

架构师

ARCHITECT

日刊 |

热点

Kotlin生态调查结果出炉：
超过6成的开发者用过Kotlin了



Geekbang | 极客邦科技

InfoQ

CONTENTS / 目录

热点 | Hot

Kotlin 生态调查结果出炉：超过 6 成的开发者用过 Kotlin 了

推荐文章 | Article

Stream 从 Python 切换到 Go 的原因

专题 | Topic

腾讯大规模分布式机器学习系统无量是如何进行技术选型的？

GitHub 的 MySQL 高可用性实践

运满满的技术架构演进之路

百度智能运维的技术演进之路

特别专栏 | Column

面向大规模 AI 在线推理的可靠性设计



架构师 2018 年 8 月刊

本期主编 杜小芳

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

30+ 技术专家 带你技术进阶

进阶的路上，高手引路比盲目直冲更重要

- 技术能力进阶的必修课
- 20余类硬技能，365天70+优质课程
- 全方位拆解场景化实战案例
- 帮你稳扎稳打从入门到精通

30 + 一线技术专家及学者



扫 码 即 可 解 锁





全球软件开发大会

▶ 聚焦

- 互联网高可用架构
- 国际化互联网业务架构
- 工程师个人成长与技术领导力
- 架构设计
- 后移动互联网时代的技术思考与实践
- 大规模基础设施DevOps探索
- 硅谷人工智能
- 人工智能与业务实践
- Java生态与创新
- 大数据系统架构
- 区块链技术与应用
- 前端新趋势
- 深度学习技术与应用
- 微服务架构 & Serverless
- 产品经理必修之用户细分与产品定位

▶ 实践

Facebook / 硅谷公司的互联网计算性能优化经验谈

LinkedIn / 推荐系统：提升用户增长与参与的利器

Uber / 核心Trip Flow容量管理

Confluent /Apache Kafka / 从0.8到2.0：那些年我们踩过的坑

快手 / 如何快速打造高稳定千亿级别对象存储平台

微软 / 集成AI开发平台实践

会议：2018年10月18–20日

培训：2018年10月21–22日

地址：上海·宝华万豪酒店

8折报名中，立减1360元

团购享更多优惠，

截至2018年8月19日

大咖助阵



专题：硅谷人工智能
夏磊 / LinkedIn高级工程师，
湾区同学技术沙龙Board Member



专题：Java生态与创新
张建锋 / 永源中间件 共同创始人



专题：互联网高可用架构
吴其敏 / 平安银行
零售网络金融事业部首席架构师



专题：研发效率提升
徐毅 / 华为 技术专家



专题：产品经理必修之用户细分与产品定位
袁店明 / Dell EMC
敏捷与精益创业咨询师

.....

分享嘉宾



张翔
三一重工
所长兼项目经理



庄振运
Facebook
计算机性能高级工程师



邹欣
微软亚洲研究院
首席研发经理



李欣 (Bruce Li)
Paypal
PayPal Risk Infra
Director level architect



Hien Luu
LinkedIn
工程经理



孙彦
Pinterest
视觉搜索团队高级工程师



施磊
Airbnb
技术经理



俞戴龙
Red Pulse
高级架构师

.....



100+技术专家的实践分享

联系我们：

热线：010-84782011 微信：qcon-0410

卷首语

沃森于 AI

DataPipeline创始人&CEO 陈诚

朋友圈最近看到一篇文章——《IBM沃森错开致命药，国内67家医院在用，秘密文件曝光严重bug》。虽然只是测试病例，但看上去沃森相当地不靠谱，给有出血症状的癌症病人开了容易导致出血的药品，严重时可致患者死亡。沃森是IBM花了150亿美元培养的AI学霸，成绩不尽如人意，部门不能盈利，失望在所难免。然而AI正如一个学习能力超强的小孩子，需要反复研习各种病例、新药，来提高判断的准确度。

无独有偶，最近也有另一篇报道说亚马逊AI的人脸识别系统遭到质疑，从535位美国参众两院议员中识别出28名“罪犯”，一时引起大量对于公共安全和执法准确率的担忧。亚马逊迅速在回应中提到在对Rekognition 系统人脸识别API的默认置信阈值被设置成了80%，建议应该使用99%。然而这样设置的话确实可以大量减少“误判”，但是却会产生更多的“漏判”，结果仍然未必会让人满意。

大众在AlphaGo之后对于AI的想象都是有点像孙悟空，在菩提祖师那里混了7年，而后一夜悟道，从此速度一日千里。然而任何创新的技术都需要快速迭代、不断更新。我们对于新技术既不需要盲目追捧，也不用质

疑攻击。人工智能之所以是人工的，不就是因为我们人类善于思考，可以辩证客观地看待问题吗？

数据的质量和模型都会对AI结果产生巨大的影响，这次我们来聊聊数据。AI的判断，高度依赖于我们输入给它的“知识”，如果输入数据不够准确、不够完整，得出的判断不靠谱也在意料之中。大部分数据分析师，每天都要花费大量的时间和精力去“整”数据，无论是数据质量，还是数据的完整性，一致性都差强人意，最高精尖的工程师也要花大力气去做最基础的数据工作，AI能不能不以事小而不为，帮我们先解决了数据的事儿？

AI不是万能的，数据不是万能的，没有数据是万万不能的。我们对于世界的认识和认知，进行抽象提取，而后成为知识。数据是这一切的基础。数据失之毫厘，AI的结果差之千里。数据工程师和AI算法工程师是背靠背的一对兄弟，相辅相成，谁也离不开谁。

创办DataPipeline之初，我们以为数据的事儿比起AI，门槛没有那么高，但是适用面广，然而越做越敬畏。如果AI最终的目的是帮助人类解决各种各样的问题，那么数据就是这个大厦最坚实的基础，基础不牢，地动山摇。我们不能只要顶层的无敌视野，却不愿意为打地基付出汗水和努力。数据的工作繁杂，日复一日，各种重复，远不如AI的高大上，然而吃不饱肚子，怎么追求精神上的富足？

回过头来看沃森的误判，病人有严重的出血症状，这个重要的信息，这条数据，究竟在哪个环节被“丢”掉了？



在微信上关注我们

InfoQ 简介

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

InfoQ 是一家全球性在线新闻 / 社区网站，创立于 2006 年，目前在全球拥有英、法、中、葡、日 5 种语言的站点。

InfoQ 中国于 2007 年由极客邦科技创始人兼 CEO 霍泰稳先生引入中国，同年 3 月 28 日，InfoQ 中文站 InfoQ.com.cn 正式上线。每年独立访问用户超过 2000 万人次。

InfoQ



国内最好的原创技术社区，一线互联网公司核心技术人员提供优质内容。订阅 InfoQ，看全球互联网技术最佳实践。

关注「InfoQ」

回复“架构师”，获取《架构师》电子书2017版合集



AI前线

提供最新最全AI领域技术资讯、一线业界实践案例、业界技术分享干货、最新AI论文解读。

关注「AI前线」

回复“AI”，下载《AI前线》系列迷你书



聊聊架构



以架构之“道”为基础，呈现更多的务实落地的架构内容。

关注「聊聊架构」

和百位架构师共聊架构



前端之巅

InfoQ 大前端技术社群：囊括前端、移动、Node 全栈一线技术，紧跟业界发展步伐。

关注「前端之巅」

回复“大前端”，下载大前端电子书



区块链前哨



掌握最前沿区块链资讯，深度分析区块链技术。从新手到精通，你只需要这一个专业助手。

关注「区块链前哨」

应对下一个互联网的到来



高效开发运维

常规运维，亦或是崛起的 DevOps，探讨如何 IT 交付实现价值。

关注「高效开发运维」

回复“DevOps”，四篇精品文章领悟 DevOps



Kotlin 生态调查结果出炉：超过 6 成的开发者用过 Kotlin 了

作者 Pusher，译者 无明

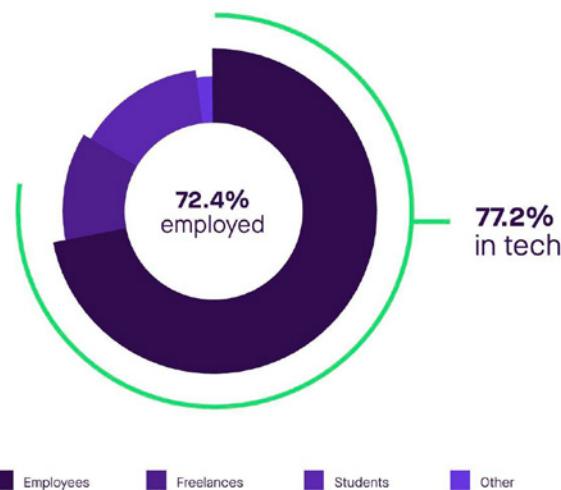


Kotlin从2011年低调问世，在短短几年间，如冲天火箭般流行起来。一年前，谷歌宣布将Kotlin指定为Android官方编程语言；来自Stack Overflow的一项问卷调查显示，超过10万名受调者表示Kotlin是他们的第二大编程语言。英国软件公司Pusher对此感到非常好奇，究竟Kotlin有什么特别的地方，让开发者如此着迷。于是，Pusher公司发起了一项针对Kotlin生态系统的问卷调查，从2018年1月份至3月份，为期三个月，受调者达到2744名。以下是这份问卷调查的结果及其简要分析。

01. 年轻的技术从业者更喜欢 Kotlin

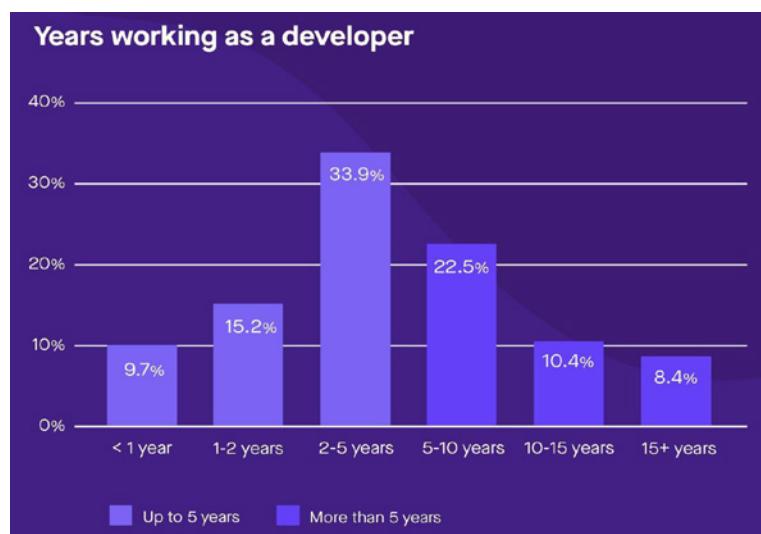
超过70%的受调者为企业雇员，企业家和承包商占了11%左右。他们

大部分都在科技行业工作，金融行业、教育领域和数字机构的比例则远远落后。保险行业和政府在采用Kotlin方面最为保守。



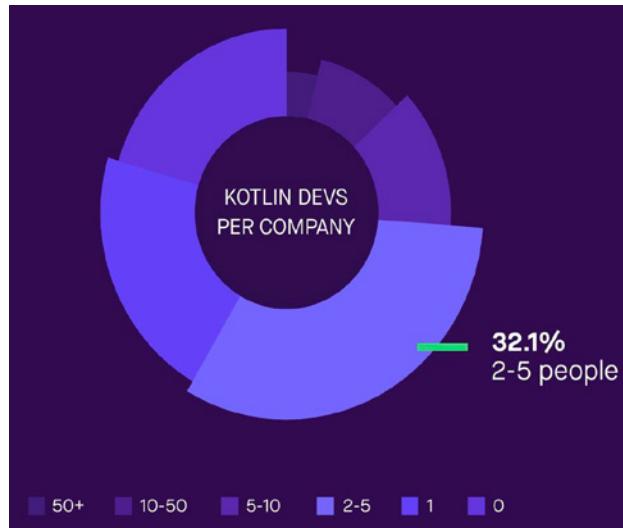
企业员工（72.4%），自由职业者（11.18%），学生（15.51%），其他（1.07%）。其中77.2%属于科技行业。

超过一半的受调者开发经验不足5年。他们似乎比参与Stack Overflow问卷调查的开发者拥有更少的经验。不过，“开发者流行度每5年会翻一番”的如意算盘仍然会奏效。



在企业中使用Kotlin的人数差异化严重。其中有三分之一的人表示，

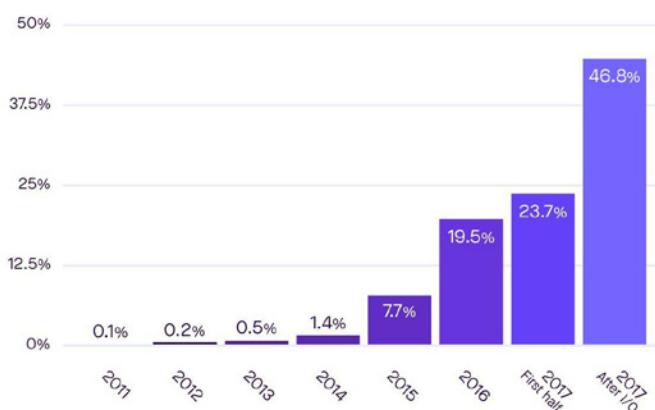
他们所在的公司有2到5个人在使用Kotlin，而这个数字恰好是一个Android团队的平均人员配置。



0人（20.6%），1人（21.1%），2-5人（32.1%），5-10人（13.1%），10-50人（9.4%），50人以上（3.7%）

02. Kotlin 正在走向成功

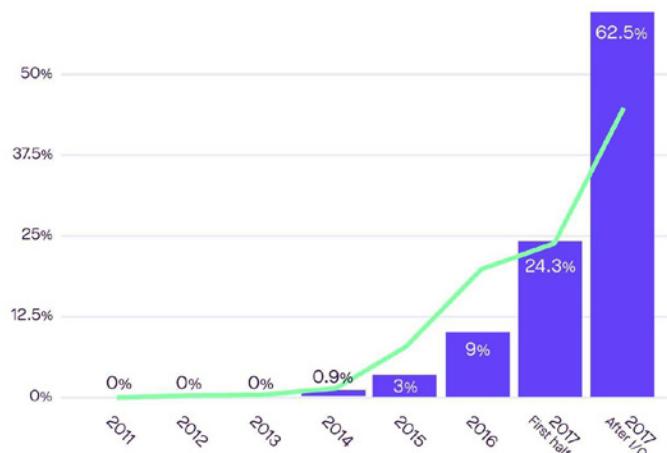
在2015年之前，Kotlin以每年翻一番的速度增长。2015年早些时候，来自Square的Jake Wharton发表了一篇文档，详细说明了他们为什么要采用Kotlin。他们的团队以开源流行的Android开发库而闻名。调查结果显示，那一年有很多人跟风，开始发表有关Kotlin的演讲和博客。



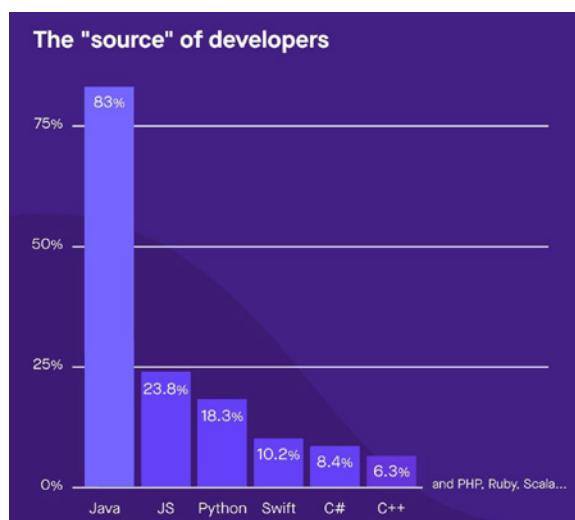
2017年5月之后，Kotlin的采用开始爆发。谷歌宣布将Kotlin作为Android的官方开发语言，大量Android开发者开始使用Kotlin。谷歌的这一举措无疑极大加快了Kotlin在未来几年的采用速度。（<https://youtu.be/Y2VF8tmLFHw>）

学生和年轻开发者非常相信谷歌的实力。刚开始，使用Kotlin的大多是有经验的专业开发者，但在谷歌做出宣布之后，更多的年轻开发者参与进来，特别是学生。

Adoption among students



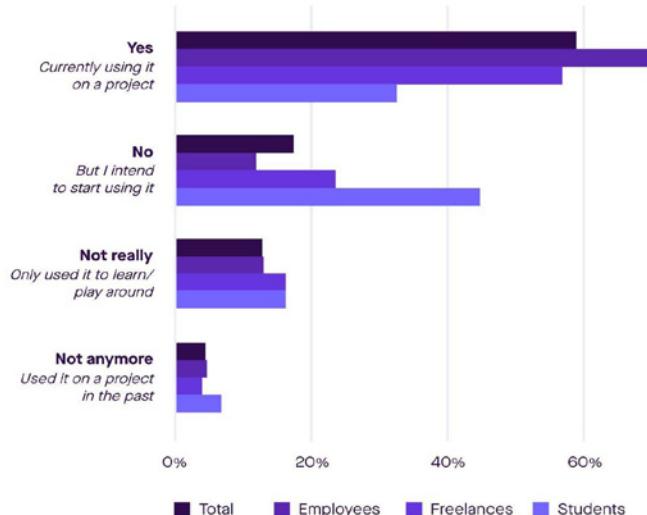
Kotlin吸引了来自各种背景的开发者。尽管Java仍然占主导地位，不过受调者当中使用其他编程语言的也很多，不过也有少部分人将Kotlin作



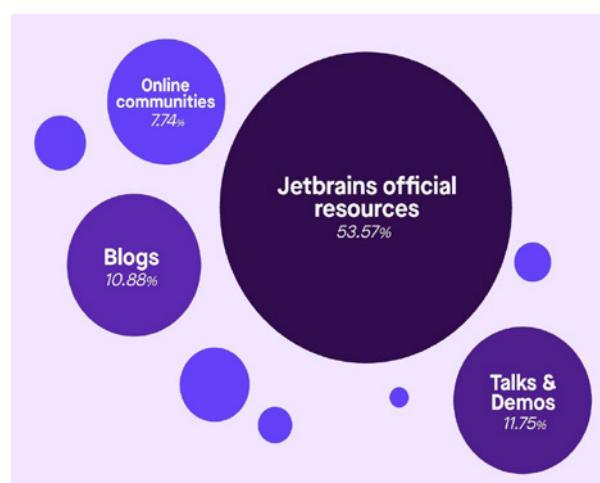
为他们的第一开发语言。

03. JetBrains 的努力如愿以偿

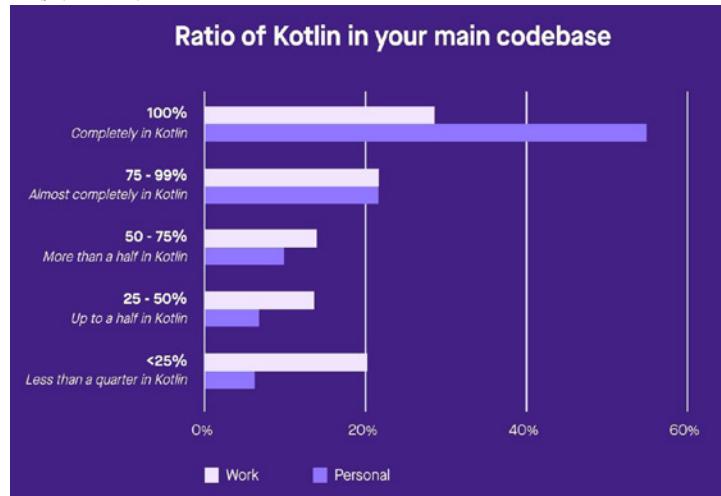
超过60%的工作者在他们工作的项目中使用Kotlin。相反，只有三分之一的学生在他们的工作和个人项目中使用Kotlin。还有将近一半的人表示会在未来使用Kotlin。



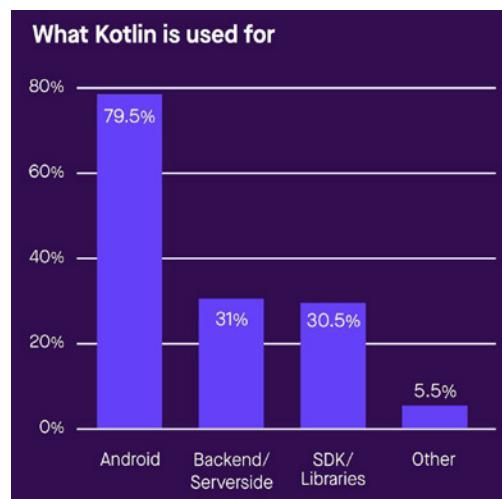
官方网站是目前最受欢迎的Kotlin学习去处。不过，学生更喜欢YouTube喝Udemy的在线教程，以及各种技术大会和演示。这些网站似乎会在接下来的几年提供视频流和实时代码服务，就像Twitch那样。



