架构师

ARCHITECT

特刊

ArchSummit

北京2018

SPECIAL ISSUE

Geekbang. 极客邦科技 | InfoQ

卷首语 架构师和业务的结合

ArchSummit组委会

最近我们在读《奈飞文化手册》这本书，它的作者是流媒体巨头Netflix（奈飞）前首席人才官帕蒂.麦考德（Patty McCord），书里提到了8条文化准则，分别是：只招成年人；要让每个人都理解公司业务；绝对坦诚；只有事实才能捍卫观点；现在组建未来需要的团队；员工和岗位要高度匹配；按照员工带来的价值付薪；离开时要好好说再见。

要让每个人都理解公司业务是很有价值的事情，对架构师来说更甚。架构师理解业务的好处，一是可以搭建一个全面的架构平台；二是可以更快速的找出需要debug的问题点。从技术选型，到架构选型，从业务建模到系统建模，架构师无一不是在做着决策。除此之外，架构师还应该是好的产品经理。技术只是手段，关键是要能做出用户喜欢的好产品。因为业务流程、业务规则、业务细节等都是变量，只有懂业务，才不至于因一个细小的业务规则的改变而成为压死骆驼的最后一根稻草。

架构师需要足够的技术的宽度，从软件到硬件，从开发到测试，从运维到安全等都需要面面俱到的了解。系统在未来的运行过程中需要运维，功能需要迭代，初次开发并交付只是它生命周期中的一小部分而已，后期的维护、改造、升级都是根据业务需求而推进的，架构设计者，也赋予了架构灵魂。所以，深入了解业务发展方向的架构师另一个使命是：系统未来的设计。

总结

- 架构师应该是让产品经理有更多的选择而不是约束。
- 架构最好是和商业愿景（收益）对齐。
- 理想中的架构应该像AppStore那样，设计成一个容器。
- 架构：将产品、技术、运营有机的结合起来。

极客时间VIP年卡

帮你打造高效能研发团队

企业VIP年卡是什么?

帮你打造高效能研发团队，365天畅看极客时间所有专栏、视频和微课。

内容覆盖架构、运维、前端、测试、人工智能等领域，一次性搞定团队全年福利及学习计划。

同时也可根据研发团队需求，提供定制化课程和学习计划，和数万人共同学习和成长。

你和团队可获得

- 365天畅看极客时间全部专栏、视频和微课（不含每日一课）
- 开发、运维、测试、架构、技术管理等全领域覆盖
- 20 余类技术硬技能，掌握一线技术实战应用
- 碎片化时间构建体系化思维，提升团队效率

全部课程365天任性看

包括年度所有专栏、视频课程、精品微课

数据结构与算法之美 | 量身打造的数据结构与算法私教课
王争 | 前Google工程师
（33134人已加入学习）

从 0 开始学架构 | 资深技术专家的实战架构心法
李运华 | 资深技术专家
（27962人已加入学习）

Java 核心技术 36 讲 | Oracle首席工程师带你修炼Java内功
杨晓峰 | Oracle首席工程师
（26957人已加入学习）

趣谈网络协议 | 像小说一样的网络协议入门课
刘超 | 网易研究院云计算技术部首席架构师
（21061人已加入学习）

持续交付 36 讲 | 量身定制你的持续交付体系
王潇俊 | 携程系统研发部总监
（4742人已加入学习）

深度拆解 Java 虚拟机 Oracle高级研究员手把手带你入门JVM
郑雨迪 | Oracle 高级研究员，计算机博士
（16392人已加入学习）

赵成的运维体系管理课 | 带你直击运维的本质
赵成 | 美丽联合集团技术服务经理
（3634人已加入学习）

深入剖析 Kubernetes | Kubernetes原来可以如此简单
张磊 | Kubernetes社区资深成员与项目维护者
（11078人已加入学习）

注：以上为极客时间部分课程，订阅数据以 App 为准

Geekbang. 极客邦科技

和数万技术人一起学习
为我的研发团队采购

目录

架构师特刊

本期主编 徐 川

流程编辑 丁晓昀

发行人 霍泰稳

ArchSummit全球架构师峰会是InfoQ中国团队推出的重点面向高端技术管理者、架构师的技术会议，50%参会者拥有8年以上工作经验。ArchSummit聚焦业界强大的技术成果，秉承“实践第一、案例为主”的原则，展示先进技术在行业中的最佳实践，以及技术在企业转型、发展中的推动作用。旨在帮助技术管理者、CTO、架构师做好技术选型、技术团队组建与管理，并确立技术对于产品和业务的关键作用。

Geekbang.
极客邦科技

极客邦企业培训与咨询

帮助企业和技术人成长

业务简介

凭借实践驱动的海量国内外一线专家资源，及对技术领域的洞察和深厚知识积累，为企业客户提供研发团队能力提升解决方案，同时为企业提供针对具体技术挑战的技术咨询服务，帮助企业提升技术竞争壁垒，真正让技术驱动业务发展。

- 海量技术讲师/专家资源
- 技术领域专业洞察和知识积累
- 专业完善的企业培训咨询服务流程

产品与服务类型

深度培训　企业内训　技术咨询　企业账号

提供9大技术领域的培训与咨询

云计算　大数据　软件架构　移动与前端　语言开发　运维与容器　人工智能　区块链　技术管理

明星讲师

李文哲

人工智能专家
美国贪心科技创始人兼CEO

人工智能 | 深度学习 | 大数据

陈靓

鹏云网络创始人
“千人计划”国家特聘专家
AWS架构师

云计算 | 架构

孙玄

转转公司架构算法部负责人
前58集团技术委员会主席

架构 | 大数据 | 机器学习

企业服务顾问：Rodin（罗丹）
咨询电话：15002200534　邮箱rodin@geekbang.org

扫码申请企业服务

破茧化蝶
中邮消费金融系统服务化和容器化实践

作者 李远鑫

消费金融是近年来新兴的热门行业，市场广阔、需求量大、行业竞争激烈。为了突破传统金融行业限制，建立一套灵活、高效、可靠的消费金融系统，支持互联网环境下高并发访问、可靠资金交易、交互式场景化的需求，中邮消费金融从基于集中式商业中间件搭建的传统信贷系统，演进为基于微服务、分布式、灵活快速迭代的互联网消费金融系统。

服务化演进

开业初期，为了能够迅速把 IT 系统搭建起来，支撑公司的业务开展，我们基于集中式的商业中间件构建了一套传统的金融信贷系统，涵盖了贷前、贷中和贷后管理，采用的是烟囱式架构，属于传统的交易型 IT 系统，这样的架构在业务高速发展的过程中显得开发迭代更新效率低下、创新困难、难以扩展，形成数据孤岛。为此，我们采用了“大中台、小前端”的中台战略，将应用架构设计为渠道、应用、共享服务中心和基础技术平台四个层次，其中共享服务中心按照业务领域来划分，尽量将原有的业务功能进行解耦，沉淀出公共部分，采用微服务的方式进行构建，整个系统架构也由原来的集中式架构转变为分布式的架构，不仅提升了系统的整体性能和可扩展性，也使新业务和新产品的开发效率有了明显的提升。本次主题的第一部分将分享中邮消费金融从烟囱式系统向多个共享服务中心演进的服务化实践。

分布式技术平台的搭建

共享服务中心建立在分布式技术平台之上，主要包括服务调用和治理框架、消息中间件、缓存、分布式文件系统、分布式数据库等。这里重点介绍服务调用、消息中间件和应用配置中心的搭建情况以及分布式事务处理的方案。

（1）**在服务调用框架方面**，我们早期对 DubboX 进行了裁剪和优化封装，作为主要的微服务开发技术，后续逐渐以 Springboot+Jersey 提供 Restful 服务、SpringCloud 替代，阿里巴巴恢复对 Dubbo 的支持，我们也计划将部分原来通过 DubboX 开发的服务迁移到目前的 Dubbo 版本。同时基于企业内部微服务体系的逐步庞大，服务的有效治理也成为了我们逐步关注的重点，通过对比和研究业界的成熟方案，以及结合我们自身的系统设计。我们最终选择了 Zookeeper 作为我们服务治理中心，即基于 Zookeeper 的 Dubbo RPC 服务与基于 Spring Cloud Zookeeper 的 http 服务管理并存的服务结构。

随着应用越来越多，系统之间的调用关系也越来越复杂，对服务的调用进行全链路跟踪和监控非常有必要。对此，我们通过 Java Agent、Byteman、OpenTracing 等技术构建了自动的日志监控埋点，基本不需要应用程序改动即可将日志埋点嵌入，从而形成完整的微服务调用链路，再通过日志搜索和图形化展示，可以实现对微服务的有效监控。

（2）**消息中间件的选择**。消费金融的特点是前端进件的业务量和并发量较大，但是大部分交易不需要强实时的响应。为了把大部分服务解耦，提升系统的吞吐率和性能，需要实用消息中间件技术。根据消费金融的特点及公司业务定位，要求消息队列必须是高可用，高性能且自主可控的，从社区的活跃度、消息堆积能力和性能来看，Kafka 更适合于我们的场景。另外，大数据处理和运维监控大部分消息队列也需要用到 Kafka，为了统一消息队列标准，便于维护，我们最终选择了 Kafka 作为消息中间件。

Kafka 在操作系统崩溃时可能会存在消息丢失的情况，当然在 Kafka1.0 版本以后，得到了比较大的优化。早期，为了保证消息的可靠投递，我们基于 Spring Kafka 幂等发送和支持事务等特性确保消息发送成功，自实现了消息多线程处理及 Offset 管理，引入 Redis 缓存 Offset，支持应用异常时从 Redis 恢复消息进行重新处理，实现 At-least-once 消费语义。

在 Springboot 2.0 发布后，消息消费通过 Spring Cloud Stream 支持 Partition 数据应用实例数及每个实例的并发数自动创建扩容，实现对每个 Partition 消息的单线程、幂等处理及整体上的多线程并发处理，Offset 通过 Spring Kafka 自动管理，实现 At-least-once 消费语义。另外很重要的一点，为了确保 At-least-once，消费端必须实现幂等性。通过优化和有效的异常处理机制，我们的系统没有出现消息丢失的情况，而且具备较高的吞吐率，达到了预期的效果。

（3）**消息医院**。使用 Kafka 消息中间件的服务必须要考虑消费端应用异常时如何进行处理，我们早期的做法是扩展 Spring Kafka，当消息消费出现异常时，将异常消息统一送进一个异常处理 topic，由各应用自己监听异常 topic 并进行相应的处理，例如重试、丢弃等。为了使消息的异常处理形成统一标准，简化异常处理，减少重复开发，我们自研实现了一个消息医院（名称来源于 Sam Newman 的《微服务设计》，也称为死信队列），将所有异常消息都发送到消息医院中，通过一个管理端界面来查看异常消息列表，设计一定的异常处理策略（如定时自动触发一定次数的重试、手工重试或直接丢弃等），作为 at- least-once 消息投递保证的必要措施。

（4）**分布式事务处理**。分布式事务处理的常见解决办法是两阶段提交 2PC、补偿事务、TCC、

基于消息的可靠事件机制。消费金融的大部分场景不需要保证强一致性，只需要保证最终一致性即可，TCC 模型过于复杂，而且可能会存在读脏数据或者中间事务状态不一致的问题，鉴于消费金融的事务大部分复杂度不高，并且可以通过场景化和多交互步骤的方式避免分布式事务，我们主要选择了可靠事件机制，通过消息的可靠投递实现最终一致性。

分布式事务处理是一项复杂的研究课题，目前没有非常完美的解决方案。分布式事务处理的关键，在于进行系统设计时，首先分析清楚强事务是否是真正的需求（例如转账），如果强事务不是真正需求，则应该通过场景化设计来尽量避免产生分布式事务。由于微服务实施后，无法通过场景化弱化事务的情况下（例如跨共享服务中心的多服务调用），应避免采用两阶段提交，尽量采用基于消息的可靠事件机制保证最终一致性。对于跨越内部系统和外部系统的分布式事务（如放款记账和银联代付），目前没有很好的解决方案，只能采取补偿事务、幂等重试的方式解决，在进行系统设计时，应遵循良好的设计原 则，确保补偿机制和重试机制足够合理和健壮。

（5）应用配置中心。在微服务架构中，服务之间有着错综复杂的依赖关系，每个服务都有自己的依赖配置，在运行期间很多配置会根据各种因素进行动态调整，传统的配置信息处理方式是将配置信息写入 xml、.properties 等配置文件中，和应用一起打包，每次修改配置信息，都需要重新进行打包，效率极低；同时在分布式应用场景下，配置的批量更新，灰度发布，集群管理、设置回滚、审计日志等开发与运维需求也对配置信息的管理提出了更高的要求。因此，我们开发了一个适用于各个应用系统、实现配置集中管理、支持信息实时更新等功能的配置应用中心。应用配置管理系统的使用，不仅使得配置信息能更统一、更有效的集中管理，同时也降低应用系统代码的耦合，使得开发、测试人员以一种更灵活的方式对环境、场景进行管理。

微服务集成及容器化演进

随着共享服务中心和微服务的不断扩充和完善，应用层与微服务之间、微服务之间的调用关系会面临变得越来越复杂的情况，如果不加以管控，会变成蜘蛛网式的调用，应用程序开发也会变得越来越复杂。为此，我们引入了微服务统一运营平台的概念，目的是将微服务在线化和可视化，对微服务进行集中管理，通过容器技术提供微服务快速组合和扩展的能力，达到业务流程的快速落地和调整，实现业务的高效拓展，另外，平台还可以对微服务组合出来的应用程序进行全面的监控和数据稽核。

首先是服务的在线化。将各共享服务中心提供的服务注册到统一运营平台的服务注册中心，注册中心提供一个可视化的界面，业务和技术人员可以方便地检索现有的共享服务，查看其用途、接口规范和目前正在使用该服务的应用。我们主要以 CDC 的模式（Consumer-Driven Contracts，消费方驱动的契约开发方式，是指从消费者业务实现的角度出发，驱动出契约，再基于契约，对提供者验证的一种测试开发方式）进行了服务接口的规范化约束，确保了开发流程的高效性和正确性。

其次是服务的集成。传统的服务组合和集成有两种方式：编排和协同，其中编排的方式一般是通过 ESB 或工作流引擎进行统一配置和管理，有明确的流程，由一个中心大脑控制；协同是事件驱动的方式，每个系统收到信号后启动处理，职责分明，细节独立，系统模块间的耦合度低。我们选择了基于事件驱动的协同方案，首先通过将现有的微服务以代理服务的方式注册到 SpringCloud DataFlow，并以可视化拖拽的方式对各种代理服务进行流程设计，通过服务间的组合串联来完成业务流程。串联的微服务之间的数据传递是通过事件驱动的模式进行，即通过 kafka

