

架构师

ARCHITECT



热点 | Hot

谷歌新发布的分布式数据库服务
Yahoo开源TensorFlowOnSpark

专题 | Topic

禁用GC, Instagram性能提升10%
复盘GC算法的发展历程及现状

观点 | Opinion

为什么Google用Apache Beam
彻底替换掉MapReduce



CONTENTS / 目录

热点 | Hot

谷歌新发布的分布式数据库服务

Yahoo 开源 TensorFlowOnSpark

推荐文章 | Article

专访 RocketMQ 联合创始人：项目思路、技术细节和未来规划

百亿级微信红包的高并发资金交易系统设计方案

观点 | Opinion

为什么 Google 用 Apache Beam 彻底替换掉 MapReduce

专题 | Topic

禁用 Python 的 GC 机制后，Instagram 性能提升 10%

复盘 GC 算法的发展历程及现状

特别专栏 | Column

大容量高并发分布式呼叫中心架构设计



架构师 2017 年 3 月刊

本期主编 郭 蕾

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

卷首语

聊一聊技术采用生命周期

InfoQ 中文站主编 郭蕾

最近，在 InfoQ 的邮箱中，我收到了一封关于 C++ 某个关键字用法的投稿文章。虽然文章结构和内容都很不错，但我还是给作者回复了拒收的邮件。原因很简单，在 InfoQ 的介绍页面中明确有写，我们关注的是技术采用生命周期中，处于“创新”与“早期应用”部分的关键技术，而非那些早已进入成熟和衰退期的技术。紧接着，一个问题随之而来，什么是技术采用周期，为什么 InfoQ 不去关注其他几个阶段的技术？今天我也想借着为《架构师》杂志写卷首语的机会，聊聊这个话题。

技术采用生命周期（Technology Adoption LifeCycle）是一个用来衡量用户对某项新技术接受程度的模型（如下图所示，图片来自 Stockfeel）。这个理论最早源于 1943 年 Ryan 和 Gross 对玉米新品种的扩散行为研究，而后 Everett M. Rogers 提出了扩散曲线，并将采用者分为五种类型。1962 年，随着 Everett M. Rogers 所撰写的《创新的扩散》一书的出版，技术采用生命周期这一研究开始被学术界和工业界所重视，逐渐变得流行起来。



技术采用生命周期的形状是一个钟形曲线。这一曲线将消费者采用新技术的过程分成五个阶段，分别包括创新者（2.5%）、早期采用者（13.5%）、早期大众（34%）、晚期大众（34%）与落后者（16%）。以 Golang 为例子，各位也可以想想是不是这样的应用模型。

- 创新者是一群技术狂热分子，他们喜欢拥抱新技术，并把这视为其生活最大的乐趣。所以在Golang发布之时，虽然有很多的坑，但却不乏众多爱好者乐于踩坑。
- 早期采用者也是新技术发展的主要推动者，但与创新者不同的是，他们并非技术专家，在选择新技术时，更愿意尊重自己的直觉和喜好。很多人在看到Golang的语法以及惊艳特性之后，也开始加入逐步探索应用，而这群人，就是早期采用者。
- 对于新技术来说，早期大众是新技术能否成功应用的关键（占比近1/3）。他们对于新技术产品的采用往往比较谨慎，在采用之前，会重点关注其他用户的使用经验。Golang到现在已经发布了1.8版本，相信有很多公司还在观望阶段，他们想在各个的技术会议或者论坛上看看，都有哪些成功应用案例，在经过一番慎重思考之后，再决定是否采用。我认为目前Golang正处于这个发展阶段。
- 晚期大众和早期大众在乎的东西大致相同。但晚期大众缺乏判断技术

或产品是否可顺利运作的能力，因此他们会等到相关标准确立，生态完善时才会决定是否采用。等到Golang的整个社区生态完善之时，这部分人才会开始考虑使用。

- 落后者对新技术没有任何兴趣，只是在迫不得已时，他们才会采用。

从上面的介绍中可以看出，创新者和早期采用者比较容易接受某个新技术或产品，而早期大众由于在选择时比较谨慎，喜欢参考其他采用者的使用经验。所以在另外一本介绍技术采用生命周期的书中，作者提到说早期采用者和早期大众之间存在一个巨大鸿沟。能否顺利跨越鸿沟，决定着这项技术 / 产品的成败。如下图所示（图片来自 Stockfeels）。

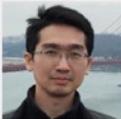
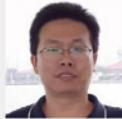


到这里，我想我已经回答了文章开头抛出的问题。我们的使命是促进软件开发领域知识与创新的传播，我们关注新技术能否被用户成功采用，我们需要帮助新技术顺利跨越应用鸿沟，我们需要让早期大众看到更多的应用案例。所以在 InfoQ 的网站、会议上，我们尽力输出新技术的最佳实践和成功案例，一切的一切，都是希望能够推动技术的发展。

最后，用我们内容价值观中经常提到的一句话来收尾：We help software development evolve faster。

海量技术干货汇集 与大咖讲师 距离



- | | | |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <p>《Maglev: A Fast and Reliable Software Network Load Balancer》
Cheng Yi
Google 高级工程师, 网络负载均衡技术 Maglev 主要作者</p> |  <p>《深入理解 Apache Beam》
Amit Sela
PayPal 架构师, Apache Beam 贡献者, PMC 成员</p> |  <p>《Mesos, 数据中心操作系统的核心》
俞捷
Mesosphere 架构师, Apache Mesos 贡献者、PMC 成员</p> |
|  <p>《聊聊开发工具的云端化》
赵扶摇
Lambda Lab 联合创始人, 前 Google 工程师</p> |  <p>《走进音视频测试体系》
罗必达
腾讯音视频实验室质量平台组组长</p> |  <p>《趣店（前趣分期）风控业务那些事》
黄浩天
趣店集团（前趣分期）架构师 & 风控技术负责人</p> |
|  <p>《人人车供应链系统技术架构演进》
路胜华
人人车业务平台部架构师 & 供应链技术负责人</p> |  <p>《大数据与流处理》
王峰 (莫问)
阿里搜索事业部资深技术专家, 数据基础设施团队负责人</p> |  <p>《技术管理的思考和实践——技术团队如何边打仗边成长》
姜华阳
美团点评销售和运营系统技术负责人</p> |
|  <p>《精益创新组合决策——更科学地决策你的投资》
姚安峰
ThoughtWorks 国内咨询首席顾问</p> |  <p>《智能运维里的时间序列：预测、异常检测和根源分析》
赵宇辰
AppDynamics 首席数据科学家</p> |  <p>《高速发展业务的架构应对实践》
陈霖
百度外卖基础架构部资深架构师</p> |
|  <p>《OCTO：千亿规模下的服务治理挑战与实践》
张熙
美团点评基础架构中心高级技术专家</p> |  <p>《百度外卖从 IDC 到云端服务迁移历程》
赵晓燕
百度外卖研发中心运维部运维经理</p> |  <p>《产品与运营分离的产品架构——网易教育的实践》
孙志岗
网易教育事业部战略总监</p> |
|  <p>《菜鸟末端业务技术架构治理实践》
章天锋 (大通)
阿里巴巴资深技术专家</p> |  <p>《腾讯 GaiaStack 容器技术深度探索》
洪志国
腾讯高级工程师</p> |  <p>《基于 Mesos 搭建 PaaS 平台你可能需要修的路》
杨成伟
爱奇艺助理研究员</p> |

大会官网: www.qconbeijing.com
议题提交: qcon@cn.infoq.com
扫码关注大会官网, 获取更多大会信息



购票咨询请联系售票经理Hanna
联系电话: 010-64738142
电子邮箱: hanna@infoq.com



8折 优惠报名中, 截至2017年3月12日
团购享受更多优惠

[北京站]
2017年4月16-18日
北京·国家会议中心

谷歌新发布的分布式数据库服务，是要打破 CAP 定理了吗？

作者 登州知府



2月14日，Google 宣布推出 Cloud Spanner 云端数据库服务的 Beta 版。Cloud Spanner 是构建在 Google Cloud Platform (GCP) 平台上的全球级分布式关系型数据库服务，主要为 OLTP 场景的核心业务应用提供服务。不同于 Bigtable、Cloud SQL 和 Cloud Datastore，此次 Google 发布的 Cloud Spanner 打破了传统关系型数据库与 NoSQL 数据库之间的壁垒，让开发者可以使用到兼具二者优点的新型数据库：支持 ACID 事务及 SQL 语义，同时具备水平扩展和跨数据中心高可用。

1. 什么是 Cloud Spanner？

Cloud Spanner 提供（跨区域 / 跨数据中心）分布式关系型数据库服务，

即所谓 NewSQL 数据库服务。

- 分布式、横向扩展（NoSQL 数据库）。
- 关系语义：Schema, ACID 事务和 SQL 查询（传统关系型数据库）。
- Cloud Spanner^[1]可以横向扩展到跨区域、跨数据中心的几百个甚至几千个节点，同时保证全局强一致性的事务。横向扩展使得 Cloud Spanner 服务既是高可用的（一个实例失效，还有其他实例），又具备高吞吐能力（多实例并行处理）。

注 1：Cloud Spanner 只支持 SQL 查询，即支持读操作。写操作是自定义的 API^[2]。

2. 这算是打破 CAP 定理了吗？

且慢，这是说 Cloud Spanner 同时提供了强一致性和高可用性吗？CAP 定理指出一个分布式系统，下列三个特性只能同时实现两个：

- 一致性（Consistency）：分布式共享的数据只能有一个值；
- 可用性（Availability）：读写操作都是 100% 可用；
- 分区（Partition）：能够容忍网络分区。

在广域网环境中，通常认为网络分区是不可避免的，分布式系统只能是一个 CP 系统或 AP 系统。为什么 Cloud Spanner 能够实现一个 CA 系统呢？

Eric Brewer 指出^[3, 4]，严格地说，Cloud Spanner 并非一个 CA 系统，但从实际效果看，“仿佛就是”一个 CA 系统。

首先，Cloud Spanner 确实会发生网络分区，此时它会选择 C 而不保证 A，因此，理论层面上，Cloud Spanner 实际上是一个 CP 系统。

其次，Cloud Spanner 实际上能够提供 5 个 9 以上的可用性，这对于大多数应用来说，足以视为高可用。

注 2：这个可用性是根据 Google 及已有客户使用 Spanner 服务的实际情况计算的。不能保证所有的应用都能达到这个程度的可用性。

3. 所谓的高可用性，究竟是指什么？

现在的问题是：一个跨区域、跨数据中心的 CP 式分布式系统，高可用的

定义是什么？如何保证高可用性？

可用性的定义：确定停用时间区间，观察一段时间内服务的停用情况。如果在某个时间点服务停用，只有当停用时间超过时间区间，才会真正被算为服务停用。如果还没到一个时间区间就已经恢复服务，则不算服务停用。

举个例子，假设停用时间区间是 10 分钟，观察 1000 分钟的服务。共发生 2 次服务停用。第一次，5 分钟后服务恢复；第二次，12 分钟后服务恢复正常。只有第二次才会被视为发生了服务停用，所有该服务的可用性是 $1 - \frac{1}{100} = 0.99$ 。

什么叫高可用？首先，服务实际上具有很高的可用性，用户的应用程序中无需包含专门处理服务停用的代码。像 Google 内部使用的 Spanner 服务，虽然达不到 100% 可用性，但是远超 5 个 9 的可用性。从 Google 和客户实际使用的情况看，可以视为一个高可用的系统。

注 3：Cloud Spanner 与 Spanner 不同，可用性估计会低一些，因此号称是 5 个 9 的可用性。但是计算可用性时，没说停用时间区间是多少。

其次，引发服务故障（包括停用）的原因很多。这里讨论 Cloud Spanner 可用性的前提是客户端工作正常，能够发送请求，因此能够发现服务是否停用。显然，仅考虑这种情况计算得到的服务可用性，比 Spanner 真正的可用性要高很多。

最后，由于 Spanner 采用特殊的网络技术，在实践中，几乎不会发生网络分区。因此，不考虑因为网络分区导致的服务不可用。

总而言之，说 Cloud Spanner 高可用是指：

- 实践中，系统处于 CA 状态的概率非常高，用户不需要编写处理服务停用的代码；
- 如果发生了服务停用，原因是网络分区的可能性也非常非常小。

4. Cloud Spanner 高可用性究竟是如何实现的？

Google 打造了私有的全球网络。以 Spanner 服务为例，所有的路由器和链接（除了客户端与服务的链接）都是由 Google 控制的。每个数据中心都至少有 3 条独立光纤连接到私有全球网，保证任何两个数据中心之间都有多条

物理路径。在同一个数据中心内，网络设备和路径也是冗余的。因此，不会出现因为网络路径中断引发的网络分区。

如果配置不当或者软件升级，导致多条路径同时中断，就会引发网络分区。因此，采用部分升级的策略，即每次升级影响到部分路径或副本，确保无虞之后，再全面升级。

当然，除了网络分区，还有很多原因能够引发服务停用。实际上，在所有引发服务事故（包括停用）的原因中，与网络相关的仅占 8% 左右。既然 Google 用私有全球网络解决了网络分区的问题，那么剩下那些问题如何解决？答案很简单：他们有一只厉害的运维队伍。

注 4：讨论了半天，实质上是告诫大家别想着达到 Google 的高可用性了。你有钱打造私有全球网吗？你有高效的运维队伍吗？没有，就购买 Google 托管的 Cloud Spanner 服务吧。

一个现实的问题是：即使采用上述手段，使得网络分区的可能性大大降低，接近于零。毕竟还不是零，万一发生了网络分区呢？答案是：如果发生网络分区，Cloud Spanner 将保证强一致性，不管可用性。至于如何保证强一致性，那是另外一个需要详细讨论的问题了。

参考资料

- [1] Introducing Cloud Spanner: a global database service for mission-critical applications
- [2] Cockroach Labs CTO 谈 Cloud Spanner
- [3] Inside Cloud Spanner and the CAP Theorem
- [4] Spanner, TrueTime and the CAP Theorem

Spark 上的深度学习框架再添新兵：Yahoo 开源 TensorFlowOnSpark

作者 刘志勇



前言

Yahoo Big ML 团队宣布开源 [TensorFlowOnSpark](#)，他们用来在大数据集群的分布式深度学习最新的开源框架。

Yahoo Big ML 团队成员 Lee Yang、Jun Shi、Bobbie Chern 和 Andy Feng 日前合著了一篇文章，详细介绍 了他们开源的 TensorFlowOnSpark 的方方面面。

Yahoo 开源的 TensorFlowOnSpark 使 Google 发起的 TensorFlow 深度学习开源框架与 Apache Spark 集群中的数据集兼容，一些组织为了处理大量不同类型的数据而进行维护，对他们来说无疑是个好消息。

Yahoo 开源 TensorFlowOnSpark 采用了 Apache 2.0 协议许可，并在 GitHub 上发布。

深度学习通常涉及大量数据进行人工神经网络训练，比如说照片，然后指

导神经网络对新数据做出最佳猜测。深度学习在很多公司非常热门。

差不多就在一年前，Yahoo 开源 [CaffeOnSpark](#)，为 Caffe 开源深度学习框架提供了 Spark 支持。而今天，Yahoo 正在做同样的工作，但这一次，带来了不同的框架：TensorFlowOnSpark。

该团队评估了 SparkNet 和 TensorFrame 等选择，但最终，他们决定建立自己的框架。他们的软件使用 Spark 工具，如 SparkSQL、Mlib 和 Python notebook 连接到 Spark 集群，但它也将和 Hadoop 合作。

Yahoo 表示，把 TensorFlow 程序移植到 TensorFlowOnSpark 相对方便，并经过反公司内部的反复验证。

InfoQ 翻译并整理本文。

正文

深度学习（DL）在最近几年快马加鞭地发展。在 Yahoo，我们发现，为了从海量数据中获得洞察力，需要部署分布式深度学习。现有的 DL 框架通常需要为深度学习设置单独的集群，迫使我们为机器学习流程创建多个程序（见图 1）。拥有独立的集群需要我们在它们之间传递大型数据集，从而引起不必要的系统复杂性和端到端的学习延迟。

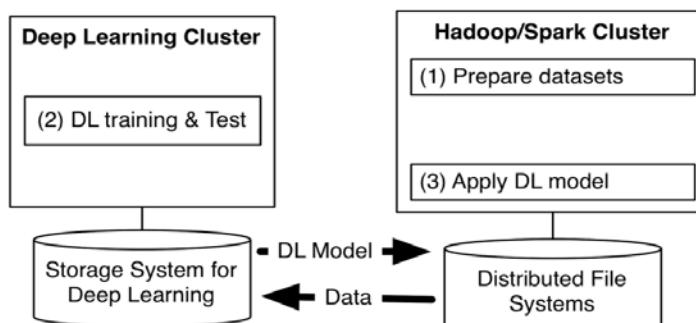


Figure 1: ML Pipeline with multiple programs on separated clusters

去年我们通过开发和发布 CaffeOnSpark 来解决 scaleout 问题，我们的开源框架，支持在相同的 Spark 和 Hadoop 集群进行分布式深度学习和大数据处理。我们在 Yahoo 使用 CaffeOnSpark 来改善我们的 [NSFW 图像检测](#)，比如自动从现场直播等自动识别电竞比赛等。借助社区的宝贵意见和贡献，CaffeOnSpark

已经升级，支持 LSTM，带有一个新的数据层，可用于训练和测试交错，还有一个 Python API 以及在 Docker 容器上的部署。对我们来说，这些极大提升了用户体验。但对于那些使用深层学习框架 TensorFlow 的用户怎么办呢？于是我们仿效之前的做法，开发了 TensorFlowOnSpark。