有很大一部分开发者在他们的工作项目和个人项目中使用Kotlin。不过，在个人项目中显然会用得更多，因为个人项目规模更小，更容易进行转换，也更方便进行实验。

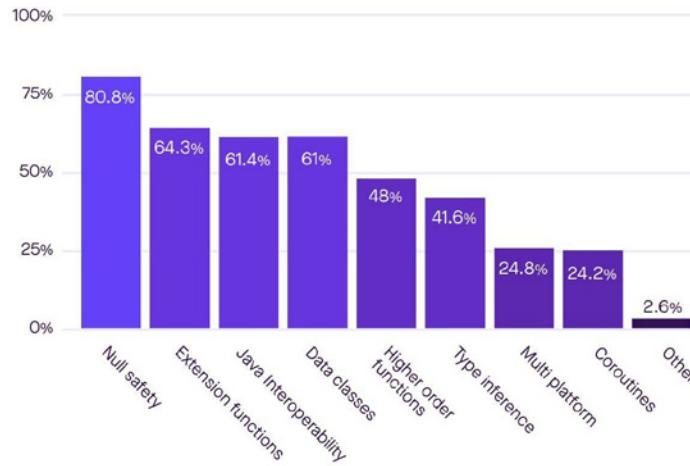


Android是Kotlin的主要使用平台。大量的专业开发者和学生使用Kotlin开发Android，而在后端，使用Kotlin的非常有经验的开发者。



04. 萝卜青菜各有所爱

每个受够了Java NullPointerException的人都喜欢Kotlin的Null安全特性。其中有4%的人选择了“一等函数”。

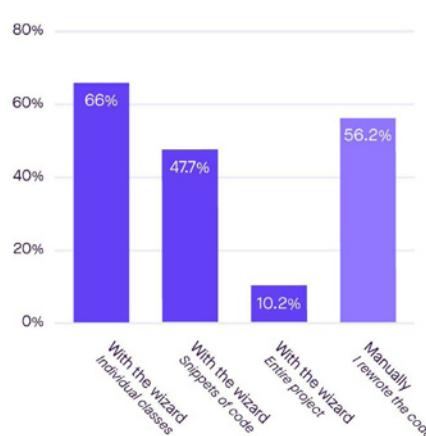


扩展函数被大量使用。77%的受调者表示，扩展函数提高了代码的可读性，特别是在进行函数式编程或在创建DSL时。越是经验丰富的开发者，越是喜欢用扩展函数。以下是扩展函数的常见使用场景：

- “我使用扩展函数……让算法从头到尾看起来很容易理解。”
- “只是为了检查null（因为我们有一个Java app）”。
- “主要是为了让代码更干净。我会尽可能控制它的作用域，除非它是一个非常通用的扩展函数”。
- “几乎用它做所有的事情。我经常使用内部扩展函数，以避免全局命名冲突”。
- “最主要将它用作DSL构建器的lambda参数”。

除了扩展Java类，人们也常常将Java代码迁移到Kotlin。超过87%的

Migrating Java to Kotlin



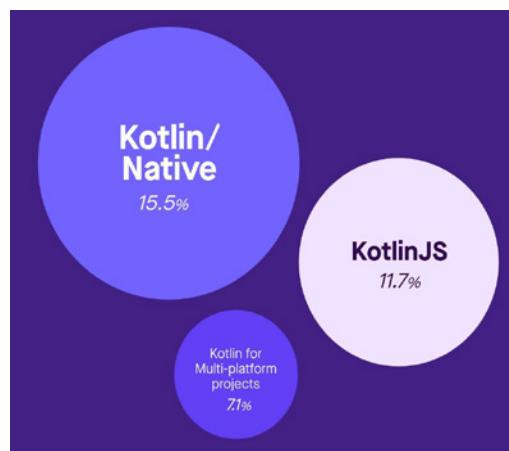
受调者已经完成了迁移。他们有的使用了迁移向导，有的直接手动修改代码。超过10%的人使用向导将整个项目迁移到Kotlin，其中有22%是学生或者经验不足一年的新手。

超过四分之一的受调者迁移到Kotlin后又回到了Java。有技术方面的原因，也有组织方面的原因。其中使用了反射或代码生成的工具是被提及最多的因素。

- “Kotlin的枚举不能包含常量。在自定义注解时（比如@IntDef），为了保持接口的整洁，需要将值保存在枚举中”。
- “我们正在使用Realm，但它不能与数据类一起使用”。
- “我们的Java代码中使用了Retrolambda，因为类型缺失，很难转到Kotlin”。
- “另一个团队不喜欢Kotlin，我们也预料不到会这样。”
- “这不是我们决定的，我们是按照公司的规则来的”。

Kotlin跨平台正在开始展现，但速度较慢。只有差不多四分之一的受调者表示，他们曾经用过跨平台支持，而且大部分使用了Koltin/Native，然后是KotlinJS。

假以时日，采用这些特性的人会增加。那个时候，Kotlin才真正有可能成为“编写一次，到处运行”的编程语言。



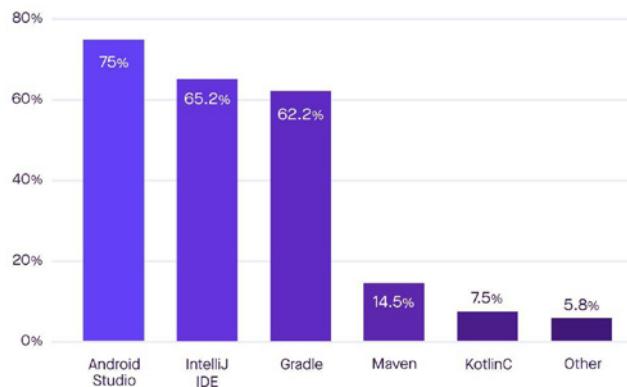
协程（coroutine）是Kotlin官方提供的异步编程模式，但因为是实验性的，所以很少被用到。只有三分之一不到的受调者使用了协程，而这些

人都拥有超过5年的开发经验。他们似乎已经在其他语言中使用过协程，所以在Kotlin中使用协程不会刚到别扭。

- “因为它是实验性质的，所以我先不用它。”
- “我们基本上将它作为fintech的解决方案，app的核心逻辑使用协程编写。”
- “我用了Arrow这个库，它使用协程实现for循环。”
- “目前正在使用协程替换CompletableFuture。协程主要用在高并行计算中的任务管理。”

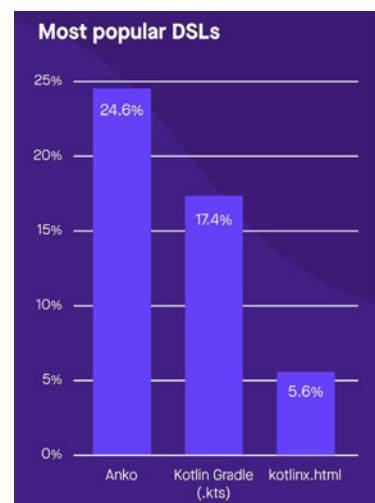
JetBrains和Android Studio几乎统领了Kotlin生态圈的开发构建工具。从图上可以看出，它们都位于Gradle（Android项目的默认构建工具）的左边，而独立编译器KotlinC的使用比例较低。

Most used build tools

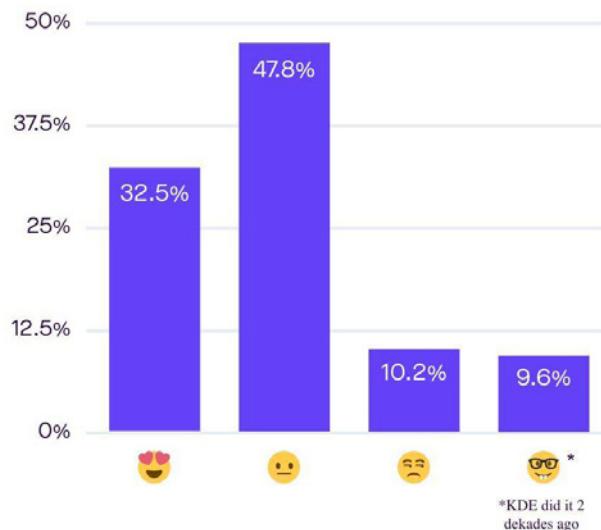


与协程类似，DSL也没有被广泛使用，因为它属于高级特性。大概有40%的受调者使用了DSL，除此之外，有四分之一的人自己开发DSL。当被问及他们都开发了哪些类型的DSL时，一般的受调者表示，他们开发的DSL都是与特定领域相关的，比如金融、大数据等。其他DSL还包括用于Android和配置工具的DSL。

学生们喜欢玩文字游戏。越是有经验的开发者，对此越是不关注。



Replacing Cs with Ks in libraries



2019 年的 Kotlin 将会怎样发展？

Kotlin的快速增长让人兴奋不已，但它是否真的像看起来的那么有前途？

因为有JetBrains和谷歌作为后盾，以及在开发者社区中广泛流行，我们可以确信，Kotlin会一路走好。但关键的问题是：它是否能够突破Android，进入到其他的领域？

JetBrains在极力推动它的多平台梦，但在未来几个月或者几年内是否能够看到令人欣喜的增长？Kotlin会成为Web、iOS或者后端开发者的新标准吗？

不管怎样，即使它需要几年时间才能跨出Android的藩篱，我们仍然会看到新的开发者加入到Kotlin的行列。他们的工具箱里将会多出一门万能的跨平台编程语言，这门语言涵盖了面向对象编程、函数式编程、脚本编程、声明式编程。这意味着Kotlin将会在编程语言领域产生重要影响，成为编程语言的标杆。

有一件事情是可以肯定的：Kotlin生态圈的发展让人拭目以待。

Stream 从 Python 切换到 Go 的原因

作者 Thierry Schellenbach 译者 安翔



Stream最近将其后端核心服务从Python改成了Go。虽然他们的某些模块仍然在使用Python，但是公司已决定从现在开始使用Go来编写对性能要求较高的代码。文中，Stream的CEO兼创始人Thierry Schellenbach将解释他们决定转向Go的原因。

影响项目或者产品编程语言选型的因素有很多。与任何技术决策一样，选择编程语言时同样需要多方面权衡，即使这样，最终的选择结果都很难是完美的。我们最近将后端的核心服务从Python改成了Go，原因有很多，好处也很多。

为了理解这一变化的重要性，需要先了解我们的产品。Stream是一套用于构建、伸缩、定制化新闻源和活动流的API。每个月为3亿多用户提

供约10亿次API请求。我们尤其关注性能和可靠性，这两点因素决定了我们制定的每项技术决策。

性能更优

Go最大的卖点在于它的性能，无论在运行还是编译时它都有突出的性能优势。它与Java或者C++的运算速度几乎相当。在实际使用中，我们发现它比Python大约快30倍。

选择快速工具对提升系统性能非常重要，因此我们对Cassandra、PostgreSQL、Redis以及其他一些技术进行了优化。然而，很多时候我们发现系统仍然存在瓶颈，而瓶颈正好在于我们的编程语言Python。Python在执行序列化、排序和聚合等计算密集型任务时需要花费很长的时间，有时比从网络上存取和检索数据花费的时间更长。我们知道这个时间是可以优化的。从Python切换到Go就可以缩短时间，这样一来，应用程序代码就更像是服务之间的粘合剂，而不再是优化中的主要瓶颈。

用Go编写的Go编译器也非常快。Stream中最复杂的微服务就采用Go编写，它的编译时间仅仅需要6秒，Java和C++等工具链则慢得多，快则一分钟，慢则数小时。

名副其实的简单

简单是Go的重要特征！我敢向你保证，阅读Go语言的代码明显感觉更加简单。我们已经从多个Python代码库中迁移出来，我们发现这些Python代码的风格和框架会因为作者的不同而风格各异，往往带有很多作者个性化的东西。而Go恰恰相反，它推崇干净的代码风格，同时要求作者编写代码时严格遵守规范，禁止作者“自作聪明”。虽然这样有时候会使用更加冗长的代码，牺牲了代码的简洁性，但是却让代码更容易阅读和理解了。这样一来，Go才得以加快开发人员阅读他人代码的速度，同时，阅读自己曾经编写的代码也更容易。

原生并发性

Go在语言层面通过goroutine和channel支持了并发。此概念源自Tony Hoare的CSP模式，它让程序员处理并发变得不再困难。

goroutine类似于操作系统的线程，但其运行消耗的系统资源更小，每个goroutine仅需几KB的堆栈空间。Go运行时可以在操作系统线程之上处理多路goroutine。虽然在后台执行，但它对于程序员来说是可见的。单个程序拥有数千个goroutine也并不罕见。比如，net/http软件包中的服务器程序针对每个HTTP请求都会创建一个goroutine。

在Go中启动goroutine非常简单，只需通过go关键字添加一个函数调用，即可启动一个goroutine，并让该函数运行在自己的goroutine中。

Go有一句重要的格言，即：不要通过共享内存来通信，相反，通过通信来共享内存。Goroutine之间通过channel进行通信，channel的使用方法与goroutine一样简单。Channel拥有类型，可以通过直观的箭头语法轻松实现goroutine之间的数据传递。尽管channel使用简单，但是其功能非常强大。在设计时只要预先稍作考虑，与传统的系统相比，使用Go便能够轻而易举地开发大规模并发系统。

使用简单的并发工具可以解决那些经常导致错误的问题。Go内置了竞态条件检测器，可以更轻松地检测异步代码中的竞争状态。

语言生态

跟C++和Java这样已经高度普及的传统语言相比，Go仍然是编译语言领域的新手。虽然目前大约只有5%的程序员知道Go，但是得益于它的易用性，这个数字在不断增长。虽然Go语言速度快且功能强，但它只有25个保留字。相比于C++的92个保留字，以及Java的53个保留字，Go显得非常简洁。过多的保留字会增加程序员的学习成本。

由于Go上手非常容易，因此组建Go开发团队相比其他语言来说更容易。Go初学者可以很快入门并精通该语言。这使得雇主甚至可以招聘其

他背景的开发人员，然后加以短期培训即可使其成为合格的Go工程师。

Go提供的内置库开箱即用且功能强大。使用“net/http”仅需几行代码即可实现HTTP服务器，并且还支持http/2、TLS和websocket。Go社区软件包的生态系统也很出色，已经出现了很多与Redis、RabbitMQ、PostgreSQL、模板以及RocksDB相关的库，它们运行稳定且更新频繁。

其他优势

在前文中我提到了Go并不鼓励程序员“自作聪明”，它并没有提供可能会节省时间的功能，比如可嵌套的三元运算符。

Go采用另一种方式来节省时间，它既没有选择制表符也没有选择空格，而是转而使用了gofmt。它是一种命令行工具，可与大多数编辑器集成并自动将代码格式化为特定的格式。即使格式不正确代码仍会编译，但是拉取请求会被忽略，除非代码通过gofmt并且能够保持整个代码库格式一致。这使得代码评审人员能够专注在代码上，而不必在格式上浪费时间。

Go有助于开发微服务。谷歌的protobuf和gRPC是微服务间通信的基础，Go对它们提供了很好的支持。作为开发人员，我们只需在清单文件中定义一项服务，工具便会自动生成客户端和服务器端代码，并且保证代码的高性能以及很低的网络负载。此外，清单文件还可以被其他语言用来生成他们自己的客户端和服务器端代码。所以，如果我们决定用其他技术来替代部分架构，之后的任务会更加简单。

Python vs. Go

Stream服务强大功能之一是feed排名。feed排名允许我们的用户为feed指定一个评分函数，以便控制排序方式。评分算法可以提供很多变量来确定排名，其中基于流行度的一个例子可能是这样的：

```
{  
  "functions": {  
    "simple_gauss": {
```

```

    "base": "decay_gauss",
    "scale": "5d",
    "offset": "1d",
    "decay": "0.3"
},
"popularity_gauss": {
    "base": "decay_gauss",
    "scale": "100",
    "offset": "5",
    "decay": "0.5"
}
},
"defaults": {
    "popularity": 1
},
"score": "simple_gauss(time)*popularity"
}

```

- 为了支持这种排名方法，Python和Go代码都需要解析表达式计算得分。在这种情况下，我们需要将字符串simple_gauss (time) * popular 变成一个函数，它将活动数据作为输入，并输出分数。
- 基于JSON配置创建部分功能。例如，我们希望“simplegauss”以五天的时间窗、一天的偏移量以及0.3的衰减因子来调用“decaygauss”。
- 解析“默认”配置，以便在活动数据中发现未定义的字段时进行回退。
- 使用步骤1中的功能对feed中的所有活动数据进行评分。

开发Python版本的排名代码需要花费大约三天时间，包括编写代码、单元测试和编写文档。接下来，团队需要大约两周的时间来优化代码。其中一项优化是将分数表达式（simple_gauss（time）* popular）转换为抽象语法树。该团队还实施了高速缓存逻辑，预先计算了将来某些时间的分数。

相比之下，开发这些代码的Go版本大约花费了四天时间，并且不需要

再对其性能实施进一步的优化。虽然Python用来开发初期版本更快，但是整体来说使用Go开发的工作量要小得多。

Go的语言特性使得在优化代码时能够节省大量的时间。使用Python时，我们不得不将表达式解析为抽象语法树，并优化和剖析每一个函数。由于Go比Python快得多，因此我们不需要花太多精力优化代码。最终的结果是，Go代码的执行速度比精心优化的Python代码大约快40倍。

用Go来构建Stream系统中的某些组件相比用Python花费了更多的时间。总体来说，开发Go代码要花费更多的精力，但团队用来优化代码性能的时间则更少。

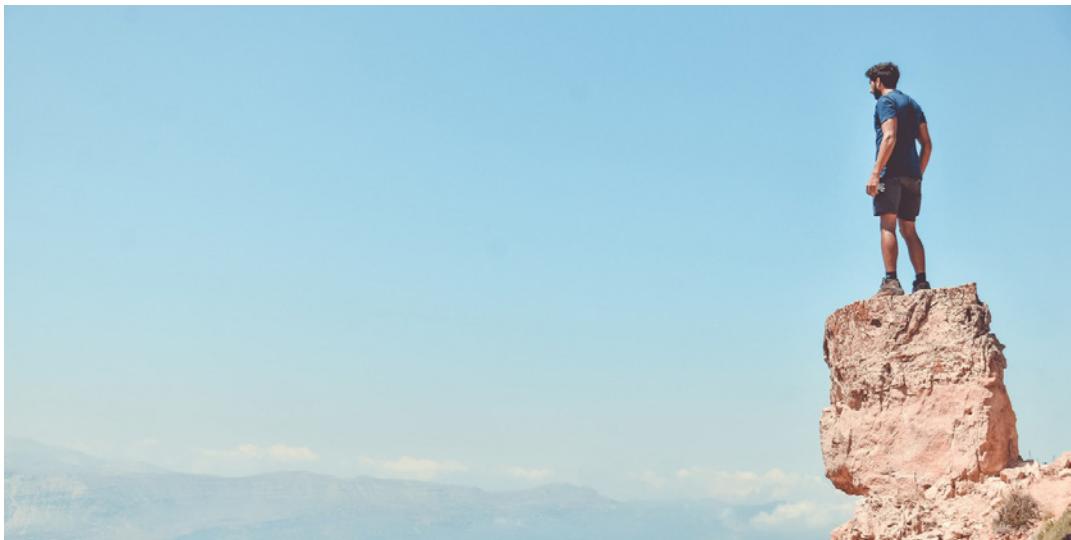
结论

Go非常适用于开发微服务。它的速度非常快，具有原生并发原语，完美支持多种现有工具，并且开发起来乐趣无穷。与Ruby或Python等脚本语言相比，编写Go代码可能需要更长的时间，但其维护成本要低得多，加之其代码无需太多优化，因此你可以节省大量的时间。

需要注意的是，对于某些适合使用Python开发的模块，Stream仍然使用Python。例如，我们的仪表板、网站以及用于个性化订阅的机器学习都使用Python实现，因为Python提供的这些工具更好用。我们不会马上完全弃用Python，但是对于性能要求较高的代码，我们今后会使用Go来编写。

腾讯大规模分布式机器学习系统无量是如何进行技术选型的？

作者 张红林



导读：在互联网场景中，亿级的用户每天产生着百亿规模的用户数据，形成了超大规模的训练样本。如何利用这些数据训练出更好的模型并用这些模型为用户服务，给机器学习平台带来了巨大的挑战。腾讯开发了一个基于参数服务器架构的机器学习计算框架——无量框架，已经能够完成百亿样本 / 百亿参数模型的小时级训练能力。无量框架提供多种机器学习算法，不但能进行任务式的离线训练，还能支持以流式样本为输入的 7×24 小时的在线训练。

