促进软件开发领域知识与创新的传播

架构师 ARCHITECT

推荐文章 | Article

通过大量测试来构建测试系统

实施微服务，需要哪些基础框架

专题 | Topic

重建还是重构

Medium开发团队谈架构设计

热点 | Hot

斑马系统：微信朋友圈广告背后的利器

观点 | Opinion

被误解的MVC和被神化的MVVM

InfoQ

2015年12月

微信斑马系统：朋友圈广告背后的利器

本文介绍我们即将推出的微信斑马系统。

Spark和Hadoop，孰优孰劣

著名大数据专家Bernard　Marr在文章中分析了Spark和Hadoop的异同。

被误解的MVC和被神化的MVVM

MVC与MVVM的深入对比。

实施微服务需要哪些基础框架

微服务架构有哪些技术关注点，需要哪些基础框架或组件来支持微服务架构。

深入了解IAM和访问控制

如果你要想能够游刃有余地使用AWS的各种服务，在安全上的纰漏尽可能地少，首先需要先深入了解 IAM。

重建还是重构

什么让重构如此困难，重建软件的风险是否比重构小，以及持续交付如何配合软件的重建。

架构师 2015年12月刊

本期主编　魏星

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

ArchSummit 全球架构师峰会 2015

[北京站]

2015年12月18日-19日
北京·国际会议中心
www.archsummit.com

精彩日程 聚焦前沿

近百业界精英压轴年度技术大戏

大会介绍

交易量逐年暴涨，双十一电商架构今年有何不同？经历了多起安全事件之后，互联网安全架构有何警示？云服务遍地开花，云架构如何能承受爆发式的增长？智能硬件产业方兴未艾，智能设备设计有何玄机？互联网银行蓄势待发，互联网架构跟银行业务又是如何结合？答案尽在ArchSummit北京2015全球架构师峰会。ArchSummit聚集了互联网行业各领域的一线知名架构师及高级技术管理者等，期待与您一起交流承载繁华业务的架构智慧。

主题演讲

袁泳
Uber软件工程师
《再造轮子之道》
&《Uber的流处理系统及实践》

王天
Twitter Senior Staff Engineer
《百花齐放，锄其九九——Twitter的技术坎坷之路》&《Twitter实时搜索引擎发展历程》

部分演讲嘉宾（排名不分先后）

张立刚
1号店架构部技术总监
系统稳定与优化-从SOA规划与治理开始

谦益（赵成）
蘑菇街运维经理
蘑菇街运维体系架构及双11关键技术分享

季虎
苏宁IT总部安全研发技术总监
简单的不像技术活——风险检测中的时间窗口计算

程实（时勤）
阿里巴巴技术专家
让数据川流不息——阿里云数据传输服务揭秘

毕鹏
点融网金融系统技术总监
高可用网贷系统技术架构的演进

唐巧
小猿搜题产品技术负责人
涅槃重生：技术到管理的跨越转型

范超霏
出门问问Ticwear系统架构师
智能手表软件体系构建及软硬件结合

张勇
360手机助手安卓技术负责人
分拆：DroidPlugin的实现原理及其应用

垂询电话：010-89880682
QQ咨询：2332883546
E-mail:arch@cn.infoq.com
更所精彩内容，请持续关注archsummit.com
了解最新日程请扫描二维码

Brought by Geekbang. 极客邦科技 InfoQ

冯是聪 北京明略软件系统有限公司联合创始人兼CTO

大规模弹性平台架构设计

在谈大规模弹性架构设计之前，先聊聊两个真实的场景：

场景一：春运火车票抢票大战。春节一直是中国最重要的传统节日，每个中国人都期盼着在这个阖家欢乐的时刻，能够相聚在一起共享天伦之乐，于是就有了中国特色的春运，外媒称之为“人类历史上规模最大的周期性迁徙”。作为春运火车票售票系统，因为峰值访问量过于庞大（2014 年尖峰日 PV 达到 144 亿次），2014 年以前 12306 系统动不动就瘫痪，12306 也因此一直受到诟病，相信大多数春运期间购买过火车票的人都记忆犹新。2015 年以后，尽管尖峰日 PV 达到令人恐怖的 297 亿次，但是由于系统经过了升级，系统瘫痪的情况大大减少。

场景二：双十一购物狂欢节。刚刚落幕的 2015 年双十一购物狂欢节，阿里巴巴集团创下了 912 亿的骄人成绩，支付宝支撑住了每秒 8.59 万笔的交易量。

上面两个案例中，如果我们为了保证系统的高可用性，按着每天访问的峰值作为标准来设计系统，绝大数情况下访问量较低时，无论是硬件资源还是网络带宽等资源都会造成巨大的浪费。那么如何从技术上解决这个问题呢？答案是大规模弹性架构设计。

大规模弹性架构的主要设计目标是系统的高可用性和高扩展性：根据系统的实际需要，系统自动弹性分配资源。在需求量大，比如峰值的时候，系统自动调配足够的资源；在访问量小，系统自动释放出部分的资源，从而实现资源的按需分配。

那么大规模弹性架构的基本工作原理是什么？其主要是利用高性能虚拟化技术，实现了计算、

存储和网络资源的统一调度和弹性分配。限于篇幅，这里仅举两个具体的技术：

1. 实现动态部署。为了应对突发峰值访问量或者可能的网络攻击，需要在应用服务器内部部署一些监控程序，由主控程序判断当前整个集群的负载情况，自动增加或者减少服务节点，并且自动部署应用程序，从而从容应对突增的业务流量。
2. 实现故障自动恢复。当一台物理机损坏时，系统需要自动监测到硬件故障，并且在第一时间内，把云服务器迁移到新的宿主机上，同时硬盘数据需要保持最后一刻的状态。数据安全是第一位，因为数据始终面临误删文件、病毒破坏、程序写错、硬件损坏等种种可能的风险。在弹性计算平台上，需要利用快照功能自动完成数据的备份。

最后，简单聊聊大规模弹性平台架构设计变化的趋势。

第一，弹性平台架构平台的设计将会愈发智能。系统将会自动收集愈来愈多的用户访问数据，跟据历史访问规律，自动地预测系统峰值的变化趋势，峰值的预估将愈加精准。简单设想一下，如果系统已有的全部资源都不能满足峰值的需求，弹性设计已经没有意义了，这时候需要提前做准预估，并采取相应的措施，并假设如果超过峰值，如何应对。

第二，以容器技术为代表的轻虚拟化技术的迅猛发展，将会进一步促进大规模弹性平台架构设计的进步。

第三，弹性计算平台将需要考虑更加安全的容灾方案。目前常见的单个物理位置的系统，始终会面临诸如地震、火灾等不可抗力的威胁，所以需要有更加安全的容灾方案。比如常见的有跨机房异地容灾，目前 Google 的核心数据已经实现了跨越 5 大洲的异地备份容灾。

微信斑马系统：朋友圈广告背后的利器

作者 张重阳

简介

随着移动互联网迅速发展，大数据技术为企业带来了前所未有的发展机遇，然而中小企业和传统行业由于其数据量缺乏且单一，技术投入不足的劣势，面对大数据技术发展带来的红利只能望洋兴叹。

本文介绍我们即将推出的微信斑马系统，该系统旨在为中小企业和传统行业提供基于微信大数据分析技术的受众分析，精准推广，激活留存和商业智能决策的全套解决方案。

功能与模块

图 1 是斑马系统的功能与模块，最底层是数据层，我们先看左边的微信原始数据，这里主要是零散的网络日志包括个人填写的信息，网络行为，社交关系等，我们对其进行提取和分析得到用户的属性和各种特征，比如，居住地，年龄，是否结婚，正在旅游等。这样一个个的用户在我们系统里就有了各自丰富的特征，这个过程叫用户画像（Persona，图 2）。

这些特征的来源，覆盖人数，更新周期各不相同，随着业务的发展和系统的不断更新，用户特征将不断完善，我们开发了一套用户特征管理查询系统，用于将用户的所有特征整合在一个存储体系下实现新特征快速加入，周期更新和快速查找。

用户特征管理查询系统的外层是面向商家和运

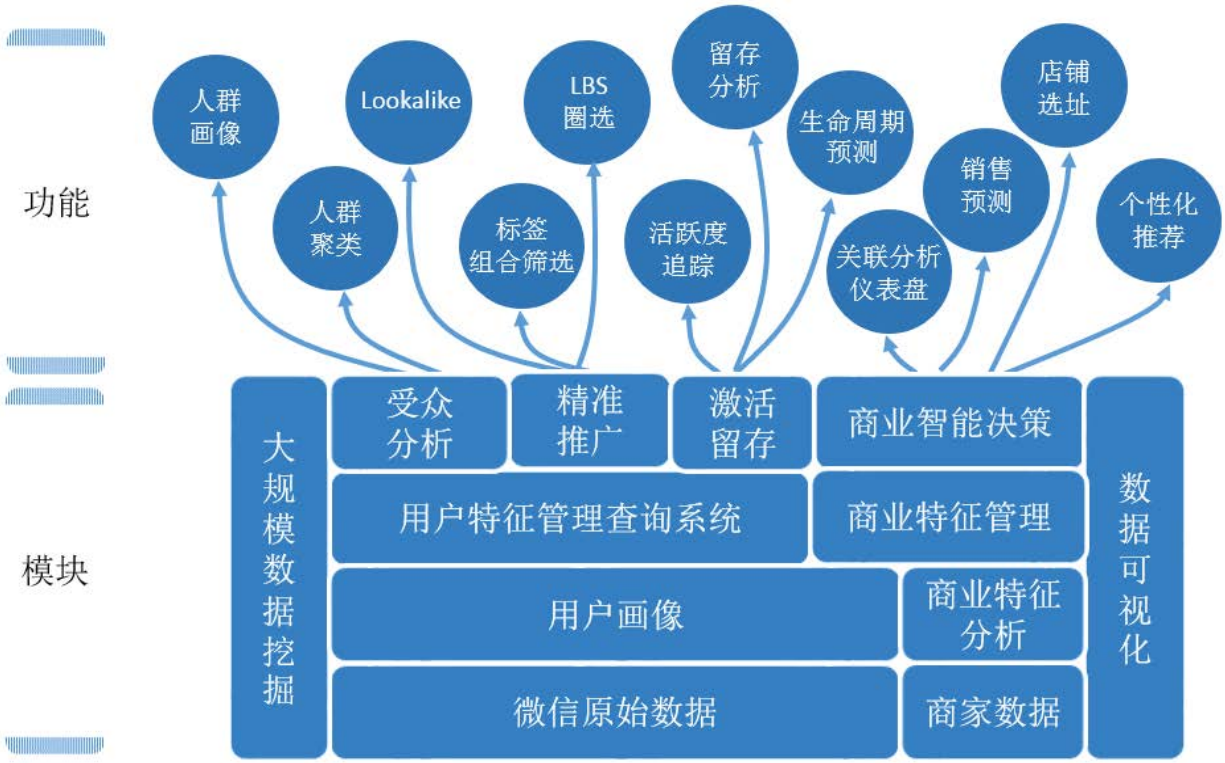


图 1

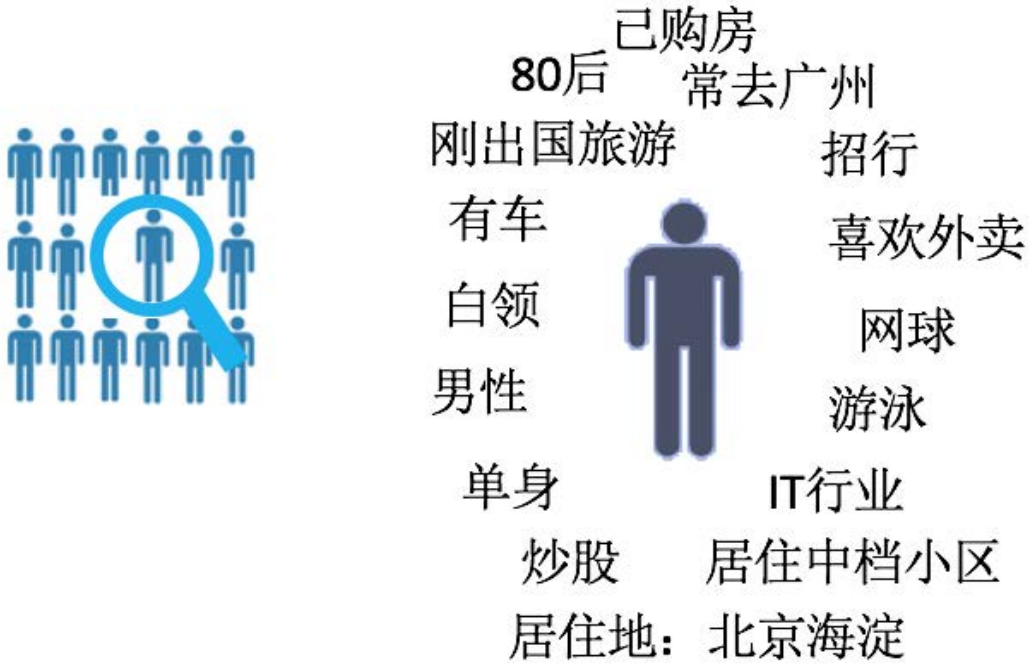


图 2

营人员的三个模块：受众分析、精准推广和激活留存。下面我来逐一介绍。

受众分析

受众分析模块可以根据商家提供的自有用户数据分析人群特征。商家有多种方式告知系统自有用户（图 3）。

1. 商家接入了微信支付，微信 Wifi，微信授权登陆等功能，当用户使用这些功能时微信斑马系统就可以自动追踪到这批

用户。
2. 已有公众号的商家其粉丝即是对应的用户群。
3. 商家可以上传自己的会员包，支持手机号，微信号，QQ 号的匹配。（无需全部的会员记录，只需要商家提供一定量的随机采样即可，考虑隐私保护问题目前系统只支持对 1 万人以上的人群做分析。）

推荐使用第一种方式，因为不仅可以获得目标用户还可以分析人群的活跃度，到店频次和周期。

该模块提供人群画像和人群聚类两个功能。

人群画像

人群画像（Audience Insights）是对人群做画像的工具，它不仅对商家选定的人群做简单的统计分析，还会将用户包的人群特征和平台上全体用户的平均水平做对比，找出最能刻画该人群的特征。同时可以在各个维度上观察人群特征，比如只看男性用户特征，或只看北京用户特征。微信斑马系统的人群画像可以结合本文后面介绍的精准推广模块的各个筛选功能对海量人群秒级画像，比如通过上传公众号列表快速生成关注人群的画像（图 4）。

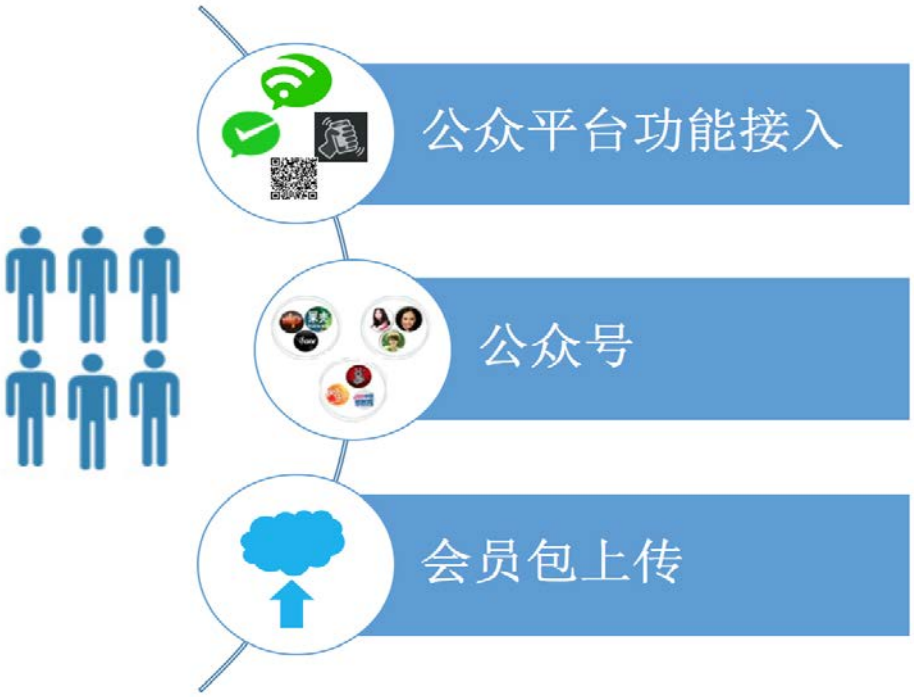


图 3

图 4

人群聚类

虽然人群画像可以通过属性选择在各个维度上观察人群特征，但需要操作人员对行业有一定了解，人群聚类可以根据用户之间的相似性自动将其分为不同的人群，便于商家分析人群的内部结构和分布特点。（图 5）