在 TensorFlow 的首次发布后，谷歌在 2016 年 4 月发布了增强的 TensorFlow 与分布式深度学习功能。在 2016 年 10 月，TensorFlow 宣布支持 HDFS。然而，在 Google 云之外，用户仍然需要一个专用于 TensorFlow 应用程序的集群。TensorFlow 程序不能部署在现有的大数据集群上，从而增加了那些希望大规模利用这种技术的成本和延迟。

为了打破这个限制，一些社区项目将 TensorFlow 连接到 Spark 集群。SparkNet 在 Spark 执行器添加了运行 TensorFlow 网络的能力。DataBricks 提出 TensorFrame，用来使用 TensorFlow 程序操纵 Apache Spark 的 DataFrames（数据帧）。虽然这些方法是在正确的方向迈出了一步，但我们检查其代码后，发现我们无法使多个 TensorFlow 进程直接相互通信，我们也无法实现异步分布式学习，我们还必须花费大量精力来迁移现有的 TensorFlow 程序。

TensorFlowOnSpark

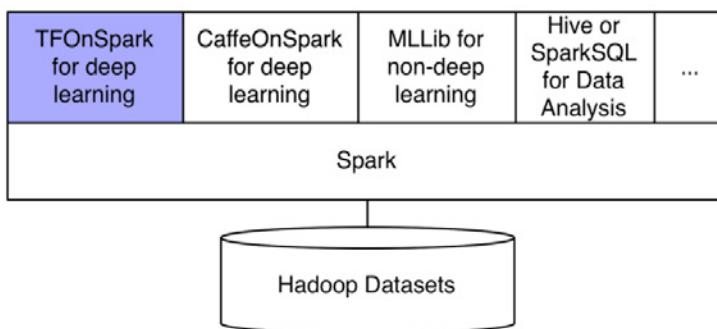


Figure 2: TensorFlowOnSpark for deep learning on Spark clusters

我们的新框架 TensorFlowOnSpark (TFoS)，支持 TensorFlow 在 Spark 和 Hadoop 集群上分布式执行。如上图 2 所示，TensorFlowOnSpark 被设计为与 SparkSQL、MLlib 和其他 Spark 库一起在一个单独流水线或程序（如 Python

notebook) 中运行。

TensorFlowOnSpark 支持所有类型的 TensorFlow 程序，可以实现异步和同步的训练和推理。它支持模型并行性和数据的并行处理，以及 TensorFlow 工具（如 Spark 集群上的 TensorBoard）。

任何 TensorFlow 程序都可以轻松地修改为在 TensorFlowOnSpark 上运行。通常情况下，需要改变的 Python 代码少于 10 行。许多 Yahoo 平台使用 TensorFlow 的开发人员很容易迁移 TensorFlow 程序，以便在 TensorFlowOnSpark 上执行。

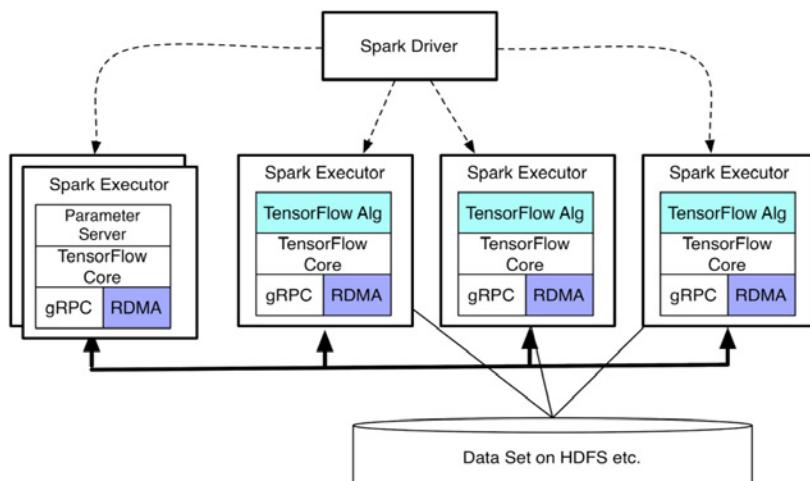


Figure 3: TensorFlowOnSpark system architecture

TensorFlowOnSpark 支持 TensorFlow 进程（计算节点和参数服务节点）之间的直接张量通信。过程到过程的直接通信机制使 TensorFlowOnSpark 程序能够在增加的机器上很轻松的进行扩展。如图 3 所示，TensorFlowOnSpark 不涉及张量通信中的 Spark 驱动程序，因此实现了与独立 TensorFlow 集群类似的可扩展性。

TensorFlowOnSpark 提供两种不同的模式来提取训练和推理数据。

1. **TensorFlow QueueRunners:** TensorFlowOnSpark 利用 TensorFlow 的 file readers 和 [QueueRunners](#) 直接从 HDFS 文件中读取数据。Spark 不涉及访问数据。

2. Spark Feeding : Spark RDD数据被传输到每个Spark执行器里，随后的数据将通过feed_dict传入TensorFlow图。

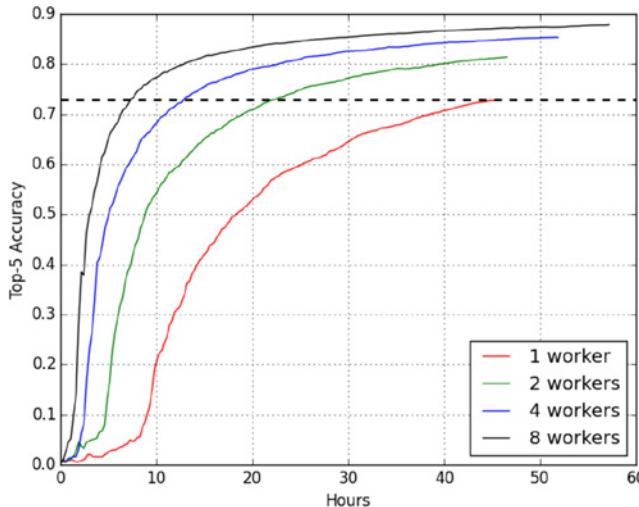


Figure 4: TFoS training of Inception networks

图 4 说明初始图像分类中同时进行的分布式训练如何使用 TFoS 中通过 QueueRunners 的一个简单设置进行扩展：每个节点一个 GPU、一个读入以及批处理为 32。四个 TFoS 工作同时进行，训练 100,000 步。两天后，当这些工作完成时，这些工作的前 5 个准确度分别为 0.730、0.814、0.854 和 0.879。精确度达到 0.730 的单计算节点工作需要 46 小时，对于双计算节点则需要 22.5 小时，4 计算节点需要 13 小时，8 计算节点工需要 7.5 小时。Tfоs 因此实现了接近模型训练的近线性可扩展性。这是非常令人鼓舞的，虽然 Tfоs 可扩展性会因不同的型号和超级数而有所不同。

分布式 TensorFlow 的 RDMA

在 Yahoo 的 Hadoop 集群上，GPU 节点通过以太网和 Infiniband 连接。Infiniband 提供更快的连接，并支持通过 RDMA 直接访问其他服务器的内存。然而，当前 TensorFlow 版本仅支持使用 gRPC} 通过以太网的分布式学习。为了加快分布式学习，我们增强了 TensorFlow C ++ 层，以支持 Infiniband 上的 RDMA。

为结合我们发布的 Tfоs，我们除了默认的“GRPC”协议外，还引入了新

的 TensorFlow 服务器协议。任何分布式 TensorFlow 程序可以通过指定利用 `tf.train.ServerDef()` 或 `tf.train.Server()` 中的 `protocol="grpc_rdma"` 来使用增强版的 TensorFlow。

使用此新协议，就需要创建 RDMA 汇集管理器以确保张量直接写入远程服务器的内存。我们最小化张量缓冲区的创建：Tensor 缓冲区在开始时分配一次，然后在一个 TensorFlow 作业的所有训练步骤中重复使用。从我们早期的实验与大型模型（如 [VGG-19 网络](#)）来看，业已证明，与现有 GRPC 相比，我们的 TDMA 实现在训练时间上显著加速了。

由于支持 RDMA 是一个高度要求的能力（见 TensorFlow issue [# 2916](#)），我们决定把现有的实现版本作为一个 alpha 版向 TensorFlow 社区开放。在接下来的几周内，我们将进一步优化 RDMA 实现，并分享一些详细的基准测试结果。

简单的 CLI 和 API

TFoS 程序由标准的 Apache Spark 命令 `spark-submit` 来启动。如下图所示，用户可以在 CLI 中指定 Spark 执行器的数目，每个执行器的 GPU 数量和参数服务器的数目。用户还可以指定是否要使用 TensorBoard (`-tensorboard`) 和 / 或 RDMA (`-rdma`)。

```
spark-submit -master ${MASTER} \
${TFoS_HOME}/examples/slim/train_image_classifier.py \
-model_name inception_v3 \
-train_dir hdfs://default/slim_train \
-dataset_dir hdfs://default/data/imagenet \
-dataset_name imagenet \
-dataset_split_name train \
-cluster_size ${NUM_EXEC} \
-num_gpus ${NUM_GPU} \
-num_ps_tasks ${NUM_PS} \
-sync_replicas \
-replicas_to_aggregate ${NUM_WORKERS} \
```

```
-tensorboard \
-rdma
```

TFoS 提供了一个高层次的 Python API (在我们示例 Python [notebook 说明](#)) :

```
TFCluster.reserve() ... construct a TensorFlow cluster from
Spark executors
TFCluster.start() ... launch Tensorflow program on the executors
TFCluster.train() or TFCluster.inference() ... feed RDD data to
TensorFlow processes
TFCluster.shutdown() ... shutdown Tensorflow execution on
executors
```

开放源码

TensorFlowOnSpark、TensorFlow 的 RDMA 增强包、多个[示例程序](#)（包括 MNIST, cifar10, 创建以来, VGG）来说明 TensorFlow 方案 TensorFlowOnSpark，并充分利用 RDMA 的简单转换过程。亚马逊机器映像也可对 AWS EC2 应用 TensorFlowOnSpark。



专访 RocketMQ 联合创始人：项目思路、技术细节和未来规划

作者 木环

编者按

这些年开源氛围越来越好，各大 IT 公司都纷纷将一些自研代码开源出来。2012 年，阿里巴巴开源其自研的第三代分布式消息中间件——RocketMQ。经过几年的技术打磨，阿里称基于 RocketMQ 技术，目前双十一当天消息容量可达到万亿级。

2016 年 11 月，阿里将 RocketMQ 捐献给 Apache 软件基金会，正式成为孵化项目。阿里称会将其打造成顶级项目。这是阿里迈出的一大步，因为加入到开源软件基金会需要经过评审方的考核与观察。坦率而言，业界还对国人的代码开源参与度仍保持着刻板印象；而 Apache 基金会中的 342 个项目中，暂时还只有 Kylin、CarbonData、Eagle 和 RocketMQ 共计四个中国技术人主导的项目。

2017 年 2 月 20 日，RocketMQ 正式发布 4.0 版本，专家称新版本适用于电商领域，金融领域，大数据领域，兼有物联网领域的编程模型。

RocketMQ 项目，究竟用怎样的技术内涵？缘何赢得了基金会的初步认可？入驻基金会可以给技术圈哪些启示？InfoQ 带着这样的疑问对两位项目联合创始人进行了专访，内容整理如下。

RocketMQ 的由来

谈起 RocketMQ 的亮点，那不得不先提一下阿里巴巴消息引擎的演进史。阿里中间件消息引擎发展到今日，前前后后经历了三代演进。

第一代，推模式，数据存储采用关系型数据库。在这种模式下，消息具有很低的延迟特性，并且很容易支持分布式事务。尤其在阿里淘宝这种高频交易场景中，具有非常广泛地应用。典型代表包括 Notify、Napoli。

第二代，拉模式，自研的专有消息存储。在日志处理方面能够媲美 Kafka 的吞吐性能，但考虑到淘宝的应用场景，尤其是其交易链路的高可靠需求，消息引擎并没有一味的追求吞吐，而是将稳定可靠放在首位。因为采用了长连接拉模式，在消息的实时方面丝毫不逊推模式。典型代表 MetaQ。

第三代，以拉模式为主，兼有推模式的高性能、低延迟消息引擎 RocketMQ，在二代功能特性的基础上，为电商金融领域添加了可靠重试、基于文件存储的分布式事务等特性，并做了大量优化。从 2012 年开始，经历了历次双十一核心交易链路检验。目前已经捐赠给 Apache 基金会。时至今日，RocketMQ 很好的服务了阿里集团大大小小上千个应用，在双 11 当天，更有不可思议的万亿级消息流转，为集团大中台的稳定发挥了举足轻重的作用。

不难看出，RocketMQ 其实是伴随着阿里巴巴整个生态的成长，逐渐衍生出来的高性能、低延迟能够同时满足电商领域和金融领域的极尽苛刻场景的消息中间件。

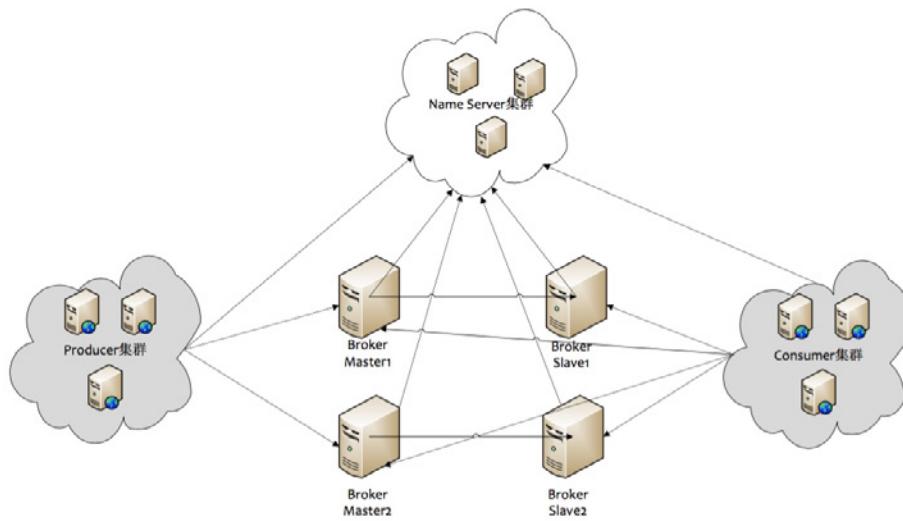
RocketMQ 的技术概览

请配图同时对 RocketMQ 的架构进行说明；请对消息的产生、传输和消费流程进行说明，最好也可以配图。

在我看来，它最大的创新点在于能够通过精巧的横向、纵向扩展，不断满足与日俱增的海量消息在高吞吐、高可靠、低延迟方面的要求。

目前 RocketMQ 主要由 NameServer、Broker、Producer 以及 Consumer 四部分构成，如下图所示。

所有的集群都具有水平扩展能力，无单点障碍。



NameServer 以轻量级的方式提供服务发现和路由功能，每个 NameServer 存有全量的路由信息，提供对等的读写服务，支持快速扩缩容。

Broker 负责消息存储，以 Topic 为纬度支持轻量级的队列，单机可以支撑上万队列规模，支持消息推拉模型，具备多副本容错机制（2 副本或 3 副本）、强大的削峰填谷以及上亿级消息堆积能力，同时可严格保证消息的有序性。除此之外，Broker 还提供了同城异地容灾能力，丰富的 Metrics 统计以及告警机制。这些都是传统消息系统无法比拟的。

Producer 由用户进行分布式部署，消息由 Producer 通过多种负载均衡模式发送到 Broker 集群，发送低延时，支持快速失败。

Consumer 也由用户部署，支持 PUSH 和 PULL 两种消费模式，支持集群消费和广播消息，提供实时的消息订阅机制，满足大多数消费场景。

经历双十一洗礼的英雄

在备战 2016 年双十一时，团队重点做了两件事情，优化慢请求与统一存储引擎。

优化慢请求：这里主要是解决在海量高并发场景下降低慢请求对整个集群带来的抖动，毛刺问题。这是一个极具挑战的技术活，团队同学经过长达 1 个多月的跟进调优，从双十一的复盘情况来看，99.996% 的延迟落在了 10ms 以内，

而 99.6% 的延迟在 1ms 以内。优化主要集中在 RocketMQ 存储层算法优化、JVM 与操作系统调优。更多的细节大家可以参考我们之前写的电子书章节《万亿级数据洪峰下的分布式消息引擎》。

再来看看统一存储引擎：主要解决的消息引擎的高可用，成本问题。在多代消息引擎共存的前提下，我们对 Notify 的存储模块进行了全面移植与替换。

这样下来，阿里巴巴内部的消息中间件就全面拥抱了 RocketMQ 的低延迟存储引擎。基于上述积极的技术准备，在 16 年双十一期间，阿里集团大约有 1.2 万亿级的消息流转总量，几乎是 15 年双十一大促的 2 倍。峰值期间，消息生产的吞吐在 2000 w/s 左右，消息消费的吞吐也近乎 1500 w/s 的量级。整个大促下来，用我们内部的话来说，如丝般顺滑。

RocketMQ VS 其他几个消息中间件

请从技术构思、实践表现和适用场景三个大方向对 RocketMQ、RabbitMQ、Kafka、ActiveMQ 和 ZeroMQ 进行对比？除了技术上的较量，可否对这些中间件背后的社区运营、商业案例应用，进行对比呢？

1、是不是CS架构？

如果需要做同类产品之间的横向对比，我们优先拿下 ZeroMQ，ZeroMQ 正如其名 0MQ，它更像是一个嵌入式的网络类库，一个专注于 transports 层的通讯组件，而不是传统意义上的 CS 架构的 MQ。