1. 背景

QQ 浏览器首页的推荐 Feeds 流。业务入口如图所示：



图 1 QB Feeds 流业务

浏览器的 Feeds 业务每天的流点击曝光日志在百亿级；为了更好的给用户提供个性化的推荐服务，如果我们取半个月的数据来训练推荐模型的话，则我们会面对一个千亿样本的状况。

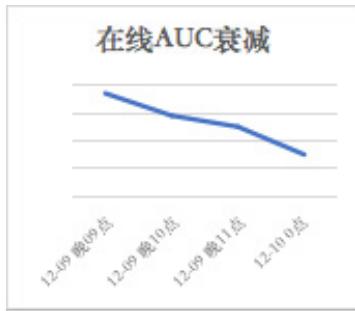


图 2 模型在线表现的时间衰减曲线

此外，对一个训练好的模型，我们观察了模型在线的指标变化，如图所示。这个图说明我们的 Feeds 流业务是一个时效性高度敏感的业务，在线用户访问的规律实时在变化，要取得最好的业务效果，我们必须不断及时的更新模型。浏览器另一个业务——识花君，需要用百万级图片预训练一个分类的图片分类模型，如果采用单机单卡的模式，大约需要半个月才能训练一个收敛的模型；如果使用 TensorFlow 的分布式训练也大概需

要一周，有没有更高效的方法呢？

针对这两个业务场景，接下来我们做一些技术分析，看看有没有一些解法。

场景 1：大数据 + 大模型

在可以低成本获得样本的场景，比如广告、Feeds 流的 ctr 预估场景，因为不需要标注我们就可以低成本的获取海量的正负样本，这就会促使我们设法从这海量的样本里学习足够的知识。

什么样的算法模型可以从海量数据里学习充分的信息呢？这里从 VC 维理论出发，我们知道一个模型可以容纳的信息是有限的；下图概括了样本数量、模型规模和模型效果之间的关系，这里我们用模型效果来侧面反映模型容纳的信息量是基于这样的假设：如果一个模型从同样规模的数据里学习到了更多的信息，那么我认为它在业务上会体现出更好的效果。这个假设当然还会有很细微的条件，但这里就不深究了。

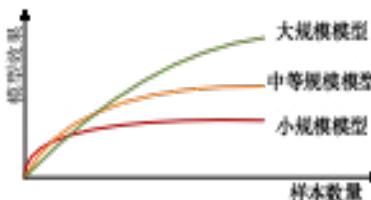


图 3 模型信息和样本、模型规模的关系

从该图我们可以直观得出一个结论，对于可以轻易获取海量样本的场景，我们需要用足够大的模型去容纳其中的信息。为避免过于直观，这里我且举一个例子，以一个亿级 Feeds 流业务为例，如果每天用户点击超过一亿，那么单天用户的 pv 可能在 5-10 亿甚至更多；如果我们取半个月的数据来训练一个 CTR 预估模型，涉及到的样本量在 200 亿左右（不考虑向下负采样），而如果我们的模型参数是样本的 10 倍的话（这个范围并不夸张），我们的模型参数数量在 2000 亿，每个参数用四字节表示，我们的模型将达到 1TB 左右；而如果我们用 double 精度则接近 2TB。

这个量级的模型如何训练？如何做在线 serving？2000 亿的原始样本

如何存放？答案是唯一的：我们需要一个分布式系统。

场景 2：大数据 + 中小模型

这是另一种场景，以某图像分类业务为例，我们要将一个标注好的图像数据集通过模型分类到几千个类目上。数据集我假设 1000w 张图片；乍一看，似乎这个和大数据关系不大，才一千万而已，但注意这里是图片，如果我们把图片的每个像素作为一个样本来对待，这个数据就大了；为什么这么说？因为我们用 CNN 类的网络来训练的时候，图片本来就是以像素输入；是的，这里的大数据其实想表达的是对算力的要求。

如果我们在单机单卡（GPU）上来训练这个分类模型（以 resnet-101 为例），可能需要 2-3 周；真的是“洛阳亲友如相问，就说我在跑 training”。对于算法同学来说，如果我们要等一个模型结果需要 3 周，这显然是很让人沮丧的一件事。那么我们有没有机会把这个时间缩短到天甚至小时级别呢？答案也是一样的：我们需要一个分布式系统。

上面两个场景也许只是鹅厂众多业务场景中的一小部分，但我相信是有一定的代表性的。这里共同的答案是我们需要一个分布式系统来应对业务场景带来的工程挑战。从机器成本的角度，我们不太可能去定制能满足需求的单台机器来解决；从人力成本的角度出发，我们也不太可能容忍模型训练速度的超级低效；因此使用相对便宜的机器构建一个面向机器学习需求的分布式系统是我们唯一的选择。

2. 分布式机器学习的架构与物理设计

分布式机器学习系统，顾名思义，和分布式文件系统、分布式后台服务类似，是一个分布式系统（这似乎是废话）；再结合机器学习就不一样了，这是一个面向机器学习场景的用相对便宜的机器组建的分布式系统（这还是废话）。那么和传统的分布式系统相比，分布式机器学习系统有哪些独有的特点呢？做这类系统的开发需要哪些算法知识和工程思维呢？

和传统的分布式系统很大的一个不同的地方在于，传统的分布式系统

是 operation-oriented；以存储系统为例，传统分布式文件系统是绝对不能接受比如一个数据块写错地方了这样的事情的。



图 4 operation-oriented system

与之不同的是，如果我们以 operation-oriented 的要求来应对分布式机器学习的问题的话，那结果会是相当悲剧的，以我们目前的算力，我们可能根本没法在可接受的时间内完成一个大模型的训练的。然而上帝关上一扇门的时候也许会帮你掀开屋顶；机器学习的模型和算法本身都是有充足的容错能力的，你丢个样本，或者丢个梯度基本不影响模型的最终收敛，而这给了分布式机器学习系统一条出路，我姑且称为 convergence-oriented system。



图 5 convergence-oriented system

如图 3 所示，convergence-oriented 系统和下山比较类似，下山的路有

无数条，中间你走偏了也无所谓，只要你的大方向是往山下即可。

因为机器学习算法自身的特点，分布式机器学习系统相比于传统分布式系统在数据通信、同步协议、容灾等方面都有极大的活动空间，也为我们在追求极致的性能打下了基础。对分布式机器学习系统的通信、同步协议有兴趣的通信可以参考之前的拙作^[1]和 Eric 的相关文章。接下来带大家游览一下应对两种场景的可用的系统架构！

2.1 参数服务器

关于参数服务器，之前的拙作^[1]已经有较多的论述，这里不再详细展开，仅作简单介绍，想深入了解的同学请根据^[1]按图索骥。

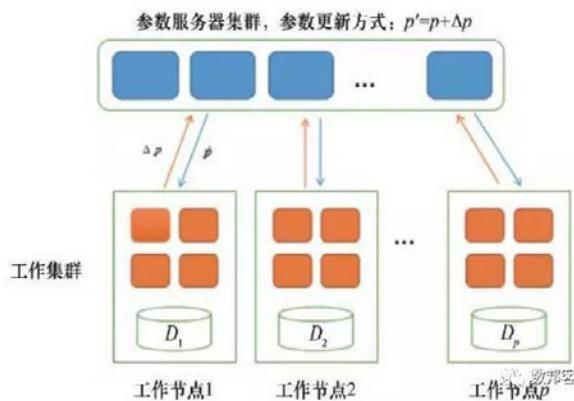


图 6 参数服务器架构示意图

如图所示，参数服务器逻辑上分为 server 和 worker 两类角色；server 负责存储模型参数，每个节点负责一个参数分片；worker 负责根据不同的数据分片来计算该数据分片涉及到的参数对应的梯度增量，并回传给 server 节点以 update 模型。因为数据和模型都是分布式存储，架构简单健壮，理论上该架构可以支持的模型规模是无限的；但是另一方面我们也应该看到，因为每个数据分片涉及到的参数分片可能分布在不同的机器上，导致我们每增加一台机器，网络的整体传输量会有所增加；如下图所示

因此，在参数服务器架构下，相对于算力瓶颈，网络更容易成为我们的瓶颈，而这又该如何解决呢？请继续往下浏览。

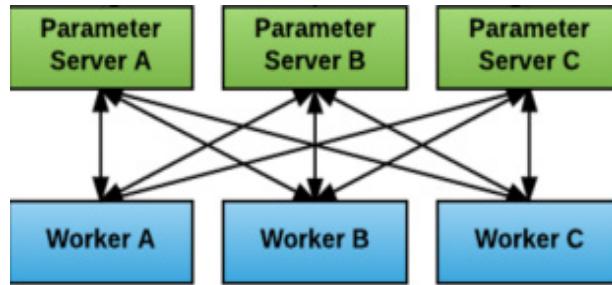


图 7 参数服务通信示意图

2.2 ring-allreduce

对于图像分类、机器翻译这类强依赖 GPU 机器的场景，我们来看看另一种情况：

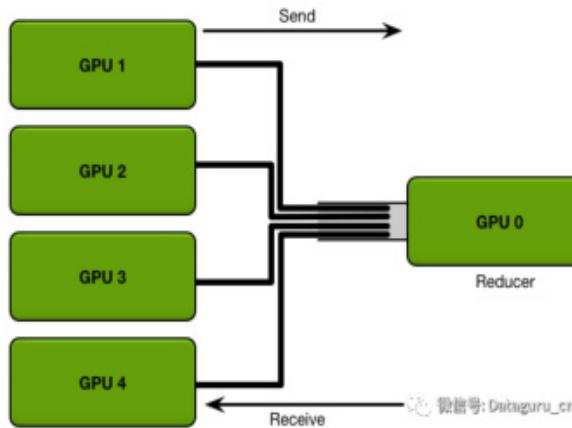


图 8 使用 GPU 构建的 ps 架构

如图所示，如果我们使用 GPU 搭建一个 PS 集群，我们将面临更为严峻的挑战；因为 GPU 的运算速度极快，我们在做参数 reduce 的时候，与 GPU0 的通信时间将成为整个系统的 dominant time 而让系统中的 GPU 心有余而力不足。为此，百度的 SVAIL 团队^[3]从高性能计算领域借鉴了 ring-allreduce 思想，构建了分布式机器学习的 ring-allreduce 架构，如图9 所示。

将 GPU 布置成环状现在以有官方组建 NCCL 可以支持，对 NCCL 原理感兴趣的可以参考^[4]等相关 paper。

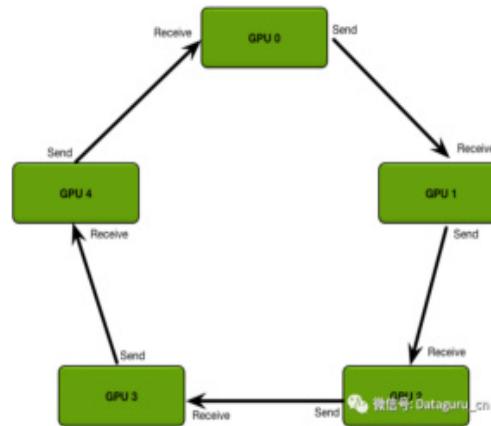


图 9 ring-allreduce 架构示意图

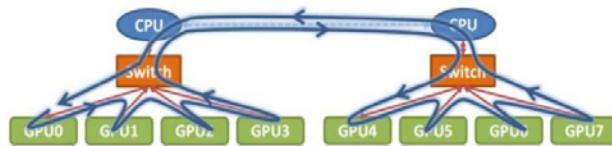


图 10 使用 NCCL 搭建的多机多卡环

如图所示，当我们使用 NCCL 将多台 GPU 机器搭建成环状结构时，我们可以看到在换上以此传输的话网络带宽可以得到比较充分的应用。接下来解释下 allreduce 的概念，一般的 reduce 概念如下

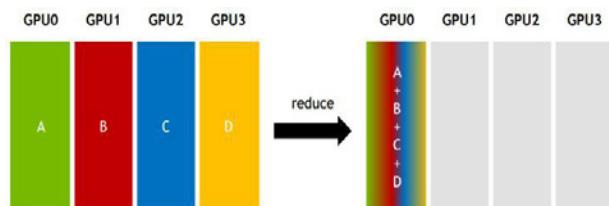


图 11 reduce 操作语义

而 allreduce 的概念如下。

因为上述图片已经足够直观，这里就不多加解释了。接下来我们介绍 ring-allreduce 为什么适合 GPU 集群数据并行的场景；考虑到中小规模的模型我们可以存放在单台机器上（单卡 or 多卡但不跨机器），每台机器根据自己的数据分片训练模型后通过环状通信来做 allreduce 操作；这样

的设定下整个系统的网络通信量不会随着机器增加而增加，而仅仅与模型和带宽有关，相对于参数服务器架构而言，这是极大的提升。详细的推导过程可以参考^[5]，我就不赘述了。实际的网络通信流程如下所示

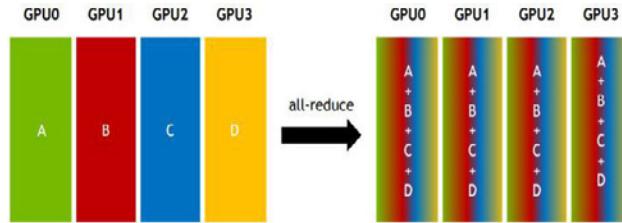


图 12 allreduce 操作语义

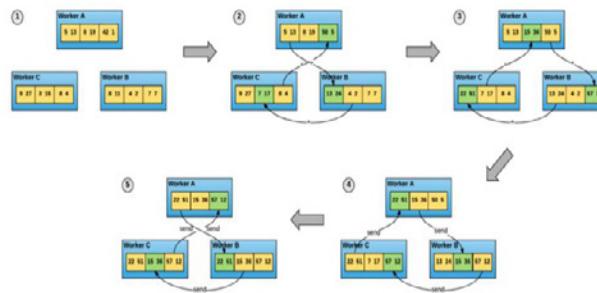


图 13 Ring-allreduce 通信的物理过程

在介绍了参数服务器和 ring-allreduce 两种不同的分布式机器学习的系统架构以后我们该如何根据自己的业务场景来合理的选择架构、算力社保、部署策略呢？请看下节

2.3 物理实现的设计选择

前两节介绍的两种逻辑架构在物理实现的时候可以有多种选择，这里做几种推演：

2.3.1 PS 数据并行

仅使用 PS 架构来支持数据并行，如图14所示。

这种架构下仅仅支持 worker 对数据进行并行计算，模型存放在集中的 server 节点，和 spark 的架构类似。因为是单节点，所以模型不可能太大，因此这个模型仅仅对照意义多一点，实际上基本不会这么用。

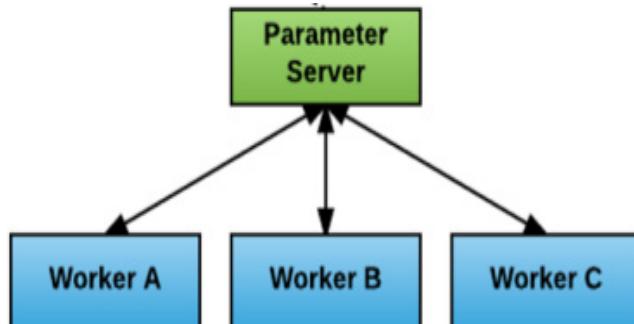


图 14 单 server 参数服务器架构

2.3.2 PS+p2p

在实现的时候，将参数服务器的 worker 和 server 两个角色融为一体，在一个进程中既有承担 server 角色的线程，又有负责 worker 的线程；因为 worker 以计算为主，server 以参数存储为主，这种融合有一定的合理性，如下图所示，虚线框表示一个物理进程，一台机器上可以部署一个 or 多个这样的物理进程。

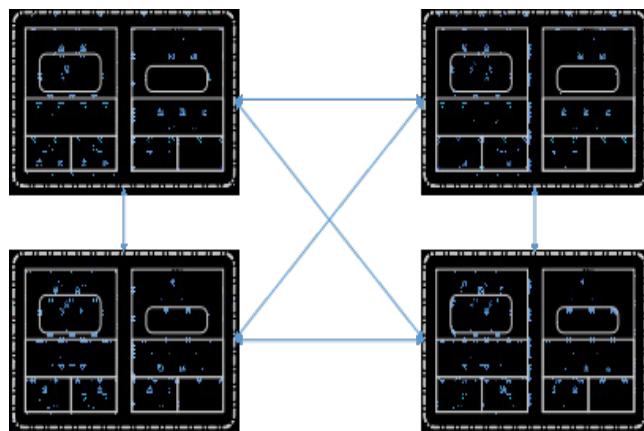


图 15 P2P 结构的参数服务器

这种架构的不足之处在我看来有两点：1. 角色耦合，较难根据机器来调配线程比；debug 也相对困难一点；2. 架构耦合，扩展的灵活性较差；调度系统交护模块、监控模块的配合、灾难恢复都有一定的风险。

2.3.3 PS 角色分离

与图 13 不同，如果我们将 worker 和 server 两个角色实现为解耦开的

两个独立进程，在可以给调度系统流出更多的活动空间。同时对架构的扩展也预留了空间，如果我们再独立一个单独的调度模块出来，则演变为下一种架构。

2.3.4 PS+scheduler

当我们把 worker 和 server 拆成两个独立的模块，并引入一个 scheduler 模块，则会形成一个比较经典的三角色分布式系统架构；worker 和 server 的角色和职责不变，而 scheduler 模块则有比较多的选择：1. 只承担和下层资源调度系统般若（类似 yarn、mesos）的交互；2. 除 1 外，额外增加对 worker、server 心跳监控、流程控制的功能；如下图所示：

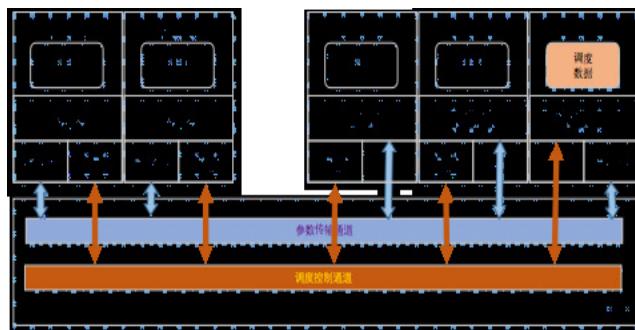


图 16 带控制模块的参数服务器

引入 scheduler 模块的另一个好处是给实现模型并行流出了空间，关于模型并行概念的理解，请参考^[1]；关于在 scheduler 模块下如何实现对模型参数的调度以达到模型并行的效果，请参考^[6]中对 SchMP 编程范式的论述；调度模块不仅有利于实现模型并行训练范式，还有其他好处；比如通过针对特定模型参数相关性的理解，对参数训练过程进行细粒度的调度，可以进一步加快模型收敛速度，甚至有机会提升模型指标。这块也是一个很值得探索的方向，有兴趣的同学可以进一步参考^[7]。熟悉分布式系统的同学可能会担心 scheduler 模块的单点问题，这个通过 raft、zab 等 paxos 协议可以得到比较好的解决，无需过于担心。

2.3.5 ring-allreduce+PS

初始的 ring-allreduce 有一个开源版本是 uber 实现的 horovod 框架，通

过测试我们重现了 horovod 论文里的加速情况，如下图所示：

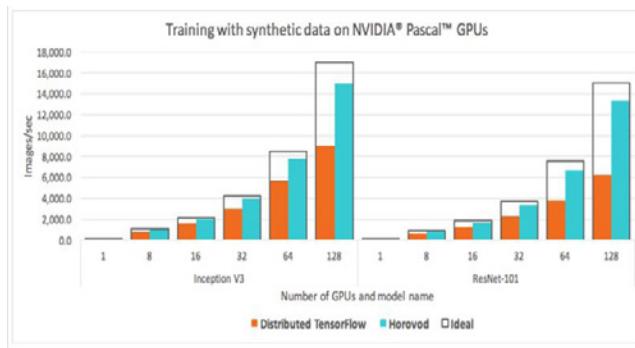


图 17 多机多卡场景下 ring-allreduce 架构加速比对照 TensorFlow 加速比

从该图可以看出 ring-allreduce 的加速比和理想加速比的斜率几乎完全一致，而 TensorFlow 的加速比则远低于次；这证明了 ring-allreduce 通信机制相对于 PS 机制在网络通信方面的优势；但与 PS 架构不同的是，初始版本的 ring-allreduce 假设模型参数需要单卡可以存下，另外如果模型中全连接层比较多，则全连接层的强耦合性结合 allreduce 类似 bsp 的同步机制，还是会让网络通信时间成为瓶颈。因此，在 ring-allreduce 环境下，我们是否可以做模型分片、同步协议的改造，比如利用 SSP 来替换 BSP，或者利用梯度压缩来加快 allreduce 进程都是值得探索的方向。

