

第 16 组研究与应用报告

注：老师觉得太多想看实操可以[直接看第 5 章实操部分](#)，前面是我们研究与学习时整理的内容与笔记

目录

1. 前言与背景	4
2. 数据中台研究	4
2.1 数据中台的作用	4
2.2 数据中台经典架构	5
2.2.1 数据采集	6
2.2.2 数据存储与计算	7
2.3 大数据技术生态	9
3. 数据仓库设计方法论	10
3.1 核心架构	10
3.2 仓库建模	11
3.3 事实表概述	13
3.4 维度表	13
3.4.1 维度表设计步骤	13
3.4.2 维度设计要点	14
3.5 数据仓库分层规划	16
3.5.1 数据调研	17
3.5.2 明确数据域	17
3.5.3 构建业务总线矩阵	18
4. 组件研究与学习	18
4.1 Zookeeper	18
4.1.1 基本介绍	19
4.1.2 Zookeeper 特点	19
4.1.3 应用场景	19
4.1.4 集群选举机制	21
4.2 raft 与分布式思想	22
4.2.1 分布式系统是什么	22
4.2.2 如何实现分布式计算	22
4.2.3 raft 在 kafka 中的实现与应用	26
4.3 kafka	35
4.3.1 Kafka 概述	35
4.3.2 Kafka 优点	36
4.3.3 Kafka 常用术语	37
4.3.4 Kafka 基础架构	39
4.3.5 Kafka 工作原理与应用场景	40

4.3.6 kafka 与其他消息中间件的对比	46
4.3.7 kafka 调优	48
4.4 Flume	60
4.4.1 Flume 概述	60
4.4.2 Flume 基本架构与工作原理	61
4.4.3 Flume 拓扑结构	63
4.4.4 Flume 数据流监控	64
4.5 Redis	67
4.5.1 Redis 概述	67
4.5.2 Redis 优势与应用场景	67
4.5.3 Redis 数据类型与基本原理	68
4.5.4 Redis 持久化	71
4.6 MongoDB	72
4.7 Flink	72
4.7.1 Flink 概述	72
4.7.2 常用名词解释	73
4.7.3 Flink 基本架构	73
4.7.4 Flink 理论研究	74
4.8 Maxwell	92
4.8.1 Maxwell 简介	92
4.8.2 Maxwell 原理	93
4.9 DataX	94
4.9.1 DataX 简介	94
4.9.2 DataX 原理	94
4.9.3 DataX 与 Sqoop 对比	95
4.10 sqoop	96
4.10.1 sqoop 简介	96
4.10.2 sqoop 原理	96
4.11 Hbase	97
4.11.1 Hbase 简介	97
4.11.2 Hbase 逻辑与物理结构	97
4.11.3 Hbase 基本架构	98
4.11.4 生产调优	99
4.12 Hive	101
4.12.1 Hive 简介	101
4.12.2 Hive 架构原理	102
4.12.3 Hive 生产调优	103
4.13 Spark	105
4.13.1 Spark 概述	105
4.13.2 Hadoop Mapreduce 与 Spark 对比	105
4.13.3 Spark 核心模块	106
4.13.4 Spark 运行环境	106

4.13.5 Spark 运行架构	107
4.13.6 提交流程	108
4.13.7 生产调优	109
4.14 Hadoop 生产调优	111
4.14.1 HDFS—核心参数	111
4.14.2 HDFS—集群压测	114
4.14.3 HDFS—多目录	117
4.14.4 HDFS—集群扩容及缩容	119
4.14.5 HDFS—存储优化	124
4.14.6 HDFS—故障排除	132
4.14.7 Hadoop 综合调优	137
4.15 DolphinScheduler	144
5. 项目与组件实践	147
5.1 raft 算法的 go 语言实现	147
5.2 Kafka 相关实践	160
5.2.1 Kafka-Eagle 监控	160
5.2.2 Kafka-Kraft 模式部署	164
5.2.3 案例 1：Flume+Kafka+SparkStreaming 实现词频统计	168
5.2.4 案例 2：Spark+Kafka 构建实时分析 Dashboard	178
5.3 Flume：高可用拓扑结构实践	195
5.3.1 Flume 拓扑部署案例	195
5.4 Redis 实操与分布式高可用部署	217
5.4.1 Redis 安装	217
5.4.2 Redis 事务实践	217
5.4.3 Redis 实现锁（乐观锁）	220
5.4.4 Redis 发布订阅	222
5.4.5 Redis 集群环境搭建	224
5.4.6 Redis 高可用	225
5.5 Flink：基于 Flink 的 CVPR2022 论文数据的分析与处理	229
5.5.1 Flink 集群搭建	229
5.5.2 Flink 作业提交	231
5.5.3 通过爬虫爬取 CVPR2022 的论文数据	232
5.5.4 使用 Flink 的批处理功能统计关键词词频	233
5.5.5 关键词云图谱统计	236
5.5.6 关键词柱状图统计	237
5.5.7 高产作者统计	237
5.6 Sqoop：基础操作与通道实践	238
5.6.1 Sqoop 安装	238
5.6.2 Sqoop 导入数据	239
5.6.3 Sqoop 导出数据	243
5.7 爬虫应用项目：Python 爬取唯品会数据→Flume→Kafka	244
5.7.1 唯品会口红商品数据爬虫	244

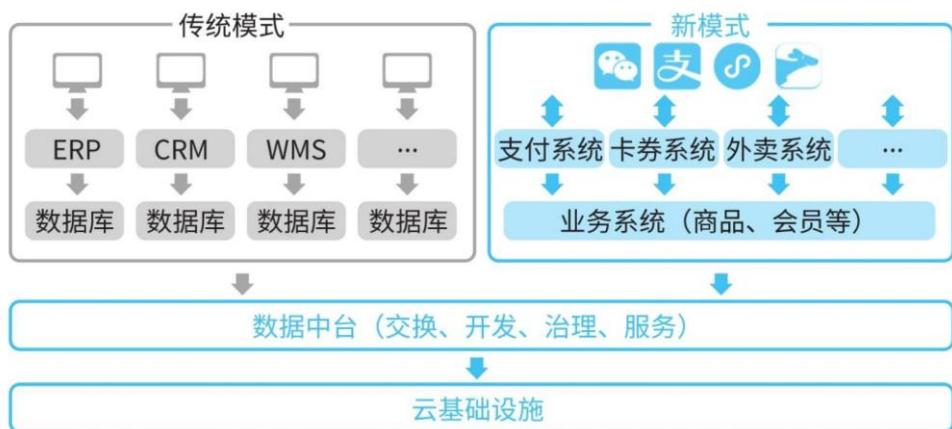
5.7.2 flume 监控数据并传输到到 kafka.....	249
5.8 DolphinScheduler 任务调度系统实操.....	253
5.8.1 部署.....	253
5.8.2 网页端查看.....	257

1. 前言与背景

伴随着云计算、大数据、人工智能等技术的迅速发展，以及这些技术与传统行业的快速融合，企业数字化、智能化转型的步伐逐渐加快。IDC 预测，到 2021 年，全球至少 50% 的 GDP 将被数字化，而每个行业的增长都会受到数字产品与服务、数据化运营的驱动。

数字化转型成功的企业，其内部和外部的交互均以数据为基础。业务的变化快速反馈在数据上，企业能够迅速感知并做出反应，而其决策与考核基于客观数据。同时，数据是活的，是流动的，越用越多，越用越有价值。随着数据与业务场景的不断交融，业务场景将逐步实现通过数据自动运转和自动优化，进而推动企业进入数字化和智能化的阶段。

传统 IT 建设方式下，企业的各种信息系统大多是独立采购或者独立建设的，无法做到信息的互联互通，导致企业内部形成多个数据孤岛。互联网、移动互联网的发展带来很多新的业务模式，很多企业尝试通过服务号、小程序、O2O 平台等新模式触达客户、服务客户，新模式是通过新的平台支撑的，产生的数据与传统模式下的数据也无法互通，这进一步加剧了数据孤岛问题。分散在各个孤岛的数据无法很好地支撑企业的经营决策，也无法很好地应对快速变化的前端业务。因此需要一套机制，通过这套机制融合新老模式，整合分散在各个孤岛上的数据，快速形成数据服务能力，为企业经营决策、精细化运营提供支撑，这套机制就是数据中台，如图所示。



2. 数据中台研究

数据中台不仅仅是技术，也不仅仅是产品，而是一套可持续“让企业的数据用起来”的机制，是一种战略选择和组织形式，是依据企业特有的业务模式和组织架构，通过有形的产品和实施方法论支撑，构建的一套持续不断把数据变成资产并服务于业务的机制。通过数据中台把数据变为一种服务能力，既能提升管理、决策水平，又能直接支撑企业业务。

2.1 数据中台的作用

数据中台是位于底层存储计算平台与上层的数据应用之间的一整套体系。数据中台屏蔽掉底层存储平台的计算技术复杂性，降低对技术人才的需求，让数据的使用成本更低。

- 通过数据中台的数据汇聚、数据开发模块建立企业数据资产。
- 通过资产管理与治理、数据服务把数据资产变为数据服务能力，服务于企业业务。

图为数据中台的生态体系图



数据中台生态体系图

1. 数据汇聚

数据汇聚是数据中台数据接入的入口。数据中台本身几乎不产生数据，所有数据来自于业务系统、日志、文件、网络等。数据汇聚是指采集各条产品线的数据，并将这些数据集中处理，存放在数据中心。数据汇聚方式一般有数据库同步、埋点、网络爬虫、消息队列等；从汇聚的时效性来分，有离线批量汇聚和实时采集。

2. 数据开发

数据开发的重点是数据存储、计算和打通，业内一般采用分层建模的存储方式，让采集上来的数据变成公司的数据资产。而就打通来说有两方面，一方面要打通用户的行为数据和用户的业务数据，从而构建更加丰富的用户画像；另一方面要打通产品线之间的数据，比如一个用户既用了A产品线的服务，又用了B产品线的服务，需要打通产品线才能挖掘出这些数据信息。

有经验的数据开发、算法建模人员利用数据加工模块提供的功能，可以快速把数据加工成对业务有价值的形式，提供给业务使用。数据开发模块主要面向开发人员、分析人员，提供离线、实时、算法开发工具，以及任务的管理、代码发布、运维、监控、告警等一系列集成工具，方便使用，提升效率。

数据体系有了数据汇聚、数据开发模块，中台已经具备传统数据仓库平台的基本能力，但注意数据要统一建设，建议数据按照贴源数据、统一数仓、标签数据、应用数据的标准统一建设。

3. 数据资产管理

数据资产管理包括对数据资产目录、元数据、数据质量、数据血缘、数据生命周期等进行管理和展示，以一种更直观的方式展现企业的数据资产，提升企业的数据意识。

4. 数据服务

中台产品可以带有一些标准服务，但是很难满足企业的服务诉求，大部分服务还是需要通过中台的能力快速定制。数据中台的服务模块并没有自带很多服务，而是提供快速的服务生成能力以及服务的管控、鉴权、计量等功能。

2.2 数据中台经典架构

数据中台经典架构如图所示



从下往上看，**第一层是数据采集层**。企业内每条产品线都会产生一定数量的业务数据，比如电商服务产品线的用户的加购（即将商品加入购物车）数据、收藏数据、下单数据等随着用户量的增大会越来越多，这些数据大部分被存储于业务数据库中；还有用户的浏览行为、点击行为，这些行为会做相应的埋点，产生的数据一般会以日志文件的形式存储。无论是业务数据库的数据还是日志文件的数据，我们都需要把它们抽取到数据中台中做统一的存放。一般数据工程师会用一些比较成熟的数据同步工具，将业务数据库的数据实时同步到数据中台，同时将离线日志数据以“T-1”的形式抽取过来，整合到一起。

第二层是数据计算层。数据中台要同步企业内多条产品线的数据，数据量相对来说是比较大的。海量的数据采用传统的存储方式是不合理的。业界一般采用分层建模的方式来存储海量数据。数据的分层主要包括操作数据层、维度数据层、明细数据层、汇总数据层和应用数据层等，通过分层建模可以令数据获得更高效、更科学的组织和存储。另外，为了保证数据指标的准确性和无歧义性，企业内部的数据指标需要通过一套严格的数据指标规范来管理，包括指标的定义、指标的业务口径、指标的技术口径、指标的计算周期和计算方式等。数据中台的产品人员、开发人员都要参考这套规范来工作，这样就能更大程度地保证数据的准确性和无歧义性。

第三层是数据服务层。数据经过整合计算后，如何被调取和使用呢？数据中台一般以接口的形式对外提供服务，开发人员将计算好的数据根据需求封装成一个个的接口，提供给数据产品和各条产品线调用。

第四层是数据应用层。数据产品分为几种：针对内部的数据产品、针对用户的数据产品、针对商家的数据产品。针对内部的数据产品一般用于服务公司的产品/运营人员和领导层。产品/运营人员更关注明细数据，比如，如果电商产品的活跃用户持续减少，数据中台如何通过数据帮助他们找出原因；领导层更关注大盘数据，比如根据公司近一年各条产品线的运营情况决定是否开发大屏类产品。针对用户的数据产品可以基于数据中台整合后的数据开发一些创新应用，比如个性化商品推荐，让“货找人”而不是“人找货”，提高了人货匹配的概率，同时也提高了用户的下单概率。针对商家的数据产品可以基于数据中台为商家提供数据服务，比如电商产品基于销售数据制作关于流行趋势、行情、店铺的数据报告等。