2、实现的哪种规范 / 协议？

接下来我们来看看 RabbitMQ、ActiveMQ、Kafka 和 RocketMQ 之间的一些对比，从设计思路上来看，RabbitMQ 是 AMQP 规范的参考实现，AMQP 是一个线路层协议，面面俱到，很系统，也稍显复杂。目前 RabbitMQ 已经成为 OpenStack IaaS 平台首选的消息服务，其背后的支持力度不言而喻。

ActiveMQ 最初主要的开发者在 LogicBlaze，现在主要开发红帽，是 JMS 规范的参考实现，也是 Apache 旗下的老牌消息服务引擎。JMS 虽说是一个 API 级别的协议，但其内部还是定义了一些实现约束，不过缺少多语言支撑。ActiveMQ 的生态堪称丰富多彩，在该 Apache 顶级项目下，拥有不少子项目，

包括由 HornetMQ 演变而来的 Artemis，基于 Scala 号称下一代 AMQ 的 Apollo 等。

3、适用何类场景？

而 Kafka 最初被设计用来做日志处理，是一个不折不扣的大数据通道，追求高吞吐，存在丢消息的可能。其背后的研发团队也围绕着 Kafka 进行了商业包装，目前在一些中小型公司被广泛使用，国内也有不少忠实的拥捧着。

RocketMQ 天生为金融互联网领域而生，追求高可靠、高可用、高并发、低延迟，是一个阿里巴巴由内而外成功孕育的典范，除了阿里集团上千个应用外，根据我们不完全统计，国内至少有上百家单位、科研教育机构在使用。关于这几个 MQ 产品更详细的特性对比，可以参考我们官网上的说明。

RocketMQ vs. ActiveMQ vs. Kafka

Messaging Product	Client SDK	Protocol and Specification	Order Message	Message Filter	Server Triggered Redelivery	Persistent Message	Retrospective Consumers	Message Priority	High Availability and Failover	Message Track	Configuration	Management and Operation Tools
ActiveMQ	Java, .NET, C++ etc.	Push model, support OpenWire, STOMP, AMQP, MQTT, JMS	Exclusive Consumer or Exclusive Queues can ensure ordering	Supported	Not Supported	Supports very fast persistence using JDBC along with a high performance journal, such as leveldB, kahaDB	Supported	Supported	Supported, depending on storage, if using kahadb it requires a ZooKeeper server	Not Supported	The default configuration is low level, user need to optimize the configuration parameters	Supported
Kafka	Java, Scala etc.	Pull model, support TCP	Ensure ordering of messages within a partition	Supported, you can use Kafka Streams to filter messages	Not Supported	High performance file storage	Supported offset indicate	Not Supported	Supported, requires a ZooKeeper server	Not Supported	Kafka uses key-value pairs format for configuration. These values can be supplied either from a file or programmatically.	Supported, use terminal command to expose core metrics
RocketMQ	Java, .NET, C++	Pull model, support TCP, JMS	Ensure strict ordering of messages, have no hot spot problem, and can scale out gracefully	Supported, you can even upload yourself custom-built filter code snippets	Supported	High performance and low latency file storage	Supported timestamp and offset 2 indicates	Not Supported	Supported, Master-Slave model, without another kit	Supported	Work out of box, user only need to pay attention to a few configurations	Supported, rich web and terminal command to expose core metrics

三项技术发力点

1. 消息的顺序

不可否认，顺序消息是 RocketMQ 功能特性上的一个卖点。目前我们做到了全局保序。需要重点说一下，这里的全局是有前提，针对某个唯一标识（能

够被 Hash 成唯一标识），比方说大卖家账号，某类产品的订单等。其技术实现原理也相对比较简单，**保证对通道的单一实例操作**，如单进程、单线程写，单进程、线程读，像 ActiveMQ 的 Exclusive Consumer 也是类似的实现。

不难看出，这种实现方式实际是在吞吐上做出了一定牺牲。另外也带来了另外一个问题 - 热点。如双十一当天，如果使用简单的散列策略，像短期内就交易过亿的天猫商家的消息会发送到一个通道里面，造成单通道，甚至单机的热点问题在最新的 RocketMQ 版本里，我们将会改进目前的实现，借此改善因为顺序导致的单通道热点问题，这个特性预计今年中旬会发布。

2. 消息的去重

消息领域有一个对消息投递的 QoS 定义，分为：最多一次（At most once），至少一次（At least once），仅一次（Exactly once）。

几乎所有的 MQ 产品都声称自己做到了 At least once。既然是至少一次，那避免不了消息重复，尤其是在分布式网络环境下，而这个缺憾归根结底也可以看做是 TCP 协议的一部分，如失败重传。业务上往往对消息重复又很敏感，RocketMQ 目前的版本是不支持去重的，我们通常建议用户通过外置全局存储自己做判重处理。在下一代的特性规划里，我们会内置解决方案。先说下业界通用做法，像 Artemis，IronMQ 等，通过**在服务端全局存储判重**。这是一个 I/O 敏感的操作，为服务端带来一定的负载。而 RocketMQ 则希望通过采取二次判重策略，有效降低服务端 I/O。

3. 分布式的挑战

首先明确分布式系统这个概念：**分布式系统是由一系列分散自治组件通过互联网并行并发协作，从而组成的一个 coherent 软件系统**。它具备资源共享，并行并发，可靠容错，透明开放等特性。像 CAP，BASE，Paxos，事务等一起构成了分布式基础理论。

这里我们再来重温下 CAP 理论：CAP 分别代表一致性（Consistency），可用性（Availability），分区容忍性（Partition tolerance）。一致性，Eric Brewer（CAP 理论提出者）用一个服务要么被执行，要么不被执行来定义

(原文: A service that is consistent operates fully or not at all)。请注意，这里的一致性是有别于数据库 ACID 属性中的 C，数据库层面的 C 指的是数据的操作不能破坏数据之间的完整性约束，如外键约束。**在分布式环境中，可以把 C 简单理解为多节点看到的是数据单一或者同一副本。**可用性，意味着服务是可用的（原文: the service is available (to operate fully or not as above)）。可用性又可以细分为写可用和读可用。在分布式环境中，往往指的是系统在确定时间内可返回读写操作结果，也即读写均可用。分区容错性，除了整个网络故障外（如光纤被掘断），其它故障（如丢包、乱序、抖动、甚至是网络分区节点 crash）都不能导致整个系统无法正确响应。（原文: No set of failures less than total network failure is allowed to cause the system to respond incorrectly.）

CAP 理论可以看做是探索适合不同应用的一致性与可用性平衡问题。

- 没有分区的情况：可以同时满足C与A，以及完整的ACID事务支持。可以选择牺牲一定的C，获得更好的性能与扩展性。
- 分区的情况：选择A（集中关注分区的恢复），需要有分区开始前、进行中、恢复后的处理策略，应用合适的补偿处理机制。像RocketMQ这样的分布式消息引擎，更多的追求AP。再强的系统也一定有容量底线，足够的容量是可用性的有效前提。通常情况下，会通过降级、限流、熔断机制来保障洪峰下的可用性。具体的技术细节可以参看电子书第一章。

另外，考虑到在金融高频交易典型场景，我们也为 RocketMQ 设计了 CP 机制，在满足分布式系统的分区容错性的前提下，牺牲系统可用性来保证数据的一致性。而技术实现上，则基于 Zab 一致性协议，利用分布式锁和通知机制，以此来保障多副本数据的一致性。

开源捐赠和社区运营

目前国内外有很多公司会把一些通用问题的解决方案，尤其是那些久经考验、愈久弥坚的产品开源出来，以期望在品牌宣传、人才引进方面有所建树。

把 RocketMQ 开源出来，甚至捐赠给 Apache，内部也是经过了深思熟虑，层层审批与讨论，期望能够在生态化、规范化、国际化、商业化方面深耕细作。

开源捐赠的想法实际上始于 2014 年。当时，我们甄选了几位 Apache 社区权威人士，遗憾的是反复沟通不断修改草案之后突然间失去了联系。2015 年，我们有幸结识了 Kylin Principal Architect 蒋旭和 VP Luke 以及 RedHat Principal Software Engineer 姜宁，请教了一些 Apache 禁忌事项，重新活跃起来了捐赠进程。接下来，最重要的是征集 champion 候选人，很开心的是 ActiveMQ VP Bruce 爽快地接收了我们的邀请，经过前前后后接近 100 封邮件来往，我们终于正式开启了 Apache 之旅。捐赠投票是在双十一当天，我们准备充分很好地回答了评委会的犀利问题。不过，面对“中国开发者不喜欢邮件沟通”突然刁难，还要感谢社区华人的防御性声明回应。经过很多磨难，投票结果总算出来了：还不算坏 10 票赞同，带 binding (IPMC 成员的有效投票) 的 +1，无反对票，正式进入孵化期。孵化成功后有望成为国内首个互联网中间件在 Apache 上的顶级项目，成为全球继 ActiveMQ，Kafka 之后，分布式消息引擎家族中的重要成员。

接下来，我们想强调下知识产权这个对大多数工程师来说陌生的领域，尤其是专利权、著作权、商标权。在国外，每年因为这些问题导致的侵权官司不在少数。而我们在开源之初，对这块的选择、保护也是极其谨慎，包括开源许可协议的选择、授权方面，代码署名权等，这些都是很好的智力保护，也是我们产品的核心竞争力之一。尊重知识，尊重产权，才能构建一个和谐积极向上的开源氛围，打造真正的自主知识产权品牌产品。

在阿里巴巴，我们基于开源引擎的 RocketMQ，为云上用户提供了商业化版本的 Aliware MQ。两个产品都是由阿里中间件消息团队出品。商业版 Aliware MQ 在支持 TCP 、HTTP 和 MQTT 协议接入，功能方面增强了运维管控方面，生态集成的能力（包括可视化的消息轨迹、资源报表统计以及监控报警、Kafka 集成等）。它在公有云上本身具备多机房部署同城高可用容灾特性，目的是满足企业级要求。

关于社区的运营，我们采取了和 Apache 顶级项目基本相似的策略。首

先，必须立足于高质量产品本身，从版本规划开始，我们建立了里程碑讨论，Features 设计，编码自测，结对 Review，集成测试，Release 讨论，Release 公告等等一系列规范且高效的软件研发流程。其次，在社区运营层面，则有一系列与社区互动的活动，如线下 meetup、workshop、ApacheCon、不定期的编程马拉松等，吸纳新的 Contributor 和 Committer 进来。

新一代 RocketMQ，蓄势待发

最近，团队也在着手构建下一代 RocketMQ，期望构建一套厂商无关的集线路层、API 层于一体的规范，这也是第四代消息引擎最大的亮点。目前，我们联系了 Twitter、Yahoo 等公司相关技术负责人，共同起草完善这一规范，而 RocketMQ 将会是第一批率先成为参考实现的产品。我们非常期望国内的 MQ 厂商亦或是分布式爱好者能够参与进来，积极在国际开源社区代表国人发声呐喊。

另外，本周，团队刚刚发布了第四代引擎的第一个版本，该版本也是进入 Apache 社区后的首次发版。按照我们的规划，将在今年 4 月左右完成整个引擎的升级重组，非常欢迎大家的使用、反馈以及参与。

最后，更多信息可以移步 [Apache RocketMQ 官网](#)、[云栖社区](#)、[中间件官方博客](#) 以及 [阿里巴巴电子书](#)。

受访者简介

王小瑞，花名誓嘉，阿里巴巴中间件消息团队负责人，具有丰富的高可用，高可靠分布式系统构建经验，主导了阿里巴巴多次双十一消息引擎的改进优化项目，拥有多项分布式领域的专利。Apache RocketMQ 联合创始人。联系方式：vintagewang@apache.org。

冯嘉，花名鼬神，阿里巴巴中间件架构师，具有丰富的分布式软件架构、高并发网站设计、性能调优经验，拥有多项分布式领域的专利。开源爱好者，专注分布式、大数据领域，关注 Hbase/Hadoop/Spark/Flink 等大数据技术栈。目前负责阿里消息中间件生态输出、云上商业化，Apache RocketMQ 联合创始人。联系方式：vongosling@apache.org。



百亿级微信红包的高并发资金交易系统设计方案

作者 方乐明

2017年1月28日，正月初一，微信公布了用户在除夕当天收发微信红包的数量——142亿个，而其收发峰值也已达到76万每秒。百亿级别的红包，如何保障并发性能与资金安全？这给微信带来了超级挑战。面对挑战，微信红包在分析了业界“秒杀”系统解决方案的基础上，采用了SET化、请求排队串行化、双维度分库表等设计，形成了独特的高并发、资金安全系统解决方案。实践证明，该方案表现稳定，且实现了除夕夜系统零故障运行。

本文将为读者介绍百亿级别红包背后的系统高并发设计方案，包括微信红包的两大业务特点、微信红包系统的技术难点、解决高并发问题通常使用的方案，以及微信红包系统的高并发解决方案。

一、微信红包的两大业务特点

微信红包（尤其是发在微信群里的红包，即群红包）业务形态上很类似网上的普通商品“秒杀”活动。

用户在微信群里发一个红包，等同于普通商品“秒杀”活动的商品上架；微信群里的所有用户抢红包的动作，等同于“秒杀”活动中的查询库存；用户抢到红包后拆红包的动作，则对应“秒杀”活动中用户的“秒杀”动作。

不过除了上面的相同点之外，微信红包在业务形态上与普通商品“秒杀”活动相比，还具备自身的特点：

首先，微信红包业务比普通商品“秒杀”有更海量的并发要求。

微信红包用户在微信群里发一个红包，等同于在网上发布一次商品“秒杀”活动。假设同一时间有 10 万个群里的用户同时在发红包，那就相当于同一时间有 10 万个“秒杀”活动发布出去。10 万个微信群里的用户同时抢红包，将产生海量的并发请求。

其次，微信红包业务要求更严格的安全级别。

微信红包业务本质上是资金交易。微信红包是微信支付的一个商户，提供资金流转服务。

用户发红包时，相当于在微信红包这个商户上使用微信支付购买一笔“钱”，并且收货地址是微信群。当用户支付成功后，红包“发货”到微信群里，群里的用户拆开红包后，微信红包提供了将“钱”转入拆红包用户微信零钱的服务。

资金交易业务比普通商品“秒杀”活动有更高的安全级别要求。普通的商品“秒杀”商品由商户提供，库存是商户预设的，“秒杀”时可以允许存在“超卖”（即实际被抢的商品数量比计划的库存多），“少卖”（即实际被抢的商户数量比计划的库存少）的情况。但是对于微信红包，用户发 100 元的红包绝对不可以被拆出 101 元；用户发 100 元只被领取 99 元时，剩下的 1 元在 24 小时过期后要精确地退还给发红包用户，不能多也不能少。

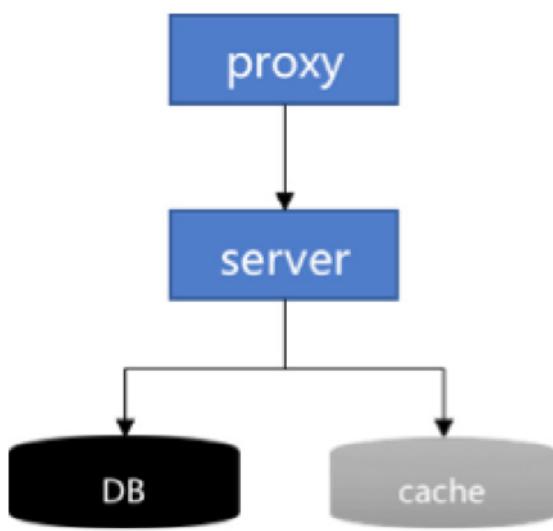
以上是微信红包业务模型上的两大特点。

二、微信红包系统的技术难点

在介绍微信红包系统的技术难点之前，先介绍下简单的、典型的商品“秒杀”系统的架构设计，如下图所示。

该系统由接入层、逻辑服务层、存储层与缓存构成。Proxy 处理请求接入，Server 承载主要的业务逻辑，Cache 用于缓存库存数量、DB 则用于数据持久化。

一个“秒杀”活动，对应 DB 中的一条库存记录。当用户进行商品“秒杀”时，系统的主要逻辑在于 DB 中库存的操作上。一般来说，对 DB 的操作流程有以下三步：



- a. 锁库存
- b. 插入“秒杀”记录
- c. 更新库存

其中，锁库存是为了避免并发请求时出现“超卖”情况。同时要求这三步操作需要在一个事务中完成（所谓的事务，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行）。

“秒杀”系统的设计难点

就在这个事务操作上。商品库存存在 DB 中记为一行，大量用户同时“秒杀”同一商品时，第一个到达 DB 的请求锁住了这行库存记录。在第一个事务完成提交之前这个锁一直被第一个请求占用，后面的所有请求需要排队等待。同时参与“秒杀”的用户越多，并发进 DB 的请求越多，请求排队越严重。因此，并发请求抢锁，是典型的商品“秒杀”系统的设计难点。

微信红包业务相比普通商品“秒杀”活动，具有海量并发、高安全级别要求的特点。在微信红包系统的设计上，除了并发请求抢锁之外，还有以下两个突出难点：

首先，事务级操作量级大。上文介绍微信红包业务特点时提到，普遍情况下同时会有数以万计的微信群在发红包。这个业务特点映射到微信红包系统设计上，就是有数以万计的“并发请求抢锁”同时在进行。这使得 DB 的压力比普通单个商品“库存”被锁要大很多倍。