3 技术成果

经过大半年的封闭开发，目前无量系统已经支持了 LR、FM、FFM、DNN 的离线训练和在线实时训练。支持了 FTRL、SGD、Adam、AmsGrad 等多种优化算法。针对不同的优化算法，我们在梯度压缩上也做了一些基本的尝试，如图18所示。

如图所示，在 LR 算法分布式训练过程中，我们过滤掉 99% 的梯度，仅传输剩下的 1% 的梯度依然可以达到模型收敛的效果；而且指标可能还有提升，我们推测可能是大范围过滤梯度引入了一些 regularition 的作用。

除了常规算法之外，我们自研了大规模 embedding+DNN 的分布式训练

支持，如图19所示。

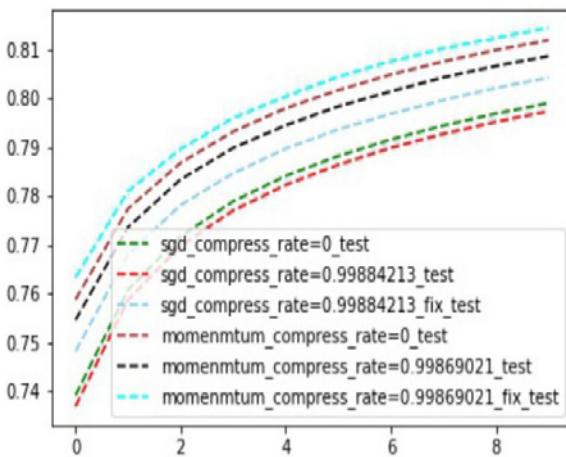


图 18 不同优化算法做梯度压缩后的收敛指标对比

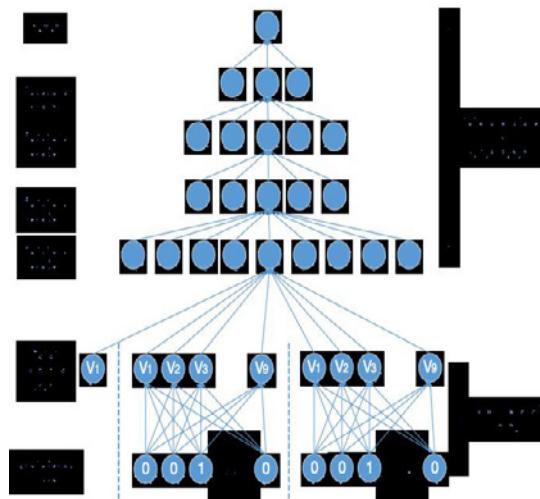


图 19 自研分布式 DNN 模型

该模型在召回和精排环节都可以应用，目前已经在线灰度。回到最开始的问题，我们封闭开发无量的一个初衷还是为了支持 Feeds 业务精排环节，那么面对大数据 + 大模型我们现在是什么情况呢？无量支持了千亿级特征空间的稀疏 LR 的分布式训练；目前在线已经实际使用到百亿特征，百亿样板，训练好的模型为了方便在单机上做 inference，我们会做一些裁剪；详细过程我可以参考我另一篇分享。

使用了基于无量系统训练的模型之后，Feeds 在线 CTR 和曝光效率都

有显著的提升，如图所示；相对提升百分比在两位数，这个提升是在基于 GBDT+ 细粒度特征的粗排基础之上的提升，因此这个结果还是非常符合业务的预期的。

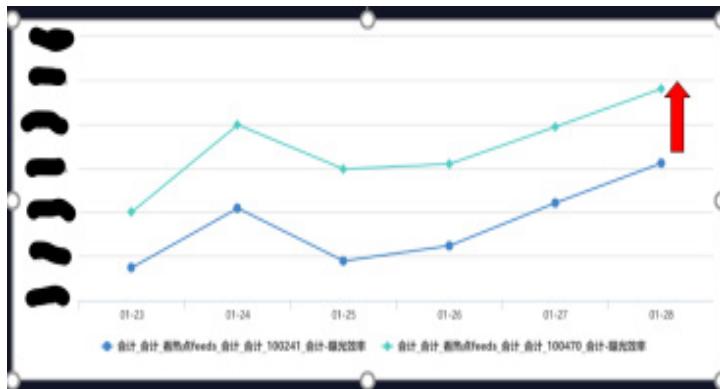


图 20 在线效果提升比例

在另一个方向上，我们基于 ring-allreduce 的架构，对大数据 + 小模型的 cv 场景已经可以做到小时级模型输出；该场景以后会做更深入的探索。

4. 团队介绍及文章计划

无量项目是 MIG 移动浏览产品部与无线运营部联合开发的，团队主要开发成员由大数据中心下的智能应用组、运营部下的计算框架组以及浏览器大资讯业务相关同学构成，主要成员如下：robertyuan、suziliu、clarebu、yancyliu、wahmingchen、burnessduan、binzhu、williamqin、carbonzhang、janwang、collinhe、joeyzhong、foxchen、brucebian 等。

本篇为系列分享的第零篇，主要介绍分布式机器学习框架的背景及可用架构；接下来我们会从系统整体概况、工程挑战、算法挑战、业务应用等角度展开系列分享，敬请期待！

致谢

特别感谢浏览器和运营部两位老板 henrys xu 和 xinliu 的支持，没有

老板的支持我们不会有办法去探索分布式训练这个领域；感谢 foxchen、taydai、brucebian 的给力支持，使得项目的进展过程中，资源的支持始终走在开发先列。感谢 rainyu、joeyzhong、janwang 的支持，在过程中对项目高度关注，经常组织大家讨论和勾兑；最后感谢 robertyuan、suziliu、clarebu、yancyliu、wahmingchen、burnessduan、binzhu、hbsun 等同学的辛苦开发，过程中有过碰撞，最终时间让我们了解彼此，共担重担！

还有很多同学在项目上线过程中提供了极大的帮助，如 larrytu、aiyima、ballwu 等和我们一起对流程、对参数，可能无法一一列出，然感激之情，不减毫厘！

引用

- [1] 大规模机器学习框架的四重境界
- [2] More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server
- [3] https://www.sohu.com/a/127596575_494939
- [4] Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations
- [5] <https://www.zhihu.com/question/63219175/answer/206697974>
- [6] STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning
- [7] Managed Communication and Consistency for Fast Data-Parallel Iterative Analytics

GitHub 的 MySQL 高可用性实践

作者 Shlomi Noach , 译者 张健欣



GitHub 使用 MySQL 作为所有非 git 项目的主要数据存储，因此 MySQL 的可用性对于 GitHub 的运维来说至关重要。站点本身、GitHub 的 API、身份验证等都需要数据库访问。我们运行多个 MySQL 集群来服务我们的不同服务和任务。我们的集群使用经典的主-副设置，其中集群的单个节点（主节点）能够接受写操作。其它集群节点（副节点）异步更新主节点的变更并服务我们的读流量。

主节点的可用性特别地重要。主节点不可用时，集群就不能接受写操作：任何需要持久化的写操作都不能被持久化。任何传入的变更，例如提交代码、提问题、用户创建、代码审查、新建代码库等等，都会失败。

为了支持写操作，我们显然需要有一个可用的写节点，即集群的主节点。但同样重要的是，我们需要能够识别，或者发现，那个节点。

遇到一个故障时，比如主节点崩溃的场景，我们必须确保保存在一个新的主节点，并且能够快速通告其身份。检测故障、运行故障恢复以及通告新主节点身份所花费的时间组成了总宕机时间。

本文阐述了GitHub的MySQL高可用性和主服务发现解决方案，这个方案使得我们能够可靠地进行跨数据中心运维、克服数据中心隔离的影响并实现故障时的短宕机时间。

高可用性目标

本文描述的解决方案是对GitHub先前实现的高可用性（HA）解决方案的迭代和改进。随着我们规模的扩大，我们的MySQL HA策略必须适应变化。我们希望对我们的MySQL和GitHub的其它服务运用相似的HA策略。

当考虑高可用性和服务发现时，一些问题可以指导你找到一个恰当的解决方案。这些问题包括但不限于：

- 你能容忍的宕机时间是多久？
- 崩溃检测的可靠性如何？你能容忍假阳性（过早进行故障恢复）吗？
- 故障恢复的可靠性如何？它在哪些情况下会失败？
- 解决方案跨数据中心能力如何？在低延迟和高延迟网络的能力如何？
- 解决方案能克服完整的数据中心故障或网络隔离的影响吗？
- 如果有的话，什么机制能够防止或减轻脑裂现象（两个服务器都宣称是指定集群的主节点，都独立地彼此无意识地接受写操作）？
- 你能够承受数据丢失吗？到什么程度？

为了说明上述一些问题，让我们先看一下我们之前的HA迭代以及为

什么我们要改变它。

远离基于 VIP 和 DNS 的服务发现

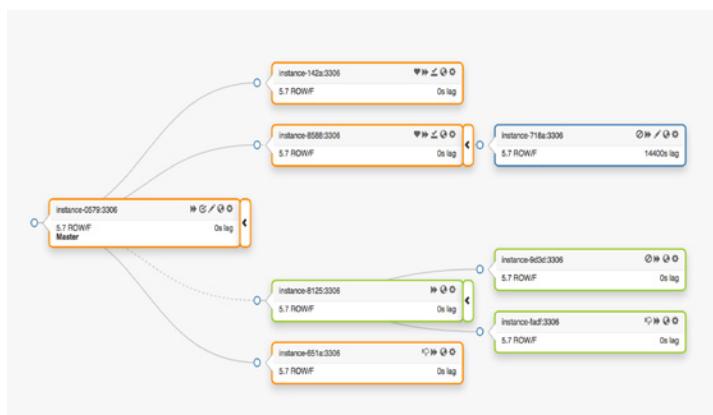
在我们之前的迭代中，我们使用：

- orchestrator用于故障监听和故障恢复
- VIP和DNS用于发现主节点

在那个迭代中，客户端通过使用一个名称，例如mysql-writer-1.github.net来发现写节点。这个名称解析为主节点获取的虚拟IP地址（Virtual IP address, VIP）。

因此，平常的时候，客户端会只解析这个名称，连接解析到的IP地址，然后找到正在另一端监听的主节点。

这个副本拓扑，跨越3个不同的数据中心：



当发生一个主节点故障事件时，一个新的服务器（副本之一），必须被提升为主节点。

orchestrator将监测到一个故障，提升一个新的主节点，然后采取行动重新分配名称/VIP。客户端并不准确地知道主节点的身份：它们所知道的只是一个名称，而那个名称现在一定解析到了新的主节点。然而，注意：

VIP是协作的：它们被数据库服务器本身声明和拥有。为了获取或释放一个VIP，一个服务器必须发送一个ARP请求。在新提升的主节点获取这个VIP之前，拥有这个VIP的服务器必须先释放这个VIP。这有一些不如

人意的效果：

- 一个故障恢复操作按顺序首先会请求挂掉的主节点释放VIP，然后请求新提升的主节点获取这个VIP。但如果老的主节点无法访问或者拒绝释放VIP呢？假设在那台服务器上一开始发生了一个故障，那么它也很可能不会及时响应或者根本不响应。
- 我们会面临裂脑处境：两个主机声称拥有相同的VIP。不同的客户端根据最短网络路径，可能会连接到其中任何一个服务器。
- 这个问题的根源是依赖于两个独立服务器的合作，而这种设置是不可靠的。
- 即使旧的主节点合作，这个工作流也浪费了宝贵的时间：切换到新的主节点时，需要等待与旧的主节点的联系。
- 而且当VIP改变时，不能保证现有的客户端与旧的服务器的连接断开，从而导致我们仍面临裂脑处境。

在我们的设置中，VIP与物理地址绑定。它们属于一个交换机或路由器。因此，我们只能将VIP重新分配给相互定位的服务器。特别是，在某些情况下，我们不能将VIP分配给在不同数据中心提升的服务器，并且必须更改DNS。

DNS的变化需要更长的时间来传播。客户端会为了预配置时间而缓存DNS名称。一个跨数据中心的故障意味着更长的宕机时间：让所有客户端意识到新主节点的身份需要花费更长时间。

仅仅这些限制就足以促使我们去寻找一种新的解决方案，但还有更多的顾虑：

- 主节点通过pt-heartbeat服务来自我注入心跳，从而达到[延迟测量和节流控制](#)的目的。这个服务必须在新提升的主节点上开启。如果可能的话，这个服务会在旧的主节点上会被关停。
- 同样地，[Pseudo-GTID](#)注入是由主节点自我管理的。它需要在新的主节点上开启，而且最好在旧的主节点上关停。

- 新的主节点被设置为可写的。如果可能的话，旧的主节点被设置为read_only。

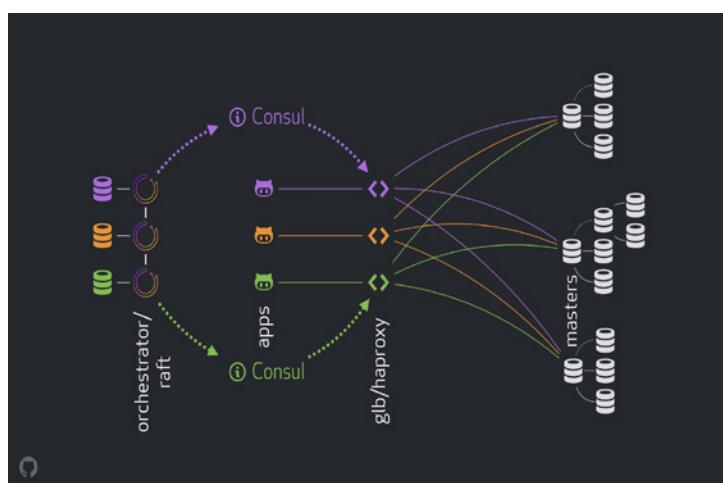
这些额外的步骤的执行时间构成了总宕机时间的一部分，并且引入了它们自己的故障和冲突。

这个解决方案是有效的，而且GitHub已经有运行良好的非常成功的MySQL故障恢复措施，但是我们想要在以下方面提升我们的高可用性：

- 不依赖数据中心。
- 克服数据中心故障的影响。
- 移除不可靠的协作工作流。
- 减少总宕机时间。
- 尽可能实现无损故障恢复。

GitHub 的高可用性方案：orchestrator、Consul 和 GLB

我们的新策略以及附带的改进，解决或者减轻了上述的许多担忧。在目前的高可用性设置中：



- orchestrator用来运行故障监听和故障恢复。我们使用了如下图所示的一个跨数据中心的orchestrator/raft。
- Hashicorp公司的用于服务发现的Consul。
- 作为客户端和写操作节点之间的代理层的GLB/HAProxy。

- 用于网络路由的anycast。

新设置完全删除了VIP和DNS更改。并且在引入更多组件的同时，我们使得组件解耦并简化了任务，还使用了稳健的解决方案。详解如下：

一个普通的工作流

平常，App通过GLB/HAProxy连接到写操作节点。

App不会意识到主节点的身份。和以前一样，它们使用一个名称。例如，cluster1的主节点会是mysql-writer-1.github.net。然而，在我们目前的设置中，这个名称会被解析到一个任播（anycast）IP。

通过anycast方法，这个名称在任何地方都被解析为相同的IP，但是流量会根据客户端位置分别进行路由。特别地，我们的每个数据中心都在多个区域部署了GLB（我们的高可用负载均衡）。到mysql-writer-1.github.net的流量通常路由到本地数据中心的GLB集群。因此，所有的客户端都是由本地代理服务的。

我们在HAProxy上运行GLB。我们的HAProxy有写操作池：每个MySQL集群一个池，而每个池有一个后端服务器作为这个集群的主节点。所有的GLB/HAProxy区域在所有的数据中心都拥有相同的写操作池，而它们都指向这些池中完全相同的后端服务器。因此，如果一个App想要向mysql-writer-1.github.net写入，这跟它与哪个GLB服务器连接无关。它将总是被路由到cluster1主节点。

就App而言，服务发现在GLB终止，并且永远不需要重新发现。流量都是在GLB上路由到正确的目的地。

那么，GLB如何知道将哪些服务器作为后端列表，以及我们如何将更改传播到GLB？

Consul的服务发现

Consul作为一种服务发现解决方案而闻名，并且还提供DNS服务。然而在我们的解决方案中，我们用它作为一个高可用的键值对（KV）存储器。

我们使用Consul的KV存储器写入集群主节点的身份。对于每个集群，都有一套KV记录表明集群的主节点的fqdn、port、ipv4和ipv6。

每个GLB/HAProxy节点都运行[consul-template](#)：一个监听Consul数据变化的服务（在我们的案例中：是指集群主节点数据的变化）。consul-template会生成一个有效的配置文件，并且能够基于配置的变化重新加载HAProxy。

因此，Consul中每个主节点身份的改变都被每个GLB/HAProxy观测，然后重新配置自身，将新的主节点设置为一个集群的后端池的单个实体，然后重新加载以反映那些变化。

在GitHub，我们在每个数据中心都有一个Consul设置，而且每个设置都是高可用的。然而，这些设置是彼此独立的。它们不会彼此复制，也不共享任何数据。

那么，Consul是如何得知变化的呢？这些信息又是如何跨平台分布的呢？

orchestrator/raft

我们运行一个orchestrator/raft设置：orchestrator节点通过raft共识相互通信。我们每个数据中心有1到2个orchestrator节点。

orchestrator负责故障检测、MySQL故障恢复并将主节点的变更通知Consul。故障恢复由单个orchestrator/raft领导节点维护，但是集群现在有一个新的主节点这个变更消息是通过raft机制传播给所有orchestrator节点的。

当orchestrator节点接收到主节点变更消息时，它们都会通知他们的本地Consul设置：它们各自调用一次KV写操作。拥有1个以上orchestrator的数据中心将会向Consul有多次（等同的）写操作。

整合工作流

在一个主节点宕机场景：

- orchestrator节点监测到故障。

- orchestrator/raft领导开始一次恢复措施，提升一个新的主节点。
- orchestrator/raft将主节点变更通告给所有raft集群节点
- 每个orchestrator/raft成员接收到一个领导变更通知。它们各自在本地Consul的KV存储器中更新新的主节点的身份。
- 每个GLB/HAProxy都运行了consul-template，监控Consul的KV存储中的变更，然后重新配置和加载HAProxy。
- 客户端流量被重定向到新的主节点。

每个组件都职责清晰，而且整个设计既解耦又简单。orchestrator不需要知道负载均衡器。Consul不需要知道信息来自哪里。代理只关心Consul。客户端只关心代理。

此外：

- 无需传播DNS变更。
- 没有TTL。
- 这个流程不需要挂掉的主节点的合作。它很大程度上被忽略了。

更多细节

为了进一步保障这个流程，我们还做了如下工作：

- HAProxy配置了一个非常短的hard-stop-after。当它用写操作池中的一个新的后端服务器重新加载时，它会自动终止任何现存的与旧的主节点的连接。
 - 通过hard-stop-after，我们甚至不需要来自客户端的配合，而且这样减轻了裂脑场景。值的注意的是，这并不是严密的，在我们杀死旧连接之前会过去一段时间。但是在那之后，我们就放心不会出现令人讨厌的意外。
- 我们并没有严格要求Consul在所有时间都是可用的。事实上，我们只需要它在故障恢复时可用。如果Consul碰巧挂掉了，GLB会继续使用上次已知的值操作，不会采取剧烈的行动。
- GLB被设置来验证新提升的主节点的身份。类似于我们的上下文

[感知MySQL池](#)，在后端服务器进行检查，来确认它确实是一个写操作节点。如果碰巧删除了Consul中的主节点信息，没有问题；空白的条目会被忽略。如果我们在Consul中误写入了一个非主节点服务器的名称，没有问题；GLB会拒绝更新它并使用上次已知的状态运行。

我们会在下面章节中进一步解决担忧并追求高可用性目标。

orchestrator/raft 故障检测

orchestrator使用一种[整体方案](#)来检测故障，因此是非常可靠的。我们不观测假阳性：我们不会过早启动故障恢复，因此不会遭受不必要的宕机时间。

orchestrator/raft进一步解决了一个完整的数据中心网络隔离的情况（即数据中心围栏）。数据中心网络隔离会引起混淆：那个数据中心中的服务器能够彼此通信。是它们与其它数据中心网络隔离了？还是其它数据中心被网络隔离了？

在一个orchestrator/raft设置中，raft领导节点是运行故障恢复的节点。领导节点是指获得大多数群体支持的节点。我们的orchestrator节点部署就是这样，没有单个数据中心占大多数支持，任何n-1个数据中心占大多数支持。

在一个完整的数据中心网络隔离事件中，那个数据中心中的orchestrator节点与其他数据中心中的对等节点断开连接。因此，在隔离的数据中心中的orchestrator节点不能成为raft集群的领导节点。如果任何这种节点碰巧成为领导节点，它也会下台。一个新的领导节点会从其它数据中心分配。这个领导节点将获得所有其它数据中心的支持，而这些数据中心能够彼此通信。