2.2.1 数据采集

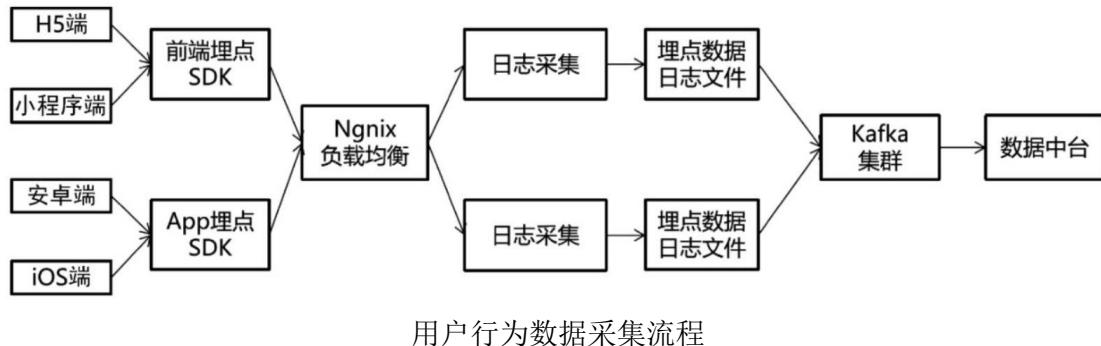
数据中台要采集的数据分为两种类型，一种是用户行为数据，一种是业务数据。

2.2.1.1 用户行为数据

用户无论在哪个客户端操作，产生的行为数据都分为两种：一种是浏览数据，一种是点击数据。这些隐性的行为数据，一般不会存储在产品线的数据库中，而是通过异步传输的方式传输并存储到数据采集服务器中。**用户行为数据的采集有如下三种方式：**（1）与第三方移动应用统计公司合作完成数据采集。（2）采用前后端

埋点结合的方式完成数据采集。(3) 采用可视化埋点与后端埋点结合的方式完成数据采集。

典型的用户行为数据采集流程，如图所示。



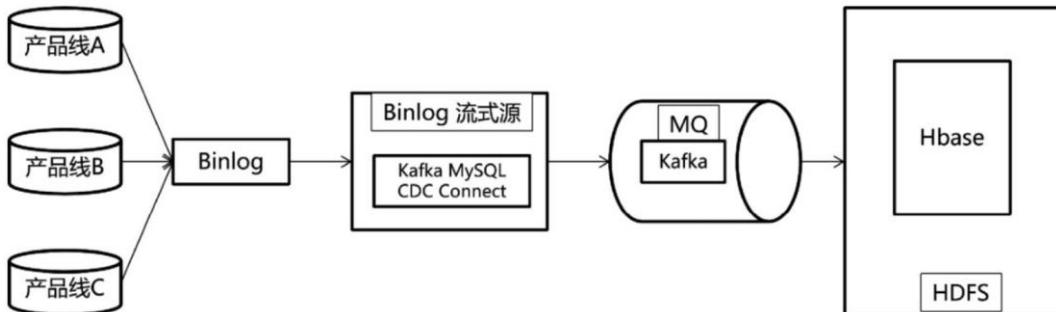
用户行为数据采集流程

步骤如下：(1) 前端工程师通过异步 HTTP 接口的形式将埋点数据发送到 Nginx 服务器，通过 Nginx 做负载均衡将日志文件采集并存储起来。(2) 通过如 Flume 之类的日志采集工具将埋点日志文件以消息的形式发送到 Kafka 集群。(3) 数据中台通过订阅 Kafka 集群的日志消息将日志文件存储在数据中台。

2.2.1.2 业务数据

业务数据一般存储在业务数据库或者业务中台中，业务数据库一般存放产品线的个性化的业务数据，业务中台一般存放通用的业务逻辑数据。比如对于电商产品来说，对坑位的管理、对活动页的管理属于个性化业务，其他产品线通常不会用到，这种业务的数据会存放在业务数据库中。

典型的业务数据采集流程，如图所示。



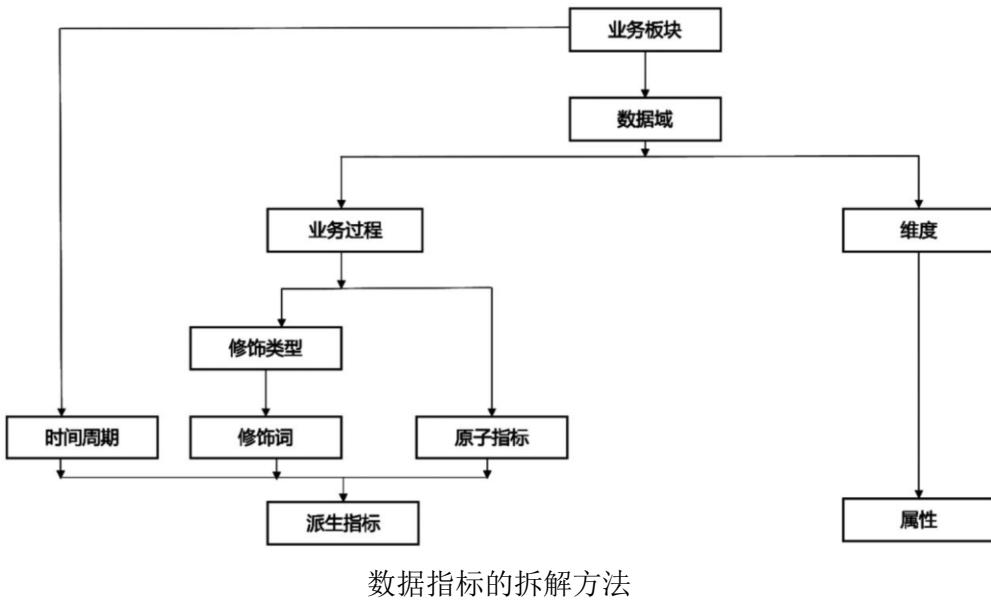
数据中台业务数据采集流程

流程如下：(1) 业务数据库产生的业务数据一般存储在关系型数据库（如 MySQL、Oracle 等）中，存储的过程会产生 Binlog 日志文件。Binlog 是一种二进制格式的文件，用于记录用户对数据库更新的 SQL 语句信息，例如更改数据库表和更改内容的 SQL 语句都会被记录到 Binlog 里，但是对库表等内容的查询不会被记录。(2) Kafka MySQL CDC Connect 读取了业务数据的 Binlog 文件后，通过 CDC（Change Data Capture，意为“变动数据捕获”，是增量抽取数据解决方案之一，能够帮助识别从上次提取之后发生变化的数据）的方式将业务库到 Kafka 等下游来消费。(3) 可以通过 Python 语言在 Kafka 消费基础上做一个调用，将生产者产生的数据消费到 Hbase。这样当 MySQL 中有相应操作时，Hbase 便会同步。

2.2.2 数据存储与计算

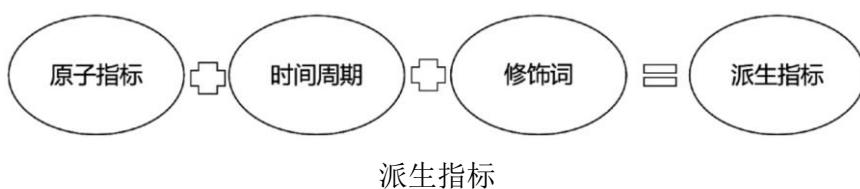
2.2.2.1 数据指标

要想实现数据中台项目，要做的第一件事就是梳理公司的数据指标体系。一个统一的标准可以提高公司内部沟通的效率，节省成本。数据指标的设定原则是：将一个数据指标拆解到不能再继续拆解为止，这样就能够最大限度地保证大家的理解无误。如图所示，这是一套指标拆解的方法。



数据指标的拆解方法

- (1) 业务板块：即面向业务的大模块，就是公司的产品线，不会经常变。比如一个公司有三条产品线分别是产品线 A、产品线 B、产品线 C，那么这三条产品线分别属于不同的业务板块。
- (2) 数据域：数据所属的领域。例如，电商产品中的用户、商品、交易等大的功能模块都属于数据域。
- (3) 业务过程：完成某个业务所涉及的全部过程。如电商业务中的下单、支付、退款等环节都属于业务过程。
- (4) 时间周期：就是统计的时间范围，如“近 30 天”“自然周”“截止到当天”等。
- (5) 修饰类型：对修饰词的描述。如电商中的支付方式、终端类型等。
- (6) 修饰词：除了维度以外的限定词，如电商支付中的微信支付、支付宝支付、网银支付等。
- (7) 原子指标：即不可再拆分的指标，比如支付金额、支付件数等指标。
- (8) 维度：是指度量单位，用来反映业务的一类属性。常见的维度有地理维度（国家、地区等）、时间维度（年、月、周、日等）、订单的维度等。
- (9) 属性：隶属于维度。如地理维度中的国家名称、省份名称等都属于属性。
- (10) 派生指标：一组对应的原子指标、修饰词、时间周期就组成了一个派生指标，如图所示。



派生指标

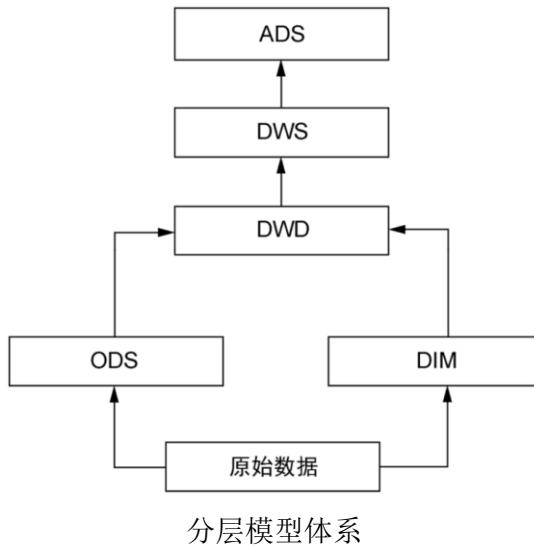
2.2.2.2 数据模型

学习数据模型首先要了解**数据仓库**。数据库和数据仓库虽然都是用来存储数据的，但数据库存储业务数据，而数据仓库存储汇总后的报表数据（如汇总各个产品线的销售数据），以支撑公司的决策分析。

一般来说，数据仓库的数据量是比较大的，而且其汇总统计的数据一般是不会再有变化的，比如公司 3 月份的交易额，这个汇总的数据不会因为新增的订单而变化。如果公司只有一条产品线，那么完全没有必要搭建数据仓库，基于数据库已经足够做统计分析了，但是当公司有多条产品线，而且要做大量的数据分析工作时，那就适合搭建数据仓库。

针对数据中台数据模型的分层，业界比较通用的分层方式是将数据模型分为 5 层：①ODS 层（Operate Data

Store, 操作数据层)、②DIM 层 (Dimension, 维度数据层)、③DWD 层 (Data Warehouse Detail, 明细数据层)、④DWS 层 (Data Warehouse Service, 汇总数据层)、⑤ADS 层 (Application Data Store, 应用数据层)。



第一层是 ODS 层和 DIM 层。 ODS 层数据是数据仓库的第一层数据，是业务数据库的原始数据的复制，例如，每条产品线的用户信息、订单信息等数据一般都是原封不动地同步到数据中台的 ODS 层中的。ODS 层的作用是在业务系统和数据仓库之间形成一个隔离层，在数据中台进行计算任务时，可以以 ODS 层的数据为基础进行计算，从而不给业务数据库增加负担。DIM 层存储的是维度数据，如城市、省份、客户端等维度的数据。

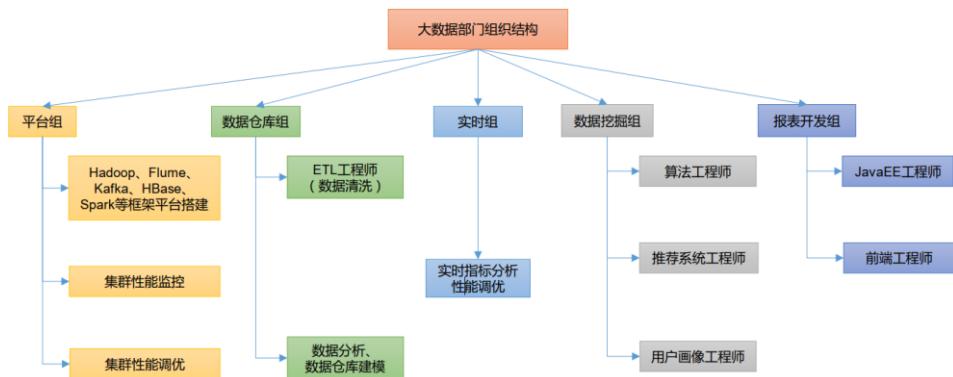
第二层是 DWD 层。 DWD 层数据是数据仓库的第二层数据，一般基于 ODS 层和 DIM 层的数据做轻度汇总。DWD 层存储经过处理后的标准数据，需要对 ODS 层数据进行再次清洗（如去空/脏数据、去超过极限的数据等操作）。DWD 层的结构和粒度一般与 ODS 层保持一致，但是 DWD 层汇总了 DIM 层的维度数据，比如在 ODS 层只能看到客户端的 ID 字段，但是在 DWD 层不但能看到客户端的 ID 字段，还能看到客户端的名称字段。

第三层是 DWS 层。 DWS 层数据是数据仓库的第三层数据，是以 DWD 层的数据为基础进行汇总计算的数据。DWS 层数据都是各个维度的汇总数据，比如某日某产品线的访问用户数、收藏用户数、加购用户数、下单用户数、支付用户数等。

