

# 架构师

ARCHITECT

7月刊

## 热点

Spark团队开源新作：全流程  
机器学习平台MLflow



# CONTENTS / 目录

## 热点 | Hot

Spark 团队开源新作：全流程机器学习平台 MLflow

Google 发布 Flutter Release Preview 1

## 理论派 | Theory

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

## 推荐文章 | Article

阿里巴巴为什么不用 ZooKeeper 做服务发现？

## 观点 | Opinion

Airbnb 弃用之后，我们还应该用 React Native 吗？

## 特别专栏 | Column

如何“计算” CEPH 读写性能

中心化 or 去中心化？聊聊交易所的辩证发展



## 架构师 2018 年 7 月刊

本期主编 蔡芳芳

提供反馈 feedback@cn.infoq.com

流程编辑 丁晓昀

商务合作 sales@cn.infoq.com

发行人 霍泰稳

内容合作 editors@cn.infoq.com



2018.8.18-19  
北京·国际会议中心

8折报名中  
立省 720!

# 区块链前沿技术与落地案例

更多内容及嘉宾，长按二维码立即揭晓



## 演讲议题

基于区块链服务构建企业区块链业务系统的实践分享与探讨

余珊 / 阿里云区块链技术负责人、阿里巴巴高级技术专家



中心化&去中心化区块链资产交易系统

巨建华 / BlueHelix 创始人/TGO 鲲鹏会会员



区块链云服务落地探讨

敖萌 / 腾讯金融云区块链首席架构师



区块链在金融风险信息共享项目中的应用实践

张大晴 / 京东金融区块链技术负责人



华为云区块链服务技术决策和落地实践

刘再耀 / 华为云区块链服务产品负责人



区块链节点攻击面及缓解措施

彭峙酿 / 360核心安全事业部安全研究员



如何打造基于区块链的金融事务及数据分析系统

谭峰 / 中证信用区块链服务技术总监



点融区块链云服务实践与思考

肖诗源 / 点融网技术总监



企业级区块链技术实践

黄海泉 / 京东区块链研发部高级架构师



RatingToken—基于大数据算法的评级实践

张文君 / 猎豹技术总监/区块链研究中心负责人



基因组数据区块链的机会和挑战

陈钢 / WeGene研发CTO/TGO 鲲鹏会会员



区块链与IOT在鸡只溯源中的协同应用实践

杨智健 / 连陌科技高级架构师



一拳定胜负：智能合约攻守道与漏洞浅析

于晓航 / 长亭科技区块链安全研究员



区块链技术在食品溯源领域的应用实践

张增骏 / 智链ChainNova技术总监/架构师



一个金融区块链技术团队2年的踩坑扫雷之路



2018年京东、微软、华为、猎豹、恒生电子等  
区块链专家联合出品



## 全球软件开发大会

### ▶ 聚焦

- 硅谷运维技术
- 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

**Red Pulse /** 基于NEO区块链的专家网络应用实践

**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：前端新趋势  
杨周璇 / 蚂蚁金服  
前端技术专家



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



俞育才  
eBay  
大数据架构师



Hien Luu  
LinkedIn  
工程经理



翟艺涛  
美团  
高级技术专家



滕昱  
Dell EMC  
工程总监



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

## 数据部门如何 All In AI

作者 丁香园资深数据架构 祝威廉

大数据部门的常见能力如下：

- 报表统计
- 算力/存储输出
- 推荐/搜索/精准营销等传统产品形态

通常，大数据部门会花费很大的力气构建数据平台，而这个数据平台除了能让研发、、算法、、分析师等角色爽一些，从宏观角度很大地节省部门人力成本、、提高效率以外，似乎对公司/其他业务部门并无直接输出。这也是很多大数据部门领导非常焦虑的地方。

那么出路在哪里呢？

### All In AI

事实上，真正能帮助业务提高效能、、提供创新产品的必然是AI。。AI是一种模式的输出，，其价值点，第一个是可以给业务每个环节赋能，比如反垃圾可以减少审核同学的工作量，智能邀请可以减轻运营同学的工作压力；第二个是创新产品，高一点的有比较常见有无人驾驶、、智

能语音产品、医疗诊断等，低一点的，则可能是某个具体的功能模块对外输出，比如知识图谱。

从上面我们可以看到，数据部门的最大价值，最终会通过AI来落地，并且还会给部门/公司提供极为丰富的想象空间。

## 如何 All In AI ?

对于这件事情，我们要仔细研究一个核心的东西：资源。

资源我们又分为：

- 平台资源，如果你还在刀耕火种阶段做开发、做算法，那么咱也别谈什么All in AI了。
- 人力资源，一场大型战争，核心还是在于看能动员的人力资源，面对海量需求，你是否有足够的人力去应付？
- 组织资源，合理的组织是能够极大地地释放生产力的。

经过这么多年的发展，平台已经很成熟了。我们知道，AI平台是基于数据平台之上的，其结构是一个金字塔形状的。所以第一步你需要有一个良好的数据平台，其次你还需要有一个AI平台，让单一算法落地变得容易。

人力资源的问题是个大问题，算法团队再大，也就是大数据部门一个子部门/组。如何在保持现有成本的情况下，扩大人力呢？AI平台对单一算法（后面我会解释什么是单一算法）问题是非常友好的，可能一个普通的工程师（甚至运营、分析师）都可以完成的。这样，部门所有的人都具备了成为AI人力的潜能。我们通过一定的培训和锻炼，可以使得研发、分析等都具备成为AI人才的潜力。需要的时候，我们提纯下即可。

回过头来看看，什么是单一算法。所谓单一算法就是具体的某个算法问题，比如对于帖子的情感分类，就是一个标准的文本分类问题。通常一个足够细化的问题，我们可以很容易将其转化为一个分类、回归、排序、规则类算法问题。现阶段，按我的了解，AI平台通常只能做到针

对单一算法的自助化。那么为了让组织更加合理高效，重构数据部门团队就很有必要了。

算法部门需要切分成三个子团队，一个是偏研究性质的，一个是偏业务性质的，还有一个则是AI平台和工具团队。

业务性质的团队常常需要用到研究性质团队的副产品以及基于AI平台和工具团队的产品之上进行工作，同时向他们反馈自己的诉求和问题。

业务算法团队通常也需要分成两个层级，一个是解决方案设计者，该角色是将一个实际的业务问题分拆成N个算法和工程问题；；一个是算法实施者，该角色只针对单一算法问题，可以在AI平台上很快地解决对应问题。

研究性质的团队可以分成三个部分，一个是读Paper，试图将学术论文转化为工业实践；一个是算法基础构建，维护比如知识图谱这种非常底层的系统；一个是创新产品，目标是利用现有的算法抽象出新产品。

通过如上方法，有了很好的平台能力，很好的人员基础，加上合理的组织，All In AI或许变得可能。

## 总结

本文我们说了为什么要All In AI，要实现All In AI 不仅仅需要有一个好的平台（数据、算法平台），也需要有良好的动员人力资源的能力，采用一个合理的算法组织架构充分利用人力资源。尤其是业务算法团队里的“解决方案设计者”，该角色能够将一个实际的业务问题分拆成N个算法和工程问题，是AI落地非常非常重要的一个角色。

# Spark 团队开源新作：全流程机器学习平台 MLflow

作者 Matei Zaharia 译者 无明



**AI前线导读：**在昨天开幕的 Spark+AI Summit 大会上，Spark 和 Mesos 的核心作者兼 Databrick 首席技术专家 Matei Zaharia 宣布推出开源机器学习平台 MLflow，这是一个能够覆盖机器学习全流程（从数据准备到模型训练到最终部署）的新平台，旨在为数据科学家构建、测试和部署机器学习模型的复杂过程做一些简化工作。Matei 表示，研究工作主要围绕着“如何为开发者提供类似谷歌 TFX、Facebook FB Learner Flow 等平台类似的好处，但是要以开放的方式——不仅在开源的意义上开放，而且是可以使用任何工具和算法的意义上开放”的想法展开。

Matei 为 MLflow 撰写了一篇介绍文章，AI 前线对文章进行了编译。这个全新的机器学习平台到底有何新特性？让我们来一探究竟。

每个做过机器学习开发的人都知道机器学习的复杂性，除了软件开发中常见的问题之外，机器学习还存在很多新的挑战。作为一家大数据解决方案公司，Databricks 与数百家使用机器学习的公司合作，所以能够清楚地了解他们的痛点，比如工具太过复杂、难以跟踪实验、难以重现结果、难以部署模型。由于这些挑战的存在，机器学习开发必须变得与传统软件开发一样强大、可预测和普及。为此，很多企业已经开始构建内部机器学习平台来管理机器学习生命周期。例如，Facebook、谷歌和优步分别构建了 FB Learner Flow、TFX 和 Michelangelo 来进行数据准备、模型训练和部署。但这些内部平台存在一定的局限性：典型的机器学习平台只支持一小部分内置算法或单个机器学习库，并且被绑定在公司内部的基础设施上。用户无法轻易地使用新的机器学习库，或与社区分享他们的工作成果。

Databricks 认为应该使用一种更好的方式来管理机器学习生命周期，于是他们推出了 MLflow，一个开源的机器学习平台。

## MLflow：开放式机器学习平台

MLflow 的灵感来源于现有的机器学习平台，但以开放性作为主要设计目标：

1. **开放接口：** MLflow 可与任何机器学习库、算法、部署工具或编程语言一起使用。它基于 REST API 和简单的数据格式（例如，可将模型视为 lambda 函数）而构建，可以使用各种工具，而不只是提供一小部分内置功能。用户可以很容易地将 MLflow 添加到现有的机器学习代码中，并在组织中共享代码，让其他人也能运行这些代码。
2. **开源：** MLflow 是一个开源项目，用户和机器学习库开发人员可以对其进行扩展。此外，利用 MLflow 的开放格式，可以轻松地跨组织共享工作流步骤和模型。

MLflow 目前仍处于 alpha 阶段，但它已经提供了一个可用的框架来处理机器学习代码。接下来将详细介绍 MLflow 和它的组件。

## MLflow 的组件 (alpha 版)

MLflow 的 alpha 版本包含了三个组件：



MLflow 的跟踪组件支持记录和查询实验数据，如评估度量指标和参数。MLflow 的项目组件提供了可重复运行的简单包装格式。最后，MLflow 的模型组件提供了用于管理和部署模型的工具。

### MLflow 的跟踪组件

MLflow 的跟踪组件提供了一组 API 和用户界面，用于在运行机器学习代码时记录参数、代码版本、度量指标和输出文件，以便在后续进行可视化。通过几行简单的代码就可以跟踪参数、度量指标和文件：

```

import mlflow

# Log parameters (key-value pairs)
mlflow.log_param("num_dimensions", 8)
mlflow.log_param("regularization", 0.1)

# Log a metric; metrics can be updated throughout the run
mlflow.log_metric("accuracy", 0.1)
...

```

```
mlflow.log_metric("accuracy", 0.45)
```

```
# Log artifacts (output files)
mlflow.log_artifact("roc.png")
mlflow.log_artifact("model.pkl")
```

用户可使用跟踪组件（通过独立脚本或 notebook）将结果记录到本地文件或服务器上，然后通过 Web UI 来查看和比较多次运行结果。团队也可以使用这些工具来比较不同用户的运行结果。

## MLflow 的项目组件

MLflow 的项目组件提供了一种用于打包可重用代码的标准格式。项目可以是一个包含代码的目录或 Git 仓库，并使用一个描述符文件来描述依赖关系以及如何运行代码。MLflow 项目通过一个叫作 MLproject 的 YAML 文件进行定义。

```
name: My Project
conda_env: conda.yaml
entry_points:
  main:
parameters:
  data_file: path
  regularization: {type: float, default: 0.1}
command: "python train.py -r {regularization} {data_file}"
validate:
parameters:
  data_file: path
command: "python validate.py {data_file}"
```

项目可以通过 Conda 来指定依赖关系。一个项目可能包含多个带有命名参数的运行入口。用户可以使用 mlflow run 命令行工具来运行项目，项目代码可以在本地，也可以在 Git 仓库里：

```
mlflow run example/project -P alpha=0.5
mlflow run git@github.com:databricks/mlflow-example.git -P
alpha=0.5
```

MLflow 将自动为项目设置合适的环境并运行它。另外，如果在项目中使用了 MLflow Tracking API，MLflow 将记住执行过的项目版本（即 Git 的提交操作）和参数，这样就可以很轻松地重新运行完全相同的代码。

无论是在企业还是在开源社区，项目格式让共享可重用代码变得更加容易。结合 MLflow 的跟踪组件，MLflow 项目为可重现性、可扩展性和实验提供了很好的工具。

## MLflow 的模型组件

MLflow 的模型组件提供了一种将机器学习模型打包成多种格式的规范，这些格式被称为“flavor”。MLflow 提供了多种工具来部署不同 flavor 的模型。每个 MLflow 模型被保存成一个目录，目录中包含了任意模型文件和一个 MLmodel 描述符文件，文件中列出了相应的 flavor。

```
time_created: 2018-02-21T13:21:34.12
flavors:
  sklearn:
    sklearn_version: 0.19.1
    pickled_model: model.pkl
  python_function:
    loader_module: mlflow.sklearn
    pickled_model: model.pkl
```

在这个例子中，模型可以与支持 sklearn 或 python\_function 的工具一起使用。

MLflow 提供了将常见模型部署到不同平台的工具。例如，任何支持 python\_function 的模型都可以部署到基于 Docker 的 REST 服务器或云平台上（如 Azure ML 和 AWS SageMaker），也可以作为 Apache Spark 的用户定义函数，用于进行批量和流式推断。如果使用 Tracking API 将 MLflow 模型输出为文件，MLflow 还会自动记住它们是由哪个项目运行生成的。

## MLflow 入门

要使用 MLflow，请按照 [mlflow.org](http://mlflow.org) 上的说明进行操作，或使用 Github 上的 alpha 版代码。

在 Databricks 上托管 MLflow

如果要使用 MLflow 的托管版本，可以在 [databricks.com/mlflow](https://databricks.com/mlflow) 注册。Databricks 上的 MLflow 与 Databricks Unified Analytics 平台集成，包括 Notebooks、Jobs，Databricks Delta 和 Databricks 安全模型，为用户提供了安全、生产就绪的方式大规模地运行现有的 MLflow 作业。

