促进软件开发领域知识与创新的传播

架构师 ARCHITECT

理论派 | Theory

推荐系统和搜索引擎的关系

观点 | Opinion

关于云迁移的经验总结

推荐文章 | Article

鏖战双十一-阿里直播平台面临的技术挑战

从无到有：微信后台系统的演进之路

热点 | Hot

OpenJDK将对Android开发产生怎样的影响

Python将迁移到GitHub

InfoQ Queue

2016年2月

OpenJDK将对Android开发产生怎样的影响

Google已决定将从下一版本的Android开始采用OpenJDK。

Python将迁移到GitHub

IPython目前的维护者，Brett Cannon，日前在Python的核心工作流邮件列表中宣布了Python将迁移到Github中

从无到有：微信后台系统的演进之路

微信从无到有，大家可能会好奇这期间微信后台做的最重要的事情是什么？

鏖战双十一-阿里直播平台面临的技术挑战

本文将针对一些技术细节，抽丝剥茧，希望通过些许的分析和介绍，能让大家有所启发。

推荐系统和搜索引擎的关系

作者结合自己的实践经验来为大家阐述两者之间的关系、分享自己的体会。

关于云迁移的经验总结

本文是作者与不同类型的技术公司就云迁移这个话题交流后的总结。

架构师 2016年2月刊

本期主编　姚梦龙

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

QCon 全球软件开发大会2016

International Software Development Conference

旧金山 上海 伦敦 北京 圣保罗 东京 纽约 里约热内卢

2016年04月21-23日 北京·国际会议中心 www.qconbeijing.com

8折优惠

截止至02月21日，最后一周

至少节省1360/张（票量有限）

团购可享更多优惠

精彩早知道

■ 一线专家、技术负责人担任专题出品人（部分）

余弦

知道创宇技术副总裁

贺岩（尤达）

蚂蚁金服资深专家

沈剑

58到家技术总监

吴凯华

腾讯社交网络质量部

副总经理

王海龙

58赶集技术工程平台部

高级总监

肖康

奇虎360系统部总监

■ 部分演讲嘉宾

郭斯杰

Twitter Staff Software Engineer

《Twitter Messaging的架构演化之路》

夏仲璞

百度工程效率部高级经理

《支持百度万人开发的工具装备及百度工程能力建设》

精彩内容策划中，欢迎自荐/推荐讲师，

线索提供：speakers@cn.infoq.com

更多精彩专题内容，

敬请关注：www.qconbeijing.com

抢票热线：010-64738142

咨询邮箱：qcon@cn.infoq.com

主办方 Geekbang. 极客邦科技 InfoQ

ArchSummit
全 球 架 构 师 峰 会

ArchSummit深圳2016启动！

实践第一 案例为主

15大热门技术专题

- 研发体系构建管理
- 移动应用架构
- 分享经济下的架构
- 互联网金融
- 大数据与个性化
- 海量服务架构探索
- 高性能高效的运维体系构建
- 云上的技术转变
- 新产业，新技术
- 游戏
- 技术创业
- 电商大促背后的技术较量
- IoT，让世界更加智能
- 互联网安全
- 社交网络

扫描二维码 进入大会官网

2016年7月15-16日

中国·深圳 南山区
华侨城洲际酒店

sz2016.archsummit.com

崔康 InfoQ中国总编辑，负责InfoQ整体内容的品牌和运营，担任QCon、ArchSummit大会的总策划。

卷首语

又到了年根，技术人们都在忙着做总结、领奖金、刷车票、挤火车……

“忙”已经成了现代人的基本属性，环顾四周，大家都在埋头苦干。朋友们之间互相抱怨着，最近忙的不可开交，就差不吃不喝不睡觉了。我也整天忙，不过更多时候和其他人一样，是瞎忙，到底应该忙些什么，哪些事情优先级是比较高的？

说到这里，很多人会想到那个关于重要和紧急的象限图，紧急的事情自然是需要优先处理的，这个就不多谈了。

还有什么事情是优先级高的？

关于人的事情，比如招聘、团队建设、晋升。这些事情表面上无法产出业务价值，但却是最重要的事情，事在人为，所有的业绩都是由团队成员产生的，联想的管理三要素“建班子、定战略、带队伍”，其中两个是有关于人才建设的。

有个小故事，一个朋友正处于创业阶段，公司发展迅速，也有百十来人的规模，一些关键岗位还缺人，整天和我抱怨事情太多，需要人来分担。我给他推荐了几个靠谱的候选人过去，复试结束的时候，我建议这个朋友亲自来参与终面，结果他摆摆手，不行太忙了，没空！当时我就无语了，一边说自己很忙缺人，一边又不亲自面试招人，就这种态度，哪里能快速找到靠谱人让自己解脱出来。所以说，有关于人的关键问题，在处理的时候，尽量放下手里的具体业务，先把人的问题解决了，后面的路也

就好走了。

关于未来的事情。

据我观察，大部分人忙的是解决现在的问题，或者过去遗留的问题，这些事情固然重要，但是都只代表了过去，难以让企业和团队在未来产生效益。所以，一定要拿出时间来做关于未来的事情，比如一个新的产品和业务方向。对于个人来说，也不能故步自封，我看到不少的牛人学习能力很强，自身有专长，但是依然保持着不断学习的习惯，学习新事物，观察新趋势，随着知识爆炸和大数据分析的普及，靠着经验和吃老本的希望越来越小了。

想起来前几天和小米的首席架构师崔宝秋老师聊大会的专题设置，我们回顾了过去几年的演讲内容，发现变化很大，也意味着 IT 领域的知识和经验更新换代很快，需要不断的抓住新的趋势。我在之前一篇有关工匠精神的文章中也提到，工匠精神不是简单的一味加班，更重要的是找准努力的方向，现实中有人会钻牛角尖，盯着一个不可能有未来的方向坚持走下去，把自己慢慢困住了。

关于新的模式和流程，这和第二点说的未来还不太一样。

团队一定要定期复盘，不断改进现有事务的流程，解放生产力。我曾经见过一个团队 lead，工作非常努力，团队也做的很辛苦。他经常抱怨，现有的流程太繁琐，消耗大量的精力。我说为什么不试着改变一下呢，他吃惊地看着我说，这怎么行，这个流程是从公司创立就是这样的，规矩是老板定的 .....

公司看的是业绩，不是流程能不能改，即使有相关的规定也不是一成不变的，谁会和钱过不去呢，当你把效果摆出来的时候，没有人会记得什么所谓的规矩，所以你值得优先花一些时间来研究现有的问题，找到改进的点。记住，一个流程的改进产生的价值要比单个业务的成绩更有价值，也更能体现一个人的水平。

老板安排的事情。这个不多说了，和你的钱包大小直接挂钩，优先处理比较好，即使有不理解的地方，也可以边做边理解。很多时候你会发现，人家当老板是有道理的。

OpenJDK将对Android开发产生怎样的影响

作者 Abel Avram 译者 邵思华

Google 已决定将从下一版本的 Android 开始采用 OpenJDK，本文将部分摘录互联网上对于这一决定的反响。

在去年年底，我们曾提到 Google 已经决定在 Android 中使用 OpenJDK，以取代基于 Harmony 实现的 Java 库（详情请见此处）。尽管这条消息在宣布时恰逢圣诞期间，但 Google 的这一决定还是在互联网上引起了很大的反响，我们将在本文中对于这些观点进行一次总结。

从这个 Git 工单可以看出，早在 2015 年二月，Google 就已经在代码中露出了切换至 OpenJDK 的计划。在去年十二月，这次代码提交中所包含的一个重要的授权信息的变化被媒体曝光了。Android N 中所使用的新 Java 库将不再基于 Apache License 2.0（APL）授权协议，而是基于 GNU GPL 2 协议，并且在版权信息中包含了以下声明：“Oracle 及其附属机构版权所有，1997，2011”。

Mozilla 的前任 CTO Andreas Gal 为此专门写了一篇标题有些骇人的博客“Oracle 将它的魔爪伸向了 Android”。他表示，Google 长期以来一直坚持使用 Harmony 的 Java 库及 Apache 授权，其原因在于：用户能够任意使用及修改 APL 代码，而无需发布这些改动。换句话说，你能够进行具有专利权的改动与改进。而这一点对于基于 LGPL 授权协议的 GNU libc 来说是不可能的。我可以确信地说，我知道为什么 Google 认为这一点很重要，因为在发布 Firefox OS 的过程中，我曾经和许多与 Google 有合作关系的芯片供应商以及 OEM 厂家进行过交流。芯片与 OEM 厂商都希望在软件层

面上表现出他们的优势，尝试对 Android 的代码进行全方位的改进。尤其是芯片厂商经常会改动类库中代码，以充分利用自家的专利芯片，而且他们不愿意公开分享这些改动。通过这种方式能够体现出他们的竞争优势，即在专利上的优势。

Bob Ross 回复了 Gal 的文章，他自称是某家 OEM 厂商的员工，对于 APL 与 GPL 的争论提出了一些见解：我们确实会对 libcore 进行一些改动，在这种场合，主要问题是进行开源会带来很大的工作量，倒不是说要保护这些代码。至少从我曾经参与过的改动来看，情况就是如此。

Bradley M. Kuhn 目前担任自由软件管理机构（Software Freedom Conservancy）的主席，同时也是自由软件基金会（Free Software Foundation）的董事会成员。他对于 GPL 可能对 Andoid 开发所造成的影响有着不同的见解。在最近的一篇博客文章“Sun、Oracle、Android、Google 以及 JDK 复制权（copyleft）的质疑”中，Kuhn 注意到 OpenJDK 授权其实属于一个“非常宽松”的协议，即包含 Classpath 例外的 GNU 协议。Kuhn 曾经参与了 Classpath 例外协议的设计与命名，这一协议旨在避免通过复制权保护的方式“感染”整个 Java 生态系统，否则所有的 Java 程序都将被迫选择可以免费使用及重新分发的方式。如此一来，从授权协议的角度来看，选择使用 OpenJDK 与使用 Harmony 也没有多大的区别了。

按照 Kuhn 的说法，采用了 Oracle 的 GPL 及 Classpath 例外协议的 JDK 与具备 Apache 授权的 Java userspace 又有多大的差别呢？它们的差别其实并不大！Android 的重新分发者已经在 kernel space 方面承担了很大的复制权责任，并且请你记住，Webkit 是基于 LGPL 授权协议的，所以说围绕着 Android 已经存在着一些比较宽松的复制权遵循责任了。如此一来，如果说某个重新分发者已经满足了以上协议，那么要遵循那些新加入 JDK 代码中的需求也不是什么麻烦事，因为这些需求只有更为宽松。

但在 Gal 看来，Oracle 对于 Android 的未来发展将产生重大的影响，这不仅仅是因为授权的原因，同时也受到 Java 的发展路线、商标、条约与专利的影响。

除了源代码之外，Oracle 还有别的方法可以控制 Java 的发展，因此 OpenJDK 所谓的自由性就好像一所监狱。你可以投票决定外墙有多高，甚至可以去参与砌墙工作，但一旦你进入这里，就只有 Oracle 能够决定你何时才能出去。Oracle 对于 OpenJDK 的路线图有很大的决定权，通过对于兼容性需求、商标、现有协议以及 API 版权控诉（Oracle 与 Google 之间的控诉）的掌握，Oracle 几乎全盘控制了 OpenJDK 的发展方向。

部分读者在 Gal 的博客中留言表示，如果 Oracle 不能胜任 OpenJDK 的发展，那么 Google 完全可以创建一个分支，并自行决定它的发展方向。

Gal 同时预测，对于 Android 来说，接下来的一年注定是艰难的一年。

由于代码与技术的混杂，这将在战术层面上牵连 Android 的开发。不夸张的说，所改动的代码将达到几百万行，并且从实现方面来看，新的 OpenJDK 与 Google 原本采用的 Harmony 代码在正确性或性能表现上有许多微妙的差别。如你所见，Google 在日期数据的处理上更正了一个针对特定边界条件的测试用例。这是由于 Harmony 与 Oracle 的 OpenJDK 的表现有所差别，因此必须对测试进行更正。

而 Android 应用的整个生态系统就建立在这些正面临着变化基础上。Android 的应用商店中

有几百万个应用都依赖于 Java 的标准类，正如上文所述，不仅必须对测试进行更正，并且由于 OpenJDK 的转换所产生的微妙差别，这些应用都有可能随机地发生错误……

由于这次的巨变，我感觉 Android N 已经很难按期发布了。Google 的做法无异于在飞行途中更换引擎，此时优先级最高的任务是保证不会坠机，至于是否能够按时抵达目的地，Google 已经没时间去担心这个了。

Brendan Eich 在一条推文中表示支持 Gal 的意见：“虽然我们的所知有限，但我同意 @andreasgal 的看法，代码的改动将带来成本与风险的上升。”

Codename One 的联合创始人之一的 Shai Almog 也同意 Google 和 Android 的开发者将不得不面对一些额外的工作，但并没有 Gal 与 Eich 所认为的那么严重，并且使用 OpenJDK 还能够多多少少让他们受益。

虽然有些改动能够让用户直接受益，但对于大多数软件开发过程来说，改动无法带来直接的受益。并且也不是所有开发者都能够完全利用每一个语言与 API 的特性。

由于基础代码库得到了统一，用户将从中受益，并且安全审核流程也可以专注于这个具有更高统一性的基础代码库了。其结果是许多标准 Java 工具在 Android 上或许能够表现得更出色……

没错，Google 需要付出一定精力以完成这一过程，也确实会有一些应用可能会受到影响。但我敢说，这次改动比起 Marshmallow（即 Android 6.0）的改动所带来的影响会小的多，并且 Google 本身有一些工具能够让许多问题得以缓解（例如 SDK 版本提示）。

有些人怀疑 Google 决定采用 OpenJDK 是否和他们与 Oracle 之间的控诉纠纷有关。而 Kuhn 则相信 Google 这次决定的背后是出于技术方面的原因：

一位 Java 业界分析师（他在这一行已经有十年以上的经验了）告诉我，他相信这一决定的主要动力是技术方面的原因。在 Android 平台上开发 userspace 应用的作者们都在寻找新的 Java 语言实现，考虑到市面上已经存在了这个合理的、具有授权的免费软件，因此 Google 就有理由选择在技术上转换至这个更有优势的代码库，毕竟它为 API 的使用者在技术层面上提供了他们想要的东西，同时也降低了维护的难度。这样看来，这个决定是非常合理的。这种说法或许没有权威人士的观点那样令人震撼，但技术方面的原因的确很可能是这个决定的主要推动力。

Android 从新版本操作系统开始将采用 OpenJDK，这一举措会带来怎样的影响，目前来说还难以进行全面的透彻分析，这很大程度上取决于 Oracle 与 Google 之间争论不休的版权控诉将走向何方。目前为止，还没有人能够清楚地说明一个 API 接口是否能够拥有版权信息，法律与法院必须明确地解答这一点。自从上一个现有版本的库开始，Android 中的部分代码，包括 Java 与 C 在内就开始了重新构建工作，某些无用的代码被删除，但依然保留了接口或头文件。那么是不是说 Android 从此就可以摆脱那些麻烦了呢？这还有待时间观察。不过采用 OpenJDK 之后应该能够起到一些缓解作用，因为 Google 如今已经满足了 Java 授权和专利许可，Oracle 也不能再对 Android 说三道四了。

Python将迁移到GitHub

作者 Sergio De Simone 译者 适兕

Python 目前的维护者，Brett Cannon，日前在 Python 的核心工作流邮件列表中宣布了 Python 将迁移到 Github 中，在与 InfoQ 的对话中，Cannon 解释了决定此次迁移花了超过一年的时间，当初主要的考虑有如下三个备选方案：

- 创建 forge.python.org 来托管 Python 的仓库；
- 迁移到 Git、GitHub、和 Phabricator；
- 迁移到 Git 和 GitLab。

最后到决定是选择了 GitHub，主要归结为以下三个方面的原因：

GitHub 和 GitLab 在功能方面基本差不多；但是，Cannon 这里特别提到，GitLab 的开源与否根本就不是决定性的因素。

活跃的开发者均熟悉 Github，无论是核心开发者还是外围的贡献者。另外就是，虽然有一些开发者明确的反对迁移到 GitHub，但是没有一个说如果社区就这么决定了就没有人使用 Github 了。

Guido van Rossum 偏爱 Github，Cannon 考虑到尽管现在 Rossum 只是偶尔贡献一下，但他的影响力仍在，为了避免潜在的冲突，应当考虑 Rossum 的感受。

Cannon 向 InfoQ 谈到他是如何做这个决定的：基本上我这次的所作的决策过程和原来做过的两次没有太大的不同，过程都是这样的：我向 PEP 请求关于问题的可能的解决办法，基于所提出的议题来进行讨论（尽管讨论通常都是流于形式的，但是 PEP 会自始自终都保留详细记录，有最新的建议一般都会通知到），制订各

种期限，比如测试实例这样的，到那时，当我要做出决策的时候，我所基于的素材就非常的丰富了。

这里值得一题的是 Python 使用 GitHub，仅用于其代码托管和代码审核的支持，这也就意味着 Python 的缺陷跟踪和维基百科不会迁移到 GitHub 上。

Cannon 还讲过 Python 的核心开发者们彼此之间也是争论的不可开交。Stefan Krah 所表达的观点其实是代表了开发者当中倾向于使用 GitLab 的人们，为此，Cannon 专门进行了回复，并收到了很多未抄送给邮件列表的私聊回复，均回答了假定 GitHub 是首选的话大家是否会停止提交代码？他还补充到，在他看来无论是迁移到 GitHub 或者是其它的仓库都会有一小部分人的不适应，但必须要妥善处理。

InfoQ 借机采访了 Brett Cannon，希望更加深入的了解到此次决定所能带来的好处，在整个流程中目前处于何种地步。等等。

你能解释下 Python 和 Python 社区迁移到 Github 有何好处吗？

我目前正在写的一个 PEP 的草稿可以解答一部分，但是我们所期望的好处是可以做到更快速的补丁审核以及人们能够更加容易的参与到社区（真正的关键还是前者，但后者属于锦上添花）。希望是所有的工具都是或可以是围绕着 GitHub 所构建，能够做到为 Python 开发团队过去需要手动去做的大量的工作均替代为自动化，减少花时间去审核补丁的时间，从而提高生产效率。（目前我们最大的问题就是在缺陷跟踪上对于外来贡献者特别的不好，甚至让他们非常的不爽）。更何况不论是开发团队还是更为广泛的开源社区均对 GitHub 熟悉有加，而且我希望的是让所有人参与其中能够更加的快速和方便。

目前的状态是什么？下一步将会做什么？

说到状态，我是在 2016 年 1 月 1 日对 Python 的开发迁移到 GitHub 上这个决定的，现在的话我正在撰写关于我们各个代码仓库迁移所需要的所有步骤的 PEP（在上一个问题的回答中我给出的链接）。一旦在我们的核心工作流邮件列表中达成共识，且 PEP 会更加的完善、突出一些细节。在那之后，我们就会开始我们的工作。

至于具体的接下来的工作，他们要做的就是解决掉那些个会妨碍代码仓库迁移的“拦路虎”。因为我们迁移仓库所花费的困难是取决于他们所需要的工具，我希望是首先解决掉所有仓库的通用问题，然后再根据具体的仓库问题具体的解决。