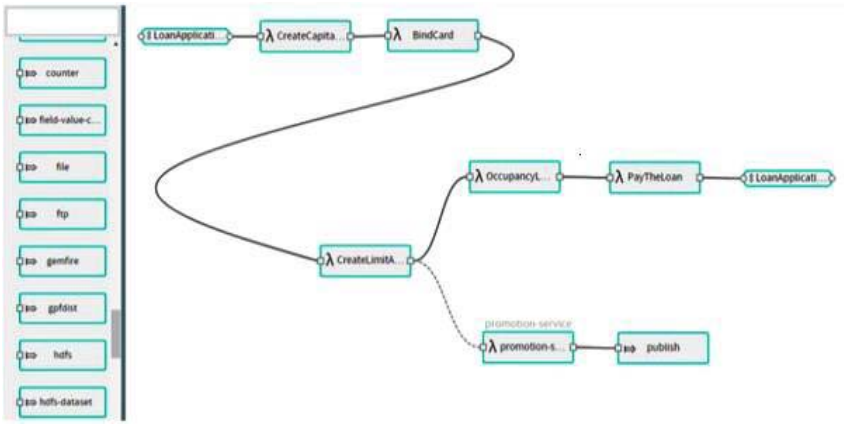
异步消息的模式，并通过消息医院和事后补偿机制进一步确保事务的最终一致性。

接下来是容器化。在系统应用微服务化的背景下，运用 Docker 技术栈将应用容器化成为了我们的目标。容器技术为实现更高效的资源利用，更快速的交付和部署，更轻松的迁移与扩展和更简单的更新管理提供了可能，同时也更适应快速变化的市场和业务需求。在应用容器化进程中，我们运用 Docker 技术对不同的微服务应用打包成相应的镜像，并将 Docker 镜像发布到私有镜像仓库统一管理。同时运用 Kubernetes 对容器和微服务进行编排和管理。

在服务集成运行环境方面，前面提到的服务在线化，向业务和技术人员展示的实际上就是集成 Docker 镜像仓库形成的微服务应用商店，通过可视化界面实现对微服务的参数动态展现和进行设计配置，Kubernetes 会根据设计过程生成的配置文件到 Docker 仓库获取相应镜像，部署到容器节点中，形成所需的服务，利用 Docker 和 Kubernetes 的特性，服务可以实现高可用、自动负载均衡、失效重建以及自动扩展。

最后，金融系统追求的是账务的准确性，需要对组合服务开发的应用进行业务稽核。业务稽核模块以业务为维度登记业务及其检验规则，实现统一应用系统的抽样数据收集，对抽样数据进行样本聚合，并根据设置的检验规则进行校对和检验，对校验异常的数据进行预警，并提供统一的监控界面。

本部分会以其中一个真实具体的业务模型为例介绍服务集成的实例，预览如下图所示。



李远鑫，中邮消费金融有限公司IT 运营部 总经理助理 & 架构师。曾参与中国邮政储蓄银行全国中间业务平台、储蓄系统逻辑大集中等大型金融项目，曾任中邮消费金融公司软件研发高级总监，现任 IT 运营部总经理助理、架构师，负责中邮消费金融公司等 IT 系统规划和架构设计，并带领团队完成从基于集中式商业中间件的金融交易系统向灵活、可靠、高性能、分布式的微服务消费金融交易系统的演进，在 2017 年支付宝 66 信用日活动中支撑日均 10 万的交易量和 30 万 PV 峰值。

Sharding-Sphere 成长记
写在分布式数据库代理端里程碑版本 3.0.0 发布之际

作者 张 亮

在历经八个月的紧张开发与精心打磨之后，Sharding-Sphere 社区为程序员献礼，将 Sharding-Sphere 3.0.0 正式版于 10 月 24 日程序员节发布。在 3.0.0 发布之际，写下此文，与大家共同回顾这段充满纪念的时光，分享我们的前进历程。

前序

关注开源圈的同学可能知道，Sharding-Sphere 的前身是 Sharding-JDBC。

起源

Sharding-JDBC 是一套扩展于 Java JDBC 层的分库分表中间件，最初起源于当当的内部应用框架 ddframe 中的数据库访问层组件。由于分库分表需求的相对普遍，并且具备独特的生命力与关注度，因此将其抽离成为独立的项目，命名为 Sharding-JDBC，并于 2016 年初开源。

Sharding-JDBC 的最初目标是透明化分库分表所带来的复杂度，包括数据源的管理、根据业务进行的 SQL 改写等。作为使用 Java 语言开发的 ddframe 框架中的一部分，Sharding-JDBC 顺其自然的选择了 JDBC 作为其分库分表扩展点的接入端。正如其名称 Sharding-JDBC 所昭示，它是在 JDBC 层进行 Sharding（分库分表）的产品。

核心功能完善

Sharding-JDBC 在其后的一年中有条不紊的发布了 1.x 的 6 个大版本更新，分别是：

- 奠定了 SQL 解析、请求路由、SQL 改写、SQL 执行和结果归并的分库分表的核心模型的 1.0.x；
- 原生支持 Spring 和行表达式的 1.1.x；
- 最大努力送达型柔性事务的 1.2.x；
- 读写分离的 1.3.x；
- 分布式主键的 1.4.x；
- 全新 SQL 解析引擎的 1.5.x。

分布式治理

在分库分表功能逐渐成熟之后，在 2017 年，Sharding-JDBC 进入了 2.x 时代。2.x 主要实现的功能是数据库治理，它可以通过注册中心提供对配置的集中化和动态化，以及对数据库和应用进行禁用和熔断。在此基础上，还增加了面向 OpenTracing 协议的链路追踪能力，并且达成了与国内优秀的 APM 产品 Apache SkyWalking 的合作协议，将 Sharding-JDBC 的追踪数据对接入 SkyWalking，并让 SkyWalking 将采用 Sharding-JDBC 作为其存储引擎成为可选项。

至此，分库分表、分布式事务和数据库治理都有了简单的雏形。

发展

随着云原生的普及，应用上云和对异构语言的无差别支持渐渐成为当今主流。仅支持 Java 的 Sharding-JDBC 已经无法满足云原生的全部需要，在业界一直争论不休的在客户端（JDBC 或其他语言客户端）还是服务端（Proxy）进行分片的优劣，也未有定论。

改名、之后再踏征途

2018 年春节前夕，随着核心开发人员的加盟，京东数科（当时还叫京东金融）加入了 Sharding-JDBC 的开发工作中，并将其定位为面向云化的数据库中间件。在客户端进行分库分表的 Sharding-JDBC，虽然可以作为轻量级微服务框架灵活应用，但却没有作为云接入端进行统一管控的能力。因此，一个 Proxy 接入端呼之欲出。

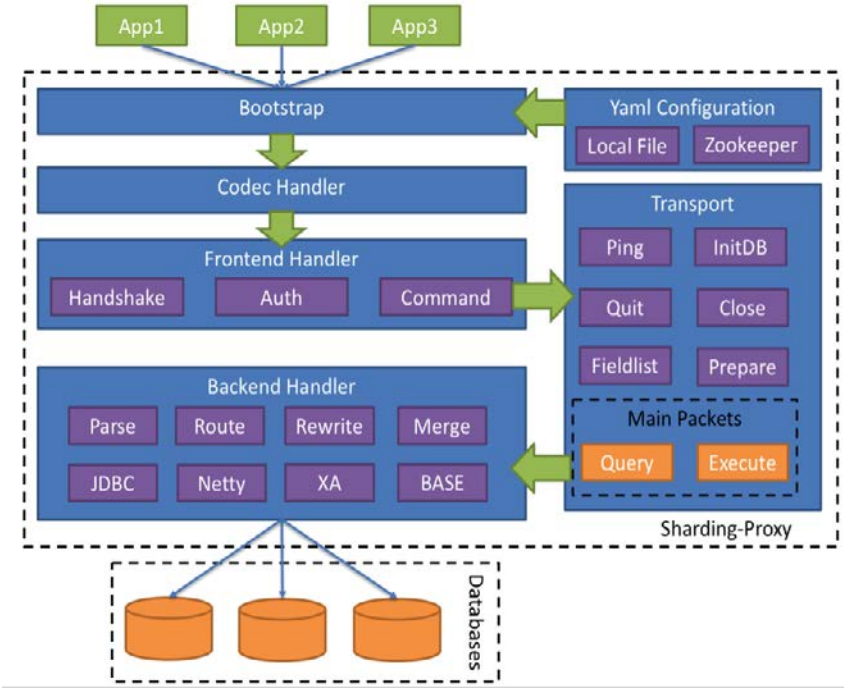
Sharding-JDBC 这个名字在过去的两年中获得了大量的积累，已经具备一定的辨识度，开发团队并不希望完全放弃掉这个名字。因此，最初将新的代理端产品命名为 Sharding-JDBC-Server，而将原有的 Sharding-JDBC 改名为 Sharding-JDBC-Driver。

经过了反复的权衡，我们发起了社区投票。最终决定保留 Sharding 这个关键词，将项目的名称正式改为 Sharding-Sphere，意为分片生态圈。无论是分布式事务还是多数据库的治理，其本源都是分片；若采用单一的无分片数据库，后续功能都将无需存在。分片生态圈由根据不同的接入端，由 3 个子项目组成，它们是基于 JDBC 客户端接入的 Sharding-JDBC（即原有项目）、基于代理端接入的 Sharding-Proxy（今年的重点更新）、以及基于 Sidecar 模式接入的 Sharding-Sidecar（明年的产品规划）。

3.0.0 于此刻正式起航，主要目标是将 Sharding-JDBC 的能力完全移植入 Sharding-Proxy，使其具备支持异构语言的能力。虽然分片的核心逻辑并未变化，但相比于 Sharding-JDBC，Sharding-Proxy 有两个难点是需要攻破的。

第一个难点是数据库协议的实现。将代理端伪装成为一个数据库，能够将接入的成本降至最低。Sharding-Proxy 选择最常用的 MySQL 协议做为首先支持的数据库协议，并完整的实现了所有的应用程序运行时所需的协议包（如：COM_QUERY、COM_STMT_PREPARE、COM_STMT_EXECUTE）。目前对于管理端使用的一些协议包还未全部实现。

第二个难点是通信框架。JDBC 层的通信是由各个数据库驱动提供商通过 BIO 的方式实现的，虽然吞吐量欠佳，但却容易实现。代理端为了更高的吞吐量，需要采用 NIO 的方式。Sharding-Proxy 采用 Netty 作为通信框架，在接入层前端实现了完全无锁的异步通信。目前接入端连接后端数据库时，仍然采用 JDBC 的方式，未来会将其完全改为 Netty 异步通信的方式，进一步提升吞吐量，达成前后端完全无锁通信的目标。以下是 Sharding-Proxy 的架构图：



在 2018 年 5 月，基本可用的 Sharding-Proxy 随着 Sharding-Sphere 3.0.0.M1 发布。

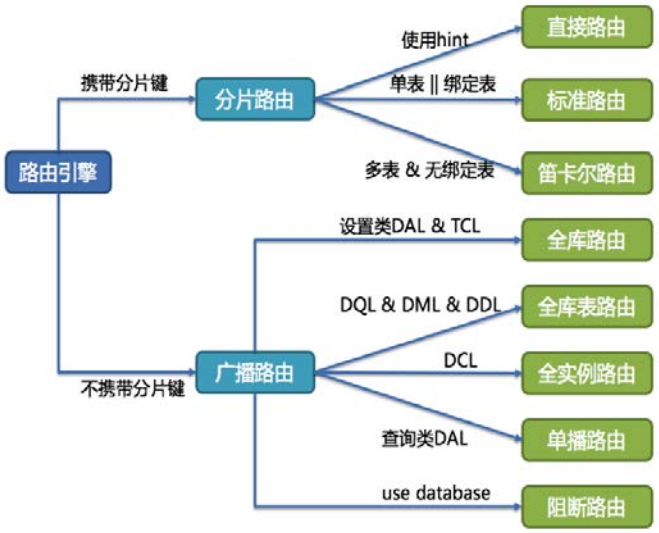
同时，由于多家公司共同参与开发，Sharding-Sphere 决定成立社区，将著作权完全归属至 Sharding-Sphere 社区，并成立了项目管理委员会（PMC），并且也完善了贡献者和提交者的晋升制度。

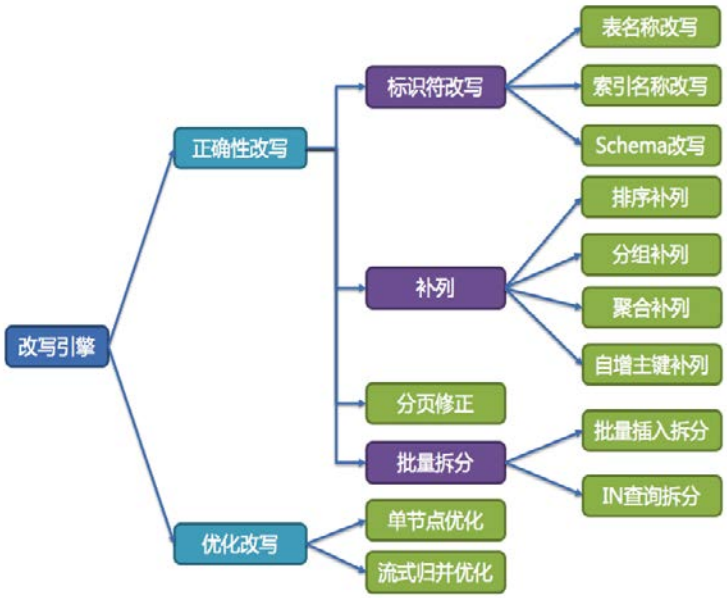
随着新的里程碑版本，Sharding-Sphere 申请了全新的域名，并重新制作官网，重装发布。