## 后续计划

MLflow 刚刚展露头角，因此还有很多事情要做。后续计划引入新组件（如监控）、与其他库的集成以及扩展已发布的组件（例如支持更多环境类型）。

MLflow 官网 <http://www.mlflow.org>

MLflow GitHub 地址 <https://github.com/databricks/mlflow>

# Google 发布 Flutter Release Preview 1

作者 覃云

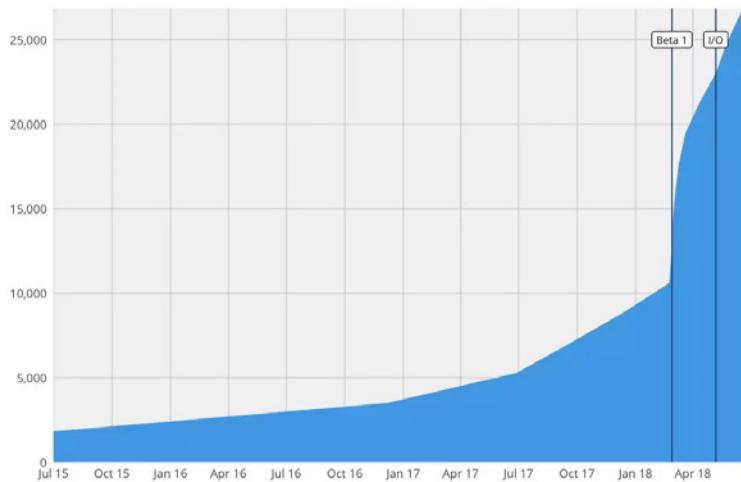


在北京 GMTC 大前端大会上，Google Flutter 高级工程师于瀟宣布 Flutter Release Preview 1发布，并宣布与阿里巴巴闲鱼团队在 Flutter 上合作，这标志着 Flutter 进入了一个新阶段。

## Flutter 一直在进步

在 Google I/O 大会上，Flutter 团队与很多 Flutter 的开发者进行了沟通，并对 Flutter 进行了改进。在那之后，Flutter 生态得到了快速地增长，Flutter 的活跃用户增长了 50%，不仅如此，在 I/O 大会之后的几周内，在全世界范围内，已有超过 150 个与 Flutter 相关的事件发生。

GitHub stars by date



Source: <http://timqian.com/star-history/>



Flutter 团队表示，Flutter 从 beta 版本到现在的 Release Preview 1，都体现了他们对稳定性和质量的信心和关注。

Flutter 预览版从社区中得到了很多支持，来自外部的贡献就包括 Flutter 对硬件键盘和条形码扫描仪、视频录制、图像的支持。此外，还有许多新软件包对 Flutter 包站点的贡献，例如 Flutter Platform Widgets，一组可自适应 iOS 或 Android 的小部件； mlkit，Firebase MLKit API 的包装类（wrapper）； 序列动画（Sequence Animation）。

在这个过程中，Flutter 团队将重点转向了场景的完整性（scenario completeness）。他们不断改进视频播放器套件，提供更广泛的格式支持和提升可靠性。并进一步扩展了对 Firebase 的支持，如 Firebase Dynamic

Links，这是一款用于创建和处理跨多个平台的链接的应用解决方案。目前已经扩大到对 ARMv7 芯片、32 位 iOS 设备的支持，这让使用 Flutter 编写的应用仍可以在全球流行的旧设备（如 iPad Mini 和 iPhone 5c）上运行。

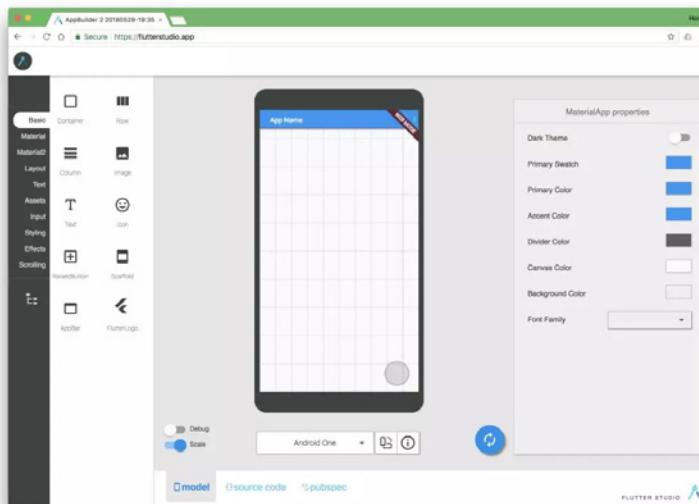
## 关于升级

你如何升级到 Flutter Release Preview 1？如果你正在使用 beta 版，只需要一个命令：

```
$ flutter upgrade
```

## Flutter 预览版工具

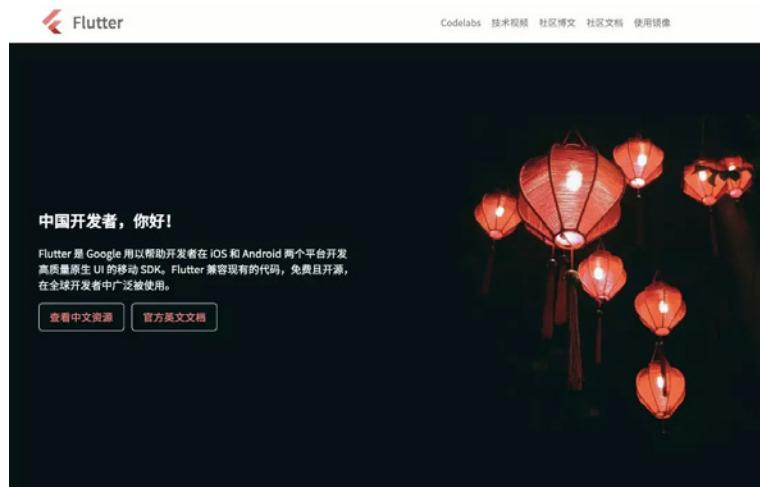
Flutter 工具也获得了重大改进，Visual Studio Code 的 Flutter 扩展添加了新的大纲视图、语句结束（statement completion）以及直接从 Visual Studio Code 启动模拟器的功能。



## Flutter 在中国

使用 Flutter 的中国开发者人数仅次于美国和印度，在中国，已经有许多大型公司采用 Flutter，阿里巴巴已经将他们基于 Flutter 的应用程序部署到数百万的设备上，而腾讯正在推出基于 Flutter 的 NOW 应用程序。目

前，中国版的 Flutter 网站已经上线，你可以访问啦！



链接：<https://flutter-io.cn/>

## 开发者评价

很多开发者对 Flutter 的评价是认为它很有趣，以下来自一位最近刚将 iOS 应用程序移植到 Flutter 的 iOS 开发人员：

“Ruby on Rails 和 Go 诞生之后，我一直对技术感到兴奋……在花了数年时间深入学习 iOS 应用开发之后，它让我失望，因为它让我疏远了那么多 Android 用户。而且，在那个时候，学习其他跨平台框架对我来说没有任何吸引力。对我而言，Flutter 应用程序一直以来只是一个试金石，但它通过了我的测试。我认为 Flutter 值得投资，因为我很享受使用它的过程。

# 独家揭秘：腾讯千亿级参数分布式 ML 系统 无量背后的秘密

作者 袁 磊



AI前线导读：千亿参数规模的模型已经被业界证明能够有效提高业务效果。如何高效训练出这样的模型？百 GB 级别的模型如何在线上实现毫秒级的响应？这些能力在各个大厂都被视为核心技术竞争力和机器学习能力的技术壁垒。要具备这样的能力，对相关系统有什么样的挑战？本文将从系统的角度去详细分析这些问题，并给出腾讯公司的无量系统对这些问题的解答。

## 简介

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。下面以网页 / 图文 / 视频推荐场景分析这些挑战，下文中均称为推荐场景。

1. 样本数量大。在推荐场景下，每天的样本量可以达到百亿量级。如果需要按一个月的样本进行训练，样本量会在千亿级别。如果每个样本平均 500 特征值，单个样本的大小就是 5KB 左右，一千亿样本的大小就是 500TB。即便只使用一周内的样本，样本数据的大小也在 100TB 这个级别。
2. 特征维度多。巨大的样本量使高维度模型的训练成为可能。为了给用户提供更合理的推荐结果，需要对用户和被推荐的文章 / 图片 / 视频进行详细的描述。各个业务都会建立起丰富的用户模型，也会对文章 / 图片 / 视频进行多维度的标注。
3. 在系统进行推荐时，还会使用到用户现在的上下文信息，比如：时间，位置，之前浏览的页面等。当这些特征被引入到模型中时，会导致特征规模的迅速增加。如果再考虑交叉等特征转换操作，模型的特征维度会轻松地膨胀到千亿甚至万亿级别。

训练性能要求高。我们面对的是百 TB 的样本和百亿 / 千亿参数的模型。而业务需要在短时间内训练出一个性能指标好的模型，以便快速上线验证。这对机器学习平台训练能力有很高的要求。

前面 1~3 点，提出的是超大规模模型的训练框架面临的挑战，然而训练出模型只是重要的第一步。最终模型需要上线为用户提供服务才能体现其业务价值。对于以往的机器学习中的中小模型，模型上线服务并不是一个特别被关注的事情。但是，当最终模型文件很大，甚至超过单机的内存大小时，模型上线服务就变成了棘手的问题。

1) 模型大但用户需要毫秒级响应。以最简单的 LR 模型为例，一个 10 亿特征的模型的大小也会达到 12GB（每个参数需要一个 8Byte 的 key 和 4Byte 的 float value）。如果是 DNN 模型，模型大小到达 TB 也是可能的。当训练好一个模型后，模型就被上线，为用户提供预测服务。

为了达到良好的用户体验，预测服务的响应时间需要在 10ms 这个量

级。以手机用户的推荐场景为例，从用户在手机上刷新页面到看到推荐结果，时间不能超过 1s，扣除掉网络通讯的开销，IDC 内在线服务的响应时间需要控制在 200ms 以内。但是，整个推荐的流程至少有召回，排序和展示控制三个阶段。

在排序阶段，需要对 200 个以上的文章进行特征拼接和点击率预估，所以模型对这 200 个文章进行点击率预估的总时间要在 30ms 以内。如何使用这么大规模的模型进行高性能，高并发的预测也对平台能力的重大考验。

2) 模型实时上线。对于资讯推荐类场景，用户的关注点变化很快。系统需要根据最新的用户行为数据调整模型，然后以最快的速度将如此大规模的模型更新到多个地区的在线预测服务。

为了解决以上挑战，我们：

1. 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。
2. 在无量框架的基础上，我们构建起自动化模型管理系统 -- 无量模型管理，模型能够在离线训练任务，在线训练集群和在线预测服务之间无缝地高效流转，实现 10GB 级模型的分钟级上线。
3. 为了提高大模型的线上预测服务，我们还开发了高性能的预测模块和服务 -- 无量预测服务，对于数十 GB 的模型，只需几毫秒即可完成 100 个样本的预测。

无量框架，无量模型管理和无量预测服务共同构成了无量系统的主要部分。下面我们将对无量系统的架构和各个主要组成部分进行详细的介绍。

## 1. 系统流程与架构

工作流程 在广告 / 推荐等场景下，模型的生产和使用过程，大致分为

几个步骤：

1. 日志采集与样本生成。通过收集用户的线上行为信息，生成模型需要的样本。这些样本可以被存储起来用于离线训练，也可以使用流式数据的方式推送给在线训练集群。
2. 模型训练。有了样本之后，在训练集群中训练出具体的模型。开发人员通过调整的超参数或模型结构来获取好的模型。
3. 模型评估。在模型被放到线上服务之前，需要对模型进行一些评估工作。
4. 模型上线预测。无量系统目前包括以上步骤中的模型训练，模型评估和上线预测。



为了让模型从训练集群到在线预测服务顺利地流转，无量系统提供了模型管理功能，能够自动化地将从训练机器导出新模型到在线预测服务。业务也能够在模型自动化上线过程中定义模型评估操作，避免训练效果差的模型被放到在线预测服务中。另外当模型上线之后，也需要验证线上的模型是否有问题。

在模型的开发过程中，超参数调试耗费了模型开发人员大量的时间。无量系统通过与般若系统结合，实现了模型训练效果的实时监控，为自动化调参提供了决策数据。无量系统正在进行自动调参工具的开发。算法人员也可以基于这些数据上实现自定义自动调参功能。

## 系统架构

在一个机器学习系统中，机器学习算法的代码只是训练或预测过程中

非常小的一部分。以下是 Google 对其机器学习系统的一个统计。

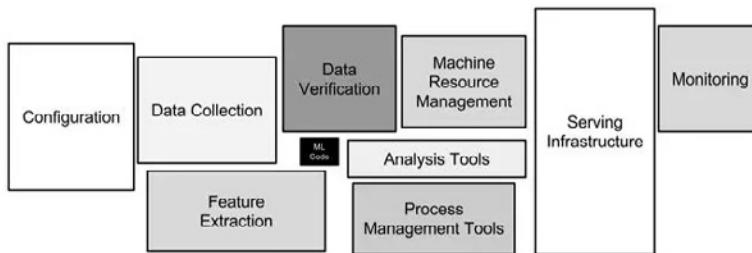
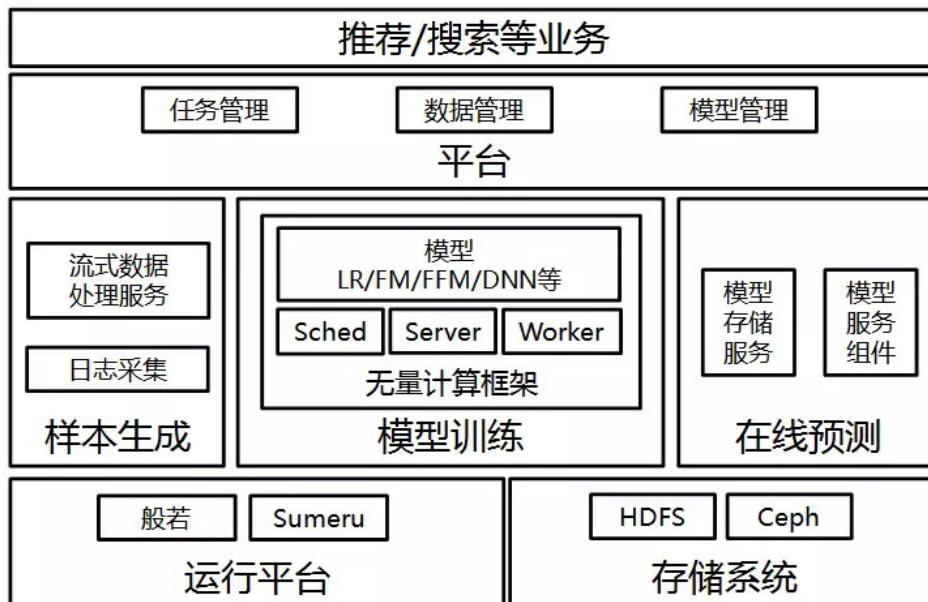


