中实现（当然也不排除会用到 vRouter）。具体 ACI 的架构，我之前也写过一篇文章，可以参阅一下。
- 3) 软件+硬件方式。盛科网络推出的 SDN 方案即属于此类（Arista 也有类似方案），本质上它是一个软件方案的思路，只是把部分对性能影响最大的操作 offload 到硬件 SDN 交换机，可以认为是一个超级网卡。并且它为 NSX 之类的软件方案提供了 SDN 交换机作为 Tunnel Gateway 来满足物理服务器跟虚机混合组网的需求。

华为和华三也都相应的都有自己的解决方案，只是目前看到的他们都是推整体云计算解决方案，没有把网络部分整出一个方案来单独卖。

无论纯软件还是硬件的 SDN 解决方案，在云计算数据中心里面，应用的越来越广泛，所以如果要谈 SDN 的落地，这是目前最大的，最不容忽视的一个。

SDN 在别的领域的应用

除了在网络虚拟化领域的应用，SDN 交换机在别的领域也有一些应用，但是从应用广度和影响力来看，都比不上在网络虚拟化中的应用。从我们自己以及国外一些案例来看，落地的 SDN 的应用，其驱动力主要可以归结为两大类：业务层面灵活性的需求，转发层面灵活性的需求。前者的价值要远大于后者。

一、业务层面灵活性的需求

这主要是强调可编程。通过开放的可编程接口，提供给用户原来无法获得的对网络配置管理和策略部署的灵活性控制。前段时间著名的运营商亚太环通（Pacnet）[宣布在天津的一个 IDC 正式启用](#)，他们宣称里面使用了 SDN 技术来为用户提供自助调整带宽的功能，其实该功能早就部署在了他们新加坡、澳大利亚，香港等国家和地区的其他数据中心。该功能是通过定制化的 SDN 交换机实现的，其中的千兆交换机是盛科提供的。这是一个很典型的体现业务灵活性的例子。用户可以在他们提供的一个界面上，随时按需修改自己的出口带宽。而且不仅如此，一个用户可能租用了他们多个数据中心，通过 SDN 创建的 MPLS 隧道把用户在多地的数据连通之后，可以通过 SDN 动态调整这些隧道的带宽，一旦出现故障或者拥塞，还可以自动重新选路。没有 SDN，要做到这一步很难。

但是大家更关心的是 SDN 在企业网里面如何用。并不是所有企业网都适合使用 SDN，什么样的企业网需要用 SDN 呢？这个问题后面再分析。国外一个著名设备商 N，他们有挺多的 SDN 案例，特别是有些案例规模还是较大的，不像某些公司挂羊头卖狗肉的宣传，我至少知道他们有一个案例是很值得拿出来讲一讲的（这个案例在国外网站上也有介绍）。他们给某电视台的一个新的网络进行了 SDN 化设计，该网络有一个特点，就是拓扑和策略都是灵活易变的，比如这个星期是为一个大型演出节目准备的，而下个星期就变成为一个体育节目准备，

如果没有 SDN，他们需要靠人工去插拔线修改拓扑，重新划分物理和逻辑网络等，非常麻烦，在人工很贵的国外，这个问题特别突出。使用了 SDN 之后，整个物理网络基本不动，每次就依靠 SDN 将网络重构。这个案例还包括无线 AP，也是 SDN 化的。而且值得关注的是，他们并非全部使用 SDN，而是一个混合的网络，既有 SDN，又有传统的。即在需要 SDN 的时候 SDN，不需要的时候就用传统，深得 SDN 的精髓。

在我们碰到的案例中，也有一个复杂度没这么高，但是需要对网络灵活控制的。该网络管理员说他管理了一个较大的实验室，每天都有人在里面做不同的实验，对这些不同的人，网络中都需要有一些不同的安全控制策略，每次都去找他该配置，他不胜其烦。而这个时候，如果建立一套用户权限体系，用户可以自行登录申请，一旦认证通过，根据他的权限，控制器可以自行下发安全控制策略到交换机上，SDN 的业务灵活性充分体现出来。

二、转发层面灵活性的需求

这主要是针对一些非常特定的场景，主要是为了匹配或者修改特定字段，通常是传统交换机不支持的（其实芯片也许能支持，只是交换机系统没做）。这些场景我们也碰到不少，比如用来做 DDoS 防攻击（日本 Sakura Internet 的应用），用来做负载均衡+NAT，用来做 TAP 应用（价格是专业的 TAP 设备的至少 1/5），用来将 PPPoE 跟 IP 区分开并灵活控制等等。还有一些用户提出来过，但是需要辅助 FPGA 或者 NP 才能做到的。这类应用主要的灵活性体现在转发面上而不是控制面。

SDN 落地的障碍

硬件 SDN 的落地进展并不顺利。虽然现在慢慢有了一些更多的案例，但是离规模部署还很远。我跟 Gartner 的曾绍清一起探讨过原因，曾说 Gartner 经过调查，形成了一些他们的看法。

Gartner 的观点认为，以下几个问题阻碍了 SDN 的落地：

- 1) 厂商的直接支持而欠缺传统渠道的支持。
- 2) SDN 的革命性变革而使销售难度大增，传统厂商偏向销售 Ethernet Fabric 等容易接受的产品。

- 3) SDN 的用户价值较难从单一产品成本分析中体现。
- 4) 用户的开发团队开发的东西，运维团队不接受。

我个人觉得 Gartner 的观点都很有道理，相对来说看得比较宏观。我根据我们的客户交流和项目实践中碰到过的一些问题，也谈谈我的看法。我的观点其实跟 Gartner 有不少相通之处，算是一枚硬币的两个面。我认为一个用户要想把 SDN 在他的网络中落地，必须同时满足这三个条件，缺一不可。而现实中，这三个条件同时满足的不多，这也导致了 SDN 的落地缓慢。

- 1) 用户必须清楚地知道自己网络中存在的问题，然后带着这些问题来寻找方案。我经常碰到一些人问我，你帮我看看 SDN 能用在我们网络中什么地方？这种用户是没办法让 SDN 落地的。SDN 是用来实现用户的业务敏捷性的，不是用来全面替代传统网络的，如果你都不知道自己有什么问题，怎么引入 SDN？我碰到的最终能落地的，都是明确知道自己网络中的问题，迫切想找方案来解决的。
- 2) 用户做决策的人必须要足够有魄力，而且能够协调开发部门（或者第三方开发）和使用部门之间的关系。某互联网大厂告诉我，他们的自研交换机项目之所以能成功，全面在自己的网络中替换商用交换机，就是因为他们的研发和运维都归一个领导管，这个领导要求运维部门必须用自己研发的交换机，有问题也在所不惜。而其它大厂之所以进展不顺，则恰好相反，研发部门和运维部门彼此独立。SDN 这里也是如此，如 Gartner 所言，SDN 的革命性变革，必然导致传统运维使用上的不适用，人都有使用自己习惯的东西的惰性，如果没有强制命令来保证运维人员使用新的工作方式，确实会比较难推。盛科就给一个互联网大厂做过一个 SDN 项目，该项目很顺利地解决了一些核心技术需求，但是反倒是在推到运维那里的时候碰到了障碍。其实那些障碍可大可小，如果严格按照传统运维规则去要求，那就会阻碍重重，但是如果愿意给与新生事物足够的耐心，让它在使用中慢慢完善，那就既可以顺利推行下去。这都取决于决策人员的魄力和权责范围。
- 3) 用户的研发部门或者第三方研发人员必须有足够的研发能力，能够有充分的理性选择合适的技术。整个 SDN 体系中的核心是什么？是交换机吗？是控制器吗？都不是！核心是应用程序。在 SDN 中，用户自己或者用户委托的第三方必须有足够的能力去研发上层应用软件，必须知道这些应用软件如何去通过控制器控制交换机。很多人通常会问 SDN 交换机厂商：你们除了交换机，还有控制器卖吗？我假设我们有，你拿去就能用吗？不能！因为设备商提供的控制器不知道用户要用来做什么应用，所以它实际上

提供的只是一些基础 API 以及实现这些 API 的内部逻辑，如何用这些 API，那是用户或者用户委托的第三方需要去考虑的事情。国外的 SDN 为什么部署得比国内多？至少我看到的原因之一是，国外有一些独立的第三方的 SDN 应用提供商，他们有能力架设起最终用户和 SDN 设备商之间的桥梁，把用户需求和 SDN 设备以及控制器结合在一起，打包交付给用户。比如前面讲的亚太环通的 SDN 应用，就是一个第三方软件提供商把盛科 SDN 交换机、另外一个厂商的 SDN 光传输设备、开源的控制器加上他们自己的应用程序结合在一起，一起交付给客户。而且他们进行技术选择的时候，非常理性不会刻意地去追求标准，他们追求的是满足客户需求，所以有不少私有化的扩展。盛科推到欧洲、日本、美国去的 SDN 设备，也都是因为有强有力第三方合作伙伴或者客户本身有强有力的研发能力。否则 SDN 交换机只能在实验室玩玩。我也很遗憾地看到过一些反面例子，本来他们或者他们的客户确实有 SDN 的需求，但是他们自己不愿意或者没有能力去针对控制器做二次开发，也不愿意花钱去请第三方开发，而在没有量的保证的时候，设备商也不愿意去做太多定制开发，最终导致落地受阻。

OpenFlow 的局限性

OpenFlow 是最广为人知的 SDN 技术，但是并非唯一。而且实践证明，仅仅靠 OpenFlow，很多事情做不了，OpenFlow 可以应用的场景非常狭窄。

关于 OpenFlow 本身的技术缺陷，很多文章都提过，我之前的书和文章里面也都详细分析过，诸如当前交换芯片支持的流表数量都有限，报文匹配和动作都不够灵活，都无法支持很多级流表等等。这些分析都是对的，但是我要告诉大家的是，这些限制根本不足为惧。为什么这么说？因为这些都是从 OpenFlow 技术规范出发得出来的结论，而不是从 SDN 应用的角度得出来的结论，换句话说，SDN 的应用，未必真的需要 OpenFlow 规范里面提到的所有技术，所以就算有限制，问题也不大。OpenFlow 真正的限制在别的地方。

运维管理的缺失

还是以我们给那个互联网大厂做的项目为例，该厂所要求的一个核心技术点别的交换机都做不到，只有盛科的能做到（因为盛科是用自己的芯片，恰好支持该功能），而且该技术也能按照客户要求使用

OpenFlow 配置出来，一时皆大欢喜。但是当该产品要转运维的时候，问题来了，运维部门要求所有入网的设备，都要满足他们的运维要求，诸如 SNMP 管理、能够查看统计、能够 ping 通该设备、能够 telnet 该设备、能够通过 Radius 到远程服务器进行管理员身份认证等，这些对交换机来说是再正常不过的基本需求了，但是所有这些东西在 OpenFlow 上都没有定义。当然你可以辩论说这属于管理面的，不属于 OpenFlow 的定义范畴，OpenFlow 只定义转发面和控制面功能，但是管理层面的不少功能依赖于转发面，比如管理员想通过带内 ping 通交换机以便检验路径的可达。还有运维人员希望交换机能支持基本的 LLDP 协议来进行邻居发现。另外一个互联网公司也给我们提出过类似的要求。

运维管理功能的缺失导致了传统运维人员的抵触是可想而知了。这光靠 OpenFlow 是无法解决的，需要引入传统的东西。

跟传统网络的交互

用户网络中通常都是存在很多传统设备的，不太可能为了引入 SDN 而把这些设备都抛弃，所以这就涉及到一个问题，需要 SDN 设备跟传统设备互通。比如有一个三层汇聚交换机，该交换机会向下发送 ARP 获取下联设备的 Mac，如果下面是个传统的主机或者三层设备，它会自动回复 ARP 请求，但是 OpenFlow 交换机没这能力，它只能把报文发送到控制器，让控制器回复，但是很多用户不想在控制器上进行开发来支持这种非核心业务。而且，实事求是的说，最高效的做法肯定是在交换机上进行回复。

还有更复杂的例子。曾经有一个软件开发商，使用盛科的交换机给一个电商开发 WAN 网的流量调度，它需要跟传统交换机进行路由协议交互，如果不在交换机上运行路由协议，就要在控制器上运行。在控制器上运行路由协议，会让控制器很复杂，而且效率低下，况且，这也并非该软件提供商的核心价值，他们也没这个能力去在控制器上做一个路由协议并把它做稳定，所以他们希望交换机上做。SDN 交换机对他们来说，核心价值是让他们可以控制报文的转发路径，从而动态调度流量，至于跟传统网络的交互，他们不希望重复发明轮子，而是希望借用交换机的传统能力。

以上两个问题，并非是说靠纯 OpenFlow 交换机完全无法满足，如果在控制器上做得足够复杂且不考虑效率，也是可以做到的。我们就有

一个云计算的客户，使用我们的纯 OpenFlow 交换机，完全靠自己开发的控制器去进行必要协议报文交互（主要是 ARP）和各种其它必要的控制，他们之所以能做到这一点，就是因为他们本身有很强的研发团队。对于大多数人来说，要走这条路是很难的，那解决方案是什么呢？就是同时支持 OpenFlow 和传统二三层处理的混合型交换机。