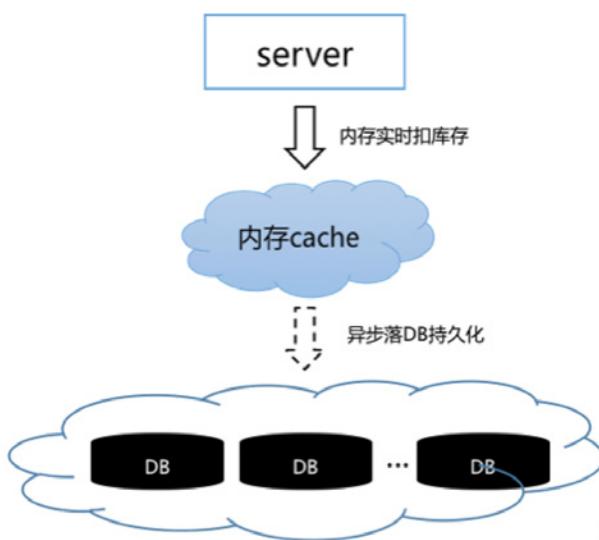
其次，事务性要求严格。微信红包系统本质上是一个资金交易系统，相比普通商品“秒杀”系统有更高的事务级别要求。

三、解决高并发问题通常使用的方案

普通商品“秒杀”活动系统，解决高并发问题的方案，大体有以下几种：

方案一，使用内存操作替代实时的 DB 事务操作。

如下图所示，将“实时扣库存”的行为上移到内存 Cache 中操作，内存 Cache 操作成功直接给 Server 返回成功，然后异步落 DB 持久化。



这个方案的优点是用内存操作替代磁盘操作，提高了并发性能。

但是缺点也很明显，在内存操作成功但 DB 持久化失败，或者内存 Cache 故障的情况下，DB 持久化会丢数据，不适合微信红包这种资金交易系统。

方案二，使用乐观锁替代悲观锁。

所谓悲观锁，是关系数据库管理系统里的一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作对某行数据应用了锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。对应于上文分析中的“并发请求抢锁”行为。

所谓乐观锁，它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回滚。

商品“秒杀”系统中，乐观锁的具体应用方法，是在 DB 的“库存”记录中维护一个版本号。在更新“库存”的操作进行前，先去 DB 获取当前版本号。在更新库存的事务提交时，检查该版本号是否已被其他事务修改。如果版本没被修改，则提交事务，且版本号加 1；如果版本号已经被其他事务修改，则回滚事务，并给上层报错。

这个方案解决了“并发请求抢锁”的问题，可以提高 DB 的并发处理能力。

但是如果应用于微信红包系统，则会存在下面三个问题：

如果拆红包采用乐观锁，那么在并发抢到相同版本号的拆红包请求中，只有一个能拆红包成功，其他的请求将事务回滚并返回失败，给用户报错，用户体验完全不可接受。

如果采用乐观锁，将会导致第一时间同时拆红包的用户有一部分直接返回失败，反而那些“手慢”的用户，有可能因为并发减小后拆红包成功，这会带来用户体验上的负面影响。

如果采用乐观锁的方式，会带来大数量的无效更新请求、事务回滚，给DB造成不必要的额外压力。

基于以上原因，微信红包系统不能采用乐观锁的方式解决并发抢锁问题。

四、微信红包系统的高并发解决方案

综合上面的分析，微信红包系统针对相应的技术难点，采用了下面几个方案，解决高并发问题。

1. 系统垂直SET化，分而治之

微信红包用户发一个红包时，微信红包系统生成一个 ID 作为这个红包的唯一标识。接下来这个红包的所有发红包、抢红包、拆红包、查询红包详情等操作，都根据这个 ID 关联。

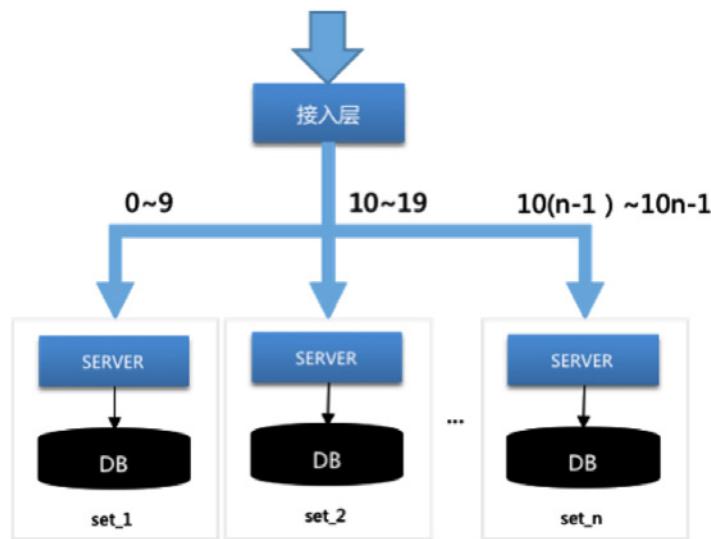
红包系统根据这个红包 ID，按一定的规则（如按 ID 尾号取模等），垂直上下切分。切分后，一个垂直链条上的逻辑 Server 服务器、DB 统称为一个 SET。

各个 SET 之间相互独立，互相解耦。并且同一个红包 ID 的所有请求，包括发红包、抢红包、拆红包、查详情等，垂直 stick 到同一个 SET 内处理，高度内聚。通过这样的方式，系统将所有红包请求这个巨大的洪流分散为多股小流，互不影响，分而治之，如下图所示。

这个方案解决了同时存在海量事务级操作的问题，将海量化为小量。

2. 逻辑Server层将请求排队，解决DB并发问题

红包系统是资金交易系统，DB 操作的事务性无法避免，所以会存在“并发抢锁”问题。但是如果到达 DB 的事务操作（也即拆红包行为）不是并发的，



而是串行的，就不会存在“并发抢锁”的问题了。

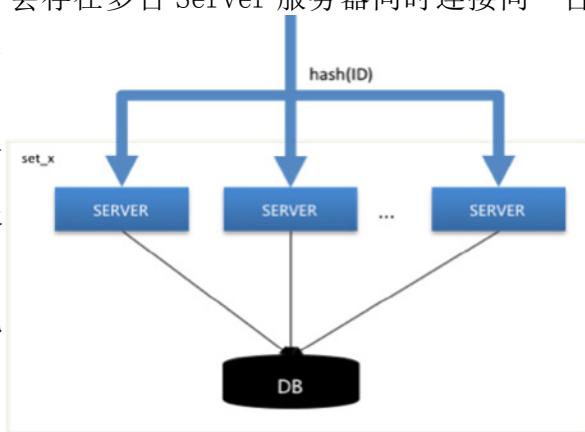
按这个思路，为了使拆红包的事务操作串行地进入 DB，只需要将请求在 Server 层以 FIFO（先进先出）的方式排队，就可以达到这个效果。从而问题就集中到 Server 的 FIFO 队列设计上。

微信红包系统设计了分布式的、轻巧的、灵活的 FIFO 队列方案。其具体实现如下：

首先，将同一个红包 ID 的所有请求 stick 到同一台 Server。

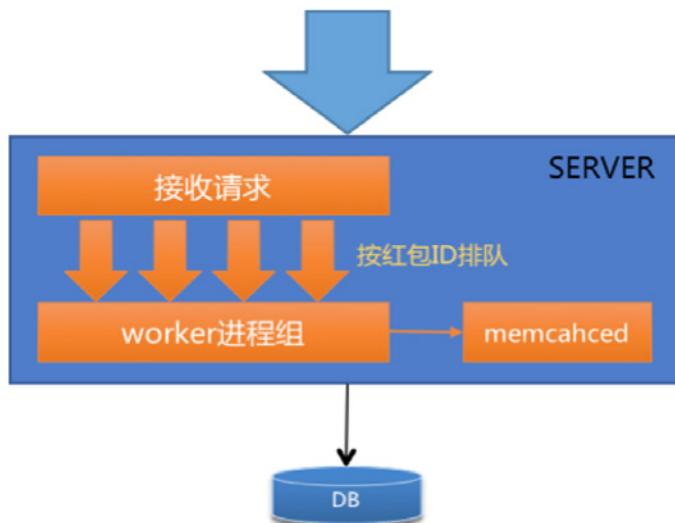
上面 SET 化方案已经介绍，同个红包 ID 的所有请求，按红包 ID stick 到同个 SET 中。不过在同个 SET 中，会存在多台 Server 服务器同时连接同一台 DB（基于容灾、性能考虑，需要多台 Server 互备、均衡压力）。

为了使同一个红包 ID 的所有请求，stick 到同一台 Server 服务器上，在 SET 化的设计之外，微信红包系统添加了一层基于红包 ID hash 值的分流，如下图所示。



其次，设计单机请求排队方案。

将 stick 到同一台 Server 上的所有请求在被接收进程接收后，按红包 ID 进行排队。然后串行地进入 worker 进程（执行业务逻辑）进行处理，从而达到排队的效果，如下图所示。



最后，增加 memcached 控制并发。

为了防止 Server 中的请求队列过载导致队列被降级，从而所有请求拥进 DB，系统增加了与 Server 服务器同机部署的 memcached，用于控制拆同一个红包的请求并发数。

具体来说，利用 memcached 的 CAS 原子累增操作，控制同时进入 DB 执行拆红包事务的请求数，超过预先设定数值则直接拒绝服务。用于 DB 负载升高时的降级体验。

通过以上三个措施，系统有效地控制了 DB 的“并发抢锁”情况。

3. 双维度库表设计，保障系统性能稳定

红包系统的分库表规则，初期是根据红包 ID 的 hash 值分为多库多表。随着红包数据量逐渐增大，单表数据量也逐渐增加。而 DB 的性能与单表数据量有一定相关性。当单表数据量达到一定程度时，DB 性能会有大幅度下降，影响系统性能稳定性。采用冷热分离，将历史冷数据与当前热数据分开存储，可以解决这个问题。

处理微信红包数据的冷热分离时，系统在以红包 ID 维度分库表的基础上，增加了以循环天分表的维度，形成了双维度分库表的特色。

具体来说，就是分库表规则像 db_xx.t_y_dd 设计，其中，xx/y 是红包 ID 的 hash 值后三位，dd 的取值范围在 01~31，代表一个月天数最多 31 天。

通过这种双维度分库表方式，解决了 DB 单表数据量膨胀导致性能下降的问题，保障了系统性能的稳定性。同时，在热冷分离的问题上，又使得数据搬迁变得简单而优雅。

综上所述，微信红包系统在解决高并发问题上的设计，主要采用了 SET 化分治、请求排队、双维度分库表等方案，使得单组 DB 的并发性能提升了 8 倍左右，取得了很好的效果。

五、总结

微信红包系统是一个高并发的资金交易系统，最大的技术挑战是保障并发性能与资金安全。这种全新的技术挑战，传统的“秒杀”系统设计方案已不能完全解决。在分析了业界“秒杀”系统解决方案的基础上，微信红包采用了 SET 化、请求排队串行化、双维度分库表等设计，形成了独特的高并发、资金安全系统解决方案，并在平时节假日、2015 和 2016 春节实践中充分证明了可行性，取得了显著的效果。在刚刚过去的 2017 鸡年除夕夜，微信红包收发峰值达到 76 万每秒，收发微信红包 142 亿个，微信红包系统的表现稳定，实现了除夕夜系统零故障。

作者简介

方乐明，现任微信支付应用产品系统负责人，主要从事微信红包、微信转账、微信群收款等支付应用产品的系统设计、可用性提升、高性能解决方案设计等，曾负责 2015、2016 和 2017 年春节微信红包系统的性能优化与稳定性提升，取得良好的效果。



为什么 Google 用 Apache Beam 彻底替换掉 MapReduce

作者 足下

“神说，要有光，就有了光”。

——《圣经》

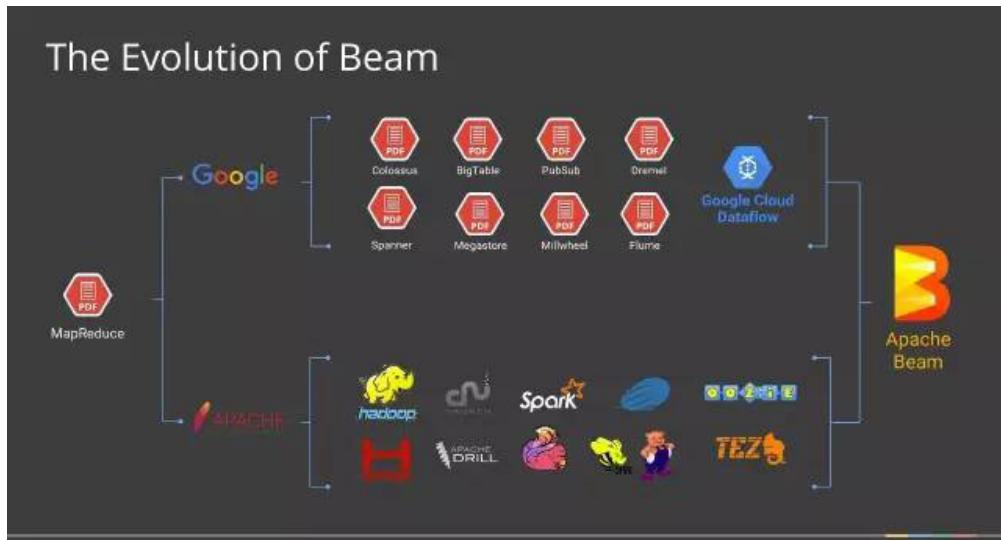
1月10日，Apache 软件基金会宣布，[Apache Beam](#) 成功孵化，成为该基金会的一个新的顶级项目，基于 Apache V2 许可证开源。

2003年，谷歌发布了著名的大数据三篇论文，史称三驾马车：Google FS、MapReduce、BigTable。虽然谷歌没有公布这三个产品的源码，但是她这三个产品的详细设计论文开启了全球的大数据时代！从 Doug Cutting 大神根据谷歌的论文实现出 Hadoop+MapReduce 的雏形，到 Hadoop 生态圈各种衍生产品的蓬勃发展，再到后来的 Spark、流式计算等等，所有的一切都要归功于、源自这三篇论文。可惜谷歌虽然开启了这个伟大的时代，却始终仅仅满足于偶尔发表一两篇论文以强调自己在理论和工程上的领导地位，从来没有亲身参与进来，尤其是没有为开源生态做出什么贡献，因而一直没有从大数据市场获得什么实在的好处。

痛定思痛，谷歌开始走开源之路，将自己的标准推广给社区。从众所周知的 Kubernetes，到 2016 年 2 月谷歌高调宣布将 Apache Beam（原名 Google

DataFlow) 贡献给 Apache 基金会孵化，再到最近大热的 Tensorflow 等等，动作不断。Apache Beam 被认为是继 MapReduce, GFS 和 BigQuery 等之后，谷歌在大数据处理领域对开源社区的又一个非常大的贡献。

也就是说，在大数据处理的世界里，谷歌一直在内部闭源，开发并使用着 BigTable、Spanner、Millwheel 等让大家久闻大名而又无缘一见的产品，开源世界演进出了 Hadoop、Spark、Apache Flink 等产品，现在他们终于殊途同归，走到一起来了。



为什么要推出开源的 Apache Beam

Apache Beam 的主要负责人 Tyler Akidau 在他的博客中提到他们做这件事的理念是：

要为这个世界贡献一个容易使用而又强大的模型，用于大数据的并行处理，同时适用于流式处理和批量处理，而且在各种不同平台上还可以移植。

那这一次为什么不是又酷酷的发表一篇论文，然后退居一旁静静的观察呢？为什么要联合一众伙伴为大家直接提供可以运行的代码了呢？原因主要有两点：

尽管在过去谷歌一直是闭源的，但在为云客户服务的过程中，谷歌已经认识到了开源软件的巨大价值，比如基于谷歌三篇论文产生的 Hadoop 社区就

是一个非常好的例子。思想上的转变使 Apache Beam 的诞生成为可能；

就 Beam 这个项目而言，要成功的必要条件之一是，必须有已经开源的 Runner 为 Beam 模型提供充分的支持，这样它才会在自建云和非谷歌云的场景下成为一个非常有竞争力的备选方案。去年 Apache Flink 在他们的系统内采用了 Beam 模型，这一条件也得到了满足；

无利不起早，谷歌这样做也是有着直接商业动机的，就是希望能尽可能多的 Apache Beam 数据处理流水线可以运行在谷歌的 Cloud Dataflow 上，别忘了这是 Apache Beam 的原型。进一步说，采用开源的方式来引导这件事，也是有许多直接好处的：

- 支持Apache Beam的Runner越多，它作为一个平台的吸引力就越大；
- 使用Apache Beam的用户越多，想在谷歌云平台上运行Apache Beam的用户也就越多；
- 开发Apache Beam过程中吸引到的伙伴越多，那对这样的数据处理模型的推广就越有利。