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

### [1] Hidden Technical Debt in Machine Learning Systems, Google.

为了让机器学习任务运行起来，需要大量的配套服务设施的支持，包括数据管理与存储，训练框架，在线预测框架和资源管理等多个模块。无量系统的系统架构如下所示：



### 系统架构图

无量系统的训练任务运行在 MIG 专门针对机器学习和深度学习任务的般若调度系统上，在线训练集群和在线预测服务部署在 Sumeru 系统上。般若系统和 Sumeru 系统均是基于 docker 容器化技术构建，为无量系

统的快速部署和扩展提供了可靠的基础设施保障。

下层存储系统支持 HDFS 和 ceph 两种分布式网络存储。HDFS 作为常用的分布式网络存储，与其他的数据分析系统无缝对接。Ceph 以其高性能与灵活的文件操作，弥补了 Hdfs 在文件操作上的不便。

日志采集使用 MIG 的灯塔系统，并配合了自研的流式数据处理服务实时生成训练样本。

通过自研的基于参数服务器架构的无量计算框架，无量系统支持了千亿级模型的任务型离线训练和流式在线训练。无量计算框架采用 C++ 实现以达到优异的性能，并支持了推荐和搜索场景常用的 LR/FM/FFM/DNN 等模型，用户只需做简单的配置即可实现超大规模模型的训练。

对于千亿级参数的模型，模型大小至少会有几十 GB。无量系统为业务的在线预测服务提供了两种模型使用模式：

- 模型服务组件。模型服务组件包含了模型版本管理和模型预测两个主要功能。由于模型服务组件对内存管理进行深度优化，业务能够在自己的预测服务中直接加载和使用 100G 以下的模型。
- 模型存储服务。当模型大小超过单机能够存放的大小时，就需要分布式的模型存储服务来进行模型管理和提供预测服务。

在样本生成，模型训练，在线预测模块之上，是无量系统的平台服务。用户在这里管理自己的数据，训练任务和模型。

至此，我们简单介绍了无量系统的各个部分。让读者能够对无量系统有一个整体的了解。下面，我们将重点介绍无量计算框架，模型管理与模型预测服务。

## 2. 无量计算框架

为了得到一个好的模型，至少需要三个方面的内容：1. 数据；2. 模型和算法；3. 计算框架。

正如前面介绍中所述，互联网用户为我们产生了大量的样本数据，为学习出一个好的模型提供了数据基础。在本节中，我们将重点介绍后面两

部分内容。

首先我们会介绍推荐场景常用的模型和算法，并由此推导出为了实现这些模型的训练，对计算框架有什么样的需求，以及无量计算框架如何满足这些需求，实现高性能的模型训练。

## 推荐模型与算法

随着商业化推荐的兴起，预测用户点击率（Click Through Rate，简称 CTR）领域得到了大量的研究关注，产生了很多 CTR 预估模型。下面对大规模场景下的几个代表性的模型做简单的对比介绍。他们分别是 LR，FM，DNN。对于推荐场景中常用的 GBDT 算法，由于其不适应大规模特征的输入，在此不做对比。

### LR 模型

LR 是一个简单而有用的线性模型。优点：它实现简单而且非常容易支持大规模特征的样本输入。在实际应用中，往往能取得不错的效果，常常被用作 baseline。缺点：由于是线性模型，需要大量的特征工程的工作来让它得到好的效果。而特征交叉等操作也直接导致了模型的特征空间急剧膨胀。

$$y_{LR}(x) = \text{sigmoid}\left(w_0 + \sum_{i=1}^N w_i x_i\right)$$

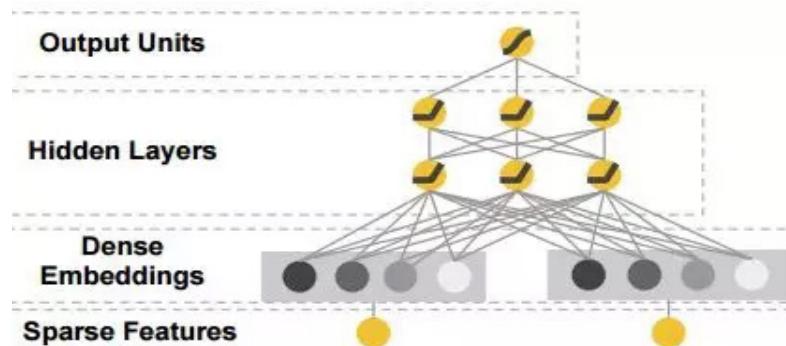
### FM 模型

FM 在 LR 线性模型的基础上，引入了二次项，使得 FM 模型能够自动学习特征之间的二阶交叉关系。优点：自动学习二阶交叉关系，减少了部分特征工程的工作。缺点：要学习二阶以上的交叉关系，仍然需要进行交叉特征选择的工作来生成相应的样本。

$$y_{FM}(\mathbf{x}) := \text{sigmoid}\left(w_0 + \sum_{i=1}^N w_i x_i + \sum_{i=1}^N \sum_{j=i+1}^N \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j\right)$$

## DNN 模型

随着深度神经网络（DNN）在图像、语音等领域的突破性发展，DNN 被引入到 CTR 模型中来，希望学习到特征之间的复杂关系，得到更好的模型。在 CTR 预估中，输入特征是高维稀疏的，不能直接使用全连接网络直接进行学习，所以用于 CTR 预估的网络一般采用 embedding 层 + 全连接层的结构。通过 embedding 层将稀疏特征转换为低维稠密特征，再输入后面的全连接层。优点：可以直接输入原始特征，减少了交叉特征的选择工作。缺点：训练调参相比 LR 和 FM 等更难。由于 DNN 稠密参数的引入，训练性能也比 LR 和 FM 更低。



### [Google 2016] Wide & Deep Learning for Recommender Systems

前面简单介绍了三种代表性的模型，在这三种基本结构上，通过不同的组合和改进，衍生出了 FFM，FNN，DeepFM，DIN 等模型结构。如果想详细了解相关的模型，请见参考文献 [3][4]。

从上面的模型基本结构，我们可以总结出 CTR 模型的参数特点：

- 超大规模稀疏的输入特征参数。LR，FM 和 DNN 的 embedding 层的输入都是稀疏的，参数值可能是一个单独的值（LR 的  $w$ ），也有可能是一个向量（FM 中的  $w+v$  和 embedding 层的  $w$ ）。
- 稠密的交叉关系参数。DNN 中全连接层参数都是稠密的。

由此可以看出，计算框架需要同时支持稀疏和稠密两种参数格式。另外，一些统计类特征（例如：文章的曝光数，点击率等）在训练中也是很

重要的。这些参数也需要在训练过程中方便地计算得到。

在推荐场景下，可推荐的内容存在一定的时效性，随着热点的变化，用户的关注点也会发生相应的变化，导致 CTR 模型应用到线上后，预测性能会随着时间的流逝而下降，所以 CTR 模型都需要进行及时的更新。在不同的业务应用场景下，这个更新频率可以是分钟级，也可能是天级别的。

然而，重新训练一个百亿规模的模型会消耗大量的时间和计算资源，为了以低廉的资源成本完成模型的及时更新，推荐场景下会采用在线训练的方式。所以计算框架需要支持多种在线训练算法。目前应用于在线训练的优化算法主要有 Ftrl, Adagrad 等。

高性能大规模并行机器学习框架 在我们的系统设计目标中有三个关键维度：

- 千亿级模型参数；
- 千亿级样本数据；
- 高性能。

如何同时提高上面的三个维度的目标，我们需要仔细分析分布式计算过程。以现在常用的基于梯度下降的分布式优化算法为例。在使用样本数据 I 进行一轮训练的过程中，有以下几个基本步骤：

---

**Algorithm 1** Distributed gradient-based optimization
 

---

```

1: Initialize  $w_0$  at every machine
2: for  $t = 0, \dots$  do
  3:   Partition  $I_t = \bigcup_{k=1}^m I_{t_k}$ 
  4:   for  $k = 1, \dots, m$  do in parallel
    5:     Compute  $g_t^{(k)} \leftarrow \sum_{i \in I_{t_k}} \partial f_i(w_t)$  on machine  $k$ 
    6:   end for
  7:   Aggregate  $g_t \leftarrow \sum_{k=1}^m g_t^{(k)}$  on machine 0
  8:   Update  $w_{t+1} \leftarrow w_t - H_t^{-1} g_t$  on machine 0
  9:   Broadcast  $w_{t+1}$  from machine 0 to all machine
10: end for
```

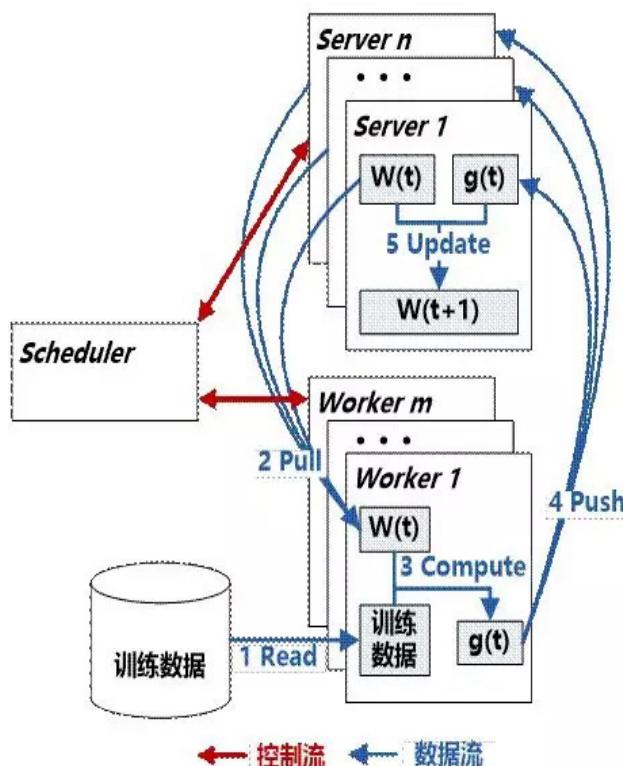
---

- 数据分片，将所有数据拆分后分配到多台机器上；
- 并行计算 g，各台机器上的计算节点按照指定算法计算梯度；
- 聚合 g，将各台机器上计算的 g 收集起来；

- 更新  $w$ , 使用上一步得到的  $g$  更新  $w$ ;
- 广播  $w$ , 将更新后的  $w$  传输给计算机器。

这样的学习逻辑通过将数据分布到多台机器上计算，有效地解决了样本数据量的问题。Hadoop 和 Spark 都采用这样的逻辑进行机器学习，Spark 由于 RDD 的方式充分利用内存来存储中间数据，大大提高了训练性能。但是在这样的训练逻辑下，存在两个问题：

1.  $w$  被存储在一台机器上，限制了框架能够训练的模型的规模，只能是单机可存储的模型，以 128G 的内存的机型为例，10 亿个参数的模型就达到他的存储极限了；
2.  $w$  被广播给各个机器。由于是广播推送方式，当模型规模变大的时候，广播操作带来的带宽成本会急剧增加。以我们的测试来说，用 Spark 训练一个百万参数的模型时就发现性能难以忍受了。



参数服务器的基本结构和工作流程图

以上分布式训练逻辑是梯度下降算法的逻辑，而现在机器学习尤其是深度学习中广泛使用的是随机梯度下降算法（SGD）。模型参数是以 minibatch（128 个样本，甚至更少）为单位来更新的。这导致参数更新频率急剧提升，带来的是巨大的网络带宽需求。所以必须要解决上面两个问题，才能够进行千亿级参数的模型训练。参数服务器架构由此产生。

从 2010 年被提出，经过了几年的发展演进，现在普遍使用的是第三代参数服务器架构。相对于前面 Algorithm 1 的流程，参数服务器有两点主要的不同：

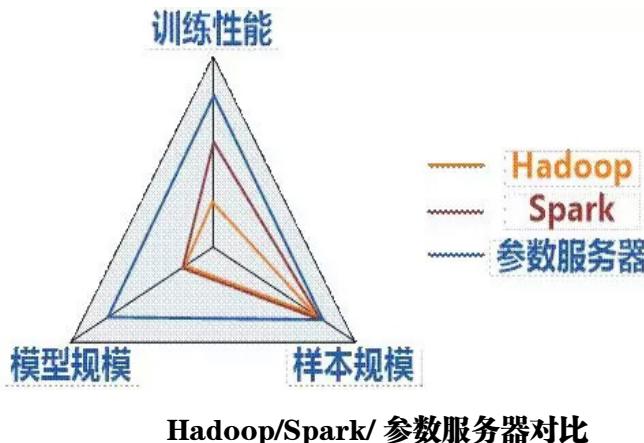
1. 有一种新的角色 Server，专门用于分布式地存储模型参数，并进行参数的更新计算。这使得能够训练的模型规模不再受限于单机的内存大小，同时将多个 worker 节点的参数请求分摊到多个 server 上，减少了单个 server 上因参数和梯度传输导致的网络瓶颈。
2. 负责计算的 Worker 节获取参数的方式是 pull 方式。由于不是被动的等待广播参数，pull 方式使得 worker 节点可以根据训练数据的需求来获取参数。尤其是在推荐场景下，样本都是非常稀疏的。

举例来说，一个模型可能有 100 亿个特征输入，而具体到一个特定的样本，只会有几百个有效特征值。所以只需要获取与这几百个有效特征值有关的参数即可，而不需要传输整个模型。