精准推广

在受众分析后商家和运营人员已经对自己的目标人群有了足够的理解，精准推广模块使用精准人群定义和相似性扩展技术为商家降低获取新用户的成本。

目前系统支持：

1. 基于 LBS 的圈选用户（图 6），适合线下 020 商家。
2. 根据对产品的认识编写用户标签的组合规则筛选受众（图 7）。

优点

- 适合无自有会员的商家使用。
- 只要规则设置的合理可以保证一定的精度。

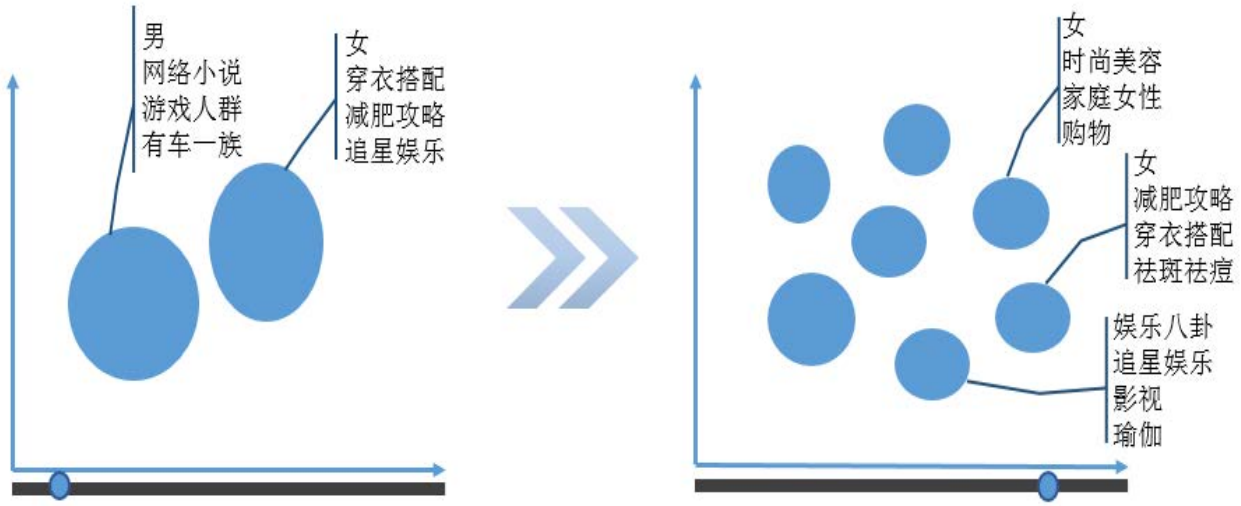


图 5

图 6

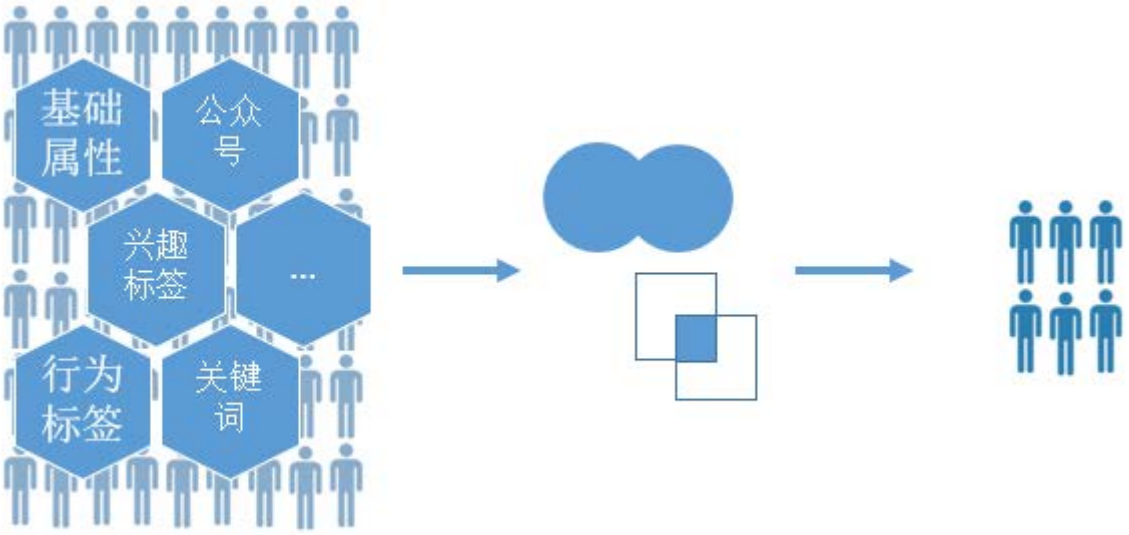


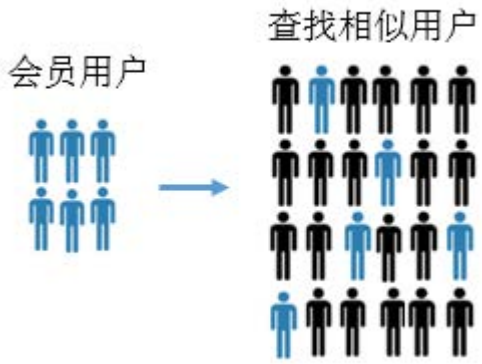
图 7

缺点

- 召回不足：达不到需要的用户数，比如一个海外旅游的公司这次的广告计划是投放 1000 万受众，但我们特征系统里海外旅游标签下只有不到 100 万人。
- 召回过度：选出的用户过多不知道如何做进一步的筛选，比如一个卖运动鞋的品牌这次投放预算最大为 1000 万受众，但我们系统里运动相关的标签下有 5000 万人。
- 筛选的质量依赖于编写规则的人对行业的了解和标签体系的熟悉程度。
- 人群复杂时用有限的标签和规则很难将受众描述清楚。

3. Lookalike

Lookalike Audiences 人群定向策略，简单的说就是用数据描述人群，以人找人，因为大部分商家都有一定量的自有用户（可能来自以往的消费记录或自身会员）那么这些用户的相似用户就是最好的受众。Lookalike 技术的产生有效弥补了标签筛选的不足，是当前社交广告中的主流方法。微信斑马系统的 Lookalike 算法不光可以对召回不足做相似性扩展，还可以对召回过度的情况做按比例精选，同时我们针对微信朋友圈社交广告人群定向的应用做了定制，使其在扩展时可以通过调节互动性参数设置扩展人群倾向于更精准还是更易于互动。如朋友圈广告人群定向时，宝马和奔驰这样的品牌广告商倾向易于产生互动的人群，而母婴，教育这类广告则更倾向于投放的精准性。



激活留存

在获取新用户后，如何提高用户的活跃度，并留住用户防止流失是运营的重点。斑马系统将用户画像与用户生命周期理论结合，利用数据挖掘技术指导商家实现个性化运营。所谓的用户生命周期是营销学里的一个理论，它将用户从获取到离开分为五个阶段（图 8）。

1、活跃度追踪

我们系统提供用户活跃度追踪模块，并给出各个活跃度上的人群特征。如果商家接入到开放

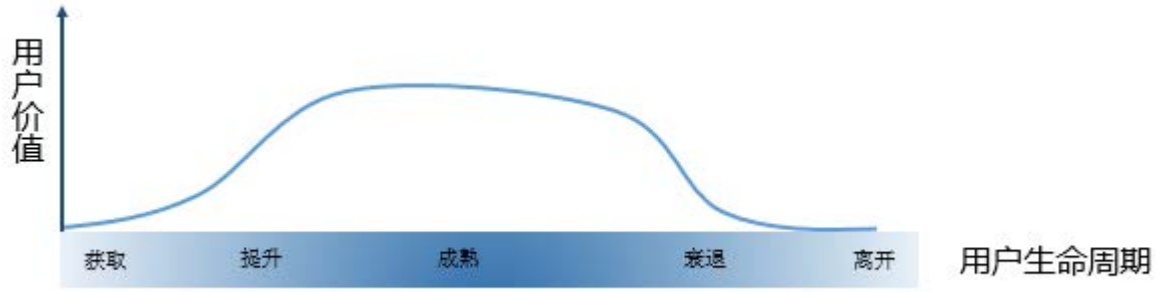


图 8

平台的微信支付等功能，那么无需要做更多事，系统会自动追踪使用该功能用户的活跃度，分析他们随时间变化的规律，否则需要周期性的手动提交数据或是将自己的数据库接入斑马系统。

用户生命周期预测

系统通过挖掘历史数据中用户活跃度的变化，为每个商家生成其独有的用户生命周期模型，自动对其用户进行预测，判断其所处的阶段，告知商家有可能流失的用户群及其流失的速度，商家可以在运营中及时做出激励和促销，防止用户流失。

留存分析

留存分析功能根据历史数据分析用户在各个维度上的留存率，对异常的部分给出预警，以便商家及时发现问题，比如发现某个地区用户流失近期显著增加可能为该地区的分店有问题。

商业智能决策

商业智能决策（BI，Business Intelligence）面向有一定规模的中型或大型企业，需要对接其数据库后做商业特征分析针对不同行业引入销售，物流，生产等维度上的信息，并将其与微信中的用户特征做可视化联合展示，微信斑马系统的 BI 不仅提供传统 BI 中的仪表盘，关联分析和下钻等功能，同时还会针对不同行业提供丰富多样的实用工具，如销售预测，店铺选址，个性化推荐等。

朋友圈广告人群定向投放系统

以上功能可以根据需要灵活组合使用，我们将受众分析，精准推广和朋友圈广告投放系统进行串联和组合，完成了微信朋友圈广告人群定向投放系统（图 9）。

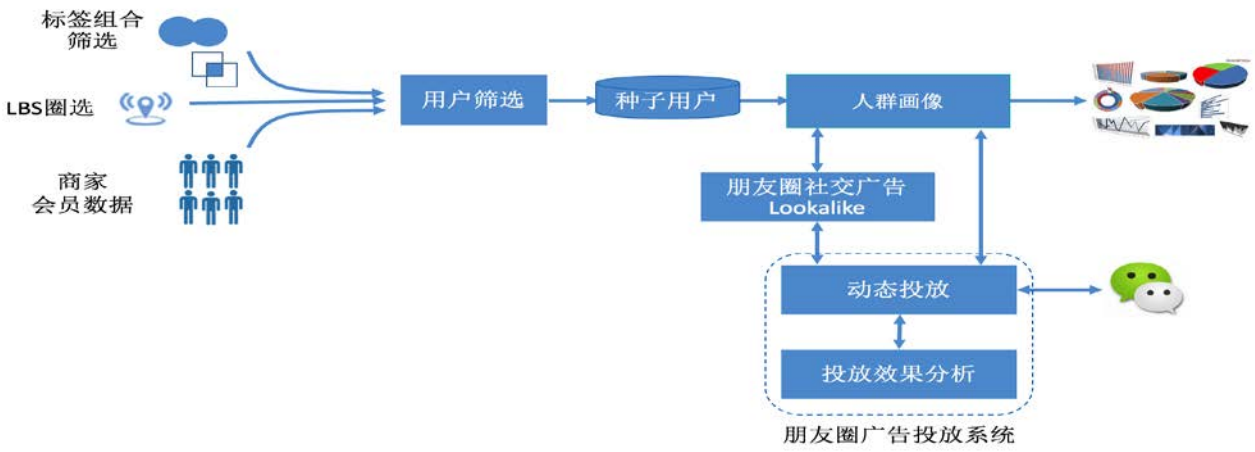
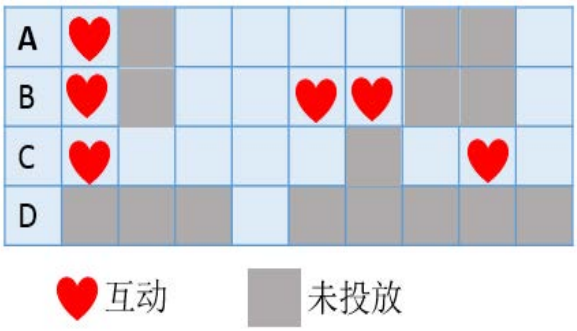


图 9

商家可以通过上传自己的会员包或是通过标签组合筛选和LBS圈选获取初期投放的种子用户，系统获取到种子用户后人群画像系统快速的展示出这批人群的特征和各项统计结果，整个过程只需要等待数秒。商家可以选择特征从不同维度上观察和筛选人群，确认后可以直接投放，当召回不足时可以使用Lookalike系统对其进行扩展，目前的扩展最大规模为种子用户的100倍，另外我们的Lookalike系统还可以对召回过量的问题设置一个0-1之间的倍数做精选，完成受众选择后进入广告投放配置页面设置投放计划，广告投放后可在效果分析页面查看广告投放的各项指标。

朋友圈社交广告Lookalike打分算法

看了上面的内容可能有读者会问朋友圈社交广告的Lookalike打分与相似度Lookalike打分有何不同？因为社交广告展示的效果很大一部分是通过用户在广告上的各项互动反馈指标来衡量的，相似度Lookalike的分数只能反映出该用户和当前商品用户群的相似度，不能保证其在广告投放后有效的正反馈，商家当然希望用户在广告上进行点赞，评论，转发，最好是关注，下单等操作，因此相似度Lookalike分数还需要使用用户在微信朋友圈广告上的各项互动因子进行缩放，互动因子表示当前用户倾向于广告互动的程度，越大越可能产生互动。假如历史上所有用户看到的广告都是一样的，那么可以通过简单的统计对其估算，但难点就在于大部分用户历史投放的广告是不一样的，而且还有部分用户之前从未看过广告。



上图是用户A，B，C，D历史广告的行为，每列为一个历史广告，红星为有互动行为，灰色为该条广告对当前用户没有投放，这里A和B对比因为投放广告内容一致，所有可以近似认为B比A互动因子更高，但对比A和C时，因为投放广告内容是不一致，即使这里C比A的互动次数多也不能断定C比A有更高的互动因子，而对于D由于其在历史上看到的广告数量很少就更没办法估算了。

我们的解决方法是根据历史数据训练了一个用户互动行为的预估模型，再对全量用户预测，预测后每个用户在各个广告上的互动率都在0和1之间。之后就可以根据在各个广告上预测出的互动率之和作为该用户互动因子的近似，这个过程是在线下事先算好的，线上打分时会直接调用当前用户的各项互动因子，并根据设置的互动性参数微调以获得更相似的用户或是更易互动的用户。

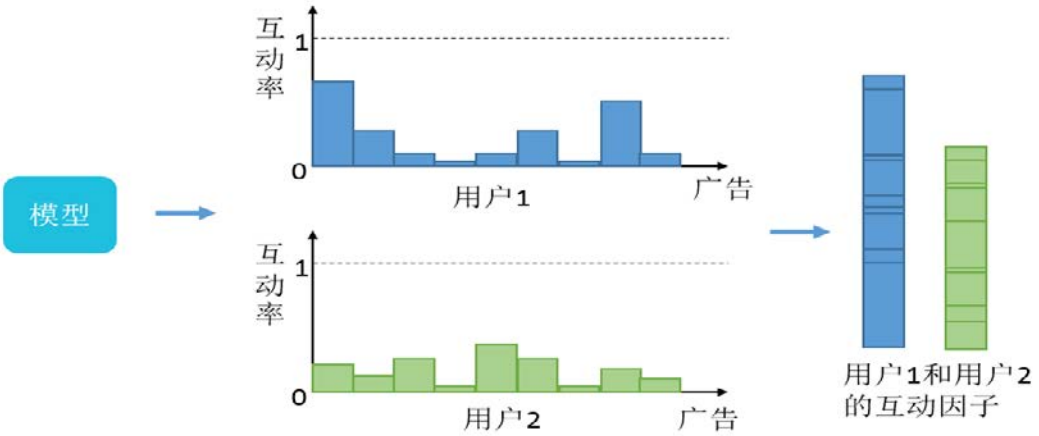


图10

线上效果

该系统已在微信朋友圈广告上使用，下表是对A，B两个广告的投放效果。（这里列出正反馈1，正反馈2，和 负反馈 三项指标，我们系统的实际指标比这复杂的多。）

A广告	正反馈1	正反馈2	负反馈
种子包5万	7.95%	6.96%	1.53%
系统Top100万	16.16%	12.54%	0.89%
系统Top500万	8.82%	8.16%	1.45%
系统Top1000万	7.00%	6.55%	1.78%
系统Top2000万	5.43%	5.13%	2.07%
盲投2600万	0.46%	0.44%	2.23%

B广告	正反馈1	正反馈2	负反馈
种子包15万	7.12%	4.17%	2.02%
系统Top150万	11.84%	11.56%	1.31%
系统Top300万	9.48%	10.14%	1.60%
系统Top750万	6.87%	7.76%	2.05%
系统Top1500万	5.24%	5.90%	2.38%
盲投3600万	0.91%	0.58%	2.06%

其中系统Top 100-2000万表示的是种子用户使用我们系统扩展后根据打分结果选出的前100-2000万用户，显然越靠前的用户效果越好。

我们系统将种子包扩展到90倍到100倍时仍能保证负反馈没有增加的同时有同等效果的正反馈，这里的盲投是根据广告计划在年龄，地域，时间等条件下随机投放的结果，可以看到其效果明显低于种子包和系统扩展的效果。