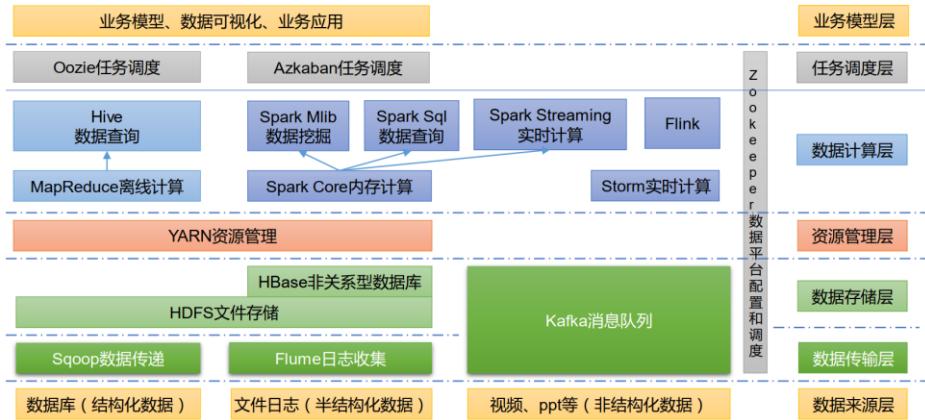
第四层是 ADS 层。 ADS 层数据是数据仓库的最后一层数据，以 DWS 层数据为基础进行数据处理。设计 ADS 层的最主要目的就是给数据可视化应用提供最终的数据。后端开发工程师基于 ADS 层的数据将最终数据结果以接口的形式展示给数据中台的应用层。

2.3 大数据技术生态

大数据 (Big Data) 指无法在一定时间范围内用常规软件工具进行捕捉、管理和处理的数据集合，是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。大数据主要解决，**海量数据的采集、存储和分析计算**问题。企业内的大数据部门内组织结构如下图所示。



目前业界的大数据开发生态系统如下图所示，相关技术栈介绍见下。

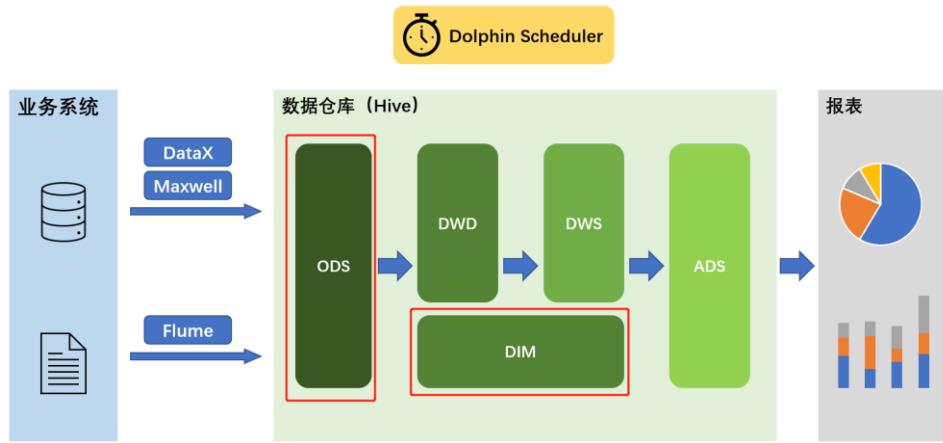


1. **Sqoop:** Sqoop 是一款开源的工具，主要用于在 Hadoop、Hive 与传统的数据库（MySQL）间进行数据的传递，可以将一个关系型数据库（例如：MySQL, Oracle 等）中的数据导进到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导进到关系型数据库中。
2. **Flume:** Flume 是一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume 支持在日志系统中定制各类数据发送方，用于收集数据。
3. **Kafka:** Kafka 是一种高吞吐量的分布式发布订阅消息系统。
4. **Spark:** Spark 是当前最流行的开源大数据内存计算框架。可以基于 Hadoop 上存储的大数据进行计算。
5. **Flink:** Flink 是当前最流行的开源大数据内存计算框架。用于实时计算的场景较多。
6. **Oozie:** Oozie 是一个管理 Hadoop 作业（job）的工作流程调度管理系统。
7. **Hbase:** HBase 是一个分布式的、面向列的开源数据库。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。
8. **Hive:** Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行。其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。
9. **ZooKeeper:** 它是一个针对大型分布式系统的可靠协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。

3. 数据仓库设计方法论

数据仓库是一个为数据分析而设计的企业级数据管理系统。数据仓库可集中、整合多个信息源的大量数据，借助数据仓库的分析能力，企业可从数据中获得宝贵的信息进而改进决策。同时，随着时间的推移，数据仓库中积累的大量历史数据对于数据科学家和业务分析师也是十分宝贵的。

3.1 核心架构



数据仓库的意义

数据模型就是数据组织和存储方法，它强调从业务、数据存取和使用角度合理存储数据。只有将数据有序的组织和存储起来之后，数据才能得到高性能、低成本、高效率、高质量的使用。

高性能：良好的数据模型能够帮助我们快速查询所需要的数据。

低成本：良好的数据模型能减少重复计算，实现计算结果的复用，降低计算成本。

高效率：良好的数据模型能极大的改善用户使用数据的体验，提高使用数据的效率。

高质量：良好的数据模型能改善数据统计口径的混乱，减少计算错误的可能性。

3.2 仓库建模

方法论

数据仓库之父 Bill Inmon 提出的建模方法是从全企业的高度，用实体关系（Entity Relationship, ER）模型来描述企业业务，并用规范化的方式表示出来，在范式理论上符合 3NF。

1) 实体关系模型

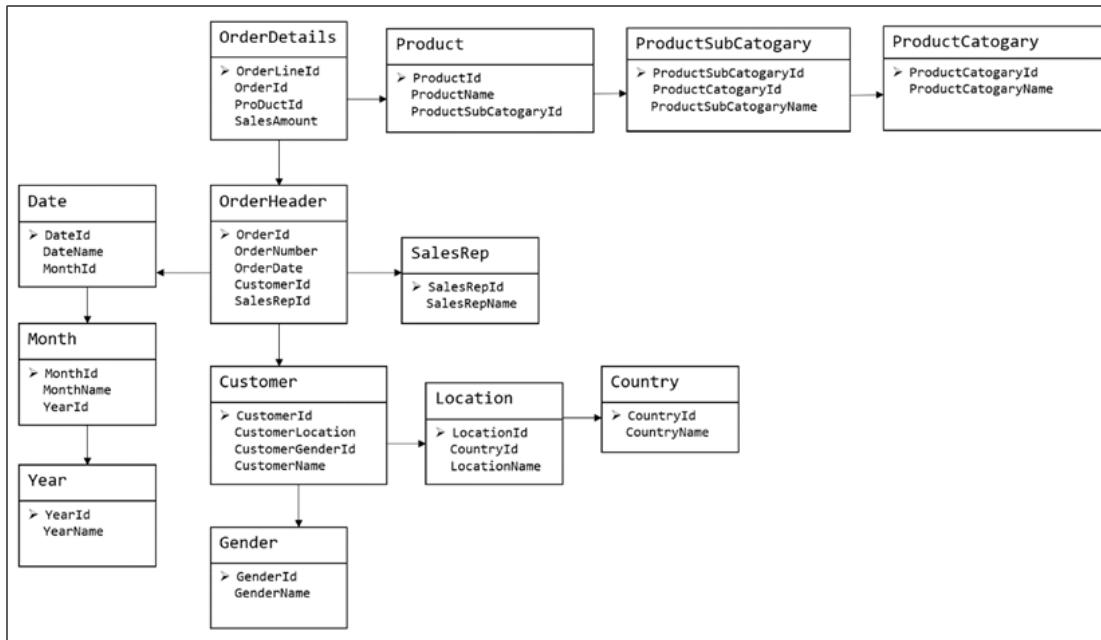
实体关系模型将复杂的数据抽象为两个概念——实体和关系。实体表示一个对象，例如学生、班级，关系是指两个实体之间的关系，例如学生和班级之间的从属关系。

2) 数据库规范化

数据库规范化是使用一系列范式设计数据库（通常是关系型数据库）的过程，其目的是减少数据冗余，增强数据的一致性。这一系列范式就是指在设计关系型数据库时，需要遵从的不同的规范。关系型数据库的范式一共有六种，分别是第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF) 和第五范式（5NF）。遵循的范式级别越高，数据冗余性就越低。

常规 ER 建模

下图为一个采用 Bill Inmon 倡导的建模方法构建的模型，从图中可以看出，较为松散、零碎，物理表数量多。



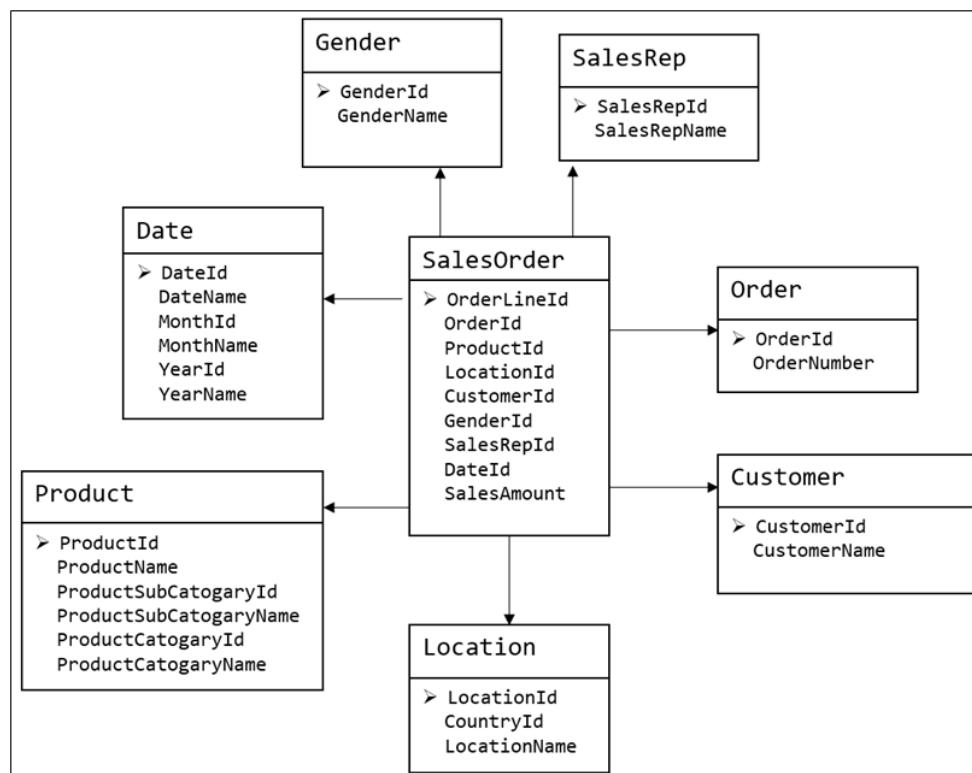
这种建模方法的出发点是整合数据，其目的是将整个企业的数据进行组合和合并，并进行规范处理，减少数据冗余性，保证数据的一致性。这种模型并不适合直接用于分析统计。

维度建模

数据仓库领域的另一位大师——Ralph Kimball 倡导的建模方法为**维度建模**。维度模型将复杂的业务通过**事实**和**维度**两个概念进行呈现。事实通常对应业务过程，而维度通常对应业务过程发生时所处的环境。

注：业务过程可以概括为一个个不可拆分的行为事件，例如电商交易中的下单，取消订单，付款，退单等，都是业务过程。

下图为一个典型的维度模型，其中位于中心的 SalesOrder 为事实表，其中保存的是下单这个业务过程的所有记录。位于周围每张表都是维度表，包括 Date (日期)，Customer (顾客)，Product (产品)，Location (地区) 等，这些维度表就组成了每个订单发生时所处的环境，即何人、何时、在何地下单了何种产品。从图中可以看出，模型相对清晰、简洁。



3.3 事实表概述

事实表作为数据仓库维度建模的核心，紧紧围绕着业务过程来设计。其包含与该业务过程有关的维度引用（维度表外键）以及该业务过程的度量（通常是可累加的数字类型字段）。

事实表通常比较“细长”，即列较少，但行较多，且行的增速快。

事实表有三种类型：分别是事务事实表、周期快照事实表和累积快照事实表，每种事实表都具有不同的特点和适用场景，下面逐个介绍。

事务型事实表

事务型事实表用来记录各业务过程，它保存的是各业务过程的原子操作事件，即最细粒度的操作事件。粒度是指事实表中一行数据所表达的业务细节程度。

事务型事实表可用于分析与各业务过程相关的各项统计指标，由于其保存了最细粒度的记录，可以提供最大限度的灵活性，可以支持无法预期的各种细节层次的统计需求。

周期型快照事实表

周期快照事实表以具有规律性的、可预见的时间间隔来记录事实，主要用于分析一些存量型（例如商品库存，账户余额）或者状态型（空气温度，行驶速度）指标。

对于商品库存、账户余额这些存量型指标，业务系统中通常就会计算并保存最新结果，所以定期同步一份全量数据到数据仓库，构建周期型快照事实表，就能轻松应对此类统计需求，而无需再对事务型事实表中大量的历史记录进行聚合了。

对于空气温度、行驶速度这些状态型指标，由于它们的值往往是连续的，我们无法捕获其变动的原子事务操作，所以无法使用事务型事实表统计此类需求。而只能定期对其进行采样，构建周期型快照事实表。

累积型快照事实表

累积快照事实表是基于一个业务流程中的多个关键业务过程联合处理而构建的事实表，如交易流程中的下单、支付、发货、确认收货业务过程。

累积型快照事实表通常具有多个日期字段，每个日期对应业务流程中的一个关键业务过程（里程碑）。

订单 id [↓]	用户 id [↓]	下单日期 [↓]	支付日期 [↓]	发货日期 [↓]	确认收货日期 [↓]	订单金额 [↓]	支付金额 [↓]
1001 [↓]	1234 [↓]	2020-06-14 [↓]	2020-06-15 [↓]	2020-06-16 [↓]	2020-06-17 [↓]	1000 [↓]	1000 [↓]