SDN 落地的建议

根据以上的分析，为了加快 SDN 落地，对用户、对 SDN 提供商、对整个行业，我有如下建议。

- 1) 要清晰地认识到 SDN 并非适合所有场景。什么样的场景适合 SDN？前面说过，SDN 应用的两大驱动力：业务层面灵活性的需求和转发层面灵活性的需求，如果你的网络足够复杂（复杂并非是规模大），一些配置管理、安全策略、流量调度策略、拓扑等经常需要变化，那非常适合 SDN，最典型的就是网络虚拟化（频繁的虚机增删、虚拟网络的变化）以及 Google B4（路径经常需要随着带宽的变化而变化）。或者你的网络中某个特定功能，在转发面上需要灵活的报文匹配或者报文编辑，传统网络的固定模式无法满足，那也可以考虑 SDN。
- 2) 对于普通企业来说，我的建议是不要追求 SDN 设备接口的标准，而是要追求接口的开放性和灵活性，因为你想需要的不是技术标准，而是要能解决你的实际问题。对于运营商或者必须要求引入多家设备提供商的大型互联网公司，如果你要引入 SDN，不要去追求南向接口的标准化，而把精力放在北向接口的标准化上。通过让每个厂商提供插件来适配北向接口的做法，来屏蔽各个厂商的差异，这是最现实的做法，否则推动起来会阻力重重，因为各个厂商都不愿意提供跟别人完全一样的设备编程接口。这一点上可以借鉴 OpenStack 的网络组件 Neutron 的做法。
- 3) 正确认识 OpenFlow 的作用。不要指望纯 OpenFlow 能够解决你所有的问题。真正能给复杂网络带来价值的 SDN 设备必定是混合型设备，而且这种混合不是简单的部分端口支持 OpenFlow，部分端口支持传统路由交换，而一定是在报文处理流程中，OpenFlow 和传统二三层处理混合在一起。让 OpenFlow 去控制你想控制的部分，其它部分对你来说无业务价值，就让它们走传统处理流程就可以了。这样跟传统网络互通性的问题，运维管理的问题也都很容易就解决了。
- 4) 不要期望整个网络全部都 SDN 化。SDN 的价值不在于让用户控制一切，而在于让用户去控制他需要控制的地方，无业务价值的部

分，完全不需要 SDN 的参与。无业务价值的部分，有的时候存在于边缘，有的时候存在于汇聚和核心，完全看场景而定。

- 5) 如果有人要进行 SDN 创业，创业的着眼点一定不要放在 SDN 交换机和 SDN 控制器，而是要放在 SDN 应用上。控制器你可以找一个开源的拿过来修改一下就行，比如 OpenDayLight, Ryu 等，SDN 交换机可以跟专业的 SDN 交换机厂商合作，但是应用部分是离最终客户最近的，最能体现价值的部分。SDN 的落地需要这样专业的第三方 SDN 应用提供商。
- 6) 不要动辄问设备商你的设备是否支持匹配 12 元组，是否支持多级流表，否则我会反问你，你为什么需要匹配 12 元组？为什么需要多级流表？不要只把支持 OpenFlow 的交换机认为是 SDN 交换机，没支持 OpenFlow 就认为是忽悠你，你要问的是，它开放的可编程接口是否能满足你的需要。同理，不要以为控制器就应该是支持 OpenFlow 的，不支持 OpenFlow 的控制器你就认为是忽悠，你要看的是它是否能通过开放的接口去控制交换机。对于 OpenFlow 交换机，不要认为只有使用 ACL 表实现的 OpenFlow 才是 OpenFlow，使用传统表项组合出来的流表就不是 OpenFlow，就是忽悠，你要问的是，使用传统表项组合出来的流表是否能满足你的需求。
- 7) 无论是用户，还是 SDN 设备、方案提供商，一定不要期望你可以做一个批量复制的东西出来，SDN 必然意味着定制化。这是一把双刃剑，一方面它可以通过定制给用户提供真正的灵活性，但是另外一方面，太多的定制导致它难以被快速推广，大型设备商的规模优势无法体现，无法依靠传统渠道去推广而不愿意去定制，而小的设备商限于人力，也没法去做太多定制。所以需要专业的第三方提供商的出现。
- 8) 在没有规模部署的前提下，不要去期望 SDN 设备会有成本优势，相反，因为定制化的研发投入，SDN 整体方案的成本反而会增加。对于用户来说，要关注的是 SDN 所带来的运维成本的下降。
- 9) 如果你要部署 SDN，必须打消买过来就能用的不现实的期望值——至少目前是这样。在你立项或者购买 SDN 设备之前，你必须要问自己，Am I ready? ready 的意思就是你需要自己有懂业务的研发团队或者愿意购买第三方的服务，或者愿意付钱让设备商给你做定制开发（如果设备商愿意的话）。

总结

我们要正确地认识 SDN，不要过高估计 SDN 的能力，也不要对 SDN 丧失信心。SDN 不会取代传统网络，甚至看不到它有占据垄断地位的

可能，但是它肯定会是现有网络的一个强力补充。SDN 落地不要太在乎标准化，要着眼于开放性。SDN 落地不仅呼吁第三方应用提供商的出现，更重要的是，SDN 用户企业中的决策者，要有足够魄力，敢于承担风险，愿意在使用中完善 SDN，要勇于拍板。国外的 Google，Facebook 有这个魄力，NTT 有这个魄力，Verizon 有这个魄力，Pacnet 有这个魄力，国内的公司没理由在这方面落后于他们。我们欣喜地看到国内某些公司已经在赶上，腾讯就是一个很好的榜样。

最后也要给所有要学习 SDN 的朋友，特别是学生朋友一个建议：学习 SDN，必须要有基本的网络知识作为基础，不懂网络就想学习 SDN 这是不现实的。

作者简介

张卫峰（[@盛科张卫峰](#)），盛科网络软件总监，数据通信和芯片设计领域资深专家，有十几年的网络实践经验，对 SDN、传统二三层交换机、数据传输设备（PTN 和 IPRAN），从管理面到协议控制面一直到芯片转发面都有着深刻的理解。

Swift 编程语言

作者 Gustavo Machado, 译者 孙镜涛

在过去的几年中，移动应用程序风靡全世界并且已经改变了我们使用互联网进行工作或者休闲的方式。为了创建移动应用程序，各种技术应运而生，同时开发过程也开始将其作为一等公民来对待。尽管移动似乎已经无处不在了，但是它的未来才刚刚开始。我们正面对着新一代的移动设备，例如可穿戴设备以及组成物联网的大量移动工具。我们将会面对新的用来展示数据和接受命令的用户界面。同时，我们将会看到越来越多的公司真正地实现移动优先。所有的这一切都将会影响我们在未来的几年中设计、开发和测试软件的方式。

InfoQ 的这篇文章是快速变化的移动技术世界中一系列文章的一部分。你可以通过这里订阅这一系列，届时如果有新文章发布那么会通知你。

苹果公司最近推出了 Swift 1.0——一门针对 iOS 和 OSX 开发的新编程语言。不要将苹果的 Swift 与老的并行脚本语言混淆。Swift 的目标是让 iOS 和 OSX 开发变得更简单，更有乐趣。在本文中，我将会解释我认为 Swift 所具有的最具杀伤力的 5 个特性以及我为什么会这样认为的原因，虽然这些特性现在依然出于测试阶段，但是却值得我们一试。

苹果已经拥有了一门编程语言——Objective-C。那么为什么还要引入另一门编程语言呢？这是因为虽然 Objective-C 在被创建的时候可能已经非常地独特，同时也很先进，但是它现在并没有当今语言的味道。例如，在消费者方面像 Ruby 这样的脚本语言已经被广泛采用，这很大程度上得益于它干净的语法。在企业领域，具有类型推理能力的强类型（类型安全的）语言更受欢迎，为了将函数式编程语言所具有的函数即对象、Lambda 表达式等经典特性引入进来，C# 和 Java（或者 Scala）等语言都做出了大量的努力。Objective-C 一直都缺少这类东西，例如干净的语法（和语法糖），类型推理。而 Swift 正是为了填补这个空白。

这并不是说 Objective-C 并不是一门优秀的编程语言。实际上，它是一门优秀的语言。但是我确实看到有足够的空间可以成功地替代 Objective-C。进一步讲，同时也要感谢 Swift 的优秀，我认为 Swift 一定会像野火那样迅速蔓延开来。

现在，就让我们看看 Swift 都提供了什么吧。从语言的角度看，Swift 是非常了不起的。苹果借鉴了 Scala 和 C# 这些现代语言的优点，构建了一门非常简单，但是功能非常强大的语言。它非常完美地融合了面向对象和函数式编程范式——但是说 Swift 是一门函数式语言是一种极大的延伸。下面就让我们看看 Swift 最具杀伤力的 5 个特性。

语法糖

从语法上讲 Swift 非常华丽。它是一门非常简单、干净的语言，同时可读性也非常好，即使以现在的标准来衡量也是如此。你马上就会发现在设计一门语言的时候简单性是一个关键要素。例如，大家所熟知的语句末尾的分号。苹果决定将分号作为可选的，虽然这看起来相关性并不是非常强，但是它却让我们看到了苹果为了尽可能地保持语法干净所做出的努力。

简单性方面的其他例子包括字符串插入，以及语言对数组和循环处理的支持。

字符串插入

```
var message = "Hello World" "The message is \(message)"  
//The message is Hello world  
var a = 1, b = 2 "The sum is \(a + b)"  
//The sum is 3
```

循环

```
var length = 10  
for i in 0..
```

数组

```
var list = ["a", "b"]  
list += ["c", "d"]
```

以上仅是 Swift 为简单性提供语言支持的一部分示例。需要注意的是你依然可以使用 Array 类的“append”方法连接数组，而苹果之所以走了

额外的一英里将其构建为语言的一部分目的是为了展示他们设计 Swift 的目标。

如果你想学习 Swift 并对一些这样的示例代码进行测试，那么可以尝试下 Xcode 6，它的代码实时预览（Playground）功能太酷了，简直无法用语言形容。实时预览功能让你能够实时地随着你的输入对代码进行测试。它会执行你在开发环境中输入的所有内容，提供与变量值、函数调用返回值以及特定代码块被执行的次数相关的详细信息。打开 Xcode 6 中的实时预览功能非常简单。



下面的图片展示了实时预览功能：

```

kido.playground — Edited
kido.playground > No Selection

1 // Playground - noun: a place where people can play
2
3 import UIKit
4
5
6 func info(items:[Int]) -> (avg:Int, min:Int, max:Int) {
7     var sum = items.reduce(0, { $0 + $1 })
8     var min = items.reduce(Int.max, { $0 > $1 ? $1 : $0 })
9     var max = items.reduce(Int.min, { $0 < $1 ? $1 : $0 })
10    return (sum / items.count, min, max)
11 }
12
13 var result = info([1, 2, 3, 4, 5, 6])
14 result.avg //3
15 result.min //1
16 result.max //6

```

函数是一等对象

越来越多的语言将函数作为一等公民并支持高级函数。例如，最近发布的 Java 8 引入了 Lambda 表达式。它的理念很简单，就是让函数可以接受函数类型的参数，同时也可以将函数作为返回值。理念的简单性奠定了它强大的基础，因为这样支持更多的抽象。例如，我们可以将一个“过滤（filter）”函数应用到一个数组，该过滤函数接受数组中的每一个条目作为参数，通过特定的标准对条目进行判定从而完成对给定数组的过滤。使用更加通用的方法的关键是能够接受函数类型的参数。下面就让我们看看定义函数的语法。

Swift 定义函数的语法与传统的 Haskell 这样的函数型语言相似，但并不完全一样。箭头 (\rightarrow) 的左边是参数以及参数的类型，右边是返回值类型。在本文的示例中，我们想要过滤一个数字列表，因而基于一个给定的数字我们会返回一个 Bool 类型。在这种情况下，函数看起来可能是这样的：

```
(Item:Int) -> Bool
```

这段代码的意思是接受一个 Int 类型的参数，返回一个 Bool 类型的值。很显然，你可以包含多个参数，但是不太明显的是，你还可以返回多个值，而这不需要创建一个容器对象。在后面的示例中，函数会返回一个元组。

一个过滤整数的函数定义可能是这样：

```
func bigNumbersOnly (item:Int) -> Bool {  
    return item > 3  
}
```

现在，我们已经创建了自己的过滤函数，下面让我们看看可以接受函数参数类型的“filter”函数。

```
var numbers = [1, 2, 3, 4, 5]  
var bigOnes = numbers.filter(bigNumbersOnly)
```

在这个示例中，filter 是一个高阶函数，因为它的参数类型是函数。在 Swift 中，函数也是对象，这意味着我们可以定义内联函数：

```
var numbers = [1, 2, 3, 4, 5]  
var bigOnes = numbers.filter({(item:Int) -> Bool in  
    return item > 3})
```

或者我们也可以将函数赋值给某个变量，稍后再使用它：

```
//define two variables of type function
var biggies = {(item:Int) -> Bool in return item > 3 }
var threes = {(item:Int) -> Bool in return item == 3 }
//decide which one to apply at runtime
var result = numbers.filter(onlyThrees ? threes :
biggies)
```

当今，将函数作为对象，让用户能够像使用参数那样引用、传递函数已经成为一种优美的标准。而 Swift 实现方式的简洁性依然是一个值得称道的地方，这一核心概念配合类型推理可以让你事半功倍。

强类型与类型推理

在企业开发领域，我们非常习惯于使用强类型（或者说类型安全的）语言。强类型语言通常会给我们带来一点额外的自信，因为它能够在编译时进行错误检查，如果这些语言也能够支持类型推理那么将会是一种非常好的体验。例如，可以这样：

```
//Without Type inference
var x:String = "bar"
//With Type inference
var y = "bar"
```