你在公告中提到邮件列表中仍然有一些争论存在，你希望以何种方式来结束这场讨论？

你指的是在我发表过决定之后在核心工作流邮件列表中的讨论吧！我对此结果非常的满意。虽然有一些人因为 GitLab 拥有一个开源的版本就愿意去选择它，但是所有人都理解我为何做出如此的决定，而且支持我的坚持。大家还是对此次迁移持积极态度的，而且利用此机会让我们的开发平台尽可能的保持平台无关性，在未来能够不懈的追求更加的简单（这一定能够实现，自从我成为核心开发者之后，这算是第三次改动了，而且 Python 到今天已经走过了第 25 个年头，依然保持强劲的势头，当然，在接下来的几年，我们是不会再做平台的变换的了）。

开源项目近期往 Github 上迁移似乎渐渐的增多，你是否有过担心，如其他人那样认为如此依托给一个商业公司是不靠谱的？你认为这会给 Python 带来困扰吗？

对于未来的平台发生改变的情况还是非常担心的（将来某个时候一定会发生）。但是我们将仅使用 GitHub 来托管代码以及用来做代码审核，前者是很容易找到替代产品，而后者的话，一旦关闭某个 pull request 历史，其临近的也就没有什么价值了。也就是说，我们会对代码审核的历史做备份，那么我们一旦找到替代者，我们可以得到有效的代码审核，因为审核历史是有价值的，哪怕是直有一条接受的提交。如果 GitHub 不能提供开放的 API 给我们访问数据以及提高壁垒的话，我们当初就不会考虑它。另外，诸如 GitLab 之类的平台会提供一些工具让你来替代 GitHub，包括 pull request 都可以导入到他们的平台，我们知道我们并不会失去什么，但是 GitHub 可以给我们最快的时间。还有就是我们的缺陷跟踪系统不会迁移到 GitHub 上去，这就让那些担心失去控制的稍稍放松一些，缩小了一些改动的范围。

鏖战双十一-阿里直播平台面临的技术挑战

作者 陈康贤

关于作者

陈康贤，淘宝花名龙隆，淘宝技术部技术专家，著有《大型分布式网站架构设计与实践》一书，在分布式系统架构设计、高并发系统设计、系统稳定性保障等领域积累了较为丰富的实践经验，对新技术有浓厚的兴趣 ，交流或简历请发送至 longlong@taobao.com。

前言：一直以来双十一都是以交易为重心，今年当然也是如此，但是这并不妨碍万能的淘宝将双十一打造的让用户更欢乐、体验更丰富、玩法更多样、内容更有趣，因此，今年也诞生了以直播为特色的游戏双十一会场，也就是本文所要着笔重点介绍的，即阿里直播平台在双十一所面临的复杂技术挑战以及技术选型的台前幕后。

大型直播的技术挑战体现在大流量、高并发场景下，视频流的处理、分发，播放质量保障，视频可用性监控，超大直播间实时弹幕及聊天互动，高性能消息通道，内容控制（包括视频内容自动鉴黄及消息文本过滤），以及系统可用性、稳定性保障等等方面。本文将针对其中的一些技术细节，抽丝剥茧，希望通过些许文字的分析和介绍，能让大家有所启发。

视频直播

对于直播平台来说，为了保障各种网络环境下能够流畅的观看视频，需要将高清的输入流转换出多路不同清晰度的视频流，以支持不同网络条件下视频清晰度的切换，而由于不同的端所支持的协议及封装格式并不完全相同，比如无线端的 HTML5 页面可以很好的支持 HLS 协议，但是对于 RTMP 这类协议基本无能为力，而 PC 端为了降低延时，需要采用 RTMP 这一类流媒体协议。因此，为了支持多终端（PC、Andriod、IOS、HTML5）观看，需要对输入流进行编码及封装格式的转换。转码完成之后，还需要对视频流进行分发，毕竟源站的负载能力有限，节点数有限，离大部分用户的物理距离远，对视频这一类十分占用带宽资源的场景来说，为了提高播放质量减少卡顿，需要尽量减少到用户的传输链路。因此，通常的做法就是将视频流进行切片存储到分布式文件系统上，分发到 CDN，或者是直接通过 CDN 进行流的二级转发，因为 CDN 离用户最近，这样才能保证直播内容对于用户的低延时，以及用户的最短路径访问。客户端对延时的要求，以及采用何种协议，决定了视频是否需要分片，分片的目的在于，通过 HTTP 协议，用户不需要下载整个视频，只需要下载几个分片，就可以播放，实际上直播与录播的技术是相通的，区别在于直播的流无法预测终结时间，而录播的视频流终止时间是已知的。对延时要求没那么高的场景来说，客户端可以采用 HLS 协议，毕竟 IOS、Andriod、HTML5 等无线端应用能够很好的兼容和支持，且常用的静态资源 CDN 可以不做相应改造，就能支持，但是 HLS 协议的一大天生缺陷就是延时较高，视频内容在切片、分发、客户端下载的过程中耗费了很长时间。对于时效性要求非常高的场景，就需要采用 RTMP、RTSP 一类的实时流媒体协议，来降低延时，并且，为了降低源站的压力，需要 CDN 边缘节点来做流的转发，那么，CDN 就必须得支持相应的流媒体协议，也就是通常所说的流媒体 CDN。

由于直播流由主播上传，如何控制违法违规内容特别是黄色内容，成了十分棘手的问题。令人欣慰的是，随着技术的发展，算法对于黄色图像的识别准确率已经很高，基本达到可以在生产环境应用的程度，因此我们也尝试在视频流分发之前，对视频帧进行提取，并且将图像交给算法进行识别，当超过预设的阈值时，可进行预警或者关停直播间。经过一段时间的实践，取得了一定的效果，降低了人力成本，但不可避免的是图像识别算法时间复杂度高，吞吐率较一般算法低。

直播整个链路是比较长的，包含有接流、转码、切片、分发、客户端下载、播放等众多环节，链路上任何一个节点有问题，都可能导致视频不能播放。因此，关键节点的监控就十分重要，除此之外，还需要对整个链路的可用性进行监控，比如，针对 HLS 协议来说，可通过监控相应的 m3u8 索引列表有没有更新，来判断视频直播流是否中断。当然，如需判断视频流中的帧有没有花屏、有没有黑屏，就更复杂了，况且，监控节点所访问的 CDN 节点与用户所访问的 CDN 节点可能并不在同一地域。当前中国的网络环境，特别是跨网段的网络访问，对于流媒体应用来说，存在较大的不可控因素，客户端网络接入环境对视频的播放有决定性影响。因此，收集终端用户的播放数据质量数据进行反馈，及时进行视频清晰度切换，显得特别重要。这些数据包括客户端的地域分布、播放卡顿信息、视频分片加载时间等等，根据这些信息反馈，可以较为全面的评估 CDN 节点部署是否合理，是否需要新增 CDN 节点，视频的转码参数对于不同机型的兼容性等等，及时进行调整以改善用户体验。（见图 1）

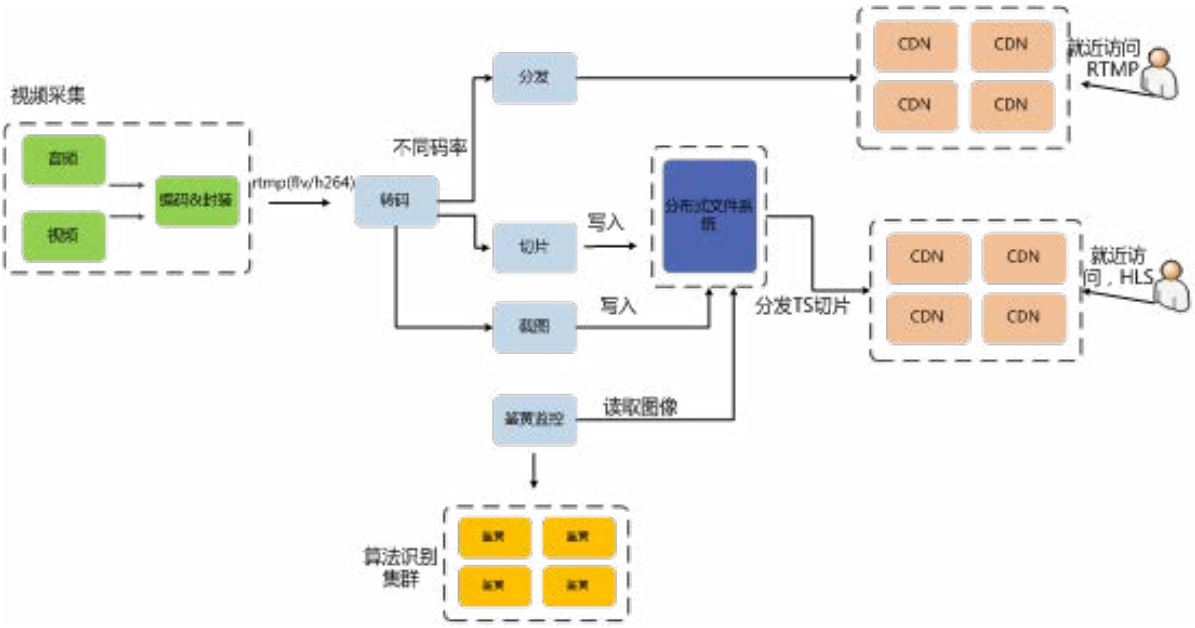


图 1 视频直播架构

消息 / 弹幕

WEB IM 应用及弹幕近年来有越来越火的趋势，是营销与气氛活跃的一种非常重要的手段。对于同时在线人数庞大的实时聊天互动、实时直播弹幕这一类场景来说，在保障消息实时性的前提条件下，将会面临非常高的并发压力。举个例子来说，假设一个活跃的直播间有 10w 人同时在线，正在直播一场热门的游戏赛事，假设每秒钟每个人说一句话，将会产生 10w 条消息，也就是 10w/s 的消息上行 QPS，而每条消息又需要广播给房间里面的每一个人，也就是说消息下行将成 10w 倍的放大，达到惊人的 10w*10w=100 亿 /s 的消息下行 QPS，而这仅仅是一场直播的 QPS，类似的直播可能有多场正在同时进行，对于消息通道来说，无疑将是一个巨大的挑战。因此，在系统设计的时候，首先要考虑的问题，就是如何降低消息通道的压力。（见图 2）

用户将信息投递到消息系统之后，系统首先对消息进行一系列的过滤，包括反垃圾、敏感关键词、黑名单等等，对于信息的过滤后面会详细介绍，此处暂且不表。为了避免系统被瞬间出现的峰值压垮，可先将消息投递到消息队列，削峰填谷，在流量的高峰期积压消息，给系统留一定裕度，降低因限流丢消息对业务产生的影响。而后端始终以固定的频率处理消息，通过异步机制保障峰值时刻系统的稳定，这是一个典型的生产者—消费者模型。对于消息的消费端，则可采用多线程模型以固定的频率从消息队列中消费消息，遍历对应房间所对应的在线人员列表，将消息通过不同的消息通道投递出去。多线程增加了系统的吞吐能力，特别是对需要将消息一次性投递给几万上十万用户这样的场景，可以异步使用大集群并行处理，提高系统的吞吐能力。异步使后端的消息投递可不受前端消息上行峰值流量的干扰，提高系统稳定性。

除了采用消息队列异步处理之外，当房间人数太多，或者消息下行压力太大的情况下，还需要进一步降低消息下行通道的压力，这就需要采用分桶策略。所谓的分桶策，实际上就是限制消息的传播范围，假设 10w 人在同一个房间

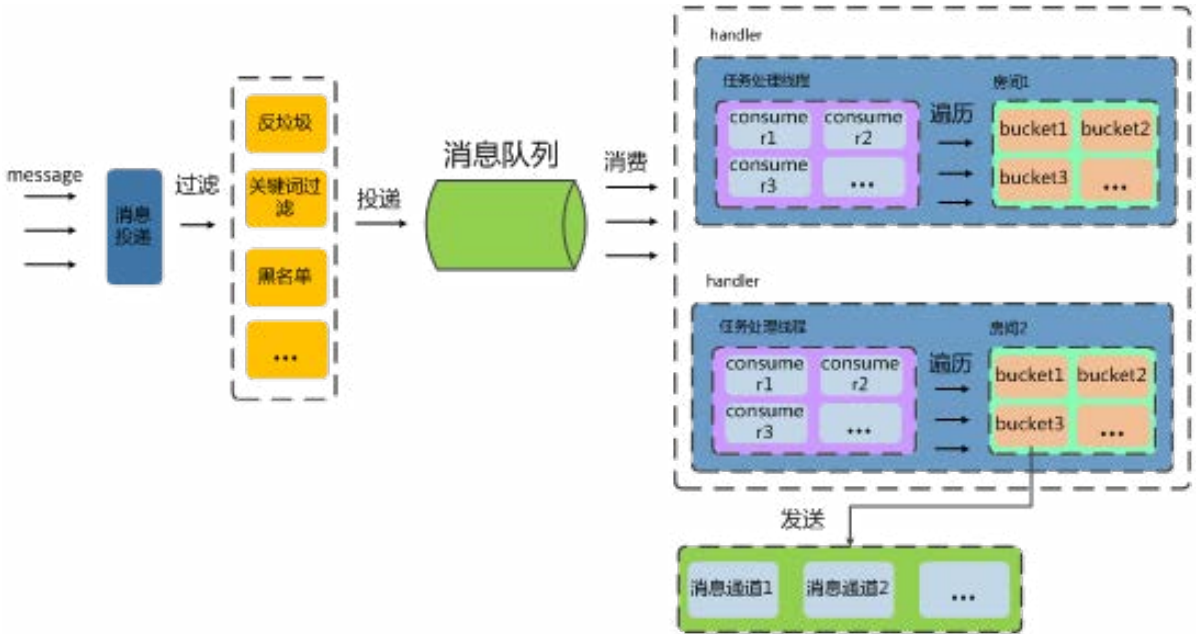


图 2 消息的投递与消费

聊天，每人说一句可能瞬间就会排满整个屏幕，消息在这种情况下基本没有可读性。为提高信息的可读性，同时也降低下行压力，可将每 10000 人（具体每个桶的容量可以根据实际需求来调整，这里只举例）放一个桶，用户发送的消息，只在一个桶或者部分桶可见，用户按照桶的维度接收消息，这样一方面前端用户接收到的消息量会少很多（跟用户所处的桶的活跃度相关），并且一条消息也不用发送给所有用户，只发送给一个或者部分桶，以降低消息下行压力。

分桶的策略有很多，最简单粗暴的方式就是根据用户随机。首先根据房间的活跃程度，预估房间该分多少个桶，然后将用户通过 hash 函数映射到各个桶，随机策略的好处是实现非常简单，可以将用户较为均衡的分配到每个桶，但是会有很多弊端。首先，准确的预估房间的活跃用户数本身就比较困难，基本靠蒙，这将导致单个桶的用户数量过大或者偏小，太大就会增加消息链路的压力，而偏小则降低用户积极性，后期调整分桶数也会很麻烦，需要将全部用户进行重 hash。另外从用户体验的角度来考虑，在直播过程中，在线用户数正常情况下会经历一个逐渐上升达到峰值然后逐渐下降的过程，由于分桶的缘故，在上升的过程中，每个桶的人是比较少的，这必然会影响到弹幕的活跃度，也可能因此导致用户流失，而在下降的过程中，逐渐会有用户退出直播，又会导致各个桶不均匀的情况出现，见图 3。

另一种方案是按需分桶，固定每个桶的大小，当现有的桶都满了之后，再开辟新的分桶，以控制每个桶的人数，使其不至于太多也不至于太少，这样就解决了之前可能出现的每个桶人数过少的问题，但是，有个问题将比之前的随机分桶更为明显，老的桶中不断有用户离开，人将逐渐减少，新开辟的桶将越来越多，如不进行清理的话，最后的结果仍然是分桶不均衡，并且会产生很多空桶，因此，就需要在算法和数据结构上进行调整，见图 4。

通过一个排序 list，每次将新增用户添加到人数最少分桶，这样可以让新用户加入最空闲的桶，以保持均衡，当桶满的时候，就不再添加新的用户，但是，当老用户离开的速度大大高于新用户进来的速度时，桶还是不均匀的，这时，就需要定期对分桶进行整理，以合并人数