安全与隐私保护

数据开放和共享是大数据时代公认的趋势，但隐私问题是最大的挑战。微信斑马系统从设计之初就将数据安全和个人隐私问题放在首位，系统遵循以下几个原则：

1. 分析一群人而不分析一个人（我们的系统目前只支持1万以上的人群画像分析）。
2. 不使用个人可辨识信息（Personal Identifiable Information），如：姓名，身份证号，手机号 等。（我们数据处理时使用无任何物理含义的User ID作为各个数据中的统一标识）
3. 通讯和聊天内容神圣不可侵犯，不保存和使用任何通讯和聊天内容。
4. 控制精度，这里的精度并不是指准确度，比如我们在分析用户住址时，只定位到小区，而不再做楼栋和楼层的定位。
5. 只保留和使用一年以内的数据。
6. 所有的标签都由算法自动化生成而不使用人工标注，工程师只负责设计算法。

Spark和Hadoop，孰优孰劣

作者 谢丽

Spark 已经取代 Hadoop 成为最活跃的开源大数据项目。但是，在选择大数据框架时，企业不能因此就厚此薄彼。近日，著名大数据专家 Bernard Marr 在一篇文章中分析了 Spark 和 Hadoop 的异同。

Hadoop 和 Spark 均是大数据框架，都提供了一些执行常见大数据任务的工具。但确切地说，它们所执行的任务并不相同，彼此也并不排斥。虽然在特定的情况下，Spark 据称要比 Hadoop 快 100 倍，但它本身没有一个分布式存储系统。而分布式存储是如今许多大数据项目的基础。它可以将 PB 级的数据集存储在几乎无限数量的普通计算机的硬盘上，并提供了良好的可扩展性，只需要随着数据集的增大增加硬盘。因此，Spark 需要一个第三方的分布式存储。也正是因为这个原因，许多大数据项目都将 Spark 安装在 Hadoop 之上。这样，Spark 的高级分析应用程序就可以使用存储在 HDFS 中的数据了。

与 Hadoop 相比，Spark 真正的优势在于速度。Spark 的大部分操作都是在内存中，而 Hadoop 的 MapReduce 系统会在每次操作之后将所有数据写回到物理存储介质上。这是为了确保在出现问题时能够完全恢复，但 Spark 的弹性分布式数据存储也能实现这一点。

另外，在高级数据处理（如实时流处理和机器学习）方面，Spark 的功能要胜过 Hadoop。

在 Bernard 看来，这一点连同其速度优势是 Spark 越来越受欢迎的真正原因。实时处理意味着可以在数据捕获的瞬间将其提交给分析型应用程序，并立即获得反馈。在各种各样的大数据应用程序中，这种处理的用途越来越多，比如，零售商使用的推荐引擎、制造业中的工业机械性能监控。Spark 平台的速度和流数据处理能力也非常适合机器学习算法。这类算法可以自我学习和改进，直到找到问题的理想解决方案。这种技术是最先进制造系统（如预测零件何时损坏）和无人驾驶汽车的核心。Spark 有自己的机器学习库 MLib，而 Hadoop 系统则需要借助第三方机器学习库，如 Apache Mahout。

实际上，虽然 Spark 和 Hadoop 存在一些功能上的重叠，但它们都不是商业产品，并不存在真正的竞争关系，而通过为这类免费系统提供技术支持赢利的公司往往同时提供两种服务。例如，Cloudera 就既提供 Spark 服务也提供 Hadoop 服务，并会根据客户的需要提供最合适的建议。

Bernard 认为，虽然 Spark 发展迅速，但它尚处于起步阶段，安全和技术支持基础设施方还不发达。在他看来，Spark 在开源社区活跃度的上升，表明企业用户正在寻找已存储数据的创新用法。

GiT GEEKBANG INTERNATIONAL TRAINING 极客邦培训

一线专家团队驱动的企业培训服务

Enterprise Training

同国内外企业深度合作，为其设置专属定制化学习课程，紧密贴合用户个性化需求的内容与形式，提供top级的学习体验。

Deep Workshop

携手全球知名一线技术专家团队分享软件研发技术管理实践，提供研发团队提升必修精选课程。

Open Class

- 工作坊学习体验、案例干货尽收
- 大时段分享
- 全程30个精选课程研修
- 纯净绿色学习环境
- 情景教学、沙盘演练
- 以本土化选题适应城市风向需求

全国服务热线：400-022-6009 | git.geekbang.org

通过大量测试来构建测试系统

作者 Ben Linders 译者 刘嘉洋

在 Agile Testing Days 2015 上，James Lyndsay 办了一个名为“测试巢”的工作坊。在这次工作坊中，他探究了如何设计微小测试的大型集合并将他们的输出显示在测试系统上。他还展示了如何使用工具来帮助我们做到这种测试。InfoQ 就他的这种测试方法采访了他。

InfoQ：你可以解释一下你用“测试巢”的意思吗？

Lyndsay： 鸟巢是用上百块脆弱的碎片组成的——它们独立来看都有欠缺，但是放在一起就可以保护一个正在成长的家庭。作为测试者，我们可以采用单独看来无聊或微不足道的方法，它们可以支持我们探究有关我们正在建立的系统的、更深的真理。他们是押韵的，这是令人愉悦的一件事。

如果想要知道我说的是什么意思，可以看一下 Mike Bostock 的“Will it Shuffle”。在这一页面上，Bostock 用建立一个 1800 尺寸的图像展示了随机算法的缺陷，它们用这个算法显示了 10000 个独立的运行结果的总结。在这张图像中，Bostock 用颜色来展示随机选择的倾向，所以图像的色彩越多越规律，算法缺陷越大。因为算法用了一个取决于浏览器端的内建函数，所以一样的代码在不同的浏览器端会显示不同的涌现性，也因此展示了不同的缺陷。一个独立的随机选择不会告诉我们这些，但是 10000 个随机选择就会暴露这些缺陷。选择并建立一张正确的图像会让我们更容易注意到这些不同。

InfoQ：你认为测试中的可视化的重要性有哪些？

Lyndsay：可视化将许多量化的结果放在一个适合我们大脑的、惊奇的视觉过程的模式中，这使得数据中的信息可以影响我们脑中的模型和我们在团队中发展的共同理解。一个图像值得用一千个数据点来构造。

InfoQ：在你的工作坊中，你问了观众他们用什么工具来使数据可视化。那你听下来，他们用的最多的是哪些工具？你建议用哪些工具？

Lyndsay：Excel。这是测试者们通用的“螺丝刀”。它可以画出很不错的图像，还可以将数据处理、过滤、排序。然而，用名字在列与列之间转换很不方便，这使我们在数据中找出原数据变得痛苦。我认为我们可以找一个大数据的工具，例如 Splunk 和 Kibana，来更好地分析测试结果，并且我也很乐意在今年晚些时候玩一玩这两个工具。但是绘图工具也是链子上的一环，所以我们也需要工具来帮助我们生成和应用数据。

InfoQ：你在工作坊中提到你经常用散点图来使数据可视化。你可以解释一下这是如何做到的吗？

Lyndsay：我用了一种工具，是 Visual Data Tools 的 DataGraph。这让我从很多测试中获取了度量表，使划分表格的列变得简单——这使得我能够在度量之中过滤、上色并调整元素的大小。DataGraph 需要付费并且只能在 OS X 中使用，但还有一个不错的开源工具——DensityDesign 的 Raw，它是浏览器端的并且有很不错的散点图。

InfoQ：在什么样的情况下你会推荐用许多自动生成的测试？

Lyndsay：我很快找到了一些有趣的、在探索性测试中驱动组件的问题。例如，代码或单元测试会引导我走向需要一个或多个输入范围的算法。如果这个算法只在一些独立的情况下被检查，我会通过组合变量生成一些细节或大概的范围，并用图来显示其中的重点。

我还在寻找集成测试中的 surprise 时有了很大的进展，特别是当一些组件是用不同技术实现时或当一个简单的解决方案用很强大的部分来建立时。

这方面没有什么特别新奇的东西了，并且性能测试与模糊测试有清晰的比较。

InfoQ：那你什么时候不推荐使用它？

Lyndsay：它对于系统中任务重的部分很容易有限制。你需要善用你的工具，并且任何意义上的完整性都应该带有怀疑——即使它看起来很显然，它通常会更深入。它不是一个有效地验证行为的方法，然而很多时候度量都是符合预期的。

有的时候你会使你写代码的同事感到灰心。但是有的时候我们需要对我们在建立的系统苛刻——远远超过简单的代码层面——来找到真理。

InfoQ：这种测试途径的好处有什么？

Lyndsay：这是一个来显示系统行为范围的快速且省事的方法——并且如果那些行为令我们感到意外，我们还有机会接近理解我们做过的真实的本质和风险。

www.speedycloud.cn
北京迅达云成科技有限公司

扫描微信获得更多详情

SpeedyCloud迅达云

- 全球部署的云计算节点，面向全球用户提供云计算服务
- 性能卓越的云主机，为应用弹性扩展提供无限可能
- 分布式的对象存储服务，提供海量、稳定、安全的存储空间
- 灵活划分的SDN网络，轻松实现网络隔离
- 高效率的云分发服务，让网站平均提速3倍以上
- 精细控制的防火墙策略，全面保护主机安全
- 功能完善的可编程API，使资源可以灵活调度

SpeedyCloud

让云服务加速您的成功！

云主机　云存储　云数据库　云缓存　云分发　SDN方案　云安全　云DNS　负载均衡

实施微服务，我们需要哪些基础框架

作者 杨波

微服务架构是当前互联网业界的一个技术热点，圈里有不少同行朋友当前有计划在各自公司开展微服务化体系建设，他们都有相同的疑问：一个微服务架构有哪些技术关注点？需要哪些基础框架或组件来支持微服务架构？这些框架或组件该如何选型？笔者之前在两家大型互联网公司参与和主导过大型服务化体系和框架建设，同时在这块也投入了很多时间去学习和研究，有一些经验和学习心得，可以和大家一起分享。

服务注册、发现、负载均衡和健康检查

和单块（Monolithic）架构不同，微服务架构是由一系列职责单一的细粒度服务构成的分布式网状结构，服务之间通过轻量机制进行通信，这时候必然引入一个服务注册发现问题，也就是说服务提供方要注册通告服务地址，服务的调用方要能发现目标服务，同时服务提供方一般以集群方式提供服务，也就引入了负载均衡和健康检查问题。根据负载均衡 LB 所在位置的不同，目前主要的服务注册、发现和负载均衡方案有三种。

第一种是集中式 LB 方案，如图 1，在服务消费者和服务提供者之间有一个独立的 LB，LB 通常是专门的硬件设备如 F5，或者基于软件如 LVS，HAproxy 等实现。LB 上有所有服务的地址映射表，通常由运维配置注册，当服务消费方调用某个目标服务时，它向 LB 发起请求，由 LB 以某种策略（比如 Round-Robin）做负载均衡后将请求转发到目标服务。LB 一般具备健康检查能力，能自动摘除不健康的服务实例。服务消费方如何发现 LB 呢？通常的做法是通过 DNS，运维人员为服务配置一个 DNS 域名，这个域名指向 LB。

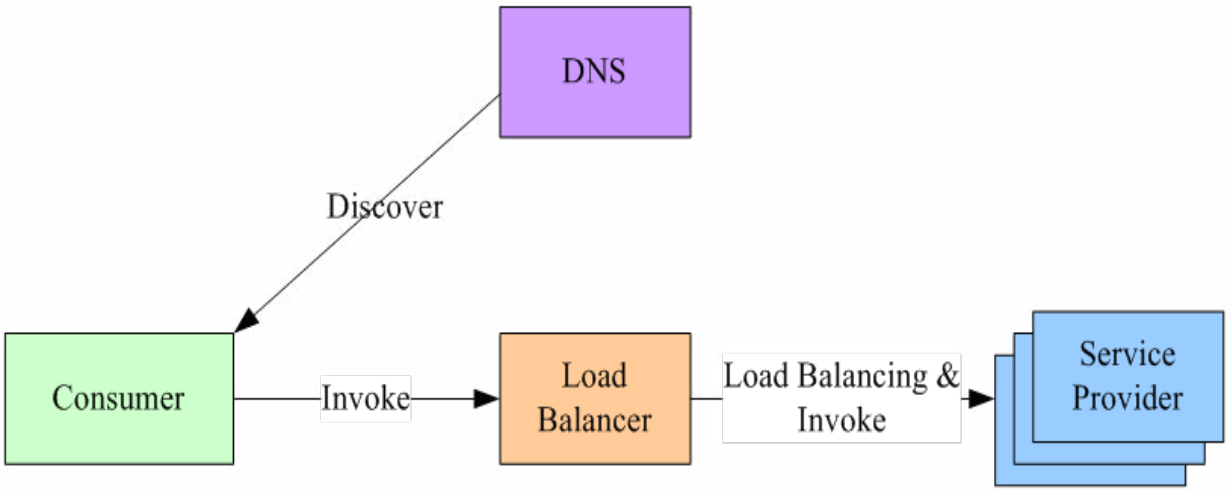


图 1 集中式 LB 方案

集中式 LB 方案实现简单，在 LB 上也容易做集中式的访问控制，这一方案目前还是业界主流。集中式 LB 的主要问题是单点问题，所有服务调用流量都经过 LB，当服务数量和调用量大的时候，LB 容易成为瓶颈，且一旦 LB 发生故障对整个系统的影响是灾难性的。另外，LB 在服务消费方和服务提供方之间增加了一跳 (hop)，有一定性能开销。

第二种是进程内 LB 方案，针对集中式 LB 的不足，进程内 LB 方案将 LB 的功能以库的形式集成到服务消费方进程里头，该方案也被称为软负载 (Soft Load Balancing) 或者客户端负载方案，图 2 展示了这种方案的工作原理。这一方案需要一个服务注册表 (Service Registry) 配合支持服务自注册和自发现，服务提供方启动时，首先将服务地址注册到服务注册表（同时定期报心跳到服务注册表以表明服务的存活状态，相当于健康检查），服务消费方要访问某个服务时，它通过内置的 LB 组件向服务注册表查询（同时缓存并定期刷新）目标服务地址列表，然后以某种负载均衡策略选择一个目标服务地址，最后向目标服务发起请求。这一方案对服务注册表的可用性 (Availability) 要求很高，一般采用能满足高可用分布式一致的组件（例如 Zookeeper, Consul, Etcd 等）来实现。

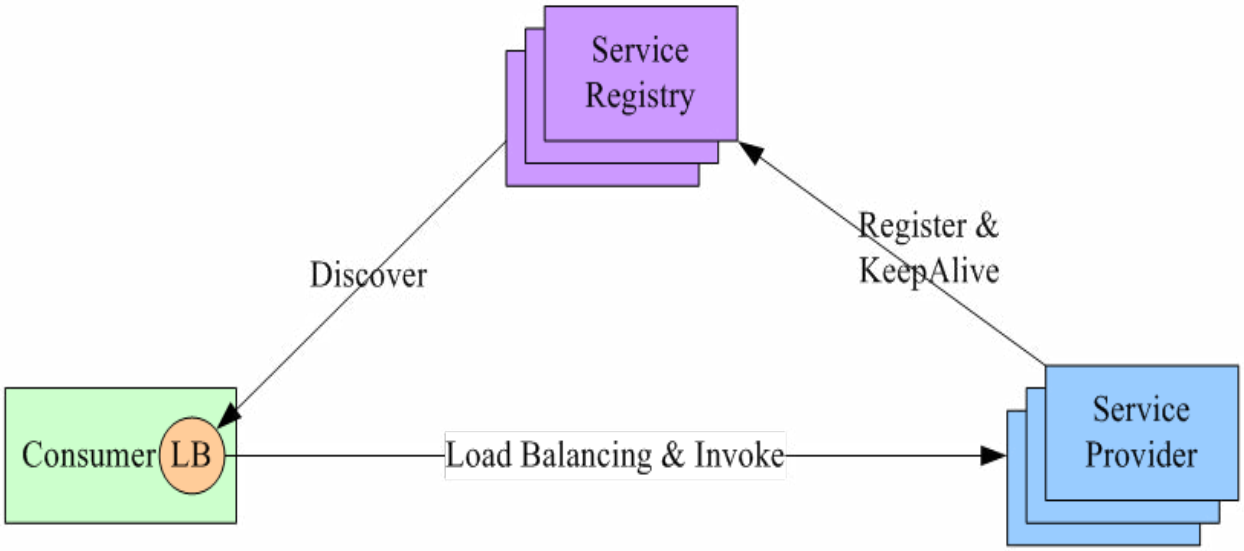


图 2 进程内 LB 方案

进程内 LB 方案是一种分布式方案，LB 和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。但是，该方案以客户库 (Client Library) 的方式集成到服务调用方进程里头，如果企业内有多种不同的语言栈，就要配合开发多种不同的客户端，有一定的研发和维护成本。另外，一旦客户端跟随服务调用方发布到生产环境中，后续如果要对客户库进行升级，势必要求服务调用方修改代码并重新发布，所以该方案的升级推广有不小的阻力。