简而言之，参数服务器架构下，多个 server 分摊参数存储和传输的压力，多个 worker 按需获取和更新参数降低了参数和梯度传输的网络需求。这使得千亿参数模型的高性能训练成为了可能。

通过上面的分析，我们得到了以下的结论。参数服务器能够在模型规模，样本数量和训练性能三方面满足我们的设计要求。

了解了通用的参数服务器架构以及其特点，我们回到无量计算框架，继续分析一个通用的参数服务器架构在实际中面临的问题以及我们的解决。在模型和算法的分析中，我们知道，要实现两类稀疏和稠密两类参数的传输与更新。



### Hadoop/Spark/ 参数服务器对比

1) 超大规模稀疏的输入特征参数。这里稀疏有两层含义。

首先，模型可能的参数是千亿个，但是因为并不是所有特征都有可能出现在训练样本中，所以一般不会所有参数都有值，一般最终的模型可能只有 1/10 的参数是有值的。如果使用了稀疏化的技术，这个比例会更低。

其次，对于每个样本只会使用到非常少的特征。在一个千亿特征的模型中，单个样本通常只会命中到几百个特征。

从上面的分析中，可以看出，参数服务器架构在大规模稀疏特征的模型训练中尤为高效。因为 worker 训练一个 minibatch 的样本时，只需要获取与这些样本相关的参数即可，如果每个样本平均有 500 个特征，那么 100 个样本最多只需要获取 5 万个特征的相关参数即可。

2) 稠密的交叉关系参数。与稀疏的输入特征参数不同，交叉关系参数规模相对较小，但是每个样本的训练会使用到全部的稠密参数。假设全连接层中最大的一层是  $1024 \times 512$ ，那么每次计算使用到的稠密参数就是在 50 万这个量级。

从这里我们可以看出，稀疏和稠密两种参数在训练过程中存在不同的性质。稀疏参数总体规模大，但是每次训练只使用到很小的一部分。稠密参数总体规模相对较小，但是每次训练都需要被全部使用到。由于两种类型的参数的性质差异，被自然地切分成了基于 Kv 和基于矩阵的数据结构。

下面我们继续分析训练各个阶段的性能问题与我们的解法。

1) 参数获取。在实际的超大规模模型的训练中，网络经常成为性能瓶颈。为了减少因为参数获取而导致的网络传输压力，我们引入了参数缓存机制，worker 并不是每个 minibatch 都从 server 获取最新的参数。然而，在分布式训练中，缓存模型参数存在训练正确性的风险。

由于在数据并行情况下，各个计算节点使用的训练数据是不同的，如果进行多次训练而不同步更新参数，则模型可能出现无法收敛的问题。在学术研究领域，这是一个训练的网络带宽与模型训练正确性保障的问题。已有不同的同步控制协议的研究。

我们的实现借鉴了 ssp 协议<sup>[5]</sup> 中有限版本差异的思想，通过控制缓存的使用次数，在保障训练正确性的前提下，减少因参数获取而导致网络传输。

2) 梯度更新。计算完成后的梯度上传也会有大量的数据需要通过网络传输。按照模型的梯度计算逻辑，所有使用到的参数都会得到相应的梯度。但是，是否需要发送某个参数的更新，或者以什么样的方式发送给 Server 却是可以选择的，这个过程称为梯度压缩。梯度压缩的方法大致可以分两类：

- 梯度量化。将梯度从double/float等原始类型量化成二值/三值等用几个bit就能表示的类型，以减少传输数据量。
- 梯度稀疏化。选择重要的梯度立即上传，不是很重要的梯度更新，则累积起来，稍后再上传。如读者对这个研究领域感兴趣，可以阅读参考文献<sup>[6][7]</sup>。

传统的梯度压缩技术存在与模型大小相当的内存消耗，所以主要使用在单机可存储的稠密模型的训练中，在无量所应对的超大规模模型的训练中，我们对现有的梯度压缩技术进行了改进，使之适应了百亿稀疏参数规模模型的训练，可以减少99%以上的梯度传输而不影响训练效果。

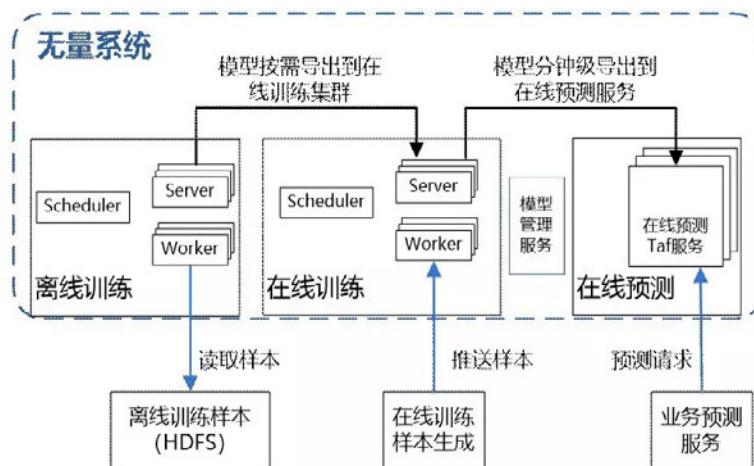
3) 梯度计算。在机器学习，尤其是深度学习过程中，模型的梯度计算过程会有大量的数值计算操作。除了使用多线程并行训练的方式充分利

用多个 cpu 的计算能力，我们还使用 SSE 等 CPU 并行计算指令和 Eigen 线性计算库实现梯度计算过程，充分利用了 CPU 芯片的流水线和并行指令能力，比单纯的多线程并行的计算性能高 10+ 倍。

在实际的生产环境中，数据被存放在 hdfs 集群上，而训练时拉取数据变得很耗时。我们通过将数据读取异步化，使得数据读取不影响训练的参数传输，梯度计算和更新过程。同时通过优化数据读取模块的内存管理和样本缓存机制，以极小的内存开销满足训练对样本随机性的需求。

### 3. 无量模型管理 -- 全流程模型管理

在推荐类业务中，文章和视频资料快速更新，社会热点随时出现和消失，用户的兴趣也经常变化。为了取得优秀的推荐效果，很多具有时效性的特征信息被加入到预测模型中，导致 CTR 模型需要及时更新。无量系统提供了全流程的模型管理服务。



#### 模型流转的基本流程

在管理超大规模的模型时，存在两个主要的挑战：

1) 模型超大导致的模型上线性能的问题。对于千亿参数的模型，如果每个参数都以 4 字节的 float 格式存储，最终存储的模型将会接近 TB 级别。如果要实现分钟级别地将新模型更新到多地的在线预测服务上，仅从数据传输和文件解析的性能上看，每次都使用全量模型的方式就是不可行的。

幸运的是，模型在训练过程中的变化是渐进的，而当模型上线时，是一个相对稳定的状态，在线训练更多的是对模型的微调。因此，对于超大规模的模型，一般采用全量 + 增量的方式进行管理。首先使用全量模型加载到线上服务，然后定期将模型的差异部分以增量的方式叠加到线上服务的模型中。

2) 模型分片导致的管理问题。在全量 + 增量的模型上线模式下，线上服务的模型对应着多个模型文件。如果线上服务出现故障需要恢复或者因为请求量上升需要扩容时，就需要使用多个模型文件来恢复出模型。在某些情况下，业务发现当前模型效果差，需要替换模型或者进行版本回滚时，需要另外的一组模型文件。

另外，不同于单机可存储的模型，在参数服务器框架下，模型被分片存储在不同的机器上。为了提高模型导出效率，多个 server 节点会并行导出多个模型分片文件。假设存在 100 个 server，那么就会有 100 个模型分片文件。给模型管理工作带来了挑战。

为了避免模型开发和使用者陷入这些管理问题，同时也为了保障系统的稳定运行，无量模型管理服务将所有模型管理的相关工作承接下来。用户只需进行必要的配置，模型管理服务就会自动地发现新版本的模型，验证模型的完整性并将新模型传输和发布到指定的在线预测服务中。

对用户完全屏蔽下层类似全量，增量，分片等细节。后期，用户还可以自定义模型验证的方法，对即将上线的模型进行模拟请求等校验，避免有效果差的模型被上线，给业务造成损失。

## 4. 无量模型服务

使用千亿参数的大模型进行线上预测，面临有许多的问题，下面我们就一些主要问题进行分析并介绍我们的方案：

1) 模型加载的内存问题。当被加载到内存中时，需要构建相关的数据结构，所消耗的内存大小会比模型文件大很多。以最简单的 LR 模型为例，每个特征只会有一个 float 类型的模型参数，一个 10 亿有值特征的模

型的文件大小大概是 12GB（每个特征 8 字节 key+4 字节值 value）。使用 stl 标准库中 unordered\_map 加载这个模型需要超过 25GB 的内存。也就是说会有超过模型大小 1 倍的内存开销，这使得单机能够存储的模型大小受到极大的制约。

我们自己实现了一个 hashmap：tlhashmap，专门针对模型稀疏参数特点进行了内存优化。内存消耗只比模型数据大 20% 左右。这意味着 tlhashmap 有效地提高了能够被单机存储的模型的大小极限。以 128GB 内存的机器为例，使用 tlhashmap，最大能支持的 lr 模型文件大小是 100GB 左右，而标准 unorderedmap 最大能支持 50GB 左右。

2) 模型服务的性能问题。为了达到良好的用户体验，预测服务的响应时间需要在 10ms 这个量级。以手机用户的推荐场景为例，从用户在手机上刷新页面到看到推荐结果，时间不能超过 1s，扣除掉网络通讯的开销，IDC 在线服务的响应时间需要控制在 200ms 以内，而整个推荐的流程至少有召回，排序和展示控制三个阶段。在排序阶段，需要对 200 个以上的文章进行特征拼接和点击率预估，所以模型对这 200 个文章进行点击率预估的总时间要在 30ms 以内。

从排序服务发出请求开始，到请求完成，至少存在两个性能瓶颈点：

1. 请求包的网络传输与编解码。为了预测文章的可能点击率，需要为每个文章准备所有的样本特征。假定每个样本有 500 个特征，那么 200 个文章的请求就有 10 万个特征。整个请求包的数据会有 1MB 左右。网络传输和编解码的性能对整个 rpc 框架都带来了极大的挑战。我们定义了一套针对模型预测场景的特征编解码格式，避开了现有 rpc 框架在编解码格式上的性能缺点，并且最大化地减少了需要传输的数据大小。
2. 模型参数查询和计算性能。为完成模型的预测功能，首先需要从模型中找到需要的参数，然后完成预测值的计算。面对超大规模的模型，首先要解决的就是模型存储方式的问题。如果模型能够单机存储，那么模型参数的查询则可以在本机完成。如果模型超

过单机存储的极限，则需要使用分布式存储的方式提供查询服务。

考虑上面的例子，一个请求需要 10 万个特征的参数，这些特征被存储在多台机器上。即使忽略预测计算时间，要保证这个请求在 30ms 之内返回，那么所有存储参数的节点都必须在 30ms 之内返回结果。这就会出现木桶现象，任何一个存储节点出现了超过 30ms 的响应延时，总体请求时间都一定会超过 30ms。这样的存储系统对请求排队是接近 0 容忍的。但推荐场景又是一个高并发的场景，预测服务需要支持每秒上万的用户请求。

无量系统开发了一套分布式模型预测服务，专门针对分布式预测场景下高并发的模型参数请求的性能问题进行优化，实现对 TB 级模型的高并发预测服务支持。

## 5. 总结

随着互联网服务的发展，越来越精细和定制化的服务需要更好的模型支持，而超大规模预测模型已经成为主流的解决方案。通过深度的研究与优化，无量系统开发了能够支持千亿级参数模型训练的高性能计算框架，并通过模型管理，模型预测服务，实现了超大规模模型的训练，管理以及上线的全流程支持。

无量系统已经支持了 LR/FM/FFM/DNN 等多种常用模型，并在移动手机浏览器业务中实际使用和验证，帮助业务取得了巨大的业务指标提升。无量系统将逐步扩展功能，比如正在探索的基于 GPU 的深度学习技术，以覆盖更多的现有业务场景以及最新的 AI 类应用场景，为业务的进一步提升提供强大的系统支持。

## 作者介绍

袁镱博士，腾讯科技有限公司高级研究员。

## 参考文献

- [1] Hidden Technical Debt in Machine Learning Systems, Google. In NIPS'15
- [2] Wide & Deep Learning for Recommender Systems, Google 2016
- [3] 常见计算广告点击率预估算法总结 <https://cloud.tencent.com/developer/article/1005915>
- [4] 深度学习在 CTR 预估中的应用 <https://mp.weixin.qq.com/s/CMZHhxAMno2GlnQCjv0BKg>
- [5] Solving the stragglerproblem with bounded staleness. In HotOS (2013).
- [6] TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning
- [7] Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training

# 阿里巴巴为什么不用 ZooKeeper 做服务发现？

作者 坤宇



站在未来的路口，回望历史的迷途，常常会很有意思，因为我们会不经意地兴起疯狂的念头，例如如果当年某事提前发生了，而另外一件事又没有发生会怎样？一如当年的奥匈帝国皇位继承人斐迪南大公夫妇如果没有被塞尔维亚族热血青年普林西普枪杀会怎样，又如若当年的丘老道没有经过牛家村会怎样？

2008 年底，淘宝开启一个叫做“五彩石”的内部重构项目，这个项目后来成为了淘宝服务化、面向分布式走自研之路，走出了互联网中间件体系之始，而淘宝服务注册中心 ConfigServer 于同年诞生。

2008 年前后，Yahoo 这个曾经的互联网巨头开始逐渐在公开场合宣讲

自己的大数据分布式协调产品 ZooKeeper，这个产品参考了 Google 发表的关于 Chubby 以及 Paxos 的论文。

2010 年 11 月，ZooKeeper 从 Apache Hadoop 的子项目发展为 Apache 的顶级项目，正式宣告 ZooKeeper 成为一个工业级的成熟稳定的产品。

2011 年，阿里巴巴开源 Dubbo，为了更好开源，需要剥离与阿里内部系统的关系，Dubbo 支持了开源的 ZooKeeper 作为其注册中心，后来在国内，在业界诸君的努力实践下，Dubbo + ZooKeeper 的典型的服务化方案成就了 ZooKeeper 作为注册中心的声名。