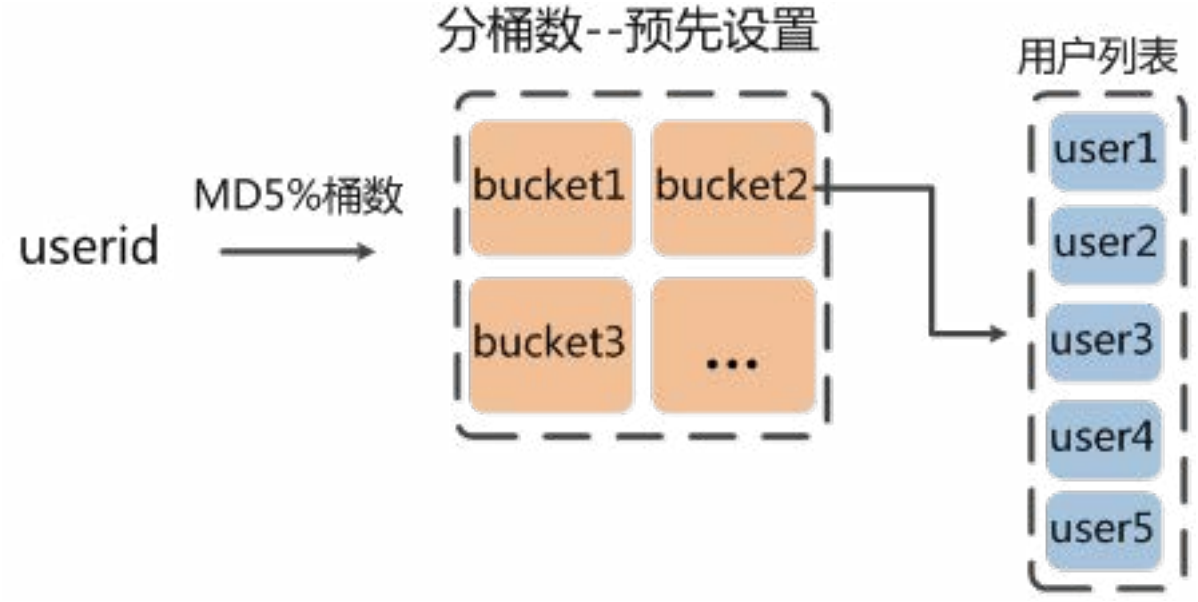


图 3 随机 hash 分桶

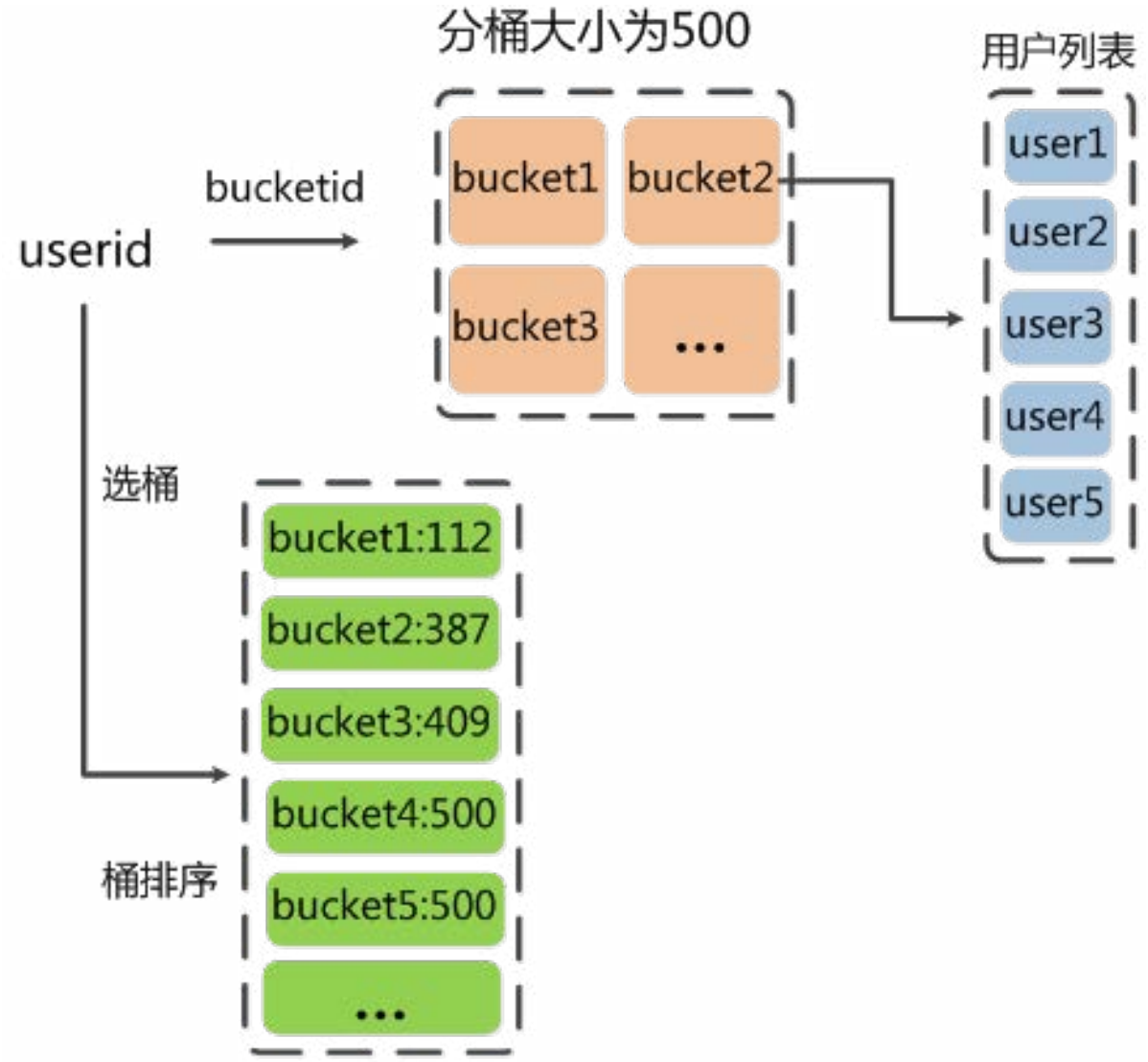


图 4 按需分桶策略

少的桶或者回收空桶，而合并的过程中，新用户又会不断的加入进来，并且，还需要保证消息发送时能读到正确的用户列表，在分布式高并发场景下，为了保证效率，有时候加锁并不是那么容易，这就有可能出现脏写与脏读，桶的整理算法将会非常复杂，有点类似于 JVM 中的内存回收算法。

与大数据量高并发场景下的分库分表策略类似，实际上分桶策略也是一种取舍权衡与妥协，虽解决了原有下行通道压力过大的问题，也引入了新问题。首先，分桶改变了原本普通用户对于消息的可见性，一条消息只对于部分桶的用户可见，而非所有桶的用户，这样不同的桶内的用户看到的消息可能是不同的，另一个问题是，以上的分桶策略有可能导致“热门桶”和“冷门桶”效应出现，即可能将很多“吐槽达人”分配到同一个桶，导致该桶的氛围十分活跃，而其他不那么活跃的用户分配到一起，以致于出现“冰火两重天”的局面，从而影响产品体验。当然，对于部分特殊的消息，如系统公告内容，或者是部分特殊角色（房间管理员、贵宾、授课的老师等等）所发送的消息，这一类消息需要广播给所有用户，这种情况下就需要系统对消息类型做区分，特殊的消息类型另作处理。

对于消息投递任务来说，需要知道消息将以什么方式被投递给谁，这样就需要动态地维护一个房间的人员列表，用户上线 / 下线及时通知系统，以便将用户添加到房间人员列表或者从房间人员列表中移除。用户上线十分简单，只需要在进入房间的时候通知系统即可，但对于下线用户的处理则有点折腾，为什么这么说呢？用户退出直播间的方式可能有多种，关闭浏览器 tab、关闭窗口、关闭电源、按下 home 键将进程切换到后台等等，有的操作可能可以获取到事件回调，但也有很多种情况是无法获取事件通知的，这样就会导致人员列表只增不减，房间的人越来越多，消息投递量也随之增加，白白的浪费了资源。为解决这一问题，就需采用心跳。

心跳指的是客户端每隔一段时间向服务端汇报在线状态，以维持服务端的在线人员列表，当同时在线人数达到一个很大的量级（如百万级）的时候，每秒心跳的 QPS 也会变得非常之高，如何保障心跳的高效率、高吞吐就成了岑待解决的问题。首先是通信协议的选择，是 HTTP 协议，还是 WebSocket，还是 TCP 协议或者其他。HTTP 协议的好处在于兼容性及跨终端，所有浏览器、Andriod、IOS 的 WebView，都能很好支持和兼容，在目前移动重于 PC 的大环境下，显得尤为重要，但是 HTTP 协议劣势也是显而易见的，作为应用层协议，单次通信所要携带的附加信息太多，效率低。WebSocket 在移动端的场景下比较合适，但是运用在 PC 端，需解决众多老版本浏览器的兼容性问题，socket.io 的出现则大大简化了这一原本非常复杂的问题。TCP 协议在传统的客户端应用上使用较多，但是对于 WEB 应用来说，存在天然障碍。使用 WebSocket 和 TCP 协议的好处显而易见的，通信效率会比 HTTP 协议高很多，并且这两种协议支持双工通信。

另一个问题是后端存储的选择，该使用怎样的存储结构来存储在线人员列表这样的数据结构，以支撑这么高的并发读写。经过优化并且使用 SSD 的关系型数据库相较以往性能已经有了很大的提升，但是对于频繁变化的大量在线人员列表来说，持久化存储实际上是没太大意义的。因此，读写性能更高的内存存储，如 memcache，redis，可能是一种更现实的选择。memcache 只能支持简单的 KV 对象存储，数据读写需要进行频繁的序列化和反序列化，但吞吐更高，而 redis 的好处在于丰富的数据类型，如 Lists、Hashs、Sets，省去了序列化和反序列化操作，并且支持更高效的分页遍历及 count 操作，见图 5。

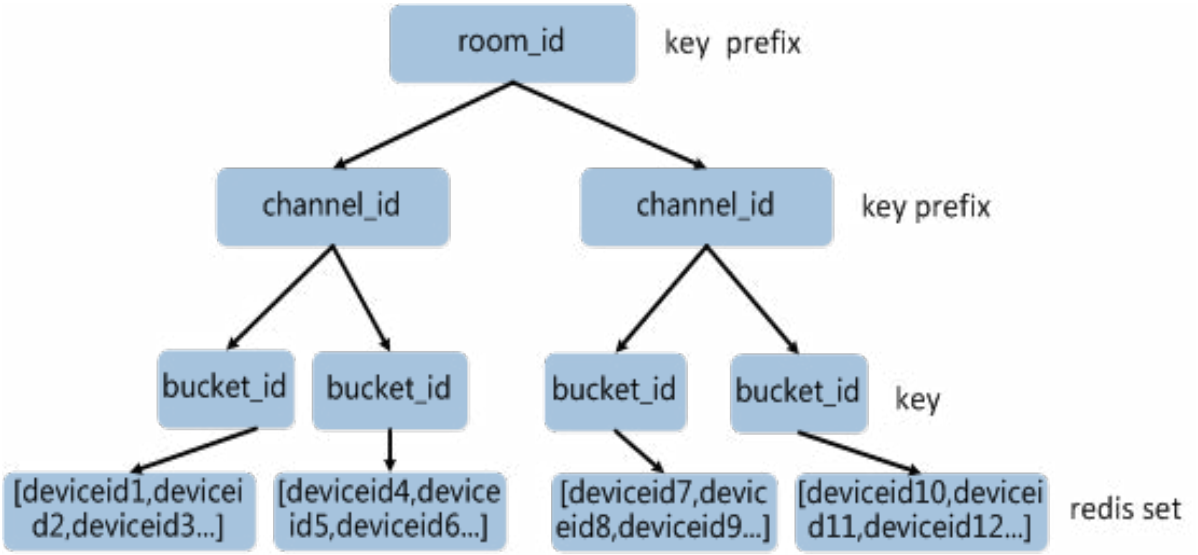


图 5 基于 redis Sets 构建的分桶存储结构

消息通道

HTTP 协议请求 / 响应式特性决定了它擅长处理浏览型业务，而对于需要与服务端进行频繁交互的即时通讯场景来说，则会显得十分蹩脚。在直播进行的过程中，用户可以对主播进行吐槽、评论，用户与用户之间也可以进行频繁的交流，需要有像弹幕、WEB IM 这样的工具来支持，这种场景下，消息的实时性尤为重要。

要实现这类场景，最简单最粗暴的方式莫过于不断地轮询应用服务器，采用拉的方式读取弹幕以及用户的聊天内容，消息的实时程度取决于拉的频率，拉的过快，可能服务器无法承受，拉的频率过低，则消息的实时性跟不上。轮询的方式太过于粗暴，需要频繁的请求服务器，成本较高，并且用户更新信息的频率实时变化，很难选择比较合理的轮询时间，因为不同时间段用户发送消息的频率是有很大差异的，对于拉取的信息，客户端需要进行筛选和去重。因此，对于 WEB 端的即时交互应用，需要采用其他解决方案，如 comit 服务端推送，或者通过 websocket 来实现类似的场景。

comet[1] 又被称作为反向 ajax(Reverse AJAX)，分为两种实现方式，一种是长轮询 (long-polling) 方式，一种是流 (streaming) 的形式。在长轮询的方式下，客户端与服务端保持 HTTP 连接，服务端会阻塞，直到服务端有信息传递或者是 HTTP 请求超时，客户端如果收到响应，则会重新建立连接，另一种方式是流的形式，服务器推送数据给客户端，但是连接并不关闭，而是始终保持，直到连接超时，超时后客户端重新建立连接，并关闭之前的连接。通过这两种方式，便可借用 HTTP 协议，来实现服务端与客户端的即时通讯。

而 WebSocket[2] 是 IETF 所提出的一种新的协议，旨在实现浏览器与服务器之间的全双工 (full-duplex) 通信，而传统的 HTTP 协议仅能够实现单向的通信，即客户端发起的到服务端的请求，虽然 comet 在一定程度上可以模拟双向通信，但是通信的效率较低，且依赖特定的服务器实现。（见图 6）

为何说 comet 的通信效率会低于 WebSocket 呢，因为不管是 comet 的长轮询机制还是流机制，都需要在请求超时后发送请求到服务端，并且，长轮询在服务端响应消息之后，需要重新建立连接，这样又带来了额外的开销。（见图 7）

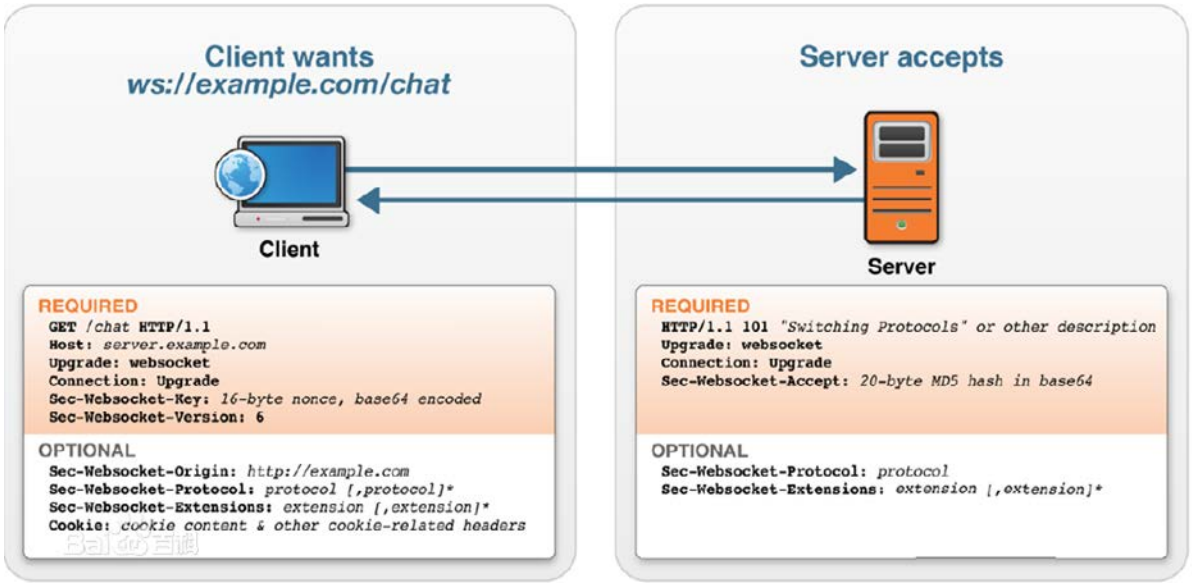


图 6 WebSocket 协议 [3]

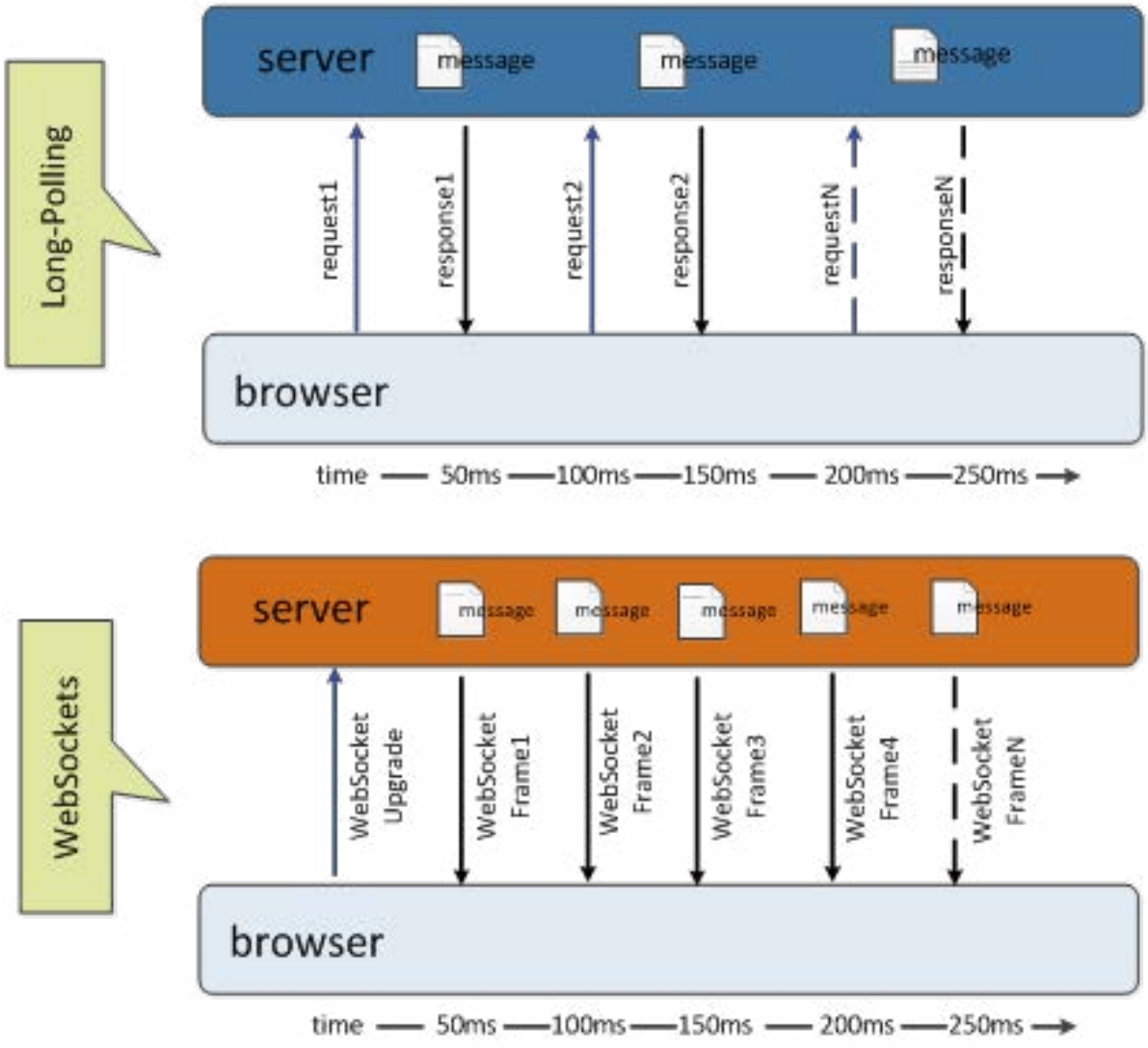


图 7 长轮询与 websocket 消息发送对比

我们知道 HTTP 协议的 Request Header 中附带了很多信息，但是这中间包含的很多信息有些场景其实并不是必须的，这样就浪费了大量的带宽及网络传输时间。而 WebSocket 协议使得浏览器与服务器只需要一次握手动作，便形成了稳定的通信通道，两者之间就可以通过 frame 进行数据传递，而消息传递所发送的 Header 是也是很小的，这样就会带来效率的提升。（见图 8）

通过 wireshark 抓包可以做一个简单的测试对比， 假设服务端每秒推送 50 条消息给用户，每条消息的长度为 10byte，对应不同的用户规模，采用 websocket 和 comet 两种不同的通信机制，所需要传递的字节数如图 8 所示，可见，随着用户规模及消息量的提升，采用 websocket 协议进行通信，将对通信效率带来数量级的提升，详细的测试过程可参见笔者博客。

引入 WebSocket 协议，虽然原则上解决了浏览器与服务端实时通信的效率问题，相较 comet 这种基于 HTTP 协议的实现方式，能获得更好的性能，但同时也引入了一些新的问题。摆在首位的便是浏览器的兼容性问题，开发 WEB 应用程序的一个最头痛的问题就是多版本浏览器的兼容，不同浏览器产商对于协议的实现有各自的理解，并且，市面上还充斥着大量低版本不支持 HTML5 协议的浏览器，且这部分用户在中国还占有较大基数，因此在产品研发的过程中不得不予以考虑。得益于 socket.io[4] 的开源，通过 websocket、Adobe Flash Socket、long-polling、streaming、轮询等多种机制的自适应切换，帮我们解决了绝大部分浏览器（包括各种内核的 webview）的兼容性问题，统一了客户端以及服务端的编程方式。对于不同平台的消息推送，我们内部也衍生了一些成熟的技术平台，封装了包括消息推送，消息订阅，序列化与反序列化，连接管理，集群扩展等工作，限于篇幅，这里就不多说了。

文本内容过滤

双十一视频直播的另一个挑战在于，如何对弹幕这一类的 UGC 内容进行过滤。根据以往经验，对淘宝来说，如果内容不经过滤直接透出，毫无疑问用户将看到满屏的广告，大量的广告信息将直接导致部分功能无法使用。因此，文本