注意，第二行的语句并没有声明变量的类型。因为 Swift 知道“bar”是一个字符串，所以我们并不需要显式地定义它的类型，当然我们也可以指定，正如第一个语句那样。这看起来好像并没有特别大的作用，但是如果推理的是函数的类型那么它就会变得十分有趣。

那么如果不使用类型我们应该如何定义之前例子中的函数呢？下面的代码展现了 Swift 的实现：

```
//$0 will map to the first parameter, $1 to the
second...
var result = numbers.filter({ return $0 == 3 })
```

没有比这更简洁的了！如果你的参数不止一个，同时想要通过名字引用参数，那么可以这样做：

```
{a, b in return a > b }
```

泛型

Swift 提供的另一个非常便利的特性是泛型。在企业开发领域，泛型首先被引入到了 C# 中，在获得了大量的关注之后 Java 也引入了该特性。使用泛型可以让开发者消除类型转换，因为编译器能够运行特定

的类型检查，而对于不支持泛型的语言而言这是无法做到的。虽然刚开始将泛型引入 C#的时候确实产生了一些争论，但是现在它已经被 C# 和 Java 社区所普遍接受。

泛型提供了一种方式可以让我们推迟类型的定义，通常（但不限于）是参数和返回值的类型。虽然它听起来很复杂，但是实际上通过一个简单的示例我们就能非常容易地理解它。

```
//At the moment of creating this function
//I am not defining what T is. The actual
//Type of T will be deferred to the call
//of the doNothing function.
func doNothing (item:T) -> T {
    return item
}

//When I call doNothing, I am implicitly
//setting the type of T to the Type of the
//parameter I am sending. In this case,
//an Array.
//Notice how the compiler knows, that the
//return type is an Array.
doNothing([1, 2, 3]).count
```

虽然上面并不是一个真实的例子，但是通过它我们能够看到在编译时确定数组内部元素类型的便利。下面是一个更简单的示例，注意编译器是如何知道数组是包含字符串类型的：

```
var list = ["hello", "world"]
list[0].uppercaseString //HELLO
```

泛型的使用范围并没有被限定于参数，我们也可以定义泛型类、枚举和结构。事实上，在前面的示例中，list 的类型是 `Array<String>`。

你可能会将 Swift 中的泛型与 Objective-C 中的协议类比，虽然它们的语法非常相似，但是概念是非常不同的。Objective-C 并不支持泛型。协议提供了一种方式去声明一个确定的实现符合某个消息契约，但是契约必须预先指定。例如，使用协议你并不能强迫数组中的所有条目都是同一类型的（无论是什么类型），但是使用泛型能够做到。除了概念上的不同之外，Swift 并不支持协议，就像 Objective-C 不支持泛型一样。

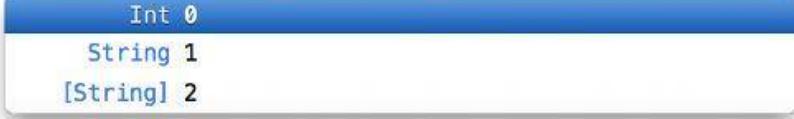
元组

元组是非常简单的概念，你可以定义一个有序的值组。当你需要将多个值作为一个参数来回传递、或者被调用的函数需要返回多个值的时候元组会非常有用。元组并不需要我们为它的值定义任何类型，编译时会完成所有的类型推导和类型检查工作。定义元组的语法如下：

```
(1, "Two", ["Three"])
```

在上面的例子中，我们创建了一个带有三个值的元组，第一个是一个整数，第二个是字符串，第三个是一个字符串类型的数组。乍一看这很像一个数组，但概念完全不同。你不能从一个元组中删除或者向里追加元素，同时注意编译器是如何知道每一个值的确切类型的：

```
5   (1, "Two", ["Three"]).0
6       Int 0
7       String 1
8       [String] 2
9
10
11
```



你可以通过元素在元组内部的位置引用它的值，正如图片所提示的那样；或者你也可以为每一个值指定一个名称。如果一个函数需要返回几个值，那么这是非常方便的，同时它也能让我们避免定义那些特定于某个函数的类或者结构。下面让我们看一个这样的例子：

```
func info(items:[Int]) -> (avg:Int, min:Int, max:Int) {
    var sum = items.reduce(0, { $0 + $1 })
    var min = items.reduce(Int.max, { $0 > $1 ? $1 : $0 })
    var max = items.reduce(Int.min, { $0 < $1 ? $1 : $0 })
    return (sum / items.count, min, max)
}

var result = info([1, 2, 3, 4, 5, 6])
result.avg //3
result.min //1
result.max //6
```

元组提供了一种非常简单的使用多个值的方法，让我们省去了定义一个特定的类或者结构的额外工作。

还有更多.....

除了上面介绍的特性之外 Swift 还有很多其他的优秀特性值得我们一看，例如属性观察器、可选链接以及扩展。

我相信 Swift 具备快速成为一门流行的 iOS 和 OSX 编程语言所需要的所有必须条件，无论是在企业领域还是在消费者领域。强类型和类型推理特性将会让它非常适合于企业开发，而它的简单性和干净的语法则会吸引那些从事消费者项目的开发人员。

关于作者

Gustavo Machado 是 KidoZen 公司的工程副总裁，他的工作是领导大家开发公司的下一代企业移动平台。他在通过不同的技术开发高度分布式系统以及敏捷实践方面拥有丰富的经验。他一直在积极地维护自己的[博客](#)，同时他的 twitter 帐号是@machadogj。

查看英文原文：[Swift Programming Language](#)

手机淘宝构架演化实践

作者 **臧秀涛**

2014年12月19日~20日，[ArchSummit北京2014大会](#)顺利举行。“[移动互联网，随时随地](#)”是非常火爆的一个专题。阿里无线事业部技术负责人[庄卓然](#)（花名南天）任出品人。来自阿里无线事业部的高级专家[李敏](#)（花名心石，微博：[@allblue 华丽地低调](#)）分享了《手机淘宝架构演化实践》（[幻灯片下载](#)）。

李敏主要负责淘宝无线客户端和无线网站基础服务、购物主链路的架构、研发方面的工作。从09年开始参与手机淘宝研发团队的组建和线上产品研发，先后负责过无线部门的社区、会员、营销、交易等多条产品线的技术工作，构建和发展了阿里无线技术体系中包括交易链路、百亿级别高性能API网关、WebApp平台等多个重要技术产品，经历和见证了阿里巴巴无线从开始之初到成为日活上亿级别电商应用技术变迁和积累。

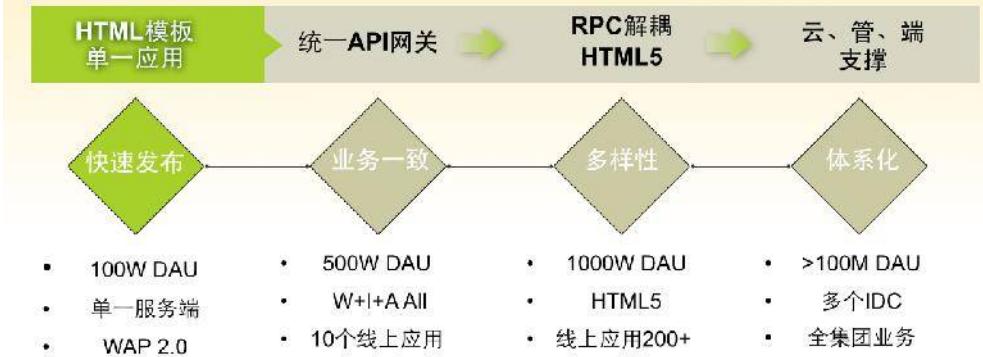
本文即根据李敏的演讲整理而成。

发展阶段

从2009年开始，DAU从100万增长到超过1亿，面临的问题、包括研发支撑所需要解决的事情各不相同。在用户量和业务复杂度的线性递增下，架构也进行了相应的演进。如下图所示，具体可以分为四个阶段：

走过的路

手淘在用户量和业务复杂度的线性递增下
架构也进行了相应的演进



- 第一阶段，手淘的前身 WAP 网站，业务初立、变化快，需要快速发布，采取 HTML 模板和单一应用，最大程度满足快速发布和修改的需要；甚至不需要改动后端的业务代码，在前面的模板上做一些修改就可以了。
- 第二阶段，DAU 的快速增长，WAP/Android/iOS 多个平台的业务起来了，需要在多个平台上进行快速的业务复制和业务管控，统一 API 网关出现。
- 第三阶段，DAU 进一步增长，线上系统越来越多，业务的多样性需求更多的体现出来，基于 HTML5 的一整套解决方案上线，更多的 HTML5 和 Native 混合的业务形态，API 网关进行进一步优化和扩展，更方便的接入方式。
- 第四阶段，当 DAU 达到 100M 的时候，全集团的业务都需要在手淘透出，API 网关被部署到更多的 IDC 机房，如何更有体系化的进行有效的研发、接入更多业务、并进行更有效的业务监控，需要更加体系化的架构治理。

API 网关

做 WAP 的时候没有所谓的 API 网关，为什么要用 API 网关呢？

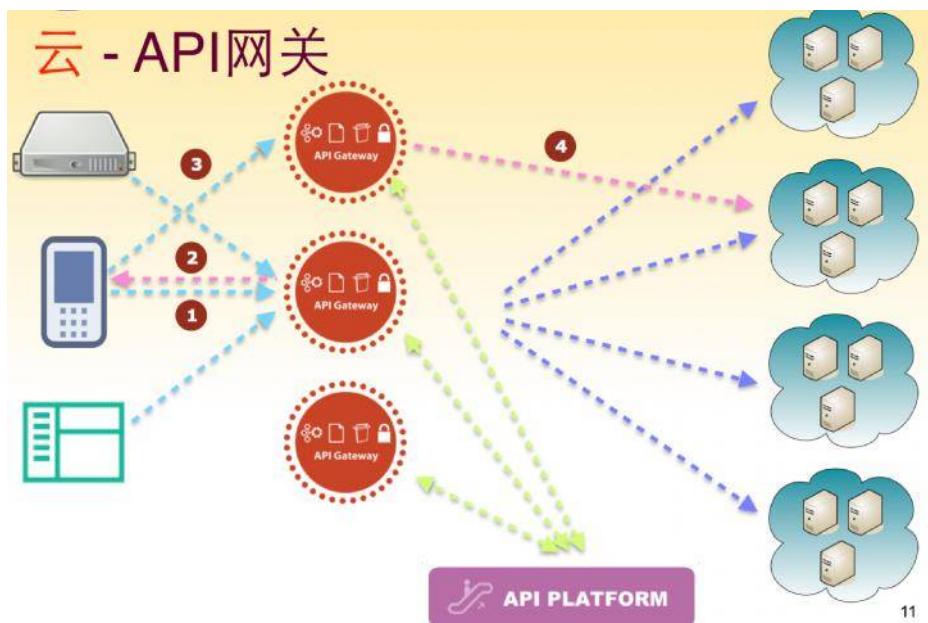
随着应用数量的增多，每个应用分别暴露的 API 出口很多，修改的话逻辑很复杂，这时候应该引入一个统一的网关。

但随着 DAU 的增长，API 网关会成为一个单点。开发团队在中间做了很多技术和架构上的努力，主要有几个关键点。一是后端接入很多应用，其实 API 网关只是通路，理论上不存在调用的上限，只要内存够

大，包括网卡的流量够的话都可以上来。二是有必要的机制做到宽阔的调用网关。还有一点，当后端业务要经过 API 网关时，其实现在业界很多都是典型的 RPC 的模式，RPC 的模式有一个绕不开的问题，就是可能要设定一些东西，这时后端服务跟 API 会有一定程度上的耦合。现阶段要接入服务，后端服务器随时都会变化，不可能后端服务变化的时候都对 API 做相应的发布，这是不现实的。所以有一套自己的 RPC 机制，解除了这种强类型的约束。

此外，可以在网关上附加很多功能，比如安全、审计，还有一些日志、审查等。

到了现在这个阶段，要进行异地部署，很多 IDC，这样的话引入 API 网关很可能会带来问题。包括今年的双 11 或者是双 12，要在多个异地机房支撑手机淘宝的业务，会有很多 API 网关。