因此，orchestrator节点就是网络隔离的数据中心之外的一个节点。在一个隔离的数据中心应该有一个主节点，orchestrator将启动故障恢复，用可用数据中心之一里的一个服务器取代它。我们通过将决策委托给非隔离

数据中心中的群体来减轻数据中心隔离。

更快的通告

可以通过更快速地通告主节点变更来进一步减少总宕机时间。这如何实现呢？

当orchestrator开始故障恢复时，它观测可被提升的服务器群。理解复制原则并遵从暗示和限制，能够基于最佳做法作出优化的决策。

需要意识到，可用于提升的服务器也是一个理想的候选者，例如：

没有什么可以阻止服务器的提升（而且用户已经潜在暗示这些服务器是首选提升对象）

这些服务器能够将其所有的兄弟节点作为复制品

在这种情况下，orchestrator首先将服务器设置为可写的，然后迅速通告服务器的提升（写入Consul KV），同时异步开始修复复制树（这个操作通常会花费更多时间）。

很可能当我们的GLB服务器完全重新加载时，复制树已经完好无损了，但这不是严格必需的。服务器可以接收写操作！

半同步复制

在MySQL的半同步复制中，在变更已经提交到一个或多个副本之前，主服务器不会承认这个事务提交。这提供了一种实现无损故障恢复的方法：任何提交到主节点的变更都已经应用或者等待被应用到某个副本。

一致性伴随着成本：可用性风险。如果没有副本确认收到变更，主节点会阻塞并且写操作会停顿。幸运的是，有一个超时配置，超过超时时间，主节点能够恢复到异步复制模式，使得写操作再次可用。

我们将我们的超时配置设置为一个合理的低值：500ms。这足够将主节点的变更传递给本地数据中心副本以及远程的数据中心。有了这个超时，我们就可以观测完美的半同步行为（不回滚到异步复制），同时在确认失败的情况下会感受到一个可接受的非常短的阻塞时间。

我们在本地数据中心副本上启用半同步，而且在主节点挂掉事件中，我们期望（尽管并不严格强制）无损故障恢复。但是，我们不会期望一个完整的数据中心故障的无损故障恢复，因为它的代价非常大。

在进行半同步超时实验时，我们还观察到一种对我们有利的现象：我们能够在主节点故障中影响理想的候选者的身份。通过在指定服务器上启用半同步并将它们标记为候选者，我们能够通过影响故障结果来减少总宕机时间。我们在实验中观察到，我们通常能够提升理想的候选者并因此快速进行通告。

心跳注入

我们选择在任何地方任何时间管理pt-heartbeat服务的开启/关闭，而不是只在提升/降级的主节点上管理pt-heartbeat服务的开启/关闭。这需要一些补丁，改变它们的read_only状态或者完全奔溃，以便使pt-heartbeat与服务器一致。

在我们当前设置中，pt-heartbeat服务运行在主节点和副本上。在主节点上，它们生成心跳事件。在副本上，它们标识服务器是read_only并周期性检查它们的状态。一旦一个服务器被提升为主节点，那个服务器上的pt-heartbeat将其标识为可写的，并开始注入心跳事件。

orchestrator 所有权委托

我们进一步委托给orchestrator：

Preudo-GTID注入

将提升的主节点设置为可写的并清除它的复制状态

如果可能的话，将旧的主节点设置为read_only

在新的主节点上，这减少了摩擦。被提升的主节点明显需要是活跃的和可访问的，否则我们不会提升它。那么，可以让orchestrator直接将变更应用到提升的主节点上。

限制和缺陷

代理层使得App意识不到主节点的身份，同时它还对主节点屏蔽了App的身份。主节点看到的都是来自代理层的连接，而我们丢失了真正连接来源的信息。

随着分布式系统的发展，我们仍然面临未处理过的场景。

尤其是，在一个数据中心隔离场景中，假设主节点是在隔离的数据中心，那个数据中心的App仍然能够向主节点写入。一旦网络恢复，这可能导致状态不一致。我们通过从非常孤立的数据中心实现一个可靠的STONITH来减轻这种裂脑现象。像之前一样，主节点降级之前会经过一些时间，并且会存在一段时间的裂脑。避免裂脑现象的运维成本非常高。

存在更多场景：故障恢复时Consul宕机；部分数据中心隔离；其它场景等。我们明白，在这种性质的分布式系统中，不可能关闭所有的漏洞，因此我们关注最重要的场景。

结果

我们的orchestrator/GLB/Consul设置提供了：

- 可靠的故障检测
- 不依赖数据中心的故障恢复
- 典型的无损故障恢复
- 数据中心网络隔离支持
- 减轻裂脑现象（更多工作仍在进行中）
- 没有合作依赖
- 大部分场景下的10-13秒的总宕机时间
 - 总宕机时间在非常少的场景下会长达20秒，在极端情况下会长达25秒。

结论

orchestratoion/proxy/service-discovery范式在解耦架构中使用了众所周知且令人信赖的组件，使得它更容易部署、运维和观测，并且每个组件都可以独立地扩大或缩小规模。我们将不断测试我们的设置，从而不断寻求改进。

关于作者

Shlomi Noach是一名软件工程师、DBA和MySQL极客，供职于GitHub。[网站](#) | [GitHub简介](#) | [Twitter简介](#)。

运满满的技術架构演进之路

作者 王东



绝大多数人应该都用过滴滴、Uber 等叫车 App，所以在线叫车已经成为日常出行的一种方式。而运满满是货车和货源的模式，和滴滴有相似之处，也存在很多差异。货车司机以往是去线下的配货站找货，现在在运满满 App 上就可以寻找周边货源。

目前运满满有 520W 司机和 125W 货主用户。货运行业有其特殊性，我们也很荣幸能采访到运满满 CTO 王东老师，从运满满最初的架构迭代，到技术中台的搭建，到当前的 AI 技术的应用，整体上了解货运平台的技术积累。

王东说，之前一个司机从南京拉货到上海，需要在上海卸货之后再去

配货站找货，而且能看到的货源也仅限于与该配货站关系好的工厂货物，很多因素会导致司机要多等待半天到一天，以至于经常会空驶回南京。

基于运满满平台，司机能看到整个上海到南京的货源情况，甚至从上海周边到南京周边的货源，这就提升了司机的选择余地，极大的降低了空驶，也提高了车货匹配的效率，在热门线路货主发出送货需求几分钟之内就能找到对应的司机。

现在平台上的大部分运单运费并没有从运满满平台内走，运满满通过提供额外的保障，一体化服务的优势来吸引用户在线上完成运费的支付过程。

与此同时，王东还说，运满满正研究无人驾驶在干线物流领域的应用，已经组建了无人驾驶事业部，集团也为用户提供了全方位如 ETC，油，保险，贷款等服务。

现在这一切看上去都很美好，但是回望来时的路，不甚感慨，灭掉了很多问题，但还有更多的问题需要消灭。

一直在打“遭遇战”

2017 年运满满和货车帮合并后成立满帮集团，整个集团不论是业务体系还是技术体系都在飞速发展。为了更好的为司机和货主服务，集团各个业务、团队开始融合。融合的过程中，业务上既要满足不停歇的新业务需求，还要不断地升级系统以确保稳定性。合并之后和货车帮的技术团队很好的配合一起来完成这件事是一个挑战。结果只用了 3 个月就实现了系统融合并上线，双方团队的匠心精神充分展现。

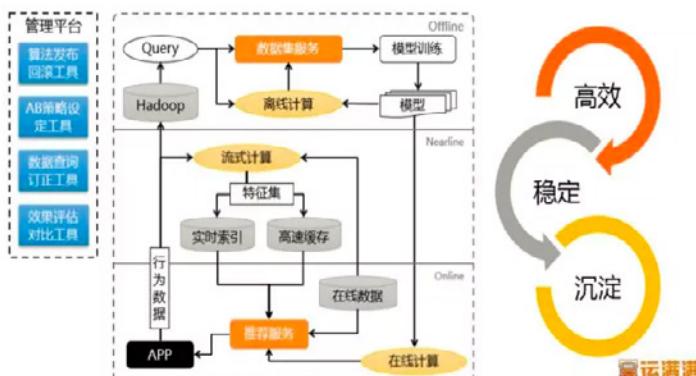
最开始运满满的系统架构设计比较简单：单体应用，一个 war 包包含了所有服务端接口，没有做服务化拆分，各模块严重耦合。随着业务量快速增长，在高峰时段系统访问量快速攀升，系统不断出现问题甚至宕机，很难诊断出问题在哪里。后来启动了服务化拆分、中心化建设。同时技术架构上开始大量使用缓存，以及数据库读写分离。App 重构项目、安全攻坚项目、运维自动化项目、Docker 化项目、稳定性体系项目也紧随其后。

系统架构迭代

第一次迭代是“服务化拆分”。运满满初期是一个单体应用，随着业务的发展开始隐隐暴露出研发效率与稳定性难以权衡的痛点，为避免业务进入高速发展时期对技术团队带来的冲击，研发团队启动了服务化拆分项目。对于系统的更新，我们从研发效率与稳定性两个基本要求出发，分析关键路径，隔离业务变化与资源体量，整体规划面向分布式系统的全链路监控。从系统复杂度、服务分级，以及业务领域对团队配合度的要求这几个方面，对团队进行盘点、重组和扩充。这个过程虽然花了很多时间与精力去分析系统，洞察团队，但事实证明这为后续的工作奠定了坚实的基础。

第二次迭代是“稳定性保障体系与业务服务中台”。在运满满系统全量服务化后，研发效率、性能、稳定性得到了巨大的提升，业务也在这个阶段进入了高速爆发期。然而，随着服务化大幅增加了系统复杂度，线上故障的定位难度与恢复时长也随之提高。研发团队围绕着故障预演、发现、止损、定位与恢复规划并落地了统一流控熔断降级、流量调度、动态分组、自动化破坏性测试、全链路 Trace、线上变更事件流等能力，大幅降低了故障数量与恢复时长。

推荐排序平台化赋能算法快速迭代



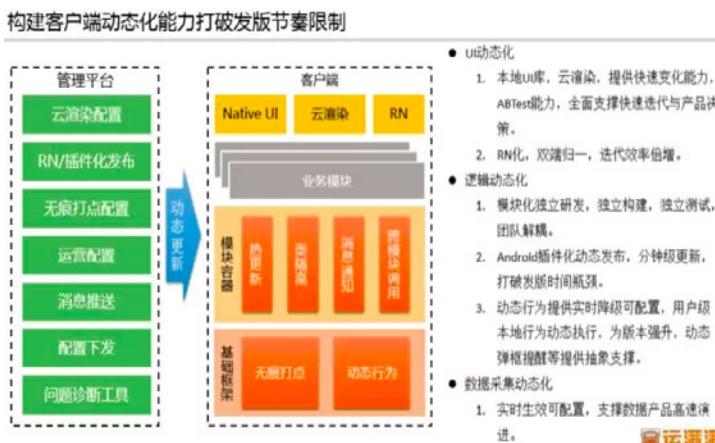
同样，系统复杂度的提升给研发效率也造成了不小的冲击，大量业务服务中存在相似或相同的逻辑，随着对业务领域的逐渐深入，我们规划并

逐步沉淀业务服务中台，以数据模型抽象化配置化，业务逻辑引擎化的思路，使大量前端业务系统的共性与差异化转变为中台调用的选项参数，以及配置能力的使用。比如用户平台、货源平台、推荐排序平台。

大道至简的分布时代应对策略

因为行业特殊性，缺少前车之鉴的参考，王东说，他们确实遇到了很多挑战，一些看似合理的产品逻辑与平台规则，在用户看来却没法解决他们的问题，反应很强烈。在一切摸着石头过河的阶段，对技术架构的快速应变能力是相当大的考验，客户端在这个问题上凸显尤为严重。客户端发版到用户最终更新需要一个长周期，如果完全依赖静态发版，版本更新周期内所有问题基本束手无策。为了快速响应业务变化，运满满对客户端动态能力建设下了很大的功夫，例如 React，动态降级 H5，安卓插件化，无痕打点。这需要大量的客户端标准化工作以及完备的基础能力建设，并且还需要建立对应的管理平台增加各能力管理的易用性。

另外，用户群体内大多数人对智能手机了解不够，例如Push收不到，没声音等手机设置问题都需要客服去协助解决。为此，我们在客户端内置了强大的问题自诊断工具，针对安卓系统的碎片化问题，这类工具的研发需要对症下药定制进行适配，达到的效果也很喜人，上线后每周此类问题的咨询量降低了两个数量级。为了降低管理成本与人员成本，这种快速应变能力在服务端同样重要，这正是业务服务中台能力的体现，非常重要。



大放异彩的算法技术

算法技术在车货之间进行匹配，最大程度降低空驶率，节省时间，提升效率上发挥了重大作用。那它和普通的车人匹配性质有何不同呢？

王东说，这就要谈到匹配的场景和特色。车货匹配在广义上，也是撮合交易的一种。如同电商、打车。在平台产品上的展现形态，也以推荐、排序、订单匹配为主。但车货匹配有极其独特的特点，比如货源是无库存的唯一品和非标准品。“唯一品”指的是每宗货源几乎各不相同，运输方案、时间各有变化，而且一次性成交后就立刻下线，完全不同于商城的热点商品推荐原则。非标准品是指，货源对车辆是有要求的，而且在不同时间、线路、种类上计价方式也不同。这一点也和打车出行场景的车人匹配有着重大差异。日常出行的车人匹配的场景是局部区域在较短时间窗口内满足供需，而车货匹配则是长时间大区域内的匹配 -- 毕竟货运计划可以长达一个月，车辆的行驶里程远大于日常打车场景。

算法技术，确实是以在这种强约束条件下取系统最优为目标的。迭代过程中，伴随算法的框架也做了大规模的升级，从数据采集存储、计算，到 T+1、T+0 的服务。实际上，算法解决了匹配方面的这几个子问题：车货基础相关性（CTR 模型），货源路线和司机画像，司机的预估行驶距离和时间（ETA 模型），区域内供需关系预测和价格模型，路线 / 货源相似性等。算法上，我们在离线部分使用 XgBoost 来处理基础相关性，用 RNN 做行驶时间预估，用 LSTM 来做时间序列预测，在线学习我们用基于 FTRL 独立发展的自研算法来处理。

从结果来说，发货信息上线的瞬间，就能准确预测潜在的车主，并进行正确的推送，40% 的货源在 30 分钟内就能建立司机到货主的联系，60% 的货源在 2 小时之内成交。大大节省了司机找货的时间成本。而且运满满还提供了大量回程甚至多路径中转回程的推荐。这些都对匹配效率，降低空驶起到了重要作用。

机器学习和深度学习的广泛采用

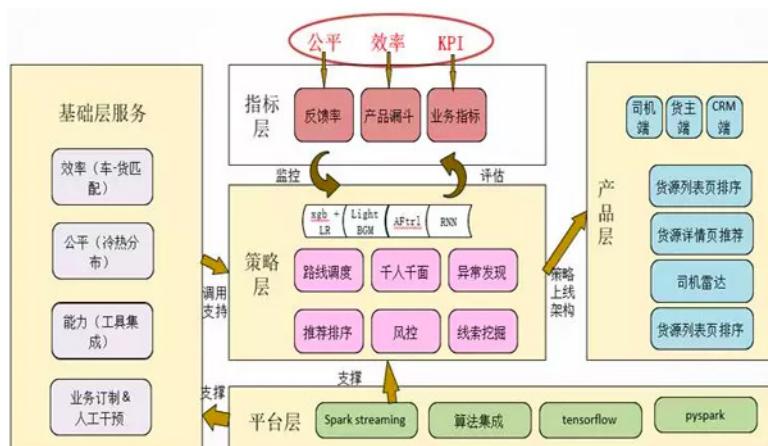
运满满利用大规模的数据采集，实时运算，借助机器学习和深度学习

技术，建立了全网最优为目标的匹配和调度方案，大幅提升运输效率（匹配，速度，节油）。

其平台的使用流程为：货主通过 App 发布货源信息，司机通过 App 筛选出符合运输条件的货源。由于在线货源的数量巨大，司机需要浏览大量货源，或频繁切换筛选条件才能找到真正合适的货源。因此开发了实时货源推荐系统和基于车辆货源供需的调度系统，来提高司机的搜货效率以及提升平台整体的成交率。除了个性化匹配外，实时性也是推荐的重点考虑要素。

具体技术细节，运满满使用 Xgboost 来预测车 - 货的基础相关性，实际是一个 CTR 和 CVR 混布模型，其中部署了在线实时系统，自研了一套基于 FTRL 算法的在线学习算法，将用户实时的行为数据结果和 Xgboost 的离线结果共同训练而得，点击预测的准确率超过 90%。但是“全网最优”的概念并不代表匹配速度最快。

货源尤其如此，有明显的好货与坏货的区分，因此要兼顾“效率 + 公平 + 业务目标”。比如在一个阶段内以 30 分钟反馈为业务目标。此外，还要考虑运输计划和车辆调度。又涉及到路线调度、ETA、供需预测等范畴。比如 ETA 是基于如下的 ETA cost 场景：为货主寻找合适的车辆，是不能通过周边车辆的直线距离来计算成本的，也要考虑限行、道路情况、天气季节、车辆状态来综合计算。那么在这里使用 RNN 来处理分裂后数千特征的输入，对到达时间的成本进行预测就是非常有效的一种方法。



因为以“全网最优”为目标，所以我们需要整合各个子系统以及相关的算法模型，如上图所示。

数据和算法产生的很大的经济价值也是很直观的。



作者简介

王东，运满满CTO。资深技术专家与管理者，曾先后负责过10多条亿级用户的产品研发管理工作，历任天猫高级技术专家、360高级总监、百度主任架构师，有过两次创业经验。热爱并信仰技术，技术功底深厚擅长分布式系统架构、海量数据拆分和分析、缓存和静态化技术使用、性能调优、无线端性能和动态化技术。并一直保持着对技术的热情和最前沿技术的探索。

百度智能运维的技术演进之路

作者 孙春鹭



随着大数据、人工智能、云计算技术的日渐成熟和飞速发展，传统的运维技术和解决方案已经不能满足需求，智能运维已成为运维的热点领域。同时，为了满足大流量、用户高质量体验和用户分布地域广的互联网应用场景，大型分布式系统的部署方式也成为了高效运维的必然之选。如何提升运维的能力和效率，是保障业务高可用所面临的最大挑战。

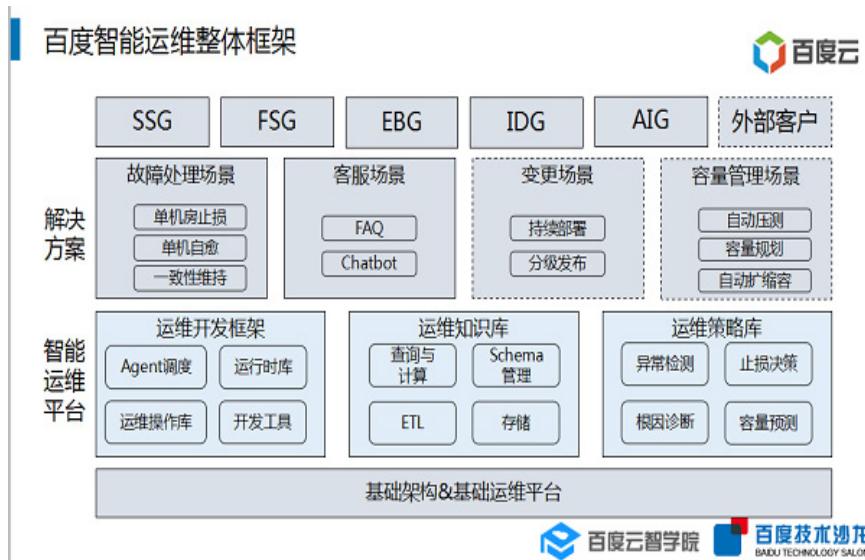
6月23日，由百度开发者中心、百度云智学院主办，极客邦科技承办的第79期百度技术沙龙邀请了来自百度智能云主任架构师王栋，百度智能云架构师哈晶晶，百度智能云资深运维架构师杨涛，百度智能云架构师章淼，百度智能云架构师余杰及百度智能云资深工程师廖洪流六位讲师，分

享百度在AIOps、DevOps上的实战经验，并以百度统一前端接入（Baidu Front End, BFE）、数据库以及Redis三个具体系统为例，介绍百度在系统架构设计和变更、监控、故障处理和性能管理等贯穿线上系统生命周期的运维层面上，如何保证系统的高可用。