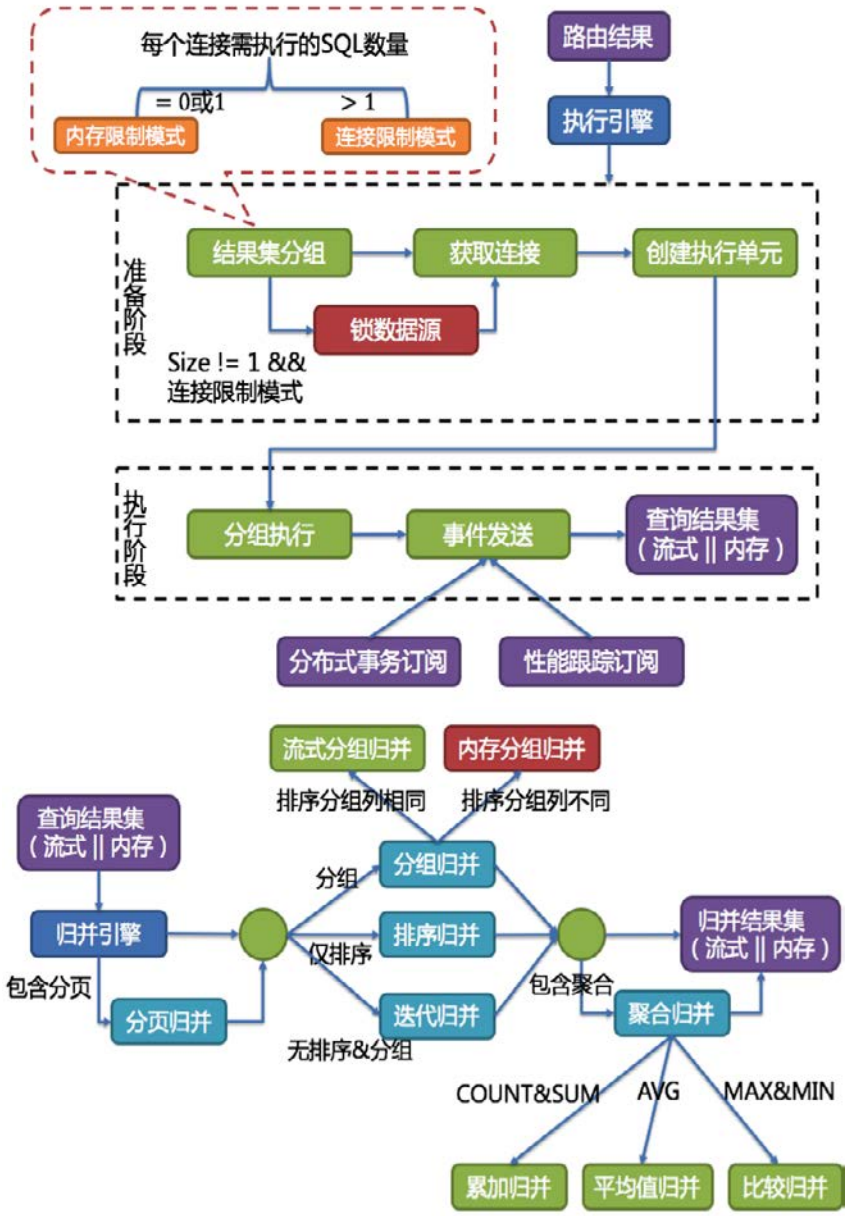
扩大范围、加强合作

Sharding-Sphere 的更名，不仅仅是接入端的增强。作为分片生态圈，更完善的分布式事务和数据库治理，也纳入了项目范围。

Sharding-Sphere 将原有的分库分表功能更名为数据分片，内容包扩核心流程、读写分离和分布式主键。Sharding-Sphere 的核心流程模块的几个重点部分可以通过一张图帮助用户理解，下面分别是路由引擎、改写引擎、执行引擎和归并引擎的剖析图。



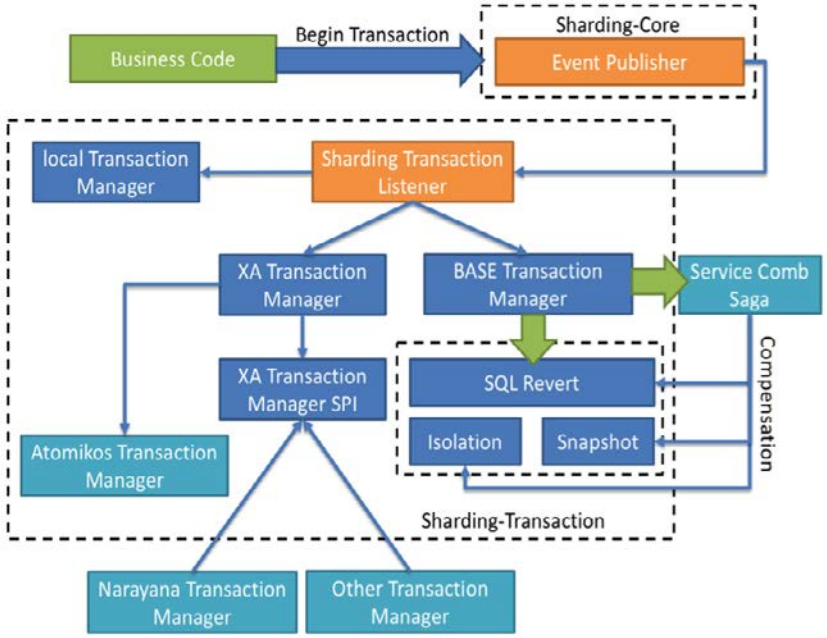




Sharding-Sphere 对分布式事务进行了重新的设计和定位。废弃掉原有的最大努力送达型柔性事务，取而代之的是采取刚柔并济的实现方案：同时支持 XA 的强一致事务，以及基于 Saga 的最终一致性事务，基于消息的最终一致性事务也在规划中。

分布式事务模块将定位从自研转向整合，即整合现有的成熟事务方案，为本地事务、XA 事务和柔性事务提供统一的分布式事务接口，并尽量弥补各个方案对数据库层面的缺失。分布式事务模块提供一

套 SPI 事务处理接口，能够无缝对接分布式事务的各个实现方案。分布式事务模块的架构图如下。



Sharding-Sphere 经过比较分析，选择采用Apache ServiceComb 的分布式事务解决方案来实现柔性事务， 通过在 ServiceComb Saga 执行引擎基础上扩展 SQL 执行模块，实现了基于分布式 Saga 的事务执行和回滚功能。

分布式事务模块将于 3.1.0 的版本发布，目前仍处于紧张的开发阶段。

在数据库治理方面，Sharding-Sphere 全数保留了之前的功能，并提供了全新的 APM 链路追踪数据，可以通过 SkyWalking 更直观的观测 Sharding-Sphere。但目前仍未包括数据库弹性扩缩功能，该部分功能将于明年规划。

在高速发展的同时，Sharding-Sphere 迎来了新的合作伙伴——翼支付。翼支付成立了创新中心部门，并投入开发资源加入到了 Sharding-Sphere 的开发团队。这使得 Sharding-Sphere 的开源社区更加多元化和健康成长。Sharding-Sphere 属于社区而非公司，因此欢迎有兴趣参与开发的公司一起打造更加多元化的社区和更加完善的项目。

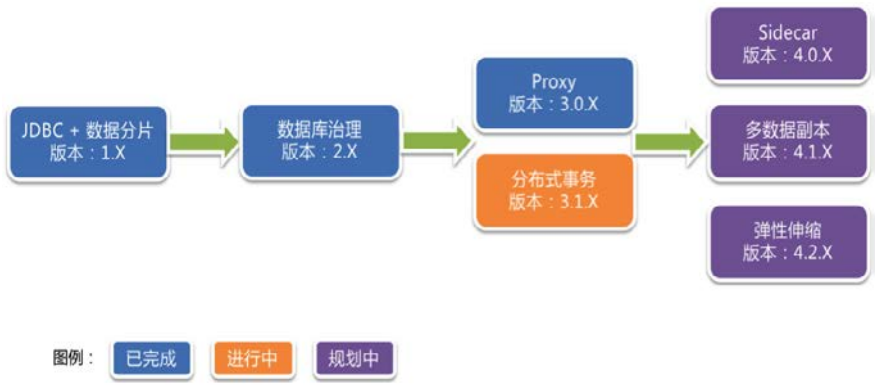
上线、然后发布

在 Sharding-Sphere 的旗下产品 Sharding-Proxy 逐渐成熟的同时，京东数科当仁不让的成为了第一个吃螃蟹的人。京东数科将部分核心业务系统通过小流量 =» 大流量 =» 全流量的流程切换到 Sharding-Proxy，目前 Sharding-Proxy 在生产环境中已经管理并运行着万级别数据节点。

在经受考验后，随之而来的 Sharding-Sphere 3.0.0.M2、3.0.0.M3 和 3.0.0.M4 相继发布。在经历了大量的性能调优和功能完善之后，终于在 10 月 24 日的程序员节发布 3.0.0 稳定版。在经历了京东数科严酷的生产环境验证后，相信 Sharding-Sphere 可以成为架构师们进行技术选型时的其中一个参考。

面向未来

Sharding-Sphere 3.0.0 的发布并非终点，而是新的起点。3.1.0 已经在同步开发，也将于不久的将来面世，提供更加优化的分布式事务解决方案。计划于明年开启的 4.0.0 对 Sidecar 模式的接入端以及自动化的弹性伸缩功能也完成了初步规划。Sharding-Sphere 的线路规划如下图。



大事记

回顾心路历程，Sharding-Sphere 立足于当下，着眼于未来。

2018.2

- Sharding-Sphere 团队升级组建，并开始着手 Sharding-Proxy 开发。

2018.5

- Sharding-JDBC 正式更名为 Sharding-Sphere, 同时上线新官网。这预示着它新时代的到来。
- Sharding-Sphere 著作版权完全归属社区 shardingsphere.io，并继续使用 Apache 2.0 协议。
- Sharding-Sphere 3.0.0.M1 发布，Sharding-Proxy 正式上线。

2018.6

- Sharding-Sphere 与 Apache ServiceComb 建立合作伙伴关系，并开始分布式事务的全面规划。
- Sharding-Sphere 与中国电信旗下翼支付建立合作伙伴关系，共同打造 Sharding-Sphere 新未来。

2018.8

- Sharding-Proxy 上线京东数科生产环境，并经受住了线上大规模生产数据的考验。
- Sharding-Sphere 3.0.0.M2 发布，数据库治理模块升级改造，提供更稳定功能。

2018.9

- Sharding-Sphere 3.0.0.M3 发布，提供对 XA 分布式事务的支持。

- Sharding-Sphere 3.0.0.M4 发布， 改造自动化执行引擎，支持多逻辑数据库切换，增强链路追踪。

2018.10

- Sharding-Sphere 3.0.0 正式版发布。

如何获取

Sharding-JDBC

```
<groupId>io.shardingsphere</groupId>
<artifactId>sharding-jdbc-core</artifactId>
<version>3.0.0</version>
```

Sharding-Proxy

```
docker pull shardingsphere/sharding-proxy
```

源码

https://github.com/sharding-sphere/sharding-sphere

https://gitee.com/sharding-sphere/sharding-sphere

官网

http://shardingsphere.io

OLAP 引擎这么多，麻袋财富为什么选择用 Kylin 做自助分析？

麻袋财富大数据平台

麻袋财富（原麻袋理财）成立于 2014 年 12 月底，是中信产业基金控股的网络借贷信息中介平台，经过 4 年平稳而快速的发展，截至目前，累计交易金额达 750 亿，已成为行业头部平台。庞大的业务量带来了数据量指数级增长，原有的数据分析处理方式已远远不能满足业务的需求。

- 流程耗时长：逻辑比较复杂的数据需求，可能会涉及到开发，产品经理，BI 等多方相关人员，通过反复的沟通，确认才能完成，而涉及人员过多增加了沟通成本，拉长项目周期。
- 资源浪费：为了促进平台的销量增长，运营会设计各种产品促销或用户促活的短期活动， 每次活动后都会进行复盘，没有产品化的活动分析通常会导致分析人员的人力浪费。
- 集群压力大：一些需要长期监测的复杂指标，每天都要进行重复的查询，而且每天都有几百个临时 SQL 提交到集群中处理，造成集群计算压力大，影响集群性能。
- 查询慢：随着数据量越来越大，往往一条聚合 SQL 需要几分钟才能出结果，数据分析师需要的快速响应要求已远远不能满足。

针对上述痛点，我们希望寻找一个工具能够给用户提供高效、稳定、便利的数据分析性能。

为什么选择 Kylin

我们调研了市面上主流的 OLAP 引擎，对比详情如下所示。

结合公司业务需求：

- 以 T+1 离线数据为主；
- 可以整合 Tableau 使用，实现自助分析；
- 常用 30 个左右的维度，100 个左右的指标，任意交叉组合，覆盖 80%+ 的固定和临时需求；
- 业务方需要观察用户从进入到离开的整个生命周期的特征，涉及数据量大，但要快速响应。

我们选择了 Kylin 作为 OLAP 分析引擎，原因如下：

	基于大数据集群的 OLAP 引擎			基于传统数据库的 OLAP 引擎		
	Kylin	Kudu	Presto	SSAS	Essbase	Cognos
技术	预计算	列式存储	大规模并行处理	Cube	Cube	Cube
安装部署	安装快捷方便，轻量级	部署简单	安装简单	安装方便	安装困难，安装后的配置麻烦	安装轻便快捷
系统易用性	简单，选择表，设置维度和度量即可构建 Cube	需要开发，门槛较高	基于内存，查询代价大	拖拽式，简单	复杂	提供建模工具，构建 Cube 效率很高。
支持标准 SQL	支持	结合 Impala 提供	支持	不支持	不支持	不支持
MDX 门槛支持	第三方插件支持	不支持	不支持	支持	支持	支持
部署监控成本	组件熟悉，运维使用成本低，有公共 web 界面	已集成到 CDH 中，便于监控	有公共 web 界面监控管理	利用SSDT开发部署，成本高	使用 OPMN 启动和停止 Essbase 服务器	有自己的 client 和 Web Explorer
社区	Apache 顶级项目（中国）	Cloudera	Facebook	微软	Oracle	IBM

- Kylin 使用预计算，以空间换时间，能够实现用户查询请求秒级响应；
- 可以结合现有 BI 工具——Tableau，实现自助分析；
- 本来需要耗时一周的需求在几分钟内出结果，开发效率提升了 10 倍以上。

本文主要介绍麻袋财富基于 CDH 大数据平台的自助分析项目实施，如何将 Apache Kylin 应用到实际场景中，目前的使用现状以及未来准备在 Kylin 上做的工作。

技术架构

系统部署方面，主要分生产环境和预上线环境，生产环境主要负责查询分析，从生产集群 Hive 上跑计算，把预计算结果存储到 HBase。如果想新增一个 Cube 的话，需要分析人员先在预上线环境上操作，再由专人对 Cube 进行优化后迁移到生产环境。

麻袋财富的自助分析架构如右图所示。

- 数据同步：Sqoop（离线场景）、Kafka（近实时场景）
- 数据源：Hive（离线场景）、Streaming Table（近实时场景）
- 计算引擎：MapReduce/Spark
- 预计算结果存储：HBase
- 自助分析工具：Tableau
- 调度系统：Azkaban

Apache Kylin 的解决方案

公司业务非常复杂，数据团队将各种业务需求高度抽象，确定好维度和度量，只需构建一个 Cube，基于该 Cube，形成通用的平台化数据产品，解放数据分析师，降低重复性工作。

Kylin 的离线构建

1. 数据建模

数据建模对 Kylin 实施来说是最重要的工作。一般使用关系数据库模型中的星型模型，但是现实中由于业务的多样性，维表的基数很大，所以一般我们把表处理成宽表并且基于宽表建 OLAP 模型，宽表不仅能解决数据模型的数据粒度问题，还能解决多表 join 的性能问题，以及维度变化、或者超高基数维度等问题。

各个业务线不同的数据特点和业务特点决定了 Kylin 的使用场景及模型设计优化方式。

- 数据规模和模型特点。从数据规模上来讲，宽表的数据量近百亿，每天的增量数据千万级以上。我们根据业务指标通过 OLAP 建模的高度抽象分析，定义了维度和度量值的关系以及底层数据粒度。
- 维度基数特点。维度基数最理想的情况是相对较小，但实际上有些维度超过了百万级接近千万级，这和业务需求及行业特点有关。除此之外指标上要用部分维度之间的笛卡尔积组合，造成很难简化 OLAP 模型，生成数据相应的开销也比较大。
- 维度粒度特点。从维度的角度来看，地域维度包含省份和城市；时间维度上需要一级划分年月日，增加了维度的复杂度。
- 指标也是维度特点。有一些指标既是维度也是度量，例如：我们需要分析在投金额为 0 的用户的行为，又需要计算用户的在投金额，所以在投金额即为维度又是度量。