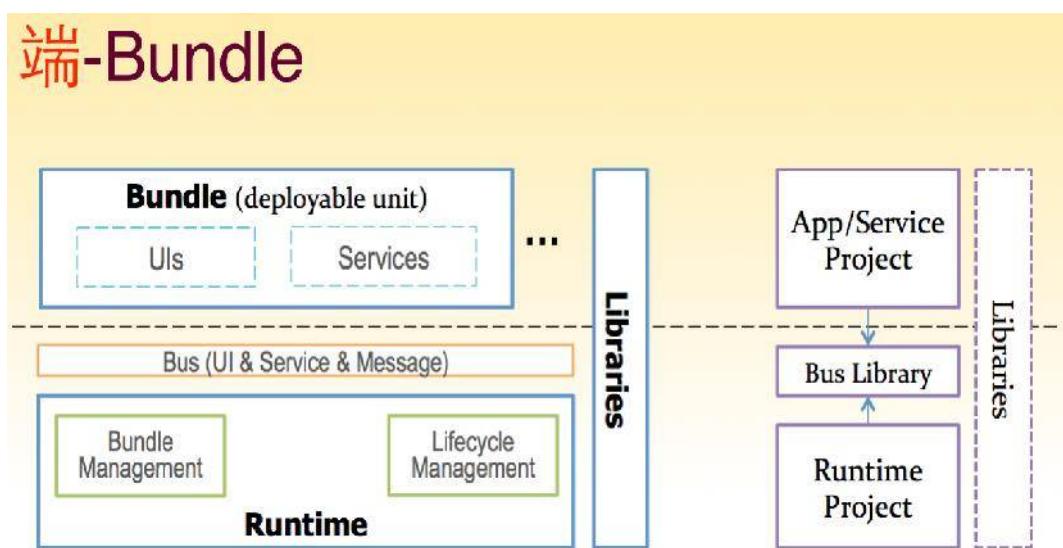


比如说像 APP 可以在中心网关上面询问，应该去哪个真正的 API 网关。然后中心 API 网关会告诉它结果，它再连接到所在地的 API 网关上，然后再向后端 API 发起调用，所有 API 的服务网关都受管控中心统一管控。比如说增加一些新的功能，上线一些新的 API，包括一些引流、切换，这些指令都会在管理平台上向各个 API 网关发送。

手机端

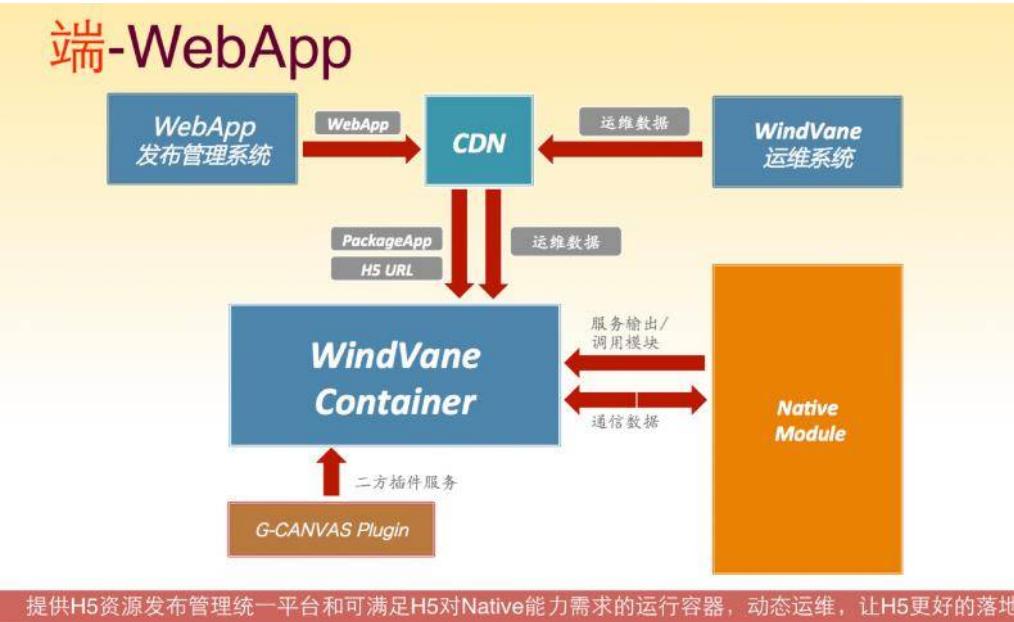
1.Bundle

去年下半年，开发团队对整个手机淘宝的架构做了比较大的调整，如下图所示，左侧是运行时的架构分布，右侧是工程代码级别的分布。在运行的时候，其他的业务团队提供的都是一个个的业务 Bundle，这是可部署的单元，包括 UI、服务和标准中间件的调用代码，下面有一个总线程，负责管理和开发好统一的 UI 服务，包括消息服务的总线。再下面是运行容器，上面跑的是所有的 Bundle 的东西，对应运行时的东西，右侧是真正在开发时候的结构。比如说聚划算，它要开发它的业务，就做一个单独的工程，然后去开发；它只用开发自己的，开发到差不多的时候，就将其代码打成一个 Bundle 提供过来，然后一起打包发出去。



2. WebApp

下图是现在手机淘宝上关于 HTML5 的整体框架图。手机淘宝上的方案大致分为两部分，中间那一部分是手机淘宝自己开发的 HTML5 的运行容器，它负责在上面跑各种各样的 WebApp，在线上有一个统一发布管理系统，它可能对性能进行检测，包括 CDN 是否符合规格，HTML 本身有没有异常等情况，经过这些必要的检测，包括审查之后，它统一发到 CDN 上。容器本身其实也会接受运行时的信息，容器接收到这两边的指令之后，它自己会做一些更新配置，也可能会装载运行，从线上系统下发新的 WebApp。此外，还可以运行 WebApp 或者是做 URL 的导航拦截，甚至做一些交互。



3.PackageApp

这是今年新的建设，整个系统是基于前面整个体系来做的，称之为 PackageApp。

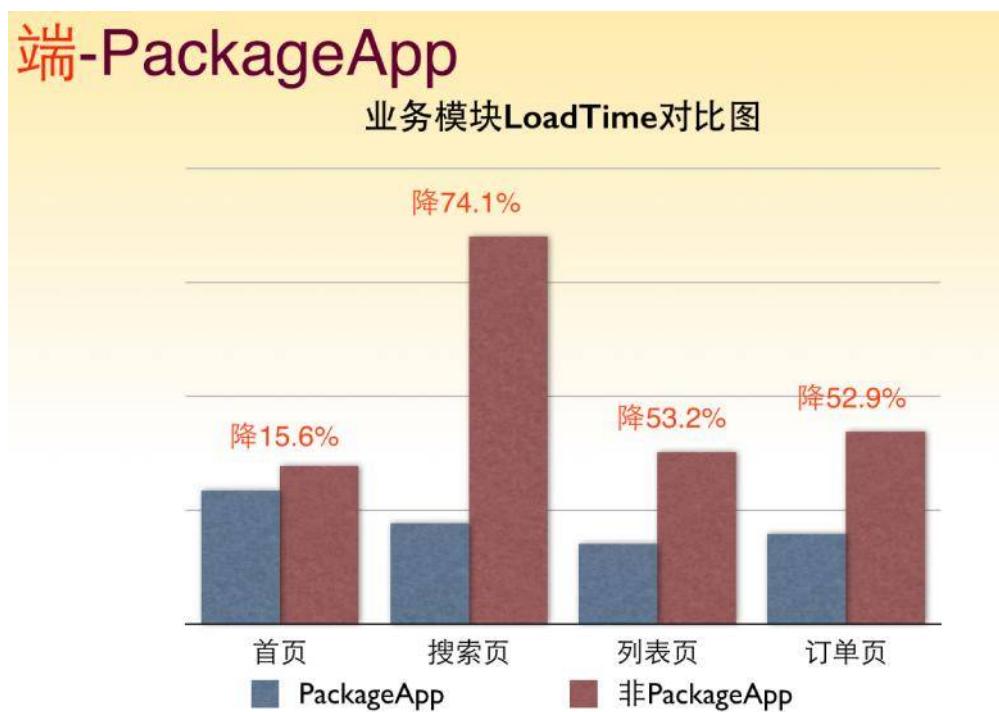


这个跟前面最大的区别就是让用户感知不到前面同步下载的过程，大概的做法是：把 HTML5 以及 WebApp 在发版之前先做一些预知放到客户端里面，前面会做两件事情，首先按照原来的逻辑运行，其次就

是在右侧的蓝图里面，它会去做一些 UI 的拦截，发现用户点击的 icon 进去之后，整个 URL 是符合用户规范的，它会启动检测机制去检查线上是不是有新的版本需要下载，如果说有的话会启动异步更新模块，从 CDN 上拉取新的 WebApp 版本，否则会走到原来的地方，最后既不影响用户去使用 WebApp，又能把自己最新的版本更新到所期望的版本，这由统一的管理平台去发布。

在方案设计之初，还思考了三个方面，首先它是标准的 URL，在点击进去之后是导航的 URL，对于前端工程师来说，他设计研发 WebApp 跟客户端或者是线上跟 HTML5 的网站是一致的。其次，手机淘宝自己的容器，制定了自己的规范，在底层的容器上面可以实现手淘定义的规范。第三，“不同网络、全量、差量更新”，这点很重要，在移动互联网场景下，到底要发起几条链接拉取资源，在 WIFI 下怎么拉取资源，其实都是不一样的。在不同网络下面，对策都不太一样。

下面是采用 PackageApp 后业务模块 LoadTime 对比图：



支撑体系

除了前面介绍的内容，比较大的电商 App，还需要一个很完备的支撑体系。如果没有的话，在线上运行的情况是不可感知的。手淘在不同的维度也做了很多支撑的工具。

1.研发支撑

在研发支撑上面，像传统的 Reivew 代码，特别是做客户端的同学几乎都会做统一的 UI 库，大家会设计模板，比较典型的，会有所谓的日常预发、线上染色等等。它的集群数量跟能够进来的用户是很有限的，通过这个环境来确认所开发的代码发布到线上可能会有什么问题。一套代码经过预发之后再发布到线上去，最后有一个染色环境，比如说用户打电话反馈遇到的问题，比如说下单下不了或者是搜索无结果，这时会有一个染色集群，把用户定位到染色集群上面，对用户专门进行一些分析，现在还没有做到直接在用户机器上做调试，但是用户到了染色集群上面，整个调用的链路会剥离出来，比较好分析用户到底发生了什么事情。

2.测试支撑

App 的测试很重要，除了比较常规的单元功能测试，还有很重要的像稳定性跟性能，以及自动化这些都是很重要的。像手机淘宝差不多是一个月左右的时间可能会迭代一个版本。比如说新的功能开发会不会影响到老的功能，智能化测试很重要，可能分成两部分，一部分是线上所有的 API，包括业务逻辑是不是正常。另一部分是新写的代码会不会有问题，因为前面架设了统一的 API 网关，所以会在网关这个层面做很多自动化的调用回归，构造很多正常用户的数据去测试线上 API 系统的返回值，包括一些异常是不是正常，来保证线上业务逻辑的正常。在客户端这一侧，则会做很多自动脚本的回归，保障整个客户端新做的代码跟原来相比没有什么问题。另外还引入了比较多的静态代码扫描，保证不会出现低级问题。

3.运维支撑

移动 App 的运维支撑跟线上不太一样。除了常见的性能跟稳定性分析，还有针对 App 的业务监控跟舆情监控。舆情监控这个应该是移动 App 所特有的环节，大家通过市场去分发，很多用户会发评论，iOS 特别明显，有人说好，有人说不好，安卓更复杂，特别是国内有大大小小非常多的应用市场，不一而足。所以怎么对舆情做一个有效的监控，既能通过舆情监控，快速收集问题，也能做一些梳理分析，找到产品或者是性能方面的提升点。

4.发布支撑

发布支撑，其实也是在大的 App 上面才会出现的，针对一个人群的发布。传统的直接是发到市场上，大家都收到了。而手淘现在有很多内部灰度和外部灰度正式发布，可能有一些内测版本只发给阿里巴巴集团内部员工，这可以通过自己做的发布系统来支撑，有比较灵活的发布策略调整：可以圈定一批用户，也可以选定一个区域，甚至可以用后台数据做合理的设置给特定的版本推送特定升级的版本。

如果 App 发到用户手上，结果发生了致命的问题，怎么办呢？其实有两种方法修复线上的问题，第一个是直接替换 Bundle，另外就是更小维度的补丁——热补丁，现在在安卓上做的比较多。

李敏还分享了一个案例，在上半年有一次大促的时候发生了一个问题，零点就要促销了，版本可能是前天刚发布给用户的，那怎么办呢？替换 Bundle 也可以做到，但是数据下载量非常大，而且刚发布不久，这样对用户影响比较大，所以选择了用热补丁修复，主要是类似于 ClassLoader 替换，用 JAVA 开发的应该知道，主要是用类替换的方式做的。在 iOS 上也有一些方案，不过还在尝试当中。

客户端监控

可以在分钟级别确定用户调用某个操作的成功次数、失败次数和失败率，实现对业务可用性的监控。

舆情平台

舆情平台是移动 App 所独有的。要获取信息，会从用户手机淘宝自己填的反馈，利用市场和微博，实时抓取，然后把内容聚合起来，进行热门词归类，筛选出一些热门的标签话题做一些智能分类，分类之后大致知道到底是支付、详情、退款出现了什么问题，确定问题的重点之后，可以直接联系用户，甚至去跟踪用户，根据这些问题去修复线上的紧急问题或者是改善产品，这就是在线上实际使用的舆情平台。通过热门词的分类排名，就可以知道某一个版本在某一个阶段最重要的问题是什么，还扩展了用户集中反馈。

比如举办一个抢红包的活动，这个活动出现了什么问题，大量的用户重复反馈这个问题，就可以把热门的话题聚集起来。另外还可以通过舆情平台确定某个技术的改造是否成功。

舆情平台早期主要用于收集一些信息，后来发现把舆情收集起来做一些大数据分析，可以得出很多自动化的结论，甚至可以验证研发的结果是好是坏。

演讲的视频会在制作完成后发布到我们的网站上，敬请期待。

手机 QQ 的移动化实践之路

作者 臧秀涛

在 2014 年 12 月 19 日~20 日举行的 ArchSummit 北京 2014 大会上，腾讯即时通讯平台部技术总监范瑞彬做了题为《手机 QQ 的移动化实践之路》（幻灯片下载）的演讲，介绍了手机 QQ 在服务海量移动用户方面经历的一些经验。