高可用性系统的架构与运维实践

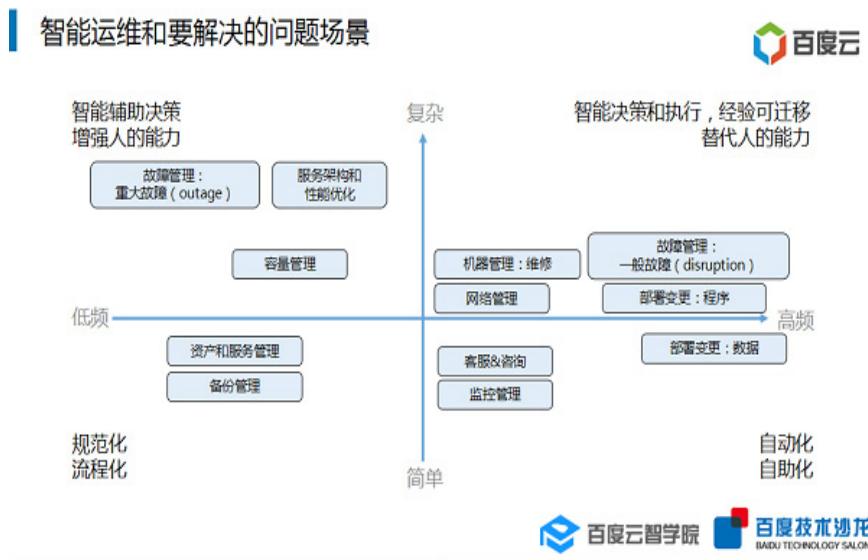
百度智能云主任架构师王栋做了开场演讲。他首先介绍了百度运维发展的历史，主要分为三个阶段：一、基础运维阶段。提供机器管理，服务管理和权限管理，保证线上基本服务运行，并对线上基本数据管理进行监控。二、开放运维时代。以开放API的形式，把第一阶段业务层面的运维交给各个业务部门。但是面临着垂直场景重复制造轮子，所积累运维知识和数据难以汇聚的问题。三、智能运维阶段。构建统一的运维知识库，一致的运维工具开发框架以及全局可见的算法复用平台。

下图为百度智能运维整体框架图。最下方是基础运维平台，提供最基本的基础运维能力，在此平台的基础上构建运维开发框架、运维知识库和运维策略库，在面临不同的场景和不同的业务将所有场景的算法抽样出来提供服务。



智能运维和要解决的问题场景

王栋现场对运维问题的复杂程度做了区分，如下图所示。纵轴表示问题的难易程度，横轴表示问题发生的频率。这样运维问题可以总结分成四个象限，对于每一个象限采取不同的应对措施。左上角低频高复杂问题，可以希望智能辅助决策，增强人的能力；右上角高频复杂问题，希望达到智能的决策，智能执行，并可迁移，而人只需做一些基本辅助工作即可；左下角低频且简单的场景，这是比较好解决的问题，只需把问题的解决策略规范化、流程化；右下角高频但是简单问题可通过自动化、自助化将问题解决。



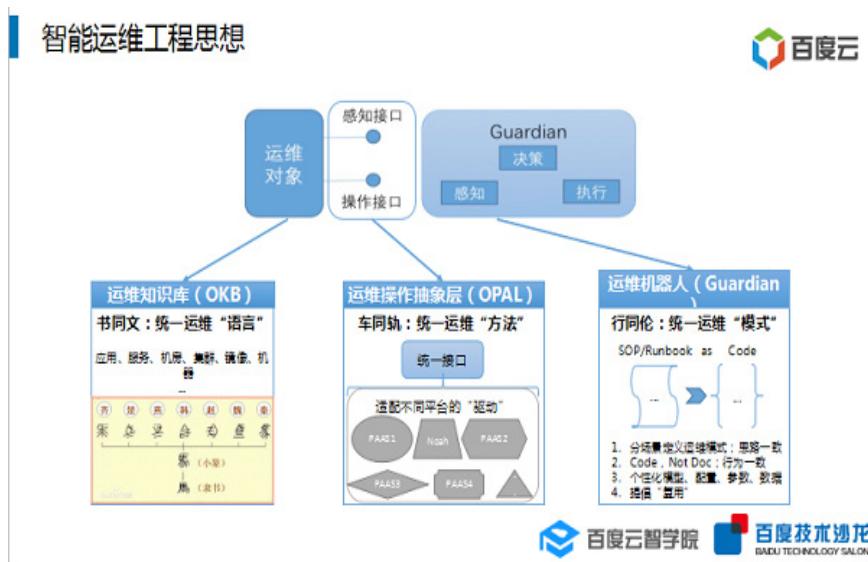
百度 AIOps 实践

百度运维经历了脚本&工具、基础运维平台、开放运维平台阶段，在2014年开始智能化运维的探索，并且围绕可用性、成本和效率方向的运维目标，在诸多运维场景落地。百度架构师，智能监控业务技术负责人，智能故障自愈方向技术负责人哈晶晶以百度故障处理场景为例，介绍百度故障预防的智能checker自动拦截异常变更，故障发现的异常检测算法，以

及故障自愈的单机房故障自动止损实践。

百度AIOps技术架构

百度智能运维将Gartner中提到AIOps的大数据和机器学习的理念应用于四大运维场景，开发成一系列的智能模型和策略，并融入到运维系统中，帮助提升运维自身的效率，以及解决传统运维方法所不能解决的挑战。



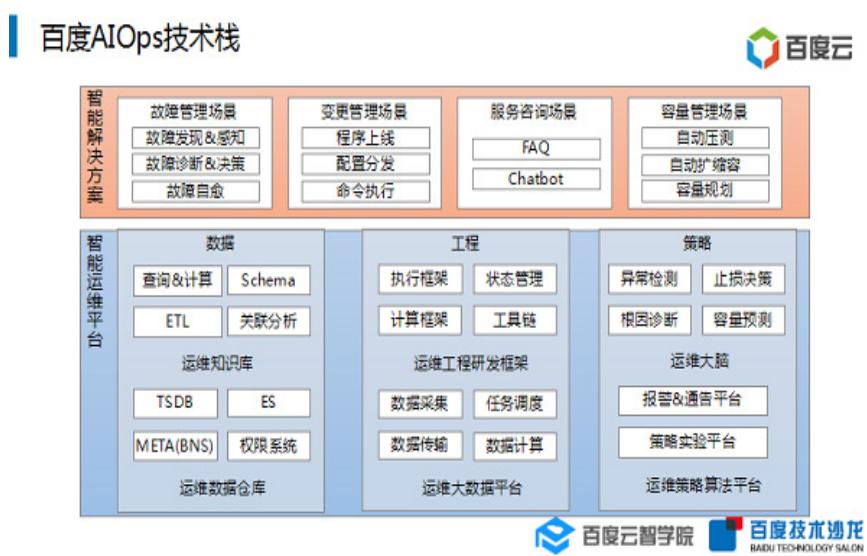
首先，为了解决不同业务线对运维对象的定义、操作接口、运维模式差异化的问题，百度提出了指导智能运维的三个原则：

- 书同文：一致运维“语言”；如运维应用、服务、机房、集群的定义；
- 车同轨：一致运维“方法”；如扩缩容执行、流量切换执行；
- 行同伦：一致运维“模式”；如故障诊断策略、弹性伸缩策略、流量调度策略。

根据以上AIOps中书同文、车同轨、行同伦的指导思想，百度基础运维和智能运维平台也聚焦在：数据、工程和策略三个方向。如下图所示。

- 数据方向：运维数据仓库&运维知识库

- 工程方向：运维大数据平台&运维工程研发框架
- 策略方向：运维策略算法平台和运维大脑（运维策略库）



该平台最终支持了故障管理、变更管理、服务咨询和容量管理场景的解决方案，并且应用到百度的内部、公有云和私有云客户。

运维知识库

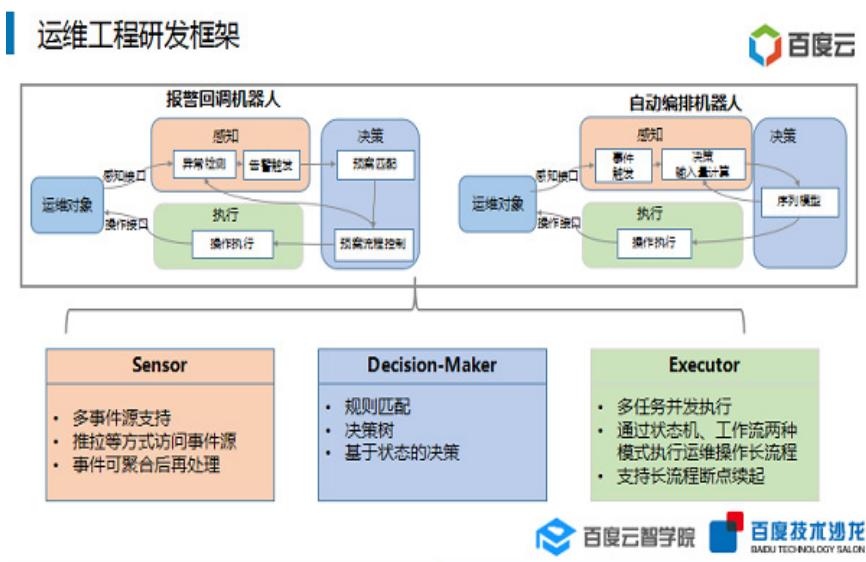
运维知识库是一个基于CMDB、数据仓库、知识图谱技术，对各类型运维数据，进行统一的ETL处理，形成一个完整的运维数据存储并且提供查询和使用的服务。该知识库系统功能第一要全第二要准，同时对整个架构的可用性要求较高，以便供运维使用。

运维数据分为元数据、状态数据和事件数据三大类：

- CMDB（MetaDB）：存储运维元数据和配置数据，包括不限于产品、人员、应用、服务、机器、实例、数据中心、网络等信息和关系。
- TSDB（基于HBase）：存储运维时序指标数据，包括不限于硬件和软件的可用性指标、资源使用率指标和性能指标。
- EventDB（基于Elasticsearch）：存储运维事件数据，包括不限于异常报警事件、故障处理事件、变更事件等。

运维工程研发框架

每个运维智能操作都可以分解成感知、决策、执行这样一个标准流程，这样一个流程叫做智能运维机器人。如下图所示。运维工程研发框架提供感知、决策、执行过程常用的组件，便于用户快速构建智能运维机器人。例如这三种组件可以组织成简单的报警回调机器人和自动编排机器人。报警回调机器人可以应用于故障自愈，自动编排机器人可用于分级变更。



先来看Sensor，Sensor是运维机器人的“眼睛”和“耳朵”。就像人有两个眼睛和两个耳朵一样。运维机器人也可以挂载多个Sensor来获取不同事件源的消息，比如监控的指标数据或者是报警事件，变更事件这些，甚至可以是一个定时器。这些消息可以通过推拉两种方式被Sensor获取到。这些消息也可以做一定的聚合，达到阈值再触发后续处理。

再来看Decision-Maker，DM是运维机器人的“大脑”，所以为了保证决策的唯一，机器人有且只能有一个DM，DM也是使用者主要扩展实现的部分。除了常见的逻辑判断规则之外，未来我们还会加入决策树等模型，让运维机器人自主控制决策路径。

最后看Executor，执行器是运维机器人的手脚，所以同样的，执行器可以并行的执行多个不同的任务。执行器将运维长流程抽象成状态机和工作流两种模式。这样框架就可以记住当前的执行状态，如果运维机器人发生了故障迁移，还可以按照已经执行的状态让长流程断点续起。

运维大脑

有了数据和工程就有运维大脑。运维大脑包括异常检测和故障诊断，这两个部分的共同基础是基本的恒定阈值异常检测算法。恒定阈值异常检测算法利用多种概率模型估计数据的概率分布，并由此产生报警阈值。在数据的特点随时间发生改变时，算法可以利用最近的数据修正概率模型，自动产生新的报警阈值。

由于许多数据具有上下波动的特性，恒定阈值法不能很好的描述数据的特点，所以百度发展了基于环比和基于同比的异常检测方法。环比的异常检测方法假设输入的时序数据总体上比较平滑，通过平滑算法预测数据的基准值。然后将数据的观测值与基准值相减即可获得残差，恒定阈值算法应用在残差上就能够检测异常。环比方法主要用于检测突增突降类型的异常，但是有些数据在发生异常时上涨或者下跌比较平缓，这就是环比算法无法胜任的了。

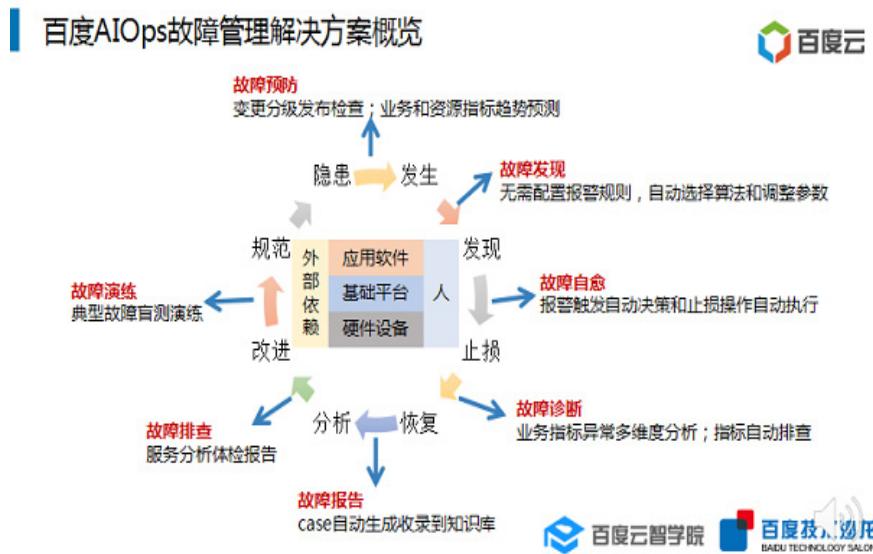
在故障诊断方面，百度基于异常检测算法开发了指标自动排查算法和多维度分析算法。指标自动排查算法能够自动扫描所有监控指标，并筛选出在故障发生前后发生剧烈变化的异常指标。

然后算法将这些异常指标整理为异常pattern，并将异常pattern排序，把与故障原因最相关的异常pattern呈现给运维工程师。

而多维度分析聚焦于带有多个tag的业务数据。它先利用异常检测算法标记异常的业务数据，然后利用信息理论的方法寻找覆盖多数异常的tag组合。这些tag组合常常可以直接指明故障的原因。如果把每个tag看作是高维空间的一个维度，异常数据相当于分布在一个超立方体中的点。寻找覆盖最多点的子立方体，所以称为多维度分析。

故障管理 AIOps 实践

故障的完整生命周期包括隐患阶段、故障发生、故障发现、故障止损、故障恢复、故障分析、故障改进和故障规范阶段，每个阶段都可以使用AIOps相关的方法提升故障管理的质量。如下图所示。



故障预防实践

互联网企业产品迭代的速度非常之快，但是有变化就会有风险，2017年的服务故障有54%是来源于发布，release是当之无愧的服务稳定性第一杀手。基于此问题，百度提出了不同的预防措施：

1. 从测试流量到真实流量，百度首先部署sandbox，这种情况下是无损失的
2. 从一个IDC到更多IDC，百度挑选流量最少的IDC，异常情况下损失较少，或者可以快速切流量止损
3. 从少数实例到更多的实例：百度先部署某个机房的1%，再部署99%

有了合理的stage，就可以基于发布平台做自动化检查的工作。在每个stage结束之后，会自动检查是否有报警发生，如果有则会停止变更。

变更通常会检查可用性指标、系统相关指标和业务逻辑类的指标。但是人工检查的时候会遇到以下问题：指标覆盖率不会很高，阈值设置困难导致的漏报&误报。使用智能Checker的程序自动检查方法可以解决这些问题。

故障发现实践

我们面临的业务种类繁多，业务指标类型众多，比如请求数、拒绝数、响应时间、流水和订单等类型的数据。不同业务不同数据的曲线，波动模式也不一样，在监控阈值配置时通常会遇到以下的问题：1.不同的监控项需要应用不同的算法。2.忙时&闲时、工作日&休息日阈值设置不同。3.后期随着业务发展需要不断完善阈值配置。4.监控指标爆发式增长，配置成本极高。

在这样的背景下，我们对数据进行分类，针对不同的场景提供不同的异常检测算法解决人工配置困难和监控漏报&误报的问题。

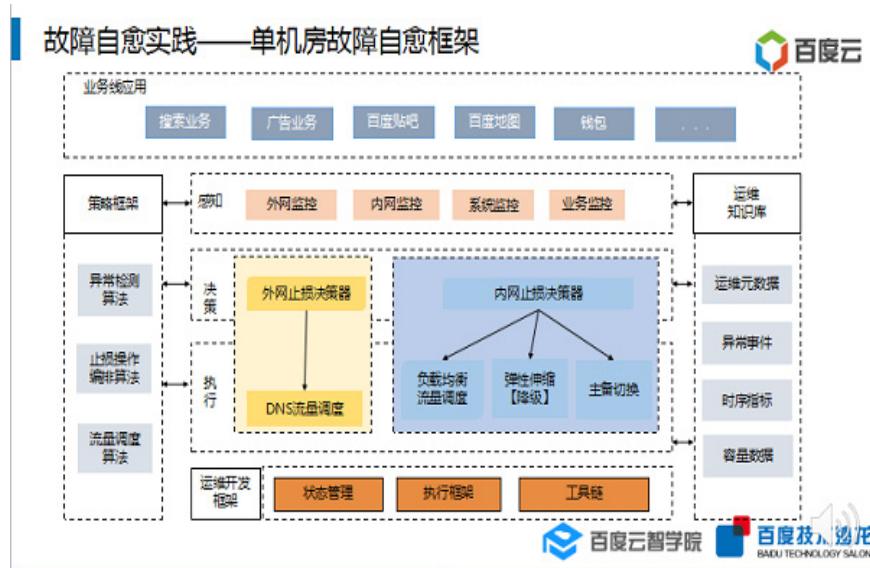
- 周期波动的数据，典型场景：广告收入、搜索流量等。算法：同比基准值检测
- 关心突变的数据，典型场景：交易订单、流水等。算法：环比基准值检测
- 关心是否超出了一定波动范围的数据，典型场景：pvlost。算法：基于概率的恒定阈值检测。

故障自愈实践

人工处理故障通常面临着响应时间不够迅速、决策不够精准、执行误操作的情况发生。故障自愈是通过自动化、智能化处理故障节省人力投入，通过设定的处理流程提高故障处理可靠性，同时降低故障时间，为业务可用性保驾护航。

单机房故障是业务的常见故障，百度的核心业务线均实现了2到5分钟内的故障自愈。如下图所示，整个这个框架充分利用了前面提到的运维策略库、运维开发框架和运维知识库构建单机房故障自愈程序。整个自愈程

序也是感知、决策、执行判断的。自愈程序分两个，一个是机房外网入口故障，通过外网监控发现问题，通过DNS流量调度来解决；另一个是在百度内网机房故障和业务单集群故障，通过业务监控发现故障，通过内网流量流量调度、服务降级和主备切换多种手段结合进行止损。



大规模数据中心变更风险应对之道

在大规模数据中心中，对生产环境的变更来自于各个方面，有机器类操作、机器环境变更、服务操作等等。这些变更无论是自动化的还是手动的，任何一次变更都会带来服务稳定性风险。百度智能云资深运维架构师杨涛从具体的案例出发，介绍百度应对变更风险的防御机制演变及最佳实践。

变更是什么？

变更就是对于生产环境，也就是线上环境进行的任何非只读动作。比如说最基础的机房网络调整变更、物理机重装重启、基础环境变更、容器实例的变更等等。这些变更有很多来源，以前最主要来源是人工，根据业务需求或者稳定性的需求进行；另外一个主要的来源是自动触发，包括发布流水线、机器自愈系统、弹性扩缩容等。

历史上出现的三次 Case

杨涛首先介绍了百度历史上出现的三次 Case:

- 误操作导致网页数据库被大规模删除
- 程序 Bug 导致网页数据库丢失 1% 数据
- 程序 Bug 导致少量虚拟机被 Kill

然后从 Case 中分析出了变更的基本模式:

1. 方案审核: 所有线上变更方案, 均需要进行方案Review
2. 变更检查: 线上变更之前以及完成后, 需要按照检查列表进行检查, 保证服务正常
3. 分级操作: 并发度、间隔、小流量、抽样、分组操作

但是同时提出, 最大的困难是如何指定合理的机制来确保所有人和系统都遵守变更的基本模式。

变更怎么做?