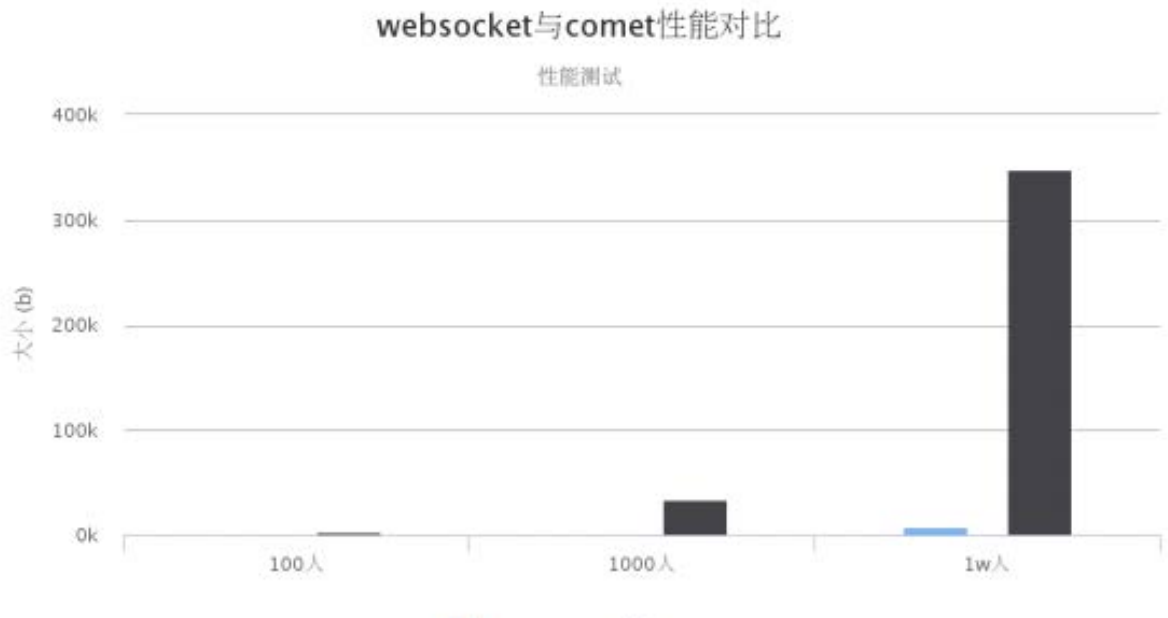


图 8 websocket 与 comet 性能对比

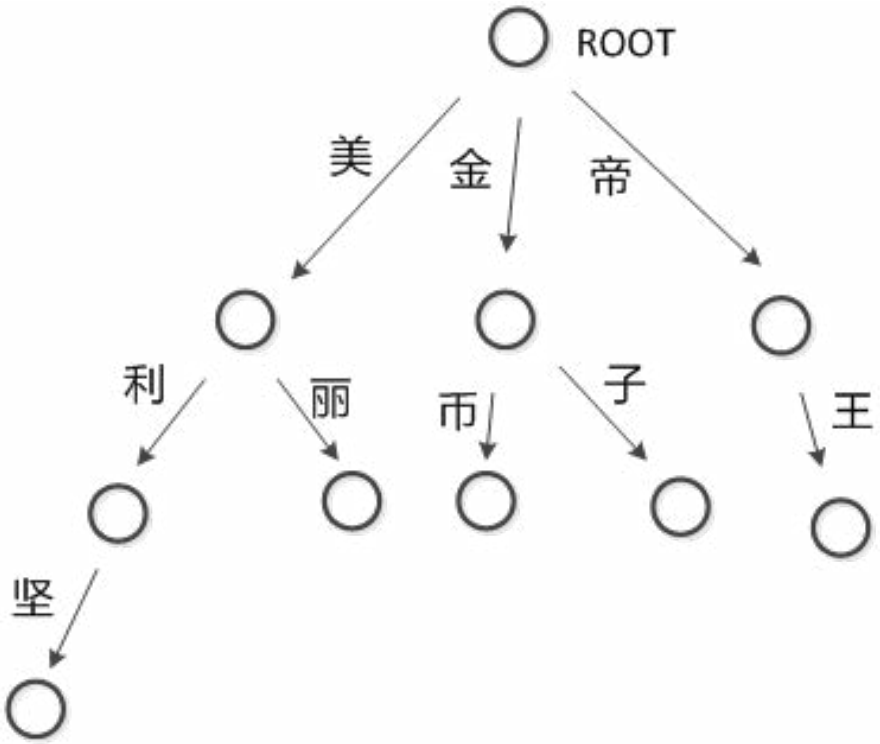


图 9 Trie 树结构

内容过滤，对于我们来说可以说是一门基本功。早期垃圾、广告内容，或者非法信息的识别，主要依赖于人工，但是当下 WEB2.0 DT 时代，对于海量的信息进行人工筛查显然不太现实，社会在发展，技术也在进步，一系列新的文本筛查及过滤技术被运用到了工程领域。

敏感词匹配

通过长时间的累积，大点的网站一般都会有一份敏感词列表，如果用户提交的内容包含敏感词，就需要进行转义（转换成 ***），或者是拒绝发表。那如何判断文本中是否包含敏感词，以及如何进行敏感词过滤呢？如果敏感词只有很少的几个，那最简单的方式当然是通过关键词搜索算法或者是正则表达式进行匹配，但实际情况显然要复杂的多，需要过滤的关键词可能有成千上万个，而正则表达式的效率实际上是比较低的，尤其在高并发场景下，对系统的吞吐能力要求非常之高。因此，这种场景下就需要有更高效的算法。目前较为常见的是采用 Trie 树算法或者是 Trie 算法的变种，如空间和时间复杂度都较为均衡的双数组 Trie 树算法来实现。Trie 树也称为字典树或者是单词查找树，它的好处在于可以利用字符串的公共前缀以减少查询时间，最大限度的减少字符串比较次数，降低关键词在内存中占用的存储空间。（见图 9）

Trie 本质上是一个有限状态机，根据输入的数据进行状态转移。假设敏感词包含“美利坚”、“美丽”、“金币”、“金子”、“帝王”，而输入的文本是“我的同桌是一个美丽的姑娘”，每读取一个字符作为输入，根据这个字符以及它的后续字符对 Trie 树进行查找，直到叶子节点，便可以找出文本中包含的违禁词“美丽”以及这个词在文本中对应的位置和频次。

当然，也可以通过构造多级的 Hash Table 来简化 Trie 树的实现，相同父节点的字放在同一个 Hash Table 中，以减少不必要的查询，对于输入的文本来说，只需要将字符一个个依次作为输入对 Hash Table 进行查找，如包含

对应的 key，则继续，直到查询到最下层叶子节点。（见图 10）

在系统变得越来越智能的同时，恶意的用户也变得越来越狡猾，他们可能会在文本当中夹杂一些干扰字符来绕过敏感词检查，如“美 * 利 * 坚”，这就需要我们对输入的文本做降噪处理，过滤掉其中的干扰字符，再进行匹配。

除了算法的实现之外，在集群环境下，还得综合考虑系统的吞吐以及机器内存的压力，由于敏感词过滤需要依赖庞大的敏感词词库，并且，还需将词库解析成 Trie 树或者多级的 Hash Table 置于内存当中，以便查找，这势必将占用大量的内存空间。因此，为了提升内存资源利用率，大部分情况我们会采用 RPC 模式部署敏感词过滤服务，但是，在高并发场景下，为了降低因引入敏感词过滤而带来的延时，提高系统的吞吐，又需要将敏感词数据推送到本地内存，通过读本地内存中的数据，降低因 RPC 带来的时延，当敏感词库有更新时，服务端需将相应的变更信息推送给其他应用。

分类算法

分类实际上就是按照某种标准来给对象贴标签，然后再根据标签进行区分，基于概率统计的贝叶斯分类算法 [5] 是最常见的分类算法，也是目前垃圾文本识别领域应用最广泛的算法。

使用贝叶斯分类算法进行二分类大致可分为这几个步骤：

1. 收集大量的垃圾内容和非垃圾内容语料，建立训练的垃圾语料集和正常内容的语料集。
2. 对语料文本进行分词，提取出独立的字符串，并且统计字符串在文本中出现的频次。
3. 每个训练语料集对应一个 hash table，比如垃圾语料集放在 hashtable_bad 中，而非垃圾语料集放在 hashtable_good 中，而 hashtable 中存储通过分词提取出的字符串以及对应的词频。
4. 计算 hashtable 所有的字符串出现的概率，即 P= 字符串的词频 / 字符串的总数。
5. 综合 hashtable_good 与 hashtable_

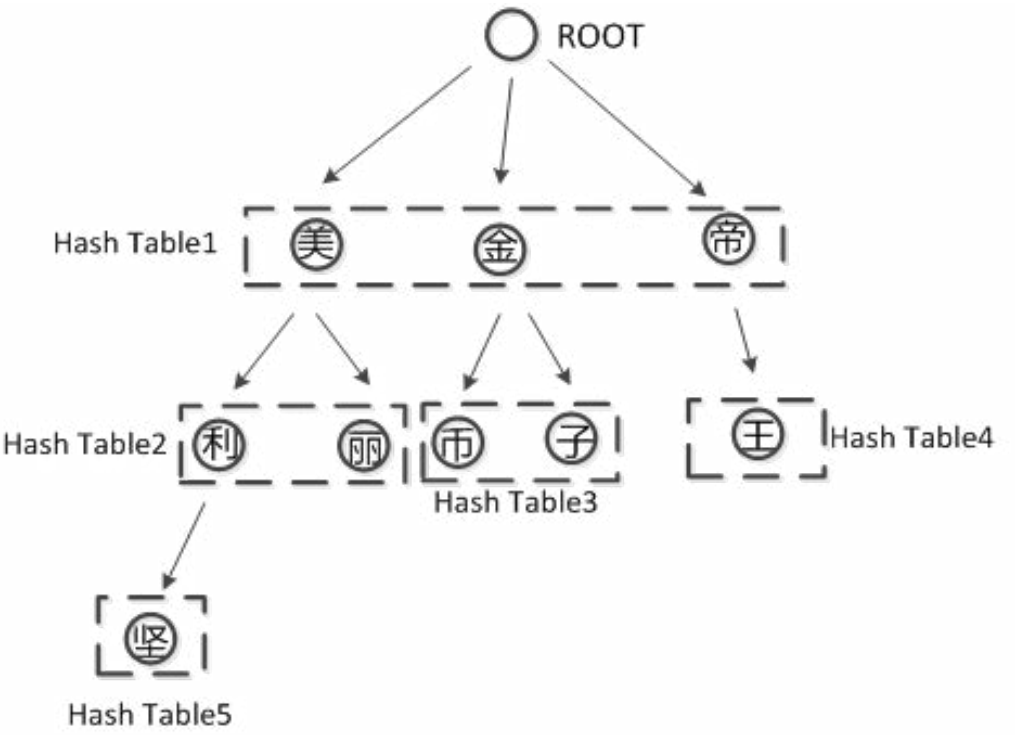


图 10 多级 Hash Table 简化 Trie 树的实现

bad，推测当一串文本中包含某个字符串时，该文本为垃圾内容的概率，对应的数学表达式如下：P(A|ki) = Pbad(ki) / [Pgood（ki） +Pbad（ki）]，其中事件 A 表示文本为垃圾内容，k1,k2 ……kn 代表提取的关键词，而 P（A|ki）则表示在文本中出现关键词 ki 时，该文本为垃圾内容的概率，Pbad（ki）为 ti 在 hashtable_bad 中的值，而 Pgood（ki）为 ki 在 hashtable_good 中的值。

6. 建立新的 hashtable_probability 存储字符串 ki 到 P（A|ki）的映射。

行文至此，贝叶斯分类的训练学习过程就完成了，接下来就可以根据 hashtable_probability 来计算文本为垃圾内容的可能性了。假设用户提交的文本内容经过分词得到 n 个关键词 k1,k2,k3……kn,hashtable_probability 中对应的值为 P1,P2……Pn ,P(A|k1,k2,k3……kn) 表示在用户提交的文本中同时出现关键字 k1,k2,k3……kn 时，该段内容为垃圾文本的概率，P(A|k1,k2,k3……kn) =P1*P2*……Pn 。当 P(A|k1,k2,k3……kn) 超过预定阈值时，可以判断该内容为垃圾内容，通过调整阀值，可以控制反垃圾系统对于内容过滤的严苛程度。

当然，以上只是简单的二分类情况，随着语料库的丰富，可以将垃圾语料进行细分，比如诈骗、广告、黄色、脏话、人身攻击等等，并且可以根据具体场景进行组合和分级使用，因为不同的使用场景对于文本中出现垃圾文本的容忍程度是不同的，要求越严苛的环境，势必误杀的概率也会增加，并且在某个场景下，一些语料可能属于垃圾广告内容，而在其他的场景下，可能又属于正常的内容。举例来说，如果在电商网站的评价中发现留有 QQ、微信、电话等联系方式，很有可能就属于广告内容，而对于社交应用来说，则属于正常内容。另外对于一些常见词，需要进行过滤，以减少对概率计算的干扰，词实际上也是有权重的，比如出现某些关键词，文档为垃圾内容的概率显著增加，可以适当提升此类词的权重。

贝叶斯算法的优势是随着语料库的不断丰富，可应对恶意用户层出不穷的“新把戏”。基于训练，算法模型可以发现已知类型的垃圾内容，但是对于新出现的类型，算法模型也无能为力，因此就需要外界的干预。要么通过人工标注，要么采用其他方式（如文本重复度判断），来发现新的垃圾文本类型，丰富训练的语料库。随着语料库的丰富，识别的准确性也会越来越高，道高一尺魔高一丈，最终在与恶意用户的斗争过程中，系统的能力也将越来越强大。

黑名单用户

对于频繁发送恶意信息的用户，可以采用禁言一段时间或者是永久禁言的的方式进行应对，这样就需要维护一份用户黑名单，当用户提交 UGC 内容的时候，先判断是否在黑名单中，如果已经加入黑名单，则拒绝发表。

黑名单的最简单实现方式莫过于采用 hash table，借用 redis 的 hash 这一类的数据结构进行内存缓存，并且定时进行持久化或是数据备份，防止宕机导致的数据丢失，这种方式实现简单，查询效率也比较高，可以满足大部分场景的使用，但是缺点也十分的明显，采用 hash table 的方式随着黑名单列表的增加，占用内存的空间越来越大，当业务需要对黑名单按一定场景进行区分和隔离的时候，这种情况尤为明显，并且，黑名单列表越大，hash 冲突也将越多，以致于查询的效率也会降低，。当对黑名单过滤要求不那么精确的时候，可以采用 Bloom Filter[6]（布隆过滤器）的方案。（见图 11）

Bloom Filter 是一个 m 位的数组，初始状态下数组所有位被置 0，需要设置黑名单时，通过一系列不同的 hash 函数，每个 hash 函数将

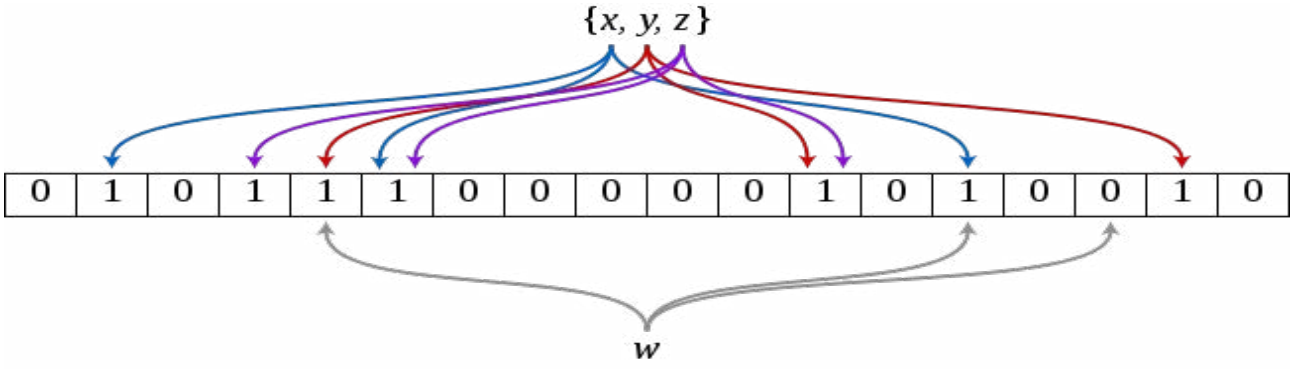


图 11 Bloom Filter 的基本原理 [7]

对应的输入元素映射到数组中的一位上，通过 hash 函数得到数组的索引，将数组对应位置 1，在查询的时候，通过相同的 hash 函数，找到对应的位，如果对应的位不都为 1 的话，则表示用户不在黑名单中，那么，当所有的位都为 1 的情况，就表示用户一定黑名单中么？也不一定，可能是其他几次用户输入刚好将本次所有位都置 1 了，如图 11 所示，当插入 x、y、z 这三个元素之后，再来查询 w，会发现 w 不在集合之中，而如果 w 经过三个 hash 函数计算所得索引处的位全是 1，那么 Bloom Filter 就会告诉你，w 在集合之中，实际上这里是误报，w 并不在集合之中，这就是所谓误报。从概率上来说，误报的几率很低，在系统可接受范围内，如需要 100% 精确判断，则 Bloom Filter 就不太合适。

基本的 Bloom Filter 是不支持删除操作的，Counting Bloom Filter 的出现解决了这个问题，它将标准 Bloom Filter 位数组的每一位扩展为一个小的计数器（Counter），在插入元素时给对应的 k（k 为哈希函数个数）个 Counter 的值分别加 1，删除元素时给对应的 k 个 Counter 的值分别减 1,Counting Bloom Filter 通过多占用几倍存储空间的代价，给 Bloom Filter 增加了删除操作，但是相比 hash table 而言，存储效率还是提升了很多。（见图 12）

系统稳定性

对于每年的双十一来说，如何在大流量高并发场景下，保障系统稳定提供服务不宕机，已经是一个经久不衰的话题，也是每年对于技术人

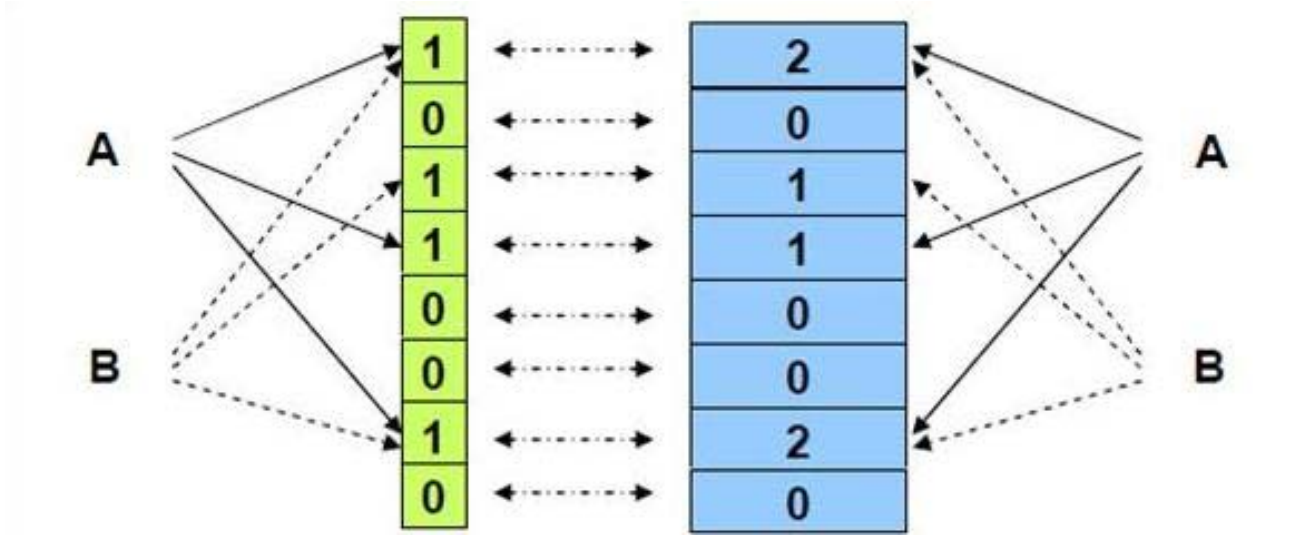


图 12 Counting Bloom Filter 的基本原理 [8]