范瑞彬（hata fan），腾讯公司即时通讯平台部技术总监，T4 专家。2004 年加入腾讯，长期负责手机 QQ 后台整体建设，完整地经历了手机 QQ 从数千人在线到亿级在线的整个过程，见证了十多年来移动互联网服务的高速发展。目前主要负责 QQ 整体的接入平台建设，在海量分布式后台架构、IM 系统设计、移动业务架构设计等方面积累了多年实践经验。

本文系根据演讲内容整理而成。

演讲主要涉及 3 个部分。第一，移动环境的特点。第二，针对移动环境的特点，如何做好接入？第三，架构设计理念方面的变化。

下图包含了云、管、端三个部分，多种终端通过多种网络访问云端的服务。上面红色部分是接入层，是首先会受到影响的。下面是逻辑存储、运营支撑和安全体系，这个系统目前终端同时在线过亿。终端每秒的请求量差不多千万，一天的请求量在数千亿的量级。

移动环境的特点

可以从三个方面看移动环境的特点。首先是移动网络，大家首先感觉是慢，而且流量又挺贵的。它还有很多种制式，差别也很大。还有终端的特点，相对 PC 来说手机终端资源（CPU、内存、电量等）是受限的，永远是不够用的。而且终端的平台很多，机型多，能力差异也非常大。还有一个很重要的特点就是移动性，可以随身携带，可以随时随地利用碎片化的时间，环境多变，使用频繁。



如何做好接入

面对这些环境的变化，接入首先会受到影响，而接入又是所有服务的基石。所以在移动环境下面，提供高质量的接入服务非常重要。如果将接入比作开车，那首先要选择一条快速的路线，这就相当于路由调度策略。另外还需要有一辆好车，车要快，类似于数据传输加速。然而选了一条好的路线，也有了一辆快车，并不代表就能快速到达目的地，尤其是非 WIFI 接入时，新增了基站、大量新增的网源系统，非常复杂。这里面有一些规则需要了解，如果不了解的话可能这些就是坑，而遇到了坑可能会翻车，所以不能把它当成黑盒，所以需要熟悉路况。路况很复杂，车也很复杂，跑的过程当中难免会遇到异常，所以还得会修车，不能抛锚，这几点是接入的几个主要工作。

1. 路由调度（选快路）

分布式接入，在南方、北方和中部选择了三个地区，各自部署了一个点，每个点也覆盖了三大运营商，这是基础工作。

这时还有一个问题，如果中小运营商用户也访问到在三大运营商部署的服务，会存在跨网访问，质量很差。所以又建设了一个内容加速机房，它有独立 IP，而且是 TCP 互联，路由直达，这样中小运营商的质量可以得到明显改善。针对海外用户，最初在香港部署了一个点，后

来又在全球各大洲，每个洲选择了一个点，海外用户就近访问加速点，通过加速点再访问国内的服务器。

部署是在不断优化的，调度也需要更精准的体现。所谓调度，无非就是说什么时候，哪些用户该连到哪些 server。所以这里可以分用户、server、时间三个维度。比如说用户，细化到每个网关；server，把用户调度到质量较高的 server 上面去；时间比较好理解，因为移动网络经常有波动，需要快速地发现波动，并及时自动干预，把它调度走。

还有一个很常见的问题，就是频繁切换网络。用户可能每天都在不同运营商，不同的网络之间来回切换。连接的时候不用域名，直接用 IP，那如何保证用户不管怎么样切换网络，都能连接到连到他应该连接的 server 上面呢？假如说用户第一次连某个网络时，他会使用本地默认列表，连到 server 时，如果 server 发现不是最优的，会及时纠正，下发一份新的 server 列表。开发团队干脆把用户最近使用的 50 个接入点统统缓存下来。

2. 数据传输加速（造快车）

做完了调度相当于选择了一条快速路线，这是不够的，还要想办法造一辆快的车。

首先是不用域名，直接用 IP。这样可以减少域名解析的开销，更重要的是，可以避免域名解析带来的各种故障，还可以减少一些被屏蔽和封掉的可能。

第二点是重用连接、预连接。比如说 QQ 里面发图的时候，其实用户还在选择图片的时候，会先把连接建立起来，而且用户传完图以后连接不会立即断掉，会维持一段时间，后面有批量图的时候可以继续使用，减少一次连接的时间可以减少几百毫秒。

第三点是精简协议和逻辑。

第四点是参数调优。比如说“拥塞窗口”（congestion window，CWND），server 的操作系统默认是 4，建议调大一点，可以调到 10，这样可以减轻慢启动对传输带来的影响。还有最大传输单元（Maximum Transmission Unit，MTU），之前跟运营商的朋友交流，他们给的建议是不要大于 1400，现在基本上都是按照这个策略来做

的。还有重新传输超时（Retransmission Timeout, RTO），一般设 3 秒左右，系统默认值设的是 1。

还有一点叫高带宽时延积环境，在这种环境下面会遇到带宽吃不满，存在浪费的问题。以前网络都是 2G 网络的功能机时代。当时传图片大部分用的都是单连接。最近几年，网络越来越多，种类也越来越多，而且网络也越来越好。所以系统也进行了改进，可以根据网络状况动态地选择合适的连接数。比如说经过理论分析以及实验验证，开发团队发现其实在 WIFI、3G、4G 比较好的网络下面传输大数据时，用双连接比单连接提升至少 10%，而且越好的网络提升效果越明显。

3. 移动网络环境不是黑盒（熟路况）

不能简单地把移动网络环境当成黑核来处理，有些细节知识是需要了解的。

第一，要了解国内移动网关的一些设置，比如说它会限制某个包传输的大小，如果超过的话直接失败。之前跟设备厂商了解，华为很多网关设置的是 10M，但是各家都不一样，也没有什么标准。还有网关很多时候对传输时间有限制，这个也是各家不一样的。了解了这些细节，肯定要很好地支持分片和断点续传，否则在某些地区可能会出问题。

第二，网关对很多标准的理解和实现，各家也是不一样的，尤其像影响比较大的，比如说对 HTTP 协议的理解和实现。比如说在 HTTP 的标准文档里面，提了一个 range，但是没有强制去做，有的厂商支持的就不太好。曾经遇到过一种情况，头部用到了 range，在分片传输图片时，运营商网关把它过滤掉了，客户端就不知道下一次该从哪里发，所以可能又从第一片开始发，这种情况下，如果客户端没有做好相关保护，就非常危险，会大量的重传，用户流量会被大量消耗。这就是需要注意的地方，尽量不要在 HTTP 头部加一些字段，需要传一些信息的话，可以放在包体里面，自己解析和理解。

第三，tcp_tw_recycle。用户说网络是好的，但是连不上 server。抓包分析发现，客户端三次握手的包已经发过来了，但是 server 没有回。这是什么问题呢？经过深入分析，研究了一下协议栈，以及一些参数设置，后来发现，假如 tcp_tw_recycle 是开启的，server 会检查对端同一个网关 IP 发过来的包，时间是不是递增的，如果不是递增的可能会

丢掉。但是移动网络环境下，这是很难保证的。关闭 `tcp_tw_recycle`，问题就解决了。后来这成了外网接入层的标准配置。

第四，端口受限。有人反映某些地区的用户连 `server` 的某些端口连不上，或者是连接质量比较差。比如曾经发现香港数码通的用户连 8080 端口的质量非常差，很慢；还发现有些机场 WIFI，比如说深圳机场 WIFI，除了 8080 和 43 之外的断口都封掉了，用其他的端口用户也连不上。开发团队意识到，不应该被动地靠用户的反馈来发现问题和解决问题，所以建设了一套自动的系统，根据海量的数据去分析，根据分析结果，`server` 在给客户端返回合适接入列表时，优先选择质量高的端口，而且也尽量注意端口搭配的多样性，这样可以提升接入质量。

最后看一下信令风暴。其实从 09 年开始，手机 QQ 这边每年会被运营商朋友拉过去一起探讨这个问题，双方本着互相理解、合作共赢的态度来对待这个问题。除了建立一些双方认可的虚拟消耗运算模型之外，双方还就移动业务达成了技术共识。比如说减少定时包，减少不必要的及时包，因为有些包真的不一定要非常及时。可以适当做一些缓存和合并。还有一点就是要有流控，万一客户端有 BUG，大量的发包，这时候有压跨移动网络的可能，并不仅仅是数据网，它对电信网也有影响，所以 `server` 这时要有能力控制客户端发包的策略和频率，这样 HTTP 服务才能让人放心一点。

4. 异常处理（会修车）

比如说业务用的是 TCP 长连接，但是请求有可能被劫持，可能会返回到奇怪的 HTML 页面，最常见的是 WIFI 的认证注册，这时需要及时准确地发现这些问题，展现出来提示用户。

有一种网络抖动叫先发后到，需要做一些保护。比如说要求客户端发包的时候一定要递增，而且即使进程被杀掉重新再起来，也要保证这次发包比上次大。借助这个大小能知道真实发包时间的先后顺序。

终端休眠，终端是为了省电，肯定有一些休眠策略，App 为了应对这些策略，有一些自动唤醒机制，做一些自己想做的事。如果用 `Wakelock`，拿到这个锁之后，系统再也进入不了休眠了，这个锁一般用两三秒就足够了，一定要慎重。

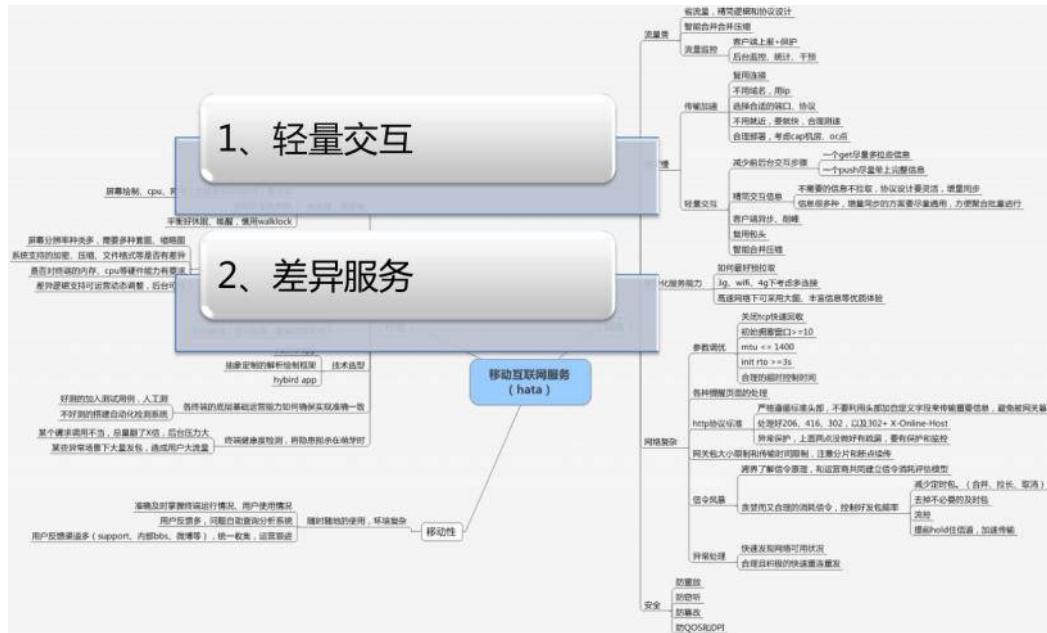
最后是 App 健康和智能检测。客户端不能保证百分之百不出问题，有时候确实有可能在一些小概率的情况下，存在异常的点，它有可能会触发一些频繁大量的发包策略。大量的发包会把用户的流量大量消耗掉，而且这还算轻的，更严重的是如果有较多用户都出现这样的问题，可能不是流量消耗的问题，而存在把移动网络压垮的可能，移动网络真的很脆弱。

《刑法》中有一条罪叫破坏通信罪，可以判三到七年，所以这种工作很危险，也是有责任的。压垮移动网络后果很严重，要想办法极早发现和干预。怎么办呢？server 可以做一些准实时的流量消耗分析，如果发现在单位时间内用户流量超过某个异常值，就及时干预，而不是事后检查。比如可以把用户直接踢下线，或者更严重一点，直接让客户端的 App 自动自杀。

除了流量的问题，还有一些问题也是很隐蔽的，比如说用户的流量看起来没有太大变化。但是可能客户端有 bug 或者是设计不当，调用了几次后台操作，这个版本发出去的话，调用可能会增加好几倍甚至更多，后台的压力就非常大，于是又是过载保护，又是紧急扩容，这个也不是开发团队希望看到的，需要有更好的办法来解决它。比如可以分析这个版本，每个用户的使用频率。如果发现这个版本与上个版本相比，使用频率变化很大，多了很多，而且又没有提前报备，这里肯定有问题，就把这个问题抓出来，这些事情靠测试同学是不现实的，他很难测试到这些问题，这些问题需要技术同学自己想办法通过做智能数据分析来发现。

架构设计的理念

根据大量的实践，开发团队提炼出两个关键词：轻量交互和差异服务。



1. 轻量交互

轻量交互，其实核心思想就是节省，主要思想是从协议层面以及逻辑层面做一些精简、合并、压缩、消峰、异步等等。

减少交互步骤。客户端一次请求，尽量把信息都拿下去，后台也尽量把相关的信息都加进去，完整地带下去，减少交互步骤。这里要求后台要多主动地做一些聚合工作，一些协议要重新设计。