2015 年双 11，ConfigServer 服务内部近 8 个年头过去了，阿里巴巴内部“服务规模”超几百万，以及推进“千里之外”的 IDC 容灾技术战略等，共同促使阿里巴巴内部开启了 ConfigServer 2.0 到 ConfigServer 3.0 的架构升级之路。

时间走向 2018 年，站在 10 年的时间路口上，有多少人愿意在追逐日新月异的新潮技术概念的时候，稍微慢一下脚步，仔细凝视一下服务发现这个领域，有多少人想到过或者思考过一个问题：服务发现，ZooKeeper 真的是最佳选择么？

而回望历史，我们也偶有迷思，在服务发现这个场景下，如果当年 ZooKeeper 的诞生之日比我们 HSF 的注册中心 ConfigServer 早一点会怎样？

我们会不会走向先使用 ZooKeeper 然后疯狂改造与修补 ZooKeeper 以适应阿里巴巴的服务化场景与需求的弯路？

但是，站在今天和前人的肩膀上，我们从未如今天这样坚定的认知到，在服务发现领域，ZooKeeper 根本就不能算是最佳的选择，一如这些年一直与我们同行的 Eureka 以及这篇文章《Eureka! Why You Shouldn't Use ZooKeeper for Service Discovery》那坚定的阐述一样，为什么你不应该用 ZooKeeper 做服务发现！

吾道不孤矣。

## 注册中心需求分析及关键设计考量

接下来，让我们回归对服务发现的需求分析，结合阿里巴巴在关键场景上的实践，来一一分析，一起探讨为何说 ZooKeeper 并不是最合适的注册中心解决方案。

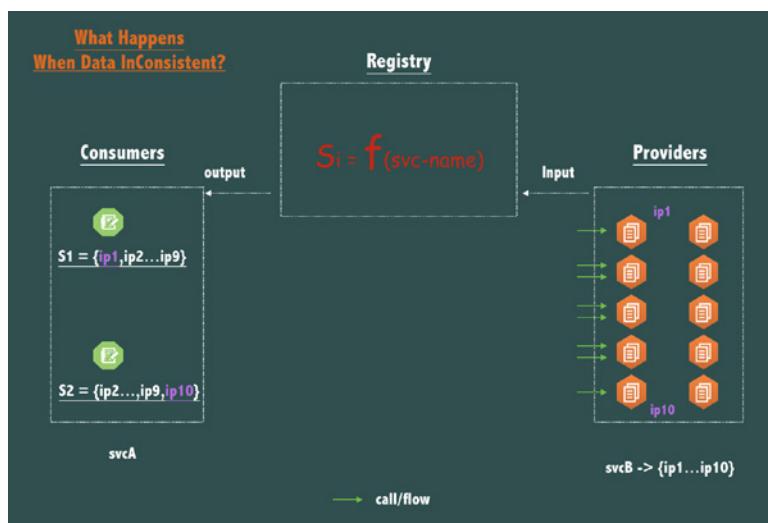
**注册中心是 CP 还是 AP 系统？** CAP 和 BASE 理论相信读者都已经耳熟能详，其业已成了指导分布式系统及互联网应用构建的关键原则之一，在此不再赘述其理论，我们直接进入对注册中心的数据一致性和可用性需求的分析：

- 数据一致性需求分析

注册中心最本质的功能可以看成是一个 Query 函数  $S_i = f(\text{service-name})$ ，以 service-name 为查询参数，service-name 对应的服务的可用的 endpoints (ip:port) 列表为返回值。

注：后文将 service 简写为 svc。

先来看看关键数据 endpoints (ip:port) 不一致性带来的影响，即 CAP 中的 C 不满足带来的后果：



如上图所示，如果一个 svcB 部署了 10 个节点 (副本 /Replica)，如果对于同一个服务名 svcB，调用者 svcA 的 2 个节点的 2 次查询返回了不一

致的数据，例如:  $S1 = \{ ip1, ip2, ip3, \dots, ip9 \}$ ,  $S2 = \{ ip2, ip3, \dots, ip10 \}$ , 那么这次不一致带来的影响是什么？相信你一定已经看出来了，svcB 的各个节点流量会有一点不均衡。

ip1 和 ip10 相对其它 8 个节点{ip2...ip9}，请求流量小了一点，但很明显，在分布式系统中，即使是对等部署的服务，因为请求到达的时间，硬件的状态，操作系统的调度，虚拟机的 GC 等，任何一个时间点，这些对等部署的节点状态也不可能完全一致，而流量不一致的情况下，只要注册中心在 SLA 承诺的时间内（例如 1s 内）将数据收敛到一致状态（即满足最终一致），流量将很快趋于统计学意义上的一致，所以注册中心以最终一致的模型设计在生产实践中完全可以接受。

- 分区容忍及可用性需求分析

接下来我们看一下网络分区（Network Partition）情况下注册中心不可用对服务调用产生的影响，即 CAP 中的 A 不满足时带来的影响。

考虑一个典型的 ZooKeeper 三机房容灾 5 节点部署结构（即 2-2-1 结构），如下图：



当机房 3 出现网络分区（Network Partitioned）的时候，即机房 3 在网络上成了孤岛，我们知道虽然整体 ZooKeeper 服务是可用的，但是节点 ZK5 是不可写的，因为联系不上 Leader。

也就是说，这时候机房 3 的应用服务 svcB 是不可以新部署，重新启动，扩容或者缩容的，但是站在网络和服务调用的角度看，机房 3 的 svcA 虽然无法调用机房 1 和机房 2 的 svcB，但是与机房 3 的 svcB 之间的网络明明是 OK 的啊，为什么不让我调用本机房的服务？

现在因为注册中心自身为了保脑裂 (P) 下的数据一致性 (C) 而放弃了可用性，导致了同机房的服务之间出现了无法调用，这是绝对不允许的！可以说在实践中，注册中心不能因为自身的任何原因破坏服务之间本身的可连通性，这是注册中心设计应该遵循的铁律！后面在注册中心客户端灾容上我们还会继续讨论。

同时我们再考虑一下这种情况下的数据不一致性，如果机房 1, 2, 3 之间都成了孤岛，那么如果每个机房的 svcA 都只拿到本机房的 svcB 的 ip 列表，也即在各机房 svcB 的 ip 列表数据完全不一致，影响是什么？

其实没啥大影响，只是这种情况下，全都变成了同机房调用，我们在设计注册中心的时候，有时候甚至会主动利用这种注册中心的数据可以不一致性，来帮助应用主动做到同机房调用，从而优化服务调用链路 RT 的效果！

通过以上我们的阐述可以看到，在 CAP 的权衡中，注册中心的可用性比数据强一致性更宝贵，所以整体设计更应该偏向 AP，而非 CP，数据不一致在可接受范围，而 P 下舍弃 A 却完全违反了注册中心不能因为自身的任何原因破坏服务本身的可连通性的原则。

## 服务规模、容量、服务联通性

你所在公司的“微服务”规模有多大？数百微服务？部署了上百个节点？那么 3 年后呢？互联网是产生奇迹的地方，也许你的“服务”一夜之间就家喻户晓，流量倍增，规模翻番！

当数据中心服务规模超过一定数量 (服务规模 = $F\{服务\ pub\ 数, 服务\ sub\ 数\}$ )，作为注册中心的 ZooKeeper 很快就会像下图的驴子一样不堪重负。



其实当 ZooKeeper 用对地方时，即用在粗粒度分布式锁，分布式协调场景下，ZooKeeper 能支持的 tps 和支撑的连接数是足够用的，因为这些场景对于 ZooKeeper 的扩展性和容量诉求不是很强烈。

但在服务发现和健康监测场景下，随着服务规模的增大，无论是应用频繁发布时的服务注册带来的写请求，还是刷毫秒级的服务健康状态带来的写请求，还是恨不得整个数据中心的机器或者容器皆与注册中心有长连接带来的连接压力上，ZooKeeper 很快就会力不从心，而 ZooKeeper 的写并不是可扩展的，不可以通过加节点解决水平扩展性问题。

要想在 ZooKeeper 基础上硬着头皮解决服务规模的增长问题，一个实践中可以考虑的方法是想办法梳理业务，垂直划分业务域，将其划分到多个 ZooKeeper 注册中心，但是作为提供通用服务的平台机构组，因自己提供的服务能力不足要业务按照技术的指挥棒配合划分治理业务，真的可行么？

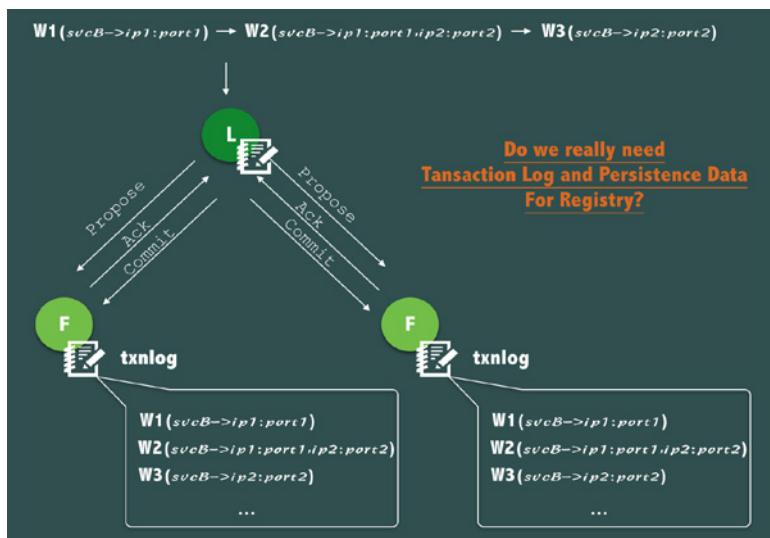
而且这又违反了因为注册中心自身的原因（能力不足）破坏了服务的可连通性，举个简单的例子，1 个搜索业务，1 个地图业务，1 个大文娱业务，1 个游戏业务，他们之间的服务就应该老死不相往来么？也许今天是肯定的，那么明天呢，1 年后呢，10 年后呢？谁知道未来会要打通几个业务域去做什么奇葩的业务创新？注册中心作为基础服务，无法预料未来的时候当然不能妨碍业务服务对未来固有联通性的需求。

## 注册中心需要持久存储和事务日志么？

需要，也需要。

我们知道 ZooKeeper 的 ZAB 协议对每一个写请求，会在每个 ZooKeeper 节点上保持写一个事务日志，同时再加上定期的将内存数据镜像（Snapshot）到磁盘来保证数据的一致性和持久性，以及宕机之后的数据可恢复，这是非常好的特性，但是我们要问，在服务发现场景中，其最核心的数据 - 实时的健康的服务的地址列表真的需要数据持久化么？

对于这份数据，答案是否定的。



如上图所示，如果 svcB 经历了注册服务 (ip1) 到扩容到 2 个节点 (ip1, ip2) 到因宕机缩容 (ip1 宕机)，这个过程中，产生了 3 次针对 ZooKeeper 的写操作。

但是仔细分析，通过事务日志，持久化连续记录这个变化过程其实意义不大，因为在服务发现中，服务调用发起方更关注的是其要调用的服务的实时的地址列表和实时健康状态，每次发起调用时，并不关心要调用的服务的历史服务地址列表、过去的健康状态。

但是为什么又说需要呢，因为一个完整的生产可用的注册中心，除了服务的实时地址列表以及实时的健康状态之外，还会存储一些服务的元数

据信息，例如服务的版本，分组，所在的数据中心，权重，鉴权策略信息，service label 等元信息，这些数据需要持久化存储，并且注册中心应该提供对这些元信息的检索的能力。

## Service Health Check

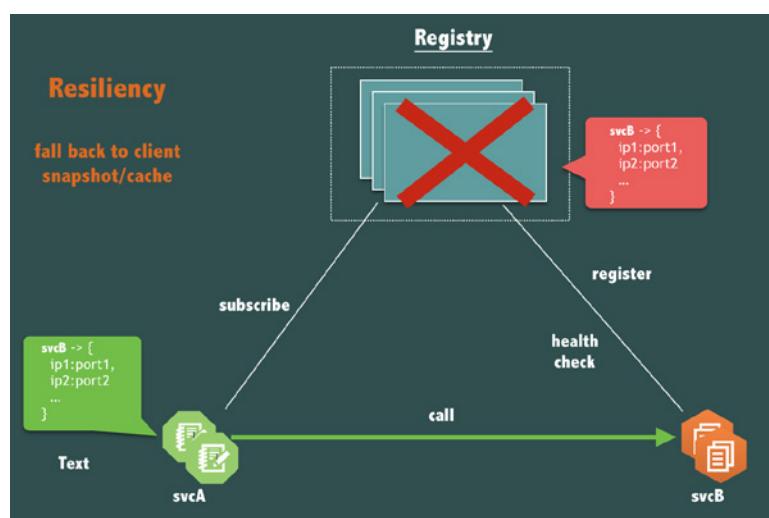
使用 ZooKeeper 作为服务注册中心时，服务的健康检测常利用 ZooKeeper 的 Session 活性 Track 机制以及结合 Ephemeral ZNode 的机制，简单而言，就是将服务的健康监测绑定在了 ZooKeeper 对于 Session 的健康监测上，或者说绑定在 TCP 长链接活性探测上了。

这很多时候也会造成致命的问题，ZK 与服务提供者机器之间的 TCP 长链接活性探测正常的时候，该服务就是健康的么？答案当然是否定的！注册中心应该提供更丰富的健康监测方案，服务的健康与否的逻辑应该开放给服务提供方自己定义，而不是一刀切搞成了 TCP 活性检测！

健康检测的一大基本设计原则就是尽可能真实的反馈服务本身的真实健康状态，否则一个不敢被服务调用者相信的健康状态判定结果还不如没有健康检测。

## 注册中心的容灾考虑

前文提过，在实践中，注册中心不能因为自身的任何原因破坏服务之



间本身的可连通性，那么在可用性上，一个本质的问题，如果注册中心（Registry）本身完全宕机了，svcA 调用 svcB 链路应该受到影响么？

是的，不应该受到影响。

服务调用（请求响应流）链路应该是弱依赖注册中心，必须仅在服务发布，机器上下线，服务扩缩容等必要时才依赖注册中心。

这需要注册中心仔细的设计自己提供的客户端，客户端中应该有针对注册中心服务完全不可用时做容灾的手段，例如设计客户端缓存数据机制（我们称之为 client snapshot）就是行之有效的手段。另外，注册中心的 health check 机制也要仔细设计以便在这种情况下不会出现诸如推空等情况的出现。