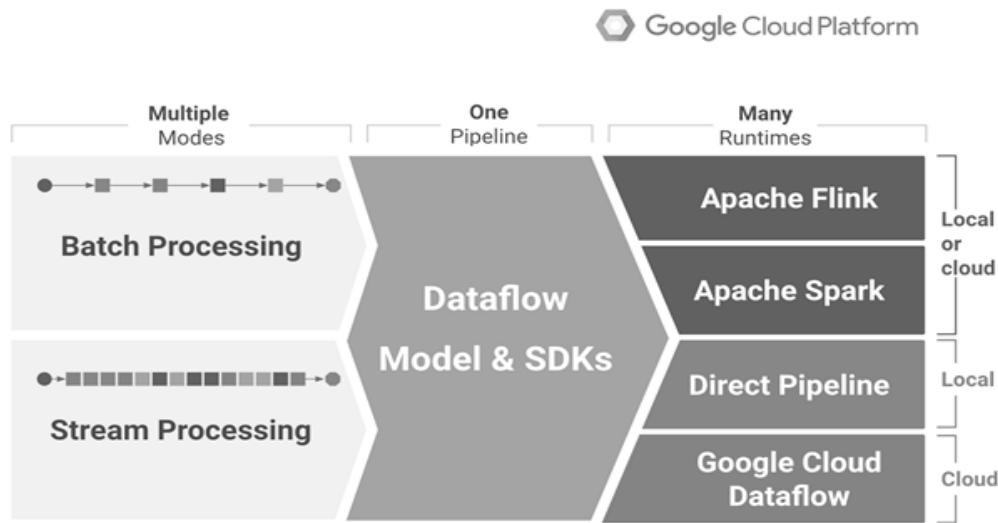
而且，好处也不会全都归于谷歌，Apache Beam 项目中的所有参与方都会受益。如果在构建数据处理流水线时存在着这样一个可移植的抽象层，那就会更容易出现新的 Runner，它们可以专注于技术创新，提供更高的性能、更好的可靠性、更方便的运维管理等。换句话说，消除了对 API 的锁定，就解放了处理引擎，会导致更多产品之间的竞争，从而最终对整个行业起到良性的促进作用。

谷歌坚信 Apache Beam 就是数据批量处理和流式处理的未来。这么做会为各种不同的 Runner 营造一个健康的生态系统，让它们之间相互竞争，而最后可以让用户得到实在的好处。

Apache Beam 是什么

要说 Apache Beam，先要说说谷歌 Cloud Dataflow。Dataflow 是一种原生的谷歌云数据处理服务，是一种构建、管理和优化复杂数据流水线的方法，用于构建移动应用、调试、追踪和监控产品级云应用。它采用了谷歌内部的技术 Flume 和 MillWheel，其中 Flume 用于数据的高效并行化处理，而

MillWheel 则用于互联网级别的带有很好容错机制的流处理。该技术提供了简单的编程模型，可用于批处理和流式数据的处理任务。她提供的数据流管理服务可控制数据处理作业的执行，数据处理作业可使用 DataFlow SDK 创建。



Apache Beam 本身不是一个流式处理平台，而是一个统一的编程框架，它提供了开源的、统一的编程模型，帮助你创建自己的数据处理流水线，实现可以运行在任意执行引擎之上批处理和流式处理任务。Beam 对流式计算场景中的所有问题重新做了一次归纳，然后针对这些问题提出了几种不同的解决模型，然后再把这些模型通过一种统一的语言给实现出来，最终这些 Beam 程序可以运行在任何一个计算平台上（只要相应平台——即 Runner 实现了对 Beam 的支持）。它的特点有以下几点。

- 统一的：对于批处理和流式处理，使用单一的编程模型。
- 可移植的：可以支持多种执行环境，包括Apache Apex、Apache Flink、Apache Spark和谷歌Cloud Dataflow等。
- 可扩展的：可以实现和分享更多的新SDK、IO连接器、转换操作库等。

Beam 特别适合应用于并行数据处理任务，只要可以将要处理的数据集分解成许多相互独立而又可以并行处理的小集合就可以了。Beam 也可以用于 ETL 任务，或者单纯的数据整合。这些任务主要就是把数据在不同的存储介质或者

数据仓库之间移动，将数据转换成希望的格式，或者将数据导入一个新系统。

Beam 主要包含两个关键的部分。

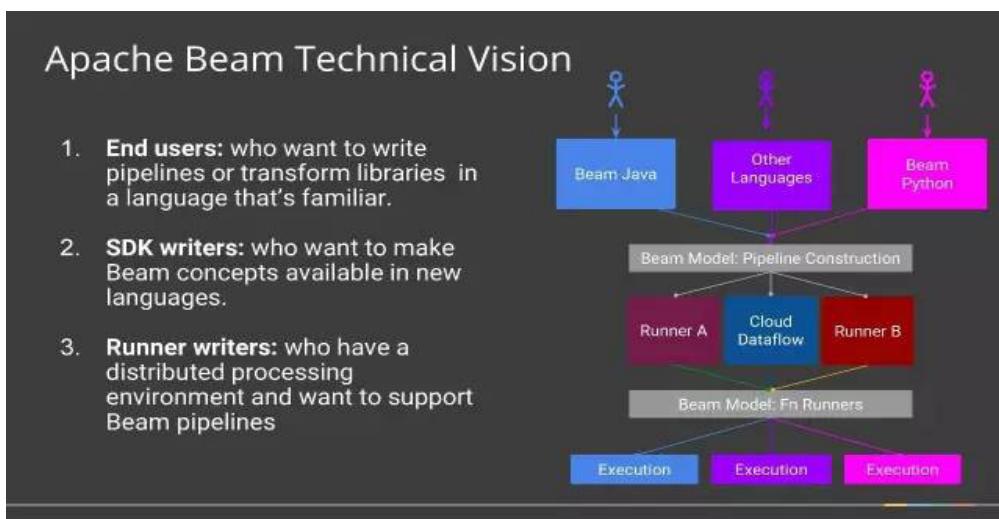
Beam SDK

Beam SDK 提供一个统一的编程接口给到上层应用的开发者，开发者不需要了解底层的具体的大数据平台的开发接口是什么，直接通过 Beam SDK 的接口，就可以开发数据处理的加工流程，不管输入是用于批处理的有限数据集，还是流式的无限数据集。对于有限或无限的输入数据，Beam SDK 都使用相同的类来表现，并且使用相同的转换操作进行处理。Beam SDK 可以有不同编程语言的实现，目前已经完整地提供了 Java, python 的 SDK 还在开发过程中，相信未来会有更多不同的语言的 SDK 会发布出来。

Beam Pipeline Runner

Beam Pipeline Runner 将用户用 Beam 模型定义开发的处理流程翻译成底层的分布式数据处理平台支持的运行时环境。在运行 Beam 程序时，需要指明底层的正确 Runner 类型。针对不同的大数据平台，会有不同的 Runner。目前 Flink、Spark、Apex 以及谷歌的 Cloud DataFlow 都有支持 Beam 的 Runner。

需要注意的是，虽然 Apache Beam 社区非常希望所有的 Beam 执行引擎都能够支持 Beam 定义的功能全集，但是在实际实现中可能并不一定。例



如，基于 MapReduce 的 Runner 显然很难实现和流处理相关的功能特性。就目前状态而言，对 Beam 模型支持最好的就是运行于谷歌云平台之上的 Cloud Dataflow，以及可以用于自建或部署在非谷歌云之上的 Apache Flink。当然，其它的 Runner 也正在迎头赶上，整个行业也在朝着支持 Beam 模型的方向发展。

那大家可以怎样与 Beam 做亲密接触呢？

如上图所示，主要有三个方面。

- 数据处理：直接使用已有的自己熟悉语言的SDK，根据Beam模型去定义并实现自己的数据处理流程。
- SDK实现：用新的编程语言去根据Beam概念实现SDK，这样大家以后在编程语言方面就可以有更多选择了。
- Runner实现：将已有的分布式数据处理平台作为一种新的Runner，接入Beam模型。

Beam 是怎么做的

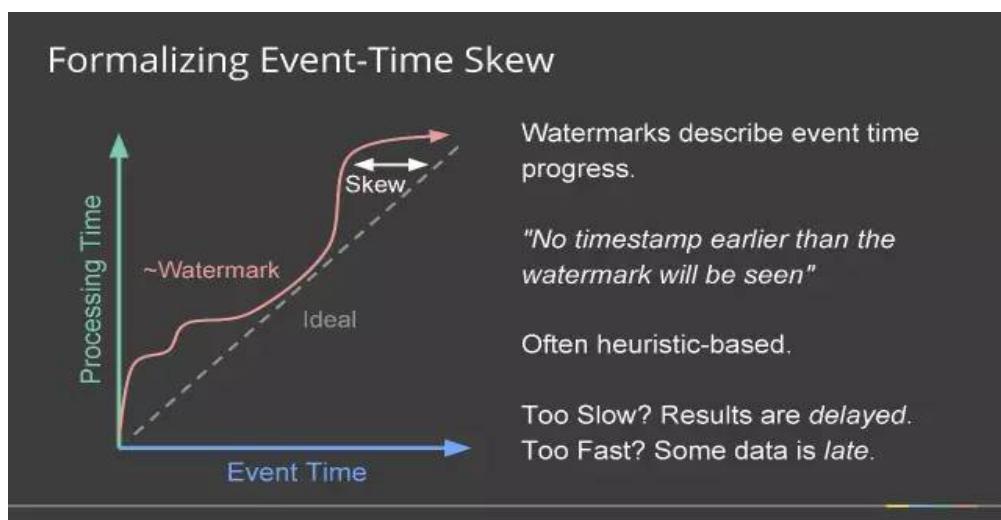
在任何一个设计开始之前，都先要确定问题，Beam 也不例外。

1. 数据。分布式数据处理要处理的数据类型一般可以分为两类，有限的数据集和无限的数据流。有限的数据集，比如一个HDFS中的文件，一个HBase表等，特点是数据提前已经存在，一般也已经持久化，不会突然消失，不会再改变。而无限的数据流，比如kafka中流过来的系统日志流，或是从Twitter API拿到的Twitter流等等，这类数据的特点是，数据动态流入，无穷无尽，无法全部持久化。一般来说，批处理框架的设计目标是用来处理有限的数据集，流处理框架的设计目标是用来处理无限的数据流。有限的数据集可以看做是无限的数据流的一种特例，但是从数据处理逻辑的角度，这两者并无不同之处。
2. 时间。Process Time是指数据进入分布式处理框架的时间，而Event-Time则是指数据产生的时间。这两个时间通常是不同的，例如，对于一个处理微博数据的流计算任务，一条2016-06-01-12:00:00发表的微博经过网络传输等延迟可能在2016-06-01-12:01:30才进入到流处理系统中。批处理任务通常进行全量的数据计算，较少关注数据的时间属

性，但是对于流处理任务来说，由于数据流是无情无尽的，无法进行全量的计算，通常是对某个窗口中得数据进行计算，对于大部分的流处理任务来说，按照时间进行窗口划分，可能是最常见的需求。

- 乱序。对于流处理框架处理的数据流来说，其数据的到达顺序可能并不严格按照Event-Time的时间顺序。如果基于Process Time定义时间窗口，数据到达的顺序就是数据的顺序，因此不存在乱序问题。但是对于基于Event Time定义的时间窗口来说，可能存在时间靠前的消息在时间靠后的消息之后到达的情况，这在分布式的数据源中可能非常常见。对于这种情况，如何确定迟到数据，以及对于迟到数据如何处理通常是很棘手的问题。

Beam 模型处理的目标数据是无限的时间乱序数据流，不考虑时间顺序或是有限的数据集可看做是无限乱序数据流的一个特例。

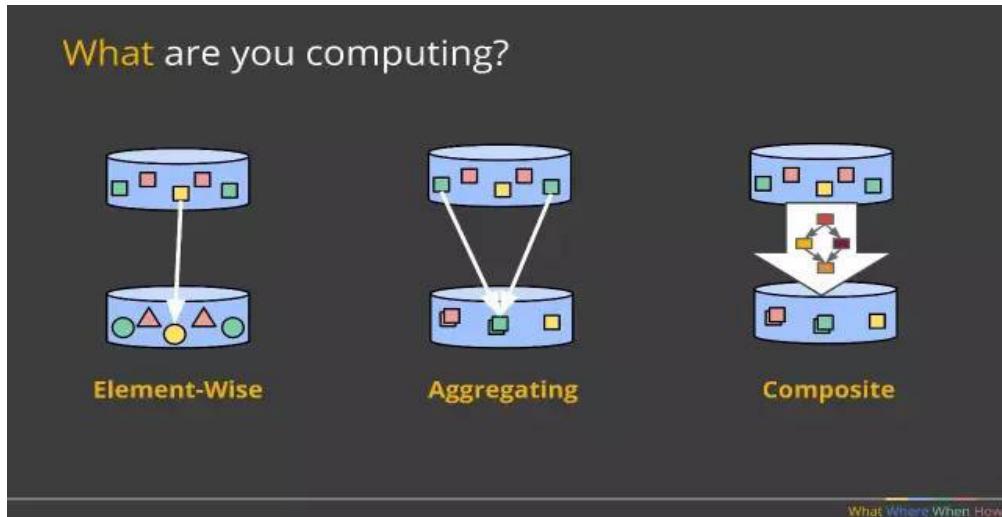


如上图，其中虚线是最理想的，表示处理时间和事件时间是相同的，红线是实际上的线，也叫水位线（Watermark），它一般是通过启发式算法算出来的。

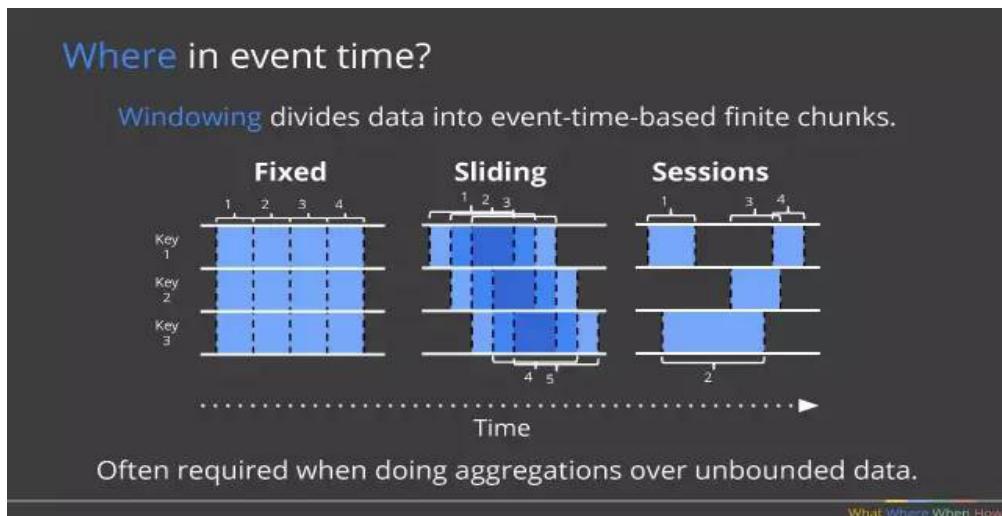
接下来从问题中抽象出四个具体的问题。

A: What are you computing, 对数据的处理是哪种类型，数据转换、聚合或者是两者都有。例如，Sum、Join 或是机器学习中训练学习模型等。在

Beam SDK 中由 Pipeline 中的操作符指定。如图：

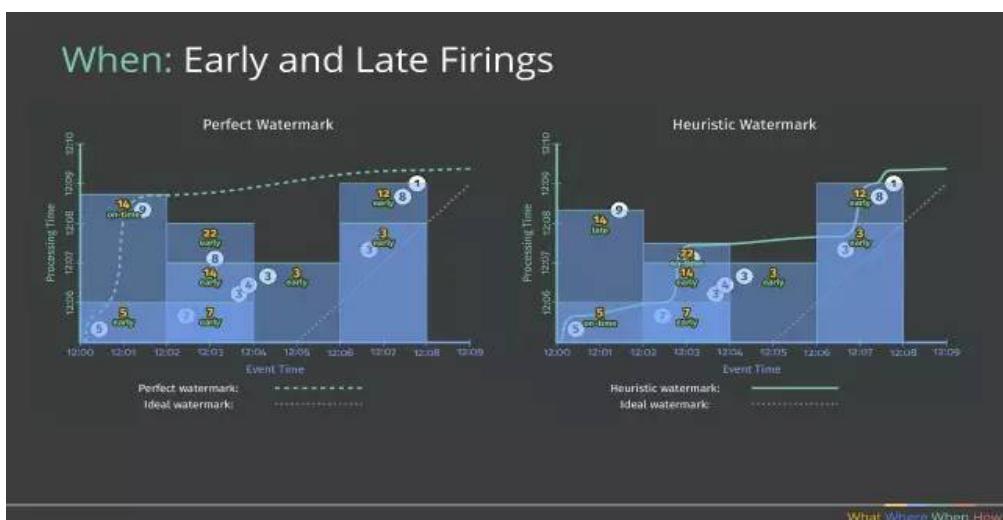
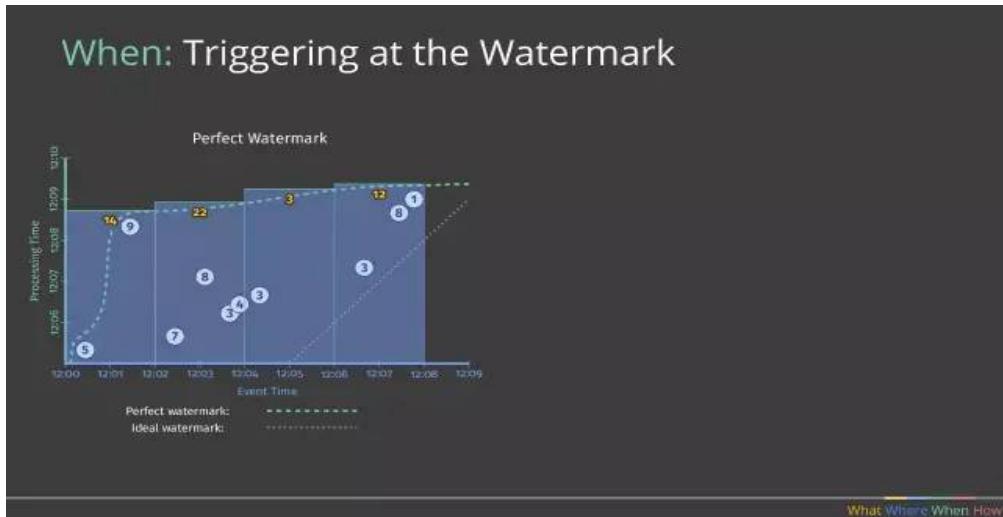