精简交互信息，尽量减少每一次传输过程中的信息。有个原则，就是不用的信息尽量不要拉，这里要求开发团队在协议设计的时候要非常灵活和细致，要支持增量更新和同步逻辑。

复用包头。一般做协议设计的时候有包头和包体，包头里面放一些账号信息、身份凭证，还有版本信息。尤其随着现在安全压力越来越大，形势越来越严峻，身份凭证可能会越来越长。其实这些信息没必要每次都带，可以在接入 server 时做一些缓存，后面的包就不需要这些信息了，这样的话每个包可以减少几十个字节，甚至更可观。这样算起来收益是很大的，不光能帮用户省流量，因为传的内容少，也能加快速度，对体验也有改善，勿以善小而不为。

智能合并压缩。这里需要对业务逻辑有深入的了解。不是所有的包一定要最快响应，可以请求分一下优先级，哪些是需要及时响应的，一定要及时保证。哪些是可以降级的，不需要很及时。可以把大量的不

需要及时响应的包做一些延迟，这个延迟不仅是为了解决运营商的消耗问题，延迟以后可以做压缩。当然具体延迟多久，可能要根据具体业务的场景来看。

客户端异步削峰。还是细分问题，把一些不重要的包或者是当前这个不是立即需要的可以往后放，先让用户快点接入进来。同样，客户端也需要注意不要把 UI 绘制跟存储放在一个线程里面做，避免卡顿。

2. 差异服务

可以把差异服务理解成个性化服务，针对网络情况和终端优化应用。下面具体看看。

怎么做好预拉取。比如说 QQ 会拉消息，拉过来的一堆消息里面可能有一些是图片消息，一开始看到这些图片消息只是缩略图。那何时去下载这些缩略图的原图呢？如果说等用户点了之后再去下载，自然大家会觉得这个很慢，体验不好。那提前下载该怎么做呢？这里要细致和全面一点。比如说可能要细分网络状况，在非 WIFI 网络下面流量是很贵的。如果把原图下载了下来，用户也不看，这样流量就浪费了，这浪费的就是钱，肯定不合适。那怎么办呢？可以设计一个算法，类似于银行家算法，给每个用户分配一个配额。比如说有 500K 的配额，预拉取一张原图，比如 100K，如果用户看了原图，配额不变；如果用户后来并没有看，则把配额减去 100K，凡是预拉取了而不看的配额都减掉，配额减到零就不会再做预拉取了，避免盲目下载造成大量的流量消耗。

在 WIFI 情况下，是不是可以简单地全下载下来呢？这样也不好。虽然用户的流量在 WIFI 下基本不收费，但是腾讯后台的出口带宽很贵。尤其是图片下载消耗非常大，这个钱是很多的。怎么办呢？可以根据业务场景做细分。比如把图片分为群图和 C2C 图，就是好友之间点对点的图。分析发现，C2C 图相对是比较重要的，用户点看原图的概率非常大。同时 C2C 图占比相对来说又是非常小的，因为大部分是群图。所以 C2C 图可以预拉取。群图还是采用类似于银行家算法的方式进行管控。

预拉取思路很简单，但是要做好，可能还需要很细致地结合网络状况，结合用户状况和业务场景做细致全面的细分，这样才能在用户体验，以及服务成本和用户成本之间找到一个恰当的平衡。

信息繁简不一。很多信息应该分类归档，针对一些好终端、好网络，尽量返回一次很好的信息，反之只给它简化版。

多种套图规格。需要根据多种屏幕来抽象和简化出几种通用的图片规格，而且可以根据不同网络状况来定每种规格的压缩率。

终端是该轻还是重。到了手机时代，因为手机本身能力比较弱，而且产品还经常要求做一些跨终端的一致体验，很多东西必须放在云端统一处理才可以。大部分场景下是轻终端重后台，具体要看业务场景。比如说有些游戏，以前 PC 时代为了防止 PC 客户端作弊，很多信息都是客户端直接给 server，一律由 server 统一去算。但是在手机游戏上面，为了减少信息传递以及减少交互次数，很可能会在终端也做适当的计算，把计算结果再发给 server，在这种特殊的场景下面，在这些逻辑方面可能终端做的更重一些。所以这个问题有普遍性，也有特殊性，要具体分析、具体看，深入理解业务场景和逻辑。

能不差异的地方就不差异，考虑全面些。设计的时候要考虑全面，要把各种平台、各种网络都考虑到。在选择压缩算法、加密算法、图片格式和文件格式时，这些东西能通用的还是尽量通用，不要给自己找麻烦。如果有些地方没法统一，必须要差异，这里要抽象简化，简化为几大类，而且这个差异点尽量使后台可调可控。

终端版本信息管理，终端运营配置管理。需要把终端版本的各种信息记录下来，然后才可以根据这些信息，再加上用户的信息灵活调整配置给不同用户更好的体验。可以根据不同地区、不同网络、不同版本、不同运营商和不同号码，以及根据不同的 CPU、不同的摄像头、不同的内存，给用户下发不同的闪屏和不同的插件，这是基础能力建设。

范瑞彬最后还用一句话做了总结：“从实践中来，到实践中去。”

物联网传输协议 MQTT

作者 郭蕾

MQTT 是一个物联网传输协议，它被设计用于轻量级的发布/订阅式消息传输，旨在为低带宽和不稳定的网络环境中的物联网设备提供可靠的网络服务。MQTT 是专门针对物联网开发的轻量级传输协议。MQTT 协议针对低带宽网络，低计算能力的设备，做了特殊的优化，使得其能适应各种物联网应用场景。目前 MQTT 拥有各种平台和设备上的客户端，已经形成了初步的生态系统。在 12 月 18 日举行的 OIOT 开放物联网大会上，IBM 的徐刚带来了《IBM MQTT：国际标准化物联网推荐协议》的主题分享，本文根据其演讲内容整理而成。

MQTT 的发展历史

在物联网中，开源和开放标准是基本的要素。MQTT 的发展历史大致如下：

- 1999 年，IBM 和合作伙伴共同发明了 MQTT 协议；
- 2004 年，[MQTT.org](#)开放了论坛，供大家广泛参与；
- 2011 年，IBM 建立了 Eclipse 开源项目 Paho，并贡献了代码，Eclipse Paho 是 MQTT 的 Java 实现版本；
- 2013 年，[OASIS MQTT](#) 技术规范委员会成立；
- 2014 年，MQTT 正式成为推荐的物联网传输协议标准。

物联网接入的挑战

物联网中的数据传输会面临很多问题，比如在网络不稳定的情况下，如果保证数据的传输没有问题，如何保证数据不被重复发送，连接断开后如何进行重连。总体来说，物联网的接入会面临以下几个方面的挑战：

- 设备、传感器。物联网接入对终端采集和控制设备要求高，且终端的改造以及网络费用成本也比较高。另外，其对终端的能耗要求也比较高。
- 网络。现有的网络传输贷款参差不齐，传输网络不稳定。
- 服务器。高并发情况下，多客户端的接入能力以及消息处理能力。

MQTT 的优势

MQTT 的设计思想是开源、可靠、轻巧、简单，MQTT 的传输格式非常精小，最小的数据包只有 2 个比特，且无应用消息头。MQTT 可以保证消息的可靠性，它包括三种不同的服务质量（最多只传一次、最少被传一次、一次且只传一次），如果客户端意外掉线，可以使用“遗愿”发布一条消息，同时支持持久订阅。MQTT 在物联网以及移动应用中的优势有：

- 可靠传输。MQTT 可以保证消息可靠安全的传输，并可以与企业应用简易集成；
- 消息推送。支持消息实时通知、丰富的推送内容、灵活的 Pub-Sub 以及消息存储和过滤；
- 低带宽、低耗能、低成本。占用移动应用程序带宽小，并且带宽利用率高，耗电量较少。

信息安全看未来物联网发展

作者 郭蕾

随着云计算、智能硬件的日益普及，物联网安全受到越来越多人的关注。物联网面临的安全威胁涉及方方面面，厂商需要在打造优质用户体验的同时，更多的关注产品安全，保证用户的信息安全。在 12 月 18 日的 OIOT 开放物联网大会上，来自 KEEN 的吕一平分享了《从信息安全看未来物联网发展》的主题演讲，本文内容根据其演讲内容整理而成。

吕一平是暮震公司 COO，从事多年信息安全技术和管理工作，目前重点关注智能硬件安全、物联网安全、车联网安全等新兴信息安全领域。他曾在微软工作时创建微软美国总部以外的第一个地区安全响应中心。

黑客是中性词

黑客是音译过来的一个词，但它是一个中性词，本身并不带有任何感情色彩，但是很多的电影电视里，都把黑客拍为偷取银行卡账号获取密码的人。其实黑客有两类人，正义的一方叫做白帽子，邪恶的一方叫做黑帽子。经常听到新闻报道里面做黑产的是黑帽子。白帽子同样有非常强的信息安全技术和能力、经验，他们是帮助厂商发现问题，尽快解决问题，最终的目标是为了保护用户和厂商的安全。

智能终端面临的威胁

从安全角度来看，可以把智能终端的安全问题分为六个层次，包括云数据存储、互联网连接、近场和物理连接、移动应用、移动操作系统、移动设备芯片。云数据存储方面的安全问题现在涉及的比较多，比如前段时间的 iCloud 泄密事件，黑客利用工具可以绕过苹果的 Find My iPhone 的安全系统，最后导致很多好莱坞女星的私密照片泄漏。互联网连接我们经常接触的可能是公共 WIFI，黑客可以利用交换机或者路由器截取用户的通话记录、短信以及网站浏览记录。近场和物理

连接主要是指 NFC 和 USB 等技术引起的安全问题。移动应用方面，未来物联网的设备也会支持移动应用，而移动应用的安全问题更不容忽视，很多应用本身就包含恶意代码，偷窃用户流量、窃取用户隐私。操作系统级别的安全涉及方方面面，不管是电脑、智能手机还是物联网硬件，都需要操作系统作为基础支撑。芯片的安全问题更为严重，如果芯片存在安全问题，那会影响到所有用到该芯片的机型，由于芯片需要和操作系统连接，而连接又需要硬件驱动支持，硬件驱动又运行在系统权限下，所有如果驱动本身存在安全问题，后果将不可设想。

微软的前车之鉴：从做产品，到做安全的产品



微软在 2001 年的时候发布了 Windows XP，当时的产品定位是用户体验好，并没有考虑安全，紧接着在 2003 年、2004 年发生了两次大规模的互联网事件。安全问题发生后，微软痛定思痛，从 2004 年开始往安全方面投入了很多精力，引入了专业的信息安全专家以及安全工程方法来保证系统的安全。同时，微软也创造了安全开发周期的概念，并且优化了安全响应流程。目前苹果使用的安全技术都是微软原创的。

如何应对未来物联网的安全问题

物联网设备与用户的人身安全息息相关，安全问题更应该重视，厂商需要发现安全问题后及时修复。首先安全应该是产品质量的重要部分，物联网厂商一定要把好安全质量关。其次，安全行业要为科技发展保驾护航，安全行业应该成为科技发展的推动力，而不是阻力，厂商不应该觉得害怕有安全问题而不使用新技术，安全是给人类的发展

保驾护航的，而不是阻止科技发展的。第三，动态的改进和视而不见，安全的问题厂商应该第一时间修复，这也是对用户负责，安全是一个动态改进的过程，安全问题是百分之百解决不完的问题，所以动态发现安全问题，动态的解决，这是所有厂商都应该有的态度。

赵海峰：大数据决定互 联网金融未来

作者 郭蕾

互联网金融持续火热，阿狸巴巴、百度、京东等巨头均进入金融行业，这些创新的金融产品在给消费者们带来惊喜的同时，也在推动着传统金融行业的变革。而互联网金融市场的竞争也日趋白热化，从一开始互联网金融就带有互联网的属性，也就是说技术将会直接影响产品以及商业模式。目前各大金融公司都在探索如何将大数据、云计算等技术与已有业务结合，期望可以在保证平台安全、稳定的同时可以提供更加精准的服务。InfoQ 此次专访了京东金融的架构师赵海峰，另外作为 ArchSummit 北京 2014 大会《互联网金融》专题的讲师，赵海峰将会站在电商平台的角度上剖析互联网金融系统架构实践。

InfoQ：据了解，从 2009 年开始您就一直负责京东交易系统的架构和研发工作，能简单介绍下京东的交易系统么？架构方面主要经历了哪些演进？

赵海峰：京东交易系统主要经历了 5 个发展阶段。但限于篇幅，我在这里只做简单概述。有兴趣可以参考京东出版的技术类书籍《京东技术解密》，我在第十五章详细的描述了后三个阶段的演化历史。

第一阶段：集中式系统，这一阶段的特点是业务简单，技术简单，以集中式系统为主。但是由于简单、集中式的系统，难以支撑订单量的成倍增长，以及日益复杂的业务需求，所以以 08 年的封闭开发为契机，京东开始技术转型。

第二阶段： worker 时代，这一阶段的特点是 worker 简单、灵活，开发快速，适应了当时业务的快速发展；但 woker 不宜维护和统一管理，可移植性差。worker 时代顶住了每年 618 订单

的成倍增长，快速适应甚至推动了业务发展（例如单品类向多品类跨越、全国发货模型的实施），10年图书促销事件，暴露了当时京东技术上的不足，但也成为京东技术升级的契机。