进程内 LB 的案例是 Netflix 的开源服务框架，对应的组件分别是：Eureka 服务注册表，Karyon 服务端框架支持服务自注册和健康检查，Ribbon 客户端框架支持服务自发现和软路由。另外，阿里开源的服务框架 Dubbo 也是采用类似机制。

第三种是主机独立 LB 进程方案，该方案是针对第二种方案的不足而提出的一种折中方案，原理和第二种方案基本类似，不同之处是，他将 LB 和服务发现功能从进程内移出来，变成主机上的一个独立进程，主机上的一个或者多个服务要访问目标服务时，他们都通过同一主机上的独立 LB 进程做服务发现和负载均衡，见图 3。

该方案也是一种分布式方案，没有单点问题，一个 LB 进程挂了只影响该主机上的服务调用方，服务调用方和 LB 之间是进程内调用，性能好，同时，该方案还简化了服务调用方，不需要为不同语言开发客户库，LB 的升级不需要服务调用方改代码。该方案的不足是部署较复杂，环节多，出错调试排查问题不方便。

该方案的典型案例是 Airbnb 的 SmartStack 服务发现框架，对应组件分别是：Zookeeper 作为服务注册表，Nerve 独立进程负责服务注册和健康检查，Synapse/HAproxy 独立进程负责服务发现和负载均衡。Google 最新推出的基于容器的 PaaS 平台 Kubernetes，其内部服务发现采用类似的机制。

服务前端路由

微服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统（例如客户的浏览器、移动设备等等）访问到，这就涉及服务的前端路由，对应的组件是服务网关 (Service Gateway)，见图 4，网关是连接企业内部和外部系统的一道门，有如下关键作用：

1. 服务反向路由，网关要负责将外部请求反向路由到内部具体的微服务，这样虽然企业内部是复杂的分布式微服务结构，但是外部系统从网关上看到的就像是一个统一的完整服务，网关屏蔽了后台服务的复杂性，同时也屏蔽了后台服务的升级和变化。
2. 安全认证和防爬虫，所有外部请求必须经过网关，网关可以集中对访问进行安全控制，比如用户认证和授权，同时还可以分析访问模式实现防爬虫功能，网关是连接企业内外系统的安全之门。
3. 限流和容错，在流量高峰期，网关可以限制流量，保护后台系统不被大流量冲垮，在内部系统出现故障时，网关可以集中做容错，保持外部良好的用户体验。
4. 监控，网关可以集中监控访问量，调用延迟，错误计数和访问模式，为后端的性能优化或者扩容提供数据支持。
5. 日志，网关可以收集所有的访问日志，进入后台系统做进一步分析。

除以上基本能力外，网关还可以实现线上引流，线上压测，线上调试 (Surgical debugging)，金丝雀测试 (Canary Testing)，数据中心双活 (Active-Active HA) 等高级功能。

网关通常工作在 7 层，有一定的计算逻辑，一般以集群方式部署，前置 LB 进行负载均衡。

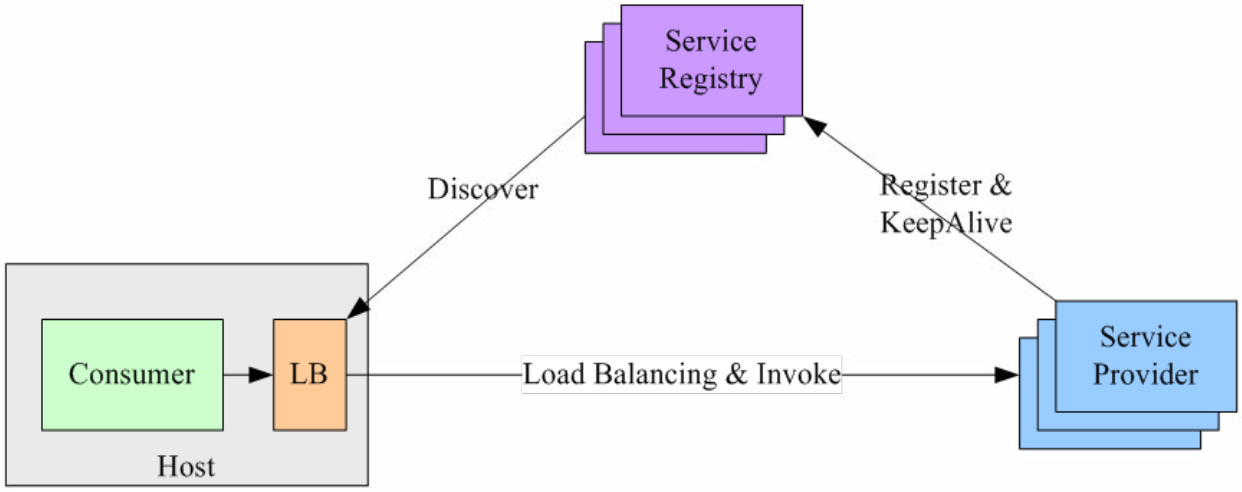


图 3 主机独立 LB 进程方案

开源的网关组件有 Netflix 的 Zuul，特点是动态可热部署的过滤器（filter）机制，其它如 HAproxy，Nginx 等都可以扩展作为网关使用。

在介绍过服务注册表和网关等组件之后，我们可以通过一个简化的微服务架构图（图 5）来更加直观地展示整个微服务体系内的服务注册发现和路由机制，该图假定采用进程内 LB 服务发现和负载均衡机制。在下图 Fig 5 的微服务架构中，服务简化为两层，后端通用服务（也称中间层服务 Middle Tier Service）和前端服务（也称边缘服务 Edge Service，前端服务的作用是对后端服务做必要的聚合和裁剪后暴露给外部不同的设备，如 PC，Pad 或者 Phone）。后端服务启动时会将地址信息注册到服务注册表，前端服务通过查询服务注册表就可以发现然后调用后端服务；前端服务启动时也会将地址信息注册到服务注册表，这样网关通过查询服务注册表就可以将请求路由到目标前端服务，这样整个微服务体系的服务自注册自发现和软路由就通过服务注册表和网关串联起来了。如果以面向对象设计模式的视角来看，网关类似 Proxy 代理或者 Façade 门面模式，而服务注册表和服务自注册自发现类似 IoC 依赖注入模式，微服务可以理解为基于网关代理和注册表 IoC 构建的分布式系统。

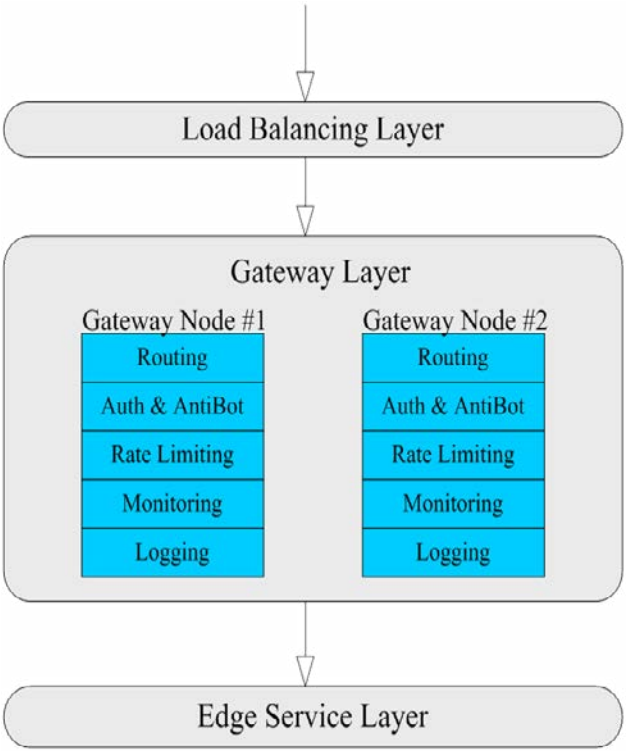


图 4 服务网关

服务容错

当企业微服务化以后，服务之间会有错综复杂的依赖关系，例如，一个前端请求一般会依赖于多个后端服务，技术上称为 1 -> N 扇出（见图 6）。在实际生产环境中，服务往往不是百分百可靠，服务可能会出错或者产生延迟，如果一个应用不能对其依赖的故障进行容错和隔

离，那么该应用本身就处在被拖垮的风险中。在一个高流量的网站中，某个单一后端一旦发生延迟，可能在数秒内导致所有应用资源（线程，队列等）被耗尽，造成所谓的雪崩效应(Cascading Failure，见图 7)，严重时可致整个网站瘫痪。

经过多年的探索和实践，业界在分布式服务容错一块探索出了一套有效的容错模式和最佳实践，主要包括：

1. 电路熔断器模式(Circuit Breaker Patten)，该模式的原理类似于家里的电路熔断器，如果家里的电路发生短路，熔断器能够主动熔断电路，以避免灾难性损失。在分布式系统中应用电路熔断器模式后，当目标服务慢或者大量超时，调用方能够主动熔断，以防止服务被进一步拖垮；如果情况又好转了，电路又能自动恢复，这就是所谓的弹性容错，系统有自恢复能力。图 8 是一个典型的具备弹性恢复能力的电路保护器状态图，正常状态下，电路处于关闭状态(Closed)，如果调用持续出错或者超时，电路被打开进入熔断状态(Open)，后续一段时间内的所有调用都会被拒绝(Fail Fast)，一段时间以后，保护器会尝试进入半熔断状态(Half-Open)，允许少量请求进来尝试，如果调用仍然失败，则回到熔断状态，如果调用成功，则回到电路闭合状态。

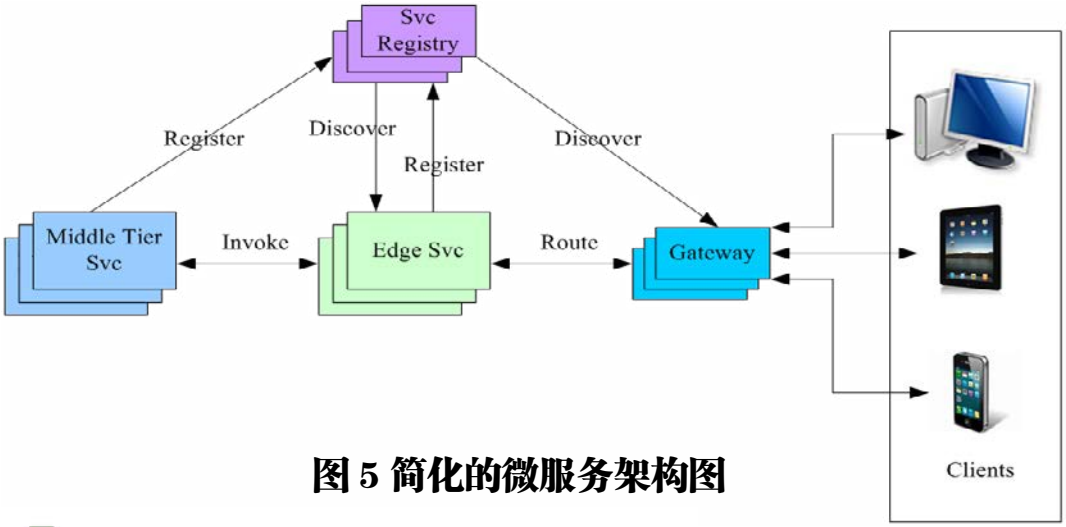


图 5 简化的微服务架构图

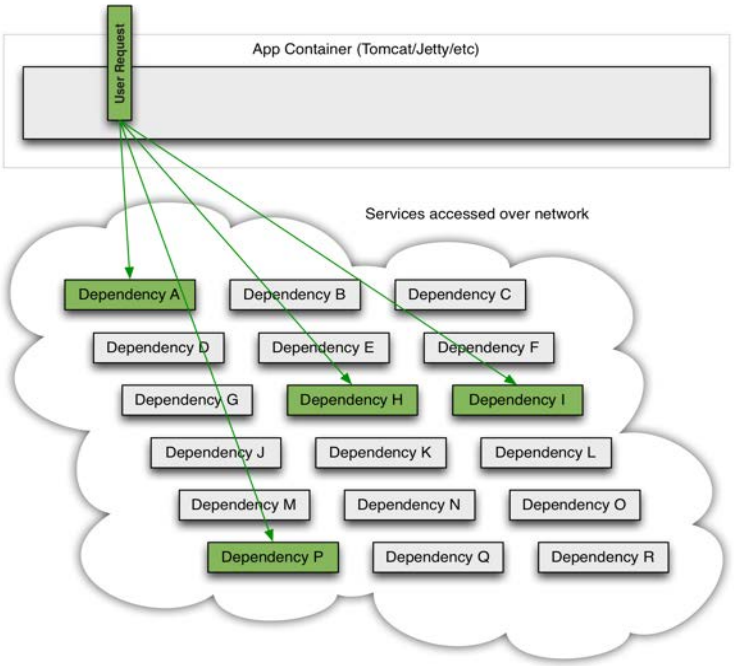


图 6 服务依赖

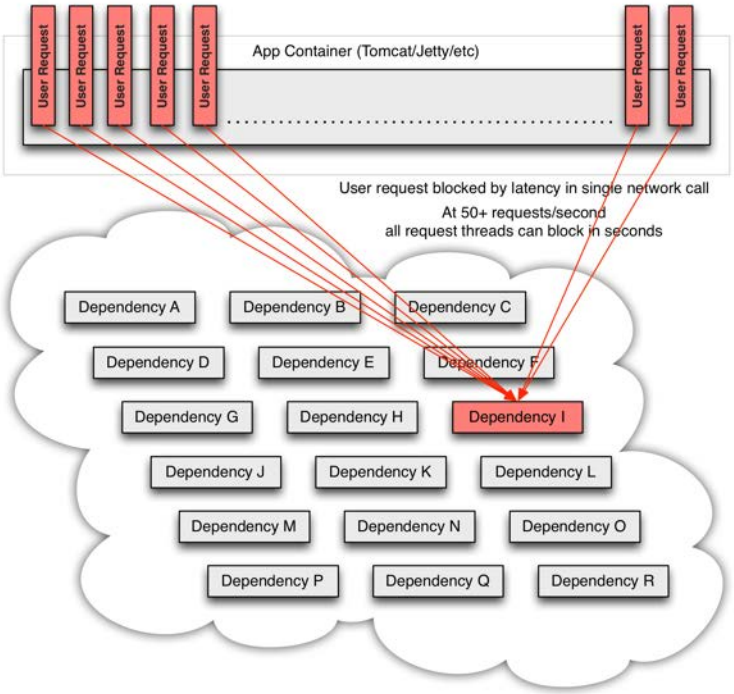


图 7 高峰期单个服务延迟致雪崩效应

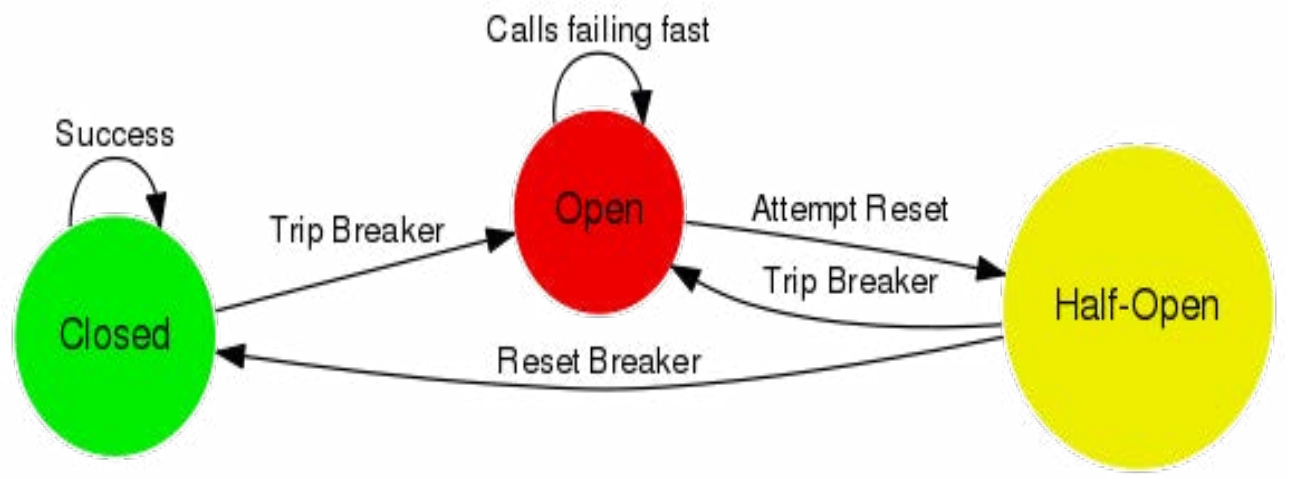


图 8 弹性电路保护状态图

2. 舱壁隔离模式(Bulkhead Isolation Pattern)，顾名思义，该模式像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响 。线程隔离(Thread Isolation)就是舱壁隔离模式的一个例子，假定一个应用程序A调用了Svc1/Svc2/Svc3三个服务，且部署A的容器一共有120个工作线程，采用线程隔离机制，可以给对Svc1/Svc2/Svc3的调用各分配40个线程，当Svc2慢了，给Svc2分配的40个线程因慢而阻塞并最终耗尽，线程隔离可以保证给Svc1/Svc3分配的80个线程可以不受影响，如果没有这种隔离机制，当Svc2慢的时候，120个工作线程会很快全部被对Svc2的调用吃光，整个应用程序会全部慢下来。
3. 限流(Rate Limiting/Load Shedder)，服务总有容量限制，没有限流机制的服务很容易在突发流量（秒杀，双十一）时被冲垮。限流通常指对服务限定并发访问量，比如单位时间只允许100个并发调用，对超过这个限制的请求要拒绝并回退。
4. 回退(fallback)，在熔断或者限流发生的时候，应用程序的后续处理逻辑是什么？回退是系统的弹性恢复能力，常见的