员的一次彻底的检阅。从大促开始之前的性能优化、依赖梳理、峰值评估，到系统压力测试，新机器采购上线，再到流控阀值的评估、降级开关配置，再到监控数据采集、容灾应急预案、全链路压测、演习，再到实时、离线数据分析等等，关于这些的介绍，相信网上相关文章都已汗牛充栋，此处就不再赘述。

总结

本篇主要介绍了阿里直播平台在双十一所面临的一些技术挑战，以及应对的方案和思考过程，从音视频直播架构，到弹幕和消息投递，再到消息通道的技术选型、文本过滤的实现机制，大型直播无论是音视频的可靠性保障、视频鉴黄，还是文本消息经过数万在线用户放大之后的消息下行，以及对于消息内容本身的过滤，都面临了很大的挑战，挑战不可怕，兵来将挡，水来土掩，迎难而上，才能有突破和创新。

参考文献

1. comet, https://software.intel.com/zh-cn/articles/comet-java-realtime-system-essay
2. WebSocket, https://tools.ietf.org/html/rfc6455
3. WebSocket 通信原理
4. socket.io 项目地址：http://socket.io/
5. 贝叶斯分类，https://en.wikipedia.org/wiki/Naive_Bayes_classifier
6. Bloom Filter, http://my.oschina.net/kiwivip/blog/133498
7. 图片来源，http://my.oschina.net/kiwivip/blog/133498
8. 图片来源，http://my.oschina.net/kiwivip/blog/133498

从无到有：微信后台系统的演进之路

作者 张文瑞

从无到有

2011.1.21 微信正式发布。这一天距离微信项目启动日约为 2 个月。就在这 2 个月里，微信从无到有，大家可能会好奇这期间微信后台做的最重要的事情是什么？

我想应该是以下三件事。

1. 确定了微信的消息模型

微信起初定位是一个通讯工具，作为通讯工具最核心的功能是收发消息。微信团队源于广研团队，消息模型跟邮箱的邮件模型也很有渊源，都是存储转发。

图 1 展示了这一消息模型，消息被发出后，会先在后台临时存储；为使接收者能更快接收到消息，会推送消息通知给接收者；最后客户端主动到服务器收取消息。

2. 制定了数据同步协议

由于用户的帐户、联系人和消息等数据都在服务器存储，如何将数据同步到客户端就成了很关键的问题。为简化协议，我们决定通过一个统一的数据同步协议来同步用户所有的基础数据。

最初的方案是客户端记录一个本地数据的快照(Snapshot)，需要同步数据时，将 Snapshot 带到服务器，服务器通过计算 Snapshot 与服务器数据的差异，将差异数据发给客户端，客户端再保存差异数据完成同步。不过这个方案有两个问题：一是 Snapshot 会随着客户端数据的增多变得越来越大，同步时流量开销大；

图 1 微信消息模型

二是客户端每次同步都要计算 Snapshot，会带来额外的性能开销和实现复杂度。

几经讨论后，方案改为由服务计算 Snapshot，在客户端同步数据时跟随数据一起下发给客户端，客户端无需理解 Snapshot，只需存储起来，在下次数据同步数据时带上即可。同时，Snapshot 被设计得非常精简，是若干个 Key-Value 的组合，Key 代表数据的类型，Value 代表给到客户端的数据的最新版本号。Key 有三个，分别代表：帐户数据、联系人和消息。这个同步协议的一个额外好处是客户端同步完数据后，不需要额外的 ACK 协议来确认数据收取成功，同样可以保证不会丢数据：只要客户端拿最新的 Snapshot 到服务器做数据同步，服务器即可确认上次数据已经成功同步完成，可以执行后续操作，例如清除暂存在服务的消息等等。

此后，精简方案、减少流量开销、尽量由服务器完成较复杂的业务逻辑、降低客户端实现的复杂度就作为重要的指导原则，持续影响着后续的微信设计开发。记得有个比较经典的案例是：我们在微信 1.2 版实现了群聊功能，但为了保证新旧版客户端间的群聊体验，我们通过服务器适配，让 1.0 版客户端也能参与群聊。

3. 定型了后台架构

微信后台使用三层架构：接入层、逻辑层和存储层。（见图 2）

- 接入层提供接入服务，包括长连接入服务和短连接入服务。长连接入服务同时支持客户端主动发起请求和服务器主动发起推送；短连接入服务则只支持客户端主动发起请求。
- 逻辑层包括业务逻辑服务和基础逻辑服务。业务逻辑服务封装了业务逻辑，是后台提供给微信客户端调用的 API。基础逻辑服务则抽象了更底层和通用的业务逻辑，提供给业务逻辑服务访问。
- 存储层包括数据访问服务和数据存储服务。数据存储服务通过 MySQL 和 SDB（广研早期后台中广泛使用的 Key-Table 数据存储系统）等底层存储系统来持久化用户数据。数据访问服务适配并路由数据访问请求到不同的底层数据存储服务，面向逻辑层提供结构化的数据服务。比较特别的是，微信后台每一种不同类型的数据都使用单独的数据访问服务和数据存储服务，例如帐户、消息和联系人等等都是独立的。

微信后台主要使用 C++。后台服务使用 Svrkit 框架搭建，服务之间通过同步 RPC 进行通讯。（见图 3）

Svrkit 是另一个广研后台就已经存在的高性能 RPC 框架，当时尚未广泛使用，但在微信后台却大放异彩。作为微信后台基础设施中最重要的一部分，Svrkit 这几年一直不断在进化。我

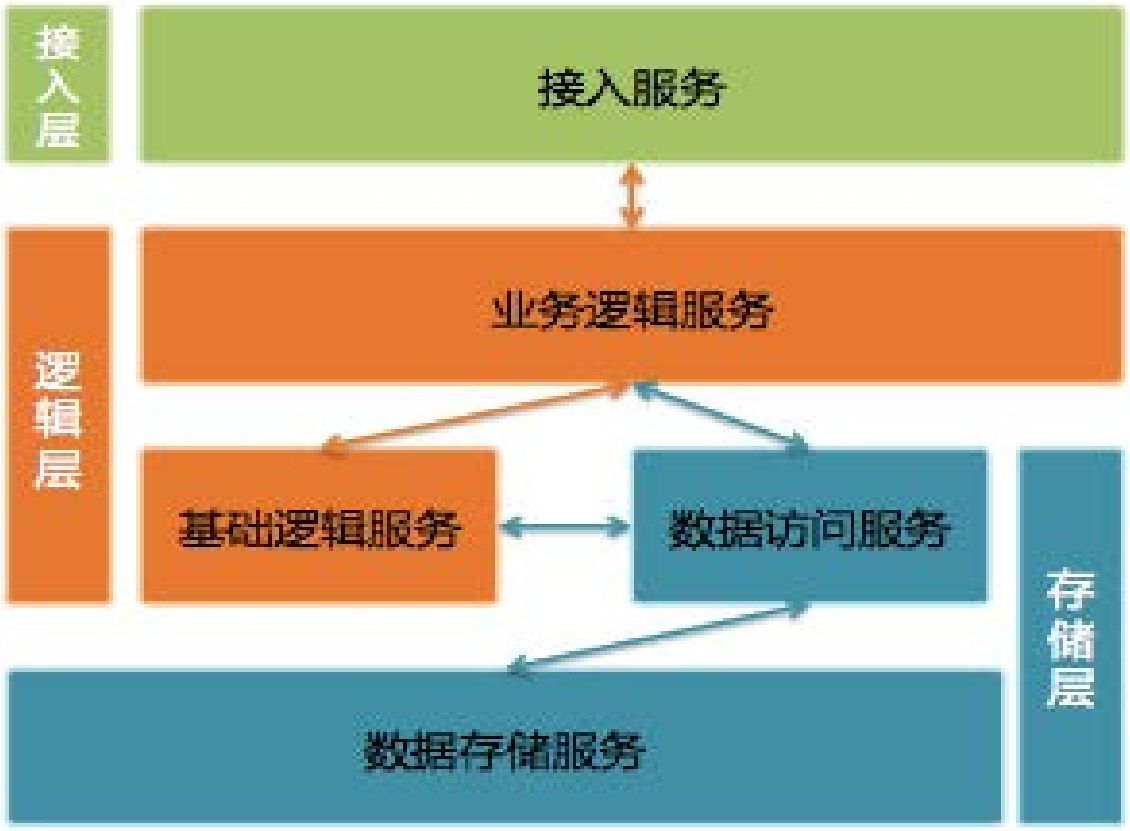


图 2 微信后台系统架构

们使用 Svrkit 构建了数以千计的服务模块，提供数万个服务接口，每天 RPC 调用次数达几十万亿次。

这三件事影响深远，乃至于 5 年后的今天，我们仍继续沿用最初的架构和协议，甚至还可以支持当初 1.0 版的微信客户端。

这里有一个经验教训——运营支撑系统真的很重要。第一个版本的微信后台是仓促完成的，当时只是完成了基础业务功能，并没有配套的业务数据统计等等。我们在开放注册后，一时间竟没有业务监控页面和数据曲线可以看，注册用户数是临时从数据库统计的，在线数是从日志里提取出来的，这些数据通过每个小时运行一次的脚本（这个脚本也是当天临时加的）统计出来，然后自动发邮件到邮件组。还有其他各种业务数据也通过邮件进行发布，可以说邮件是微信初期最重要的数据门户。

2011.1.21 当天最高并发在线数是 491，而今天这个数字是 4 亿。

小步慢跑

在微信发布后的 4 个多月里，我们经历了发布后火爆注册的惊喜，也经历了随后一直不温不火的困惑。

这一时期，微信做了很多旨在增加用户好友量，让用户聊得起来的功能。打通腾讯微博私信、群聊、工作邮箱、QQ/ 邮箱好友推荐等等。对于后台而言，比较重要的变化就是这些功能催生了对异步队列的需求。例如，微博私信需要跟外部门对接，不同系统间的处理耗时和速度不一样，可以通过队列进行缓冲；群聊是耗时操作，消息发到群后，可以通过异步队列来异步完成消息的扩散写等等。

图 4 是异步队列在群聊中的应用。微信的群聊是写扩散的，也就是说发到群里的一条消息会给群里的每个人都存一份（消息索引）。为什

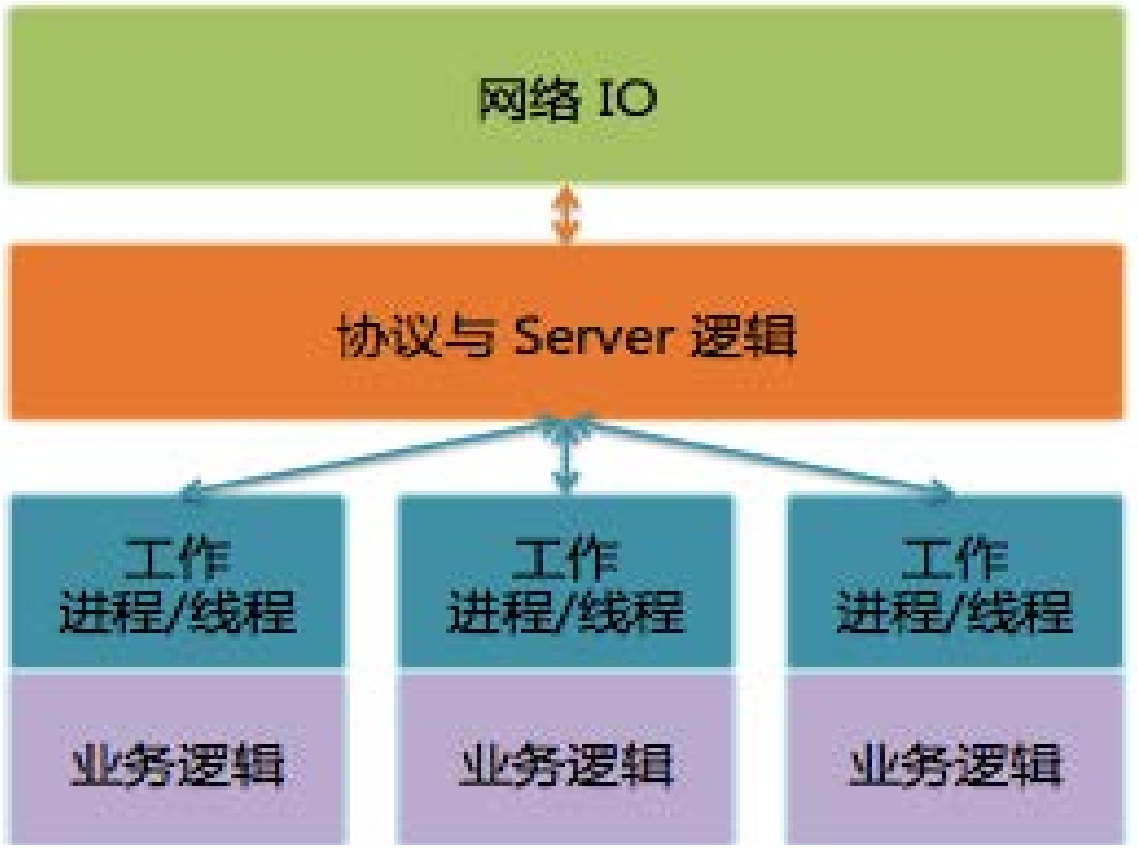


图 3 Svrkit 框架

么不是读扩散呢？有两个原因：

- 群的人数不多，群人数上限是 10（后来逐步加到 20、40、100，目前是 500），扩散的成本不是太大，不像微博，有成千上万的粉丝，发一条微博后，每粉丝都存一份的话，一个是效率太低，另一个存储量也会大很多；
- 消息扩散写到每个人的消息存储（消息收件箱）后，接收者到后台同步数据时，只需要检查自己收件箱即可，同步逻辑跟单聊消息是一致的，这样可以统一数据同步流程，实现起来也会很轻量。

异步队列作为后台数据交互的一种重要模式，成为了同步 RPC 服务调用之外的有力补充，在微信后台被大量使用。

快速成长

微信的飞速发展是从 2.0 版开始的，这个版本发布了语音聊天功能。之后微信用户量急速增长，2011.5 用户量破 100 万、2011.7 用户量破 1000 万、2012.3 注册用户数突破 1 亿。

伴随着喜人成绩而来的，还有一堆幸福的烦恼。

- 业务快速迭代的压力

微信发布时功能很简单，主要功能就是发消息。不过在发语音之后的几个版本里迅速推出了手机通讯录、QQ 离线消息、查看附近的人、摇一摇、漂流瓶和朋友圈等等功能。

有个广为流传的关于朋友圈开发的传奇——朋友圈历经 4 个月，前后做了 30 多个版本迭代才最终成型。其实还有一个鲜为人知的故事——那时候因为人员比较短缺，朋友圈后台长时间只有 1 位开发人员。

- 后台稳定性的要求

用户多了，功能也多了，后台模块数和机器量在不断翻番，紧跟着的还有各种故障。

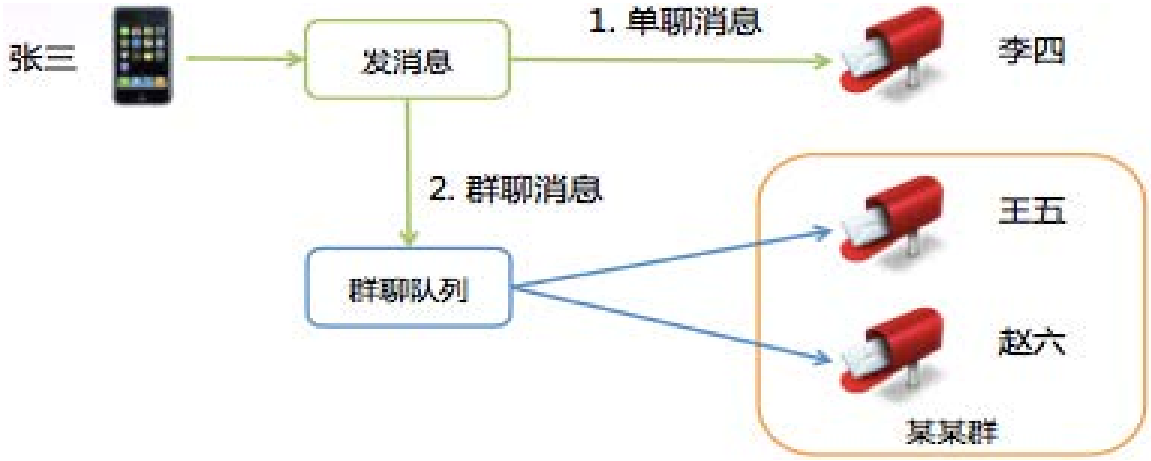


图 4 单聊和群聊消息发送过程

帮助我们顺利度过这个阶段的，是以下几个举措：

1. 极简设计

虽然各种需求扑面而来，但我们每个实现方案都是一丝不苟完成的。实现需求最大的困难不是设计出一个方案并实现出来，而是需要在若干个可能的方案中，甄选出最简单实用的那个。

这中间往往需要经过几轮思考——讨论——推翻的迭代过程，谋定而后动有不少好处，一方面可以避免做出华而不实的过度设计，提升效率；另一方面，通过详尽的讨论出来的看似简单的方案，细节考究，往往是可靠性最好的方案。

2. 大系统小做

逻辑层的业务逻辑服务最早只有一个服务模块（我们称之为 mmweb），囊括了所有提供给客户端访问的 API，甚至还有一个完整的微信官网。这个模块架构类似 Apache，由一个 CGI 容器（CGIHost）和若干 CGI 组成（每个 CGI 即为一个 API），不同之处在于每个 CGI 都是一个动态库 so，由 CGIHost 动态加载。

在 mmweb 的 CGI 数量相对较少的时候，这个模块的架构完全能满足要求，但当功能迭代加快，CGI 量不断增多之后，开始出现问题：

1） 每个 CGI 都是动态库，在某些 CGI 的共用逻辑的接口定义发生变化时，不同时期更新上线的 CGI 可能使用了不同版本的逻辑接口定义，会导致在运行时出现诡异结果或者进程 crash，而且非常难以定位；