B: Where in event time, 数据在什么范围内计算？例如，基于 Process-Time 的时间窗口？基于 Event-Time 的时间窗口？滑动窗口等等。在 Beam SDK 中由 Pipeline 中的窗口指定：



C: When in processing time, 何时将计算结果输出？在这里引入了一个 Trigger 机制，Trigger 决定何时将计算结果发射出去，发射太早会丢失一部

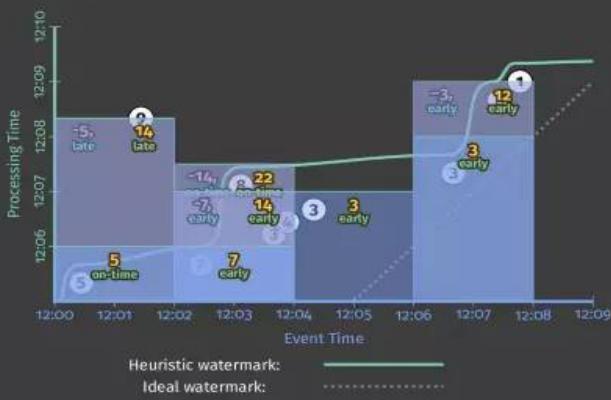
分数据，丧失精确性，发射太晚会导致延迟变长，而且会囤积大量数据，何时 Trigger 是由水位线来决定的，在 Beam SDK 中由 Pipeline 中的水位线和触发器指定。



D: How do refinements relate, 迟到数据如何处理？例如，将迟到数据计算增量结果输出，或是将迟到数据计算结果和窗口内数据计算结果合并成全量结果输出。在 Beam SDK 中由 Accumulation 指定。

Beam 模型将“WWWH”四个维度抽象出来组成了 Beam SDK，用户在基于

How: Add Newest, Remove Previous



What Where When How

Beam SDK 构建数据处理业务逻辑时，每一步只需要根据业务需求按照这四个维度调用具体的 API 即可生成分布式数据处理 Pipeline，并提交到具体执行引擎上执行。“WWWH”四个维度的抽象仅仅关注业务逻辑本身，和分布式任务如何执行没有任何关系。

在 QCon 旧金山 2016 上，Apache Beam 的创始人 Tyler Akidau 分享了该项目的设计理念和架构。

友商的看法

随着分布式数据处理不断发展，新的分布式数据处理技术也不断被提出，业界涌现出了越来越多的分布式数据处理框架，从最早的 Hadoop MapReduce，到 Apache Spark、Apache Storm，以及更近的 Apache Flink、Apache Apex 等。新的分布式处理框架可能带来的更高的性能、更强大的功能和更低的延迟，但用户切换到新的分布式处理框架的代价也非常大：需要学习一个新的数据处理框架，并重写所有的业务逻辑。解决这个问题的思路包括两个部分，首先，需要一个编程范式，能够统一、规范分布式数据处理的需求，例如，统一批处理和流处理的需求。其次，生成的分布式数据处理任务应该能够在各个分布式执行引擎上执行，用户可以自由切换分布式数据处理任务的执行引擎与执行环境。Apache Beam 正是为了解决以上问题而提出的。

如 Apache Beam 项目的主要推动者 Tyler Akidau 所说：为了让 Apache Beam 能成功地完成移植，我们需要至少有一个在部署自建云或非谷歌云时，可以与谷歌 Cloud Dataflow 相比具备足够竞争力的 Runner。如 Beam 能力矩阵所示，Flink 满足我们的要求。有了 Flink，Beam 已经在业界内成了一个真正有竞争力的平台。

对此，Data Artisan 的 Kostas Tzoumas 在他的博客中说：在谷歌将他们的 Dataflow SDK 和 Runner 捐献给 Apache 孵化器成为 Apache Beam 项目时，谷歌希望我们能帮忙完成 Flink Runner，并且成为新项目的代码提交者和 PMC 成员。我们决定全力支持，因为我们认为：1、对于流处理和批处理来说 Beam 模型都是未来的参考架构；2、Flink 正是一个执行这样数据处理的平台。在 Beam 成形之后，现在 Flink 已经成了谷歌云之外运行 Beam 程序的最佳平台。

我们坚信 Beam 模型是进行数据流处理和批处理的最佳编程模型。我们鼓励用户们在实现新程序时采用这个模型，用 Beam API 或者 Flink DataStream API 都行。

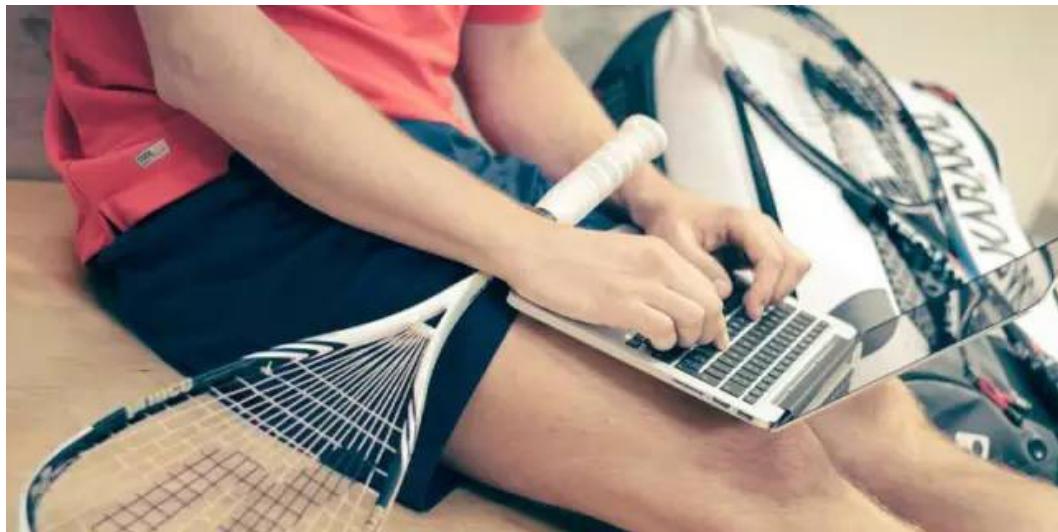
目前主流流数据处理框架 Flink、Spark、Apex 以及谷歌的 Cloud DataFlow 等都有了支持 Beam 的 Runner。

结论

“在谷歌公司里已经没人再使用 MapReduce 了”！谷歌云的主要负责人 Mete Atamel 如是说。谷歌坚信 Apache Beam 就是数据批处理和流处理的未来。Apache Beam 的模型对无限乱序数据流的数据处理进行了非常优雅的抽象，“WWWH”四个维度对数据处理的描述非常清晰与合理，Beam 模型在统一了对无限数据流和有限数据集的处理模式的同时，也明确了对无限数据流的数据处理方式的编程范式，扩大了流处理系统可应用的业务范围。随着 Apache Beam 的成功孵化，随着越来越多的编程语言可用、越来越多的分布式数据处理平台支持 Beam 模型，我们的确可以尽情畅想美好的未来。在今年 4 月的 QCon 北京 2017 上，QCon 将邀请 PayPal 架构师、Apache Beam 贡献者及 PPMC 成员 Amit Sela 来进一步分享 Apache Beam 的相关技术。

禁用 Python 的 GC 机制后，Instagram 性能提升 10%

作者 刘志勇



通过关闭 Python 垃圾回收（Garbage Collection, GC）机制（通过回收和释放未使用的数据来回收内存），Instagram 的性能可以提高 10%。是的，你没有听错！通过禁用 GC，我们可以减少内存占用并提高 CPU LLC 缓存命中率。如果你想知道为什么，那么就来阅读 Chenyang Wu 和 Min Ni 为此撰写的文章。

作者 Chenyang Wu 是 Instagram 的软件工程师，Min Ni 是 Instagram 的技术经理。

我们如何管理 Web 服务器

Instagram 的 web 服务器以多进程的模式运行在 Django 上，主进程分叉创建几十个工作进程，用来接收传入的用户请求。对于应用程序服务器，我们使用带前置模式的 uWSGI 来利用主进程和工作进程之间的内存共享。

为了防止 Django 服务器运行到 OOM，uWSGI 主进程提供了一种机制，当其

RSS 内存超过阈值时重新启动工作进程。

了解内存

我们开始研究工作 RSS 内存为什么在由主进程产生后迅速增长。一个观察是，即使 RSS 存储器以 250MB 开始，其共享内存下降非常快：在几秒钟内从 250MB 降到约 140MB（共享内存的大小可以从 /proc/PID/smaps 读取）。这里的数字是无趣的，因为它们一直在变动，但共享内存丢弃的规模很有趣：大约 1/3 的总内存。接下来，我们想要了解为什么共享内存在工作器产生伊始就变为每个进程的私有内存。

我们的理论：读时复制

Linux 内核有一个称为写入复制 (Copy-on-Write, CoW) 的机制，用作分叉进程的优化。子进程通过与其父进程共享每个内存页开始。仅当页面被写入时复制到子内存空间的页面（有关详细信息，请参阅维基百科上的 Copy_on_Write 词条）。

但在 Python 中，由于引用了计数，事情变得有趣了。每次我们读取一个 Python 对象时，解释器将增加其引用计数，这本质上是对其底层数据结构的写入。这就导致了 CoW。因此，使用 Python，我们就进行读时复制 (Copy-on-Read, CoR)！

```
#define PyObject_HEAD \
    _PyObject_HEAD_EXTRA \
    Py_ssize_t ob_refcnt; \
    struct _typeobject *ob_type; \
    ... \
typedef struct _object { \
    PyObject_HEAD \
} PyObject;
```

那么问题是：我们是在写时复制不可变对象（如代码对象）么？给定 PyCodeObject 确实是 PyObject 的“子类”，那么答案显然为：是。我们的第

一个想法，是禁用对 PyCodeObject 的引用计数。

尝试 1：禁用代码对象的引用计数

在 Instagram，我们先做简单的事情。考虑到这是一个实验，我们对 CPython 解释器做了一些小的修改，验证了引用计数对代码对象没有改变，然后将 CPython 应用到我们的一个生产服务器。

结果令人失望，因为共享内存没有变化。当我们试图找出原因时，我们意识到没有任何可靠的指标来证明分析是否正确，也不能证明共享内存和代码对象的副本之间的关系。显然，这里缺少一些什么东西。由此获得的经验是：在运作之前证明你的理论。

分析页面故障

当我们在 Google 上搜索关于 Copy-on-Write 的资料后，了解到 Copy-on-Write 与系统中的页面错误是相关联的。每个 CoW 在过程中触发页面错误。Linux 附带的 Perf 工具允许记录硬件 / 软件系统事件，包括页面错误，甚至可以提供堆栈跟踪！

Overhead	Samples	Command	Shared Object	Symbol
- 17.08%	198	uwsgi	uwsgi	[.] collect.part.7
- collect.part.7				
- _PyObject_GC_New				
+ 98.54% PyDict_New				
+ 1.46% list_iter				
+ 13.43%	356	uwsgi	_scrypt.so	[.] crypto_scrypt
+ 11.99%	444	uwsgi	libc-2.20.so	[.] __int_malloc
+ 7.11%	291	uwsgi	uwsgi	[.] PyObject_Malloc
+ 5.91%	218	uwsgi	uwsgi	[.] PyObject_GenericGetAttr
+ 4.68%	176	uwsgi	uwsgi	[.] PyEval_EvalFrameEx
+ 4.61%	167	uwsgi	uwsgi	[.] PyFrame_New
+ 2.37%	27	mc-eccc-pool	libc-2.20.so	[.] __int_malloc
+ 2.29%	96	uwsgi	libc-2.20.so	[.] __memcpy_sse2_unaligned
+ 2.25%	48	cfgator-sub	libc-2.20.so	[.] __int_malloc

于是我们运行了一个 prod 服务器，重启服务器后，等待它进行分叉，得到了一个工作进程的 PID，然后运行以下命令：

```
perf record -e page-faults -g -p <PID>
```

我们就有了一个新的想法，看看当页面错误如果发生在堆栈跟踪的过程中会发生什么。

结果出乎意料，并没有复制代码对象，最大的疑凶是 collect，它属于 gcmodule.c，并在触发垃圾回收时被调用。在阅读了 GC 在 CPython 中的工作原理后，我们得出了以下理论：基于阈值确定性地触发 CPython 的 GC。默认阈值非常低，因此它在很早的阶段就开始了。它维护对象的分代链接列表，并且在 GC 期间，链接列表被洗牌。因为链接列表结构与对象本身一起存在（就像 ob_refcount），在链接列表中改写这些对象将导致页面被 CoW，这是一个不幸的副作用。

```
/* GC information is stored BEFORE the object structure. */
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;/
        union _gc_head *gc_prev;
        Py_ssize_t gc_refs;
    } gc;
    long double dummy; /* force worst-case alignment */
} PyGC_Head;
```

尝试 2：尝试禁用 GC

既然是 GC 捆了我们一刀，那就禁用它！

我们引导脚本添加了一个 gc.disable() 调用，然后重启了服务器。我们重新启动了服务器，但是，很不幸！如果我们再次查看 perf，将会看到 gc.collect 仍然被调用，并且内存仍然被复制。利用 GDB 的一些调试，我们发现，使用的一个第三方库 (msgpack) 调用 gc.enable() 将其恢复，因此 gc.disable() 在引导时被清除。

修补 msgpack 是我们要做的最后一件事，因为它意味着我们没有注意到其他库在未来也会做同样的事情。首先，我们需要证实禁用 GS 实际上是很有帮助的。答案存在于 gcmodule.c 中。作为 gc.disable 的替代，我们做了

gc.set_threshold(0)，这一次，没有任何库被恢复过来。

这样，我们成功地将每个工作进程的共享内存从 140MB 提高到 225MB，并且每台机器在主机上的总内存使用量减少了 8GB。这就为整个 Django 集群节约了 25% 的内存。有了这么大的头部空间，，我们能够运行更多的进程或运行带有更高的 RSS 内存阈值。实际上，这样的改进将 Django 层的吞吐量提高了 10% 以上。

尝试 3：需要完全禁止 GC

在我们尝试了一堆设置之后，我们决定在更大的范围内尝试：集群。反馈相当快，因为禁用 GC 后，重启 Web 服务器变得很慢，以至于我们的连续部署被中断了。通常重启耗时不到 10 秒钟，但禁用 GC 后，有时候，耗时会超过 60 秒。

```
2016-05-02_21:46:05.57499 WSGI app 0 (mountpoint='') ready in
115 seconds on interpreter 0x92f480 pid: 4024654 (default app)
```

PRC	sys	41.17s	user	1m48s	#proc	451	#run	22	#tslpi	395	#tslpu	3	#zombie	31	clones	190	#exit	77
CPU	sys	589%	user	850%	irq	28%	idle	1722%	wait	18%	steal	0%	guest	0%	curlf	2.60GHz	curscal	100%
CPL	avg1	5.77	avg5	6.21	avg15	6.43			csw	704450			intr	672928		numcpu	32	
MEM	tot	15.5G	free	1.6G	cache	358.9M	buff	46.0M	slab	350.2M	shmem	85.1M	vmbal	0.0M	hptot	0.0M	hpuse	0.0M
SMP	tot	0.0M	free	0.0M											vmcom	66.8G	vnlim	7.8G
PAG	scan	435956	steal	433140	stall	4854									swin	0	swout	0
DSK	sda	busy	24%	read	279	write	150	KiB/r	71	KiB/w	8	MBr/s	3.92	MBr/s	0.24	avio	2.79 ms	
NET	transport	tcpi	109255	tcpo	135770	udpi	173	udpo	176	tcppo	899	tcppo	848	tcprs	0	udpie	0	
NET	network	ipi	109444	ipo	134727	ipfrw	0	deliv	109444						icmpe	16	icmpe	6
NET	lo	---- pcki	61062	pcko	61062	si	80 Mbps	so	80 Mbps	erri	0	erro	0	drpi	0	drpo	0	
NET	eth0	---- pcki	52671	pcko	77363	si	39 Mbps	so	26 Mbps	erri	0	erro	0	drpi	0	drpo	0	
PID	TID	MINFLT	MAJFLT	VSTEXT	VSLIBS	VDATA	VSTACK	VSIZE	RSIZE	PSIZE	VGRW	RGROW	SWAPSZ	RUID	EUID	MEM	CMD	1/28
1355149	-	401	8	2748K	130.7M	4.4G	164K	5.1G	372.5M	0K	512K	-26.7M	OK	root	root	2%	uwsgi	
1356405	-	13216	0	2748K	130.9M	4.2G	164K	4.8G	360.8M	0K	-0.2G	-36.3M	OK	root	root	2%	uwsgi	
1358066	-	29566	0	2748K	130.7M	4.2G	164K	4.7G	357.7M	0K	-0.3G	-39.1M	OK	root	root	2%	uwsgi	
1355057	-	22645	0	2748K	130.7M	4.1G	164K	4.7G	352.4M	0K	-0.1G	-41.5M	OK	root	root	2%	uwsgi	
1359520	-	28992	0	2748K	130.7M	4.2G	164K	4.8G	352.0M	0K	-0.3G	-37.9M	OK	root	root	2%	uwsgi	
1355057	-	22645	0	2748K	130.7M	4.1G	164K	4.7G	352.4M	0K	-0.1G	-41.5M	OK	root	root	2%	uwsgi	
1359520	-	28992	0	2748K	130.7M	4.2G	164K	4.8G	352.0M	0K	-0.3G	-37.9M	OK	root	root	2%	uwsgi	
1357586	-	30114	0	2748K	130.7M	4.1G	168K	4.6G	347.0M	0K	-0.2G	-41.0M	OK	root	root	2%	uwsgi	
1362457	-	25100	0	2748K	130.7M	3.9G	164K	4.4G	346.6M	0K	-0.3G	-37.3M	OK	root	root	2%	uwsgi	
1358067	-	27515	0	2748K	130.7M	3.9G	164K	4.5G	343.2M	0K	-0.3G	-39.4M	OK	root	root	2%	uwsgi	
1358352	-	31036	0	2748K	130.7M	4.0G	168K	4.6G	342.5M	0K	-0.3G	-41.4M	OK	root	root	2%	uwsgi	
1354801	-	23617	0	2748K	130.7M	4.0G	164K	4.6G	342.1M	0K	-0.3G	-42.4M	OK	root	root	2%	uwsgi	
1363067	-	16144	0	2748K	130.7M	4.2G	168K	4.8G	340.8M	0K	-0.2G	-35.0M	OK	root	root	2%	uwsgi	
1362194	-	13460	1	2748K	130.7M	4.1G	164K	4.6G	340.8M	0K	-0.2G	-37.7M	OK	root	root	2%	uwsgi	
1363301	-	8848	0	2748K	130.7M	4.0G	164K	4.6G	332.4M	0K	77044K	-20.2M	OK	root	root	2%	uwsgi	
1363964	-	20480	0	2748K	130.7M	4.1G	164K	4.7G	320.1M	0K	-0.3G	-33.0M	OK	root	root	2%	uwsgi	
1365054	-	23372	0	2748K	130.7M	4.1G	164K	4.7G	309.9M	0K	-0.3G	-36.6M	OK	root	root	2%	uwsgi	