处理策略有，直接抛出异常，也称快速失败（Fail Fast），也可以返回空值或缺省值，还可以返回备份数据，如果主服务熔断了，可以从备份服务获取数据。

Netflix 将上述容错模式和最佳实践集成到一个称为 Hystrix 的开源组件中，凡是需要容错的依赖点（服务，缓存，数据库访问等），开发人员只需要将调用封装在 Hystrix Command 里头，则相关调用就自动置于 Hystrix 的弹性容错保护之下。Hystrix 组件已经在 Netflix 经过多年运维验证，是 Netflix 微服务平台稳定性和弹性的基石，正逐渐被社区接受为标准容错组件。

服务框架

微服务化以后，为了让业务开发人员专注于业务逻辑实现，避免冗余和重复劳动，规范研发提升效率，必然要将一些公共关注点推到框架层面。服务框架（图 9）主要封装公共关注点逻辑，包括：

1. 服务注册、发现、负载均衡和健康检查，假定采用进程内 LB 方案，那么服务自注册一般统一做在服务器端框架中，健康检查逻辑由具体业务服务定制，框架层提供调用健康检查逻辑的机制，服务发现和负载均衡则集成在服务客户端框架中。
2. 监控日志，框架一方面要记录重要的框架层日志、metrics 和调用链数据，还要将日志、metrics 等接口暴露出来，让业务层能根据需要记录业务日志数据。在运行环境中，所有日志数据一般集中落地到企业后台日志系统，做进一步分析和处理。
3. REST/RPC 和序列化，框架层要支持将业务逻辑以 HTTP/REST 或者 RPC 方式暴露出来，HTTP/REST 是当前主流 API 暴露方式，在性能要求高的场合则可采用 Binary/RPC 方式。针对当前多样化的设备类型（浏览器、普通 PC、无线设备等），框架层要支持可定制的序列化机制，例如，对浏览器，框架支持输出 Ajax 友好的 JSON 消息格式，而对无线设备上的 Native App，框架支持输出性能高的 Binary 消息格式。
4. 配置，除了支持普通配置文件方式的配置，框架层还可集成动态运行时配置，能够在运行时针对不同环境动态调整服务的参数和配置。
5. 限流和容错，框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。
6. 管理接口，框架集成管理接口，一方面可以在线查看框架和服务内部状态，同时还

图 9 服务框架

可以动态调整内部状态，对调试、监控和管理能提供快速反馈。Spring Boot 微框架的 Actuator 模块就是一个强大的管理接口。

7. 统一错误处理，对于框架层和服务的内部异常，如果框架层能够统一处理并记录日志，对服务监控和快速问题定位有很大帮助。
8. 安全，安全和访问控制逻辑可以在框架层统一进行封装，可做成插件形式，具体业务服务根据需要加载相关安全插件。
9. 文档自动生成，文档的书写和同步一直是一个痛点，框架层如果能支持文档的自动生成和同步，会给使用 API 的开发和测试人员带来极大便利。Swagger 是一种流行 Restful API 的文档方案。

当前业界比较成熟的微服务框架有 Netflix 的 Karyon/Ribbon，Spring 的 Spring Boot/Cloud，阿里的 Dubbo 等。

运行期配置管理

服务一般有很多依赖配置，例如访问数据库有连接字符串配置，连接池大小和连接超时配置，这些配置在不同环境（开发 / 测试 / 生产）一般不同，比如生产环境需要配连接池，而开发测试环境可能不配，另外有些参数配置在运行期可能还要动态调整，例如，运行时根据流量状况动态调整限流和熔断阀值。目前比较常见的做法是搭建一个运行时配置中心支持微服务的动态配置，简化架构如图 10。

动态配置存放在集中的配置服务器上，用户通过管理界面配置和调整服务配置，具体服务通过定期拉 (Scheduled Pull) 的方式或者服务器推 (Server-side Push) 的方式更新动态配置，拉方式比较可靠，但会有延迟同时有无效网络开销（假设配置不常更新），服务器推方式能及时更新配置，但是实现较复杂，一般在服务和配置服务器之间要建立长连接。配置中心还要解决配置的版本控制和审计问题，对于大规模服务化环境，配置中心还要考虑分布式和高可用问题。

配置中心比较成熟的开源方案有百度的 Disconf，360 的 QConf，Spring 的 Cloud Config 和阿里的 Diamond 等。

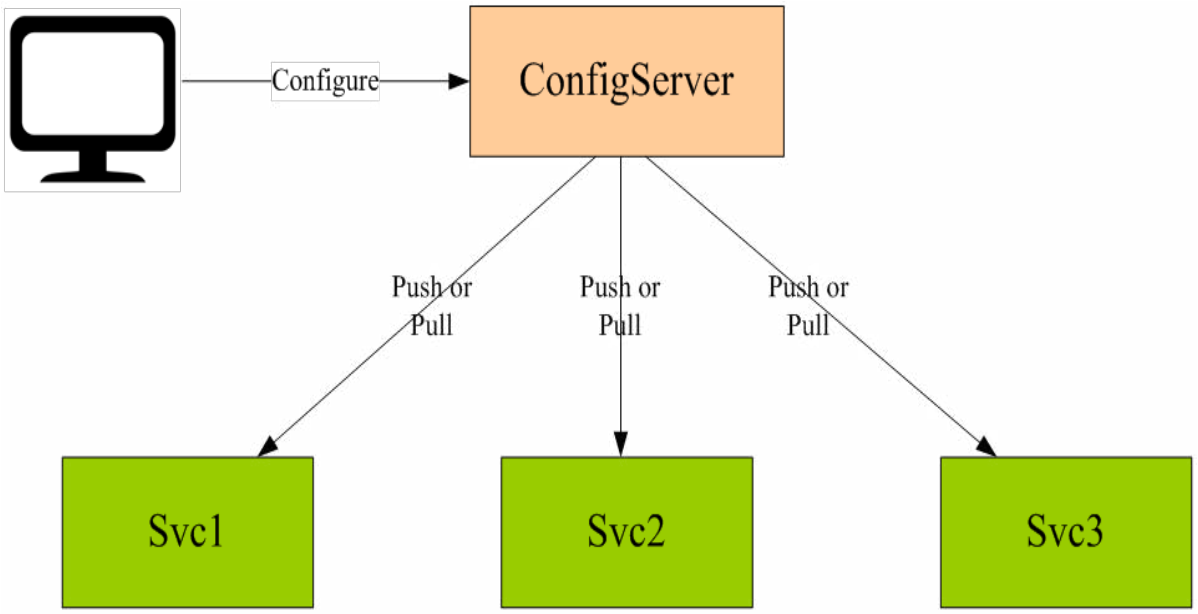


图 10 服务配置中心

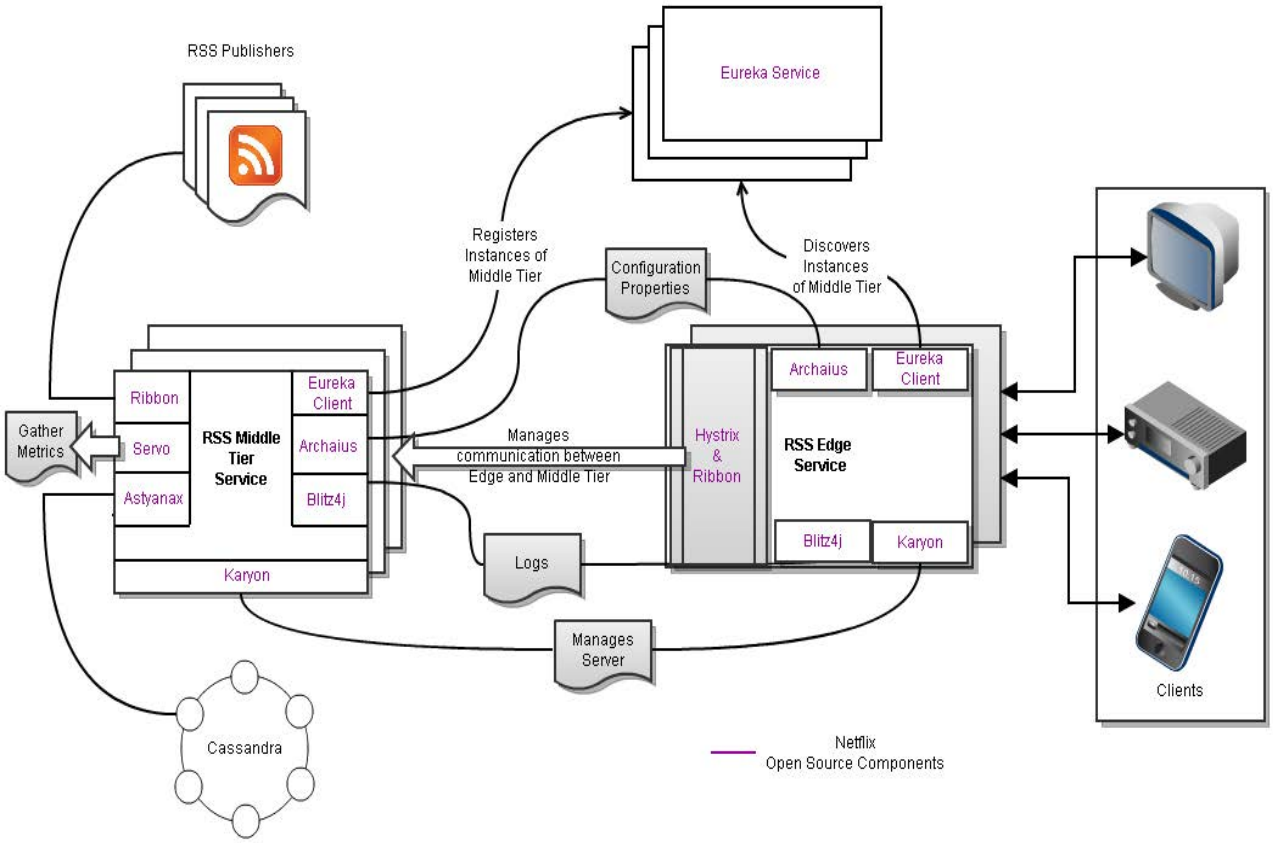


图 11 基于 Netflix 开源组件的微服务框架

Netflix 的微服务框架

Netflix 是一家成功实践微服务架构的互联网公司，几年前，Netflix 就把它的几乎整个微服务框架栈开源贡献给了社区，这些框架和组件包括：

1. Eureka：服务注册发现框架
2. Zuul：服务网关
3. Karyon：服务端框架
4. Ribbon：客户端框架
5. Hystrix：服务容错组件
6. Archaius：服务配置组件
7. Servo：Metrics 组件
8. Blitz4j：日志组件

图 11 展示了基于这些组件构建的一个微服务框架体系，来自 recipes-rss。

Netflix 的开源框架组件已经在 Netflix 的大规模分布式微服务环境中经过多年的生产实战验证，正逐步被社区接受为构造微服务框架的标准组件。Pivotal 去年推出的 Spring Cloud 开源产品，主要是基于对 Netflix 开源组件的进一步封装，方便 Spring 开发人员构建微服务基础框架。对于一些打算构建微服务框架体系的公司来说，充分利用或参考借鉴 Netflix 的开源微服务组件（或 Spring Cloud），在此基础上进行必要的企业定制，无疑是通向微服务架构的捷径。

个推 GeTui.com
最专业的推送技术服务商

- 为应用提供消息推送技术解决方案
- APP三分钟即可快速集成云推送SDK
- 免去自行开发成本，提高用户活跃度与黏性
- 有效提高ARPU值，增加应用收益

个推1.0-实时推送

消息即时送达，提高用户活跃度，多个APP合并通道省电省流量，应用进程互相看护，活跃有保障，精确数据报表，数据实时统计反馈

个推2.0-智能推送

智能标签，建立精准用户画像，避免盲目推送，实现精细化运营，分组对比测试，优化推送决策，强大的数据联盟，清晰的行业报告

个推3.0-应景推送

精准捕捉用户场景，合适地点触发消息，进入、停留离开、情景化推送，性别、年龄、爱好、智能标签分类，学校、商场、火车站、地理围栏调用

个推官网：www.GeTui.com
客服电话：4006-808-606
商务咨询：support@getui.com
客服 QQ：2880983150

关注个推

官网注册

重建还是重构

作者 Ben Linders 译者 覃璐

在 Agile Testing Days 2015 大会上， Wouter Lagerweij 谈到了如何重建一个遗留系统而不是重构它，来帮助团队采取敏捷实践，比如测试驱动开发，自动化测试，持续交付。他的谈话基于他的博客文章 Don’t Refactor. Rebuild. Kinda。

InfoQ 采访了 Lagerweij 关于是什么让重构如此困难，重建软件的风险是否比重构小，以及持续交付如何配合软件的重建。InfoQ 同时请教了他关于重建和重构的建议。

InfoQ：您能解释一下是什么使得重构这么困难吗？

Lagerweij：只要你去做，重构是一项相当简单的实践。同样的例子是单元测试。只要你坚持为代码编写测试用例，或者清理代码中的设计问题，一小步一小步的，这并不困难。

可是你越是把这些丢在一边，捡起来的时候就越困难。这就是为什么每个人都一直说“技术债务”。可是实际上它通常不是指 Ward Cunningham 所创造的技术债务（参见 Ward Explains Debt Metaphor）。

软件行业并不是唯一发生这些事情的地方。只是询问一些活跃在医疗保健行业的专业医师。了解到外科医生坚持在手术前洗手能降低一半的手术并发症，但是即使有积极的宣传和结构化的清单，它依然很难被遵守。

所以，由于一些团队推迟重构，我们有一些混乱，或者说“遗留代码库”。由于这些团队没有足够的重构经验（否则他们已经这样做了），他们无疑不能完成修复这些混乱代码的工作。

此时这些团队认识到他们应该为他们的“债务”做些事情，他们不得不接受一个系统，在其中努力学习用以提升自己系统的技能！当小的改动在系统的不同部位导致不可预知的结果时，是对重构中积极的学习经历不利的。而且任何开发人员都知道为封闭的，紧耦合的系统添加单元测试是一个困难和不愉快的经历。

我想我要说的是，如果你等到遇到这些麻烦才开始学习这些技能，很可能你将不会成功的应用它们。也就是说，在掌握它们之前，你更可能会先放弃它们。

InfoQ: 在您看来重建软件比重构的风险更小？您能解释一下吗？

Lagerweij：说实话，如果你的团队成员知道如何去改进一个遗留系统，重构始终是更好的选择。它的风险更小，并且比任何类型的重写的开销要少。

但是不幸的是这些技能仍然非常稀少。如果团队中的成员之前没有做过这类工作，之后你的所有工作都会慢于预期。你会在开发团队和公司中增加挫败感。

有一些组织发现自己陷入了困境。他们简直没有专业人员来解决他们的技术问题。他们不能构造有竞争力的新功能。甚至找到几个有经验的人也很难，常常是“太小，太迟”的情况。

这种情况下重写是更有吸引力的。

InfoQ：您能给出一些例子来展示团队如何来重建软件以及做持续交付吗？是什么让它们成为一个好的组合？

Lagerweij：重写的一个好处是你能重新开始。这意味着这次你能确保你这样做是正确的。当然，大多数时候，你不能。就像我之前说的，处理这些事情最好的方式，例如测试和重构，就是持续不断的做。但是如果你之前从来没有做好过，它怎么会突然就能工作？

我在谈话中说过的，我们已经在我当前工作的团队试过。我们使用测试驱动开发（也叫做行为驱动开发），我们会保持代码的整洁，确保始终有 100% 的单元测试覆盖率。他们不再害怕他们的遗留系统，他们知道他们能够做到这一点。

我们也同意，确保我们不会受到放松控制的诱惑，我们从第一天起就做到完全的持续部署。这意味着开发者每次推送代码到 GitHub 中时，它会自动构建，测试并部署一直到生产。这使得我们任何时候都能极好的关注在保持高质量上。你不能推迟那些测试，因为推送未测试的代码会破坏构建，使得整个团队等着你。但是你也不想跳过测试（或者写一个未检查的单元测试只是为了糊弄覆盖率），因为你实际上可能破坏生产。你自己，很明显。