第三阶段：SOA 时代开始，特点是简单服务化，可移植性强，业务解耦，各系统专注于自己的功能优化和性能优化；SOA 的深度和广度不够彻底，当时的数据一致性（例如 SQLServer 复制延迟），数据库写入单点问题等是京东的几大技术顽疾。从 09 年开始，五大业务中间件逐一建立，开启京东 SOA 的先河。11 年的封闭开发，完成了交易系统的架构升级：.net 转型 Java；交易系统服务化、多点写入和跨机房容灾，等等。

第四阶段：SOA 深化，这个阶段的特点是引入并扩展开源框架；对海量的服务进行治理和流程编排；通过事件触发机制，驱动业务流程，降低数据延迟；提升系统性能等等。211 限时达项目，重构了订单生产流程，将整个生产过程从 30 分钟缩短到 10 秒以内，根除了数据复制延迟问题；SOP 合单项目，改变了交易系统、生产系统、财务系统等的交互方式，将服务事件化、流程化，可管理、可编排，为后来更复杂的订单业务提供了强大的技术支撑。

第五阶段：京东云时代，这个阶段的特点是云计算、虚拟化、大数据。云盘、JFS、UMP、统一日志等等优秀产品应运而生，并在各大系统普及；不但为业务系统提供了更强大的业务支撑，而且为技术系统的量化运营提供了依据。

InfoQ：为什么选择加入京东金融？京东金融目前是怎么样的一个规模？

赵海峰：我之所以离开京东交易平台部，加入京东金融有以下几个原因：

2013 年之后，京东交易系统的基础架构已经日益成熟，足以支撑未来几年的交易量和新业务的爆炸式增长；从技术上来说，已经没什么大的挑战。一个例证就是：从 08 年起，每到 618 和双十一这两个购物狂欢节的当天，研发作战指挥部就灯火通明、场面异常火爆；而到 13 年后，这样的火爆场面已经不复存在，只有零零星星的几个人在监控系统。从以前紧张火爆的

现场指挥，到后来的悠然观测，确实非常自豪和满足；但也有几分落寂。

我非常看好互联网金融，金融在人们经济生活中处于核心的地位。一旦金融被互联网化，整个世界将为之改变。然而大的机遇必然面临大的挑战，而这正是我所期待的！目前京东金融处于蓬勃发展期，拥有基金、保险、理财、贷款、支付、众筹等各种互联网金融服务以及京东白条、京东小金库、京小贷等多个金融产品，并在不断拓展中。就像 09 年的京东电商，今天的京东金融或许还很弱小，但明天的京东金融将无比强大！

InfoQ：京东金融这块的业务更为复杂吧，能谈谈京东金融是如何保证系统的可靠性的么？

赵海峰：京东金融和京东电商的复杂维度是不同的（电商业务牵扯更广，包括信息流、资金流、物流；以及交易、支付、生产、仓储、配送、财务等各个环节的生态圈；而金融业务中资本运作的深度，是电商业务无法比拟的）。

在保证系统的可靠性上，京东金融借鉴了很多电商系统的经验：前端的 CDN、缓存、负载均衡，到应用层的分布式部署、容灾；再到数据层的读写分离、多点写入、分库分表等等。SOA 理论中的服务多层级划分、流程编排等，在金融系统中更有用武之地。将一些基础功能抽象成基础服务，进而构建出核心服务和复杂应用，就像搭积木一样。这样在迅速扩展新业务的同时，仍能保证系统的稳定性。

InfoQ：大数据决定互联网金融未来，京东金融是怎么用好现有的这些大数据的？

赵海峰：目前京东金融在各个产品线上，大量采用大数据的相关技术。例如：京东白条的营销推广、额度授信；刚刚上线的京小贷的贷前、贷中、贷后管理；金融平台的风险报警系统等等。通过数据的采集、分析、决策、实施，再到新数据的采集，如此循环往复，来分析用户行为、资金风险、信用评级等，达到最优的用户体验、最低的风险和成本，以及资本配置的最优化——而这正是金融的最终目标。

InfoQ：如何在快速迭代开发新功能时，保证系统的准确性和可靠性？

赵海峰：SOA 理论中的服务编排思想，更适合于金融系统。从复杂的业务中，抽象出基础服务组件。这些基础组件通用并且经历很多考验，非常稳定可靠，可以直接使用，不需重新开发，这样不仅缩短了开发周期，而且降低了系统风险。通过流程引擎可对这些基础组件进行流程编排，组装成更复杂的应用，就像是搭积木一样，从而达到快速迭代开发的目的，又能保障系统的准确性和可靠性。同时还可以在系统中加一个独立的角色：质量检查员。质量检查员对每个关键应用进行业务检查，一旦出现问题，系统自动做出处理。这个系统是独立的，从而避免很多金融系统风险。

为何 Asana 开始用 TypeScript

作者 谢丽

团队任务管理平台提供商 [Asana](#) 的一个代码库里有上万行的 JavaScript 代码。随着时间推移，其中的代码越来越多，维护也越来越难。他们需要一种不同的语言。最终，他们出于以下几个方面的考虑选择了 TypeScript。

首先，TypeScript 可以产生简洁的 JavaScript 代码，更容易与现有的代码集成。它与 JavaScript 有非常紧密的映射关系，熟悉 JavaScript 的开发人员很容易学习使用，而且对于生成的 JavaScript 代码，更容易推断出其性能。

其次，有强大的工具支持，并且能够轻松地与 JavaScript 庞大而活跃的开源社区融合，如 IntelliJ/WebStorm 对 TypeScript 提供了强大的支持，而 [DefinitelyTyped](#) 是一个生机勃勃的社区，为许多开源库提供了 TypeScript 定义。TypeScript 还通过积极维护的 gulp/grunt 提供了很棒的构建工具。

再者，TypeScript 支持强类型，使编译器和 IDE 能够在编译时而不是运行时发现错误，IntelliJ 对其也有很好地支持。支持重构以及更好的代码导航功能有助于编写出质量更高的代码；上述两点都有助于提高代码编写效率。

最后，TypeScript 支持静态类型。这有两个方面的好处：一是可以确保客户端和服务器端采用同样的协议；二是允许跳过运行时检查，如 [React.js 可以从中获得显著的性能提升](#)。

另外，将浏览器代码迁移到 TypeScript 只是 Asana 正在进行的几项重大改进中的一项，感兴趣的读者可以关注其[官方博客](#)。

为什么不要把 ZooKeeper 用于服务发现

作者 谢丽

ZooKeeper 是 Apache 基金会下的一个开源的、高可用的分布式应用协调服务。许多公司都把它用于服务发现。但在云环境中，面对设备及网络故障时的恢复能力是需要重点考虑的问题。因此，将应用部署在云上，就必须要预见到硬件故障、网络延迟以及网络分区等问题，进而构建出恢复能力强的系统。Peter Kelley 是个性化教育初创公司 Knewton 的一名软件工程师。他认为，从根本上讲，把 ZooKeeper 用于服务发现是个错误的做法，理由如下。

在 ZooKeeper 中，网络分区中的客户端节点无法到达 Quorum 时，就会与 ZooKeeper 失去联系，从而也就无法使用其服务发现机制。因此，在用于服务发现时，ZooKeeper 无法很好地处理网络分区问题。作为一个协调服务，这没问题。但对于服务发现来说，信息中可能包含错误要好于没有信息。虽然可以通过客户端缓存和其它技术弥补这种缺陷，像 [Pinterest](#) 和 [Airbnb](#) 等公司所做的那样，但这并不能从根本上解决问题，如果 Quorum 完全不可用，或者集群分区和客户端都恰好连接到了不属于这个 Quorum 但仍然健康的节点，那么客户端状态仍将丢失。

更重要地，上述做法的本质是试图用缓存提高一个一致性系统的可用性，即在一个 CP 系统之上构建 AP 系统，这根本就是错误的方法。服务发现系统从设计之初就应该针对可用性而设计。

抛开 CAP 理论不说，ZooKeeper 的设置和维护非常困难，以致 Knewton 多次因为错误的使用出现问题。一些看似很简单的事情，实际操作起来也非常容易出错，如在客户端重建 Watcher，处理 Session 和异常。另外，ZooKeeper 本身确实也存在一些问题，如 [ZOOKEEPER-1159](#)、[ZOOKEEPER-1576](#)。

由于这些问题的存在，他们切换到了 Eureka。这是一个由 Netflix 开发的、开源的服务发现解决方案，具有可用性高、恢复能力强的特点。相比之下，它有如下优点。

如果一个服务器出现问题，Eureka 不需要任何类型的选举，客户端会自动切换并连接到一个新的 Eureka 服务器。当它恢复时，可以自动加入 Eureka 节点集群。而且，按照设计，它可以在零停机的情况下处理更广泛的网络分区问题。在出现网络分区的情况下，Eureka 将继续接受新的注册并发布。这可以确保新增服务仍然可以供分区同侧的任意客户端使用。

Eureka 有一个服务心跳的概念，可以阻止过期数据：如果一个服务长时间没有发送心跳，那么 Eureka 将从服务注册中将其删除。但在出现网络分区、Eureka 在短时间内丢失过多客户端时，它会停用这一机制，进入“自我保护模式”。网络恢复后，它又会自动退出该模式。这样，虽然它保留的数据中可能存在错误，却不会丢失任何有效数据。

Eureka 在客户端会有缓存。即使所有 Eureka 服务器不可用，服务注册信息也不会丢失。缓存在这里是恰当的，因为它只在所有的 Eureka 服务器都没响应的情况下才会用到。

Eureka 就是为服务发现而构建的。它提供了一个客户端库，该库提供了服务心跳、服务健康检查、自动发布及缓存刷新等功能。使用 ZooKeeper，这些功能都需要自己实现。

管理简单，很容易添加和删除节点。它还提供了一个清晰简洁的网页，上面列出了所有的服务及其健康状况。

Eureka 还提供了 REST API，使用户可以将其集成到其它可能的用途和查询机制。

总之，云平台并不总是可靠，服务发现需要具备尽可能高的可用性和恢复能力，而 Eureka 恰恰是针对这种情况而设计的。

Java 8 新特性：全新的 Stream API

作者 廖雪峰

Java 8 引入了全新的 Stream API。这里的 Stream 和 I/O 流不同，它更像具有 Iterable 的集合类，但行为和集合类又有所不同。

Stream API 引入的目的在于弥补 Java 函数式编程的缺陷。对于很多支持函数式编程的语言，map()、reduce()基本上都内置到语言的标准库中了，不过，Java 8 的 Stream API 总体来讲仍然是非常完善和强大，足以用很少的代码完成许多复杂的功能。

创建一个 Stream 有很多方法，最简单的方法是把一个 Collection 变成 Stream。我们来看最基本的几个操作：

```
public static void main(String[] args) {  
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4,  
    5, 6, 7, 8, 9, 10);  
    Stream<Integer> stream = numbers.stream();  
    stream.filter((x) -> {  
        return x % 2 == 0;  
    }).map((x) -> {  
        return x * x;  
    }).forEach(System.out::println);  
}
```

集合类新增的 stream()方法用于把一个集合变成 Stream，然后，通过 filter()、map()等实现 Stream 的变换。Stream 还有一个 forEach()来完成每个元素的迭代。

为什么不在集合类实现这些操作，而是定义了全新的 Stream API？Oracle 官方给出了几个重要原因：

一是集合类持有的所有元素都是存储在内存中的，非常巨大的集合类会占用大量的内存，而 Stream 的元素却是在访问的时候才被计算出

来，这种“延迟计算”的特性有点类似 Clojure 的 lazy-seq，占用内存很少。

二是集合类的迭代逻辑是调用者负责，通常是 for 循环，而 Stream 的迭代是隐含在对 Stream 的各种操作中，例如 map()。

要理解“延迟计算”，不妨创建一个无穷大小的 Stream。

如果要表示自然数集合，显然用集合类是不可能实现的，因为自然数有无穷多个。但是 Stream 可以做到。

自然数集合的规则非常简单，每个元素都是前一个元素的值+1，因此，自然数发生器用代码实现如下：

```
class NaturalSupplier implements Supplier<Long> {  
  
    long value = 0;  
  
    public Long get() {  
        this.value = this.value + 1;  
        return this.value;  
    }  
}
```

反复调用 get()，将得到一个无穷数列，利用这个 Supplier，可以创建一个无穷的 Stream：

```
public static void main(String[] args) {  
    Stream<Long> natural = Stream.generate(new  
NaturalSupplier());  
    natural.map((x) -> {  
        return x * x;  
    }).limit(10).forEach(System.out::println);  
}
```

对这个 Stream 做任何 map()、filter()等操作都是完全可以的，这说明 Stream API 对 Stream 进行转换并生成一个新的 Stream 并非实时计算，而是做了延迟计算。

当然，对这个无穷的 Stream 不能直接调用 forEach()，这样会无限打印下去。但是我们可以利用 limit()变换，把这个无穷 Stream 变换为有限的 Stream。

利用 Stream API，可以设计更加简单的数据接口。例如，生成斐波那契数列，完全可以用一个无穷流表示（受限 Java 的 long 型大小，可以改为 BigInteger）：

```
class FibonacciSupplier implements Supplier<Long> {  
  
    long a = 0;  
    long b = 1;  
  
    @Override  
    public Long get() {  
        long x = a + b;  
        a = b;  
        b = x;  
        return a;  
    }  
}  
  
public class FibonacciStream {  
  
    public static void main(String[] args) {  
        Stream<Long> fibonacci = Stream.generate(new  
FibonacciSupplier());  
  
        fibonacci.limit(10).forEach(System.out::println);  
    }  
}
```