杨涛介绍了变更的四大风险, 其实本质上就是人的风险:

第一是操作不一致的风险

操作内容受操作者自身经验、知识深度、对服务的了解程度、稳定性意识而不同, 从而制定出完全不同的变更方案, 并有不同的变更流程。

解决方案有二, 一是制定流程规范, 变更之后要有变更方案评审。百度的实践是一周完成全部变更计划, 然后再审核、发单、检查; 二是制定标准SOP手册, 形成指导日常工作的规范, 所有的人参照标准的 SOP 进行线上变更, 从而保证操作内容一致性。

第二是操作不准确的风险

变更方案和具体实际执行不一致, 特别是手动的误操作的风险。这个解决方案就是运维最基本的能力 - 自动化。而 SOP 进行自动化的时候, 需要有先后顺序, 主要根据如下标准选择: 复杂程度, 风险程度, 操作频次。

第三是流程退化的风险

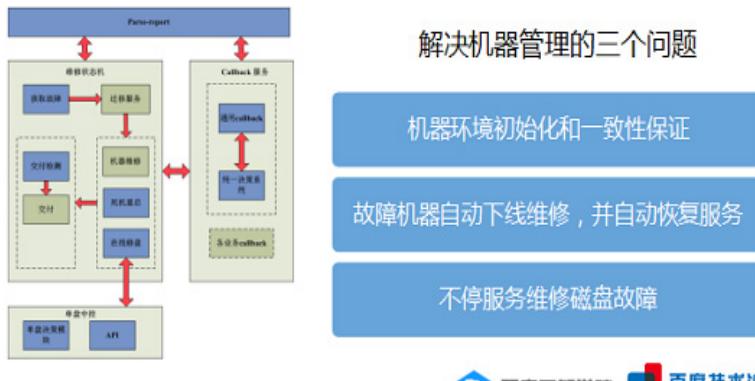
流程存在退化情况，刚开始遵守流程，随着时间的推移，例外越来越多；另外自动化的脚本或系统，维护成本较高，其很难实现全流程（如变更自动检查）。于是可以通过软件工程的方式解决问题 – 平台化。平台化的要点是：使用API关联相关系统，提供稳定有效的服务，对基础流程进行标准化，保证流程可执行。

以Ultron为例，它解决了机器管理的三个问题。第一个问题是怎么做机器环境初始化和怎么保证线上所有机器一致性。第二个是故障机器自动维修，自动恢复服务。第三个问题是如何在不停服务维修磁盘故障。Ultron 中每一台机器都有一个状态机，依赖百度的标准化服务，当机器发生硬件故障时进行服务迁移，维修成功后又加入到资源池，保证容量的稳定性。

变更怎么做 – 平台化：解决流程退化的风险



典型案例 – Ultron 机器流转系统



第四是检查不充分的风险

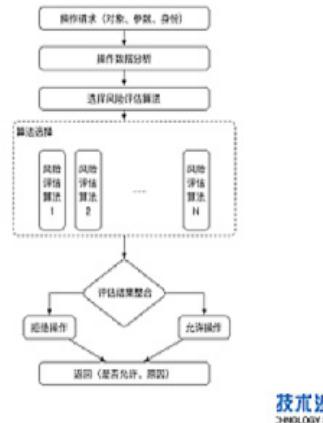
解决方法是检查机制，分为变更后的检查和变更前的检查。变更后的检查主要是通过联动 SLA 系统、监控系统、冒烟平台等第三方系统，进行变更效果检查。

而变更前的检查，则重点关注操作风险的防御，在操作尚未实施的时候就将危险操作拦截住，从而保证线上服务的稳定性。

变更怎么做 – 检查即服务：解决检查不充分的风险

百度云

- 变更前检查: 通用的防御机制
- 风险评估算法示例
 - **限额**: 每小时只允许进行 $<=3$ 次宕机自动修复
 - **并发度**: 同时只能有 $<=100$ 台机器的Agent在升级中
 - **SLI**: 服务可用性低于99.99%时不允许进行新版本发布
 - **互斥**: 当网络进行调整时, 不允许下线机器
 - **正确性检查**: 当机器上有VM时, 不允许重装宿主机

技术沙龙
TECHNOLOGY SALON

更多的问题

最后, 杨涛以一个开放性的 Case 结束了本次分享:

- 当自动化平台不可用的时候, 人工执行了虚机迁移操作, 操作错误出现故障

这个 Case 引申的问题很多:

- 如何保证自动化平台的高可用以及进行风险控制
- 如何保证人在习惯了自动化平台后, 不丧失故障处理的能力

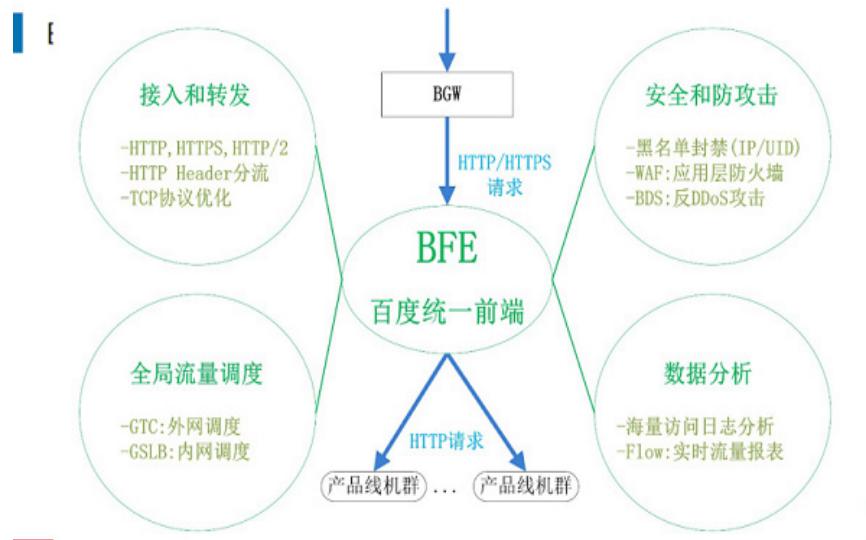
百度统一前端平台技术面面观

网络接入服务是用户和后台服务间的桥梁, 对服务质量影响巨大。百度智能云架构师章淼介绍了BFE研发中包括网络协议、网络安全、高性能系统在内的多个技术方向, 以及提升平台稳定性和研发效率的研发方法优化。

百度统一前端

百度统一前端BFE分为四个版块, 上游是四层的网关BEW, 下面作为七层的转发网关具有七大功能, 第一是转发, 包括多种协议, 除了最基本的HTTP, HTTPS, 还有HTTP2。第二是流量调度, 一个是外部还有一个

内网。第三是安全和反攻击。第四个是海量访问日志分析与做实时流量报表。

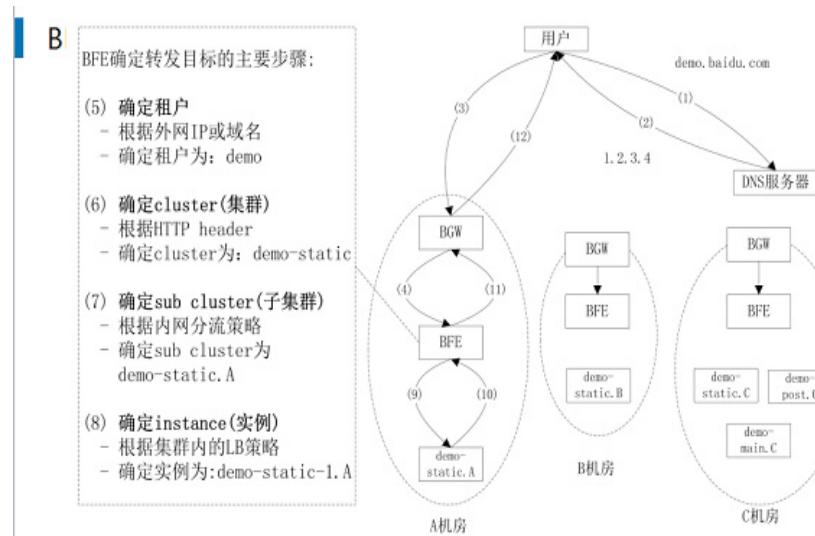


BFE若干技术点的深入

章淼将BFE的技术点分了四个方向，首先是接入转发，第二点全局流量调度系统，第三点数据分析，第四点平台运维和运营。

转发模型

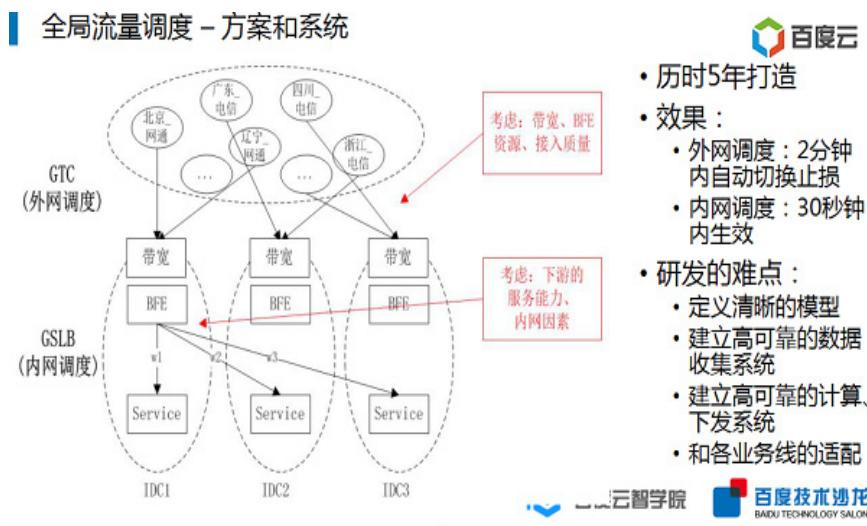
如下图所示为BFE基础转发的步骤。用户解析域名，当达到第三步



时，可以拿到IP地址请求四层网关BW，BW会把流量转到BFE。左侧是BFE四步要做的工作，首先根据外网的IP或者运营确定租户，第二步确定它属于哪一个集群，第三步是BFE要确定它属于哪一个子集群，最后确定把这个流量打到哪一个实例。

全流量调度系统

全局流量调度架构分为两层：GTC(外网) + GSLB(内网)。下面是内网调度，任务是把BFE接入到流量，转发到下流多个应用的集群上。这个机制在2013年上线，当出现问题时可以通过内网调度执行。内网的处理百度主要考虑了两个因素，首先是到BFE流量，第二是考虑下游的流量是什么样的，同时考虑内网的因素，以本地优先为原则，如果出现流量大于本地流量的情况下，要负载均衡这是内网。



数据分析

章森介绍了数据分析在BFE的价值，首先可以产生业务相关的报表，还可以用它了解下游集群的健康状况，另外还可以感知外部网络的状况。

那么BFE是如何实现准实时流量报表呢？百度自己定义了一个系统，内部称之为FLOW。采用多级的方式，在BFE做一次汇计算，把汇计

算结果打到一级汇聚，打到二级汇聚，最后把数据结果存十几个数据库 TSDB。

平台运维和运营

运维：保证整个平台的稳定运行

- 监控：转发引擎对外暴露数千个变量

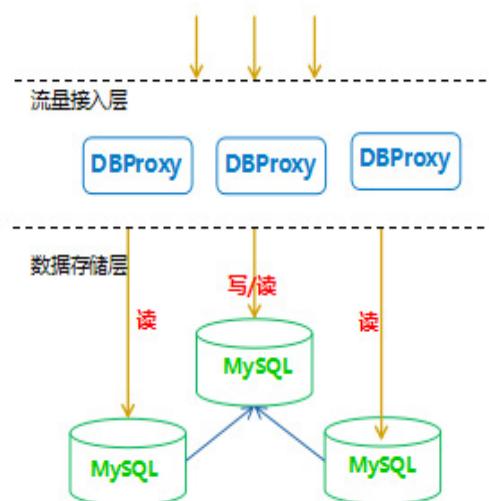
运营：提高用户的满意度

- 投入2年以上时间研发用户平台
- 用户配置在2分钟内自动下发生效
- 每个租户都有独立的数据报表
- 完整的用户手册

百度数据库运维 Redis 异地多活实践

最后，由百度智能云架构师余杰和百度智能云资深工程师廖洪流共同介绍百度DBA的MySQL服务和百度Redis异地多活实践。全面呈现百度MySQL服务生命周期内服务运维保障以及百度在使用分布式缓存系统时会遇到的问题以及对应架构的演化过程。

数据库高可用



当前百度MySQL提供的架构为三层架构，业务方面使用的是BGW方式以及内部的BNS服务，中间层为自研中间层的代理，最底层为MySQL集群服务。如上图所示。

对 MHA 架构的调研：

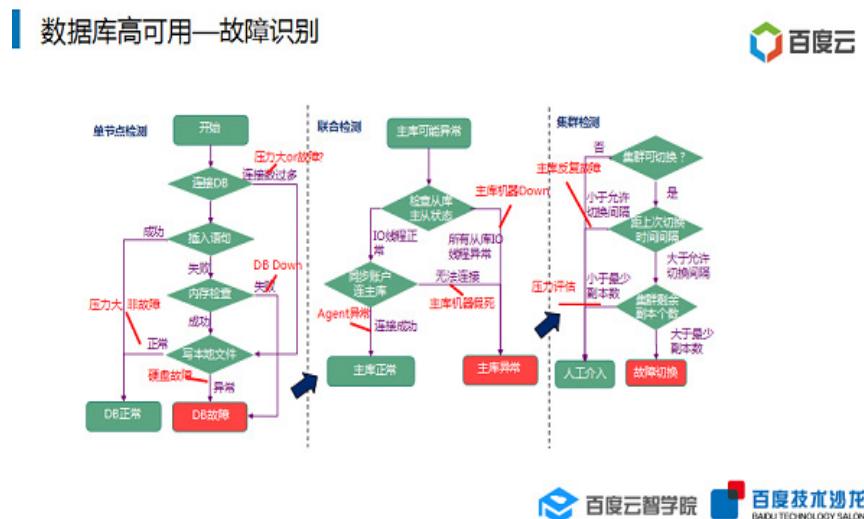
MHA（Master High Availability）是一套优秀的作为MySQL高可用性环境下故障切换的高可用软件。在考虑不用代理的情况下，使用Manager提供的服务，直接对租户进行对接，可以处理在一些简单场景下对于高可用的需求，且MHA内部有一些数据补齐的能力和处理方式。

MHA 无法满足百度当前面临的需求，原因如下：

- 故障识别方面的一些处理方式无法满足当前遇到的场景；
- 由于MHA对集群内部信任关系的强依赖，出于对安全方面的考虑，百度不允许在上万台机器之间建设信任关系；
- 还有一些数据补齐，选取主库过程的一些问题。

数据库高可用—故障识别

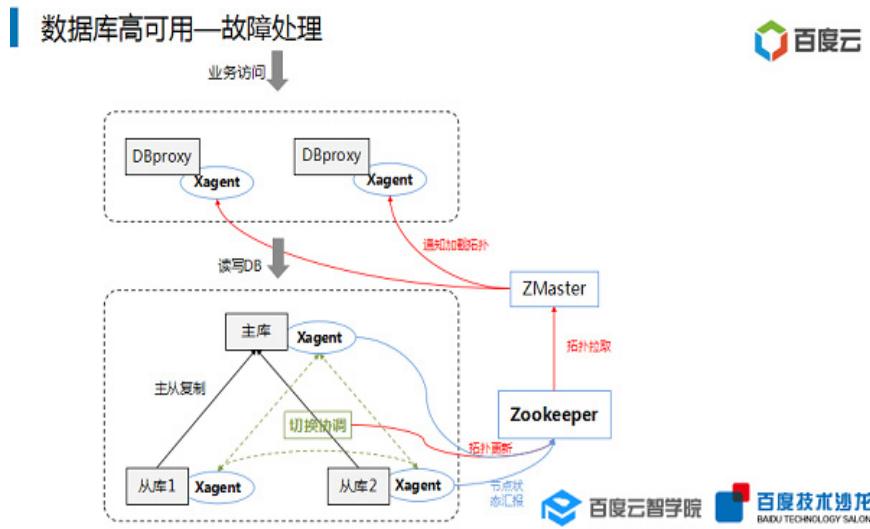
结合完整数据库内部识别故障。首先收集节点信息以及状态，查看连接数，判断是否是由于MySQL实例自身的压力问题或其他问题导致感知到DB有异常的状态，进而上升到联合从库的信息检测当前的主库是不是正常。检测感知异常是否是由于假死或压力过大，然后上升集群内部的联



合诊治机制。最终上升到整体数据库APP检测机制，以此来决策到底要进行怎样的切换。同时，在切换时要考虑主从之间延时的问题。基于前面的感知，以及识别，做真正的故障处理。

数据库高可用—故障处理

当在前面的识别阶段感知到做的主从切换的时候，百度会在代理层把主库完全替换掉，这个问题在一定程度解决切换的过程中出现主库重新写的问题。接下来就是选择主库的过程，当真正拓扑完成后，会将完成信息通过网通节点发送至代理节点。这一选取新主库的过程，就是进行故障处理的过程。



数据库高可用—脑裂问题解决

对这一问题的解决方案为引入第三方仲裁机制和机房区域内分机房的监测机制。通过两个管控节点，一个是区域内的Agent，另外一个是全局管控节点，识别是否可以和其他区域内的实例通信，进一步判断是否属于区域性的网络问题，以及是否会出现脑裂，通过一系列的机制，来制定相应的决策。

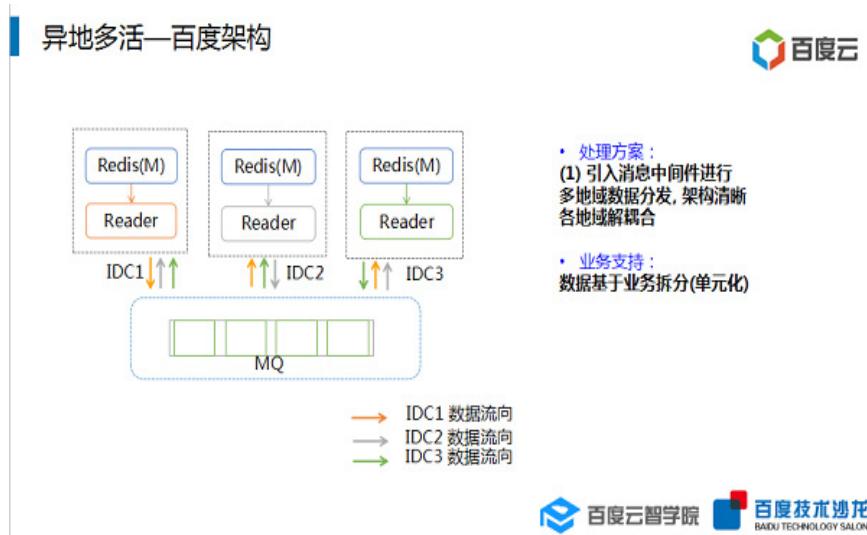
一旦识别当前出现区域性网络问题，Zmaster可以暂停本区域有主

库，屏蔽区域内不可管制的部分实例的信息，杜绝了出现脑裂问题。

区域网络恢复后，Zmaster和Xmaster会检测整体主从架构，再恢复区域网络代理信息，最终通过自动化方式，恢复至整体可用。

Redis 异地多活

随着业务发展，百度需要将服务部署至多个地域，同时要求数据一致。为了满足这个需求，百度提出了主地域的概念，所有数据写到主地域，其他从地域通过Redis自带的复制功能实现同步，这样就实现了不同地域间的数据同步。同时考虑到多地域之间数据主机房出现故障能够止损自愈，百度对整机房切换方案进行了支持。另外由于考虑到服务有可能不断扩容的需求，实现了在线扩容。



百度Redis架构是如图所示。设置一个Reader，和地域之间的关系是1: 1。每一个Redis只对应一个Reader，而这个Reader同步数据的目的地不是其他地域的Redis，而是一个消息中间件，通过消息中间件的转发能力，实现地域的同步，而所有的Reader只负责将本地的信息传到Redis。