重现这个 bug 非常伤脑筋，因为它不是确定性的。经过大量实验后，一个真正的 re-top 在顶部显示了。当发生这种情况时，主机上的可用内存骤降到接近零并跳回，强迫所有的高速缓存内存撤出。然后到所有的代码 / 数据需要

从磁盘读取 (DSK 100%) 的时刻，一切都慢吞吞的。

听上去很奇怪，Python 会在关闭解释器之前做最后一个 GC，这会在很短的时间内，导致内存使用量产生巨大的飞跃。再者就是，我想先证明它，然后弄清楚如何正确处理它。因此，我在 uWSGI 的 python 插件中注释掉 Py_Finalize 的调用，问题就消失了。

但显然的是，我们不能对 Py_Finalize 只是一禁了之。因为我们有一堆重要的清理，要用到依赖它的 atexit 钩子。最后，我们所做的就是，在 CPython 添加一个运行时标志，来完全禁用 GC。

最后，我们开始将这个做法推广到更大的规模。此后，我们在整个集群进行尝试，但是，连续部署再次被中断了。不过，这次它只是在旧 CPU 型号 (Sandybridge) 的机器上中断了，甚至更难重现。经验教训：要多测试旧式客户端 / 旧型号，因为他们最容易被中断。

Samples: 1K of event 'page-faults', Event count (approx.): 30066				
Overhead	Samples	Command	Shared Object	Symbol
+ 14.12%	168	uwsgi	libc-2.20.so	[.] _int_free
+ 14.03%	227	uwsgi	uwsgi	[.] insertdict
- 13.74%	151	uwsgi	uwsgi	[.] PyType_Modified.part.29
- PyType_Modified.part.29				
- 97.82% PyType_ClearCache				
Py_Finalize.part.3				
uwsgi_plugins_atexit				
0				
+ 2.18% PyType_Modified.part.29				
- 11.65% uwsgi	182	uwsgi	uwsgi	[.] PyObject_GC_UnTrack
+ PyObject_GC_UnTrack				
+ 6.58%	87	uwsgi	uwsgi	[.] dict_dealloc
- 6.04%	81	uwsgi	uwsgi	[.] PyObject_Free
- PyObject_Free				
- 93.29% dict_dealloc				
- 93.45% dict_dealloc				
- 98.36% instance_dealloc				
dict_dealloc				
insertdict				
PyDict_SetItem				
_PyModule_Clear				
PyImport_Cleanup				
Py_Finalize.part.3				
uwsgi_plugins_atexit				
0				

因为我们的连续部署是一个相当快的过程，为了真正捕获发生了什么，我在 rollout 命令添加了一个单独的 atop。这样我们就能够抓住高速缓存内存真的很低的一个时刻。所有 uWSGI 进程触发了很多 MINFLT (minor page faults, 小页面错误)。

再次通过 perf 得出的概要，我们再次看到了 Py_Finalize。在关机时，除了最终的 GC，Python 做了一堆清理操作，如破坏类型对象和卸载模块。这又一次损害了共享内存。

尝试 4：关闭 GC 的最后一步：无须清理

为什么我们需要清理？这个进程将会死掉去，我们将得到另一个替代品。我们真正关心的是清理应用程序的 atexit 钩子。至于 Python 的清理，我们不必这样做。下面是在 bootstrapping 脚本中的结束：

```
# gc.disable() doesn't work, because some random 3rd-party
library will
# enable it back implicitly.
gc.set_threshold(0)
# Suicide immediately after other atexit functions finishes.
# CPython will do a bunch of cleanups in Py_Finalize which
# will again cause Copy-on-Write, including a final GC
atexit.register(os._exit, 0)
```

基于这个事实，atexit 函数以注册表的相反顺序运行。atexit 函数完成其他清除，然后调用 os._exit(0) 来退出最后一步的当前进程。

随着这两条线的变化，我们终于完成了整个集群的推广。在仔细调整内存阈值后，我们获得了 10% 的全局性能提升！

回顾

在回顾这次性能的提升时，我们有两个疑问。

首先，没有垃圾回收的话，因为所有的内存分配不会释放，Python 内存就不会爆破吗？（记住，在 Python 内存中没有真正的堆栈，因为所有的对象

都是在堆上分配的。)

幸运的是，这并非事实。Python 中用于释放对象的主要机制仍然是引用计数。当一个对象被解除引用（调用 Py_DECREF）时，Python 运行时总是检查其引用计数是否降到零。在这种情况下，将调用对象的释放器。垃圾回收的主要目的是打破引用计数不起作用的参考周期。

```
#define Py_DECREF(op) \
    do { \
        if (_Py_DEC_REFTOTAL < _Py_REF_DEBUG_COMMA \
            --((PyObject*)(op))->ob_refcnt != 0) \
            _Py_CHECK_REFCNT(op) \
        else \
            _Py_Dealloc((PyObject*)(op)); \
    } while (0)
```

打破增益

第二个问题：增益来自哪里？

禁用 GC 的增益是两倍：

- 我们为每个服务器释放了大约8GB的RAM，用于为内存绑定服务器生成创建更多的工作进程，或者降低CPU绑定服务器生成的工作程序刷新率；
- 随着每周期CPU指令（IPC）增加约10%，CPU吞吐量也随之提高。

```
# perf stat -a -e cache-misses,cache-references -- sleep 10
Performance counter stats for 'system wide':
      268,195,790 cache-misses # 12.240 % of all cache refs
[100.00%]
      2,191,115,722      cache-references
      10.019172636 seconds time elapsed
```

禁用 GC 时，高速缓存未命中率有 2~3% 的下降，主要原因是 IPC 增加 10% 所致。CPU 高速缓存未命中的代价太高了，因为它使 CPU 管道停滞。对 CPU 缓存命中率的微小改进通常可以显著提高 IPC。使用较少的 CoW，具有不同虚拟

地址（在不同的工作进程中）的更多 CPU 高速缓存线指向相同的物理存储器地址，导致更好的高速缓存命中率。

我们可以看到，并非每个组件都按预期工作，有时，结果可能会非常令人惊讶。所以继续挖掘、四处观望，你会惊讶事情究竟是如何运作的！

复盘 GC 算法的发展历程及现状，其实 Go 语言并没有什么大的突破

作者 薛命灯



最近，我读到一些大肆宣传 Go 语言最新垃圾回收器的文章，这些文章对垃圾回收器的描述让我感到有些厌烦和恼怒，因为这其中有些是来自 Go 项目，他们宣称 GC 技术正迎来巨大突破。

下面 Go 团队在 2015 年 8 月发布的新垃圾回收器的启动声明：

Go 正在构建一个划时代垃圾回收器，2015 年，甚至到 2025 年，或者更久……
Go 1.5 的 GC 把我们带入了一个新时代，垃圾回收停顿不再成为使用新语言的障碍。应用程序可以很容易地随着硬件进行伸缩，而且随着硬件越来越强大，GC 不再是构建可伸缩软件的阻碍。一个新的时代正在开启。

Go 团队不仅宣称他们已经解决了 GC 停顿问题，而且让整件事情变得更加“傻瓜”化：

从更高层面解决性能问题的方式之一是增加 GC 选项，为不同的性能问题

设置不同的选项。程序员可以通过选项为他们的应用程序找到合适的设置。不过，这种方式的不足之处在于，选项（就是经常用到的配置参数）数量会不断增加，到最后很可能会需要一部“GC 选项操作者就业草案”。Go 不想继续走这条路。相反，我们只提供了一个选项，也就是 GOGC。

而且，因为不需要支持太多的选项，运行时团队可以集中精力根据真实的用户反馈来改进运行时。

我相信很多 Go 语言用户对新的运行时还是很满意的。不过我对之前的那些观点有异议，对于我来说，它们是对市场的误导。这些观点在博客圈内重复出现，我想是时候对它们进行深入分析了。

事实上，**Go 的 GC 并没有真正实现任何新的想法或做出任何有价值的研究**。他们在声明里也承认，他们的回收器是一种并发的标记并清除回收器，而这种想法在 70 年代就有了。他们的回收器之所以还值得一提，完全是因为它对停顿时间进行了改进，而这是以牺牲 GC 其它方面的特性为代价的。Go 相关的技术讨论和发行材料并没有提到他们在这个问题上所做出的折衷，让那些不熟悉垃圾回收技术的开发人员不知道这些问题的存在，还暗示 Go 的其它竞争者制造的都是垃圾。而 Go 也在强化这种误导：

为了创建划时代的垃圾回收器，我们在很多年前就采用了一种算法。Go 的新回收器是一种并发的、三基色的、标记清除回收器，它的设计想法是由 Dijkstra 在 1978 年提出的。它有别于现今大多数“企业”级垃圾回收器，而且我们相信它跟现代硬件的属性和现代软件的低延迟需求非常匹配。

读完这段声明，你会感觉过去 40 年的“企业”级 GC 研究没有任何进展。

GC 理论基础

下面列出了在设计垃圾回收算法时要考虑的一些因素：

- 程序吞吐量：回收算法会在多大程度上拖慢程序？有时候，这个是通过回收占用的CPU时间与其它CPU时间的百分比来描述的。
- GC吞吐量：在给定的CPU时间内，回收器可以回收多少垃圾？
- 堆内存开销：回收器最少需要多少额外内存开销？如果回收器在回收

垃圾时需要分配临时的内存，对于程序内存使用是否会有严重影响？

- 停顿时间：回收器会造成多长时间的停顿？
- 停顿频率：回收器造成的停顿频率是怎样的？
- 停顿分布：停顿有时候很短暂，有时候很长？还是选择长一点但保持一致的停顿时间？
- 分配性能：新内存的分配是快、慢还是无法预测？

压缩：当堆内存里还有小块碎片化的内存可用时，回收器是否仍然抛出内存不足（OOM）的错误？如果不是，那么你是否发现程序越来越慢，并最终死掉，尽管仍然还有足够的内存可用？

- 并发：回收器是如何利用多核机器的？
- 伸缩：当堆内存变大时，回收器该如何工作？
- 调优：回收器的默认使用或在进行调优时，它的配置有多复杂？
- 预热时间：回收算法是否会根据已发生的行为进行自我调节？如果是，需要多长时间？
- 页释放：回收算法会把未使用的内存释放回给操作系统吗？如果会，会在什么时候发生？
- 可移植性：回收器是否能够在提供了较弱内存一致性保证的CPU架构上运行？
- 兼容性：回收器可以跟哪些编程语言和编译器一起工作？它可以跟那些并非为GC设计的编程语言一起工作吗，比如C++？它要求对编译器作出改动吗？如果是，那么是否意味着改变GC算法就需要对程序和依赖项进行重新编译？

可见，在设计垃圾回收器时需要考虑很多不同的因素，而且它们中有些还会影响到平台生态系统的整体设计。我甚至都不敢确定以上给出的列表是否包含了所有因素。

设计领域的工作相当复杂，垃圾回收作为计算机科学的一个子领域，人们对它有着广泛的理论研究。学院派和软件行业会时不时地提出新的算法和它们的实现。不过，目前还没有人可以找出一种可以应付所有场景的算法。

折衷无处不在

让我们说得更具体一点。

第一批垃圾回收算法是为单核机器和小内存程序而设计的。那个时候，CPU 和内存价格昂贵，而且用户没有太多的要求，即使有明显的停顿也没有关系。这个时期的算法设计更注重最小化回收器对 CPU 和堆内存的开销。也就是说，除非内存不足，否则 GC 什么事也不做。而当内存不足时，程序会被暂停，堆空间会被标记并清除，部分内存会被尽快释放出来。

这类回收器很古老，不过它们也有一些优势——它们很简单，而且在空闲时不会拖慢你的程序，也不会造成额外的内存开销。像 Boehm GC 这种守旧的回收器，它甚至不要求你对编译器和编程语言做任何改动！这种回收器很适合用在桌面应用里，因为桌面应用的堆内存一般不会很大。比如虚幻游戏引擎，它会在内存里存放数据文件，但不会被扫描到。

计算机专业课程经常把会造成停顿（STW）的标记并清除 GC 算法作为授课内容。在工作面试时，有时候我会问候选人一些 GC 相关的问题，他们要么把 GC 看成一个黑盒，要么对 GC 一窍不通，要么认为现今仍然在使用这种老旧的技术。

标记并清除算法存在的最大问题是它的伸缩性很差。在增加 CPU 核数并加大堆空间之后，这种算法几乎无法工作。不过有时候你的堆空间不会很大，而且停顿时间可以接受。那么在这种情况下，你或许可以继续使用这种算法，毕竟它不会造成额外的开销。

反过来说，或许你的一台机器就有数百 G 的堆空间和几十核的 CPU，这些服务器可能被用来处理金融市场的交易，或者运行搜索引擎，停顿时间对于你来说很敏感。在这种情况下，你或许希望使用一种能够在后台进行垃圾回收，并带来更短停顿时间的算法，尽管它会拖慢你的程序。

不过事情并不会像看上去的那么简单！在这种配置高端的服务器上可能运行着大批量的作业，因为它们是非交互性的，所以停顿时间对于你来说无关紧要，你只关心它们总的运行时间。在这种情况下，你最好使用一种可以最大化吞吐量的算法，尽量提高有效工作时间和回收时间之间的比率。

问题是，根本不存在十全十美的算法。没有任何一个语言运行时能够知道你的程序到底是一个批处理作业系统还是一个对延迟敏感的交互型应用。这也就是为什么存在“GC 调优”——并不是我们的运行时工程师无所作为。这也反映了我们在计算机科学领域的能力是有限的。

分代理论假说

从 1984 年以来，人们就已知道大部分的内存对象“朝生夕灭”，它们在分配到内存不久之后就被作为垃圾回收。这就是分代理论假说的基础，它是整个软件产品线领域最贴合实际的发现。数十年来，在软件行业，这个现象在各种编程语言上表现出惊人的一致性，不管是函数式编程语言、命令式编程语言、没有值类型的编程语言，还是有值类型的编程语言。

这个现象的发现是很有意义的，我们可以基于这个现象改进 GC 算法的设计。新的分代回收器比旧的标记并清除回收器有很多改进：

- GC 吞吐量：它们可以更快地回收更多的垃圾。
- 分配内存的性能：分配新内存时不再需要从堆里搜索可用空间，因此内存分配变得很自由。
- 程序的吞吐量：分配的内存空间几乎相互邻接，对缓存的利用有显著的改进。分代回收器要求程序在运行时要做一些额外的工作，不过这点开销完全可以被缓存的改进所带来的好处抵消掉。
- 停顿时间：大多数时候（不是所有）停顿时间变得更短。

不过分代回收器也引入了一些缺点：

- 兼容性：分代回收器需要在内存里移动对象，在某些情况下，当程序对指针进行写入时还需要做一些额外的工作。也就是说，GC 必须跟编译器紧紧地绑定在一起，这也就是为什么 C++ 里没有分代回收器。
- 堆内存开销：分代回收器通过在内存空间里移动对象实现垃圾回收。这个要求有额外的空间用来拷贝对象，所以这些回收器会带来一些堆内存开销。另外，它们需要维护指针映射表，从而带来更大的开销。
- 停顿时间分布：尽管大部分 GC 停顿时间都很短，不过有一些仍然要求

在整个堆内进行彻底的标记并清除操作。

- 调优：分代回收器引入了“年轻代”，或者叫“eden空间”，程序性能对这块区域的大小非常敏感。
- 预热时间：为了解决上述的调优问题，有一些回收器根据程序的运行情况来决定年轻代的大小，而如果是这样的话，那么GC的停顿时间就取决于程序的运行时间长短。在实际当中，只要不是作为基准，这个算不上什么大问题。

因为分代算法的优点，现代垃圾回收器基本上都是基于分代算法。如果你能承受得起，就会想用它们，而一般来说你很可能会这样。分代回收器可以加入其它各种特性，一个现代回收器将会集并发、并行、压缩和分代于一身。