还有一些简单的心理作用，没有人真正关心什么让测试覆盖率从 2.1% 到 2.0%，但是当它从 100% 掉到 99.9% 时，整个团队会要求一个解释。

你仍然需要经历一个学习过程。重新开始并不会让你突然做的更好。它只是创造一个更大成功率的机会。

InfoQ：对于正在考虑用重建软件代替重构的团队，您有什么建议？

Lagerweij：首先，在发布之前不要尝试和重建任何东西。找到一种方式将新系统和老系统融合，并尽快为你的新系统的用户提供价值。有些东西比如 strangler pattern 或 branch by abstractio 能帮助你。如果你不这样做，你的项目要么在某个时候被取消，要么永远继续下去，但无疑不会有好的结果。

其次，停止糊弄自己。使用一个严格要求规定的工具，比如持续部署，会感觉在项目的表明需求上增加了额外的负担，但是这个规定会帮助你避免陷入和之前同样的陷阱，写出一个完全的新的遗留系统。它会促使你学习新技能，例如持续重构，如何测试驱动你的代码，有多少种不同类型的测试需要被掌握，如何自动化部署，如何处理监控和错误处理。所有的这些都是和到生产的途径变得短和直接紧密相关的。

最后，可能是最重要的，涉及到客户！尽管比通常的要多，重写的诱惑是得到“和老系统做同样的事”的响应，而不是因为客户交流有任何进一步的需要。但是我们需要知道客户现在需要什么。我们也需要知道他不再需要什么。我们都熟悉 80/20 法则关于被使用的功能。你看，重构不仅仅只是发生在代码层面，审查需求，业务流程甚至是商业模式都是同样的重要。

当然，它仍然会有大量的工作要做，但是如果你坚持这些原则，一旦你做到了，你会得到干净的代码，一个学习型团队，和高兴的客户。

Medium开发团队谈架构设计

作者 张天雷

背景

说到底，Medium 是个社交网络，人们可以在这里分享有意思的故事和想法。据统计，目前累积的用户阅读时间已经超过 14 亿分钟，合两千六百年。

我们支持着每个月两千五百万的读者以及每周数以万计的文章发布。我们不想 Medium 的文章以阅读量为成功的依据，而是观点取胜。在 Medium，文章的观点比作者的名头更重要。在这里，对话促进想法，并且很看重文字的力量。

我是 Medium 开发团队的负责人，此前在 Google 工作，负责开发 Google+ 和 Gmail，还创立了 Closure 项目。业余时间我喜欢滑雪跳伞和丛林冒险。

团队介绍

说起团队我非常自豪，这是一群富有好奇心而且想法丰富的天才，大家凑到一块是想做大事的。

团队以跨功能的任务驱动，这样每个人既可以专攻，又可以毫无压力的对整个架构有所贡献。我们的理念就是接触的方面越多，对团队的锻炼越大。更多关于团队的理念见此。

在工作组织方面，我们有着很大的自由度，当然作为一个公司组成，我们还是有季度目标的，并且鼓励敏捷开发模式。我们使用 GitHub 进行 code review 和问题跟踪，用 Google Apps 作为邮件、文档和表单系统。跟很多团队习惯使用 Trello 不同，我们是 Slack 和 slack 机器人的重度用户。

原始架构

最开始的时候，Medium 部署在 EC2 上，用 Node.js 实现，后来公测的时候迁移到了 DynamoDB。

其中有个节点用来处理图片，负责将复杂的处理工作转向 GraphicsMagick。还有一个节点用作后台的 SQS 队列处理。

我们用 SES 处理邮件，S3 做静态元素服务器，CloudFront 做 CDN，nginx 作为反向代理，Datadog 用来监控，Pagerduty 用来告警。

在线编辑器用了 TinyMCE。上线之前我们已经开始使用 Closure 编译器以及部分的 Closure 库，但是模板还是用的 Handlebars。

当前架构

虽然 Medium 表面看起来很简单，但是了解其后台的复杂性后，你会大吃一惊。有人会说，这就是个博客啊，用 Rails 之类的一周就能搞定了。

总之，闲话不多说，我们自底向上介绍以后再做判断。

运行环境

Medium 目前运行在 Amazon 虚拟私有云，使用 Ansible 做系统管理，它支持配置文件模式，我们将文件纳入代码版本管理，这样就可以随时回滚随时掌控。

Medium 的后台是个面向服务的架构，运行了大概二十几个产品服务。划分服务的依据取决于这部分功能的独立性，以及对资源的使用特性。

Medium 的主体仍然是 Node.js 完成，方便前端和后端的代码共享，主要是文章编辑和发布这个过程。Node 大部分时候不错，但阻塞 event 循环的时候会有性能问题。为了缓解，我们在每台机器上启动多个 Node 实例，将对性能要求比较高的任务分配给专门的实例。同时我们还深入 V8 运行时环境查看更加细节的耗时，基本上是 JSON 去串行化的时候的对象具体化耗时较多。

我们还用 Go 语言做了一些辅助服务。因为 Go 非常容易编译打包和发布。相比 Java 语言的冗长罗嗦和虚拟机，Go 语言在类型安全方面做的很到位。就个人习惯来讲，我比较喜欢在团队内部推广强类型语言，因为这类语言能够提高项目的清晰度，不纠结。

目前静态元素大部分是通过 CloudFlare 提供的，还有 5% 通过 Fastly，5% 通过 CloudFront，这么做是为了让两者的缓存得到更新，用于一些紧急的情况。最近我们在应用流量上也使用了 CloudFlare，当时主要是为了防止 DDOS 攻击，但随之而来的性能提升也是我们愿意看到的。

我们使用 Nginx 和 HAProxy 做反向代理和负载均衡，来满足我们所需功能的维恩图。

我们仍然使用 Datadog 来监控，Pagerduty 来告警。现在又增加了 ELK（Elasticsearch、Logstash、Kibana）来进行产品问题调试。

数据库

DynamoDB 仍然是我们的主力数据库，但是用起来也不是毫无问题。目前遇到的比较棘手的是大 V 用户展开和虚拟 event 过程中的热键问题。我们专门在数据库前面做了一个 Redis 缓存集群，来缓解这些问题。到底为开发者优化还是为产品稳定性优化的问题通常会引发争执，我们也一直在尝试中和两者的矛盾。

目前我们开始在存储新数据上使用 Amazon Aurora，它可以提供更灵活的查询和过滤功能。

我们使用 Neo4J 存储 Medium 网络中实体之间的关系，运行在有两个副本的主节点上。用户、文章、标签和收藏都属于图中的节点。边则是在实体创建和用户进行推荐高亮等动作时生成。我们通过在图中游走来过滤和推荐文章。

数据平台

早期我们对数据非常渴望，不断尝试数据分析框架来辅助商业和产品决策。最近我们则是利用同样的框架来反馈产品系统，支持 Explore 等数据驱动功能。

我们采用 Amazon Redshift 作为数据仓库，为生产工具提供可变存储和处理系统。我们持续将诸如用户和文章等核心数据从 Dynamo 导入 Redshift，还将诸如文章被浏览被滚动等 event 日志从 S3 导入 Redshift。

任务通过一个内部调度和监控工具 Conduit 调度。我们用了一个基于断言的调度模型，只有条件满足的时候，任务才会执行。从产品角度来讲，这是不可或缺的：数据制造方应该与数据消费方隔离，还要简化配置，保持系统的可预见和可调试性。

Redshift 的 SQL 检索目前运行不错，但我们时不时需要读取和存储数据，所以后期增加了 Apache Spark 作为 ETL，Spark 具有很好的灵活性和扩展能力。随着产品的推进，估计后面 Spark 会成为我们数据流水线的主要工具。

我们使用 Protocol Buffers 作为 schema 来确保分布式系统的各层次间保持同步，包括移动应用、web 服务和数据仓库等。通过定制化的选项，我们将 schema 标记上更加细化的配置，如带有表名和索引，以及长度等校验约束。

用户也需要保持同步，这样移动端和网页端就可以保持日志的一致性了，同时方便产品科学家们用同样的方式解析字段。我们帮助项目成员从 .proto 文件中生成消息、字段和文档等内容，进而利用所得数据开展研究。

图片服务器

我们的图片服务器现在用 Go 语言实现，采用瀑布型策略来提供处理过的图片。服务器使用 groupcache，是 memcahce 的替代品，可以帮助减轻服务器之间的重复工作。而内存级缓存则是用了一个 S3 的持续缓存。图片的处理是请求来触发的。这给了我们的架构设计师灵活改变图片展示的自由度，为不同平台优化，而且避免了大量的生成不同尺寸图片的操作。

目前 Medium 对图片主要支持放缩和裁剪，但原始版本中还支持颜色清洗和锐化等操作。处理动图很痛苦，具体后续可以写一篇文章来解释。

文本标注

文本标注是个有意思的功能，用了一个小型 Go 服务器，跟 PhantomJS 接口形成渲染进程。

我一直想要把渲染进程换到 Pango，但是在实践过程中，能在 HTML 中摆放图片的能力的确更灵活。而从功能的使用频率来看，这意味着更容易开发和管控。

自定义域名

我们允许用户为其 Medium 文章设置个性化域名。我们想做成单点登录且 HTTPS 全覆盖，因此实现起来颇有难度。我们专门准备了一批 HAProxy 服务器用来管理证书，并向主要应用服务器引导流量。初始化一个域的时候需要一些手动的工作，但是通过与 Namecheap 的定制

化整合，我们将其大部分转换为自动化。证书验证和发布链接由专门服务负责。

网站前端

网页端这块，我们有自主研发的单网页应用框架，使用 Closure 标准库。我们使用 Closure 模板渲染客户端和服务端，然后使用 Closure 编译器来缩减代码并划分模块。编辑器是我们网页端应用最复杂的部分，具体参见 Nick 此前的文章。

iOS

我们的两个应用都是原生的，尽量避免使用网页视图。

在 iOS 上，我们使用了一系列的自建框架，以及系统原生组件。在网络层，我们用 NSURLSession 发起请求，用 Mantle 解析 JSON 并映射到模型。我们还有一层基于 NSKeyedArchiver 的缓冲层。对于将条目渲染为共同主题的列表，我们有一个通用方法，这让我们能够快速为不同类型的内容构建新列表。文章界面是一个定制布局的 UICollectionView。我们使用共享组件来渲染全文界面和预览界面。

应用代码的每一次提交都会编译后推送给 Medium 员工，这样我们能够很快尝试新版本。应用商店的版本是滞后于新版本的，但我们也一直在尝试更快的发布，虽然可能仅仅是几处小更新。

对于测试，我们使用 XCTest 和 OCMock。

Android

在 Android 方面，我们与当前的 SDK 和支持库版本保持一致。我们并没有使用任何复杂的框架，而是倾向于为重复出现的问题构建持续性的模式。我们利用 guava 弥补 Java 中所有的缺失。另一方面来讲，我们也倾向于使用第三方库来解决特别的问题。我们还利用 protocol buffers 定义了 API，用以生成应用中的对象。

我们利用 mockito 和 robolectric。我们会开发一些高层测试来运转 activity 和 poke：刚添加 screen 或要重构的时候，先创建一些基本的版本，随着我们复现 bug 它们也会进化。我们还会开发一些底层测试，来检测一个特定的类：随着新功能的增加我们会创建测试，这能够帮助我们思考和设计底层是如何交互的。

每个提交都会作为 alpha 版本自动推送到 play 商店，然后到 Medium 员工（包括我们的 Hatch，Medium 内部版）。推送大部分发生在周五，我们会把 alpha 版本发送给测试小组，请他们用整个周末进行测试。然后，周一我们会从 beta 版推进至正式产品版。因为最近一批代码总是随时可以推送，因此一旦发现很严重的 bug，我们就可以立即修复正式产品版。当我们怀疑某些新功能的时候，可以给测试小组更长的时间。开发比较亢奋的时候，也可能发布地更加频繁。

AB 测试及其他

我们所有的客户端都用了服务器端提供的功能标记，称为 variants，用于 A|B 测试以及指导未完成功能的开发。

剩下还有一些框架相关的内容我没有提及：Algolia 让我们在搜索相关功能上快速迭代，SendGrid 处理邮件，Urban Airship 用来发送提醒，SQS 用来处理队列，Bloomd 用作布隆过滤器，PubSubHubbub 和 Superfeedr 用作提供 RSS 等等。

编译、测试和部署

我们积极拥抱持续集成技术，随时随地准备发布，使用 Jenkins 来负责相关事宜。

我们曾经使用 Make 作为编译系统，但是后来迁移到 Pants。

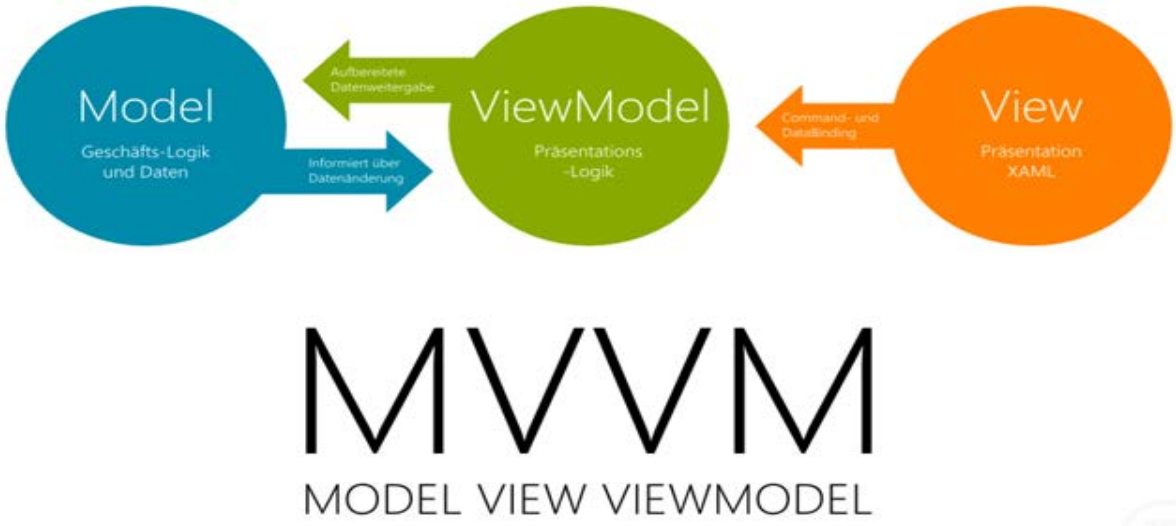
测试方面我们采用单元测试和 HTTP 层面功能测试两者结合的方式。所有提交的代码都需要通过测试才能够合并。我们跟 Box 团队合作，利用 Cluster Runner 来分布式运行测试，保证效率，而且能够和 GitHub 很好的整合在一起。

我们大概不到 15 分钟就可以把某阶段的系统部署，顺利编译通过，留作正式产品的备选。主应用服务器通常一天要部署五次，多的时候十次。
我们采用蓝绿部署。正式产品版本的流量发送给一个 canary 实例，发布进程会监控部署过程的错误率，必要时候通过调整内部 DNS 回滚。

面向未来

到此，讲了足够多的干货！为了重构产品，获得更好的阅读体验，还有很长的路要走。我们仍然在努力为作者和发布者设计更多的功能。打比方来讲，线上阅读还是一片绿地，面对它有着无限可能，我们始终抱着开放的心态设计和实现功能。未来我们会努力用各种功能为用户提供高质量内容和价值。

被误解的MVC和被神化的MVVM



作者 唐巧

被误解的 MVC

MVC 的历史

MVC 全称是 Model View Controller，是模型（model）－视图（view）－控制器（controller）的缩写。它表示的是一种常见的客户端软件开发框架。

MVC 的概念最早出现在二十世纪八十年代的施乐帕克实验室中（对，就是那个发明图形用户界面和鼠标的实验室），当时施乐帕克为 Smalltalk 发明了这种软件设计模式。

现在，MVC 已经成为主流的客户端编程框架，在 iOS 开发中，系统为我们实现好了公共的视图类：UIView，和控制器类：UIViewController。大多数时候，我们都需要继承这些类来实现我们的程序逻辑，因此，我们几乎逃避不开 MVC 这种设计模式。

但是，几十年过去了，我们对于 MVC 这种设计模式真的用得好吗？其实不是的，MVC 这种分层方式虽然清楚，但是如果使用不当，很可能让大量代码都集中在 Controller 之中，让 MVC 模式变成了 Massive View Controller 模式。

Controller 的臃肿问题何解

很多人试图解决 MVC 这种架构下 Controller 比较臃肿的问题。我还记得半年前 InfoQ 搞了一次移动座谈会，当时 BeeFramework 和 Samurai-Native 的作者老郭问了我一句话：

「什么样的内容才应该放到 Controller 中？」。但是当时因为时间不够，我没能展开我的观点，这次正好在这里好好谈谈我对于这个问题的想法。

我们来看看 MVC 这种架构的特点。其实设计模式很多时候是为了 Don't repeat yourself 原则来做的，该原则要求能够复用的代码要尽量复用，来保证重用。在 MVC 这种设计模式中，我们发现 View 和 Model 都是符合这种原则的。