ZooKeeper 的原生客户端并没有这种能力，所以利用 ZooKeeper 实现注册中心的时候我们一定要问自己，如果把 ZooKeeper 所有节点全干掉，你生产上的所有服务调用链路能不受任何影响么？而且应该定期就这一点做故障演练。

## 你有没有 ZooKeeper 的专家可依靠？

ZooKeeper 看似很简单的一个产品，但在生产上大规模使用并且用好，并不是那么理所当然的事情。如果你决定在生产中引入 ZooKeeper，你最好做好随时向 ZooKeeper 技术专家寻求帮助的心理预期，最典型的表现是在两个方面：

- 难以掌握的 Client/Session 状态机

ZooKeeper 的原生客户端绝对称不上好用，Curator 会好一点，但其实也好的有限，要完全理解 ZooKeeper 客户端与 Server 之间的交互协议也并不简单，完全理解并掌握 ZooKeeper Client/Session 的状态机（下图）也并不是那么简单明了：

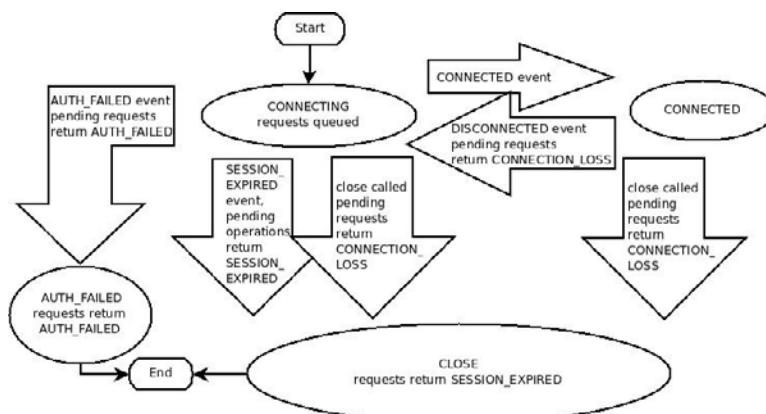
但基于 ZooKeeper 的服务发现方案却是依赖 ZooKeeper 提供的长连接 /Session 管理，Ephemeral ZNode，Event&Notification，ping 机制上，所以要用好 ZooKeeper 做服务发现，恰恰要理解这些 ZooKeeper 核心的机制

原理，这有时候会让你陷入暴躁，我只是想要个服务发现而已，怎么要知道这么多？而如果这些你都理解了并且不踩坑，恭喜你，你已经成为 ZooKeeper 的技术专家了。

- 难以承受的异常处理

我们在阿里巴巴内部应用接入 ZooKeeper 时，有一个《ZooKeeper 应用接入必知必会》的 WIKI，其中关于异常处理有过如下的论述：

如果说要选出应用开发者在使用 ZooKeeper 的过程中，最需要了解清楚的事情？那么根据我们之前的支持经验，一定是异常处理。



当所有一切（宿主机，磁盘，网络等等）都很幸运的正常工作的时候，应用与 ZooKeeper 可能也会运行的很好，但不幸的是，我们整天会面对各种意外，而且这遵循墨菲定律，意料之外的坏事情总是在你最担心的时候发生。

所以务必仔细了解 ZooKeeper 在一些场景下会出现的异常和错误，确保您正确的理解了这些异常和错误，以及知道您的应用如何正确的处理这些情况。

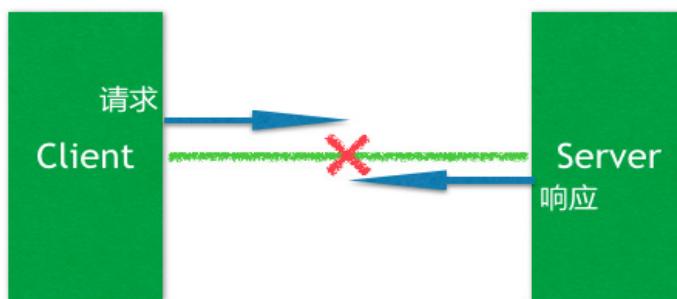
- ConnectionLossException 和 Disconnected 事件

简单来说，这是个可以在同一个 ZooKeeper Session 恢复的异常 (Recoverable)，但是应用开发者需要负责将应用恢复到正确的状态。

发生这个异常的原因有很多，例如应用机器与 ZooKeeper 节点之间网

络闪断，ZooKeeper 节点宕机，服务端 Full GC 时间超长，甚至你的应用进程 Hang 死，应用进程 Full GC 时间超长之后恢复都有可能。

要理解这个异常，需要了解分布式应用中的一个典型的问题，如下图：



在一个典型的客户端请求、服务端响应中，当它们之间的长连接闪断的时候，客户端感知到这个闪断事件的时候，会处在一个比较尴尬的境地，那就是无法确定该事件发生时附近的那个请求到底处在什么状态，Server 端到底收到这个请求了么？已经处理了么？因为无法确定这一点，所以当客户端重新连接上 Server 之后，这个请求是否应该重试（Retry）就要打一个问号。

所以在处理连接断开事件中，应用开发者必须清楚处于闪断附近的那个请求是什么（这常常难以判断），该请求是否是幂等的，对于业务请求在 Server 端服务处理上对于“仅处理一次”“最多处理一次”“最少处理一次”语义要有选择和预期。

举个例子，如果应用在收到 ConnectionLossException 时，之前的请求是 Create 操作，那么应用的 catch 到这个异常，应用一个可能的恢复逻辑就是，判断之前请求创建的节点的是否已经存在了，如果存在就不要再创建了，否则就创建。

再比如，如果应用使用了 exists Watch 去监听一个不存在的节点的创建的事件，那么在 ConnectionLossException 的期间，有可能遇到的情况是，在这个闪断期间，其它的客户端进程可能已经创建了节点，并且又已

经删除了，那么对于当前应用来说，就 miss 了一次关心的节点的创建事件，这种 miss 对应用的影响是什么？是可以忍受的还是不可接受？需要应用开发者自己根据业务语义去评估和处理。

- SessionExpiredException 和 SessionExpired 事件

Session 超时是一个不可恢复的异常，这是指应用 Catch 到这个异常的时候，应用不可能在同一个 Session 中恢复应用状态，必须要重新建立新 Session，老 Session 关联的临时节点也可能已经失效，拥有的锁可能已经失效……

我们阿里巴巴的小伙伴在自行尝试使用 ZooKeeper 做服务发现的过程中，曾经在我们的内网技术论坛上总结过一篇自己踩坑的经验分享

在该文中中肯的提到：

### 采用zookeeper的EPHEMERAL节点机制实现服务集群的陷阱

… 在编码过程中发现很多可能存在的陷阱，毛估估，第一次使用 zk 来实现集群管理的人应该有 80% 以上会掉坑，有些坑比较隐蔽，在网络问题或者异常的场景时才会出现，可能很长一段时间才会暴露出来 …

这篇文章已经分享到云栖社区，你可以点击 [这里](#) 详细阅读。

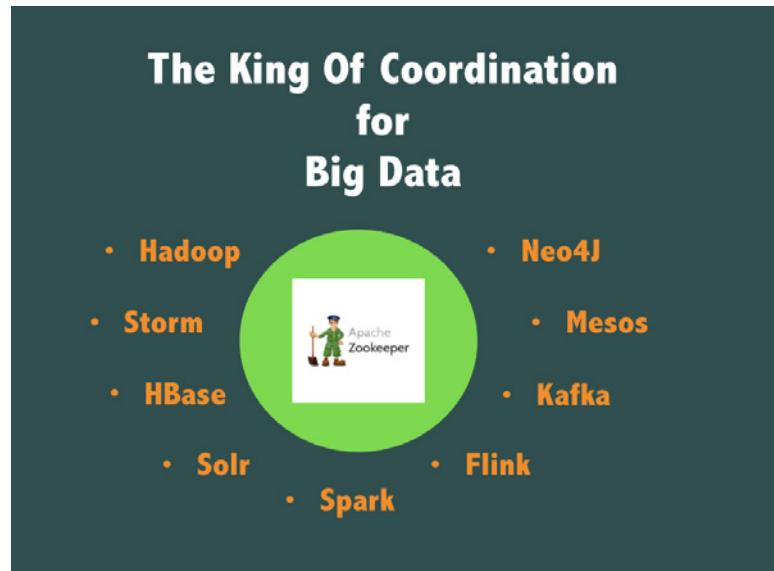
## 向左走，向右走

阿里巴巴是不是完全没有使用 ZooKeeper？并不是！

熟悉阿里巴巴技术体系的都知道，其实阿里巴巴维护了目前国内乃至世界上最大规模的 ZooKeeper 集群，整体规模有近千台的 ZooKeeper 服务节点。

同时阿里巴巴中间件内部也维护了一个面向大规模生产的、高可用、更易监控和运维的 ZooKeeper 的代码分支 TaoKeeper，如果以我们近 10 年在各个业务线和生产上使用 ZooKeeper 的实践，给 ZooKeeper 用一个短语评价的话，那么我们认为 ZooKeeper 应该是 “The King Of Coordination for Big Data” !

在粗粒度分布式锁，分布式选主，主备高可用切换等不需要高 TPS 支



持的场景下有不可替代的作用，而这些需求往往多集中在大数据、离线任务等相关的业务领域，因为大数据领域，讲究分割数据集，并且大部分时间分任务多进程 / 线程并行处理这些数据集，但是总是有一些点上需要将这些任务和进程统一协调，这时候就是 ZooKeeper 发挥巨大作用的用武之地。

但是在交易场景交易链路上，在主营业务数据存取，大规模服务发现、大规模健康监测等方面有天然的短板，应该竭力避免在这些场景下引入 ZooKeeper，在阿里巴巴的生产实践中，应用对 ZooKeeper 申请使用的时候要进行严格的场景、容量、SLA 需求的评估。

所以可以使用 ZooKeeper，但是大数据请向左，而交易则向右，分布式协调向左，服务发现向右。

## 结语

感谢你耐心的阅读到这里，至此，我相信你已经理解，我们写这篇文章并不是全盘否定 ZooKeeper，而只是根据我们阿里巴巴近 10 年来在大规模服务化上的生产实践，对我们在服务发现和注册中心设计及使用上的经验教训进行一个总结，希望对业界就如何更好的使用 ZooKeeper，如何更好的设计自己的服务注册中心有所启发和帮助。

最后，条条大路通罗马，衷心祝愿你的注册中心直接就诞生在罗马。

## 参考文章

- [1]<https://medium.com/knerd/eureka-why-you-shouldnt-use-zookeeper-for-service-discovery-4932c5c7e764>
- [2]<https://yq.aliyun.com/articles/227260>

# Airbnb 弃用之后，我们还应该用 React Native 吗？

作者 Charlie Cheever 译者 无明



近日，Airbnb 发表了一组由 5 篇博文组成的系列文章，他们在文章中宣布停止使用 React Native，并将其从代码库中移除，转而使用 Swift/Objective-C/Java/Kotlin。

在过去的几年中，在谈及“是否应该使用 React Native”这个话题时，通常都会有人指出，Airbnb 这家世界级的公司在产品方面做得非常出色，他们在 React Native 上投入了大量精力，并且正在使用它。然而，现在出现了大反转：一家关心产品质量的顶级公司对 React Native 进行了大量投入，在经过非常仔细的研究之后，决定弃它而去。对于任何想使用 React Native 的人来说，这都是一件非常可怕的事情。

另一方面，Airbnb 是 React Native 开源社区的重要贡献者。react-native-maps 和 Lottie 是两个非常重要的库（都包含在 Expo SDK 中），最初由 Airbnb 开发。Leland Richardson 在进入谷歌之前，曾经是该社区最著名的人物之一。人们肯定会记住他们所做出的贡献。

不过，如果你仔细阅读这一系列文章的内容，他们大部分时间都在说 React Native 相当不错，但不适合 Airbnb。就个人而言，这并不会让我感到十分惊讶。数以万计的开发者正在考虑使用 React Native，我与他们中的很多人进行过交谈，我发现考虑使用 React Native 的团队大致可以分为三大类，其中两类团队很可能会取得成功并乐在其中，而对另外一类团队来说可能是个噩梦。

## 我应该在项目中使用 React Native 吗？

这里有一个快速指南，可以帮助你和你的团队决定是否应该在项目中使用 React Native。

### 1. 你使用 React Native 从头开始构建一个新应用，并希望使用 JavaScript 开发所有的东西

如果是这种情况，人们通常都会很开心，并可以获得更好的结果。这个时候 Expo 非常适合你，你可以使用大量内置的原生模块，在不需要使用 Xcode 或 Android Studio 的情况下即可完成所有的事情，升级到新版本几乎毫不费劲，而且可以随时轻松推送代码更新，无需向应用商店提交新版本。如果由于某种原因，你需要使用原生代码开发一两个页面，并且已经定义好这些页面的边界，那么这么做通常也没什么问题。如果是我从头开始创建一个新应用，我会选择使用 Expo/React Native。

### 2. 你正在使用 React Native 开发少量的二级页面

如果你正在考虑使用 React Native 开发一些简单的二级页面，如设置、常见问题解答或“关于”页面（可能只需要把它们嵌入到 WebView），那么你就走运了。从使用体验方面看，这些东西不需要与应用程序的其他部分有密切联系，但整体观感却更像是“原生”的。

### 3. 你有一个使用 Swift/Java/Objective-C/Kotlin 开发的应用程序，现在你想要使用 React Native 开发其中的一部分

这种“棕色地带”（brownfield）的例子——你有一个使用 Swift 和 Java 开发的应用程序，你想要在跨多个视图或屏幕的地方引入 React Native——更难应付。对于 Airbnb 在这条道路上遇到了很多挫折，我并不感到惊讶。有经验的原生开发者在学习第二种完全不同的技术栈时也会感到沮丧。如果你在同一屏幕上同时使用原生视图和 React Native 视图，在 React Native 方面，你会将数据保存在 JS 对象中，而在原生方面，你会将数据保存在 Swift/Java 的数据结构中，要跟踪客户端状态就会变得很困难。因为 React Native 目前只有一个异步桥接，要在这个层面进行集成，可能会非常复杂，而且效率低下。现在想象一下其他 10 个类似的问题（导航、布局、委托方法、版本控制等）。如果一种技术的开发者总是要去处理另一种技术的最坏情况，那么最终一定会走上一条不归路。