累积型快照事实表主要用于分析业务过程（里程碑）之间的时间间隔等需求。例如前文提到的用户下单到支付的平均时间间隔，使用累积型快照事实表进行统计，就能避免两个事务事实表的关联操作，从而变得十分简单高效。

3.4 维度表

维度表是维度建模的基础和灵魂。前文提到，事实表紧紧围绕业务过程进行设计，而维度表则围绕业务过程所处的环境进行设计。维度表主要包含一个主键和各种维度字段，维度字段称为维度属性。

3.4.1 维度表设计步骤

1) 确定维度（表）

在设计事实表时，已经确定了与每个事实表相关的维度，理论上每个相关维度均需对应一张维度表。需要注意到，可能存在多个事实表与同一个维度都相关的情况，这种情况需保证维度的唯一性，即只创建一张维度表。另外，如果某些维度表的维度属性很少，例如只有一个**名称，则可不创建该维度表，而把该表的维度属性直接增加到与之相关的事实事表中，这个操作称为**维度退化**。

2) 确定主维表和相关维表

此处的主维表和相关维表均指**业务系统**中与某维度相关的表。例如业务系统中与商品相关的表有 `sku_info`, `spu_info`, `baseTrademark`, `base_category3`, `base_category2`, `base_category1` 等, 其中 `sku_info` 就称为商品维度的主维表, 其余表称为商品维度的相关维表。维度表的粒度通常与主维表相同。

3) 确定维度属性

确定维度属性即确定维度表字段。维度属性主要来自于业务系统中与该维度对应的主维表和相关维表。维度属性可直接从主维表或相关维表中选择, 也可通过进一步加工得到。

确定维度属性时, 需要遵循以下要求:

(1) 尽可能生成丰富的维度属性

维度属性是后续做分析统计时的查询约束条件、分组字段的基本来源, 是数据易用性的关键。维度属性的丰富程度直接影响到数据模型能够支持的指标的丰富程度。

(2) 尽量不使用编码, 而使用明确的文字说明, 一般可以编码和文字共存。

(3) 尽量沉淀出通用的维度属性

有些维度属性的获取需要进行比较复杂的逻辑处理, 例如需要通过多个字段拼接得到。为避免后续每次使用时的重复处理, 可将这些维度属性沉淀到维度表中。

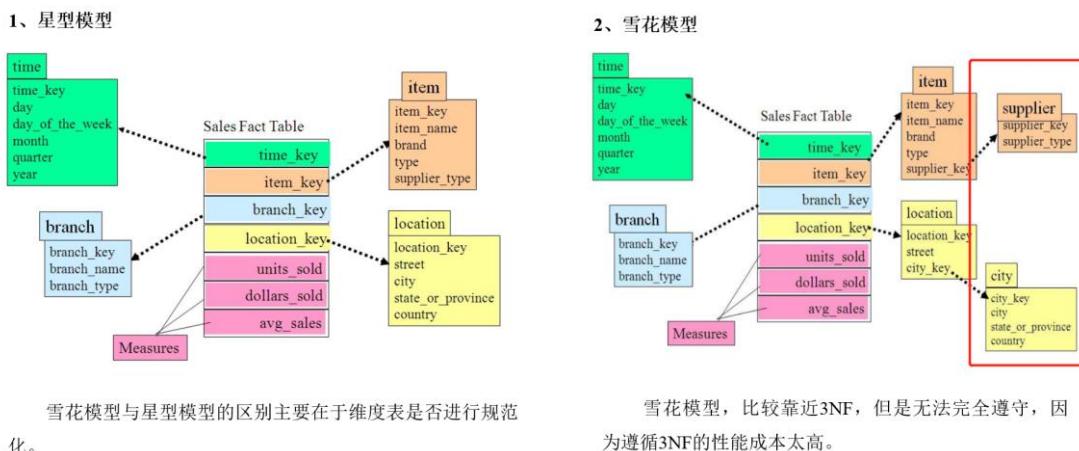
3.4.2 维度设计要点

3.4.2.1 规范化与反规范化

规范化是指使用一系列范式设计数据库的过程, 其目的是减少数据冗余, 增强数据的一致性。通常情况下, 规范化之后, 一张表的字段会拆分到多张表。

反规范化是指将多张表的数据冗余到一张表, 其目的是减少 join 操作, 提高查询性能。

在设计维度表时, 如果对其进行规范化, 得到的维度模型称为雪花模型, 如果对其进行反规范化, 得到的模型称为星型模型。



数据仓库系统的主要目的是用于数据分析和统计, 所以是否方便用户进行统计分析决定了模型的优劣。采用雪花模型, 用户在统计分析的过程中需要大量的关联操作, 使用复杂度高, 同时查询性能很差, 而采用星型模型, 则方便、易用且性能好。所以出于易用性和性能的考虑, 维度表一般是很不规范化的。

3.4.2.2 维度变化

维度属性通常不是静态的, 而是会随时间变化的, 数据仓库的一个重要特点就是反映历史的变化, 所以如何保存维度的历史状态是维度设计的重要工作之一。保存维度数据的历史状态, 通常有以下两种做法, 分别是全量快照表和拉链表。

1) 全量快照表

离线数据仓库的计算周期通常为每天一次，所以可以每天保存一份全量的维度数据。这种方式的优点和缺点都很明显。

优点是简单而有效，开发和维护成本低，且方便理解和使用。

缺点是浪费存储空间，尤其是当数据的变化比例比较低时。

2) 拉链表

拉链表的意义就在于能够更加高效的保存维度信息的历史状态。

(1) 什么是拉链表

拉链表，记录每条信息的生命周期，一旦一条记录的生命周期结束，就重新开始一条新的记录，并把当前日期放入生效开始日期。

如果当前信息至今有效，在生效结束日期中填入一个极大值（如9999-12-31）。

用户ID	姓名	手机号码	开始日期	结束日期
1	张三	136****9090	2019-01-01	2019-01-01
1	张三	137****8989	2019-01-02	2019-01-09
1	张三	147****1234	2019-01-10	9999-12-31

(2) 为什么要做拉链表

拉链表适合于：数据会发生变化，但是变化频率并不高的维度（即：缓慢变化维）。

比如：用户信息会发生变化，但是每天变化的比例不高。如果数据量有一定规模，按照每日全量的方式保存效率很低。比如：1亿用户*365天，每天一份用户信息。（做每日全量效率低）

每日全量表

用户ID	姓名	手机号码	dt
1	张三	136****9090	2019-01-01
1	张三	136****9090	2019-01-02
1	张三	136****9090	2019-01-03
...
1	张三	136****9090	2019-05-11
1	张三	155****1212	2019-05-12

拉链表

用户ID	姓名	手机号码	开始日期	结束日期
1	张三	136****9090	2019-01-01	2019-05-11
1	张三	155****1212	2019-05-12	9999-12-31

(3) 如何使用拉链表

通过，`生效开始日期<=某个日期`且`生效结束日期>=某个日期`，能够得到某个时间点的数据全量切片。

1) 拉链表数据

用户ID	姓名	开始时间	结束时间
1	张三	2019-01-01	9999-12-31
2	李四	2019-01-01	2019-01-02
2	李小四	2019-01-03	9999-12-31
3	王五	2019-01-01	9999-12-31
4	赵六	2019-01-02	9999-12-31

2) 例如获取`2019-01-01`的历史切片：`select * from user_info where start_date<='2019-01-01' and end_date>='2019-01-01'`

用户ID	姓名	开始时间	结束时间
1	张三	2019-01-01	9999-12-31
2	李四	2019-01-01	2019-01-02
3	王五	2019-01-01	9999-12-31

3) 例如获取`2019-01-02`的历史切片：`select * from order_info where start_date<='2019-01-02' and end_date>='2019-01-02'`

用户ID	姓名	开始时间	结束时间
1	张三	2019-01-01	9999-12-31
2	李四	2019-01-01	2019-01-02
3	王五	2019-01-01	9999-12-31
4	赵六	2019-01-02	9999-12-31

3.4.2.3 多值维度

如果事实表中一条记录在某个维度表中有多条记录与之对应，称为多值维度。例如，下单事实表中的一条记录为一个订单，一个订单可能包含多个商品，所会商品维度表中就可能有多条数据与之对应。

针对这种情况，通常采用以下两种方案解决。

第一种：降低事实表的粒度，例如将订单事实表的粒度由一个订单降低为一个订单中的一个商品项。

第二种：在事实表中采用多字段保存多个维度值，每个字段保存一个维度 id。这种方案只适用于多值维度个数固定的情况。

建议尽量采用第一种方案解决多值维度问题。

3.4.2.4 多值属性

维表中的某个属性同时有多个值，称之为“多值属性”，例如商品维度的平台属性和销售属性，每个商品均有多个属性值。

针对这种情况，通常可以采用以下两种方案。

第一种：将多值属性放到一个字段，该字段内容为 `key1:value1, key2:value2` 的形式，例如一个手机商品的平台属性值为“品牌:华为，系统:鸿蒙，CPU:麒麟 990”。

第二种：将多值属性放到多个字段，每个字段对应一个属性。这种方案只适用于多值属性个数固定的情况。

3.5 数据仓库分层规划

优秀可靠的数仓体系，需要良好的数据分层结构。合理的分层，能够使数据体系更加清晰，使复杂问题得以简化。以下是该项目的分层规划。

数据应用层 (ADS)

存放各项统计指标结果。

汇总数据层 (DWS)

基于上层的指标需求，以分析的主题对象作为建模驱动，构建公共统计粒度的汇总表。

公共维度层 (DIM)

基于维度建模理论进行构建，存放维度模型中的维度表，保存一致性维度信息。

明细数据层 (DWD)

基于维度建模理论进行构建，存放维度模型中的事实表，保存各业务过程最小粒度的操作记录。

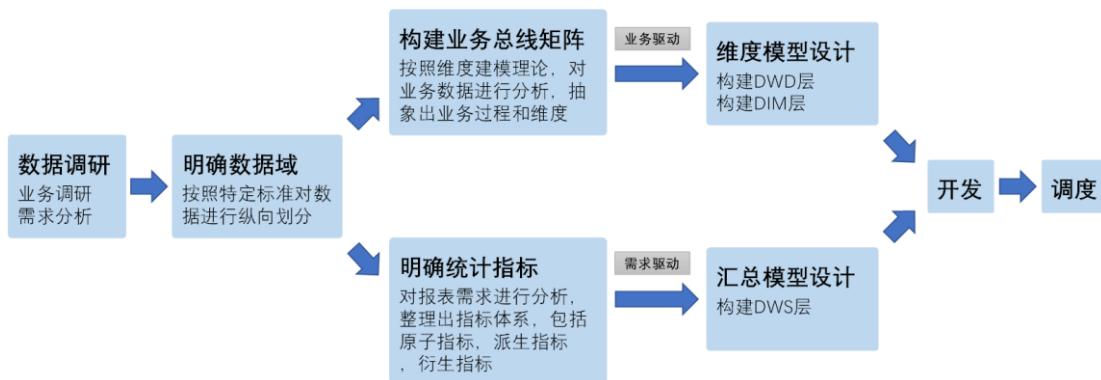
原始数据层 (ODS)

存放未经过处理的原始数据，结构上与源系统保持一致，是数据仓库的数据准备区。

分层简称	全称
ODS	Operation Data Store
DWD	Data Warehouse Detail
DIM	Dimension
DWS	Data Warehouse Summary
ADS	Application Data Service

数据仓库构建流程

以下是构建数据仓库的完整流程。



3.5.1 数据调研

数据调研重点要做两项工作，分别是业务调研和需求分析。这两项工作做的是否充分，直接影响着数据仓库的质量。

1) 业务调研

业务调研的主要目标是熟悉业务流程、熟悉业务数据。

熟悉业务流程要求做到，明确每个业务的具体流程，需要将该业务所包含的每个**业务过程**一一列举出来。

熟悉业务数据要求做到，将数据（包括埋点日志和业务数据表）与业务过程对应起来，明确每个业务过程会对哪些表的数据产生影响，以及产生什么影响。产生的影响，需要具体到，是新增一条数据，还是修改一条数据，并且需要明确新增的内容或者是修改的逻辑。

2) 需求分析

典型的需求指标如，最近一天各省份手机品类订单总额。

分析需求时，需要明确需求所需的**业务过程**及**维度**，例如该需求所需的业务过程就是买家下单，所需的维度有日期，省份，商品品类。

3) 总结

做完业务分析和需求分析之后，要保证每个需求都能找到与之对应的业务过程及维度。若现有数据无法满足需求，则需要和业务方进行沟通，例如某个页面需要新增某个行为的埋点。

3.5.2 明确数据域

数据仓库模型设计除横向的分层外，通常也需要根据业务情况进行纵向划分数据域。

划分数据域的意义是**便于数据的管理和应用**。

通常可以根据业务过程或者部门进行划分，本项目根据业务过程进行划分，需要注意的是一个业务过程只能属于一个数据域。

下面是本数仓项目所需的所有业务过程及数据域划分详情。

数据域	业务过程
交易域	加购、下单、取消订单、支付成功、退单、退款成功
流量域	页面浏览、启动应用、动作、曝光、错误
用户域	注册、登录
互动域	收藏、评价
工具域	优惠券领取、优惠券使用（下单）、优惠券使用（支付）

3.5.3 构建业务总线矩阵

业务总线矩阵中包含维度模型所需的所有事实（业务过程）以及维度，以及各业务过程与各维度的关系。矩阵的行是一个个业务过程，矩阵的列是一个个的维度，行列的交点表示业务过程与维度的关系。

一个业务过程对应维度模型中一张事务型事实表，一个维度则对应维度模型中的一张维度表。所以构建业务总线矩阵的过程就是设计维度模型的过程。但是需要注意的是，总线矩阵中通常只包含事务型事实表，另外两种类型的事实表需单独设计。