对于 View 来说，你如果抽象得好，那么一个 App 的动画效果可以很方便地移植到别的 App 上，而 Github 上也有很多 UI 控件，这些控件都是在 View 层做了很好的封装设计，使得它能够方便地开源给大家复用。

对于 Model 来说，它其实是用来存储业务的数据的，如果做得好，它也可以方便地复用。比如我当时在做有道云笔记 iPad 版的时候，我们就直接和 iOS 版复用了所有的 Model 层的代码。在创业做猿题库客户端时，iOS 和 iPad 版的 Model 层代码再次被复用上了。当然，因为和业务本身的数据意义相关，Model 层的复用大多数是在一个产品内部，不太可能像 View 层那样开源给社区。

说完 View 和 Model 了，那我们想想 Controller，Controller 有多少可以复用的？我们写完了一个 Controller 之后，可以很方便地复用它吗？结论是：非常难复用。在某些场景下，我们可能可以用 addSubViewController 之类的方式复用 Controller，但它的复用场景还是非常非常少的。

如果我们能够意识到 Controller 里面的代码不便于复用，我们就能知道什么代码应该写在 Controller 里面了，那就是那些不能复用的代码。在我看来，Controller 里面就只应该存放这些不能复用的代码，这些代码包括：

- 在初始化时，构造相应的 View 和 Model。
- 监听 Model 层的事件，将 Model 层的数据传递到 View 层。
- 监听 View 层的事件，并且将 View 层的事件转发到 Model 层。

如果 Controller 只有以上的这些代码，那么它的逻辑将非常简单，而且也会非常短。

但是，我们却很难做到这一点，因为还是有很多逻辑我们不知道写在哪里，于是就都写到了 Controller 中了，那我们接下来就看看其它逻辑应该写在哪里。

如何对 ViewController 瘦身

objc.io 是一个非常有名的 iOS 开发博客，它上面的第一课《Lighter View Controllers》上就讲了很多这样的技巧，我们先总结一下它里面的观点：

- 将 UITableView 的 Data Source 分离到另外一个类中。
- 将数据获取和转换的逻辑分别到另外一个类中。
- 将拼装控件的逻辑，分离到另外一个类中。

你想明白了吗？其实 MVC 虽然只有三层，但是它并没有限制你只能有三层。所以，我们可以将 Controller 里面过于臃肿的逻辑抽取出来，形成新的可复用模块或架构层次。

我个人对于逻辑的抽取，有以下总结。

将网络请求抽象到单独的类中

新手写代码，直接就在 Controller 里面用 AFNetworking 发一个请求，请求的完数据直

接就传递给 View。入门一些的同学，知道把这些请求代码移到另外一个静态类里面。但是我觉得还不够，所以我建议将每一个网络请求直接封装成类。

把每一个网络请求封装成对象其实是使用了设计模式中的 Command 模式，它有以下好处：

- 将网络请求与具体的第三方库依赖隔离，方便以后更换底层的网络库。实际上我们公司的 iOS 客户端最初是基于 ASIHttpRequest 的，我们只花了两天，就很轻松地切换到了 AFNetworking。
- 方便在基类中处理公共逻辑，例如猿题库的数据版本号信息就统一在基类中处理。
- 方便在基类中处理缓存逻辑，以及其它一些公共逻辑。
- 方便做对象的持久化。

大家如果感兴趣，可以看我们公司开源的 iOS 网络库：YTKNetwork。它在这种思考的指导下，不但将 Controller 中的代码瘦身，而且进一步演化和加强，现在它还支持诸如复杂网络请求管理，断点续传，插件机制，JSON 合法性检查等功能。

这部分代码从 Controller 中剥离出来后，不但简化了 Controller 中的逻辑，也达到了网络层的代码复用的效果。

将界面的拼装抽象到专门的类中

新手写代码，喜欢在 Controller 中把一个个 UILabel ，UIButton，UITextField 往 self.view 上用 addSubView 方法放。我建议大家可以用两种办法把这些代码从 Controller 中剥离。

方法一：构造专门的 UIView 的子类，来负责这些控件的拼装。这是最彻底和优雅的方式，不过稍微麻烦一些的是，你需要把这些控件的事件回调先接管，再都一一暴露回 Controller。

方法二：用一个静态的 Util 类，帮助你做 UIView 的拼装工作。这种方式稍微做得不太彻底，但是比较简单。

对于一些能复用的 UI 控件，我建议用方法一。如果项目工程比较复杂，我也建议用方法一。如果项目太紧，另外相关项目的代码量也不多，可以尝试方法二。

构造 ViewModel

谁说 MVC 就不能用 ViewModel 的？MVVM 的优点我们一样可以借鉴。具体做法就是将 ViewController 给 View 传递数据这个过程，抽象成构造 ViewModel 的过程。

这样抽象之后，View 只接受 ViewModel，而 Controller 只需要传递 ViewModel 这么一行代码。而另外构造 ViewModel 的过程，我们就可以移动到另外的类中了。

在具体实践中，我建议大家专门创建构造 ViewModel 工厂类，参见工厂模式。另外，也可以专门将数据存取都抽将到一个 Service 层，由这层来提供 ViewModel 的获取。

专门构造存储类

刚刚说到 ViewModel 的构造可以抽奖到一个 Service 层。与此相应的，数据的存储也应该由专门的对象来做。在小猿搜题项目中，我们由一个叫 UserAgent 的类，专门来处理本地数据的存取。

数据存取放在专门的类中，就可以针对存取做额外的事情了。比如：

- 对一些热点数据增加缓存

- 处理数据迁移相关的逻辑

如果要做得更细，可以把存储引擎再抽象出一层。这样你就可以方便地切换存储的底层，例如从 sqlite 切换到 key-value 的存储引擎等。

小结

通过代码的抽取，我们可以将原本的 MVC 设计模式中的 ViewController 进一步拆分，构造出 网络请求层、ViewModel 层、Service 层、Storage 层等其它类，来配合 Controller 工作，从而使 Controller 更加简单，我们的 App 更容易维护。

另外，不知道大家注意到没，其实 Controller 层是非常难于测试的，如果我们能够将 Controller 瘦身，就可以更方便地写 Unit Test 来测试各种与界面的无关的逻辑。移动端自动化测试框架都不太成熟，但是将 Controller 的代码抽取出来，是有助于我们做测试工作的。

希望本文能帮助大家掌握正确使用 MVC 的姿势，在下一节里，我将分享一下我对 MVVM 的看法。

被神化的 MVVM

MVVM 的历史

MVVM 是 Model-View-ViewModel 的简写。

相对于 MVC 的历史来说，MVVM 是一个相当新的架构，MVVM 最早于 2005 年被微软的 WPF 和 Silverlight 的架构师 John Gossman 提出，并且应用在微软的软件开发中。当时 MVC 已经被提出了 20 多年了，可见两者出现的年代差别有多大。

MVVM 在使用当中，通常还会利用双向绑定技术，使得 Model 变化时，ViewModel 会自动更新，而 ViewModel 变化时，View 也会自动变化。所以，MVVM 模式有些时候又被称作：model-view-binder 模式。

具体在 iOS 中，可以使用 KVO 或 Notification 技术达到这种效果。

MVVM 的神化

在使用中，我发现大家对于 MVVM 以及 MVVM 衍生出来的框架（比如 ReactiveCocoa）有一种「敬畏」感。这种「敬畏」感某种程度上就像对神一样，这主要表现在我没有听到大家对于 MVVM 的任何批评。

我感觉原因首先是 MVVM 并没有很大程度上普及，大家对于新技术一般都不熟，进而不敢妄加评论。另外，ReactiveCocoa 本身上手的复杂性，也让很多人感觉到这种技术很高深难懂，进而加重了大家对它的「敬畏」。

MVVM 的作用和问题

MVVM 在实际使用中，确实能够使得 Model 层和 View 层解耦，但是如果你需要实现 MVVM 中的双向绑定的话，那么通常就需要引入更多复杂的框架来实现了。

对此，MVVM 的作者 John Gossman 的批评应该是最为中肯的。John Gossman 对 MVVM 的批评主要有两点。

第一点：数据绑定使得 Bug 很难被调试。你看到界面异常了，有可能是你 View 的代码有 Bug，也可能是 Model 的代码有问题。数据绑定使得一个位置的 Bug 被快速传递到别的位置，要定位原始出问题的地方就变得不那么容易了。

第二点：对于过大的项目，数据绑定需要花费更多的内存。

某种意义上来说，我认为就是数据绑定使得 MVVM 变得复杂和难用了。但是，这个缺点同时也被很多人认为是优点。

ReactiveCocoa

函数式编程（Functional Programming）和响应式编程（React Programming）也是当前很火的两个概念，它们的结合可以很方便地实现数据的绑定。于是，在 iOS 编程中，ReactiveCocoa 横空出世了，它的概念都非常新，包括：

- 函数式编程（Functional Programming），函数也变成一等公民了，可以拥有和对象同样的功能，例如当成参数传递，当作返回值等。看看 Swift 语言带来的众多函数式编程的特性，就你知道这多 Cool 了。
- 响应式编程（React Programming），原来我们基于事件（Event）的处理方式都弱了，现在是基于输入（在 ReactiveCocoa 里叫 Signal）的处理方式。输入还可以通过函数式编程进行各种 Combine 或 Filter，尽显各种灵活的处理。
- 无状态（Stateless），状态是函数的魔鬼，无状态使得函数能更好地测试。
- 不可修改（Immutable），数据都是不可修改的，使得软件逻辑简单，也可以更好地测试。

哇，所有这些都太 Cool 了。当我看到的时候，我都鸡冻了！

我们应该客观评价 MVVM 和 ReactiveCocoa

但是但是，我突然想到，我好象只需要一个 ViewModel 而已，我完全可以简单地做一个 ViewModel 的工厂类或 Service 类就可以了，为什么要引入这么多框架？现有的 MVC 真的有那么大的问题吗？

直到现在，ReactiveCocoa 在国内外还都是在小众领域，没有被大量接受成为主流的编程框架。不只是在 iOS 语言，在别的语言中，例如 Java 中的 RxJava 也同样没有成为主流。

我在这里，不是想说 ReactiveCocoa 不好，也不是想说 MVVM 不好，而是想让大家都能够有一个客观的认识。ReactiveCocoa 和 MVVM 不应该被神化，它是一种新颖的编程框架，能够解决旧有编程框架的一些问题，但是也会带来一些新问题，仅此而已。如果不能使好的驾驭 ReactiveCocoa，同样会造成 Controller 代码过于复杂，代码逻辑不易维护的问题。

总结

有一些人总是追赶着技术，有什么新技术不管三七二十一立马就用，结果被各种坑。

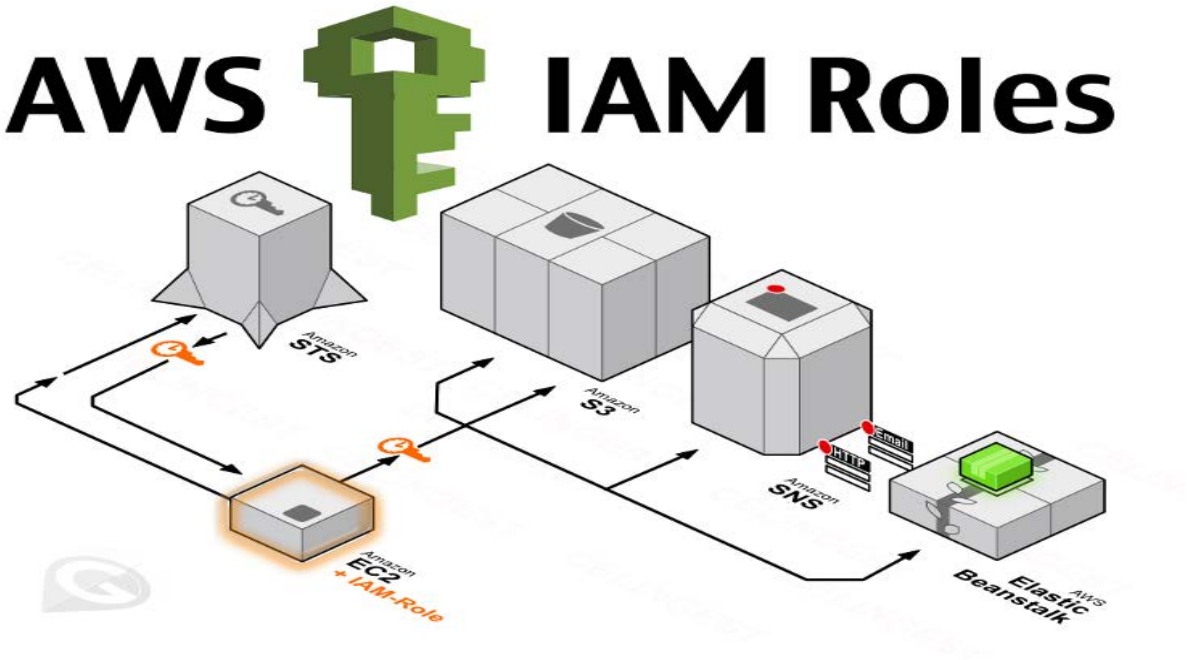
又有一些人，总是担心新技术带来的技术风险，不愿意学习。结果现在还有人在用 MRC 手动管理引用计数。

而我想说，我们需要保持的是一个拥抱变化的心，以及理性分析的态度。在新技术的面前，不盲从，也不守旧，一切的决策都应该建立在认真分析的基础上，这样才能应对技术的变化。

AWS Summit

AWS技术峰会2015 · 上海

AWS系列：深入了解IAM和访问控制



作者 陈天

编者按：本文系 InfoQ 中文站向陈天的约稿，这是 AWS 系列文章的第一篇。以后会有更多文章刊出，但并无前后依赖的关系，每篇都自成一体。读者若要跟随文章来学习 AWS，应该至少注册了一个 AWS 账号，事先阅读过当期所介绍服务的简介，并在 AWS management console 中尝试使用过该服务。否则，阅读的效果不会太好。

写在前面：访问控制，换句话说，谁能在什么情况下访问哪些资源或者操作，是绝大部分应用程序需要仔细斟酌的问题。作为一个志存高远的云服务提供者，AWS 自然也在访问控制上下了很大的力气，一步步完善，才有了今日的 IAM：Identity and Access Management。如果你要想能够游刃有余地使用 AWS 的各种服务，在安全上的纰漏尽可能地少，那么，首先需要先深入了解 IAM。

基本概念

按照 AWS 的定义：

IAM enables you to control who can do what in your AWS account.