2） 所有 CGI 放在一起，每次大版本发布上线，从测试到灰度再到全面部署完毕，都是一个很漫长的过程，几乎所有后台开发人员都会被同时卡在这个环节，非常影响效率；

3） 新增的不太重要的 CGI 有时稳定性不好，某些异常分支下会 crash，导致 CGIHost 进程无法服务，发消息这些重要 CGI 受影响没法运行。

于是我们开始尝试使用一种新的 CGI 架构——Logicsvr。

Logicsvr 基于 Svrkit 框架。将 Svrkit 框架和 CGI 逻辑通过静态编译生成可直接使用 HTTP 访问的 Logicsvr。我们将 mmweb 模块拆分为 8 个不同服务模块。拆分原则是：实现不同业务功能的 CGI 被拆到不同 Logicsvr，同一功能但是重要程度不一样的也进行拆分。例如，作为核

心功能的消息收发逻辑，就被拆为 3 个服务模块：消息同步、发文本和语音消息、发图片和视频消息。

每个 Logicsvr 都是一个独立的二进制程序，可以分开部署、独立上线。时至今日，微信后台有数十个 Logicsvr，提供了数百个 CGI 服务，部署在数千台服务器上，每日客户端访问量几千亿次。

除了 API 服务外，其他后台服务模块也遵循“大系统小做”这一实践准则，微信后台服务模块数从微信发布时的约 10 个模块，迅速上涨到数百个模块。

3. 业务监控

这一时期，后台故障很多。比故障更麻烦的是，因为监控的缺失，经常有些故障我们没法第一时间发现，造成故障影响面被放大。

监控的缺失一方面是因为在快速迭代过程中，重视功能开发，轻视了业务监控的重要性，有故障一直是兵来将挡水来土掩；另一方面是基础设施对业务逻辑监控的支持度较弱。基础设施提供了机器资源监控和 Svrkit 服务运行状态的监控。这个是每台机器、每个服务标配的，无需额外开发，但是业务逻辑的监控就要麻烦得多了。当时的业务逻辑监控是通过业务逻辑统计功能来做的，实现一个监控需要 4 步：

1. 申请日志上报资源；
2. 在业务逻辑中加入日志上报点，日志会被每台机器上的 agent 收集并上传到统计中心；
3. 开发统计代码；
4. 实现统计监控页面。

可以想象，这种费时费力的模式会反过来降低开发人员对加入业务监控的积极性。于是有一天，我们去公司内的标杆——即通后台（QQ 后台）取经了，发现解决方案出乎意料地简单且强大：

1）故障报告

之前每次故障后，是由 QA 牵头出一份故障报告，着重点是对故障影响的评估和故障定级。新的做法是每个故障不分大小，开发人员需要彻底复盘故障过程，然后商定解决方案，补充出一份详细的技术报告。这份报告侧重于：如何避免同类型故障再次发生、提高故障主动发现能力、缩短故障响应和处理过程。

2）基于 ID-Value 的业务无关的监控告警体系。（见图 5）

监控体系实现思路非常简单，提供了 2 个 API，允许业务代码在共享内存中对某个监控 ID 进行设置 Value 或累加 Value 的功能。每台机器上的 Agent 会定时将所有 ID-Value 上报到监控中心，监控中心对数据汇总入库后就可以通过统一的监控页面输出监控曲线，并通过预先配置的监控规则产生报警。

对于业务代码来说，只需在要被监控的业务流程中调用一下监控 API，并配置好告警条件即可。这就极大地降低了开发监控报警的成本，我们补全了各种监控项，让我们能主动及时地发现问题。新开发的功能也会预先加入相关监控项，以便在少量灰度阶段就能直接通过监控曲线了解业务是否符合预期。

4. KVSvr

微信后台每个存储服务都有自己独立的存储模块，是相互独立的。每个存储服务都有一个业务访问模块和一个底层存储模块组成。业务访问层隔离业务逻辑层和底层存储，提供基于 RPC 的数据访问接口；底层存储有两类：SDB 和 MySQL。

图 5 基于 ID-Value 的监控告警体系

SDB 适用于以用户 UIN(uint32_t) 为 Key 的数据存储，比方说消息索引和联系人。优点是性能高，在可靠性上，提供基于异步流水同步的 Master-Slave 模式，Master 故障时，Slave 可以提供读数据服务，无法写入新数据。

由于微信账号为字母 + 数字组合，无法直接作为 SDB 的 Key，所以微信帐号数据并非使用 SDB，而是用 MySQL 存储的。MySQL 也使用基于异步流水复制的 Master-Slave 模式。

第 1 版的帐号存储服务使用 Master-Slave 各 1 台。Master 提供读写功能，Slave 不提供服务，仅用于备份。当 Master 有故障时，人工切读服务到 Slave，无法提供写服务。为提升访问效率，我们还在业务访问模块中加入了 memcached 提供 Cache 服务，减少对底层存储访问。

第 2 版的帐号存储服务还是 Master-Slave 各 1 台，区别是 Slave 可以提供读服务，但有可能读到脏数据，因此对一致性要求高的业务逻辑，例如注册和登录逻辑只允许访问 Master。当 Master 有故障时，同样只能提供读服务，无法提供写服务。

第 3 版的帐号存储服务采用 1 个 Master 和多个 Slave，解决了读服务的水平扩展能力。

第 4 版的帐号服务底层存储采用多个 Master-Slave 组，每组由 1 个 Master 和多个 Slave 组成，解决了写服务能力不足时的水平扩展能力。

最后还有个未解决的问题：单个 Master-Slave 分组中，Master 还是单点，无法提供实时的写容灾，也就意味着无法消除单点故障。另外 Master-Slave 的流水同步延时对读服务有很大影响，流水出现较大延时会导致业务故障。于是我们寻求一个可以提供高性能、具备读写水平扩展、没有单点故障、可同时具备读写容灾能力、能提供强一致性保证的底层存储解决方案，最终 KVSvr 应运而生。

KVSvr 使用基于 Quorum 的分布式数据强一致性算法，提供 Key-Value/Key-Table 模型的存储服务。传统 Quorum 算法的性能不高，KVSvr 创造性地将数据的版本和数据本身做了区分，将 Quorum 算法应用到数据的版本的协商，再通过基于流水同步的异步数据复制提供了数据强一致性保证和极高的数据写入性能，另外 KVSvr 天然具备数据的 Cache 能力，可以提供高效的读取性能。

KVSvr 一举解决了我们当时迫切需要的无单点故障的容灾能力。除了第 5 版的帐号服务外，很快所有 SDB 底层存储模块和大部分 MySQL 底层存储模块都切换到 KVSvr。随着业务的发展，KVSvr 也不断在进化着，还配合业务需要衍生出了各种定制版本。现在的 KVSvr 仍然作为核心存储，发挥着举足轻重的作用。

平台化

2011.8 深圳举行大运会。微信推出“微信深圳大运志愿者服务中心”服务号，微信用户可以搜索“szdy”将这个服务号加为好友，获取大会相关的资讯。当时后台对“szdy”做了特殊处理，用户搜索时，会随机返回“szdy01”，“szdy02”，…，“szdy10”这 10 个微信号中的 1 个，每个微信号背后都有一个志愿者在服务。

2011.9 “微成都”落户微信平台，微信用户可以搜索“wechengdu”加好友，成都市民还可以在“附近的人”看到这个号，我们在后台给这个帐号做了一些特殊逻辑，可以支持后台自动回复用户发的消息。

这种需求越来越多，我们就开始做一个媒体平台，这个平台后来从微信后台分出，演变成了微信公众平台，独立发展壮大，开始了微信的平台化之路。除微信公众平台外，微信后台的外围还陆续出现了微信支付平台、硬件平台等等一系列平台。（见图 6）

走出国门

微信走出国门的尝试开始于 3.0 版本。从这个版本开始，微信逐步支持繁体、英文等多种语言文字。不过，真正标志性的事情是第一个海外数据中心的投入使用。

1. 海外数据中心

海外数据中心的定位是一个自治的系统，也就是说具备完整的功能，能够不依赖于国内数据中心独立运作。

1）多数据中心架构

系统自治对于无状态的接入层和逻辑层来说很

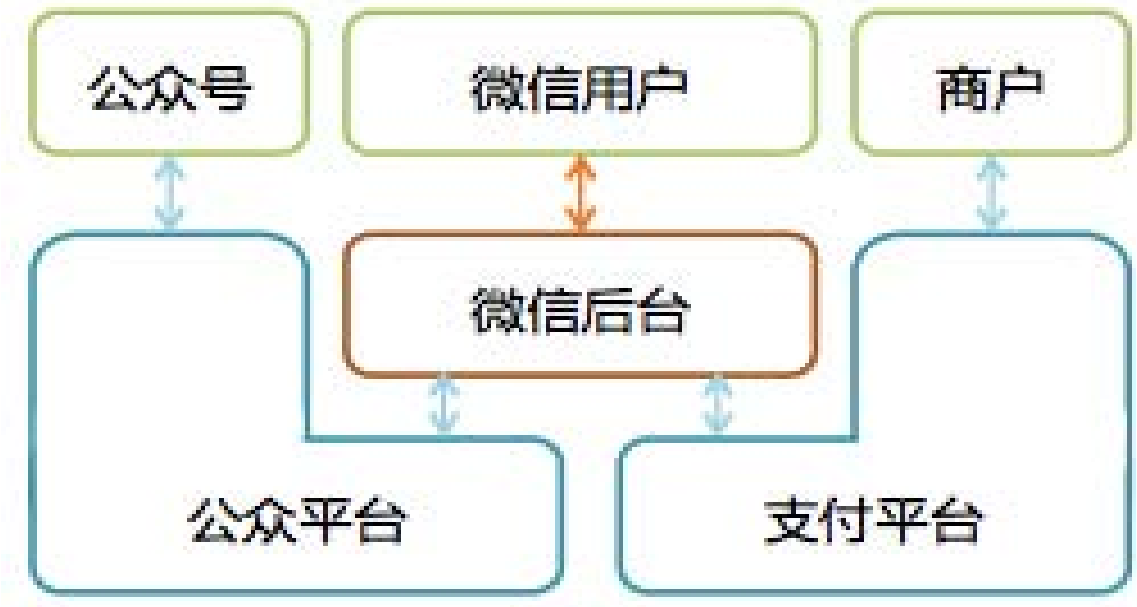


图 6 微信平台

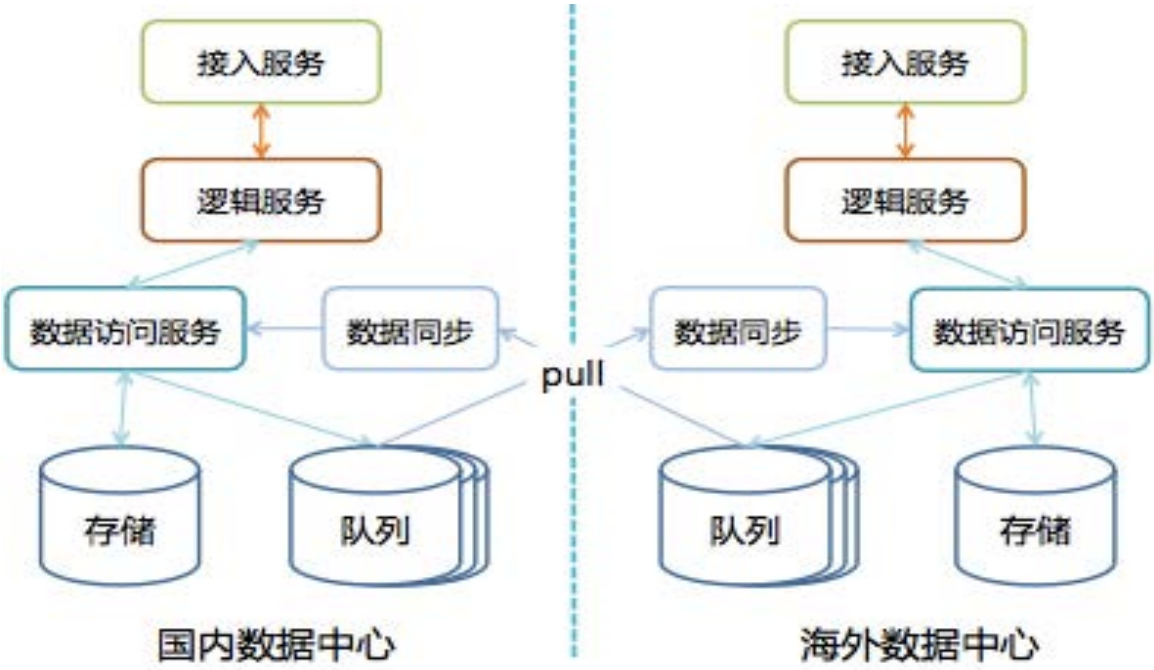


图 7 多数据中心架构

简单，所有服务模块在海外数据中心部署一套就行了。（见图 7）

但是存储层就有很大麻烦了——我们需要确保国内数据中心和海外数据中心能独立运作，但不是两套隔离的系统各自部署，各玩各的，而是一套业务功能可以完全互通的系统。因此我们的任务是需要保证两个数据中心的数据一致性，另外 Master-Master 架构是个必选项，也即两个数据中心都需要可写。

2）Master-Master 存储架构

Master-Master 架构下数据的一致性是个很大的问题。两个数据中心之间是个高延时的网络，意味着在数据中心之间直接使用 Paxos 算法、或直接部署基于 Quorum 的 KVSvr 等看似一劳永逸的方案不适用。

最终我们选择了跟 Yahoo! 的 PNUTS 系统类似的解决方案，需要对用户集合进行切分，国内用户以国内上海数据中心为 Master，所有数据写操作必须回到国内数据中心完成；海外用户以海外数据中心为 Master，写操作只能在海外数据中心进行。从整体存储上看，这是一个 Master-Master 的架构，但细到一个具体用户的数据，则是 Master-Slave 模式，每条数据只能在用户归属的数据中心可写，再异步复制到其他数据中心。（见图 8）

3）数据中心间的数据一致性

这个 Master-Master 架构可以在不同数据中心间实现数据最终一致性。如何保证业务逻辑在这种数据弱一致性保证下不会出现问题？

这个问题可以被分解为 2 个子问题：

- 用户访问自己的数据

用户可以满世界跑，那是否允许用户就近接入数据中心就对业务处理流程有很大影响。如果允许就近接入，同时还要保证数据一致性不影响业务，就意味着要么用户数据的 Master 需要可以动态的改变；要么需要对所有业务逻辑进行仔细梳理，严格区分本数据中心和跨数据

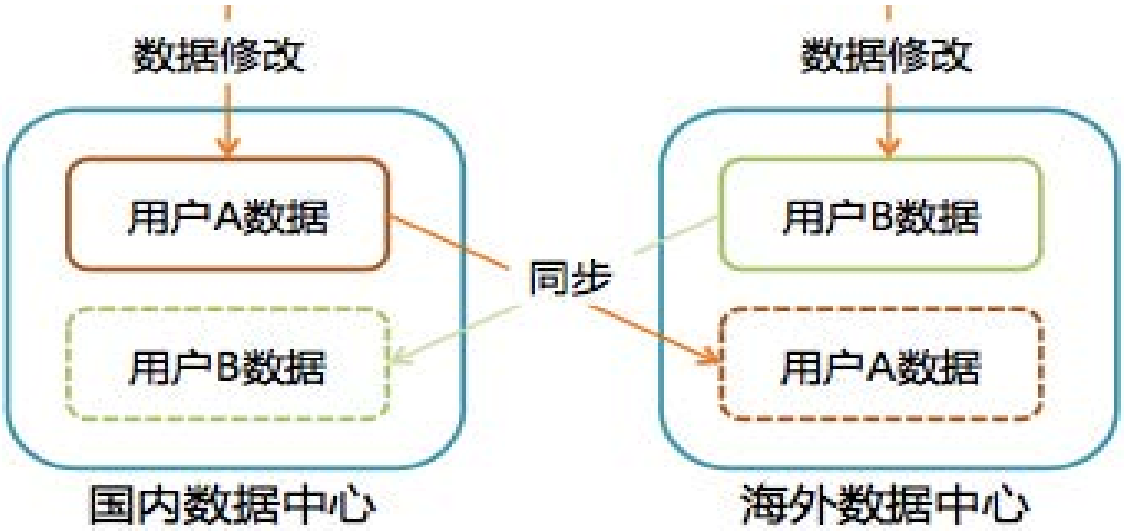


图 8 多数据中心的数据 Master-Master 架构

中心用户的请求，将请求路由到正确的数据中心处理。

考虑到上述问题会带来很高昂的实现和维护的复杂度，我们限制了每个用户只能接入其归属数据中心进行操作。如果用户发生漫游，其漫游到的数据中心会自动引导用户重新连回归属数据中心。

这样用户访问自己数据的一致性问题就迎刃而解了，因为所有操作被限制在归属数据中心内，其数据是有强一致性保证的。此外，还有额外的好处：用户自己的数据（如：消息和联系人等）不需要在数据中心间同步，这就大大降低了对数据同步的带宽需求。

- 用户访问其他用户的数据

由于不同数据中心之间业务需要互通，用户会使用到其他数据中心用户创建的数据。例如，参与其他数据中心用户创建的群聊，查看其他数据中心用户的朋友圈等。

仔细分析后可以发现，大部分场景下对数据一致性要求其实并不高。用户稍迟些才见到自己被加入某个其他数据中心用户建的群、稍迟些才见到某个好友的朋友圈动态更新其实并不会带来什么问题。在这些场景下，业务逻辑直接访问本数据中心的数据。

当然，还是有些场景对数据一致性要求很高。比方说给自己设置微信号，而微信号是需要在整个微信帐号体系里保证唯一的。我们提供了全局唯一的微信号申请服务来解决这一问题，所有数据中心通过这个服务申请微信号。这种需要特殊处置的场景极少，不会带来太大问题。

4）可靠的数据同步

数据中心之间有大量的数据同步，数据是否能够达到最终一致，取决于数据同步是否可靠。为保证数据同步的可靠性，提升同步的可用性，我们又开发一个基于 Quorum 算法的队列组件，这个组件的每一组由 3 机存储服务组成。与一般队列的不同之处在于，这个组件对队列写入操作进行了大幅简化，3 机存储服务不需要相

互通讯，每个机器上的数据都是顺序写，执行写操作时在 3 机能写入成功 2 份即为写入成功；若失败，则换另外一组再试。因此这个队列可以达到极高的可用性和写入性能。每个数据中心将需要同步的数据写入本数据中心的同步队列后，由其他数据中心的数据重放服务将数据拉走并进行重放，达到数据同步的目的。

2. 网络加速

海外数据中心建设周期长，投入大，微信只在香港和加拿大有两个海外数据中心。但世界那么大，即便是这两个数据中心，也还是没法辐射全球，让各个角落的用户都能享受到畅快的服务体验。

通过在海外实际对比测试发现，微信客户端在发消息等一些主要使用场景与主要竞品有不小的差距。为此，我们跟公司的架构平台部、网络平台部和国际业务部等兄弟部门一起合作，围绕海外数据中心，在世界各地精心选址建设了数十个 POP 点（包括信令加速点和图片 CDN 网络）。另外，通过对移动网络的深入分析和研究，我们还对微信的通讯协议做了大幅优化。微信最终在对比测试中赶上并超过了主要的竞品。

精耕细作

1. 三园区容灾

2013.7.22 微信发生了有史以来最大规模的故障，消息收发和朋友圈等服务出现长达 5 个小时的故障，故障期间消息量跌了一半。故障的起因是上海数据中心一个园区的主光纤被挖断，近 2 千台服务器不可用，引发整个上海数据中心（当时国内只有这一个数据中心）的服务瘫痪。