Go 的并发回收器

Go 是一种命令式的值类型编程语言，它的内存访问模式跟 C# 类似。C# 是基于分代理论假说的，所以很自然地，.NET 使用的是分代回收器。

实际上，Go 程序经常被用来处理请求和响应，就像 HTTP 服务器一样。也就是说，Go 程序表现出了十足的分代行为。Go 团队正在发掘一种他们称之为“面向请求的回收器”的东西。据悉，它很可能是一种重新命名过的分代回收器，只不过加入了经过调整的分代策略。

在其它语言运行时上可以模拟这种回收器，特别是那些请求和响应类型的处理器，只要确保年轻代大到可以容下请求所生成的垃圾。

不过，目前 Go 的回收器并不是分代的，它在后台运行的仍然是老旧的标记并清除回收器。

Go 这样做有一个好处——它的停顿时间非常短，不过除此之外，其它方面会变得很糟糕。比如说呢？

- GC吞吐量：清理堆内存垃圾所需要的时间随着堆的大小而伸缩。简单地说，程序使用越多的内存，内存就释放得越慢，你的电脑因此需要花更多的时间在垃圾回收上。除非你的程序完全不进行并行处理，你的CPU核数可以无限制地让给GC使用。

- **压缩：**因为没有压缩，堆空间最终会发生碎片化。后面我会讲到堆的碎片化问题。因为碎片化，你将无法从缓存使用中获得任何好处。
- **程序吞吐量：**因为GC每次需要做很多工作，会抢占程序的CPU时间，从而拖慢程序。
- **停顿时间分布：**任何并发的垃圾回收器都会遇到Java世界的“并发模式故障”问题：程序制造垃圾的速度超过了GC线程清理垃圾的速度。在这种情况下，运行时只能暂停程序，等待当前GC结束。所以说，Go虽然宣称它们的GC停顿时间很短暂，但这个说法只有在GC有足够CPU时间的情况下才能成立。另外，Go的编译器不具备可靠暂停线程的特性，这意味着停顿时间的长短很大程度上取决于程序的代码（例如，在Go子程序里对大型二进制对象进行BASE64解码会让停顿时间变长）。

堆内存开销：通过标记并清除的方式来回收堆空间速度很慢，所以你需要额外的空间来确保不会出现“并发模式故障”问题。Go 默认使用 100% 的堆内存开销……也就是说，你的程序需要双倍的内存来运行。

我们可以从 `golang-dev` 上的一些帖子中看到 Go 在这方面做出的折衷，比如：

`Service 1` 比 `Service 2` 分配了更多的内存，所以停顿更加频繁。不过两个服务每次停顿的时间下降了一个数量级。我们可以看到，在对两个服务进行切换以后，CPU 的使用增长了大约 20%。

Go 让停顿时间下降了一个数量级，但却是以更慢的垃圾回收为代价。这是一种更好的折衷吗？或者说停顿时间已经足够好了吗？这些问题在帖子里并没有得到回答。

不过在这种情况下，通过增加更多的硬件换取更短的停顿时间已经毫无意义。如果你的服务器停顿时间从 10 毫秒下降到 1 毫秒，你的用户会感觉得到吗？如果你需要为此投入双倍的机器，你还会这么做吗？

Go 不断优化停顿时间以便保证 GC 的吞吐量，它似乎不惜以拖慢程序的速度为代价，哪怕可以缩短一点点的停顿时间。

与 Java 的比较

HotSpot 虚拟机提供了几种 GC 算法，可以通过命令行来选择使用哪一种算法。这些算法的停顿时间不像 Go 所宣称的那么短，毕竟它们要在各个因素间做出平衡。通过比较不同的算法可以对它们有一个直观的感受。重启程序可以切换 GC 算法，因为编译工作在程序运行之时就已完成，不同算法所需要的各种屏障可以根据具体需要被添加到编译的代码里。

现代计算机使用的默认算法是吞吐量回收算法。这种算法是为批处理作业而设计的，默认情况下不对停顿时间做任何限制（不过可以通过命令行指定）。跟这种默认行为相比较，人们会觉得 Java 的 GC 简直有点糟糕了：默认情况下，Java 试图让你的程序运行尽可能的快，使用尽可能少的内存，但停顿时间却很长。

如果你很在意停顿时间，或许可以使用并发标记并清除回收器（CMS）。这种回收器跟 Go 的回收器最为接近。不过 CMS 也是分代回收器，所以它的停顿时间仍然会比 Go 的要长一些：在年轻代被压缩时，程序会被暂停，因为回收器需要移动对象。CMS 里有两种停顿，第一种是快速的停顿，可能会持续 2 到 5 毫秒，第二种可能会超过 20 毫秒。CMS 是自适应的，因为它的并发性，它需要预测何时需要启动垃圾回收（类似 Go）。你需要对 Go 的堆内存开销进行配置，而 CMS 会在运行时自适应调整，避免发生并发模式故障。不过 CMS 仍然使用标记并清除策略，堆内存仍然会出现碎片化，所以还是会出现问题，程序还是会被拖慢。

最新的 Java 回收器叫作“G1”（Garbage First）。在 Java 8 和 Java 9 里，它都是默认的回收器。它是一种“一刀切”的回收算法，它会尽量满足我们的各种需求。它几乎集并发、分代和堆空间压缩于一身。它在很大程度上可以自我调节，不过因为它无法知道你的真正需求（这个跟其它所有的回收器一样），所以你仍然可以对它做出折衷：告诉它可用的最大内存和预期的停顿时间（以毫秒为单位），它会通过自我调节来达到预期的停顿时间。

默认的预期停顿时间是 100 毫秒，只有指定了更低的预期停顿时间才能看到更好的效果：G1 会优先考虑程序的运行速度，而不是停顿时间。不过停顿

时间并非一直保持一致，大部分情况下都非常短（少于 1 毫秒），在压缩堆空间时停顿会长一些（超过 50 毫秒）。G1 具有良好的伸缩性，有人在 TB 级别的堆上使用过 G1。G1 还有一些很有意思的特性，比如堆内字符串去重。

Red Hat 开发了一种新的算法 Shenandoah，并把它贡献给 OpenJDK，不过它并不会被用在 Java 9 里，除非自己从 Red Hat 编译一个特别版的 Java。不管堆有多大，该算法都会保证很短的停顿时间，同时可以对堆空间进行压缩。

这种算法会使用额外的堆内存和更多的屏障：在程序运行的同时在堆内移动对象，并维护指针的读写操作。在这方面，它跟 Azul 的“无停顿”回收器有点类似。

结论

这篇文章的目的并不在于说服你使用某种语言或工具。只是希望你能意识到，垃圾回收是一个很复杂的问题，而且相当复杂，一大堆计算机科学家已经为此研究了数十年。如果有任何所谓的突破性进展，一定要谨慎看待。它们可能看起来很奇怪，或者更像是对折衷的伪装，到最后只会暴露无遗。

不过如果你愿意牺牲其它方面来获得最小化的停顿时间，那么可以看看 Go 的 GC。

大容量高并发分布式呼叫中心架构设计

作者 张修路 刘小桃



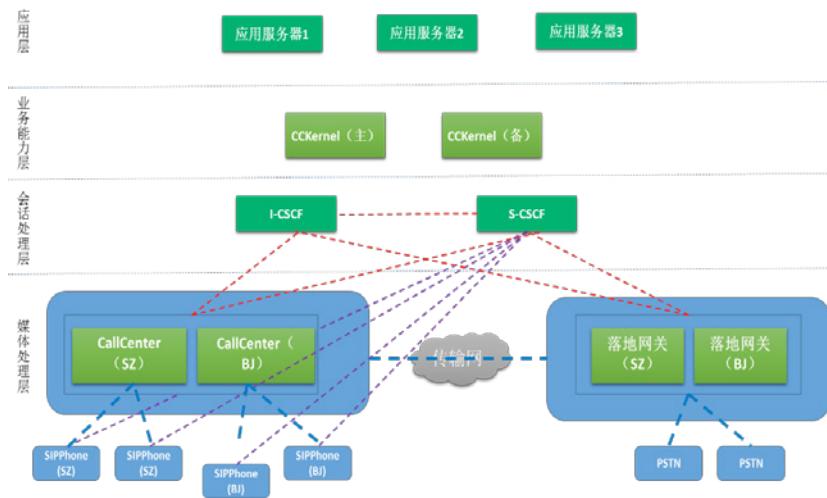
传统呼叫中心集成复杂，成本高，建设周期长。随着互联网以及 IT 的发展，呼叫中心走向云化，云呼叫中心的需求是分布式部署，用户可以就近快速接入，快速集成；架构的核心需求是：

1. 如何支持分布式的网关和分布式坐席；
2. 如何支持大容量高并发的业务场景；
3. 云部署的呼叫中心如何确保可靠性和安全性。

本文我们将通过对云之讯云呼叫中心架构分析，介绍我们设计云呼叫中心的一些经验。

(一) 云分布式呼叫中心及平台架构介绍

1. 云呼叫中心的场景



系统说明

- I-CSCF: 负责处理与落地网关的呼叫流程与信令管理。
- S-CSCF: 负责处理用户侧的信令，包括用户注册、心跳管理和用户侧的呼叫管理。
- Callcenter: 负责呼叫过程中的媒体转发、录音、IVR放音等功能。

CCkernel: 负责呼叫中心的排队管理、用户状态等核心功能

部署说明

- 呼叫中心系统在各地部署POP接入点并通过专线连接。
- 使用呼叫中心的企业用户就近接入分布式POP点，避免语音在Internet传输影响语音质量。
- 企业用户使用的PSTN网关就近接入POP点，以节约长途话费。

设计理念

- 业务与控制分离，信令与媒体分离，提升网络的灵活性和可靠性，通过服务化拆分和微服务的方式，构建分布式大容量的可靠性解决方案。

- 关键业务部件如CCkernel、I-CSCF、S-CSCF等通过主备方式提升可靠性，并支持异地容灾。
- 媒体处理节点Callcenter节点支持分布式集群部署，部署在各地POP点，Callcenter出现故障，I-CSCF和S-CSCF能够自动发现，并通过SIP信令引导呼叫接续到其它Callcenter节点。

(二) 技术挑战与架构设计

基于分布式的呼叫中心设计目标，针对呼叫中心的面临的挑战，我们提供了如下的解决方案。

1. 智能路由

由于设计目标需要企业用户就近接入，落地网关也需要就近接入，业务各POP点通过专线连接，就需要实现智能化的选路方案，以确保用户通过最优的路由接入。

- 各企业用户登录中心节点，中心节点根据用户的IP地址判断运营商和位置，指引该用户就近接入POP点，所在POP点出现故障，则指引用户接入备份POP点
- 落地网关通过固定配置就近接入主用POP点，主用POP故障，自动切换到备份POP点
- 各POP点通过专线连接，通过OSPF方式实现自动选路，并监控相关路由，确保最佳选路

2. 通话接续速度

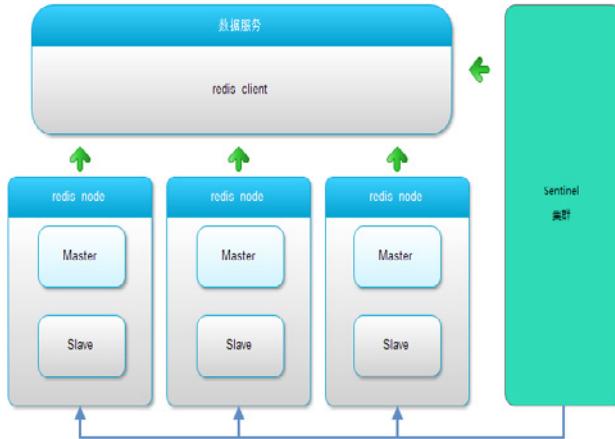
通话接续速度是影响用户体验的第一要素，通过架构实现了分布式数据库和内存数据库并进行算法优化，把数据缓存到离 Callcenter 最近的内存数据，以确保数据查询速度，尽快转接，同时优化路由算法，做到最短选路，在整个呼叫中心内部，最多只需要经过 2 个节点就可以进入 PSTN 网络。

3. 语音质量

1. 路由管理：在用户侧部署监控模块，实施监控到主用POP节点和备份POP节点的丢包和延迟情况，主用POP节点出现网络异常，下一次通话

切换到备份POP节点

- 语音编码：根据实施监测的网络情况，以及用户话机和落地网关的编解码能力，动态选择最优编解码方案。



(三) 关键技术设计

1. 内存数据库Redis设计

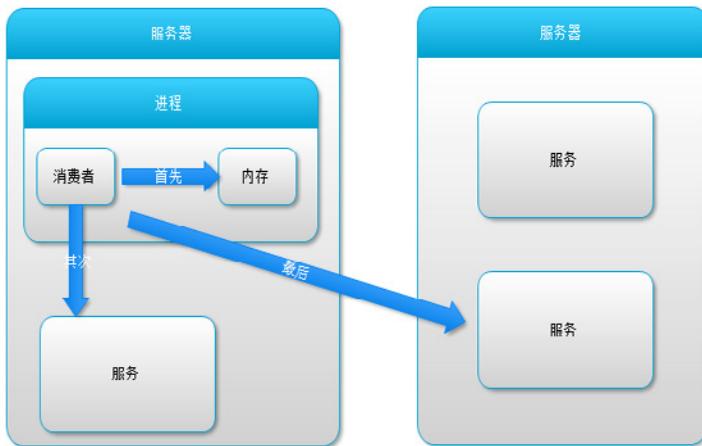
整个呼叫过程中，涉及大量的路由策略，每一次通过都需要多次数据库操作，为提升呼叫速度；同时，关键部件如CCkernel、I-CSCF、S-CSCF等部件都通过双机或者集群提升可靠性，为了保证双机、集群切换过程中，呼叫不受影响，我们采用了内存数据库Redis。

- CCkernel把所有排队、相关组件信息保存在Redis中，CCkernel切换过程，整个用户数据不损失，业务逻辑不受损。
- I-CSCF和S-CSCF把所有呼叫相关的信息保存在Redis中，I-CSCF和S-CSCF切换，已有呼叫连接的数据均在Redis中，呼叫不会受到影响。

2. 微服务路由设计

呼叫中心系统需要支持高并发的大容量呼叫，同时要支持复杂的路由策略，为了提升接续速度，减少部件交互，我们对路由呼叫处理进行分级处理，热点数据缓存在进程中，部分数据存在在本机的缓存内，对于低频访问数据和复杂业务逻辑进行统一管理，根据路由策略，进行分级查询。

(四) 系统可靠性设计理念



- 信令和业务处理节点CCkernel、I-CSCF、S-CSCF通过双机实现备份，同时支持异地部署和容灾。
- I-CSCF出现故障，业务可以自动切换到异地的I-CSCF，落地网关通过SIP option监测到故障，切换到异地的I-CSCF。
- S-CSCF出现故障，业务可以自动切换到异地的S-CSCF，通过DNS指引用户切换到异地S-CSCF。
- 媒体处理节点集群部署，I-CSCF和S-CSCF实时监控，通过呼叫信令指引用户切换到备份节点。

(五) 安全性

系统安全

- 公网关闭所有非必要服务端口，及时更新版本和补丁。
- 实现完善的认证与鉴权机制，防止非法用户登录。

业务安全

- 通过呼叫中心系统实时监测各用户呼叫情况，动态分析呼叫数据，发现可疑呼叫立刻通知管理人员进行人工介入。
- 录音监控系统：可以按照客户或者某一路呼叫进行录音，录音快速回传给客户进行，人工进行分析。

智 能 时 代 的 大 前 端

GMTC 2017 GLOBAL MOBILE TECH CONFERENCE

全 球 移 动 技 术 大 会

2017.6.9-10 / 北京·国际会议中心

12大专题大公布 HOT

- Native动态化专场
- 新技术专场
- 性能优化专场
- Web框架实践专场
- 大前端专场
- 移动Web优化专场
- 新平台专场
- 移动架构专场
- 工程化专场
- 移动AI专场
- 开源实践专场
- 解决方案专场

3月26日前购票低至**2160**元/张 (全价3600元/张) 团购更加优惠

票务咨询: 18618231445 / alfred@infoq.com

> 扫码了解更多





主办方 Geekbang · InfoQ
极客邦科技

2017年7月7日－7月8日
—
深圳·华侨城洲际酒店

7 折购票(截至3月5日)
立减2040

19大专题，全新出炉

- ◆ 区块链以及金融新技术
- ◆ 推荐系统架构实践
- ◆ 低延迟系统架构设计
- ◆ 移动以及轻应用
- ◆ 创新的智能应用
- ◆ 机器学习架构
- ◆ 大规模存储系统
- ◆ 大数据框架
- ◆ 架构师成长路线
- ◆ 研发团队建设和工程文化
- ◆ 基于微服务的软件新架构
- ◆ 社交网络与视频直播
- ◆ 运维新挑战
- ◆ 云架构新动态
- ◆ 大规模企业级性能优化
- ◆ 安全之战
- ◆ 电商之核心架构
- ◆ 技术创业
- ◆ 研发工具

咨询: 010-89880682 / 18515221946

Q Q: 2332883546

微信: 497788321

进入官网了解详情
archsummit.com





本期主要内容：OpenAPI 规范 3.0 版接近最终发布；谈谈技术选型；2016 年 JavaScript 领域中最受欢迎的“明星”们；服务拆分与架构演进；左耳朵耗子：我对 GitLab 误删除数据库事件的几点思考



解读2016

许多年后，如果我们回过头来评点，也许 2016 年是非常重要的一个时间节点。



顶尖技术团队访谈录 第七季

本次的《中国顶尖技术团队访谈录》·第七季。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。



架构师特刊 机器学习实践

趁着近年来深度学习的热潮强势复苏，机器学习很快地从一个很少被大众关注的技术主题，转变为被很多人使用的管理工具和开发工具。