2. Kylin Cube 设计

从 Kylin Cube 模型上来说，由于 Cube 需要满足多种场景的需求，业务上需要多个维度互相灵活组合，与分析人员反复沟通，最终确认 Cube 的维度及度量。

Cube 模型概况：

- 19 个维度：包括省市、操作系统、设备型号、性别、绑卡状态、投资等级等。
- 10 个度量：包括数据量、访问次数、登录用户数、浏览量、投资金额、年化金额等。
- 增量构建：某一 Cube 源数据增量为 300 万，Build 完一天数据 Cube 大小为 87.79GB。

3. Cube 设计的优化

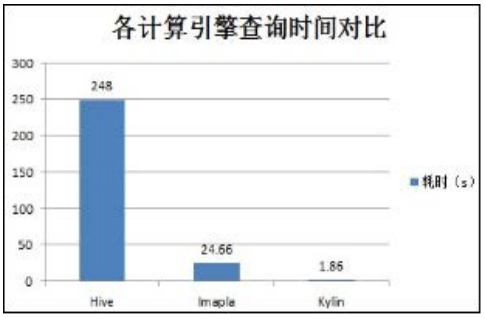
Cube Build 过程中常见遇到的是性能问题，例如 SQL 查询过慢、Cube 构建时间过长甚至失败、 Cube 膨胀率过高等等。究其原因，大多数问题都是由于 Cube 设计不当造成的。因此，合理地进行 Cube 优化就显得尤为重要。

优化方案：

- 维度精简：去除查询中不会出现的维度，如数据创建日期。
- 强制维度：把每次查询都需要的维度设为强制维度（Mandatory Dimensions）。
- 层次维度： 把有层次的维度（省市或年月日）设为层次维度（Hierarchy Dimensions）。
- 联合维度：把用户关心的维度组合设成联合维度（Joint Dimensions）。
- 调整聚合组：设置多个聚合组，每个聚合组内设置多组联合维度。不会同时在查询中出现的维度分别包含在不同聚合组。
- 调整 Rowkeys 排序，对于基数高的维度，若在这个维度上有过滤、查询，则放在前面，常用的维度放在前面。

4. Cube 优化成果

根据上面的优化方案， 把 assist_date 和 source 设为强制维度，把 province，city 设为层次维度，再根据使用频率和基数高低排序，最终的优化成果如下。



- 查询性能：秒级响应。
- 构建时间：缩短 31%。
- Cube 大小：减小 42%。
- 查询性能详情：业务明细表：10 亿。
- SQL 语句：求每个城市的在投金额。

Kylin 实时增量构建

为了减低 OLAP 分析的延时，在 Kylin 中添加 Streaming Table 实现准实时分析的功能，Kylin 以 Streaming Table 为数据源，Streaming Table 消费 Kafka 中的数据。模型中多增加一个 timestamp 类型的字段，用作时间序列。在实践过程中，模型优化了如下参数：

```
kylin.Cube.algorithm=inmem
kylin.Cube.algorithm.inmem-concurrent-threads=8
kylin.Cube.max-building-segments=600
```

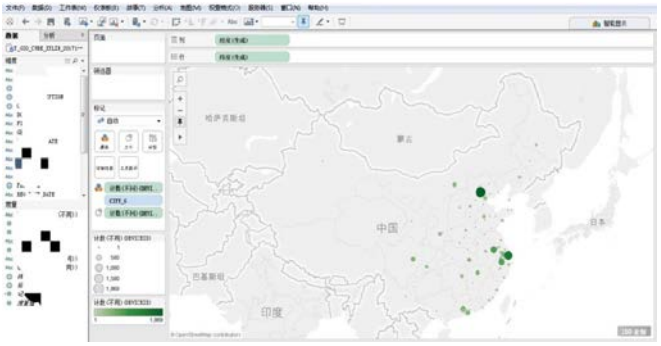
Kylin 整合 Tableau

公司采用 Kylin 2.4.0 版本和 Tableau 9.0 版本, 在前者提供预计算结果的前提下, 希望结合 Tableau 能够给数据分析师提供更方便、快速的数据自助分析。

在本机上安装与 Tableau 版本对应的 Kylin ODBC Drive，Tableau 连接 Kylin 时选择 Kylin 的 ODBC Driver，然后选择 Kylin 的数据源 Fact Table 和 Join Table，并按 Kylin Cube 模型 join 起来，就可以实现拖拉出结果的即席查询，上钻、下钻、旋转等目标。分析人员摆脱了编写冗长 SQL，漫长等待的过程，可以根据自己的需求进行数据分析。其中一个使用场景如下图所示，展示每个地区的活跃人数。

实施中的经验总结

1. Tableau 拖拉维度出结果慢



解决：查看 kylin.log，发现耗时最长的都是 select * from fact，所以让这条 SQL 尽可能快的失败，可以修改 kylin.properties 的参数：

```
kylin.query.max-scan-bytes 设置为更小的值
kylin.storage.partitin.max-scan-bytes 设置成更小的值
```

2. Kylin 整合 Tableau 创建的计算字段一定是 Cube 中包含的，若 Cube 中没有包含该计算字段，那么在 Tableau 中计算会显示通信错误，因为 Cube 的预计算中不含此值。

3. 使用实时增量时报错：

解决：这是由于 Kylin 2.4.0 版本和 Kafka 的 3.0.0 版本不匹配，把 Kylin 降了一个版本 Kylin 2.3.2 即可。

4. 字段类型转换：在 double 类型的数据转换为 String 时，会自动转换为保留一位小数的字符串，例如 112 转换成了 112.0，导致 join 的时候无法 join 成功。解决：当我们要转换的数值只有整形没有小数时，我们可以先把数值类型转换成 bigint 类型，使用 bigint 类型存储的数值不会采用科学计数法表示。

5. 空值产生的数据倾斜：行为表中对游客的 user_id 是置空的，如果取其中的 user_id 和 用户表中的 user_id 关联，会碰到数据倾斜的问题。解决：把空值的 user_id 变成一个字符串加上随机数，把倾斜的数据分到不同的 reduce 上，由于 null 值关联不上，处理后并不影响最终结果。

6. Kylin ODBC Driver 安装是根据 Tableau 版本的，不是根据操作系统而定的。例如，Windows 版本是 64 位的，Tableau 版本是 32 位，就需要装 32 位的 ODBC。

未来规划

Kylin 给我们带来了很多便利，节约了查询时间和精力。随着技术的不断进步，还有许多问题需要解决，还需要不断探索和优化，例如 Kylin 对明细数据的查询支持不理想，但是有时需要查询明细数据；删除 Cube，HBase 中的表不会自动删除，影响查询性能，需要手动清理等。

Apache Beam 实战指南：玩转 KafkaIO 与 Flink

作者 张海涛

一. 概述

大数据发展趋势从普通的大数据，发展成 AI 大数据，再到下一代号称万亿市场的 IOT 大数据。技术也随着时代的变化而变化，从 Hadoop 的批处理，到 Spark Streaming，以及流批处理的 Flink 的出现，整个大数据架构也在逐渐演化。

Apache Beam 作为新生技术，在这个时代会扮演什么样的角色，跟 Flink 之间的关系是怎样的？Apache Beam 和 Flink 的结合会给大数据开发者或架构师们带来哪些意想不到的惊喜呢？

二. 大数据架构发展演进历程

2.1 大数据架构 Hadoop

最初做大数据是把一些日志或者其他信息收集后写入 Hadoop 的 HDFS 系统中，如果运营人员需要报表，则利用 Hadoop 的 MapReduce 进行计算并输出，对于一些非计算机专业的统计人员，后期可以用 Hive 进行统计输出。

2.2 流式处理 Storm

业务进一步发展，运营人员需要看到实时数据的展示或统计。例如电商网站促销的时候，用于统计用

户实时交易数据。数据收集也使用 MQ，用流式 Storm 解决这一业务需求问题。

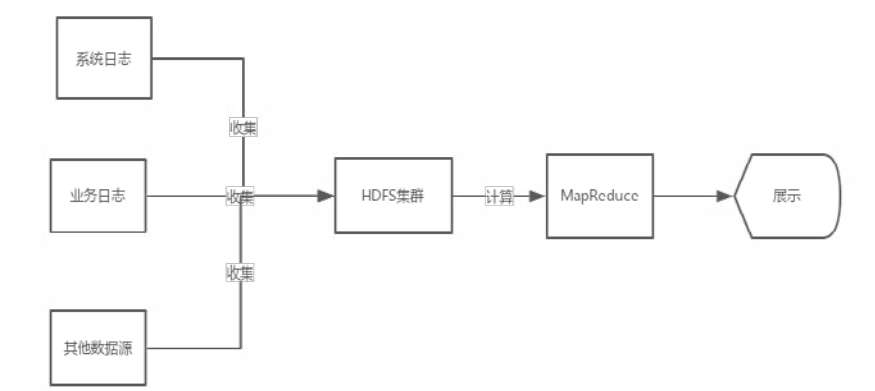


图 2-1 MapReduce 流程图

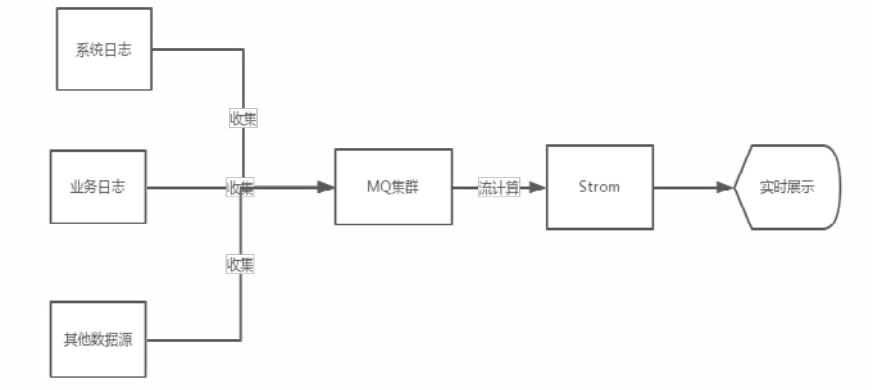


图 2-2 Storm 流程图

2.3 Spark 批处理和微批处理

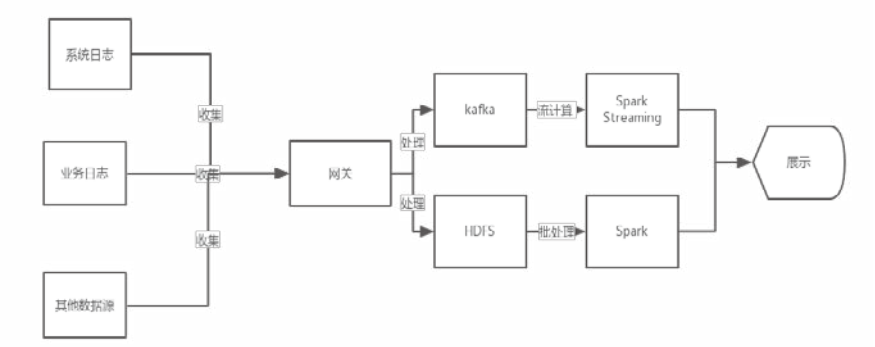


图 2-3 Spark 流程图

业务进一步发展，服务前端加上了网关进行负载均衡，消息中心也换成了高吞吐量的轻量级 MQ Kafka，数据处理渐渐从批处理发展到微批处理。

2.4 Flink：真正的流批处理统一

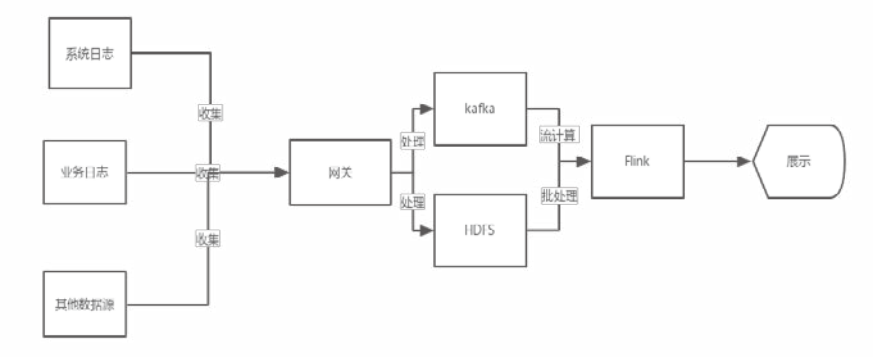


图 2-4 Flink 流程图

随着 AI 和 loT 的发展，对于传感设备的信息、报警器的警情以及视频流的数据量微批计算引擎已经满足不了业务的需求，Flink 实现真正的流处理让警情更实时。

2.5 下一代大数据处理统一标准 Apache Beam

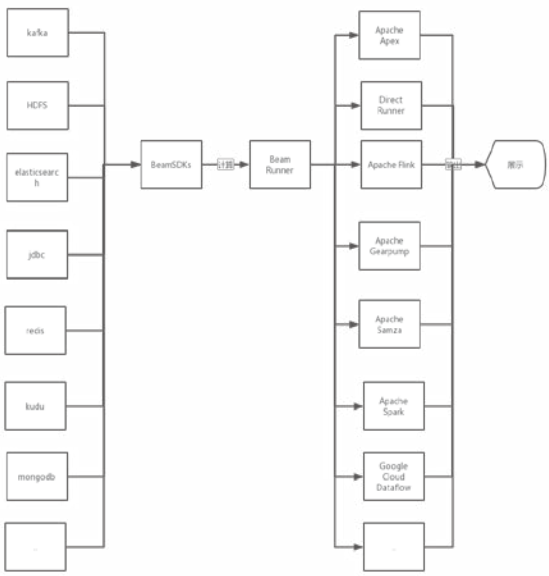


图 2-5 Apache Beam 流程图

BeamSDKs 封装了很多的组件 IO，也就是图左边这些重写的高级 API，使不同的数据源的数据流向后面的计算平台。通过将近一年的发展，Apache Beam 不光组件 IO 更加丰富了，并且计算平台在当初最基本的 Apache Apex、Direct Runner、Apache Flink、Apache Spark、Google Cloud Dataflow 之上，

又增加了 Gearpump、Samza 以及第三方的 JStorm 等计算平台。

为什么说 Apache Beam 会是大数据处理统一标准呢?