它提供了用户（users）管理、群组（groups）管理、角色（roles）管理和权限（permissions）管理等供 AWS 的客户来管理自己账号下面的资源。

1、首先说用户（users）。在 AWS 里，一个 IAM user 和 unix 下的一个用户几乎等价。你可以创建任意数量的用户，为其分配登录 AWS management console 所需要的密码，以及使用 AWS CLI（或其他使用 AWS SDK 的应用）所需要的密钥。你可以赋予用户管理员的权限，使其能够任意操作 AWS 的所有服务，也可以依照 Principle of least privilege，只授权合适的权限。下面是使用 AWS CLI 创建一个用户的示例：

```
saws> aws iam create-user --user-name tyrchen
{
    "User": {
        "CreateDate": "2015-11-03T23:05:05.353Z",
        "Arn": "arn:aws:iam::<ACCOUNT-ID>:user/tyrchen",
        "UserName": "tyrchen",
        "UserId": "AIDAISBVIGXYRRQLDDC3A",
        "Path": "/"
    }
}
```

当然，这样创建的用户是没有任何权限的，甚至无法登录，你可以用下面的命令进一步为用户关联群组，设置密码和密钥：

```
saws> aws iam add-user-to-group --user-name --group-name
saws> aws iam create-login-profile --user-name --password
saws> aws iam create-access-key --user-name
```

2、群组（groups）也等同于常见的unix group。将一个用户添加到一个群组里，可以自动获得这个群组所具有的权限。在一家小的创业公司里，其AWS账号下可能会建立这些群组：

- Admins：拥有全部资源的访问权限
- Devs：拥有大部分资源的访问权限，但可能不具备一些关键性的权限，如创建用户
- Ops：拥有部署的权限
- Stakeholders：拥有只读权限，一般给manager查看信息之用

创建一个群组很简单：

```
saws> aws iam create-group --group-name stakeholders
{
    "Group": {
        "GroupName": "stakeholders",
        "GroupId": "AGPAIVGNNEGMEPLHXY6JU",
        "Arn": "arn:aws:iam::<ACCOUNT-ID>:group/stakeholders",
        "Path": "/",
        "CreateDate": "2015-11-03T23:15:47.021Z"
    }
}
```

然而，这样的群组没有任何权限，我们还需要为其添加policy：

```
saws> aws iam attach-group-policy --group-name --policy-arn
```

在前面的例子和这个例子里，我们都看到了ARN这个关键字。ARN是Amazon Resource Names的缩写，在AWS里，创建的任何资源有其全局唯一的ARN。ARN是一个很重要的概念，它是访问控制可以到达的最小粒度。在使用AWS SDK时，我们也需要ARN来操作对应的资源。

policy是描述权限的一段JSON文本，比如AdministratorAccess这个policy，其内容如下：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
```

用户或者群组只有添加了相关的policy，才会有相应的权限。

3、角色（roles）类似于用户，但没有任何访问凭证（密码或者密钥），它一般被赋予某个资源（包括用户），使其临时具备某些权限。比如说一个EC2实例需要访问DynamoDB，我们可以创建一个具有访问DynamoDB权限的角色，允许其被EC2 Service代入（AssumeRule），然后创建EC2的instance-profile使用这个角色。这样，这个EC2实例就可以访问DynamoDB了。当然，这样的权限控制也可以通过在EC2的文件系统里添加AWS配置文件设置某个用户的密钥（AccessKey）来获得，但使用角色更安全更灵活。角色的密钥是动态创建的，更新和失效都无须特别处理。想象一下如果你有成百上千个EC2实例，如果使用某个用户的密钥来访问AWS SDK，那么，只要某台机

器的密钥泄漏，这个用户的密钥就不得不手动更新，进而手动更新所有机器的密钥。这是很多使用 AWS 多年的老手也会犯下的严重错误。

4、最后是权限（permissions）。AWS 下的权限都通过 policy document 描述，就是上面我们给出的那个例子。policy 是 IAM 的核心内容，我们稍后详细介绍。

每年的 AWS re:invent 大会，都会有一个 session：Top 10 AWS IAM Best Practices，感兴趣的读者可以去 YouTube 搜索。2015 年的 top 10（top 11）如下：

- users: create individual users
- permissions: Grant least priviledge
- groups: manage permissions with groups
- conditions: restrict priviledged access further with conditions
- auditing: enable cloudTrail to get logs of API calls
- password: configure a strong password policy
- rotate: rotate security credentials regularly.
- MFA: enable MFA (Multi-Factor Authentication) for priviledged users
- sharing: use IAM roles to share access
- roles: use IAM roles for EC2 instances
- root: reduce or remove use of root

这 11 条 best practices 很清晰，我就不详细解释了。

按照上面的原则，如果一个用户只需要访问 AWS management console，那么不要为其创建密钥；反之，如果一个用户只使用 AWS CLI，则不要为其创建密码。一切不必要的，都是多余的——这就是安全之道。

使用 policy 做访问控制

上述内容你若理顺，IAM 就算入了门。但真要把握好 IAM 的精髓，需要深入了解 policy，以及如何撰写 policy。

前面我们看到，policy 是用 JSON 来描述的，主要包含 Statement，也就是这个 policy 拥有的权限的陈述，一言以蔽之，即：谁在什么条件下能对哪些资源的哪些操作进行处理。也就是所谓的撰写 policy 的 PARCE 原则：

- Principal：谁
- Action：哪些操作
- Resource：哪些资源
- Condition：什么条件
- Effect：怎么处理（Allow/Deny）

我们看一个允许对 S3 进行只读操作的 policy：

```
001 {
002   “Version”: “2012-10-17”,
003   “Statement”: [
004     {
005       “Effect”: “Allow”,
006       “Action”: [
007         “s3:Get*”,
008         “s3:List*”
009       ],
010       “Resource”: “*”
011     }
012   ]
013 }
```

其中，Effect 是 Allow，允许 policy 中所有列出的权限，Resource 是 *，代表任意 S3 的资源，Action 有两个：s3:Get* 和 s3:List*，允许列出 S3 下的资源目录，及获取某个具体的 S3 Object。

在这个 policy 里，Principal 和 Condition 都没有出现。如果对资源的访问没有任何附加条

件，是不需要 Condition 的；而这条 policy 的使用者是用户相关的 principal（users, groups, roles），当其被添加到某个用户身上时，自然获得了 principal 的属性，所以这里不必指明，也不能指明。

所有的 IAM managed policy 是不需要指明 Principal 的。这种 policy 可以单独创建，在需要的时候可以被添加到用户、群组或者角色身上。另一大类 policy 是 Resource policy，它们不能单独创建，只能依附在某个资源之上（所以也叫 inline policy），这时候，需要指明 Principal。比如，当我希望对一个 S3 bucket 使能 Web hosting 时，这个 bucket 里面的对象自然是要允许外界访问的，所以需要如下的 inline policy：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource":
    "arn:aws:s3:::corp-fs-web-bucket/*"
        }
    ]
}
```

这里，我们对于 arn:aws:s3:::corp-fs-web-bucket/* 这个资源的 s3:GetObject，允许任何人访问（Principal: *）。

有时候，我们希望能更加精细地控制用户究竟能访问资源下的哪些数据，这个时候，可以使用 Condition。比如对一个叫 tyrchen 的用户，只允许他访问 personal-files 这个 S3 bucket 下和他有关的目录：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:ListBucket"
            ],
            "Effect": "Allow",
            "Resource": [
    "arn:aws:s3:::personal-files"
            ],
            "Condition": {
                "StringLike": {
                    "s3:prefix": [
                        "tyrchen/*"
                    ]
                }
            }
        },
        {
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Effect": "Allow",
            "Resource": [
    "arn:aws:s3:::personal-files/
    tyrchen/*"
            ]
        }
    ]
}
```

这里我们用到了 StringLike 这个 Condition，只有当访问的 s3:prefix 满足 tyrchen/* 时，才为真。我们还可以使用非常复杂的 Condition：

```
"Condition": {
    "IPAddress": {"aws:SourceIP":
    ["10.0.0.0/8", "4.4.4.4/32"]},
    "StringEquals": {"ec2:ResourceTag/
    department": "dev"}
}
```

在一条 Condition 下并列的若干个条件间是 and 的关系，这里 IPAddress 和 StringEquals 这两个条件必须同时成立；在某个条件内部则是 or 的关系，这里 10.0.0.0/8 和 4.4.4.4/32 任意一个源 IP 都可以访问。这个条件最终的意思是：对于一个 EC2 实例，如果其 department 标签是 dev，且访问的源 IP 是 10 网段的内网地址或者 4.4.4.4/32 这个外网地

址，则 Condition 成立。

讲完了 Condition，我们再回到之前的 policy。

这条 policy 里有两个 statement，前一个允许列出 arn:aws:s3:::personal-files 下 prefix 是 tyrchen/* 里的任何对象；后一个允许读写 arn:aws:s3:::personal-files/tyrchen/* 里的对象。注意这个 policy 不能写成：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:ListObject",
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Effect": "Allow",
            "Resource": [
    "arn:aws:s3:::personal-files/tyrchen/*"
            ]
        }
    ]
}
```

因为这样的话，用户在 AWS management console 连在 personal-files 下列出 /tyrchen 这个根目录的机会都没有了，自然也无法进行后续的操作。这样的 policy 其权限是不完备的。

上面的 policy 可以被添加到用户 tyrchen 身上，这样他就可以访问自己的私人目录；如果我们创建一个新用户叫 lindsey，也想做类似处理，则需要再创建一个几乎一样的 policy，非常不符合 DRY（Don't Repeat Yourself）原则。好在 AWS 也考虑到了这一点，它支持 policy variable，允许用户在 policy 里使用 AWS 预置的一些变量，比如 ${aws:username}。上述的 policy 里，把 tyrchen 替换为 ${aws:username}，就变得通用多了。

以上是 policy 的一些基础用法，下面讲讲 policy 的执行规则，它也是几乎所有访问控制方案的通用规则（见图 1）：

- 默认情况下，一切资源的一切行为的访问都是 Deny
- 如果在 policy 里显式 Deny，则最终的结果是 Deny
- 否则，如果在 policy 里是 Allow，则最终结果是 Allow
- 否则，最终结果是 Deny

什么情况下我们会用到显式 Deny 呢？请看下面的例子：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:*",
        "s3:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:AWS-REGION-IDENTIFIER:ACCOUNT-ID-WITHOUT-HYPHENS:table/EXAMPLE-TABLE-NAME",
        "arn:aws:s3:::EXAMPLE-BUCKET-NAME",
        "arn:aws:s3:::EXAMPLE-BUCKET-NAME/*"
      ]
    },
    {
      "Effect": "Deny",
      "NotAction": [
        "dynamodb:*",
        "s3:*"
      ],
      "NotResource": [
        "arn:aws:dynamodb:AWS-REGION-IDENTIFIER:ACCOUNT-ID-WITHOUT-HYPHENS:table/EXAMPLE-TABLE-NAME",
        "arn:aws:s3:::EXAMPLE-BUCKET-NAME",
        "arn:aws:s3:::EXAMPLE-BUCKET-NAME/*"
      ]
    }
  ]
}
```

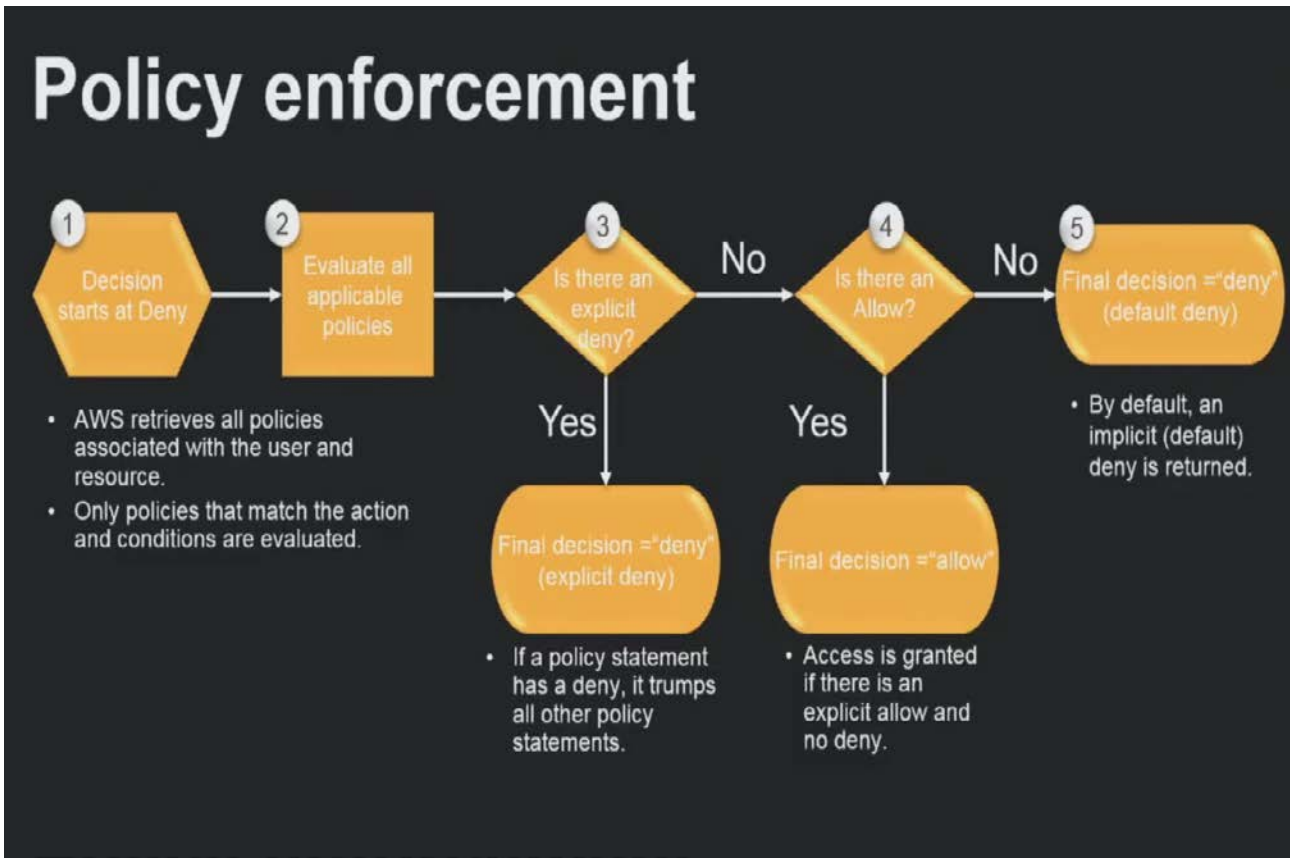


图 1

在这个例子里，我们只允许用户访问 DynamoDB 和 S3 中的特定资源，除此之外一律不允许访问。我们知道一个用户可以有多重权限，属于多个群组。所以上述 policy 里的第一个 statement 虽然规定了用户只能访问的资源，但别的 policy 可能赋予用户其他资源的访问权限。所以，我们需要第二条 statement 封死其他的可能。按照之前的 policy enforcement 的规则，只要看见 Deny，就是最终结果，不会考虑其他 policy 是否有 Allow，这样杜绝了一些隐性的后门，符合 Principle of least privilege。

我们再看一个生产环境中可能用得着的例子，来证明 IAM 不仅「攘内」，还能「安外」。假设我们是一个手游公司，使用 AWS Cognito 来管理游戏用户。每个游戏用户的私人数据放置于 S3 之中。我们希望 congito user 只能访问他们各自的目录，IAM policy 可以定义如下：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:ListBucket"],
      "Resource":
    ["arn:aws:s3:::awesome-game"],
      "Condition": {"StringLike":
    {"s3:prefix": ["cognito/*"]}}
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3:::awesome-
    game/cognito/${cognito-identity.
    amazonaws.com:sub}",
        "arn:aws:s3:::awesome-
    gamecognito/${cognito-identity.
    amazonaws.com:sub}/*"
      ]
    }
  ]
}
```

最后，讲一下如何创建 policy。很简单：

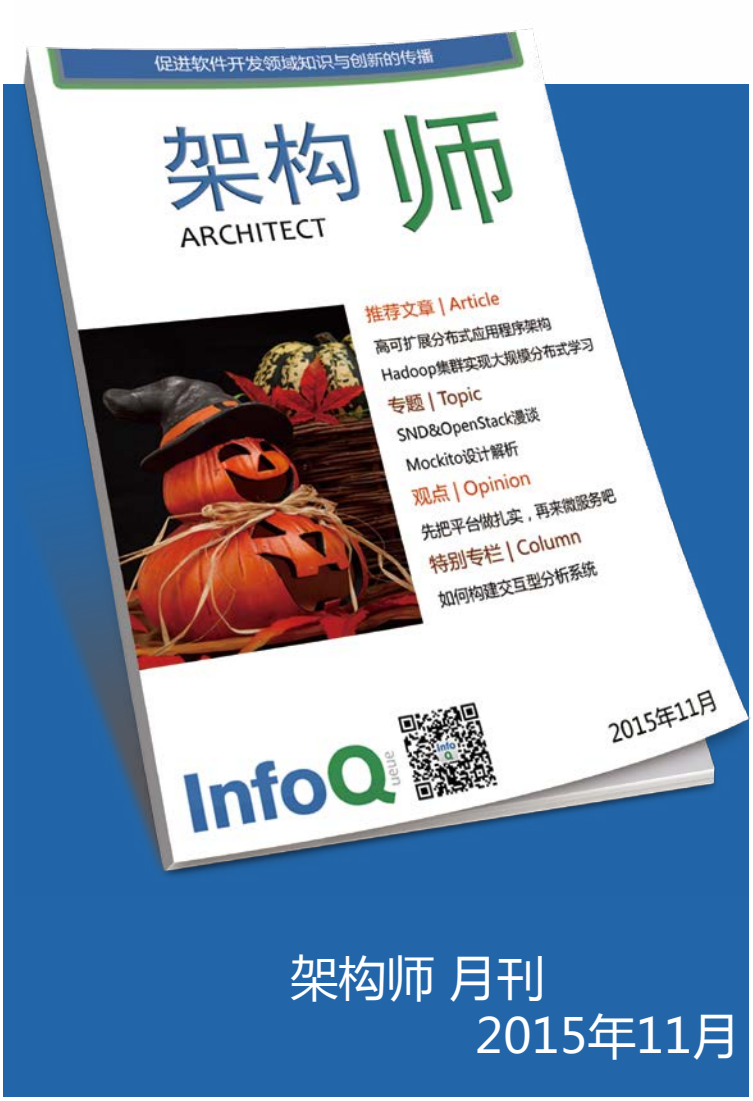
```
001 $ aws iam create-policy --policy-name
      --policy-document
```

其中 policy-document 就是如上所示的一个个 JSON 文件。

参考文档

- IAM policy overview
- policy variables
- policy evaluation logic

InfoQ 中文站　2015迷你书



架构师 月刊
2015年11月

本期主要内容：高可扩展分布式应用程序的架构原则，雅虎如何在Hadoop集群上实现大规模分布式深度学习，SND&OpenStack漫谈，反模式的经典　--Mockito设计解析，先把平台做扎实，再来微服务吧，如何构建交互型分析系统

开源启示录
第二季

开源软件的未来在于建立一个良性循环，以参与促进繁荣，以繁荣促进参与。在这里，我们为大家呈现本期迷你书，在揭示些许开源软件规律的之外，更希望看到有更多人和企业参与到开源软件中来。

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。

云生态专刊

《云生态专刊》是InfoQ为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。