按照事务型事实表的设计流程，**选择业务过程→声明粒度→确认维度→确认事实**，得到的最终的业务总线矩阵见以下表格。

数据域	业务过程	粒度	维度									度量
			时间	用户	商品	地区	活动(具体规则)	优惠券	支付方式	退单类型	退单原因类型	
交易域	加购物车	一次加购物车的操作	√	√	√							商品件数
	下单	一个订单中的一个商品项	√	√	√	√	√	√				下单件数/下单原始金额/下单最终金额/活动优惠金额/优惠券优惠金额
	取消订单	一次取消订单操作	√	√	√	√	√	√				下单件数/下单原始金额/下单最终金额/活动优惠金额/优惠券优惠金额
	支付成功	一个订单中的一个商品项的支付成功操作	√	√	√	√	√	√	√			支付件数/支付原始金额/支付最终金额/活动优惠金额/优惠券优惠金额
	退单	一次退货操作	√	√	√	√				√	√	退单件数/退单金额
	退款成功	一次退款成功操作	√	√	√	√			√			退款件数/退款金额
流量域	页面浏览	一次页面浏览记录	√	√		√						浏览时长
	动作	一次动作记录	√	√	√	√		√				无事实(次数1)
	曝光	一次曝光记录	√	√	√	√	√					无事实(次数1)
	启动应用	一次启动记录	√	√		√						无事实(次数1)
用户域	错误	一次错误记录	√	√								无事实(次数1)
	注册	一次注册操作	√	√								无事实(次数1)
	登录	一次登录操作	√	√		√						无事实(次数1)
工具域	领取优惠券	一次优惠券领取操作	√	√				√				无事实(次数1)
	使用优惠券(下单)	一次优惠券使用(下单)操作	√	√				√				无事实(次数1)
	使用优惠券(支付)	一次优惠券使用(支付)操作	√	√				√				无事实(次数1)
互动域	收藏商品	一次收藏商品操作	√	√	√							无事实(次数1)
	评价	一次取消收藏商品操作	√	√	√							无事实(次数1)

后续的 DWD 层以及 DIM 层的搭建需参考业务总线矩阵。

4. 组件研究与学习

4.1 Zookeeper

在 2000 年左右，Google、Yahoo 等大型互联网公司开始研究分布式系统的技术，并提出了 MapReduce、GFS 等重要的分布式系统框架。这些框架在分布式计算方面取得了很大的成功，但是在协调和管理方面仍然存在问题。

为了解决这些问题，Yahoo Labs 的一些研究人员开始着手研发一个新的分布式协调服务。2007 年，他们开发出了 Zookeeper，最初是为了解决 Hadoop 框架中的协调问题。随着时间的推移，Zookeeper 被广泛应用于各种分布式系统中，成为了分布式系统中重要的协调服务之一。

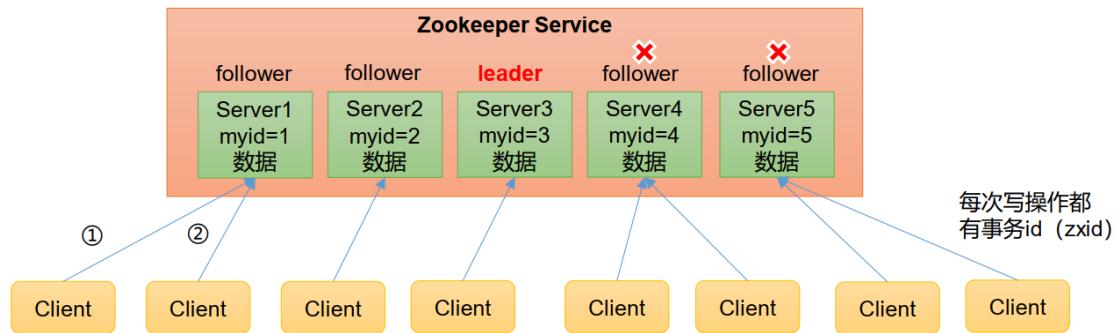
随后的 2007 年-2021 年，Zookeeper 进行了多次新版本的发布，拓展和优化了许多功能。2021 年 Zookeeper 宣布进入维护模式，不再发布新的功能版本，但会继续提供 bug 修复和安全更新。Zookeeper 经历了十多年的发展，成为了分布式系统中重要的协调服务之一，并且被广泛应用于各种大规模分布式系统的实现中。



4.1.1 基本介绍

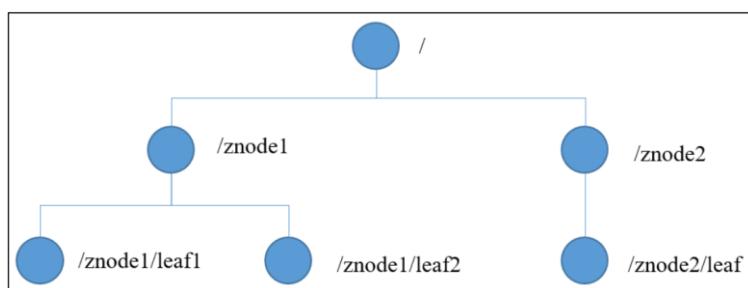
Zookeeper 是 Apache Hadoop 项目子项目，为分布式框架提供协调服务，是一个树形目录服务。Zookeeper 从设计模式角度来理解：是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生变化，Zookeeper 就将负责通知已经在 Zookeeper 上注册的那些观察者做出相应的反应。

4.1.2 Zookeeper 特点



- Zookeeper 是一个领导者（Leader），多个跟随者（Follower）组成的集群
- 集群中只要有半数以上节点存活就能正常服务，所以 Zookeeper 适合部署奇数台服务器
- 全局数据一致，每个 Server 保存一份相同的数据副本，Client 无论连接到哪个 Server，数据都是一致
- 更新的请求顺序执行，来自同一个 Client 的请求按其发送顺序依次执行
- 数据更新原子性，一次数据更新要么成功，要么失败
- 实时性，在一定的时间范围内，Client 能读到最新数据
- 心跳检测，会定时向各个服务提供者发送一个请求（实际上建立的是一个 Socket 长连接）

ZooKeeper 数据模型的结构与 Unix 文件系统很类似，整体上可以看作是一棵树，每个节点称做一个 ZNode。每一个 ZNode 默认能够存储 1MB 的数据，每个 ZNode 都可以通过其路径唯一标识。

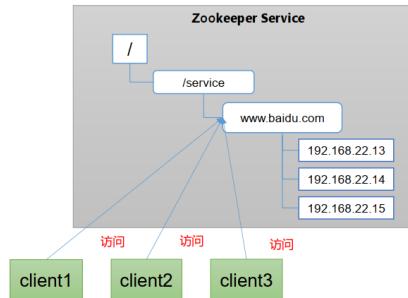


4.1.3 应用场景

提供的服务包括：统一命名服务、统一配置管理、统一集群管理、服务器节点动态上下线、软负载均衡等。

统一命名服务：在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。例如：IP不容易记住，而域名容易记住。

软负载均衡：在 Zookeeper 中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求。



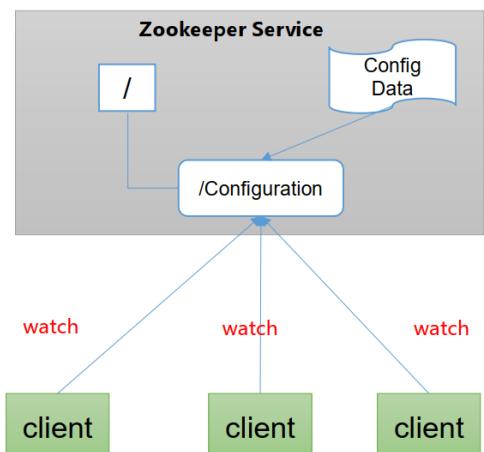
统一配置管理

分布式环境下，配置文件同步非常常见。

- 一般要求一个集群中，所有节点的配置信息是一致的，比如 Kafka 集群。
- 对配置文件修改后，希望能够快速同步到各个节点上。

配置管理可交由 ZooKeeper 实现。

- 可将配置信息写入 ZooKeeper 上的一个 Znode。
- 各个客户端服务器监听这个 Znode。
- 一旦 Znode 中的数据被修改，ZooKeeper 将通知各个客户端服务器。



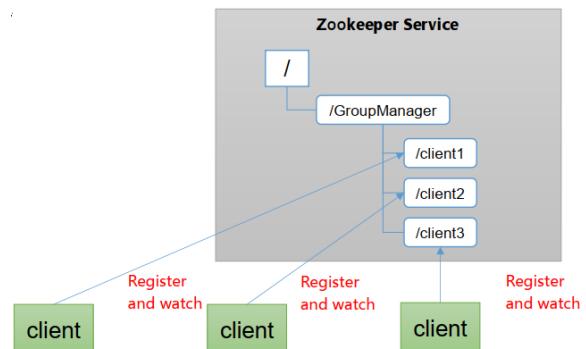
统一集群管理

分布式环境中，实时掌握每个节点的状态是必要的。

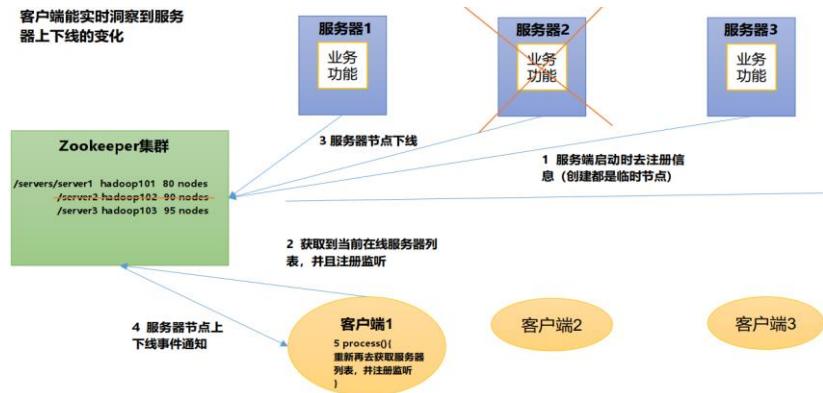
- 可根据节点实时做出一些调整。

ZooKeeper 可以实现实时监控节点状态变化。

- 可将节点信息写入 ZooKeeper 上的一个 ZNode。
- 监听这个 ZNode 可获取它的实时状态变化。



服务器动态上下线



4.1.4 集群选举机制

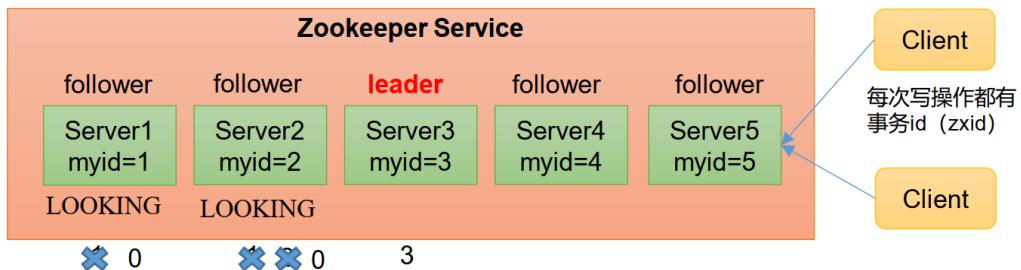
首先需要了解几个 ID 值的概念

SID: 服务器 ID。用来唯一标识一台 ZooKeeper 集群中的机器，每台机器不能重复，和 myid 一致。myid 是在 zookeepe 根目录/自建文件夹 目录下创建一个文件，内容只有一个数字，这就是本台主机的 ID 号。

ZXID: 事务 ID。ZXID 是一个事务 ID，用来标识一次服务器状态的变更。在某一时刻，集群中的每台机器的 ZXID 值不一定完全一致，这和 ZooKeeper 服务器对于客户端“更新请求”的处理逻辑有关。

Epoch: 每个 Leader 任期的代号。没有 Leader 时同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加。

第一次启动



(1) 服务器 1 启动，发起一次选举。服务器 1 投自己一票。此时服务器 1 票数 1 票，不够半数以上（3 票），选举无法完成，服务器 1 状态保持为 LOOKING；

(2) 服务器 2 启动，再发起一次选举。服务器 1 和 2 分别投自己一票并交换选票信息：此时服务器 1 发现服务器 2 的 myid 比自己目前投票推举的（服务器 1）大，更改选票为推举服务器 2。此时服务器 1 票数 0 票，服务器 2 票数 2 票，没有半数以上结果，选举无法完成，服务器 1, 2 状态保持 LOOKING

(3) 服务器 3 启动，发起一次选举。此时服务器 1 和 2 都会更改选票为服务器 3。此次投票结果：服务器 1 为 0 票，服务器 2 为 0 票，服务器 3 为 3 票。此时服务器 3 的票数已经超过半数，服务器 3 当选 Leader。服务器 1, 2 更改状态为 FOLLOWING，服务器 3 更改状态为 LEADING。

(4) 服务器 4 启动，发起一次选举。此时服务器 1, 2, 3 已经不是 LOOKING 状态，不会更改选票信息。交换选票信息结果：服务器 3 为 3 票，服务器 4 为 1 票。此时服务器 4 服从多数，更改选票信息为服务器 3，并更改状态为 FOLLOWING；