因为很多现在大型公司都在建立自己的“大中台”，建立统一的数据资源池，打通各个部门以及子公司的数据，以解决信息孤岛问题，把这些数据进行集中式管理并且进行后期的数据分析、BI、AI 以及机器学习等工作。这种情况下会出现很多数据源，例如之前用的 MySQL、MongodDB、HDFS、HBase、Solr 等，如果想建立中台就会是一件令人非常苦恼的事情，并且多计算环境更是让技术领导头疼。Apache Beam 的出现正好迎合了这个时代的新需求，它集成了很多数据库常用的数据源并把它们封装成 SDK 的 IO，开发人员没必要深入学习很多技术，只要会写 Beam 程序就可以了，大大节省了人力、时间以及成本。

三．Apache Beam 和 Flink 的关系

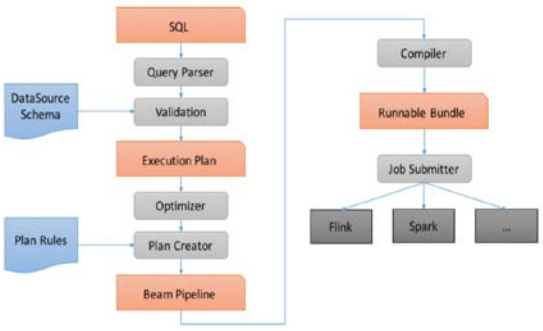
随着阿里巴巴 Blink 的开源，Flink 中国社区开始活跃起来。很多人会开始对各种计算平台进行对比，比如 Storm、Spark、JStorm、Flink 等，并且有人提到之前阿里巴巴开源的 JStorm 比 Flink 性能高出 10-15 倍，为什么阿里巴巴却转战基于 Flink 的 Blink 呢? 在最近 Flink 的线下技术会议上，阿里巴巴的人已经回答了这一问题。其实很多技术都是从业务实战出来的，随着业务的发展可能还会有更多的计算平台出现，没有必要对此过多纠结。

不过，既然大家最近讨论得这么火热，这里也列出一些最近问的比较多的、有代表性的关于 Beam 的问题，逐一进行回答。

1. Flink 支持 SQL，请问 Beam 支持吗?

现在 Beam 是支持 SQL 处理的，底层技术跟 Flink 底层处理是一样的。

Beam SQL 现在只支持 Java，底层是 Apache Calcite 的一个动态数据管理框架，用于大数据处理和一些流增强功能，它允许你自定义数据库功能。例如 Hive 使用了 Calcite 的查询优化，当然还有 Flink 解析和流 SQL 处理。Beam 在这之上添加了额外的扩展，以便轻松利用 Beam 的统一批处理 / 流模型以及对复杂数据类型的支持。 以下是 Beam SQL 具体处理流程图。



Beam SQL 一共有两个比较重要的概念。

- SqlTransform：用于 PTransforms 从 SQL 查询创建的接口。
- Row：Beam SQL 操作的元素类型。例如：PCollection<Row>。

在将 SQL 查询应用于 PCollection 之前，集合中 Row 的数据格式必须要提前指定。 一旦 Beam SQL 指定了 管道中的类型是不能再改变的。PCollection 行中字段 / 列的名称和类型由 Schema 进行关联定义。您可以使用 Schema.builder() 来创建 Schemas。

示例：

```
// Define the schema for the records.
Schema appSchema =
Schema
  .builder()
  .addInt32Field("appId")
  .addStringField("description")
  .addDateTimeField("rowtime")
  .build();
// Create a concrete row with that type.
Row row =
Row
  .withSchema(appSchema)
  .addValues(1, "Some cool app", new Date())
  .build();
// Create a source PCollection containing only that row
PCollection<Row> testApps =  PBegin
  .in(p)
  .apply(Create
  .of(row)
  .withCoder(appSchema.getRowCoder()));
```

也可以是其他类型，不是直接是 Row，利用PCollection<T>通过应用 ParDo 可以将输入记录转换为 Row 格式。如：

```
// An example POJO class.
class AppPojo {
 Integer appId;
 String description;
 Date timestamp;
}
// Acquire a collection of POJOs somehow.
PCollection<AppPojo> pojos = ...
// Convert them to Rows with the same schema as defined above via a DoFn.
PCollection<Row> apps = pojos
 .apply(
   ParDo.of(new DoFn<AppPojo, Row>() {
     @ProcessElement
     public void processElement(ProcessContext c) {
       // Get the current POJO instance
       AppPojo pojo = c.element();
```

```
        // Create a Row with the appSchema schema
          // and values from the current POJO
          Row appRow =
                Row
                  .withSchema(appSchema)
                  .addValues(
                    pojo.appId,
                    pojo.description,
                    pojo.timestamp)
                  .build();
        // Output the Row representing the current POJO
           c.output(appRow);
      }
    }));
```

2. Flink 有并行处理，Beam 有吗？

Beam 在抽象 Flink 的时候已经把这个参数抽象出来了，在 Beam Flink 源码解析中会提到。

3. 我这里有个流批混合的场景，请问 Beam 是不是支持？

这个是支持的，因为批也是一种流，是一种有界的流。Beam 结合了 Flink，Flink dataset 底层也是转换成流进行处理的。

4. Flink 流批写程序的时候和 Beam 有什么不同？底层是 Flink 还是 Beam?

打个比喻，如果 Flink 是 Lucene，那么 Beam 就是 Solr，把 Flink 的 API 进行二次重写，简化了 API，让大家使用更简单、更方便。此外，Beam 提供了更多的数据源，这是 Flink 不能比的。当然，Flink 后期可能也会往这方面发展。

四. Apache Beam KafkaIO 源码剖析

Apache Beam KafkaIO 对 kafka-clients 支持依赖情况。

KafkaIO 是 Kafka 的 API 封装，主要负责 Apache Kafka 读取和写入消息。如果想使用 KafkaIO，必须依赖 beam-sdks-java-io-kafka ，KafkaIO 同时支持多个版本的 Kafka 客户端，使用时建议用高版本的或最新的 Kafka 版本，因为使用 KafkaIO 的时候需要包含 kafka-clients 的依赖版本。

Apache Beam KafkaIO 对各个 kafka-clients 版本的支持情况如表4-1。

Apache Beam V2.1.0 版本之前源码中的 pom 文件都显式指定了特定的 0.9.0.1 版本支持，但是从 V2.1.0 版本和 V2.1.1 两个版本开始已经替换成了 kafka-clients 的 0.10.1.0 版本，并且源码中提示 0.10.1.0 版本更安全。这是因为去年 Kafka 0.10.1.0 之前的版本曝出了安全漏洞。在 V2.2.0 以后的版本中，Beam 对 API 做了调整和更新，对之前的两种版本都支持，不过需要在 pom 中引用的时候自己指定 Kafka 的版本。但是在 Beam V2.5.0 和 V2.6.0 版本，源码中添加了以下提示：

```
 * <h3>Supported Kafka Client Versions</h3>
 * KafkaIO relies on <i>kafka-clients</i> for all its interactions with the Kafka
```

```
cluster.
    * <i>kafka-clients</i> versions 0.10.1 and newer are supported at runtime. The older
versions
    * 0.9.x - 0.10.0.0 are also supported, but are deprecated and likely be removed in
near future.
    * Please ensure that the version included with the application is compatible with the
version of
    * your Kafka cluster. Kafka client usually fails to initialize with a clear error
message in
    * case of incompatibility.
    */
```

Apache Beam 版本	kafka-clients 版本
V0.4.0	0.9.0.1
V0.5.0	0.9.0.1
V0.6.0	0.9.0.1
V2.0.0	0.9.0.1
V2.1.0	0.10.1.0
V2.1.1	0.10.1.0
V2.2.0	0.9.0.1 , 0.10.1.0
V2.3.0	0.9.0.1 , 0.10.1.0
V2.4.0	0.9.0.1 , 0.10.1.0
V2.5.0	0.10.1.0+
V2.6.0	0.10.1.0+

表 4-1 KafkaIO 与 kafka-clients 依赖关系表

也就说在这两个版本已经移除了对 Kafka 客户端 0.10.1.0 以前版本的支持，旧版本还会支持，但是在以后不久就会删除。所以大家在使用的时候要注意版本的依赖关系和客户端的版本支持度。

如果想使用 KafkaIO，pom 必须要引用，版本跟 4-1 表中的对应起来就可以了。

```
<dependency>
 <groupId>org.apache.beam</groupId>
 <artifactId>beam-sdks-java-io-kafka</artifactId>
 <version>...</version>
</dependency>
 <dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-clients</artifactId>
<version>a_recent_version</version>
<scope>runtime</scope>
 </dependency>
```

KafkaIO 读写源码解析

KafkaIO 源码链接。在 KafkaIO 里面最主要的两个方法是 Kafka 的读写方法。

KafkaIO 读操作

```
pipeline.apply(KafkaIO.<Long, String>read()
    .withBootstrapServers("broker_1:9092,broker_2:9092")
    .withTopic("my_topic")
// use withTopics(List<String>) to read from multiple topics.
    .withKeyDeserializer(LongDeserializer.class)
    .withValueDeserializer(StringDeserializer.class)
 // Above four are required configuration.returns PCollection<KafkaRecord<Long, String>>
    // Rest of the settings are optional :
   // you can further customize KafkaConsumer used to read the records by   adding more
    // settings for ConsumerConfig. e.g :
    .updateConsumerProperties(ImmutableMap.of("group.id",   "my_beam_app_1"))
    // set event times and watermark based on 'LogAppendTime'. To provide   a custom
    // policy see withTimestampPolicyFactory(). withProcessingTime() is   the default.
    // Use withCreateTime() with topics that have 'CreateTime' timestamps.
     .withLogAppendTime()
    // restrict reader to committed messages on Kafka (see method   documentation).
     .withReadCommitted()
     // offset consumed by the pipeline can be committed back.
     .commitOffsetsInFinalize()
     // finally, if you don't need Kafka metadata, you can drop it.g
    .withoutMetadata() // PCollection<KV<Long, String>>
    )
    .apply(Values.<String>create()) // PCollection<String>
```

1. 指定 KafkaIO 的模型，从源码中不难看出这个地方的 KafkaIO<K,V> 类型是 Long 和 String 类型，也可以换成其他类型。

```
pipeline.apply(KafkaIO.<Long, String>read() pipeline.apply(KafkaIO.<Long, String>read()
```

2. 设置 Kafka 集群的集群地址。

```
.withBootstrapServers("broker_1:9092,broker_2:9092")
```

3.设置 Kafka 的主题类型，源码中使用了单个主题类型，如果是多个主题类型则用 withTopics (List<String>) 方法进行设置。设置情况基本跟 Kafka 原生是一样的。

```
.withTopic("my_topic") // use withTopics(List<String>) to read from multiple topics.
```

4. 设置序列化类型。Apache Beam KafkaIO 在序列化的时候做了很大的简化，例如原生 Kafka 可能要通过 Properties 类去设置，还要加上很长一段 jar 包的名字。

Beam KafkaIO 的写法：

```
    .withKeyDeserializer(LongDeserializer.class)
    .withValueDeserializer(StringDeserializer.class)
```

原生 Kafka 的设置：

```
   Properties props = new Properties();
   props.put("key.deserializer","org.apache.kafka.common.serialization.
ByteArrayDeserializer");
   props.put("value.deserializer","org.apache.kafka.common.serialization.
ByteArrayDeserializer");
```

5) 设置 Kafka 的消费者属性，这个地方还可以设置其他的属性。源码中是针对消费分组进行设置。

```
    .updateConsumerProperties(ImmutableMap.of("group.id", my_beam_app_1"))
```

6) 设置 Kafka 吞吐量的时间戳，可以是默认的，也可以自定义。

```
    .withLogAppendTime()
```

7) 相当于 Kafka 中"isolation.level", "read_committed" ，指定 KafkaConsumer 只应读取非事务性消息，或从其输入主题中提交事务性消息。流处理应用程序通常在多个读取处理写入阶段处理其数据，每个阶段使用前一阶段的输出作为其输入。通过指定 read_committed 模式，我们可以在所有阶段完成一次处理。针对"Exactly-once" 语义，支持 Kafka 0.11 版本。

```
    .withReadCommitted()
```

8) 设置 Kafka 是否自动提交属性"AUTO_COMMIT"，默认为自动提交，使用 Beam 的方法来设置。

```
    set CommitOffsetsInFinalizeEnabled(boolean commitOffsetInFinalize)
.commitOffsetsInFinalize()
```

9) 设置是否返回 Kafka 的其他数据，例如 offset 信息和分区信息，不用可以去掉。

```
    .withoutMetadata() // PCollection<KV<Long, String>>
```

10) 设置只返回 values 值，不用返回 key。例如 PCollection<String>，而不是 PCollection<Long，String>。

```
    .apply(Values.<String>create()) // PCollection<String>
```

KafkaIO 写操作

写操作跟读操作配置基本相似，我们看一下具体代码。

```
   PCollection<KV<Long, String>> kvColl = ...;
   kvColl.apply(KafkaIO.<Long, String>write()
     .withBootstrapServers("broker_1:9092,broker_2:9092")
     .withTopic("results")
     .withKeySerializer(LongSerializer.class)
     .withValueSerializer(StringSerializer.class)
     // You can further customize KafkaProducer used to write the records by adding more
     // settings for ProducerConfig. e.g, to enable compression :
```

```
      .updateProducerProperties(ImmutableMap.of("compression.type", "gzip"))
     // You set publish timestamp for the Kafka records.
      .withInputTimestamp() // element timestamp is used while publishing to Kafka
      // or you can also set a custom timestamp with a function.
      .withPublishTimestampFunction((elem, elemTs) -> ...)
       // Optionally enable exactly-once sink (on supported runners). See JavaDoc for
withEOS().
      .withEOS(20, "eos-sink-group-id");
      );
```

下面这个是 Kafka 里面比较重要的一个属性设置，在 Beam 中是这样使用的，非常简单，但是要注意这个属性.withEOS 其实就是 Kafka 中"Exactly-once"。

```
    .withEOS(20, "eos-sink-group-id");
```

在写入 Kafka 时完全一次性地提供语义，这使得应用程序能够在 Beam 管道中的一次性语义之上提供端到端的一次性保证。它确保写入接收器的记录仅在 Kafka 上提交一次，即使在管道执行期间重试某些处理也是如此。重试通常在应用程序重新启动时发生（如在故障恢复中）或者在重新分配任务时（如在自动缩放事件中）。Flink runner 通常为流水线的结果提供精确一次的语义，但不提供变换中用户代码的副作用。如果诸如 Kafka 接收器之类的转换写入外部系统，则这些写入可能会多次发生。

在此处启用 EOS 时，接收器转换将兼容的 Beam Runners 中的检查点语义与 Kafka 中的事务联系起来，以确保只写入一次记录。由于实现依赖于 runners checkpoint 语义，因此并非所有 runners 都兼容。Beam 中 FlinkRunner 针对 Kafka 0.11+ 版本才支持，然而 Dataflow runner 和 Spark runner 如果操作 kafkaIO 是完全支持的。

关于性能的注意事项

"Exactly-once" 在接收初始消息的时候，除了将原来的数据进行格式化转换外，还经历了 2 个序列化 / 反序列化循环。根据序列化的数量和成本，CPU 可能会涨的很明显。通过写入二进制格式数据（即在写入 Kafka 接收器之前将数据序列化为二进制数据）可以降低 CPU 成本。

关于参数

- numShards——设置接收器并行度。存储在 Kafka 上的状态元数据，使用 sinkGroupId 存储在许多虚拟分区中。一个好的经验法则是将其设置为 Kafka 主题中的分区数。
- sinkGroupId——用于在 Kafka 上将少量状态存储为元数据的组 ID。它类似于与 KafkaConsumer 一起使用的使用 groupID。每个作业都应使用唯一的 groupID，以便重新启动 / 更新作业保留状态以确保一次性语义。状态是通过 Kafka 上的接收器事务原子提交的。有关更多信息，请参阅 KafkaProducer.sendOffsetsToTransaction（Map，String）。接收器在初始化期间执行多个健全性检查以捕获常见错误，以便它不会最终使用似乎不是由同一作业写入的状态。