### 4. 你的公司里有一个 iOS 团队和一个 Android 团队

即使你是前面两种情况中的一种，那些自认为拥有最强 iOS 程序员和 Android 程序员的公司也很难对 React Native 感到满意。iOS 程序员对此尤为不满，他们一般会认为 JS 是对公司代码库的污染，Android 程序员的感受则相对缓和一些。

即使你将 React Native 用在它所擅长的领域，但因为一些非技术问题，要让原生开发和 React Native 开发在企业中大规模并存仍然很困难。

至于它的价值，我几乎赞同 Airbnb 博文中列出的所有对 React Native 的指责。我们可以在 Expo 中发现很多相同的东西。而且我自己列出的受挫清单比这个要长得多。不过，这项技术在很多方面都表现得非常好，而且会越来越好。例如，人们普遍认为，想要获得良好的导航观感就要使用原生导航，但现在来自 Expo 团队的 Brent 已经在 react-navigation 库上进行了大量工作，它很好用，观感也很不错。它已经成为大多数 React Native 应用程序导航的最佳选择。

我对这个项目表示乐观，因为 Facebook 的一些最有热情的人（特别

是 Hector、Sophie 和 Ram) 正在重构 React Native，并制定了很多明智的计划，解决了一些最重要的问题。

我认为 React Native 正处在一个很好的位置，其中的一个原因是微软在新版本的 Office 中使用了 React Native。

从宏观角度来看，使用像 JavaScript 这样的脚本语言来开发移动应用程序几乎是不可避免的，因为使用 Swift/Objective-C/Java/Kotlin 这些语言来开发 UI 效率太差。此外，每个应用程序都要开发两次（或者如果算上 Web 就是三次），这根本就是个大麻烦。无论是 React Native、Flutter 还是其他羽翼未满的新产品，它们也都大致如此。就我个人而言，我对获胜者的机会猜测如下：React Native 55%、Flutter 15%，其他我们还看不到的 30%。

# 如何“计算”CEPH读写性能

作者 任祥



Ceph作为SDS的杰出代表，借着云计算的东风，可谓是红遍大江南北，老少皆知（这个有点扯，回家问问你爷爷听过ceph没）。不管是SDS也好，还是SDN也好，所谓SDX系列均是以高度灵活性著称，这就给玩家一个广阔的想象空间。所有的部件均是灵活自主可控的，你可以自己随意组装，就像自己装电脑一样。

那么问题来了，这样搭建出来的集群到底能达到什么水平呢？读写性能有多高？不要告诉我搭建出来测试一下不就知道了，如果测试结果不能满足业务需求呢，再建一套？那客户还留你何用！

## 进化三部曲

- 1.最近一直在思考一个存储集群的读写性能到底能达到什么水平？
- 2.跟构成集群所用单块磁盘到底是怎样的关系？
- 3.能不能通过计算得出集群的性能指标？

## 路人甲的思考

在了解ceph的原理之前，大脑中有这样一个概念：既然集群是有很多块磁盘构成的，那么集群的性能应该就是所有磁盘的性能之和，假如一个集群100块磁盘构成，那么集群的整体性能应该就是这100块磁盘之和。例如单块磁盘的IO有500，那么集群的IO应该就是 $500 \times 100 = 50K$ 。想到这里，心里还挺满意的，这意味着使用ceph集群不但能够获得大容量，而且性能也是成倍的增加！

## 与君初见

慢慢了解到一些ceph的基础知识以后发现，好像现实并不是想象中的那么美好，特别是看了一遍陈导总结的《Ceph读写性能估算方法》之后，发现原来这里面还有很多值得思考的点，例如：三副本（一份文件存三份，只有1/3的空间利用率）、waf（写放大，二次写journal导致写IO减半）、还有各种元数据、DB、wal占用集群空间。这样算下来，我组装的集群可用容量减到三分之一，磁盘IO减半，最后集群的性能还不如单块磁盘的性能高。难道ceph是江湖骗子拿来忽悠的？

## 无形胜有形

听君一席话，胜读十年文档，经过高人点拨之后茅塞顿开。从容量上来说，ceph可以灵活配置副本数量，合理规划空间使用。如果你觉得默认的三副本策略太浪费磁盘，也可以配置两副本数来实现50%的磁盘空间使用率，或者单副本（不过我还是劝你等到头脑清醒的时候再做决定）实现100%。

除了副本策略外，ceph还支持纠删码策略，通过灵活配置K+M比例，

合理规划磁盘冗余度，来实现整个集群的高可靠性，例如，按照K=3，M=2配置纠删码冗余策略，当有一组数据写入时，该数据会被分成3份分别写入3个OSD中，同时ceph也会生成2个编码块写入到另外2个OSD中，一次写入操作，实际上会产生 $(K+M)/K$ 倍的实际写入量，在保证集群高可靠性的同时大大提高磁盘使用率。

从读写性能上来说，并不是上文所说的创建集群之后由于写放大的缘故，导致整个集群的性能下降为单块磁盘的50%。大家都知道ceph是一种分布式的存储系统，同时也是一种对象存储。当然这里所说的对象并不是指对用户提供对象存储的功能，这里的对象是ceph或者说rados在内部处理数据的方式。通过外部接口（对象存储接口RGW、块存储接口RBD、文件系统存储cephFS、当然还有我们自研的HDFS接口、流媒体接口等）写入的数据分片，生成许多小的对象，然后再把这些小的对象均匀的写入到各个物理磁盘上面。

了解过ceph的人应该都听过PG的概念，这些内部的小对象就是通过PG这个模块来进行分组，属于同一组的内部小对象会有相同的物理磁盘位置。例如第一组小对象的三个副本会写入到osd1、2、3上面，第二组小对象写入到osd2、3、4上面。如果集群内部的PG数量足够多，用户数据就会均匀的分布到各个物理磁盘上面，这样同一时刻用户存入的数据就会同时在不同的物理磁盘上面进行写入。虽然集群写入速度不是所有磁盘之和，但也是多块磁盘性能累加的，最后用户感知到的性能也是可观的。

## 预测“未来”

了解了大概原理以后，如何根据用户提供的硬件资源来预估集群的性能呢？

## 计算公式

### FileStore + 多副本

条件假设一：

1. 假设每块磁盘作为一个OSD，这个OSD的journal和data都放在这块

磁盘上。所有数据都是先写到journal上，然后再写到data上，也就是单OSD的写放大系数是2；

2. 假设OSD个数为N；
3. 假设副本数是M，数据直到写入M个OSD之后才响应，因此对于多副本存储池，写放大系数是M；
4. 因为Ceph软件会损耗CPU资源，也会损耗一些性能，损耗系数定为0.7；
5. 假设单块SSD的4K随机读IOPS是R，4K随机写IOPS是W。

在不考虑网络瓶颈和CPU瓶颈的情况下，Ceph存储池的IOPS估算公式是：

1.  $4K\text{随机读IOPS} = R * N * 0.7$
2.  $4K\text{随机写IOPS} = W * N * 0.7 / (2 * M)$

条件假设二：

1. 假设每块SATA磁盘作为一个OSD，有一块NVME磁盘专门作为journal。所有数据都是先写到journal上，然后再同步到data上，也就是单OSD的写放大系数就变成1（假设NVME性能大于所有本机SATA盘之和）；
2. 假设OSD个数为N；
3. 假设副本数是M，数据直到写入M个OSD之后才响应，因此对于多副本存储池，写放大系数是M；
4. 因为ceph软件会损耗CPU资源，也会损耗一些性能，损耗系数定为0.7；
5. 假设单块SSD的4K随机读IOPS是R，4K随机写IOPS是W。

在不考虑网络瓶颈和CPU瓶颈的情况下，Ceph存储池的IOPS估算公式是：

1.  $4K\text{随机读IOPS} = R * N * 0.7$
2.  $4K\text{随机写IOPS} = W * N * 0.7 / (M)$

**BlueStore + 多副本**

条件假设一：

1. 假设每块磁盘作为一个OSD，该磁盘划为2块分区：一个分区作为裸盘来写入数据，另一块做BlueFS用来跑RocksDB。因此我们一次写入的流程可以简化成下图：数据会被直接写入到data分区(裸盘)中，而对象元数据会被写到RocksDB和RocksDB的WAL中，随后RocksDB将数据压缩后存放到磁盘中。我们不再需要在文件系统层做journal，而WAL只在覆写操作时才会用到，因此在副本数量为N的条件下，我们可以推测WAF将收敛于N，也就是单OSD的写放大系数是1。
2. 假设OSD个数为N；
3. 假设副本数是M，数据直到写入M个OSD之后才响应，因此对于多副本存储池，写放大系数是M；
4. 由于ceph软件会损耗CPU资源，也会损耗一些性能，损耗系数定为0.7；
5. 假设单块SSD的4K随机读IOPS是R，4K随机写IOPS是W。

在不考虑网络瓶颈和CPU瓶颈的情况下，ceph存储池的IOPS估算公式是：

1.  $4\text{K随机读IOPS} = R * N * 0.7$
2.  $4\text{K随机写IOPS} = W * N * 0.7 / (M)$

注意：在BlueStore中，磁盘分区会以‘bluestore\_min\_alloc\_size’的大小分配管理，这个数值默认为64KiB。也就是说，如果我们写入<64KiB的数据，剩余的空间会被0填充，也即是‘Zero-filled data’，然后写入磁盘。也正是这样，BlueStore的小文件随机写性能并不好，因此在小文件计算式可以适量减少损耗系数。

条件假设二：

1. 假设每块SATA磁盘作为一个OSD，有一块NVME磁盘专门作跑RocksDB。数据会被直接写入到data分区(裸盘)中，而对象元数据会被写到RocksDB和RocksDB的WAL中，也就是单OSD的写放

大系数就变成1;

2. 假设OSD个数为N;
3. 假设副本数是M，数据直到写入M个OSD之后才响应，因此对于多副本存储池，写放大系数是M;
4. 因为ceph软件会损耗CPU资源，也会损耗一些性能，损耗系数定为0.7;
5. 假设单块SSD的4K随机读IOPS是R，4K随机写IOPS是W。

在不考虑网络瓶颈和CPU瓶颈的情况下，ceph存储池的IOPS估算公式是：

1.  $4K\text{随机读IOPS} = R * N * 0.7$
2.  $4K\text{随机写IOPS} = W * N * 0.7 / (M)$

## FileStore + 纠删码

相比较多副本冗余策略，纠删码的出现大大节省了磁盘空间，如果我们有N个OSD，并且按照K=3, M=2配置纠删码冗余策略。一次写入操作，实际上会产生 $(K+M)/K$ 倍的实际写入量。而由于这里使用的存储后端是FileStore，journaling of journal问题会让写放大两倍，因此结合纠删码本身特性，WAF最终会收敛于 $(K+M)/K * 2$ 。

在不考虑网络瓶颈和CPU瓶颈的情况下，Ceph存储池的IOPS估算公式是：

1.  $4K\text{随机读IOPS} = R * N * 0.7$
2.  $4K\text{随机写IOPS} = W * N * 0.7 * 2 * K / (K+M)$

## BlueStore + 纠删码

有了前文的铺垫后，相信到这一步大家能够自己推演出BlueStore在纠删码策略下的写入性能推导公式。由于解决了journaling of journal问题，每次写入不再需要通过文件系统，因此WAF最终将会收敛于 $(K+M)/M$ 。

在不考虑网络瓶颈和CPU瓶颈的情况下，Ceph存储池的IOPS估算公式是：

1. 4K随机读IOPS = R\*N\*0.7
2. 4K随机写IOPS = W\*N\*0.7K/(K+M)

## 目标是星辰大海

通过上述计算得出的结果就可以直接用来作为参考了么？答案当然是否定的。

以上的计算我们仅仅是考虑了磁盘这一单一变量，除此之外还有网络、cpu资源等都需要列入计算。

ceph是一个分布式的系统，任何一个短板都会影响集群的对外输出性能。

例如网络资源：假设单节点有10块SATA磁盘，每块的读带宽是100MB/s，按照上面的公式，单节点的读带宽大概是 $100*10*8=8G/s$ 。假设此时机器上面只有两个千兆端口，即使是做了RR捆绑，也只能提供2G/s带宽，此时集群的性能参考可能就是网络端口的极限值了。

再例如CPU资源：按照经验值单核处理的写IO大概是2000左右，假设一台机器配置了20块50K IOPS的SSD，此时的性能极限就很有可能被CPU限制了。

总之，如何在纷繁的选择中，描绘出最美“画面”，是一件很“艺术”的事情。

# 中心化 or 去中心化？聊聊交易所的辩证发展

作者 Linda



## 编者语

数字货币的交易分为很多环节，包括充值、下单、订单撮合、资金结算、提现等。中心化的平台包办了以上所有过程，而其优势在于可以快速进行订单的撮合。但是，一旦系统被黑客入侵，一个小小的漏洞就可能酿成巨额损失。因此，去中心化交易所的概念在圈内广泛流传，相关应用已经落地，Waves DEX、BitShares、NXT、CounterParty等也都为大家熟知。

为了将去中心化交易模式与传统交易模式系统对比，从功能、技术架

构、应用趋势等角度呈献给开发者，我们采访了 BlueHelix CEO 巨建华先生。他曾担任火币网CTO、小羸科技创新研究院负责人，从2014年开始一直从事区块链业务，对数字资产交易和区块链技术应用有深入理解和实践。

2018全球区块链生态技术大会有幸邀请到巨建华老师来现场分享《中心化&去中心化区块链资产交易系统》，在此了解大会详情。在此之前，我们先同建华老师，聊聊交易所的辩证发展。以下内容根据采访内容整理而成。

### “中心化”模式频遇黑客，去中心化交易所能否扭转乾坤？

自比特币出现以来，一些知名的中心化代币交易所接二连三遭受黑客攻击，从Mt.Gox被盗走85万个比特币，到Bitstamp被盗损失510万美元，再到Bithumb的3万多名客户数据泄露和财产损失，后有Coincheck遭黑客入侵损失达4亿美元，还有今年3月币安遭黑客攻击疑似在其他交易所做空获利事件……