(5) 服务器 5 启动，同 4 一样当小弟。

非第一次启动

(1) 当 ZooKeeper 集群中的一台服务器出现以下两种情况之一时，就会开始进入 Leader 选举：

- 服务器初始化启动。
- 服务器运行期间无法和 Leader 保持连接。

(2) 而当一台机器进入 Leader 选举流程时，当前集群也可能会处于以下两种状态：

- 集群中本来就已经存在一个 Leader。对于第一种已经存在 Leader 的情况，机器试图去选举 Leader 时，会被告知当前服务器的 Leader 信息，对于该机器来说，仅仅需要和 Leader 机器建立连接，并进行状态同步即可。
- 集群中确实不存在 Leader。假设 ZooKeeper 由 5 台服务器组成，SID 分别为 1、2、3、4、5，ZXID 分别为 8、8、8、7、7，并且此时 SID 为 3 的服务器是 Leader。某一时刻，3 和 5 服务器出现故障，因此开始进行 Leader 选举。原则为：EPOCH 大的直接胜出 → EPOCH 相同，事务 id 大的胜出 → 事务 id 相同，服务器 id 大的胜出。SID 为 1、2、4 的机器投票情况如下：

	EPOCH	ZXID	SID	结果
SID1	1	8	1	✗
SID2	1	8	2	✓
SID4	1	7	4	✗

4.2 raft 与分布式思想

4.2.1 分布式系统是什么

分布式系统是一组电脑，透过网络相互连接传递消息与通信后并协调它们的行为而形成的系统。组件之间彼此进行交互以实现一个共同的目标。把需要进行大量计算的工程数据分割成小块，由多台计算机分别计算，再上传运算结果后，将结果统一合并得出数据结论。

其每个节点有如下特征：

- 不同的计算节点有其本地内存，但节点之间互相不共享
- 节点之间通过消息来传递信息

对于整个系统而言，有以下特征：

- 所处理的问题可分解为数个规模较小但求解过程相似的问题
- 整个系统可承受一定程度的节点失效（对于 raft 而言，n 个节点可忍受 n-1 个节点失效，对于 paxos 协议而言，2n+1 个节点能够忍受 n 个节点失效）

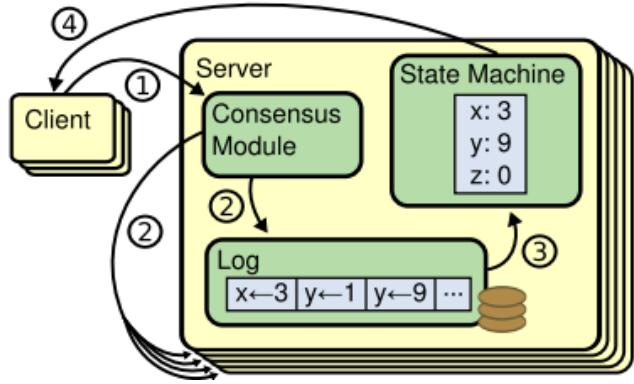
为什么需要分布式

- 如果不采用集群，算力增长和成本增长不呈现正比关系
- 高可靠性的硬件需要更高的费用
- 数据安全性需求，异地备份等
- 单台设备的 IO 限制，更高的并发 IO 需求

4.2.2 如何实现分布式计算

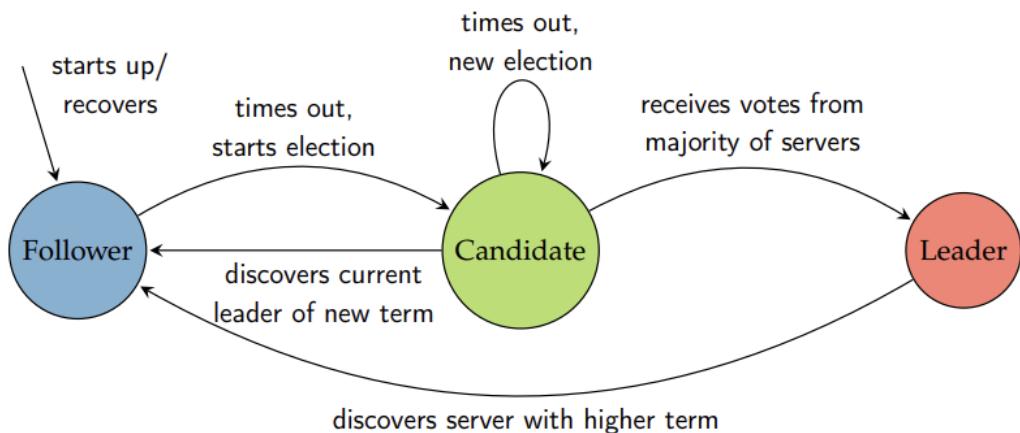
4.2.2.1 复制状态机（Replicated State Machine）

多个节点在初始状态相同时，执行一串相同的指令，最后的状态也应相同。Raft 算法的本质是维护一个具有最全面的信息的日志，并在每台设备上执行日志中的指令来确保所有设备的一致性。



4.2.2.2 Raft

Raft 是一种用于管理复制的共识算法日志。它对于一致性的实现和 paxos 一样好，并且它与 Paxos 一样高效，但它的结构不同来自 Paxos；这使得 Raft 比 Paxos 也为构建实用系统提供了更好的基础。为了增强可理解性，Raft 将共识的关键要素分离出来，比如领导者选举、日志复制和安全，它强制执行更强的一致性，以减少数量必须考虑的状态。用户研究的结果表明，raft 协议相对于 paxos 协议，其更为容易理解。



Follower: 被动节点，在超时前仅仅会对 Leader 的信号产生响应

Candidate: 活跃节点，尝试成为 Leader，并且会向其他节点发送选举信号（尝试获得其他节点的选票）

Leader: 领导整个集群的运行，并且作为唯一与外界交互的节点

初始态:

所有的节点均处于 Follower 状态，并随机设置一个超时等待时长（150ms-300ms）

选举时:

成为 Candidate 的节点向其他节点发送选举信号，请求其他节点为其投票，当其他节点收到该信号时会将 Candidate 节点的日志时间戳与自身日志时间戳进行比对，如果 Candidate 时间戳较大则接受该信号，反之则拒绝。当一个节点获得半数以上的选票时其成为新的 Leader，其余 Candidate 则放弃选举，否则所有节点重设超时计时器开始新一轮的选举。

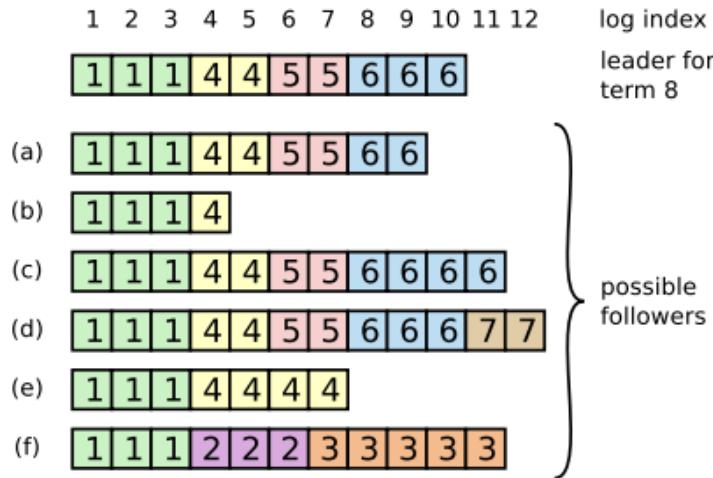
运行时:

成为 Leader 的节点负责作为对外交互的接口，并每隔一段时间向其余节点发送心跳信号，该过程是为了保证 Leader 节点的日志始终是最新且最全面的并成功同步至其余节点。当 Leader 宕机，其余节点接受不到心跳信号，在一定时间后（通常为 150ms—300ms 中的随机一个数值）转变为 Candidate 状态开始新一轮的选举。

日志更新:

一旦领导者被选出，他就开始响应客户需求。每个客户端请求都包含一个命令由复制的状态机执行。领导人将命令作为新条目附加到其日志中，然后与其他每个并行发出 AppendEntries RPC 服务器复制条目。当条目被

安全复制（如下所述），领导者申请进入其状态机并返回该状态机的结果执行给客户。如果随从崩溃或运行缓慢，或者如果网络数据包丢失，领导者会无限期地重试 AppendEntries RPC（即使在它已经响应客户端），直到所有追随者最终存储所有日志条目。



可能的代码实现：

```
C++
uint64_t raft::append_log_entry(const std::string & data) {
    if (cur_role_ != LEADER)
        return RC_NOT_LEADER;

    log_entry_sptr new_entry = make_log_entry(data.size());

    new_entry->idx = log_->last_entry_idx() + 1;
    new_entry->term = core_map()->cur_term;
    new_entry->cfg = false;
    log_->copy_log_data(new_entry.get(), data);

    LOG_INFO << "appending entry: " << LOG_ENTRY_TO_STRING(new_entry);

    if(log_->append(new_entry, true) != RC_GOOD)
        return 0;

    return new_entry->idx;
}
```

4.2.2.3 Paxos

概述

Paxos 是一系列协议，用于在不可靠或易出错的处理器网络中解决共识问题。共识是一组参与者就一个结果达成一致的过程。当参与者或他们的通信可能遇到故障时，这个问题就变得困难了。

共识协议是分布式计算的状态机复制方法的基础，正如 Leslie Lamport 所建议并由 Fred Schneider 调查的那样。状态机复制是一种将算法转换为容错的分布式实现的技术。临时技术可能会使重要的故障案例得不到解决。Lamport 等人提出的原则性方法。确保所有案件都得到安全处理。

Paxos 协议于 1989 年首次提交，并以希腊 Paxos 岛上使用的虚构成法共识系统命名，Lamport 在那里写道，议会必须运作“即使立法者不断进出议会会议厅”。它后来于 1998 年作为期刊文章发表。

Paxos 系列协议包括处理器数量、学习约定值之前的消息延迟数量、各个参与者的活动级别、发送的消息数量和失败类型之间的一系列权衡。尽管没有确定性的容错共识协议可以保证异步网络的进展（Fischer、Lynch 和 Paterson 的论文证明了这一结果），但 Paxos 保证了安全性（一致性），以及可能阻止其进展的条件很难挑

起。

Paxos 通常用于需要持久性的地方（例如，复制文件或数据库），其中持久状态的数量可能很大。即使在某些有限数量的副本无响应期间，该协议也会尝试取得进展。还有一种机制可以删除永久失败的副本或添加新副本。

历史变革

1. 初版 Paxos

Paxos 最初由 Leslie Lamport 在 1990 年提出，用于解决分布式系统中节点之间的数据一致性问题。初版 Paxos 仅仅是一个基本的算法框架，缺乏实际可行的实现细节。

2. Multi-Paxos

Multi-Paxos 是对 Paxos 的扩展，它可以更有效地执行复制状态机，并减少 Paxos 中消息传输的数量。它是将多个实例的 Paxos 运行合并在一起，以提高系统的效率和可扩展性。

3. Fast Paxos

Fast Paxos 是对 Paxos 的改进，它可以通过减少消息传输的数量和延迟来提高算法的性能。Fast Paxos 通过增加轻量级投票和依赖于更少的节点来实现这一点。

4. Generalized Paxos

Generalized Paxos 是对 Paxos 的进一步扩展，它提供了一种更通用的一致性协议，可用于处理复杂的数据结构，例如图和数据库。Generalized Paxos 通过引入“冲突仲裁者”来解决多个提案之间的竞争。

5. Multi-Leader Paxos

Multi-Leader Paxos 是对 Paxos 的扩展，它允许多个领导者同时提交提案。Multi-Leader Paxos 是为具有多个写入节点的系统设计的，允许并发提交的提案，并减少消息传输的数量。

paxos 基于的假设

处理单元

- 处理单元以任意速度运行。
- 处理单元可能会遇到故障。
- 具有稳定存储的处理单元可能会在发生故障后重新加入协议（遵循崩溃恢复故障模型）。
- 处理器不得串通、撒谎或以其他方式试图破坏协议。（也就是说，不会发生拜占庭故障。请参阅拜占庭 Paxos 以获取容忍由进程的任意/恶意行为引起的故障的解决方案。）

网络

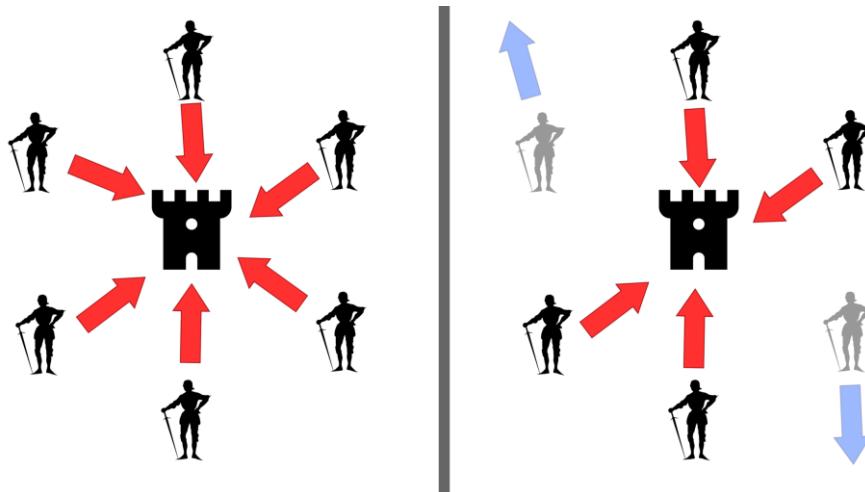
- 处理器可以向任何其他处理器发送消息。
- 消息是异步发送的，可能需要任意长的时间才能传递。
- 消息可能会丢失、重新排序或重复。
- 消息在没有损坏的情况下传递。（也就是说，不会发生拜占庭故障。请参阅拜占庭 Paxos 以获取容忍消息通道的任意/恶意行为引起的损坏消息的解决方案。）

处理器数量

一般来说，共识算法可以使用 $n=2F+1$ 个处理器，可以同时忍受 F 个处理器同时失败：换句话说，非故障进程的数量必须严格大于故障进程的数量。然而，使用重新配置，可以采用一种协议，只要不超过 F 次同时失败，该协议就能在任何数量的总失败中幸存下来。对于 Paxos 协议，这些重新配置可以作为单独的配置来处理。