五. Apache Beam Flink 源码剖析

Apache Beam FlinkRunner 对 Flink 支持依赖情况

Flink 是一个流和批处理的统一的计算框架，Apache Beam 跟 Flink API 做了无缝集成。在 Apache Beam 中对 Flink 的操作主要是 FlinkRunner.java，Apache Beam 支持不同版本的 flink 客户端。我根据不同版本列了一个 Flink 对应客户端支持表如下。

Apache Beam 版本	Flink 版本
V0.4.0	1.1.2
V0.5.0	1.1.2
V0.6.0	1.2.0
V2.0.0	1.3.0
V2.1.0	1.3.0
V2.1.1	1.3.0
V2.2.0	1.3.0
V2.3.0	1.4.0
V2.4.0	1.4.0
V2.5.0	1.4.0
V2.6.0	1.5.2

图 5-1 FlinkRunner 与 Flink 依赖关系表

从图 5-1 中可以看出，Apache Beam 对 Flink 的 API 支持的更新速度非常快，从源码可以看到 2.0.0 版本之前的 FlinkRunner 是非常 low 的，并且直接拿 Flink 的实例做为 Beam 的实例，封装的效果也比较差。但是从 2.0.0 版本之后 ，Beam 就像打了鸡血一样 API 更新速度特别快，抛弃了以前的冗余，更好地跟 Flink 集成，让人眼前一亮。

Apache Beam Flink 源码解析

因为 Beam 在运行的时候都是显式指定 Runner，在 FlinkRunner 源码中只是成了简单的统一入口，代码非常简单，但是这个入口中有一个比较关键的接口类 FlinkPipelineOptions。

请看代码:

```
/** Provided options. */
private final FlinkPipelineOptions options;
```

通过这个类我们看一下 Apache Beam 到底封装了哪些 Flink 方法。

首先FlinkPipelineOptions 是一个接口类，但是它继承了 PipelineOptions、ApplicationNameOptions、StreamingOptions 三个接口类，第一个 PipelineOptions 大家应该很熟悉了，用于基本管道创建；第二个 ApplicationNameOptions 用于设置应用程序名字；第三个用于判断是流式数据还是批数据。源代码如下:

```
public interface FlinkPipelineOptions  extends
PipelineOptions, ApplicationNameOptions, StreamingOptions {
//....
}
```

1. 设置 Flink Master 方法，这个方法用于设置 Flink 集群地址的 Master 地址。可以填写 IP 和端口，或者是 hostname 和端口，默认 local 。当然测试也可以是单机的，在 Flink 1.4 利用 start-local.sh 启动，而到了 1.5 以上就去掉了这个脚本，本地直接换成了 start-cluster.sh。大家测试的时候需要注意一下。

```
/**
* The url of the Flink JobManager on which to execute pipelines.   This can either be the the   * address of a cluster JobManager, in the form "host:port" or one of the special Strings   * "[collection]" will execute the pipeline on Java Collections while "[auto]" will let the system
 */
@Description( "Address of the Flink Master where the Pipeline should  be executed. Can"+ "[collection] or [auto].")
void setFlinkMaster(String value);
```

2. 设置 Flink 的并行数，属于 Flink 高级 API 里面的属性。设置合适的 parallelism 能提高运算效率，太多了和太少了都不行。设置 parallelism 有多种方式，优先级为 api>env>p>file。

```
@Description("The degree of parallelism to be used when distributing operations onto workers.")
@Default.InstanceFactory(DefaultParallelismFactory.class)
Integer getParallelism();
void setParallelism(Integer value);
```

3. 设置连续检查点之间的间隔时间（即当前的快照）用于容错的管道状态。

```
@Description("The interval between consecutive checkpoints (i.e.snapshots of the current"
@Default.Long(-1L)
Long getCheckpointingInterval();
void setCheckpointingInterval(Long interval)
```

4. 定义一致性保证的检查点模式，默认为"AT_LEAST_ONCE"，在 Beam 的源码中定义了一个枚举类 CheckpointingMode，除了默认的"AT_LEAST_ONCE"，还有"EXACTLY_ONCE"。

- "AT_LEAST_ONCE"：这个模式意思是系统将以一种更简单地方式来对 operator 和 udf 的状态进行快照：在失败后进行恢复时，在 operator 的状态中，一些记录可能会被重放多次。
- "EXACTLY_ONCE"：这种模式意思是系统将以如下语义对 operator 和 udf(user defined function) 进行快照：在恢复时，每条记录将在 operator 状态中只被重现 / 重放一次。

```
@Description("The checkpointing mode that defines consistency guarantee.")
@Default.Enum("AT_LEAST_ONCE")
CheckpointingMode getCheckpointingMode();
void setCheckpointingMode(CheckpointingMode mode);
```

5. 设置检查点的最大超时时间，默认为 20*60*1000(毫秒)=20(分钟)。

```
@Description("The maximum time that a checkpoint may take before being discarded.")
@Default.Long(20 * 60 * 1000)
Long getCheckpointTimeoutMillis();
void setCheckpointTimeoutMillis(Long checkpointTimeoutMillis);
```

6. 设置重新执行失败任务的次数，值为 0 有效地禁用容错，值为 -1 表示使用系统默认值（在配置中定义)。

```
@Description(
"Sets the number of times that failed tasks are re-executed. "
+ "A value of zero effectively disables fault tolerance. A value of -1 indicates "+
"that the system default value (as defined in the configuration) should be used.")
@Default.Integer(-1)
Integer getNumberOfExecutionRetries();
void setNumberOfExecutionRetries(Integer retries);
```

7. 设置执行之间的延迟，默认值为 -1L。

```
@Description(
  "Sets the delay between executions. A value of {@code -1} "
   + "indicates that the default value should be used.")
@Default.Long(-1L)
Long getExecutionRetryDelay();
void setExecutionRetryDelay(Long delay);
```

8. 设置重用对象的行为。

```
@Description("Sets the behavior of reusing objects.")
@Default.Boolean(false)
Boolean getObjectReuse();
void setObjectReuse(Boolean reuse);
```

9. 设置状态后端在计算期间存储 Beam 的状态，不设置从配置文件中读取默认值。注意：仅在执行时适用流媒体模式。

```
@Description("Sets the state backend to use in streaming mode. "
@JsonIgnore
AbstractStateBackend getStateBackend();
void setStateBackend(AbstractStateBackend stateBackend);
```

10. 在 Flink Runner 中启用 / 禁用 Beam 指标。

```
@Description("Enable/disable Beam metrics in Flink Runner")
@Default.Boolean(true)
BooleangetEnableMetrics();
```

```
voidsetEnableMetrics(BooleanenableMetrics);
```

11. 启用或禁用外部检查点，与 CheckpointingInterval 一起使用。

```
@Description(
"Enables or disables externalized checkpoints."
+"Works in conjunction with CheckpointingInterval")
@Default.Boolean(false)
BooleanisExternalizedCheckpointsEnabled();
voidsetExternalizedCheckpointsEnabled(BooleanexternalCheckpoints);
```

12. 设置当他们的 Wartermark 达到 + Inf 时关闭源，Watermark 在 Flink 中其中一个作用是根据时间戳做单节点排序，Beam 也是支持的。

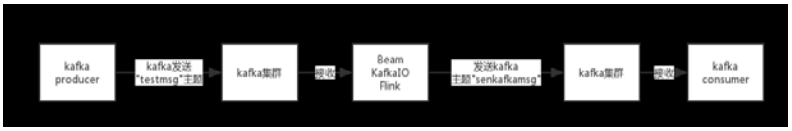
```
@Description("If set, shutdown sources when their watermark reaches +Inf.")
@Default.Boolean(false)
BooleanisShutdownSourcesOnFinalWatermark();
voidsetShutdownSourcesOnFinalWatermark(BooleanshutdownOnFinalWatermark);
```

剩余两个部分这里不再进行翻译，留给大家去看源码。

六. KafkaIO 和 Flink 实战

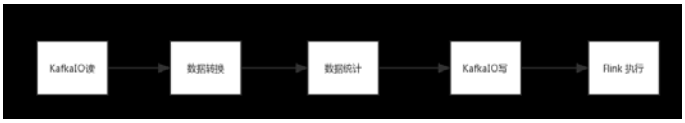
本节通过解读一个真正的 KafkaIO 和 Flink 实战案例，帮助大家更深入地了解 Apache Beam KafkaIO 和 Flink 的运用。

设计架构图和设计思路解读



Apache Beam 外部数据流程图

设计思路：Kafka 消息生产程序发送 testmsg 到 Kafka 集群，Apache Beam 程序读取 Kafka 的消息，经过简单的业务逻辑，最后发送到 Kafka 集群，然后 Kafka 消费端消费消息。



Apache Beam 内部数据处理流程图

Apache Beam 程序通过 kafkaIO 读取 Kafka 集群的数据，进行数据格式转换。数据统计后，通过 KafkaIO 写操作把消息写入 Kafka 集群。最后把程序运行在 Flink 的计算平台上。

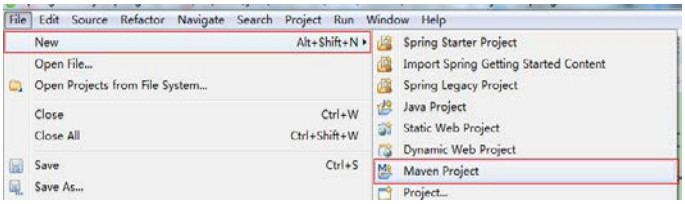
软件环境和版本说明

- 系统版本 centos 7。
- Kafka 集群版本： kafka_2.10-0.10.1.1.tgz。
- Flink 版本：flink-1.5.2-bin-hadoop27-scala_2.11.tgz。

Kafka 集群和 Flink 单机或集群配置，大家可以去网上搜一下配置文章，操作比较简单，这里就不赘述了。

实践步骤

1. 新建一个 Maven 项目。

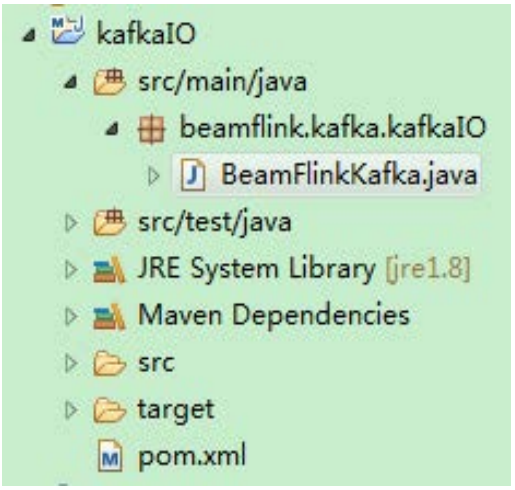


2. 在 pom 文件中添加 jar 引用。

```
 <dependency>
<groupId>org.apache.beam</groupId>
<artifactId>beam-sdks-java-io-kafka</artifactId>
<version>2.4.0</version>
</dependency>
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-clients</artifactId>
<version>0.10.1.1</version>
</dependency>
<dependency>
<groupId>org.apache.beam</groupId>
<artifactId>beam-runners-core-java</artifactId>
<version>2.4.0</version>
</dependency>
<dependency>
<groupId>org.apache.beam</groupId>
<artifactId>beam-runners-flink_2.11</artifactId>
<version>2.4.0</version>
</dependency>
```

```
 <dependency>
 <groupId>org.apache.flink</groupId>
<artifactId>flink-java</artifactId>
<version>1.5.2</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-clients_2.11</artifactId>
<version>1.5.2</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-core</artifactId>
<version>1.5.2</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-runtime_2.11</artifactId>
 <version>1.5.2</version>
 <!--<scope>provided</scope>-->
 </dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-streaming-java_2.11</artifactId>
 <version>1.5.2</version>
 <!--<scope>provided</scope>-->
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-metrics-core</artifactId>
 <version>1.5.2</version>
 <!--<scope>provided</scope>-->
</dependency>
```

3. 新建 BeamFlinkKafka.java 类（右图）。



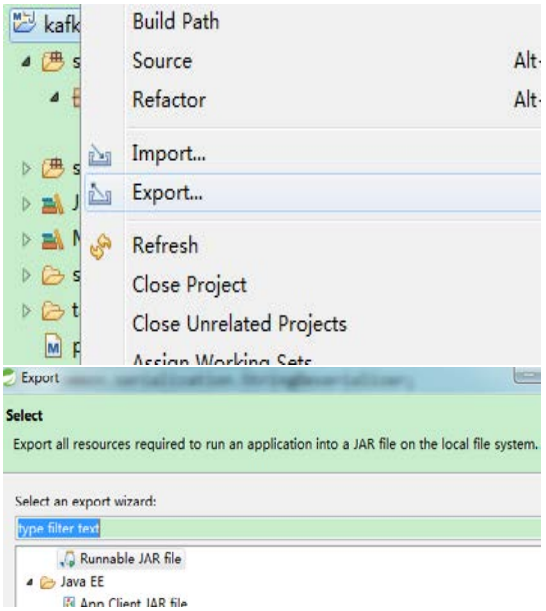
4. 编写以下代码：

```
    public static void main(String[] args) {
    // 创建管道工厂
    PipelineOptions options =
PipelineOptionsFactory.create();
    // 显式指定 PipelineRunner: FlinkRunner 必
须指定如果不制定则为本地
    options.setRunner(FlinkRunner.class);
```

```
    // 设置相关管道
     Pipeline pipeline = Pipeline.create(options);
     // 这里 kV 后说明 kafka 中的 key 和 value 均为 String 类型
    PCollection<KafkaRecord<String, String>> lines =
    pipeline.apply(KafkaIO.<String,
    // 必需设置 kafka 的服务器地址和端口
    String>read().withBootstrapServers("192.168.1.110:11092,192.168.1.119:11092,192.168.1.120:11092")
       .withTopic("testmsg")// 必需设置要读取的 kafka 的 topic 名称
       .withKeyDeserializer(StringDeserializer.class)// 必需序列化 key
       .withValueDeserializer(StringDeserializer.class)// 必需序列化 value
      .updateConsumerProperties(ImmutableMap.<String, Object>of("auto.offset.reset",
"earliest")));// 这个属性 kafka 最常见的 .
     // 为输出的消息类型。或者进行处理后返回的消息类型
    PCollection<String> kafkadata = lines.apply("Remove Kafka Metadata", ParDo.of(new
DoFn<KafkaRecord<String, String>, String>() {
   private static final long serialVersionUID = 1L;
   @ProcessElement
   public void processElement(ProcessContext ctx) {
   System.out.print(" 输出的分区为 ----: " + ctx.element().getKV());
    ctx.output(ctx.element().getKV().getValue());// 其实我们这里是把 " 张海涛在发送消息 ***" 进
行返回操作
   }
  }));
    PCollection<String> windowedEvents = kafkadata.apply(Window.<String>into(FixedWindows.
of(Duration.standardSeconds(5))));
    PCollection<KV<String, Long>> wordcount = windowedEvents.apply(Count.<String>perEleme
nt()); // 统计每一个 kafka 消息的 Count
    PCollection<String> wordtj = wordcount.apply("ConcatResultKVs", MapElements.via( // 拼
接最后的格式化输出（Key 为 Word，Value 为 Count）
    new SimpleFunction<KV<String, Long>, String>() {
    private static final long serialVersionUID = 1L;
    @Override
    public String apply(KV<String, Long> input) {
    System.out.print(" 进行统计：" + input.getKey() + ": " + input.getValue());
      return input.getKey() + ": " + input.getValue();
     }
    }));
    wordtj.apply(KafkaIO.<Void,String>write().withBootstrapServe
rs("192.168.1.110:11092,192.168.1.119:11092,192.168.1.120:11092")// 设置写会 kafka 的集群配置地址
    .withTopic("senkafkamsg")// 设置返回 kafka 的消息主题
    // .withKeySerializer(StringSerializer.class)// 这里不用设置了，因为上面 Void
    .withValueSerializer(StringSerializer.class)
    // Dataflow runner and Spark 兼容， Flink 对 kafka0.11 才支持。我的版本是 0.10 不兼容
```

```
    //.withEOS(20, "eos-sink-group-id")
    .values() // 只需要在此写入默认的 key 就行了，默认为 null 值
  ); // 输出结果
  pipeline.run().waitUntilFinish();
    }
```