如果想取得数列的前 10 项，用 limit(10)，如果想取得数列的第 20~30 项，用：

```
List<Long> list =  
fibonacci.skip(20).limit(10).collect(Collectors.toList());
```

最后通过 collect()方法把 Stream 变为 List。该 List 存储的所有元素就已经是计算出的确定的元素了。

用 Stream 表示 Fibonacci 数列，其接口比任何其他接口定义都要来得简单灵活并且高效。

计算 π 可以利用 π 的展开式：

```
/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - ...
```

把 π 表示为一个无穷 Stream 如下：

```

class PiSupplier implements Supplier<Double> {

    double sum = 0.0;
    double current = 1.0;
    boolean sign = true;

    @Override
    public Double get() {
        sum += (sign ? 4 : -4) / this.current;
        this.current = this.current + 2.0;
        this.sign = ! this.sign;
        return sum;
    }
}

Stream<Double> piStream = Stream.generate(new
PiSupplier());
piStream.skip(100).limit(10)
    .forEach(System.out::println);

```

这个级数从 100 项开始可以把 的值精确到 3.13~3.15 之间:

```

3.1514934010709914
3.1317889675734545
3.1513011626954057
3.131977491197821
3.1511162471786824
3.1321589012071183
3.150938243930123
3.132333592767332
3.1507667724908344
3.1325019323081857

```

利用欧拉变换对级数进行加速，可以利用下面的公式:

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

用代码实现就是把一个流变成另一个流:

```

class EulerTransform implements Function<Double,
Double> {

    double n1 = 0.0;
    double n2 = 0.0;
}

```

```
double n3 = 0.0;

@Override
public Double apply(Double t) {
    n1 = n2;
    n2 = n3;
    n3 = t;
    if (n1 == 0.0) {
        return 0.0;
    }
    return calc();
}

double calc() {
    double d = n3 - n2;
    return n3 - d * d / (n1 - 2 * n2 + n3);
}
}

Stream<Double> piStream2 = Stream.generate(new
PiSupplier());
piStream2.map(new EulerTransform())
.limit(10)
.forEach(System.out::println);
```

可以在 10 项之内把 π 的值计算到 3.141~3.142 之间：

```
0.0
0.0
3.16666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
```

还可以多次应用这个加速器：

```
Stream<Double> piStream3 = Stream.generate(new
PiSupplier());
piStream3.map(new EulerTransform())
.map(new EulerTransform())
.map(new EulerTransform())
.map(new EulerTransform())
.limit(20)
.forEach(System.out::println);
```

20 项之内可以计算出极其精确的值：

```
...
3.14159265359053
3.1415926535894667
3.141592653589949
3.141592653589719
```

可见用 Stream API 可以写出多么简洁的代码，用其他的模型也可以写出来，但是代码会非常复杂。

作者简介

廖雪峰，十年软件开发经验，业余产品经理，精通 Java / Python /Ruby/Visual Basic/Objective C/Lisp 等编程语言，对开源框架有深入研究，著有《Spring 2.0 核心技术与最佳实践》一书，多个业余开源项目托管在 GitHub。

褚霸：不要为了开源而开源

作者 郭蕾

RDS（Relational Database Service）是一种即开即用、稳定可靠、可弹性伸缩的在线数据库服务，具有多重安全防护措施和完善的性能监控体系，并提供专业的数据库备份、恢复及优化方案，使用户能专注于应用开发和业务发展。褚霸在 OSC 源创会年终盛典上分享了阿里巴巴如何使用开源软件构建 RDS 关系型数据库服务的实践经验，会后，InfoQ 专门采访了褚霸，与他共同探讨了阿里云 RDS 背后的技术挑战并听他讲述了他与开源的故事。

InfoQ：在分布式系统中，如何保证数据的一致性是很多公司面临的挑战。阿里云 **RDS** 作为云数据库解决方案，是如何保证数据的一致性的？

褚霸：我们也是使用的 MySQL 的开源版本，所以在数据一致性方面和其它公司遇到的问题实际上是一样的。我们遇到的问题，别人也会遇到。遇到问题不可怕，可怕的是没有那颗解决问题的心和解决问题的能力。首先，我们有非常强大的开源团队，他们对 MySQL 非常熟悉，有能力根据需要定制符合我们需求的数据库，这可能是最重要的，阿里巴巴在开源和基础设施方面投入大量的人力和财力，我认为这样的投入效果还是非常明显的。

其次，当我们的可用性和数据安全性有冲突的时候，我们一定会牺牲可用性去保证数据安全，这是我们的理念。我们必须有这个理念，去保障数据的安全。在这个理念基础上，我们会去做各种优化。从最基础的开始，我们需要能够检测出主备不一致，所以我们做了一整套的检测机制，比如通过主备打快照的方式保证不影响用户正常使用。该机制来保证能够及时发现数据的不一致。接下来就需要解决不一致的问题，目前业界的解决方案也有很多，比如用主库覆盖掉备库，或者自动重搭备库一遍。这些方案虽然都能解决问题，我们也有用过，但是都特别慢，我们需要在一致性和性能间做一个平衡。现在我们会单

独把不一致性的数据挑出来，专门通过增量的方式去解决不一致的这些数据。

InfoQ：阿里在开源技术的研究和贡献是非常大的，能否分享一下阿里在数据库这一块，使用和参与到开源社区的情况？

褚霸：我们的产品中大量使用到了开源软件，比如我们的数据通道就是用 Erlang 系统做的，在 Erlang 使用过程中我们踩过很多坑，当然也都把解决方案回馈给了社区。我们用 Erlang 做过很多的模块，这些模块都和底层业务监控有关，和业务无关，属于基础设施，今天我们把这些代码开源，那社区就可以在我们的基础上使用，我们踩过的坑，不要让其它人再踩了，未来我们也将开源更多的基础模块来回馈社区。

除了代码之外，我们更多的会向社区分享我们的开源软件使用经验，因为在现在的开源世界里，大部分的代码贡献者都是草根，他们的代码质量不一定符合工业标准。有些代码在普通环境跑跑没有问题，但是在商业环境或者有严格性能要求的场景下，它不一定能用。所以我经常和团队的同学们讲，在使用开源软件前，一定要把它研究明白，把它当成一个白盒子使用。如果是黑盒子我们绝对不会用，一定要把软件研究明白，就像这个软件的代码是你写的，想改哪里就该改哪里，只有做到这一点，才能在产品中使用。因为一旦用到产品里面，实际上是没有回头路的，你必须往前走，你不能回撤。所以，我们的同学都锻炼出来了。我们有经验后，肯定需要把相关的经验、踩过的坑分享给同行者，分享给社区。比如我们每个月都会发布 MySQL 内核月报，报告最新的数据库趋势和我们线上的 BUG 如何被发现和修正的。

InfoQ：您刚才也提到了 Erlang，阿里云在这门语言的使用上应该有丰富的经验，您能解释下为什么阿里云会选择这门语言吗？

褚霸：Erlang 是爱立信开源的一门语言动态类型编程语言，最早是设计用来做交换机的。交换机有几个特点，第一个是软实时系统，打电话肯定不能有延迟。第二个是兼容性。在做交换机的时候并不知道要接哪个设备，更不知道设备对应的操作系统，所以在设计之初它就有一个好的兼容特性。第三个是稳

定，从一开始 Erlang 就使用完全不同于其它语言的设计理念，在语言中又集成了一个完整的操作系统，它有自己的进程，所有的进程和我们现在讲的操作系统的进程一模一样，只不过 Erlang 的粒度更小。

Erlang 的操作系统内部有自己的调度系统，它的调度系统比操作系统的还复杂。比如说有大量的内存管理，并且都非常精细。Erlang 非常重要的一个特性就是可以做到调度公平、资源隔离，这个非常符合云计算的特点。因为云计算就是这样的，大家的资源一起分，这个理念和 Erlang 是一致的，所以用 Erlang 去做这件事就会非常的自然。还有就是 Erlang 支持热升级，它可以做到在线热升级。发现问题的时候，通过一个热升级包，就可以把最新的代码部署上去，整个进程不需要停止，用户不受影响，我们非常看中的 Erlang 的这个能力。

InfoQ：最近亚马逊推出了全新的数据库引擎 Aurora，号称比 MySQL 快 5 倍，您怎么看这件事？

褚霸：软件是绕不开硬件的，MySQL 出现的那个年代硬件条件比较差，当时服务器的内存和硬盘容量都比较小，而现在我们线上的服务器内存都比那个时候的硬盘容量还大好多倍，Aurora 就是在这样的背景下推出的。

现在 SSD 很廉价，整个圈子对数据的安全性以及数据库的性能要求也更高了，Aurora 的设计理念其实和 MySQL 的设计理念是不一样的，它是基于现有的技术而设计的，时代在变，未来肯定也是这样的一个趋势（实际上刚刚发布不久的 MySQL 5.7.5 相比于 5.6 也有了大幅提升），所以我觉得这并不惊讶，阿里云也在向这方面努力，我们也在往前走，但是最重要的还是踏踏实实静下心来做。

InfoQ：您怎么看国内的开源趋势？您对开源社区有什么样的建议？

褚霸：我很早就开始玩开源，也是看到最近几年开源在国内的飞速发展。1999 年的时候，在整个社区中，只有你会写一个 TCP 服务器，再开源出来，那就很牛了。如何再能做下优化，能够处理超过 1024 个连接，那就更牛了，这是那时候的水

平，那个年代学的基本都是国外的科研技术。而今天，我们能够处理的连接数可能已经翻了 100 倍。我是在中国社区长大的，一路走来也看到国内社区也在茁壮成长，特别是最近几年，中国人贡献的项目越来越多，国产开源项目也越来越多，这个从开源中国发布的数据可以看到。

开源是大的趋势，在某些方面开源软件已经主导创新，开源无处不在。我相信这个观念深入人心，趋势一定是不可逆的。我相信，开源会走得越来越快，而且今天任何一家公司，你不可能绕过开源技术而闭门造车。

国内有很多开源项目，但是项目的质量并不高。我们去招聘的时候，很多人会把自己 GitHub 地址帖上去。我们上去一看，这个项目是两三年前做的，已经不再维护，并且还只是个简单的练习项目。所以说，做开源的初衷很重要，你是为了什么去做这个项目？为了面试？为了吹牛？个人认为开源最大的一个驱动力是解决真实的问题，而不是为了开源而开源。开源项目需要沉淀，需要静下心来持续跟进，把项目做深、做精。

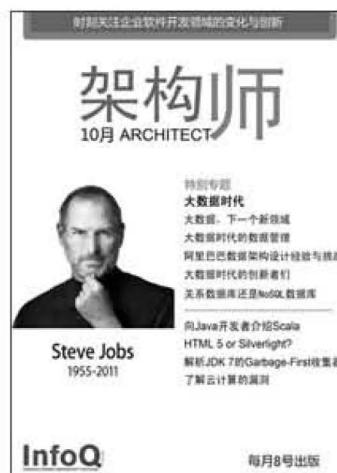
采访嘉宾

余锋（花名褚霸）是阿里巴巴核心系统技术专家，有超过 15 年的网络和底层系统开发经验，专注于高性能分布式服务器的研究和实现，擅长构建大规模集群存储服务器。2012 年 7 月 23 日，阿里巴巴宣布推淘宝等七大事业群，被马云称作“七剑”，组成集团 CBBS 大市场，余锋是在其中负责 RDS 数据库的资深专家。

架构师

www.infoq.com/cn/architect

每月8号出版



水仙——一月的花语

水仙花是点缀元旦和春节的冬季时令花。水仙具有宜人的芳香，是水仙雕刻艺术的主要材料，它的花还能制造高级的芳香油。水仙在水中养殖，因此有一个非常雅致的名字“凌波仙子”。

最常见的品种是单瓣水仙，花瓣为双色花瓣，内圈由一黄色杯状花瓣所构成，外围则有六片白色花瓣，雌蕊一个，雄蕊则有六个，三个较长、三个较短。水仙的花自叶丛中生出，花苞着于茎顶，花苞外面有一层很薄的苞膜，水养一定时间后苞膜自然破裂，花瓣逐渐展平，花朵开放。一个苞膜内一般有4~8个花朵，最多可达15个。水仙通常在农历年间开放，预示着来年的好运气。

文人墨客在书房摆放一盆水仙花，芳馨盈室，能够营造出文雅、舒适、清新的气氛。



促进软件开发领域知识与创新的传播

架构师

ARCHITECT

**人物 | People**

褚霸：不要为了开源而开源

解读2014 | Review

解读2014之云计算篇

解读2014之iOS篇

解读2014之Docker篇

专题 | Topic

手机淘宝构架演化实践

从信息安全看未来物联网发展

赵海峰：大数据决定互联网金融未来

InfoQ

2015年01月

架构师 2015 年 1 月刊

每月 8 号出版

本期主编：郭蕾

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.com新浪微博：<http://weibo.com/infoqchina>商务合作：sales@cn.infoq.com**本期主编 郭蕾**

InfoQ 技术编辑，文艺范儿程序员。在 CRM 行业厮混 3 年多，喜欢技术写作和社区运营，信奉见城彻先生的那句话：偏执、冒险、狂妄的人终是英雄。我不是英雄，但我会努力成为英雄。