4.2.2.4 拜占庭容错

在存在消息丢失的不可靠信道上试图通过消息传递的方式达到一致性是不可能的。因此在系统中存在除了消息延迟或不可送达的故障以外的错误，包括消息被篡改、节点不按照协议进行处理等，将会潜在地会对系统造成针对性的破坏。



解决方案：

如果忠诚的（非错误的）将军对他们的策略有多数同意，就可以实现拜占庭容错。可以为丢失的消息提供默认投票值。例如，可以为丢失的消息赋予“空”值。此外，如果协议是空票占多数，则可以使用预先指定的默认策略（例如撤退）。这个故事在计算机系统上的典型映射是，计算机是将军，他们的数字通信系统链接是信使。虽然这个问题在类比中被表述为决策和安全问题，但在电子学中，它不能仅通过加密数字签名来解决，因为不正确的电压等故障可以通过加密过程传播。因此，一个组件可能对一个组件起作用而对另一个组件出现故障，这会阻止就该组件是否有故障形成共识。

4.2.3 raft 在 kafka 中的实现与应用

4.2.3.1 概述

Apache Kafka 使用了一种名为 KRaft (Kafka Raft) 的协议，用于消除 Apache Kafka 对 ZooKeeper 的依赖性进行元数据管理，这大大简化了 Kafka 的架构，使元数据管理的责任集中在 Kafka 本身，而不是在 ZooKeeper 和 Kafka 之间分割。KRaft 模式利用 Kafka 中的新 quorum 控制器服务，该服务取代了以前的控制器，并利用了基于事件的 Raft 共识协议的变体。

KRaft 协议的事件驱动性质意味着，与基于 ZooKeeper 的控制器不同，quorum 控制器无需在变为活动状态之前从 ZooKeeper 加载状态。当领导权变更时，新的活动控制器已经在内存中拥有所有已提交的元数据记录。此外，KRaft 协议中用于跟踪集群元数据的同一事件驱动机制，现在受益于事件驱动以及用于通信的实际日志。

IP-500 的目标是去除 Kafka 对 ZooKeeper 的依赖，将元数据存储在 Kafka 自身中。这可以避免控制器状态与 ZooKeeper 状态之间的所有问题。而不是向 broker 推送通知，broker 应该从事件日志中消费元数据事件。这确保元数据更改总是按相同的顺序到达。broker 将能够在本地文件中存储元数据。当它们启动时，他们只需要从控制器读取已更改的内容，而不是完整的状态。这将使我们可以支持更多的分区，同时消耗更少的 CPU。

在新的架构中，三个控制器节点替代了三个 ZooKeeper 节点。控制器节点和 broker 节点在不同的 JVM 中运行。控制器节点为元数据分区选举一个领导者。与控制器推送更新给 broker 不同，broker 从这个领导者拉取元数据更新。

4.2.3.2 源码解析

Batch.java

该文件是 Kafka 的实现，定义了一个 Batch 类，用于保存一批记录。Batch 中包含了记录的 offset、epoch、append timestamp、size、last offset 和记录本身。Batch 类提供了一些创建 Batch 的静态方法：control() 用于创建 Control Batch，data() 用于创建 Data Batch。同时还实现了 Iterable 接口，可以迭代 Batch 中的所有记录。

BatchReader.java

这是一个 Java 接口文件，名称为 BatchReader.java。该接口定义了用于从 RaftClient 发送提交的数据到已注册的 RaftClient.Listener 实例的方法和访问器。BatchReader 接口的目的是在 IO 线程外推送阻塞操作，以确保缓慢的状态机不会影响复制。BatchReader 是一个迭代器，它定义了用于提供批次数据的方法，包括 baseOffset() 和 lastOffset()。BatchReader 还实现了 AutoCloseable 接口，它定义了一个 close() 方法，负责关闭读取器。

CandidateState.java

这是一个 Java 代码文件，文件名为“CandidateState.java”，它是 Apache Kafka 的 raft 协议的一部分。它实现了“EpochState”接口，表示该类是有关选举时的状态。它维护了一些变量和计时器，记录 Raft 协议中 Candidate 状态的生命周期，包括记录和处理选票和重试次数，检查是否已经被选为 Leader，以及进入退避阶段等。该类还提供了一些方法用于处理选票，检查候选人状态和计时器。

ControlRecord.java

这是一个 Java 类文件，文件名为 ControlRecord.java，位于 org.apache.kafka.raft 包中。这个类主要用于实现对 Kafka raft 协议中的控制记录进行处理。它包含一个构造函数，用于将消息与记录类型进行匹配，以及一些用于获取记录类型和消息版本的方法。此外，它还重写了 Object 类的 equals，hashCode 和 toString 方法。

ElectionState.java

该文件是一个 Java 类文件，位于 org.apache.kafka.raft 包中。该类是 Raft 选举状态的一个封装，用于在每个状态更改之后存储在磁盘上。它包含一些静态方法用于创建不同类型的选举状态，并提供一些方法用于查询节点是否为领导者或已投票的候选人，并返回选举状态的字符串形式。

EpochState.java

该文件是 Apache Kafka 中的 raft 模块中的核心接口文件 EpochState.java。它定义了一个 EpochState 接口来表示状态机当前的 Epoch 状态。该接口包含了一些抽象方法和默认实现，可以获得高水印、判断是否可以授予某个候选人投票权，并可以获取当前的选举状态和当前时刻 epoch 值。同时，实现该接口的类也可以搜索当前状态的简要说明。

ExpirationService.java

这是一个 Java 接口文件，定义了一个 Raft 协议的过期服务 ExpirationService。其中包含一个方法 failAfter(long timeoutMs)，用于创建一个新的 CompletableFuture 对象，如果在 timeoutMs 时间内未完成，则会生成一个异常，该异常为 org.apache.kafka.common.errors.TimeoutException。这个接口是为了在 Raft 协议中使用超时限制。

FileBasedStateStore.java

该程序文件是一个实现了 QuorumStateStore 接口的本地文件存储状态的类。该类可读取选举状态（ElectionState）以及写入最新的状态到本地文件中。该类利用 Json 格式存储数据，并且包含了数据版本号以进行易于反序列化。该类提供了清除状态的接口。

FollowerState.java

这个程序文件是 Java 语言编写的，位于 org.apache.kafka.raft 包中，并且实现了 EpochState 接口。该类表示 Kafka 中的一个 Follower 节点的状态，维护了当前节点的关键信息，如当前的领导者 ID、当前的投票人 ID 集合、高水位标记、当前从领导者获取日志的超时时间等等。另外，该类还实现了更新高水位标记、获取下一次超时时间、更新当前正在获取快照等方法，并提供了相应的日志输出。

Isolation.java

该文件是 raft 模块的一部分，包含了 Java 的一个枚举类型“Isolation”，该枚举类型包括两个值：

"COMMITTED"和"UNCOMMITTED"。它实现了 Raft 协议中的隔离（isolation）机制，用于描述提交和未提交的日志。此外，文件中还包含了版权信息和许可证信息。

KafkaRaftClient.java

这是一个实现 Kafka 式 Raft 协议的 Java 类，用于复制存储的日志。它实现了 Kafkaesque 版本的 Raft 协议，其中领导者选举是 Raft 协议的一个完备实现，但复制是通过副本获取驱动的，而我们使用 Kafka 的日志协调协议来截断日志，以使其符合选举后的共同点。它提供了以下 API：投票请求、开始新选举、结束当前选举、获取请求、快照请求等等。

LeaderAndEpoch.java

该代码文件为 Java 语言编写的 org.apache.kafka.raft.LeaderAndEpoch 类。它是一个成员变量为 OptionalInt 和 int 的类，分别表示领袖节点的 ID 和当前时期。它还提供了用于读取和操作这些成员变量的方法，例如 leaderId 和 epoch。此外，它还提供了 isLeader 方法，以判断给定节点是否为领袖节点。此类还包含重写的 equals、hashCode 和 toString 方法。有一个静态常量 UNKNOWN，表示未知的领袖和时期。

LeaderState.java

这是一个 Java 源文件，位于 org.apache.kafka.raft 包下。该文件定义了 Raft 协议中领导者的状态机，实现了 EpochState 接口，并包含了一些用于维护领导者状态、维护记录的方法和实例变量。其中，处理附加记录请求、提交记录和读取记录的主要工作是由 BatchAccumulator 完成的。

LogAppendInfo.java

该文件是 kafka 项目中 raft 模块的一个 Java 类，在类中定义了 LogAppendInfo 类，用于记录追加到日志中的批处理记录的元数据，该元数据包括第一个和最后一个偏移量。

LogFetchInfo.java

这个文件是 raft.zip.extract\src\main\java\org\apache\kafka\raft 目录下的 LogFetchInfo.java 文件。该文件定义了一个 LogFetchInfo 类，用于包含从日志中获取的记录的元数据，包括记录自身和开始偏移量。这个类提供了两个成员变量：records 和 startOffsetMetadata。构造函数可以用提供的参数来初始化这两个成员变量。该文件还导入了 org.apache.kafka.common.record.Records 包。

LogOffsetMetadata.java

该文件是 Kafka raft（一种分布式日志系统）中的一个 Java 类，名为 LogOffsetMetadata，用于存储特定本地日志偏移量的元数据。该类包含两个公共字段，一个是偏移量 offset (long)，另一个是一个可选的元数据 metadata (OffsetMetadata 类型)。该类还有两个公共构造函数，用于初始化 offset 和 metadata。此外，类还实现了 toString、equals 和 hashCode 方法。

NetworkChannel.java

该程序文件是一个 Java 接口，命名为 NetworkChannel，位于 org.apache.kafka.raft 包下。

这个接口声明了三个方法，分别是 newCorrelationId()、send() 和 updateEndpoint()，用于生成新的请求 ID、发出站请求和更新连接信息。

该接口还扩展了 Java 标准库中的 Closeable 接口，从而可以关闭该网络通道。

注释提到，该接口设计时没有假设请求的顺序或是每个出站请求都会收到响应，因此应该是一个非常简单的网络接口。

OffsetAndEpoch.java

该文件是 Kafka 中用于实现 raft 协议的一部分。

该文件定义了一个名为 OffsetAndEpoch 的类，它实现了 Comparable 接口和其中的 compareTo 方法，用于实现 OffsetAndEpoch 对象的比较。OffsetAndEpoch 类有两个变量：offset 和 epoch，分别表示时间戳和版本。同时，该类还包含了构造函数、getter 方法、equals 方法、hashCode 方法和 toString 方法。这些方法具有实现常规对象操作的功能。

OffsetMetadata.java

该文件是在 Apache Kafka 项目中的 RAFT 模块中定义的，它定义了一个接口"OffsetMetadata"，该接口用于在日志实现中创建一个不透明的元数据类型。适用于具有分布式复制功能的分布式系统。

QuorumState.java

该程序文件是实现一个 Raft 算法的状态机，用于管理节点的当前状态以及保证仅有效的状态转换。它定义了可能的状态转换以及如何触发它们，包括 Unattached、Voted、Candidate、Leader 和 Follower 等五个状态，以及 Observers 的状态转换。该程序文件还定义了一些过渡状态，如 Resigned 和 UnknownLeader 等。该文件由 Java 语言编写，使用了 Apache Kafka 和 SLF4J 日志库。

QuorumStateStore.java

这是 RAFT 协议的一个 Java 实现，其中定义了一个接口 QuorumStateStore，用于维护选举状态的读取和写入。该接口只支持读取和写入选举状态。其中定义了三个静态变量和三个方法，分别为读取最新的选举状态、写入最新的选举状态和清除所有存储状态以重新开始。

RaftClient.java

该文件是一个 Java 源代码文件，位于 raft.zip.extract\src\main\java\org\apache\kafka\raft 目录下。该文件定义了一个接口 RaftClient，它是一个 RAFT 协议的客户端。该客户端可以用于向分布式日志中追加记录，并可以接收来自分布式日志中已提交的记录。该接口提供了一组用于追加记录和处理已提交记录的方法，同时还提供了一组方法用于处理领导者的更改，以及节点的关闭和退出。

RaftConfig.java

该程序文件是 Raft 分布式一致性算法的配置文件，包含了 Raft quorum 投票节点的特定配置，包括投票节点的地址、选举超时时间、获取超时时间、backoff 机制等。同时，该文件也提供了一些用于解析投票节点连接字符串（如 1@localhost:9092,2@localhost:9093,3@localhost:9094）的方法，以便方便地配置投票节点。

RaftMessage.java

该文件为 Apache Kafka 中的 Raft 模块的一个接口文件，定义了 RaftMessage 接口，接口包含了两个方法，用于获取该消息的 correlationId 和 API 数据。该接口实现了 Apache Kafka 公共协议包中的 ApiMessage 接口。

RaftMessageQueue.java

这是一个 Java 接口文件，它定义了一个用于序列化 Raft 请求和响应的消息队列 RaftMessageQueue，并描述了它的四个方法。

- poll(long timeoutMs) 阻塞等待新消息的到来，如果超时将返回 null，或者在任何事件可用之前调用了 wakeup() 方法也返回 null。
- add(RaftMessage message) 向队列中添加新的消息。
- isEmpty() 检查队列中是否有待处理的消息。
- wakeup() 唤醒在 poll(long timeoutMs) 方法中阻塞的线程，如果没有任何消息可用，则导致返回 null。

该接口还提到了另一个类 org.apache.kafka.raft.internals.BlockingMessageQueue，该类封装了一个阻塞队列，可以用于模拟 RaftMessageQueue 的实现而不依赖于系统时间。