5. 打包 jar，本示例是简单的实战，并没有用 Docker，Apache Beam 新版本是支持 Docker 的。

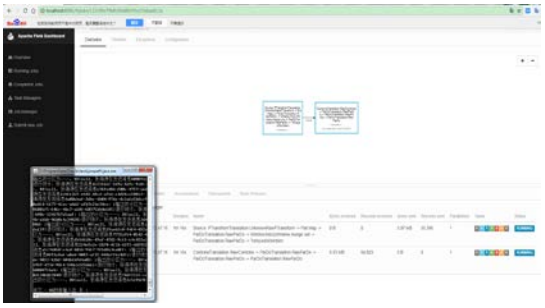


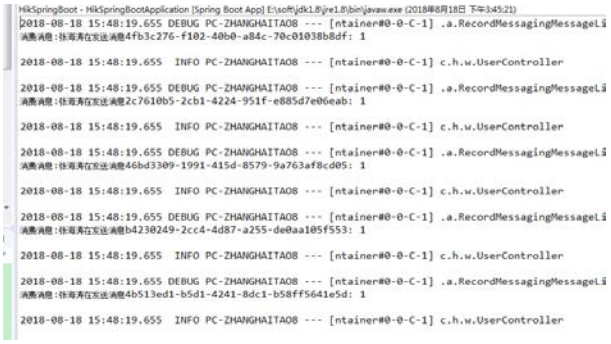
6. 通过 Apache Flink Dashboard 提交 job

7. 查看结果，程序接收的日志如下图所示。

七. 实战解析

本次实战在源码分析中已经做过详细解析，在这里不做过多的描述，只选择部分问题再重点解释一下。此外，如果还没有入门，甚至连管道和 Runner 等概念都还不清楚，建议先阅读本系列的第一篇文章《Apache Beam 实战指南之基础入门》。





1. FlinkRunner 在实战中是显式指定的，如果想设置参数怎么使用呢？其实还有另外一种写法，例如以下代码：

```
//FlinkPipelineOptions options =PipelineOptionsFactory.as(FlinkPipelineOptions.class);
//options.setStreaming(true);
//options.setAppName("app_test");
//options.setJobName("flinkjob");
//options.setFlinkMaster("localhost:6123");
//options.setParallelism(10);// 设置 flink 的并行度
// 显式指定 PipelineRunner：FlinkRunner，必须指定，如果不指定则为本地
options.setRunner(FlinkRunner.class);
```

2. Kafka 有三种数据读取类型，分别是“earliest”，“latest”，“none”，分别的意思如下所述。

- earliest：当各分区下有已提交的 offset 时，从提交的 offset 开始消费；无提交的 offset 时，从头开始消费。
- latest：当各分区下有已提交的 offset 时，从提交的 offset 开始消费；无提交的 offset 时，消费新产生的该分区下的数据 。
- none：topic 各分区都存在已提交的 offset 时，从 offset 后开始消费；只要有一个分区不存在已提交的 offset，则抛出异常。

```
.updateConsumerProperties(ImmutableMap.<String,Object>of("auto.offset.reset",
"earliest")));
```

3. 实战中我自己想把 Kafka 的数据写入，key 不想写入，所以出现了 Kafka 的 key 项为空，而 values 才是真正发送的数据。所以开始和结尾要设置个.values()，如果不加上就会报错。

```
KafkaIO.<Void, String>write()
.values() // 只需要在此写入默认的 key 就行了，默认为 null 值
```

八. 小结

随着 AI 和 loT 的时代的到来，各个公司不同结构、不同类型、不同来源的数据进行整合的成本越来越高。Apache Beam 技术的统一模型和大数据计算平台特性优雅地解决了这一问题，相信在 loT 万亿市场中，Apache Beam 将会发挥越来越重要的角色。

张海涛，目前就职于海康威视云基础平台，负责云计算大数据的基础架构设计和中间件的开发，专注云计算大数据方向。Apache Beam 中文社区发起人之一，如果想进一步了解最新 Apache Beam 动态和技术研究成果，请加微信 cyrjkj 入群共同研究和运用。

如何实现靠谱的分布式锁?

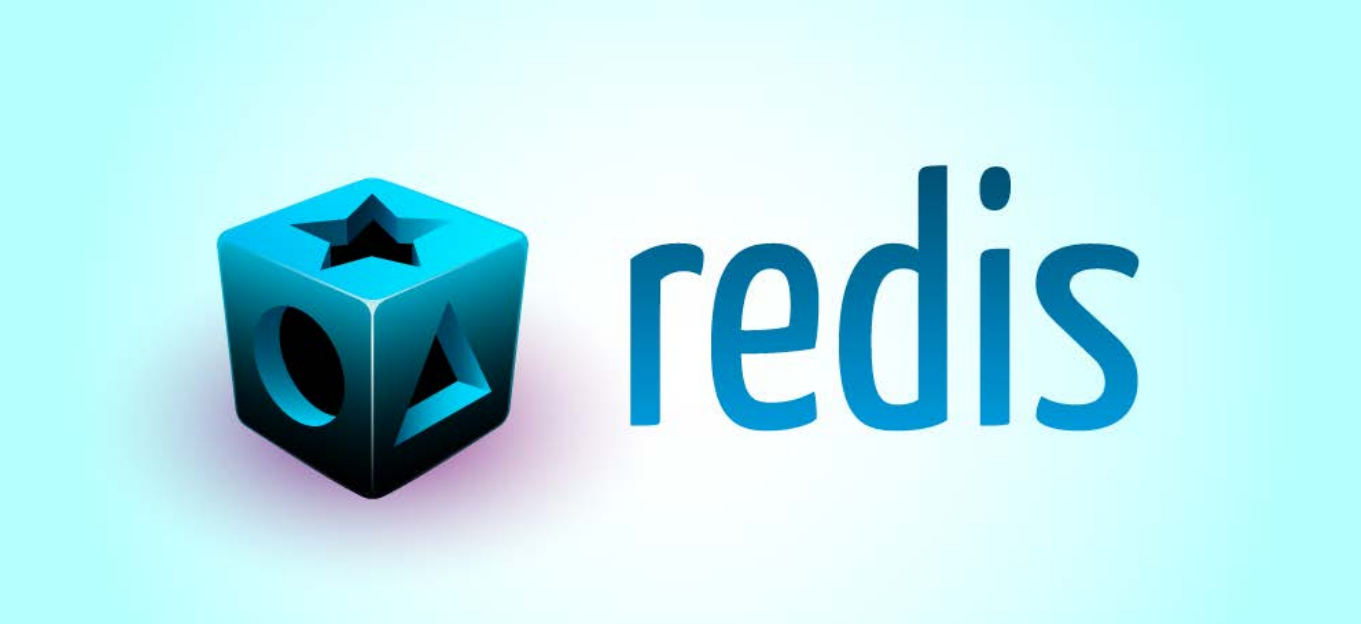
作者 鞠明业

分布式锁，是用来控制分布式系统中互斥访问共享资源的一种手段，从而避免并行导致的结果不可控。基本的实现原理和单进程锁是一致的，通过一个共享标识来确定唯一性，对共享标识进行修改时能够保证原子性和和对锁服务调用方的可见性。由于分布式环境需要考虑各种异常因素，为实现一个靠谱的分布式锁服务引入了一定的复杂度。

分布式锁服务一般需要能够保证：

- 同一时刻只能有一个线程持有锁；
- 锁能够可重入；
- 不会发生死锁；
- 具备阻塞锁特性，且能够及时从阻塞状态被唤醒；
- 锁服务保证高性能和高可用。

当前使用较多的分布式锁方案主要基于 redis、zookeeper 提供的功能特性加以封装来实现的，下面我们会简要分析下这两种锁方案的处理流程以及它们各自的问题。



1. 基于 Redis 实现的锁服务

加锁流程

```
SET resource_name my_random_value NX PX max-lock-time
```

注：资源不存在时才能够成功执行 set 操作，用于保证锁持有者的唯一性；同时设置过期时间用于防止死锁；记录锁的持有者，用于防止解锁时解掉了不符合预期的锁。

解锁流程

```
if redis.get("resource_name") ==  " my_random_value"
     return redis.del("resource_name")
    else
     return 0
```

注：使用 lua 脚本保证获取锁的所有者、对比解锁者是否所有者、解锁是一个原子操作。

该方案的问题如下。

1. 通过过期时间来避免死锁，过期时间设置多长对业务来说往往比较头疼，时间短了可能会造成：持有锁的线程 A 任务还未处理完成，锁过期了，线程 B 获得了锁，导致同一个资源被 A、B 两个线程并发访问；时间长了会造成：持有锁的进程宕机，造成其他等待获取锁的进程长时间的无效等待

2. Redis 的主从异步复制机制可能丢失数据，会出现如下场景：A 线程获得了锁，但锁数据还未同步到 slave 上，master 挂了，slave 顶成主，线程 B 尝试加锁，仍然能够成功，造成 A、B 两个线程并发访问同一个资源

2. 基于 ZooKeeper 实现的锁服务

加锁流程

1. 在 /resource_name 节点下创建临时有序节点。

2. 获取当前线程创建的节点及 /resource_name 目录下的所有子节点，确定当前节点序号是否最小，是则加锁成功。否则监听序号较小的前一个节点。

注：zab 一致性协议保证了锁数据的安全性，不会因为数据丢失造成多个锁持有者；心跳保活机制解决死锁问题，防止由于进程挂掉或者僵死导致的锁长时间被无效占用。具备阻塞锁特性，并通过 watch 机制能够及时从阻塞状态被唤醒。

解锁流程

删除当前线程创建的临时接点。

该方案的问题在于

通过心跳保活机制解决死锁会造成锁的不安全性，可能会出现如下场景：持有锁的线程 A 僵死或网络故障，导致服务端长时间收不到来自客户端的保活心跳，服务端认为客户端进程不存活主动释放锁，线程 B 抢到锁，线程 A 恢复，同时有两个线程访问共享资源。

基于上诉对现有锁方案的讨论，我们能看到，一个理想的锁设计目标主要应该解决如下问题：

1. 锁数据本身的安全性；
2. 不发生死锁；
3. 不会有多个线程同时持有相同的锁。

而为了实现不发生死锁的目标，又需要引入一种机制，当持有锁的进程因为宕机、GC 活者网络故障等各种原因无法主动过释放锁时，能够有其他手段释放掉锁，主流的做法有两种：

1. 锁设置过期时间，过期之后 Server 端自动释放锁；
2. 对锁的持有进程进行探活，发现持锁进程不存活时 Server 端自动释放。

实际上不管采用哪种方式，都可能造成锁的安全性被破坏，导致多个线程同时持有同一把锁的情况出现。因此我们认为锁设计方案应在预防死锁和锁的安全性上取得平衡，没有一种方案能够绝对意义上保证不发生死锁并且是安全的。而锁一般的用途又可以分为两种，实际应用场景下，需要根据具体需求出发，权衡各种因素，选择合适的锁服务实现模型。无论选择哪一种模型，需要我们清楚地知道它在安全性上有哪些不足，以及它会带来什么后果。

- 为了效率，主要是避免一件事被重复的做多次，用于节省 IT 成本，即使锁偶然失效，也不会造成数据错误，该种情况首要考虑的是如何防止死锁。
- 为了正确性，在任何情况下都要保证共享资源的互斥访问，一旦发生就意味着数据可能不一致，造成严重的后果，该种情况首要考虑的是如何保证锁的安全。

3. SharkLock 一些设计选择

锁信息设计如下：

```
lockBy：client 唯一标识
condition：client 在加锁时传给 server，用于定义 client 期望 server 的行为方式
lockTime：加锁时间
txID：全局自增 ID
lease：租约
```

如何保证锁数据的可靠性

SharkLock 底层存储使用的是 sharkStore，SharkStore 是一个分布式的持久化 Key-Value 存储系统。采用多副本来保证数据安全，同时使用 raft 来保证各个副本之间的数据一致性。

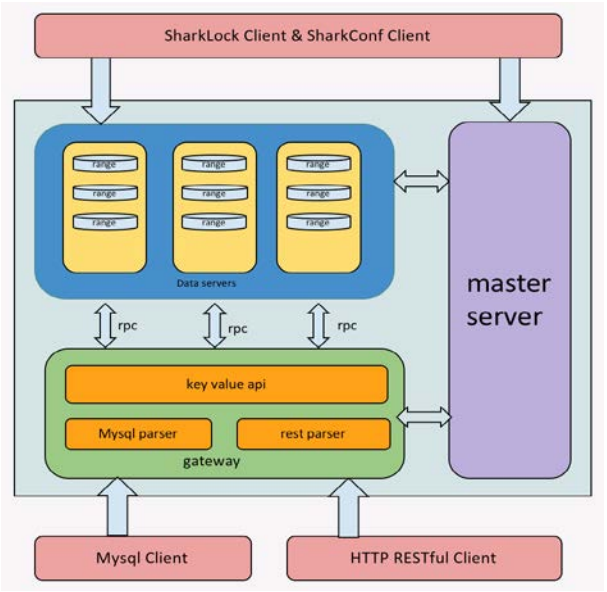
如何预防死锁

1. Client 定时向 Server 发送心跳包，Server 收到心跳包之后，维护 Server 端 Session 并立即回复，Client 收到心跳包响应后，维护 Client 端 Session。心跳包同时承担了延长 Session 租约的功能。
2. 当锁持有方发生故障时，Server 会在 Session 租约到期后，自动删除该 Client 持有的锁，以避免锁长时间无法释放而导致死锁。Client 会在 Session 租约到期后，进行回调，可选择性的决策是否要结束对当前持有资源的访问。
3. 对于未设置过期的锁，也就意味着无法通过租约自动释放故障 Client 持有的锁。因此额外提供了一种协商机制，在加锁的时候传递一些 condition 到服务端，用于约定 Client 端期望 Server 端对异常情况的处理，包括什么情况下能够释放锁。譬如可以通过这种机制实现 server 端在未收到十个心跳请求后自动释放锁，Client 端在未收到五个心跳响应后主动结束对共享资源的访问。
4. 尽最大程度保证锁被加锁进程主动释放 。
    - 进程正常关闭时调用钩子来尝试释放锁。
    - 未释放的锁信息写文件，进程重启后读取锁信息，并尝试释放锁。