故障时，我们曾尝试把接入到故障园区的用户切走，但收效甚微。虽然数百个在线模块都做了容灾和冗余设计，单个服务模块看起来没有单点故障问题；但整体上看，无数个服务实例散布在数据中心各个机房的 8 千多台服务器内，各服务 RPC 调用复杂，呈网状结构，再加上缺乏系统级的规划和容灾验证，最终导致故障无法主动恢复。在此之前，我们知道单个服务出现单机故障不影响系统，但没人知道 2 千台服务器同时不可用时，整个系统会出现什么不可控的状况。

其实在这个故障发生之前 3 个月，我们已经在着手解决这个问题。当时上海数据中心内网交换机异常，导致微信出现一个出乎意料的故障，在 13 分钟的时间里，微信消息收发几乎完全不可用。在对故障进行分析时，我们发现一个消息系统里一个核心模块三个互备的服务实例都部署在同一机房。该机房的交换机故障导致这个服务整体不可用，进而消息跌零。这个服务模块是最早期（那个时候微信后台规模小，大部分后台服务都部署在一个数据园区里）的核心模块，服务基于 3 机冗余设计，年复一年可靠地运行着，以至于大家都完全忽视了这个问题。

为解决类似问题，三园区容灾应运而生，目标是将上海数据中心的服务均匀部署到 3 个物理上隔离的数据园区，在任意单一园区整体故障时，微信仍能提供无损服务。

1）同时服务

传统的数据中心级灾备方案是“两地三中心”，即同城有两个互备的数据中心，异地再建设一个灾备中心，这三个数据中心平时很可能只有一个在提供在线服务，故障时再将业务流量切换到其他数据中心。这里的主要问题是灾备数据中心无实际业务流量，在主数据中心故障时未必能正常切换到灾备中心，并且在平时大量的备份资源不提供服务，也会造成大量的资源浪费。

三园区容灾的核心是三个数据园区同时提供服务，因此即便某个园区整体故障，那另外两个园区的业务流量也只会各增加50%。反过来说，只需让每个园区的服务器资源跑在容量上限的2/3，保留1/3的容量即可提供无损的容灾能力，而传统“两地三中心”则有多得多的服务器资源被闲置。此外，在平时三个园区同时对外服务，因此我们在故障时，需要解决的问题是“怎样把业务流量切到其他数据园区？”，而不是“能不能把业务流量切到其他数据园区？”，前者显然是更容易解决的一个问题。

2）数据强一致

三园区容灾的关键是存储模块需要把数据均匀分布在3个数据园区，同一份数据要在不同园区有2个以上的一致的副本，这样才能保证任意单一园区出灾后，可以不中断地提供无损服务。由于后台大部分存储模块都使用KVSvr，这样解决方案也相对简单高效——将KVSvr的每1组机器都均匀部署在3个园区里。

3）故障时自动切换

三园区容灾的另一个难点是对故障服务的自动屏蔽和自动切换。即要让业务逻辑服务模块能准确识别出某些下游服务实例已经无法访问，然后迅速自动切到其他服务实例，避免被拖死。我们希望每个业务逻辑服务可以在不借助外部辅助信息（如建设中心节点，由中心节点下发各个业务逻辑服务的健康状态）的情况下，能自行决策迅速屏蔽掉有问题的服务实例，自动把业务流量分散切到其他服务实例上。另外，我们还建设了一套手工操作的全局屏蔽系统，可以在大型网络故障时，由人工介入屏蔽掉某个园区所有的机器，迅速将业务流量分散到其他两个数据园区。

4）容灾效果检验

三园区容灾是否能正常发挥作用还需要进行实际的检验，我们在上海数据中心和海外的香港数据中心完成三园区建设后，进行了数次实战演习，屏蔽单一园区上千台服务，检验容灾效果是否符合预期。特别地，为了避免随着时间的推移某个核心服务模块因为某次更新就不再支持三园区容灾了，我们还搭建了一套容灾拨测系统，每天对所有服务模块选取某个园区的服务主动屏蔽掉，自动检查服务整体失败量是否发生变化，实现对三园区容灾效果的持续检验。

2. 性能优化

之前我们在业务迅速发展之时，优先支撑业务功能快速迭代，性能问题无暇兼顾，比较粗放的贯彻了“先扛住再优化”的海量之道。2014年开始大幅缩减运营成本，性能优化就被提上了日程。

我们基本上对大部分服务模块的设计和实现都进行了重新review，并进行了有针对性的优化，这还是可以节约出不少机器资源的。但更有效的优化措施是对基础设施的优化，具体的说是对Svrkit框架的优化。Svrkit框架被广泛应用到几乎所有服务模块，如果框架层面能把机器资源使用到极致，那肯定是事半功倍的。

结果还真的可以，我们在基础设施里加入了对协程的支持，重点是这个协程组件可以不破坏原来的业务逻辑代码结构，让我们原有代码中使用同步RPC调用的代码不做任何修改，就可以直接通过协程异步化。Svrkit框架直接集成了这个协程组件，然后美好的事情发生了，原来单实例最多提供上百并发请求处理能力的服务，在重编上线后，转眼间就能提供上千并发请求处理能力。Svrkit框架的底层实现在这一时期也做了全新的实现，服务的处理能力大幅提高。

3. 防雪崩

我们一直以来都不太担心某个服务实例出现故障，导致这个实例完全无法提供服务的问题，这个在后台服务的容灾体系里可以被处理得很好。最担心的是雪崩：某个服务因为某些原因出现过载，导致请求处理时间被大大拉长。于是服务吞吐量下降，大量请求积压在服务的请求队列太长时间了，导致访问这个服务的上游服务出现超时。更倒霉的是上游服务还经常会重试，然后这个过载的服务仅有的一点处理能力都在做无用功（即处理完毕返回结果时，调用端都已超时放弃），终于这个过载的服务彻底雪崩了。最糟糕的情况是上游服务每个请求都耗时那么久，雪崩顺着 RPC 调用链一级级往上传播，最终单个服务模块的过载会引发大批服务模块的雪崩。

我们在一番勒紧裤腰带节省机器资源、消灭低负载机器后，所有机器的负载都上来了，服务过载变得经常发生了。解决这一问题的有力武器是 Svrkit 框架里的具有 QoS 保障的 FastReject 机制，可以快速拒绝掉超过服务自身处理能力的请求，即使在过载时，也能稳定地提供有效输出。

4. 安全加固

近年，互联网安全事件时有发生，各种拖库层出不穷。为保护用户的隐私数据，我们建设了一套数据保护系统——全程票据系统。其核心方案是，用户登录后，后台会下发一个票据给客户端，客户端每次请求带上票据，请求在后台服务的整个处理链条中，所有对核心数据服务的访问，都会被校验票据是否合法，非法请求会被拒绝，从而保障用户隐私数据只能用户通过自己的客户端发起操作来访问。

新的挑战

1. 资源调度系统

微信后台有成千的服务模块，部署在全球数以万计的服务器上，一直依靠人工管理。此外，微信后台主要是提供实时在线服务，每天的服务器资源占用在业务高峰和低谷时相差很大，在业务低谷时计算资源被白白浪费；另一方面，很多离线的大数据计算却受制于计算资源不足，难以高效完成。

我们正在实验和部署的资源调度系统（Yard）可以把机器资源的分配和服务的部署自动化、把离线任务的调度自动化，实现了资源的优化配置，在业务对服务资源的需求有变化时，能更及时、更弹性地自动实现服务的重新配置与部署。

2. 高可用存储

基于 Quorum 算法的 KVSvr 已经实现了强一致性、高可用且高性能的 Key-Value/Key-Table 存储。最近，微信后台又诞生了基于 Paxos 算法的另一套存储系统，首先落地的是 PhxSQL，一个支持完整 MySQL 功能，又同时具备强一致性、高可用和高性能的 SQL 存储。

推荐系统和搜索引擎的关系

作者　陈运文

陈运文，博士，达观数据 CEO；中国知名大数据技术专家，国际计算机学会（ACM）会员，中国计算机学会（CCF）高级会员，复旦大学计算机博士和杰出毕业生；在国际顶级学术期刊和会议上发表多篇 SCI 论文，多次参加 ACM 国际数据挖掘竞赛并获得冠军荣誉；曾担任盛大文学首席数据官（CDO），腾讯文学高级总监、数据中心负责人，百度核心技术研发工程师，在大数据挖掘、用户个性化建模、文本信息处理、推荐和搜索技术等方面有丰富的研发和管理经验。

从信息获取的角度来看，搜索和推荐是用户获取信息的两种主要手段。无论在互联网上，还是在线下的场景里，搜索和推荐这两种方式都大量并存，那么推荐系统和搜索引擎这两个系统到底有什么关系？区别和相似的地方有哪些？本文作者有幸同时具有搜索引擎和推荐系统一线的技术产品开发经验，结合自己的实践经验来为大家阐述两者之间的关系、分享自己的体会。

主动或被动：搜索引擎和推荐系统的选择

获取信息是人类认知世界、生存发展的刚需，搜索就是最明确的一种方式，其体现的动作就是“出去找”，找食物、找地点等，到了互联网时代，搜索引擎（Search Engine）就是满足找信息这个需求的最好工具，你输入想要找的内容（即在搜索框里输入查询词，或称为

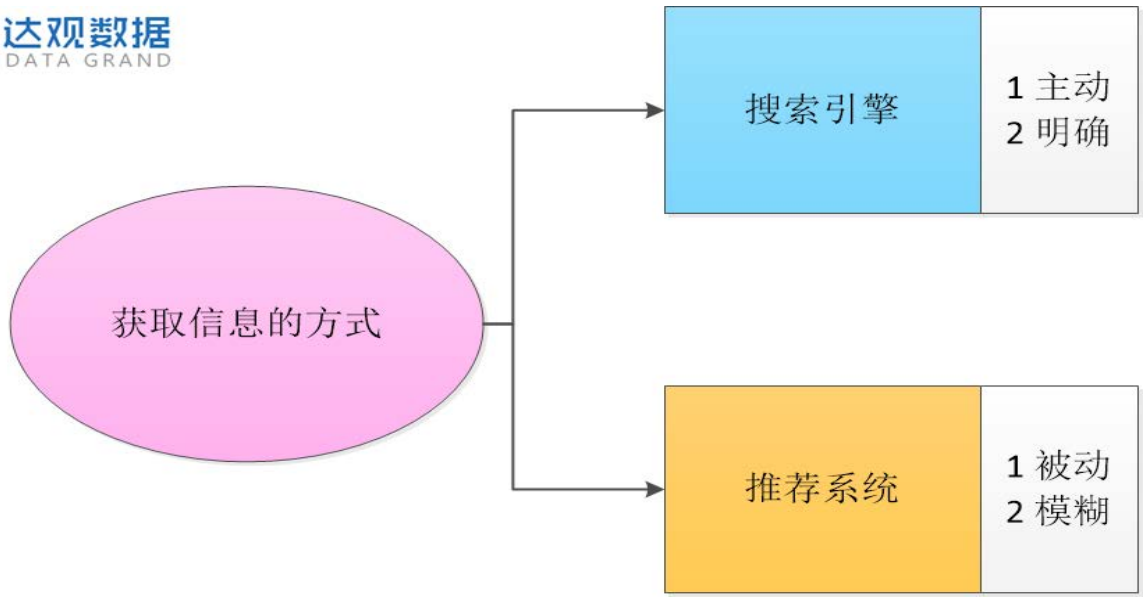


图 1 搜索引擎和推荐系统是获取信息的两种不同方式

Query），搜索引擎快速的给你最好的结果，这样的刚需催生了 Google、百度这样的互联网巨头。

但是获取信息的方式除了搜索外，还有另一类，称为推荐系统（Recommendation System，简称 Recsys），推荐也是伴随人类发展而生的一种基本技能，你一定遇到这样的场景，初来乍到一个地方，会找当地的朋友打听“嗨，请推荐下附近有啥好吃好玩的地方吧！”——知识、信息等通过推荐来传播，这也是一种获取信息的方式。

搜索和推荐的区别如图 1 所示，搜索是一个非常主动的行为，并且用户的需求十分明确，在搜索引擎提供的结果里，用户也能通过浏览和点击来明确的判断是否满足了用户需求。然而，推荐系统接受信息是被动的，需求也都是模糊而不明确的。以“逛”商场为例，在用户进入商场的时候，如果需求不明确，这个时候需要推荐系统，来告诉用户有哪些优质的商品、哪些合适的内容等，但如果用户已经非常明确当下需要购买哪个品牌、什么型号的商品时，直接去找对应的店铺就行，这时就是搜索了。（见图 2）

很多互联网产品都需要同时满足用户这两种需求，例如对提供音乐、新闻、或者电商服务的网站，必然要提供搜索功能，当用户想找某首歌或某样商品的时候，输入名字就能搜到；与此同时，也同时要提供推荐功能，当用户就是想来听好听的歌，或者打发时间看看新闻，但并不明确一定要听哪首的时候，给予足够好的推荐，提升用户体验。

个性化程度的高低

除了主被动外，另一个有趣的区别是个性化程度的高低之分。搜索引擎虽然也可以有一定程度的个性化，但是整体上个性化运作的空间是比较小的。因为当需求非常明确时，找到结果的好坏通常没有太多个性化的差异。例如搜“天气”，搜索引擎可以将用户所在地区的信息作补足，给出当地天气的结果，但是个性化补足后给出的结果也是明确的了。

但是推荐系统在个性化方面的运作空间要大得多，以“推荐好看的电影”为例，一百个用户有一百种口味，并没有一个“标准”的答案，推荐系统可以根据每位用户历史上的观看行

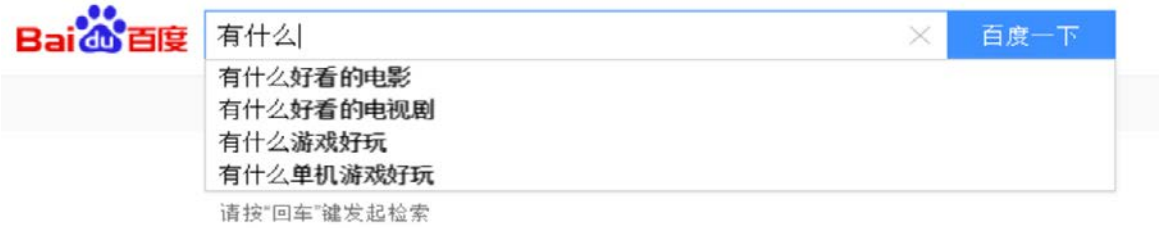


图 2 从搜索词中可以看出，用户有大量个性化推荐的需求

为、评分记录等生成一个对当前用户最有价值的结果，这也是推荐系统有独特魅力的地方。虽然推荐的种类有很多（例如相关推荐、个性化推荐等），但是个性化对于推荐系统是如此重要，以至于在很多时候大家干脆就把推荐系统称为“个性化推荐”甚至“智能推荐”了。

快速满足还是持续服务

开发过搜索引擎的朋友都知道，评价搜索结果质量的一个重要考量指标是要帮用户尽快的找到需要的结果并点击离开。在设计搜索排序算法里，需要想尽办法让最好的结果排在最前面，往往搜索引擎的前三条结果聚集了绝大多数的用户点击。简单来说，“好”的搜索算法是需要让用户获取信息的效率更高、停留时间更短。

但是推荐恰恰相反，推荐算法和被推荐的内容（例如商品、新闻等）往往是紧密结合在一起的，用户获取推荐结果的过程可以是持续的、长期的，衡量推荐系统是否足够好，往往要依据是否能让用户停留更多的时间（例如多购买几样商品、多阅读几篇新闻等），对用户兴趣的挖掘越深入，越“懂”用户，那么推荐的成功率越高，用户也越乐意留在产品里。

所以对大量的内容型应用来说，打造一个优秀的推荐系统是提升业绩所不得不重视的手段。

推荐系统满足难以文字表述的需求

目前主流的搜索引擎仍然是以文字构成查询词（Query），这是因为文字是人们描述需求最简洁、直接的方式，搜索引擎抓取和索引的绝大部分内容也是以文字方式组织的。

因为这个因素，我们统计发现用户输入的搜索查询词也大都是比较短小的，查询词中包含 5 个或 5 个以内元素（或称 Term）的占总查询量的 98% 以上（例如：Query“达观数据地址”，包含两个元素“达观数据”和“地址”）。

但另一方面，用户存在着大量的需求是比较难用精炼的文字来组织的，例如想查找“离我比较近的且价格 100 元以内的川菜馆”、“和我正在看的这条裙子同款式的但是价格更优惠的其他裙子”等需求。

一方面几乎没有用户愿意输入这么多字来找结果（用户天然都是愿意偷懒的），另一方面搜索引擎对语义的理解目前还无法做到足够深入；所以在满足这些需求的时候，通过推荐系统设置的功能（例如页面上设置的“相关推荐”、“猜你喜欢”等模块），加上与用户的交互（例如筛选、排序、点击等），不断积累和挖掘用户偏好，可以将这些难以用文字表达的需求良好的满足起来。

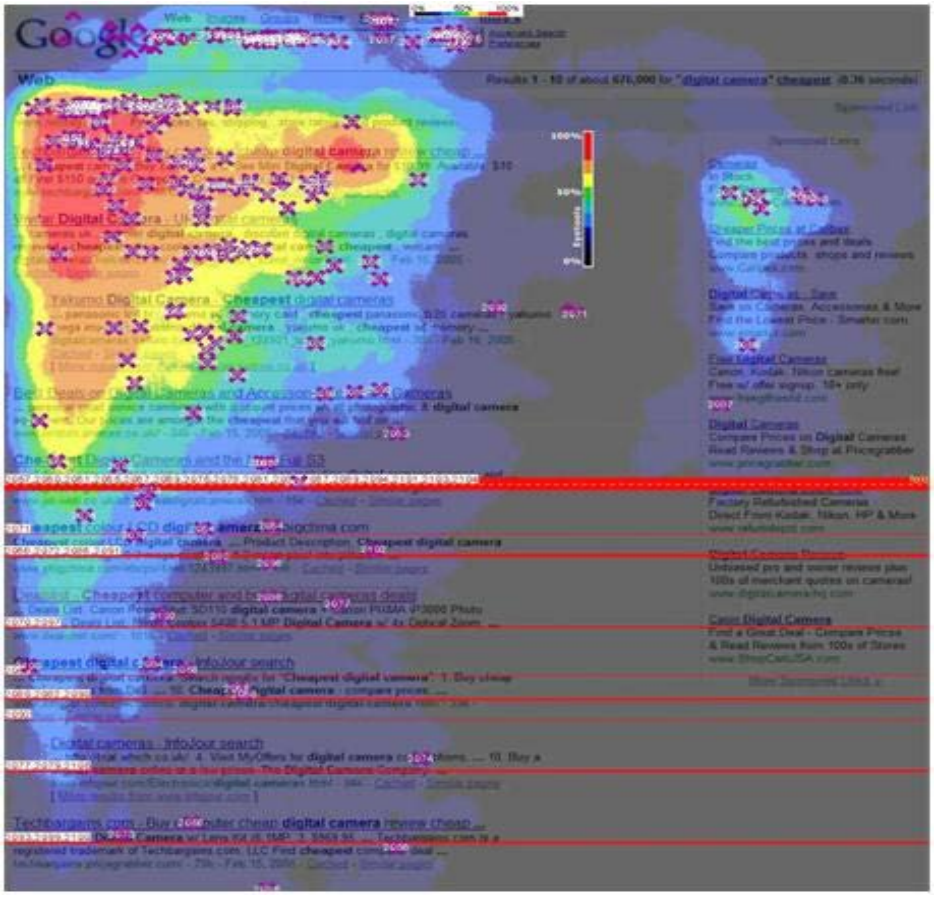


图 3 搜索引擎充分体现的马太效应：头部内容吸引了绝大部分点击

形象的来说，推荐引擎又被人们称为是无声的搜索，意思是用户虽然不用主动输入查询词来搜索，但是推荐引擎通过分析用户历史的行为、当前的上下文场景，自动来生成复杂的查询条件，进而给出计算并推荐的结果。