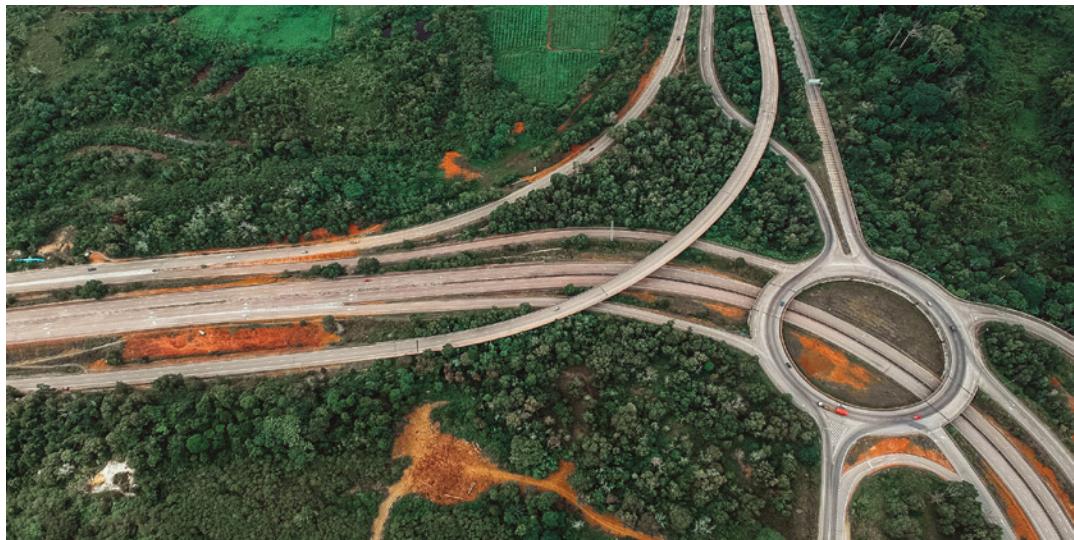
但是在真正实践方案时会遇到什么技术难点？

Redis跟Reader数据同步采取什么方案？很自然用Redis主从同步来做。但是主从同步是可靠的吗？先简单回顾一下Redis原生的增量同步的

方案是什么，原生的增量同步是数据写入了Redis Master，Redis Master有一个环形队列，当Redis跟Master进行数据同步的时候，它会先尝试使用它当前同步点，就是Offset，当这个Offset在这个里面会一次性同步给Slave。但是这里面存在什么问题呢？当你的Offset不在这个序列里面，这个存在全量同步，同时还有一个问题在于它为了保证数据一致性，Slave进行全量同步的时候，先将自己本身数据清掉，清掉以后再进行同步。所以针对这个问题百度做了一些思考，在AOF基础上做了一些调整。采用按照一定大小进行切割的方式，同时引入OPID的概念。每一次操作名由OPID决定，这样从库拿原生的命令，还有OPID的信息，如果主从关系断了，它会拿现在OPID请求Master，Master会查找，找到这个OPID，并基于AOF的数据进行同步。

面向大规模 AI 在线推理的可靠性设计

作者 宋翔



概览

在 AI 项目中，大多时候开发者的关注点都集中在如何进行训练、如何调优模型、如何达到满意的识别率上面。但对于一个完整项目来说，通常是需求推动项目，同时，项目也最终要落到实际业务中来满足需求。

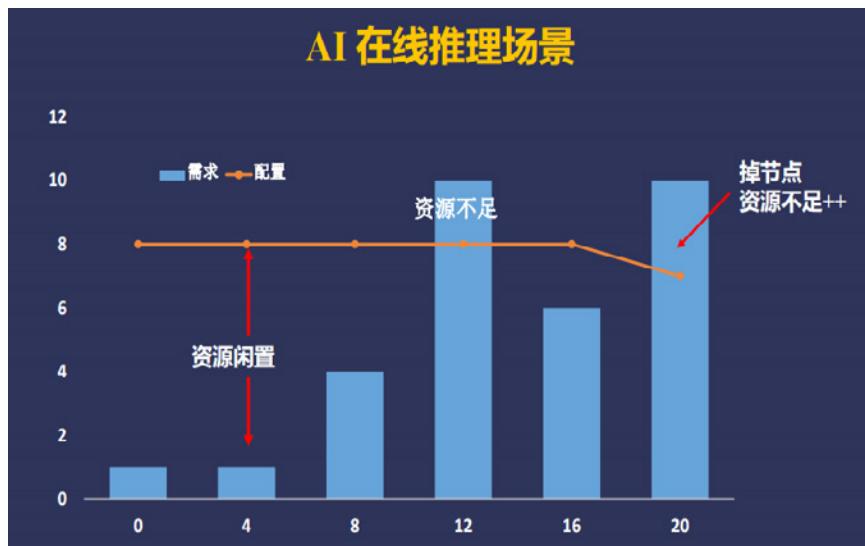
对于常用的 AI 训练和机器学习工具如 TensorFlow，它本身也提供了 AI serving 工具 TensorFlow Serving。利用此工具，可以将训练好的模型简单保存为模型文件，然后通过的脚本在 TensorFlow Serving 加载模型，输入待推理数据，得到推理结果。

但与拥有较固定计算周期和运行时长的 AI 训练不同，AI 推理的调用会随着业务的涨落而涨落，经常出现类似白天高、夜间低的现象。且在大规模高并发的节点需求情况下，常规的部署方案，明显无法满足此类需求，此时需要使用更专业的 AI 推理模型和扩缩容、负载均衡等技术完成预测推理。

UAI Inference 采用类似 Serverless 的架构，通过请求调度算法、定制扩缩容策略，自动完成 AI 请求的负载均衡，实行节点动态扩容和回收，可提供数万的 AI 在线推理服务节点。

某 AI 在线推理一天内的请求访问情况

AI 推理（Inference）的在线执行有两大关键因素：一是通过 GPU/CPU 对数据进行快速决策，二是对访问请求的实时响应。下图为某一 AI 在线推理场景 24 小时内的资源使用情况，其中，横轴为时间、纵轴为用户资源请求量，橙色线现表示资源配置情况。



凌晨 00:00-8:00 点，用户基本处于睡眠状态，此刻的资源请求较少，闲置资源较多；8:00 以后，手机等设备使用量增多，推理访问请求逐渐上升；直至中午，设备访问达到高峰，请求量超过设定的资源量，系统纺问

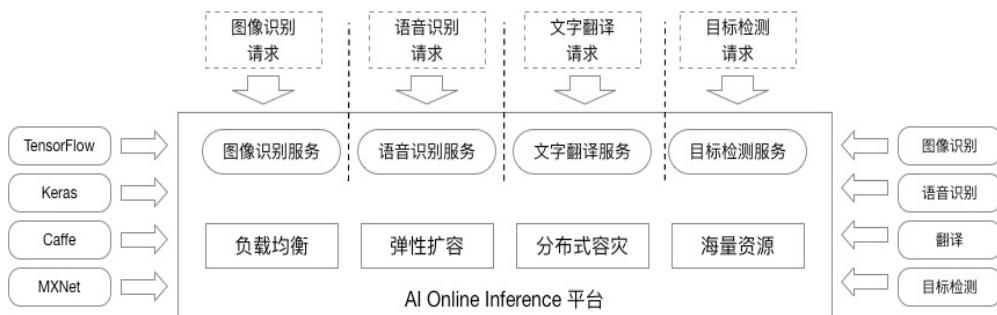
出现延迟；之后在线使用量降低，部分资源又将闲置……

可以看到，一天内不同的时间段，访问量会随着用户作息规律而出现相应的起伏，若是将资源配置设置过小，则会导致计算资源不足，系统吞吐量变低，致使访问延迟。但若投入过多的配置，又会产生大量的闲置资源，增加成本。

面向大规模的 AI 分布式在线推理设计与实现

UAI Inference 整体架构

为了应对在线推理对实时扩缩容以及大规模节点的需求，UAI Inference 在每一台虚拟机上都部署一个AI在线服务计算节点，以类似 Serverless 的架构，通过 SDK 工具包和 AI 在线服务 PaaS 平台，来加载训练模型并处理推理（Inference）请求。整体架构如下：

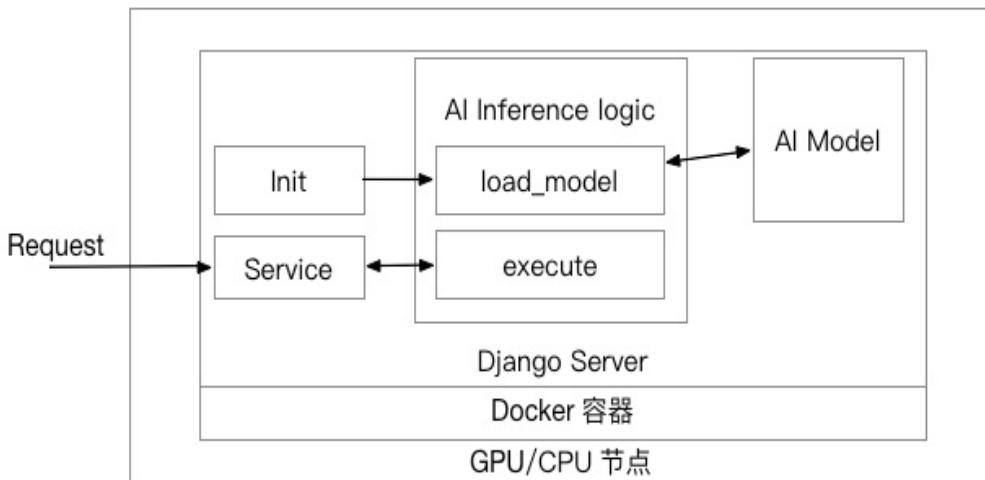


SDK 工具包：主要负责模型加载。包含接口代码框架、代码和数据打包模板以及第三方依赖库描述模板。用户根据 SDK 工具包内的代码框架编写接口代码，准备好相关代码和 AI 模型以及第三方库列表，然后通过打包工具将训练模型进行打包。

任务打包完毕后，系统自动将业务部署在 AI 在线推理 PaaS 平台上处理推理请求。这里，平台每个计算节点都是同构的，节点具有相等的计算能力，以保证系统的负载均衡能力。此外，动态扩缩容、分布式容灾等弹性可靠设计也是基于该平台实现。

在线推理实现原理

在实现上，系统主要采用CPU/GPU 计算节点来提供推理任务的基础算力，通过Docker容器技术封装训练任务，内置 Django Server 来接受外部HTTP请求。下图展现了处理请求的简单原理与流程：



在初始化过程中（init），Django Server 会先根据 conf.json 加载 AI Inference 模块，然后调用该模块的 load_model 将 AI 模型加载到 Django Http 服务器中；在处理推理请求时，Django 服务器会接受外部的http请求，然后再调用 execute 函数来执行推理任务并返回结果。

这里，采用容器技术的好处是可以将运行环境完全隔离，不同任务之间不会产生软件冲突，只要这些 AI 服务在平台节点上运行满足延时要求，就可进行在AI线推理服务部署。

功能特性

UAI Inference适用于常见的大规模 AI 在线服务场景，如图像识别、自然语言处理等等。整体而言，该系统具有以下功能特点：

- 面向 AI 开发：通过预制的 NVidia GPU 执行环境和容器镜像，UAI Inference 提供基于 Docker 的 Http 在线服务基础镜像，支持 TensorFlow、Keras、Caffe、MXNet 多种 AI 框架，能快速 AI 算法

的在线推理服务化。

- 海量计算资源：拥有十万核级别计算资源池，可以充分保障计算资源需求。且系统按照实际计算资源消耗收费，无需担心资源闲置浪费。
- 弹性伸缩、快速扩容：随着业务的高峰和低峰，系统自动调整计算资源配置比，对计算集群进行横向扩展和回缩。
- 服务高可用：计算节点集群化，提供全系统容灾保障，无需担心单点错误。
- 用户隔离：通过Docker容器技术，将多用户存储、网络、计算资源隔离，具有安全可靠的特性。
- 简单易用：支持可视化业务管理和监控，操作简单。

在线推理的可靠性设计

因为推理请求是随着访问量的变化而变化的，因此，在线推理的可靠性设计，考虑以下几点：1) 充足资源池，保证在高并发情况下，系统有足够的计算资源使请求访问正常；2) 负载均衡：将请求合理的分配到各节点当中；3) 请求调度算法：用于计算资源的实时调度；4) 性能监控：查看用户访问状态，为系统扩缩容做参考；5) 高可用部署：保证在单节点宕机时，系统能够正常运行。

负载均衡

UAI Inference 为每个在线服务提供了自动负载均衡能力，当用户提交同构、独立的 AI 在线推理容器镜像时，平台会根据请求的负载创建多个计算节点，并使用负载均衡技术将请求转发到计算集群中。

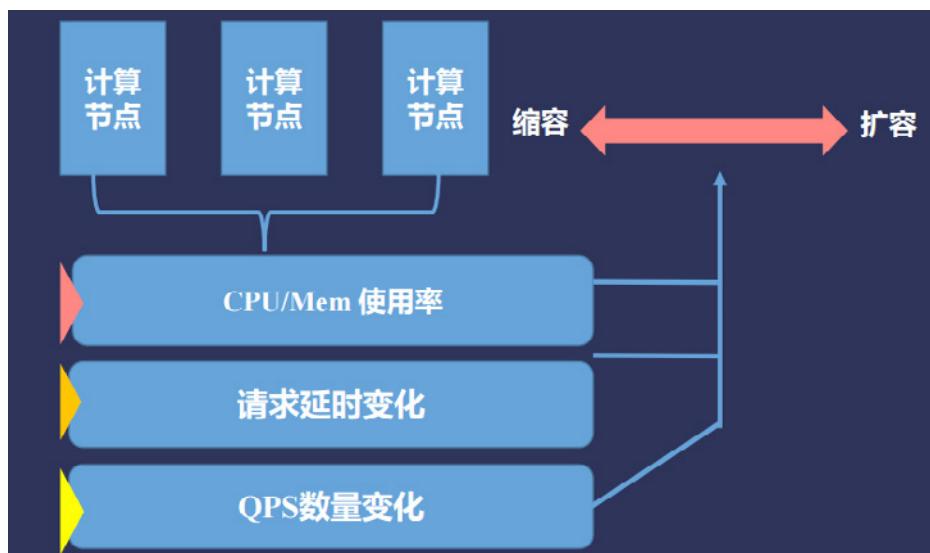
如图所示，负载均衡主要包括网络层和转发层。网络层中，同一个交换机（IP）可以接多个后端节点，通过请求调度算法将请求分配到各个计算节点当中。调度算法可以采用Hashing、RR、Shortest Expected Delay 等，其中，Hashing适用于长链接请求，Shortest Expected Delay适用于短

链接请求。目前，AI Inference采用RR的方式在计算节点间调度请求。整个系统最底层是一个统一的资源池，用以保证充足的计算资源。



动态扩缩容

在实现扩容之前，需要通过监控了解各节点当前的在线推理状态，这里，主要是通过实时收集节点的负载（CPU、内存）、请求的 QPS 和延时信息，来制定动态的扩容和缩容策略。

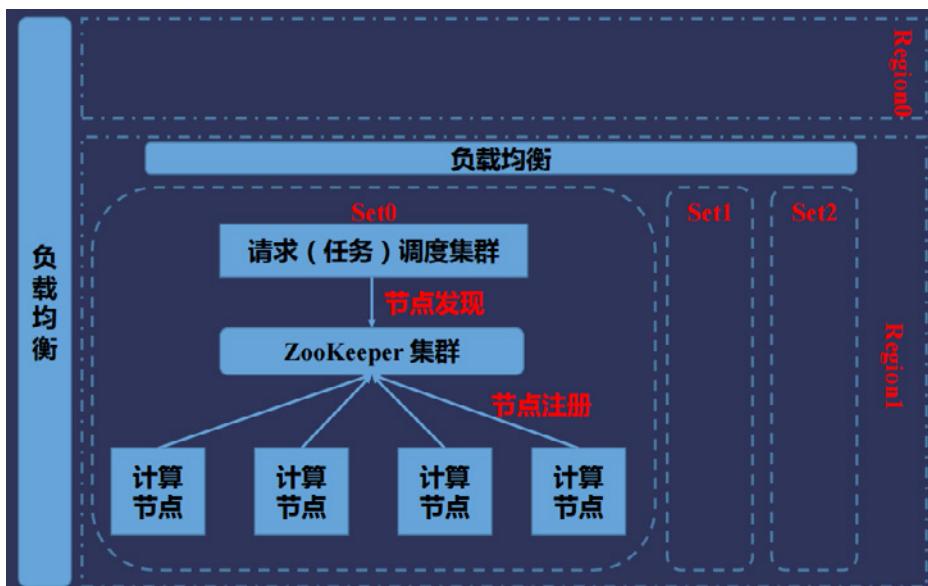


系统状态实时监控

此外，AI Inference 系统将 Http 请求、延时和 Http 返回码实时记录成日志，然后通过数据统计来在图形界面展示 Http 请求量、延时、成功率等信息。平台会实时收集所有计算节点的 stdout 数据，并录入日志系统，用户可以通过观察线上运行日志来了解线上运行状态，并根据监控信息自动选择扩容和缩容。

高可用

除了基本的扩缩容和负载均衡，我们也通过将计算节点集群化的方式，提供全系统容灾保障。如下图所示，系统会把整个服务切分成多个 set，部署在跨机房的某几个机架或者区域里面，当某一个机房或者 set 容机时，其他地区的在线推理处理还在进行。这种方式的好处是当出现单点故障时，其他区域的计算节点能够保证整个在线推理请求的正常执行，避免因单节点故障导致的系统不可用。



总结

本文通过对 UAI Inference 的实现原理、架构设计以及弹性扩缩容、

负载均衡、高可用等可靠策略的介绍，讲解了大规模、高并发在线推理请求时，UCloud 的部分解决策略和方案，希望能够抛砖引玉，为其他开发者做AI在线推理部署时带来新的思路。

截止目前，UAI Inference提供了CPU/GPU数万节点的在线推理服务。未来，我们会兼顾高性能在线服务和高性价比的在线服务两个方向，同时提供针对GPU硬件和CPU硬件的优化技术，进一步提升在线服务的效率。同时也会着力于公有云和私有云的结合，后期将会推出私有云的在线推理服务。

作者介绍

宋翔，UCloud高级研发工程师。负责UCloud AI产品的研发和运营工作，曾先后于系统领域顶级会议Eurosyst、Usinex ATC等发表论文，在系统体系架构方面具有丰富的经验。

智能时代的新运维

CNUTCon

全球运维技术大会

▶ 聚焦12大专题

- AIOps
- 自动化运维平台实践
- 监控与分析 (APM)
- 日志处理
- 性能优化
- 运维管理
- 数据库运维
- CI/CD
- 微服务
- Kubernetes
- 运维新技术
- SRE



会议：2018年11月16–17日

培训：2018年11月18–19日

地址：上海 光大会展中心大酒店

6折报名中，立减
团购享更多优惠，截止20



▶ 联席主席



刘国华
阿里巴巴 研究员



吴其敏
平安银行
零售网络金融事业部首席架构师



涂彦
腾讯 游戏运维总监



李涛
百度 工程效率部负责人
百度平台化委员会秘书长

▶ 专题出品人



尤勇
美团点评
高级技术专家



鲍永成
京东商城
技术总监



岳洪达
百度 智能云事业部
智能运维经理



王潇俊
携程旅行网
系统研发部高级总监



曹晓翔
阿里巴巴
资深技术专家



刘建
搜狗
资深架构师



李文韬
eBay Manager of Site
Reliability Engineering



王磊
华为技术有限公司
资深架构师



余珂
爱奇艺
高级总监



联系我们：

售票咨询(电话)：13269078023

售票咨询(邮箱)：piaowu@geekbang.org

AiCon

全球人工智能与机器学习技术大会

2018.12.20-21

北京·国际会议中心

6折 预售中，现在报名
立减 1440元

团购报名，可享受更多优惠



联系我们：热线 18514549229 微信（同手机号）

大会简介

AICon全球人工智能与机器学习技术大会是由极客邦科技旗下 InfoQ 中国主办的技术盛会，大会为期2天，主要面向各行业对AI技术感兴趣的中高端技术人员。大会聚焦AI最前沿技术、产业化和商业化的动态，将重点关注人工智能的落地实践，关注人工智能技术领域的行业变革与技术创新，与企业一起探寻AI的边界。

大会聚焦

机器学习

强化学习

NLP

自动驾驶

计算机视觉

智能硬件

AI与业务实践

搜索推荐以及Feed流

AI+区块链

往届嘉宾



颜水成
360人工智能研究院
院长及首席科学家



袁进辉
(老师木)
一流科技创始人



刘海锋
京东商城
总架构师&技术VP



山世光
中科院智能信息处理
重点实验室
常务副主任



张重阳
微信小程序
商业技术负责人



张浩
饿了么
技术副总裁



吴华
百度
技术委员会主席



胡时伟
第四范式
首席架构师



徐盈辉
菜鸟网络
人工智能部资深总监



架构师 月刊 2018年7月

本期主要内容 : Spark 团队开源新作 : 全流程机器学习平台 MLflow ; Google 发布 Flutter Release Preview 1 ; 独家揭秘 : 腾讯千亿级参数分布式机器学习系统无量背后的技术门道 ; 阿里巴巴为什么不用 ZooKeeper 做服务发现 ?



Kubernetes指南

《Kubernetes 指南》开源电子书旨在整理平时在开发和使用 Kubernetes 时的参考指南和实践心得，更是为了形成一个系统化的参考指南以方便查阅。



2018 , 进击的大前端

全栈与大前端，前端工程师进阶该如何抉择？



《英雄联盟》 在线服务运维之道

拳头公司基础设施团队的工程师们分享了他们运维在线服务的发展历程，介绍了他们从手动部署到自动运维的演变过程。