如何确保锁的安全性

1. 尽量不打破谁加锁谁解锁的约束，尽最大程度保证锁被加锁进程主动释放。
    - 进程正常关闭时调用钩子来尝试释放锁。
    - 未释放的锁信息写文件，进程重启后读取锁信息，并尝试释放锁。
2. 依靠自动续约来维持锁的持有状态，在正常情况下，客户端可以持有锁任意长的时间，这可以确保它做完所有需要的资源访问操作之后再释放锁。一定程度上防止如下情况发生。
    - 线程 A 获取锁，进行资源访问。
    - 锁已经过期，但 A 线程未执行完成。
    - 线程 B 获得了锁，导致同时有两个线程在访问共享资源。
3. 提供一种安全检测机制，用于对安全性要求极高的业务场景。
    - 对于同一把锁，每一次获取锁操作，都会得到一个全局增长的版本号。
    - 对外暴露检测 API checkVersion（lock_name,version），用于检测持锁进程的锁是不是已经被其他进程抢占（锁已经有了更新的版本号）。
    - 加锁成功的客户端与后端资源服务器通信的时候可带上版本号，后端资源服务器处理请求前，调用 checkVersion 去检查锁是否依然有效。有效则认为此客户端依旧是锁的持有者，可以为其提供服务。
    - 该机制能在一定程度上解决持锁 A 线程发生故障，Server 主动释放锁，线程 B 获取锁成功，A 恢复了认为自己仍旧持有锁而发起修改资源的请求，会因为锁的版本号已经过期而失败，从而保障了锁的安全性。

4. SharkStore 简介

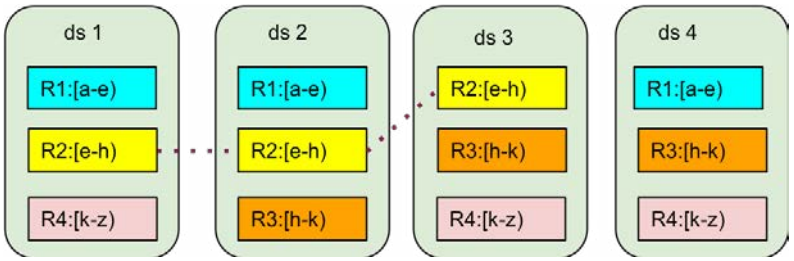


基本模块

- Master Server 集群分片路由等元数据管理、扩容和 Failover 调度等。
- Data Server 数据存储节点，提供 RPC 服务访问其上的 KV 数据。
- Gateway Server 网关节点，负责用户接入。

Sharding

SharkStore 采用多副本的形式来保证数据的可靠性和高可用。同一份数据会存储多份，少于一半的副本宕机仍然可以正常服务。 SharkStore 的数据分布如下图所示。



扩容方案

当某个分片的大小到达一定阈值，就会触发分裂操作，由一个分片变成两个，以达到扩容的目的。

- Dataserver 上 range 的 leader 自己触发。 leader 维持写入操作字节计数，每到达 check size 大小，就异步遍历其负责范围内的数据，计算大小并同时找出分裂时的中间 key 如果大小到达 split

size，向 master 发起 AskSplit 请求，同意后提交一个分裂命令。分裂命令也会通过 raft 复制到其他副本。

- 本地分裂。分裂是一个本地操作，在本地新建一个 range，把原始 range 的部分数据划拨给新 range，原始 range 仍然保留，只是负责的范围减半。分裂是一个轻量级的操作。

Failover 方案

Failover 以 range 的级别进行。range 的 leader 定时向其他副本发送心跳，一段时间内收不到副本的心跳回应，就判断副本宕机，通过 range 心跳上报给 master。由 master 发起 failover 调度。 Master 会先删除宕机的副本然后选择一个合适的新节点，添加到 range 组内之后通过 raft 复制协议来完成新节点的数据同步。

Balance 方案

Dataserver 上的 range leader 会通过 range 心跳上报一些信息，每个 dataserver 还会有一个节点级别的 node 心跳。 Master 收集这些信息来执行 balance 操作。Balance 通过在流量低的节点上增加副本，流量高的节点上减少副本促使整个集群比较均衡，维护集群的稳定和性能。

5. Raft 实践：MultiRaft

1. 心跳合并

以目标 dataserver 为维度，合并 dataserver 上所有 raft 心跳 心跳只携带 range ids，心跳只用来维护 leader 的权威和副本健康检测 range ids 的压缩，比如差量 + 整型变长 Leader 类似跟踪复制进度，跟踪 follower commit 位置。

2. 快照管理控制

建立 ack 机制，在对端处理能力之内发送快照；控制发送和应用快照的并发度，以及限速；减少对正

常业务的冲击。

6. Raft 实践：PreVote

Raft 算法中，leader 收到来自其他成员 term 比较高的投票请求会退位变成 follower。

因此，在节点分区后重加入、网络闪断等异常情况下，日志进度落后的副本发起选举，但其本身并无法被选举为 leader，导致集群在若干个心跳内丢失 leader，造成性能波动 。

针对这种情况，在 raft 作者的博士论文中，提出了 prevote 算法：在发起选举前，先进行一次预选举 Pre-Candidate， 如果预选举时能得到大多数的投票，再增加 term，进行正常的选举。 prevote 会导致选举时间变长 (多了一轮 RPC)，然而这个影响在实践中是非常小的， 可以有利于集群的稳定，是非常值得的实践。

7. Raft 实践：NonVoter

一个新的 raft 成员加入后，其日志进度为空；新成员的加入可能会导致 quorum 增加，并且同时引入了一个进度异常的副本；新成员在跟上 leader 日志进度之前，新写入的日志都无法复制给它；如果此时再有原集群内一个成员宕机， 很有可能导致集群内可写副本数到不到 quorum，使得集群变得不可写。很多 raft 的实现中，都会引入了一种特殊身份的 raft 成员 (non-voting 或者 learner) Learner 在计算 quorum 时不计入其内，只被动接收日志复制，选举超时不发起选举；在计算写入操作是否复制给大多数 (commit 位置) 时，也忽略 learner。 sharkstore raft 会在 leader 端监测 learner 的日志进度， 当 learner 的进度跟 leader 的差距小于一定百分比 (适用于日志总量比较大) 或者小于一定条数时 (适用于日志总量比较小)， 由 leader 自动发起一次 raft 成员变更，提升 leaner 成员为正常成员。

SharkStore 目前已经开源，有兴趣的同学可详细了解，期待能跟大家能够一块儿沟通交流。

2018年DevOps促进现状报告
DevOps精英的方法与习惯

作者 Sarah Schlothauer 译者 邵思华

近期，DevOps Research and Assessment 组织发布了 2018 年 DevOps 促进现状报告。来自各行各业的 1800 多名调查者提交了问卷，内容涵盖了云基础设施、领导力与学习文化、交付效能、数据库实践，等等。

今年是 DORA 连续第五年发布该报告，在报告中可看到新的关注点以及更广泛的调查方向。在 2014 年时，只有 16% 的人员表示他们效力于 DevOps 团队。而在 2018，这个数字已经增长到 27% 这一水平。让我们来解读一下这份报告，看看报告中的结论对于这个持续发展中的 DevOps 世界具有怎样的意义。

DevOps 精英级团队

DORA 的软件交付效能基准将团队划分为三种类型：高效能、中效能与低效能团队，对团队的评价取决于他们的总体产出。发布频率、变更响应时间、服务恢复时间，以及变更故障率等指标全部包括在内。

有 15% 的团队被划分为低效能团队、37% 为中效能，而 48% 的团队属于高效能团队。其中在高效能团队中的 7% 可归为精英级团队，他们的表现可称卓越。

精英级执行团队在以下几个方面有着突出的表现：

- 代码发布频率高 46 倍；
- 代码提交至发布的速度快 2555 倍；
- 变更故障率少 7 倍；
- 事故恢复时间快 2604 倍。

那么，是哪些关键的因素影响了普通团队的效能呢？是因为这些团队对于软件开发与发布往往采取了过于谨慎的做法。

团队的专注将影响你的成功

低效能团队如何转变为高效能团队，乃至成为精英级团队呢？

报告指出，“研究表明，对于各家软件供应商来说，高效能的执行团队与低效能团队相比，手工操作的比例减少了许多，他们将更多地时间用于新工作的开展，而在修复安全漏洞或缺陷的时间上则减少许多。”

改进效能的一大关键是自动化能力，自动化能够加速任务的完成，改进质量与一致性。其结果是团队能够更多地投入在更有价值的工作任务上。由于自动化能力处理了各种低级别工作任务，使人力得到了解放。

现实情况是，团队对时间的利用方式造成了低效能的结果。尤其值得注意的是，各种计划外的工作、安全问题、干扰、客户支持任务以及缺陷是防碍低效能团队向前进步的绊脚石。

精英团队能够将 50% 的时间投入在新工作中，相比之下，低效能团队只能投入 30% 的时间。低效能团队还需要投入 15% 的时间用于客户支持工作，与之相比，精英团队在这方面只需投入 5% 的时间。

继续增长的云计算

毋庸置疑，云计算市场仍在不断增长之中。从结果来看，只有少部分用户仍然游离在外。

报告中有 17% 的调查者仍然没有使用云厂商的服务。与此同时，AWS 在最受欢迎的云平台中占据了头把交椅，投票率为 52%，Azure 屈居次席，占比为 34%。

此外，使用多个云服务的方式已经逐渐普及。41% 的调查者表示他们选择了单一云服务，与之相比，有 40% 的调查者使用了多个云平台的服务。造成这一现象的原因有以下几点：可用性、灾难恢复计划、对于单一服务商信心不足，以及法律合规性原因。

那么，精英团队在云平台方面的常规操作是怎样的呢？“选择平台即服务的调查者与其他调查者相比有 1.5 倍的可能性成为精英用户，并且遵循了云原生最佳实践的用户有 1.8 倍的可能性成为精英团队。选择通过基础设施即代码方式管理云发布的用户有 1.8 倍的可能性成为精英，最后是选择容器技术的用户有 1.5 倍的可能性成为精英团队。”

不过，云平台的应用情况似乎遇到了某些障碍。用户并没有充分利用云平台所带来的决定性的特点，从而对软件交付的效能产生的影响。DORA 列举了 5 项云计算的基本特征，但这几方面的数据都不理想。

以下是在调查者反馈中对于云平台的特性表示赞同或强烈赞同的比例。

- 按需分配，自主服务：46%
- 广泛的网络访问性：46%
- 资源池：43%
- 快速的弹性能力：45%
- 符合标准的服务：48%

以上这些特征的评分都没有超过半数的选择，这也反映出团队在效率方面的不足。

“文化是 DevOps 的重要组成部分”

组织的文化如何才能健康地发展？DORA 引用了社会学家 Ron Westrum 所提出的看法。Westrum

发现“企业的文化可通过安全感与效能产出进行预测”。

企业的文化可能是病态的（权力导向），官僚的（制度导向），也可能是生机勃勃的（效能导向）。每一种文化在处理协作、职责、风险、新事物与面对问题时的反应都是不同的。为了 DevOps 能够成功实施，应当满足某些基本的文化条件，包括打破团队壁垒，以及实施新的方式方法。

如何利用这些已知的内容使你的团队得到改进呢？请记住其中的精华部分：“谷歌的研究者对 180 个工程团队进行了研究分析，他们发现，高效能的团队通常在心理上有安全感，或者说在团队中不会害怕冒风险。除此之外，其他因素还包括可依赖性、工作的结构化与清晰度、工作的意义与个人影响力。”

高效能团队的其他重要特点还包括浓厚的学习氛围，学习是一种有价值的投资，它能够带来效能的回馈。为了减轻团队的负担，学习任务应当在工作时间完成，而不是在周末或加班后作为额外的任务。工作中学习的方式能够让团队更快地适应未知的变化。

受众面的提高

最后，2018 年的报告得到了更多女性从业者的反馈，这也反映出近期行业性别分布状况的变化。今年，有 12% 的调查者确定为女性，并且按调查者的表示，在团队中有 25% 的成员为女性。这是个非常大的变化：去年仅有 6% 的调查者为女性，进入 IT 业的女性队伍正在不断壮大。

此外，有 4% 的调查者并未留下性别信息，还有不到 1% 选择了非二元性别的选项。

今年，DORA 首次在问卷中问及调查者的伤残状况，有 6% 的调查者表示他们有某部分的身体残疾，另有 9% 的调查者未作出选择。

效能方面的关键结论

如果你需要更多信息，以促进你的团队进入精英团队的领域，可以参考以下报告结果。

开源非常重要，“高效能团队深入应用开源软件的可能性是其他类型软件的 1.75 倍，而这些团队在未来提高开源软件使用度的可能性也是其他团队的 1.5 倍。”

外包会带来效能下降，“低效能团队将整部分功能进行外包的可能性几乎是高效能团队的 4 倍，这些功能包括测试或运维等等。”

在任何行业中都存在高效能的团队，“我们在具有高合规性要求及无合规性要求的行业中都能看到高效能团队的存在。”

团队的信任促进公司的发展，“我们发现，当团队主管让成员能够自治地开展工作时，就能够提高团队信任感。”

TGO 鲲鹏會

在 TGO 鲲鹏会，成为全球科技领袖

About | TGO 简介

TGO 是汇聚全球科技领导者的高端社群，我们希望让所有孤军奋战的科技管理者都找到属于自己的圈子。在这里，您可以冲破职业晋升道路中的成长局限，搭建快速连接每一个科技管理者的沟通桥梁，打破传统企业与互联网企业之间的技术壁垒，探索业务与技术交融下产生的无限可能…

会员人数	全球分会	全年学习活动
700+	10	240+ 场次

Rights | 会员权益

高频活动，开拓视野。十余种多维度的活动形式，深入解读热门话题，丰富行业观察视角，获取最佳实践路径。

每月小组交流
8~10人小组私密独享
的线下深度交流

专属学习活动
全年20+场，学习、体验
名企拜访无所不包

高端会议自由参加
QCon、ArchSummit
AICon 等全免费

免费知识专栏
极客时间 App 所有
付费专栏免费阅读

破解研发难题
多级会员体系
为你建立人才库

高端资源对接
技术、产品、服务
彼此共享，强强联合

个人品牌打造
专业形象照、媒体专访
会议分享服务

企业尊享优惠
超低价格采购极客邦科技
企业服务和产品

前沿资讯获取
每日 / 每周资讯
开阔视野，提升认知

权益持续更新
CTO/CPO 特训营、导师
私享会直投刊物等研发中

扫描二维码了解更多内容