马太效应和长尾理论

马太效应（matthew effect） 是指强者愈强、弱者愈弱的现象，在互联网中引申为热门的产品受到更多的关注，冷门内容则愈发的会被遗忘的现象。马太效应取名自圣经《新约 • 马太福音》的一则寓言： “凡有的，还要加倍给他叫他多余；没有的，连他所有的也要夺过来。”

搜索引擎就非常充分的体现了马太效应——如下面的 Google 点击热图，越红的部分表示点击多和热，越偏紫色的部分表示点击少而冷，绝大部分用户的点击都集中在顶部少量的结果上，下面的结果以及翻页后的结果获得的关注非常少。这也解释了 Google 和百度的广告为什么这么赚钱，企业客户为什么要花大力气做 SEM 或 SEO 来提升排名——因为只有排到搜索结果的前面才有机会。（见图 3）

有意思的是，与“马太效应”相对应，还有一个非常有影响力的理论称为“长尾理论”。

长尾理论（Long Tail Effect）是“连线”杂志主编克里斯 • 安德森（Chris Anderson）在 2004 年 10 月的“长尾”（Long Tail）一文中最早提出的，长尾实际上是统计学中幂率（Power Laws）和帕累托分布特征（Pareto Distribution）的拓展和口语化表达，用来描述热门和冷门物品的分布情况。Chris Anderson 通过观察数据发现，在互联网时代由于网络技术能以很低的成本让人们去获得更多的信息和选择，在很多网站内有越来越多的原先被“遗忘”的非最热门的事物重新被人们关注起来。事实上，每一个人的品味和偏好都并

图 4 推荐系统和长尾理论

非和主流人群完全一致，Chris 指出：当我们发现得越多，我们就越能体会到我们需要更多的选择。如果说搜索引擎体现着马太效应的话，那么长尾理论则阐述了推荐系统发挥的价值。（见图 4）

一个实际的例子就是亚马逊（Amazon）网络书店和传统大型书店的数据对比。市场上出版发行的图书种类超过了数百万，但是其中大部分图书是无法在传统大型书店上架销售的（实体店铺空间有限），而能放在书店显著位置（例如畅销书 Best Seller 货架）上的更是凤毛麟角，因此传统书店的经营模式多以畅销书为中心。但是亚马逊等网络书店的发展为长尾书籍提供了无限广阔的空间，用户浏览、采购这些长尾书籍比传统书店方便得多，于是互联网时代销售成千上万的小众图书，哪怕一次仅卖一两本，但是因为这些图书的种类比热门书籍要多得多，就像长长的尾巴那样，这些图书的销量积累起来甚至超过那些畅销书。正如亚马逊的史蒂夫·凯赛尔所说：“如果我有 10 万种书，哪怕一次仅卖掉一本，10 年后加起来它们的销售就会超过最新出版的《哈利·波特》！”

长尾理论作为一种新的经济模式，被成功的应用于网络经济领域。而对长尾资源的盘活和利用，恰恰是推荐系统所擅长的，因为用户对长尾内容通常是陌生的，无法主动搜索，唯有通过推荐的方式，引起用户的注意，发掘出用户的兴趣，帮助用户做出最终的选择。

盘活长尾内容对企业来说也是非常关键的，营造一个内容丰富、百花齐放的生态，能保障企业健康的生态。试想一下，一个企业如果只依赖 0.1% 的“爆款”商品或内容来吸引人气，那么随着时间推移这些爆款不再受欢迎，而新的爆款又没有及时补位，那么企业的业绩必然会有巨大波动。

只依赖最热门内容的另一个不易察觉的危险是潜在用户的流失：因为只依赖爆款虽然能吸引一批用户（简称 A 类用户），但同时也悄悄排斥了对这些热门内容并不感冒的用户（简称 B 类用户），按照长尾理论，B 类用户的数量并不少，并且随时间推移 A 类用户会逐步转变为 B 类用户（因为人们都是喜新厌旧的），所以依靠推荐系统来充分满足用户个性化、差异化的需求，让长尾内容在合适的时机来曝光，维护企业健康的生态，才能让企业的运转更稳定，波动更小。

评价方法的异同

搜索引擎通常基于 Cranfield 评价体系，并基于信息检索中常用的评价指标，例如 nDCG（英文全称为 normalized Discounted Cumulative Gain）、Precision-Recall（或其组合方式 F1）、P@N 等方法，具体可参见之前发表于 InfoQ 的文章《怎样量化评价搜索引擎的结果质量 陈运文》。整体上看，评价的着眼点在于将优质结果尽可能排到搜索结果的最前面，前 10 条结果（对应搜索结果的第一页）几乎涵盖了搜索引擎评估的主要内容。让用户以最少的点击次数、最快的速度找到内容是评价的核心。

推荐系统的评价面要宽泛的多，往往推荐结果的数量要多很多，出现的位置、场景也非常复杂，从量化角度来看，当应用于 Top-N 结果推荐时，MAP（Mean Average Precison）或 CTR（Click Through Rate，计算广告中常用）是普遍的计量方法；当用于评分预测问题时，RMSE（Root Mean Squared Error）或 MAE（Mean Absolute Error）是常见量化方法。

由于推荐系统和实际业务绑定更为紧密，从业务角度也有很多侧面评价方法，根据不同的业务形态，有不同的方法，例如带来的增量点击，推荐成功数，成交转化提升量，用户延长的停留时间等指标。

搜索和推荐的相互交融

搜索和推荐虽然有很多差异，但两者都是大数据技术的应用分支，存在着大量的交叠。近年来，搜索引擎逐步融合了推荐系统的结果，例如右侧的“相关推荐”、底部的“相关搜索词”等，都使用了推荐系统的产品思路和运算方法（如右图红圈区域）。

在另一些平台型电商网站中，由于结果数量巨大，且相关性并没有明显差异，因而对搜索结果的个性化排序有一定的运作空间，这里融合运用的个性化推荐技术也对促进成交有良好的帮助。

搜索引擎中融合的推荐系统元素

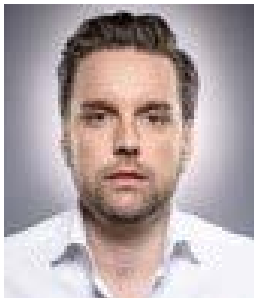
推荐系统也大量运用了搜索引擎的技术，搜索引擎解决运算性能的一个重要的数据结构是倒排索引技术（Inverted Index），而在推荐系统中，一类重要算法是基于内容的推荐（Content-based Recommendation），这其中大量运用了倒排索引、查询、结果归并等方法。另外点击反馈（Click Feedback）算法等也都在两者中大量运用以提升效果。

本文总结

作为大数据应用的两大类应用，搜索引擎和推荐系统既相互伴随和影响，又满足不同的产品需求。在作为互联网产品的连接器：连接人、信息、服务之间的桥梁，搜索和推荐有其各自的特点，本文对两者的关系进行了阐述，分析了异同。它们都是数据挖掘技术、信息检索技术、计算统计学等悠久学科的智慧结晶，也关联到认知科学、预测理论、营销学等相关学科，感兴趣的读者们可以延伸到这些相关学科里做更深入的了解。

关于云迁移的经验总结

作者 Alois Reitbauer 译者 邱广



Alois Reitbauer 是 Ruxit 公司的首席布道师。在他职业生涯的绝大部分时间里，他都在进行监控工具构建以及应用性能微调。他常以演讲嘉宾的身份出席会议，同时他还是一名博客作者、作家和寿司狂人，Alois 近期的工作地点是 Linz、Boston 和 San Francisco。

在近期举办的 Dynatrace Perform 大会上，我与不同类型的技术公司就云迁移这个话题进行了广泛的交流。所接触的专家从 SaaS 公司到电子商务公司和云服务提供商。公司规模大小不一，既有初创型规模的，也有大至巴西最大的电子商务公司之一，以及建立 SaaS 业务的公司。尽管这些公司之间颇为不同，但他们对云迁移项目的关键要点有着高度一致的看法。本文涵盖了前 5 大共识。

总会发生迫使你转向云的事情

讨论小组都认同，实施云迁移不是平白无故做出的决定。所有的与会者都是因为遇上了不可抗拒的事件，才被迫转向云的。对 B2W 来说，这个不可抗拒的事件就是，在一次严重的生产事故后，他们决定朝微服务的方向去重构整个环境。当基于交易量来动态扩展独立组件成为云战略的一个关键部分时，迁移到云端也就成为了他们战略的关键组成。对行业领先的 BAR

考试服务提供商 BARBRI 而言，驱使他们决定向云迁移的关键动力来源于数据中心的重建需求。

向云迁移的关键约束在于成本

当问及迁移到云的动机时，第一个想到的就是降低成本。所有的与会者都同意，降低运行成本是云迁移的核心驱动力。对更小的初创公司来说，IaaS 的弹性模型使它们能够对所需的基础设施进行及时的投资。对成熟的公司而言，走向云迁移的关键事件是对重建数据中心的再投资。一旦你面临一笔大的投资，就会对战略问题三思而后行。“对我来说，当下在数据中心上的花费是我的首要成本。”BARBRI 的 IT 主管 Mark Kaplan 说。需要重点指出的是，低成本的收益需要一段时间才能显现出来。在过渡期，公司同时运行云和非云设施。“仍然需要为接下来的两年时间做好成本节约计划，以保障项目成功”， Kaplan 指出。

APM 工具的相关工作应在 CIO 议程上

实施云迁移同时带来了一件有意思的事情，APM 工具的相关工作不能缺席 CIO 议程。通过云服务的弹性和灵活性，公司能够更直接和快速地进行成本优化。BARBRI 的 Greg Birdwell 指出，“我们使用 APM 工具不光是为了监控基础设施的健康度。如果我发现有服务消耗的 CPU 或内存资源较其它服务少很多，我就能切换到更廉价的实例上去。这能立即为公司节约成本。”BARBRI 的 Mark Kaplan 说，他们使用 Ruxit 来精确地缕清环境的依赖关系和资源需求，这些都是完成迁移的基础。全球各地的 CIO 们盯着 APM 工具，根据监控数据计算成本收益，这样的景象也许我们还要再过一段时间才能看到，但事情的发展正在朝这个方向演进。

敏捷压倒成本收益

从讨论小组那获得的一个令人非常惊讶的事实是，尽管降低成本是云迁移的一大核心动力，但所有的与会者都认同他们现在花在云基础设施上的钱更多了。“如果运行在一个传统的数据中心上，对我们会更便宜。” Stilnest 的 Michael Aigner 说。“但我们还是把宝全部押在云上，因为我们获得了比节省一小笔钱更宝贵的东西：敏捷。”所有的小组成员都认为云服务通过支持新功能的敏捷开发模式，从而帮助产品获得更短的上市时间，这点是客户们非常欣赏的。

云迁移影响了文化

与会者认为，云迁移不是一个技术或经济游戏规则颠覆者，实际上它广泛地影响到了公司的文化。一旦开始运行在云上，你就已经转向 DevOps 了。“我们所有的基础设施现在都在代码里” Stilnest 的 Michael Aigner 说道。“我期望我们的运维团队能更像开发人员，像开发人员那样工作”，B2W 公司的 Alexander Ramos 说。开发人员持续不断地投入到生产问题的解决中，根据 Ruxit 的 Anita Engleder 的说法，以 DevOps 牵头常常促使开发团队成为应用运维的一部分。Engleder 认为“不管怎样，开发人员都比我更了解他们自己的代码，所以他们应当为监控负责”。B2W 公司的 Alexander 把这点推向了一个极端：“我给开发人员发了寻呼机，他们和其他人一样都随时待命。这改变了很多东西。”

遴选云服务提供商是一大挑战

讨论小组认为，遴选出合适的云服务提供商是一个重大的挑战。云服务提供商的选择依赖于你对基础设施的需求以及云服务提供商自身的灵活度。同时，当评估价格点时，就不仅仅只需考虑基础设施状况了。“如果你想要办件事

情，就会希望电话那端能有人指望得上。可靠可信任的合作伙伴对云迁移的每一步都很关键。”CenturyLink Cloud的Bob Stolzberg说。

成功的迁移建立在严谨规划的基础上

CenturyLink的Bob Stolzberg指出，管理层的赞助支持和端到端的规划对成功交付非常关键。Bob Stolzberg经历过了非常多的迁移项目，他的建议相当清晰：“制定一个执行计划，争取成功完成，但不要追求完美”。Stolzberg说，“你还需要制定回滚策略。如果事情搞砸了，就会想要重回安全节点。”

云迁移不代表迁移到公有云上

对许多公司来说，云迁移意味着迁移到公有云上去，事实上往公有云迁移只是其中的一个选项。尤其是当监管方面的要求非常重要时，公有云就力不从心了。Avocado咨询公司的Romain Bigeard指出，这类情况公司的出路是投奔私有云。他认为“利用私有云基础设施也能够获得云环境带来的许多好处”。对这类公司，关键是构建一个可编程的基础设施，进而提高在软件交付过程中的敏捷度。

做出正确的行动

对会谈参与方做的一个非正式的调查显示了来宾们云迁移战略里头更深层次的内容。大多数决定先从迁移新应用到云端开始。一旦积累到一些经验了，然后开始迁移已有的前置应用到云端。要么一次性把所有的都迁移过去，要么一开始只迁移某些应用服务到云端。

基础设施即服务是迁移最常见的原因

尽管云服务提供商还提供了类似容器即服务（Containers-as-a-service）和PaaS的选项，来宾们仍然更青睐基础设施即服务。此外还有许多额外的云服务可以优化应用交付。

理解依赖关系是最大的挑战，紧随其后的是规模估算

云迁移最大的挑战是理解现有基础设施内的依赖关系，这点得到了广泛的认同。“有时我想知道为什么这两个应用之间存在会话。但找不到说明文档，所以我不得不甩开Ruxit上的所有数据” B2W的Alexander说。排在挑战榜单第二位的是规模估算。不同规模的云实例的价格差异非常明显，因此从节约成本的角度来看，关键在于选用正确的实例类型。传统上，公司为安全起见会购买足够大的机器。而在云上，公司只想买最小可能的基础设施，有需要时再进行扩展。尤其是在变动场景下，目前数据中心服务器的规模常常是过度配置了。正如前述所提到的，基础设施上的花费直接即时，节约成本是关键。

云监控也不尽相同

一旦迁移到云端上了，很多事情就变了。所有这些全新的云服务成为应用的一部分的同时，也意味着需要使用工具进行原生监控和管理应用。挑战榜下一个是云环境的扩展。成本节约背后所有的理论要求只运行所需的，在纸面上也许很好做到，但是如果除了基于资源消耗的扩展，工具对其它方面只提供了非常有限的功能，这时自动扩展环境显得很困难。成本节约最终会是成果显著的，但在一开始你需要把注意力放在确保迁移努力确有成效上面，而不是让其影响了性能和用户体验。等所有事情都正常运行了，才在扩容和缩容这两个方向上进行容量规划，并经常性地这样操作。在一个基于价格模型的消费世界里，时间能造就其中的财务差异。

AWS 迷你书

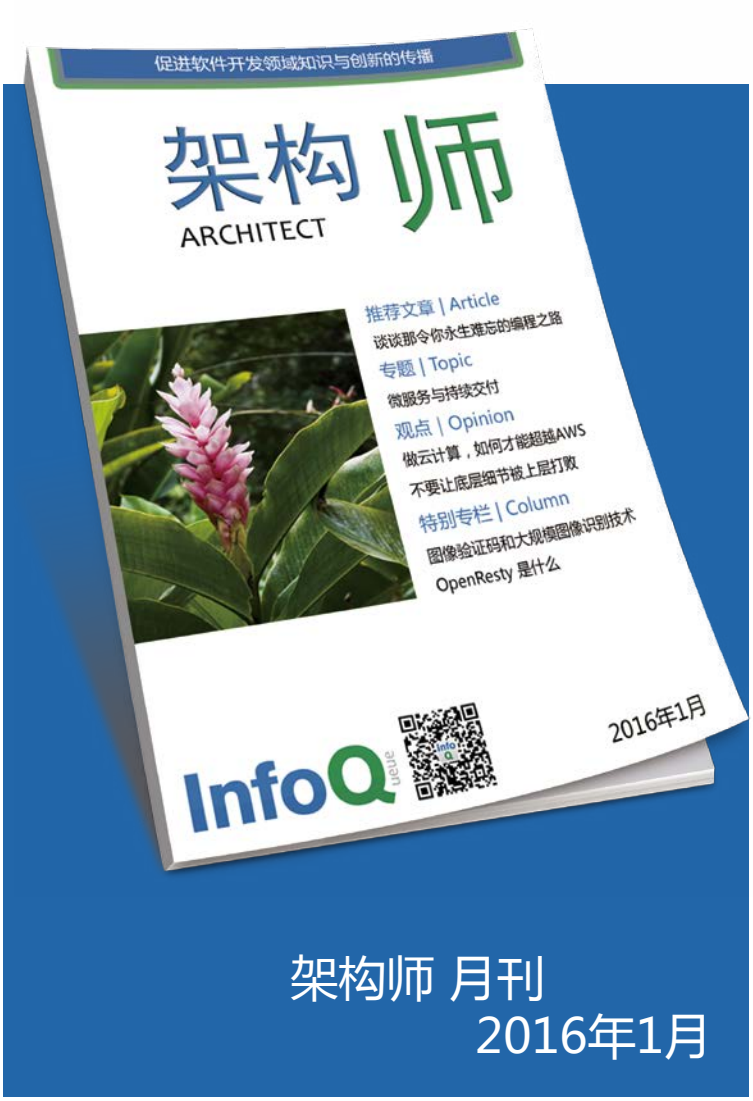
云计算

让数据管理变得更轻松

扫描下载迷你书

亚马逊 aws | InfoQueue

InfoQ 中文站　迷你书



架构师 月刊 2016年1月

本期主要内容：Google确认下一个Android版本将不会使用Oracle的Java　API，转而使用开源的OpenJDK替代，谈谈那令你永生难忘的编程之路，微服务与持续交付，梁胜：做云计算，如何才能超越AWS？　SpeedyCloud研发总监李孟：不要让底层细节被上层打散，图像验证码和大规模图像识别技术，锤子手机发布会提到的 OpenResty 是什么？

解读2015

希望“解读2015”年终技术盘点系列文章，能够给您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。

顶尖技术团队访谈录 第四季

本次的《中国顶尖技术团队访谈录》·第四季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。

10个精选的容器应用案例

《10个精选的容器应用案例》是InfoQ旗下的CNUT（容器技术俱乐部）推出的容器应用白皮书，旨在通过优秀的案例引导国内社区和企业正确使用相关的容器技术，以加速容器技术在国内的普及。