我们可以看到，黑客将矛头指向了这些比特币交易所的“中心化”模式。人们也很困惑，为什么去中心化的产品要依靠中心化的交易所来进行交易，去中心化的交易所还来不来？这就需要从整个行业的发展现状说起。

## 目前，去中心化交易所是作为中心化交易所的补充而存在

现阶段，数字货币的交易场所有3种：中心化交易所，去中心化交易所，和场外 OTC 点对点交易。场外点对点交易因为点对点交易带来的撮合效率低下，基本上只作为大额交易和规避法律风险的交易途径存在。当前的主流交易都是在中心化交易所内完成的。

而因为数字货币本身的去中心化特性，很多人也在反思是否有更好的交易方式，来避免上文提到的中心化交易所的种种弊端，于是通过“去中心化交易所”的模式来解决问题，越来越受到人们的关注。

去中心化交易所的追求的本质是在信任层面去中心化，从技术实现的

角度来讲，就是通过密码学和区块链的特性实现由技术达到的信任。在去中心化交易所中，资金的释放是由用户通过数字签名直接授权的，原则上讲是不可能被盗的。

但是，由于资金的流动性受限，限制了交易者的机会；用户需要支付更高的交易手续费；并且拥有相对较差的体验。

目前来看，去中心化交易所是作为中心化交易所的补充存在的。原因是很多小币种因各种原因无法挂牌主流的中心化交易所而选择在去中心化交易所挂牌，或是某些新发行的币种在登陆主流的中心化交易所之前会被去中心化交易所抢先挂牌。

出现这种现象，是因为区块链发展早期，并没有智能合约等基础设施，中心化交易所是唯一的选择。后期虽然基于智能合约与跨链技术的去中心化交易所开始出现，但在用户体验上仍然很难与中心化交易所匹敌。

当然，我们也观察到去中心化交易所的进步，目前IDEX（是一个基于以太坊的分散式智能合约交易平台，提供实时交易）的日均成交量已经达到1000BTC左右，看上去比较可观，但还远远不及主流交易所的成交量。

预计在区块链底层基础设施完善前，市场依旧会是由主流交易所主导，只有当撮合效率、交易成本以及用户学习成本能与中心化交易所相似时，中心化交易所才有可能占据更大的交易份额。

得出上述结论的具体原因，要从用户需求的角度来看。用户“交易”的目的是将自己手中的币以一个能接受的价格兑换成另一种币。交易能否达成，取决于两个方面：效率和代价。

效率，是指用户想交易时，能不能快速找到对手方，能不能快速找到理想的价格，能不能快速完成交易，交易完成后能不能快速交割；代价包括行业的共识推广代价，用户的教育学习代价，以及用户在每一笔交易的背后，付出的显性及隐性代价。

## 新的交易理念，正在推动数字货币交易业务的改革

我们可以看到，现实生活中的股票、外汇以及其它的金融衍生品，都经历过从 case by case，点对点逐步过渡到区域中心化、国家中心化的一个演化过程。理论上，效率最大化和代价最小化的终极形态应该是，全球的某一个品类的交易都在同一个地方集中完成，但是受限于不同国家的监管差异，这样的终极形态并没有出现。但是这些中心化的交易所背后都是有国家信用作保障。

而在数字货币交易市场，有一些项目的“去中心化”，落在资产托管层，让资产去中心化托管在智能合约上，甚至不托管资产，仍留在交易参与方自己的手上。这样带来的问题是，如果是托管在同一个智能合约上，这个合约本身又是一个“中心”，可能出现bug或其它的安全问题，如果是托管在不同的智能合约上，或者不托管，那交易完以后的交割过程效率就非常低下。

虽然目前去中心化交易所还不是主流，但这些新的交易理念，正在不停的推动着数字货币交易业务的改革，一些全新的交易模式也正在探索过程中。去中心化交易所在未来很长一段时间会和中心化交易所并存，向不同需求的用户提供服务。

## 两种模式的交易所在技术架构与功能方面存在哪些差异？

去中心化交易所模式简单，它只需要承担主要的资产托管、撮合交易及资产清算。而不需要承担像中心化交易所所需要承担的非交易的功能像账户体系、KYC、法币兑换等。

传统交易所，基本上都是使用数据库存储交易帐户信息，使用不开源的代码来实现功能，并部署在公司所有的内网机器或云虚拟主机上。它的运行过程对于外界来说更像是一个黑盒子。

这就导致了交易所内部钱包的开发营运人员或拿到保存私钥机器权限的黑客，就能挪动用户的资产；交易和清结算模块的开发人员或黑客，就能够做内部交易，甚至直接修改账目实现难以追查的非法资产转移；交易平台营运方本身也可以不受监管的制造假交易量影响市场行情。

去中心化交易所，一般都采用智能合约方式编写交易撮合和清结算逻辑，并且将合约代码开源出来供所有人查看。这样的话，代码是公开的，又是运行在链上，就可以在一定程度上保障用户的资金安全，防止内部交易。

但是，去中心化交易所也存在不稳定因素。为了获得用户信心，合约代码是需要开源的，谁都可以拿这份代码资金单独部署一套交易所，长期下去会导致多个交易平台出现，多个交易平台的竞争将导致市场流动性的分割，进而导致单个交易市场的深度不足，影响到参与用户的交易机会成本，也会带来价格不稳定的因素。

去中心化交易所因为区块链本身的特性和不同去中心化交易所的技术方案不同，目前很难在技术层面集中解决去中心化交易所带来的流动性分割的问题，一般是通过市场自动平衡价格的力量来解决。

虽然也有部份团队在研究“通过成熟的跨链技术来实现跨去中心化交易所间的业务协作”，却面临着巨大的技术挑战和用户体验问题，难以完美实现流动性整合。但相信随着未来区块链底层技术的发展，这些问题会逐步得到解决。

## 交易所的“去中心化”如何落地？

既然“去中心化”已是大势所趋，那就会不断有新的去中心化交易所概念出来。同时，各个去中心化的交易所都有自己的优点和缺点，它们引入了新的概念模型来重构交易模式，却在功能上存在很大的差异，究竟哪些模式的发展潜力比较大？

### 1. 选择合适的模型

目前，去中心化交易所主要有三大类：

- 第一类是继续基于撮合的模式，跟中心化交易所一样。但为了解决用户资产安全、交易透明等问题，选择把资产托管、订单簿维护、交易撮合、清结算四大步骤中的一个或多个放在链上进行。

把资产托管到链上的比如 Bitshare 比特股，托管到智能合约上的比如使用 0x 协议的ethfinex、district0x、Augur和melonport等。把交易撮合托管到链上的典型例子是以太坊上最早的去中心化交易所EtherDelta。订单簿如果托管在链上，意味着挂单和撤单都需要付出手续费成本，目前很少使用。近期新出现的像 dex.top 一般都采用中心化撮合，异步清结算上链的方式，来达到提高性能和流动性、降低延时的目的。

- 第二种是采用半中心化的“网关”或交易对手方来聚集订单，增强流动性。典型的例子是 kyber，它提供了一个“代币储备库”机制，卖方提前将代币存储到储备库中，等待买方过来下单。买方下单后能够立即完成成交和结算，而且这个过程是发生在链上的。
- 第三种是点对点的场外交易模式，以Airswap为代表。严格来说 Airswap 并不是一个交易所，而只是一个信息中介：它为交易用户提供寻找有交易意向的对手的一种机制，还提供了一种价格协商（包括从外部可信源获取价格建议）和订单传递机制，在整个交易过程中 Airswap 都不托管资产，结算也由公开的智能合约完成。

从目前来看，我们认为这三种模式都有各的优势，通过不同的技术选择实现了不同程度的资产安全级别，各自具有不同的交易成本和交易效率，短期内除了用户积累导致的差异外，没有具体明显的差异，长期来看不同模式的去中心化交易所都将占据一定的市场份额，但成为市场主流还面临很大的挑战。

## 2. 从系统设计入手

在数字货币领域有一句非常贴切的格言：“Don't trust, verify! ”，意思是说，这个领域我们不应该轻易的相信任何人，而是应该自己去验证。用什么验证？机制，协议，算法，秘钥，等等。

当前中心化交易所存在着种种问题，其本质在于怎么能让用户自己去验证资产安全，验证交易所没有监守自盗，验证每一笔撮合交易是合规合理的（没有恶意爆仓杠杆期货），验证交易价格不是被人操控的，验证交易所跟发币的项目方没有在私底下操作任何的猫腻。

在现实世界中，这些都是通过国家监管机构来规范和审查的。但在数字货币世界，这些都需要设计精巧的机制协议算法等来让市场参与者自行验证。

因此，去中心化交易所系统设计的重点实现目标是，确保用户和交易指令不受中心化的机构操纵，解决资产托管控制和交易指令的产生，以及解决交易结果的结算问题。比如在托管钱包层，可以通过设计多签名地址或共享密码的方式，让交易所未得到用户授权时，不能私自变动用户的资产；在交易和撮合层面也通过密码学实现交易授权的检验，保证订单和成交记录的有效授权和交易数据的公开透明，不可篡改和伪造。

### 3. 性能优化因地制宜

去中心化交易所的困难在于：交易所，特别是支持高频交易、量化交易等高级金融功能的交易所，它产生的数据量是非常大的，它对数据的一致性，及时性的要求是非常高的，这与目前各种区块链实现的特性从本质上就是冲突的。

通过区块链技术改造中心化交易所业务，使中心化交易所具备去中心化交易的可信度，目前也是各个中心化交易平台的主流发展方向，我们也看到目前各个交易平台都在发展自己的底层区块链团队，试图通过技术解决这类问题提高交易所可信度和竞争力。

### 4. 协议规范极具挑战

去中心化交易所和整个数字货币、区块链领域现阶段最大的挑战，还是在于并没有达成足够广泛的社会共识。

从技术角度来说，就是没有形成事实上的协议规范，导致各个底层链和各个项目方都是孤军作战，无法形成合力，从而将导致交易深度分散，

影响用户直接利益。

同时去中心化交易所对于KYC等严格金融临管的合规执行落地能力有限，将会导致监管环境恶化，很大程度会影响到去中心化交易所的发展。

## 5. 盈利模式舍变求稳

从用户的角度出发，“交易”的参与用户都是为了盈利而来的，而盈利，基本上可以概括为“开源节流”四个字。

开源，即提高收益，让用户能便捷的买到更具升值价值的代币，能方便的使用各种金融工具手段降低亏损风险；节流，即降低交易成本，包括降低手续费、机会成本、用户的学习成本、寻找成本等。

这些都可以参照现实世界的各种金融服务，为数字货币交易参与方提供各种增值服务，各种辅助工具来实现。

总的来说，当前的数字货币交易所，包括中心化和去中心化的，在金融服务和衍生品方面都还没有达到传统金融市场的成熟阶段，目前阶段只要将现实世界金融行业已经证明有效的各种金融工具和金融业务模式搬过来改造和创新，都会存在较多的盈利机会。

## 6. 与“中心化”辩证取舍

快速崛起的EOS，号称用21个超级节点实现多中心交易，这看上去也算是“去中心化”。而5月29日发现的EOS安全漏洞引起业内关注。可以看出，去中心化改造的过程中，依旧存在严峻的安全考验。

无论是去中心化交易所还是中心化交易所都是软件，是软件就有漏洞。从这个角度看，二者面临着同样的代码风险，不能过分强调去中心化交易所的安全性。去中心化交易所在发展的过程中，一定会经受与中心化交易所相同的考验。

中心化和去中心化，应该是不同业务场景不同阶段的自然选择，而不应该是人为预先设定。即使是EOS号称“区块链3.0”的系统，背后的开发团队仍然是中心化的。

因为在业务的开发迭代阶段，有很多的不确定性需要尝试，需要快速

演进，所以必须中心化，“集中力量办大事”；在业务稳定成熟以后，为了获得更大范围的共识和应用，必须去中心化，以增加系统影响范围，扩大生态圈。

就计算机软件的本质来说，不管系统处于什么阶段，系统（包括软件实现，机制协议设定等）都不可能是百分百完美没有BUG，也不可能百分之百能永远适应业务需求。一个良好运转的软件系统，并不是依靠它完美的当前实现来保障，而是依赖它拥有根据需要随时修复和升级、演进增强的机制来实现，一个不能演进的系统是没有生命力的。

在“去中心化交易所”中，智能合约和区块链共识协议的本质决定了去中心化交易所在系统迭代方面存在着天然的约束，这即是它在业务的可信度层面的优势，也是它做为软件系统的劣势。如何面对软件系统生命周期的“发现问题”→“提出解决方案”→“基于方案获得共识”→“开发实现上线更新”→“发现新的问题”的循环过程，才是真正意义上完全去中心化的交易所面临的挑战。

## 7. 应用场景由“小”及“大”

短时间内去中心化交易系统仍只适合小币种的交易，对于价格变化迅速的大币种来说，目前的技术水平还难以满足用户对速度的要求。

因为数字货币本身跨国界的特性，去中心化的，由共识规则和开放源代码的智能合约以及加密算法支撑的去中心化交易所，最终一定可以实现交易所的终极形态：对于同一个品类的交易，全球的用户都能共享流动性，享受同样的价格和服务，再也不会被各种人为设置的壁垒所分割，真正达到“世界大同”的境界。

## “混战”长期持续

目前去中心化交易所相对于中心化交易所最大的卖点是安全、可信，如果中心化交易所可以向传统金融中的交易所演进，由政府或一个具有公信力的组织来监管，那么中心化交易所在可信度上也是可以提高的。

去中心化交易所跟中心化交易所并不是无法共存的矛盾体，事实上，我们认为，在未来很长一段时间里，都会是以中心化为主，去中心化为辅。

随着区块链技术和金融共识的演进，慢慢过渡到以区块链支撑的中心化交易所形态。每个中心化交易所，实际上都是一个去中心化体系中的超级节点，交易所的交易指令和资产都在可信的区块链上执行和流转，同时接受监管机构的全面监管，通过技术和监管的双重手段解决业务作恶的可能，从根源上解决了当前中心化交易所的各种问题，可以为交易提供更好的交易体验，同时也能推动数字货币行业向着更健康的方向发展。



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





## 架构师 月刊 2018年6月

本期主要内容：Google 力推的那些前端技术，最近有何进展？如何做好文本关键词提取？从达观数据应用的三种算法说起；优秀架构师必须掌握的架构思维；区块链没有未来，是时候抛弃它了



## 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。