RaftRequest.java

该程序文件是一个 Java 类文件，文件名为 RaftRequest.java，位于文件路径 raft.zip.extract\src\main\java\org\apache\kafka\raft 中。该类属于 Kafka 项目中的 org.apache.kafka.raft 包，表示 Raft 协议中的一个请求。该类包含两个静态内部类 Inbound 和 Outbound，分别代表进入和离开节点的请求。该类还实现了 RaftMessage 接口，并包含了消息的关联 ID、数据内容以及创建时间属性，并且使用了 CompletableFuture 类来表示请求和回应之间的异步关系。

RaftResponse.java

该文件是 RaftResponse 类的源代码文件，它是 Kafka Raft 协议的响应消息类。该类定义了 RaftResponse 抽象类和其 Inbound 和 Outbound 内部类。RaftResponse 类实现了 RaftMessage 接口，并包含了与消息相关的 correlationId 和 data 成员变量，以及对应的构造方法和 getter 方法。Inbound 和 Outbound 类都继承自 RaftResponse 类，分别表示入站响应和出站响应，并分别包含了 sourceId 和 requestId 成员变量，以及对应的

构造方法和 getter 方法。文件中还包含了版权声明和许可证信息。

RaftUtil.java

该文件为 Kafka 中 Raft 模块的工具类。代码提供了一些实用方法，这些方法允许读取和修改 Kafka 中基于 Raft 的日志。给定一个 API 密钥和一个错误代码，它提供了一个错误响应。它还提供了一些方法来创建 Fetch 请求和响应，以及一些用于验证给定请求或响应中是否包含正确的 TopicPartition 的辅助方法。

ReplicatedCounter.java

这是一个 Java 类，文件名为 ReplicatedCounter.java。它的命名空间是 org.apache.kafka.raft，主要定义一个计数器，可以存储其值在 Apache Kafka 的 RAFT 协议集群中，这意味着在集群中多个副本之间进行复制，并确保最终一致性。与其它 Raft 协议中的典型实现不同，这个计数器的实现将值和元数据作为一个简单数字进行传输。例如，用户必须有效地对这个计数器进行 "claim" 或 "ack" 操作，这个 Raft 客户端库通过 Raft 协议与集群中的主节点进行通信。这些 "claim" 操作来自另一个计数器，必须来自此计数器之前的值。当 "claim" 被 Raft 集群中的主节点处理时，计数器值将被更新并复制到集群中的所有副本。如果主节点因某种原因下线，那么这个计数器将切换到一个新的主节点，当然，这个新节点必须同样的支持 RAFT 协议。

ReplicatedLog.java

该文件是 Apache Kafka 中 ReplicatedLog 接口的定义文件，定义了使用 Apache Kafka 实现的 Raft 协议实现的复制日志的操作和方法。它包含了写入、读取、删除、截断日志等方法的定义，以及元数据信息如日志末尾、高水位、日志开始的偏移和剪枝信息。此外，还有获取、存储和读取快照的方法。并且，该文件中还包含了对合法偏移量和纪元的验证方法。

RequestManager.java

该程序文件是一个 Java 类文件，名称为“RequestManager.java”，属于“org.apache.kafka.raft”包。这个类实现了一个“Raft”算法中的请求管理器，提供了处理与其他节点通信时的请求和相应的功能。这个请求管理器维护了各个节点之间的连接状态，处理请求的重试以及超时等情况，并提供了找到可用节点的功能。

ResignedState.java

此文件是 Apache Kafka 的一个模块，实现了一个 RAFT 分布式一致性算法中的一种状态，即“已辞职状态（Resigned State）”，当一位领袖因为某种原因放弃了它的领导地位但是又没有完全退出时，就会进入这种状态。这个状态不接受任何写操作并且不能为其他节点投票。当该领袖想要重新参与选举成为领袖时，就会给所有投票者发送“EndQuorumEpoch”请求，并在此状态中跟踪该请求的交付。本文件实现了一些与此状态相关功能，比如计时、挑选可接受“EndQuorumEpoch”请求（Preferred Successors），接收“EndQuorumEpoch”返回等。

VotedState.java

这是一个 Java 类文件，位于 org.apache.kafka.raft 包下，文件名为 VotedState.java。该类实现了 EpochState 接口，并表示投票状态。其中，该类的主要属性包括投票 ID，有限状态机的 epoch，选举超时时间及选举计时器、投票人集合等。该类提供了一些方法实现相关状态的操作，比如更新选举计时器、判断选举超时是否到达、是否能够授予某个候选人投票权等等。

NotLeaderException.java

它定义了一个“NotLeaderException”类，继承自“RaftException”类，并用于指示由于当前不是领导者或时代与当前领导者时代不同，因此不允许操作。该类有三个构造函数，分别采用不同的参数来构造异常对象。该文件属于 Kafka 项目中的 Raft 模块。

RaftException.java

该文件是名为 RaftException 的 Java 类，位于 org.apache.kafka.raft.errors 包中。该类继承了 KafkaException 类，并被用作 Kafka Raft 实现中生成的顶级异常类型。它包含三个构造函数，可以用来创建不同级别的异常，如有字符串消息的异常、带异常原因的异常、以及只有异常原因的异常。

BatchAccumulator.java

该文件是 Kafka 的 Raft 模块中的 BatchAccumulator.java 实现。它是一个批处理累加器，可以将一个列表中的记录追加到批量中，然后将该批量添加到处理队列中。可以使用附加或原子附加定义单个或连续批次，以使

用尽可能少的批次处理所有记录。它还定义了具有控制消息的批量的处理方法。此外，它实现了需要往下游 Drain batches 的 logic，并定义了线程安全模式。

BatchMemoryPool.java

该文件是一个 Java 类文件，文件名是 BatchMemoryPool.java。该类实现了 MemoryPool 接口，提供了一种简单的内存池实现，用于管理 ByteBuffer。BatchMemoryPool 类有以下主要特点：

- 该内存池是一个无界的内存池，使用双端队列来存储 ByteBuffer。
- tryAllocate()方法用于分配一个 ByteBuffer，如果内存池中有空闲的 ByteBuffer，则从内存池中返回；如果内存池中没有空闲的 ByteBuffer，则创建一个新的 ByteBuffer 并返回。
- release()方法用于释放先前分配的 ByteBuffer。如果内存池中的空闲 ByteBuffer 数量已达到最大值，则释放的 ByteBuffer 将会被系统回收；否则，该 ByteBuffer 将被重新放入内存池中等待下次使用。
- releaseRetained()方法用于释放内存池中所有的空闲 ByteBuffer。
- BatchMemoryPool 类使用 ReentrantLock 实现线程安全。

KafkaRaftMetrics.java

这是一个名为 KafkaRaftMetrics 的类，位于 org.apache.kafka.raft.internals 包中。此类实现了 AutoCloseable 接口，提供了一组度量 Raft 算法性能的方法。具体包括：记录当前节点的状态（leader、follower 等）、记录选举时间、记录提交日志时间、记录获取日志时间、记录附加日志时间、记录空轮询时间等。此类的主要功能是将这些度量标准委托给 Metrics 对象处理（Metrics 对象是与 KafkaMetrics 相似的对象，KafkaMetrics 用于记录 Kafka 服务器的度量标准）。

RecordsIterator.java

该文件是 Kafka Raft 模块中的 RecordsIterator 类。它提供了一个迭代器，可以迭代 Raft 协议的批次消息或快照记录，并反序列化这些消息。它使用了 Kafka 的共同协议，支持数据和控制记录，以及基于控制记录的类型分配操作。该类可以执行 CRC 校验，以确保读取的记录具有正确的头；也支持变更批次的大小，以尽可能利用内存。

CandidateStateTest.java

该程序文件是一个 Java 源代码文件，位于路径

"raft.zip.extract\src\test\java\org\apache\kafka\raft\CandidateStateTest.java"。它包含一个名为 CandidateStateTest 的 JUnit 测试类，用于测试 CandidateState 类的功能。CandidateState 类用于表示 Kafka 中的 RAFT 候选者状态。测试分别测试不同的投票情况下的机器人节点间状态变化，并测试各种异常条件。测试使用 MockTime 和 LogContext 类进行模拟，并使用 JUnit 的 assertions 断言库进行断言。

KafkaRaftClientTest.java

这个项目文件是 KafkaRaftClient 的单元测试代码，包含了各种针对该类中不同方法的测试用例。文件首先导入了各种依赖包，包括一些测试用例中需要用到的模拟工具，然后依次实现了测试初始化单节点集群、初始化作为 leader、拒绝同一时期内同一个节点的选票、拒绝另一个节点在同一时期的选票、接收来自更高选票节点 epoch 的授予选票等测试用例。每个测试用例的实现都通过 @Test 或 @ParameterizedTest 修饰。

LeaderStateTest.java

这是一个 Java 源代码文件，文件名为 LeaderStateTest.java，位于 raft.zip.extract\src\test\java\org\apache\kafka\raft 目录下。它是 Apache Kafka 中实现 Raft 协议的类的单元测试文件之一，测试了 LeaderState 类的多个方法，包括节点投票，高水位更新，节点更新状态等。测试用例覆盖了正常情况、非法情况以及边缘情况。

MockExpirationService.java

该文件是一个 Java 语言编写的单元测试类文件，位于 org.apache.kafka.raft 包下。该测试类实现了 ExpirationService 接口以及 MockTime.Listener 接口。它模拟了一个过期服务，使用了一个基于优先队列的延迟队列实现，来实现超时时间的判断。在过期时间到达时，将会抛出 TimeoutException 异常。

MockLog.java

该程序文件为一个 Java 代码文件，代码实现了一个类 MockLog，实现了接口 ReplicatedLog。其中，MockLog 是一个模拟的日志，用于测试和开发。该程序包含了在这个类中实现的一些方法，如 appendAsLeader、read、truncateTo 等。这些方法实现了将 MockLog 当做一个日志来使用的基本功能，如添加日志、读取日志等。

QuorumStateTest.java

该程序文件是 QuorumStateTest.java，位于 org.apache.kafka.raft 包下的 src\test\java 目录中。它是一个测试文件，用于测试 QuorumState 类的各种行为和状态转换。该类属于 Apache Kafka 中实现 Raft 协议的一部分，用于维护当前节点在 Raft 集群中的状态，并实现了选举和日志复制等功能。该测试文件包括多个测试用例，用于测试 QuorumState 的各种状态转换和行为，以确保其正确性和可靠性。

RaftClientTestContext.java

这个文件是一个 Java 测试文件，位于“org.apache.kafka.raft”包下。它包含一个 java 类 RaftClientTestContext，说明这是一个 RAFT（复制日志分区）相关的客户端测试。此类最重要的方法之一是“becomeLeader()”，它意味着如果它还不是领导者，它将切换成领导者，以进行后续测试。此文件还包含大量的辅助方法和构建器，用于模拟 RAFT 客户端的不同状态，其中一些方法被其他测试文件使用。

RaftEventSimulationTest.java

该文件是一个 Java 测试文件，用于测试 Kafka 的 Raft 协议实现。它包含了四个测试方法，测试集群 leader 选举、leader 失效、集群分裂、集群网络分区等场景下 Raft 协议的正确性和可用性。在测试中使用了模拟节点、消息路由、调度器等组件，通过模拟节点宕机、消息丢失、网络分区等不同的场景，验证 Raft 协议集群是否可以正确地选举出 leader，维持数据一致性，正确地进行数据复制和恢复等操作。

VotedStateTest.java

该测试文件用于测试 VotedState 类的不同方法的行为和输出。VotedState 类是与 Kafka Raft 协议相关的类，用于表示节点所在的协议状态，例如当前选举周期，已投票的节点 ID 和具有最大提交索引的日志偏移量元数据等。测试包括验证新创建的 VotedState 的各种属性，测试选举超时是否得到正确跟踪，以及测试是否正确评估是否要授予投票权。

4.2.3.3 总结

文件名	功能简述
NetworkChannel.java	实现网络通信接口
OffsetAndEpoch.java	定义了用于比较 Offset 和 Epoch 的类
OffsetMetadata.java	创建一个不透明的元数据类型
QuorumState.java	Raft 算法状态机，管理节点的当前状态和状态转换
QuorumStateStore.java	维护选举状态的读取和写入
RaftClient.java	实现 Raft 协议的客户端接口
RaftConfig.java	存储 Raft quorum 投票节点的特定配置
RaftMessage.java	Raft 协议的消息接口
RaftMessageQueue.java	序列化 Raft 请求和响应的消息队列
RaftRequest.java	Raft 协议中的请求接口
RaftResponse.java	Raft 协议中的响应接口
RaftUtil.java	实现处理 Raft 协议的工具类
ReplicatedCounter.java	存储计数器在 Raft 协议集群中，确保最终一致性
ReplicatedLog.java	实现基于 Raft 协议的日志操作
RequestManager.java	实现 Raft 算法中的请求管理器

这些文件实现了一个基于 Raft 协议的分布式一致性算法。

文件名	功能概述
UnattachedState.java	Raft 投票机制中的未投票状态的处理
ValidOffsetAndEpoch.java	包括 DIVERGING、SNAPSHOT、VALID 三种类型的记录偏移量和时代，用于 Raft 算法（分布式一致性算法）
VotedState.java	Raft 投票机制中的投票状态的处理
BufferAllocationException.java	当无法为操作分配内存而导致操作失败时抛出的异常
NotLeaderException.java	用于表示因不是领导者或时代与当前领导者时代不同而不允许执行操作的异常
RaftException.java	作为 Kafka Raft 实现中生成的顶级异常类型之一，用于表示异常的级别
BatchAccumulator.java	用于将单个或连续批次记录追加到批处理中，并将其添加到处理队列中的批处理累加器
BatchBuilder.java	用于将一批记录收集起来构建成一个批处理的类
BatchMemoryPool.java	管理 ByteBuffer 的内存池
BlockingMessageQueue.java	处理 RaftMessage 消息的阻塞队列
CloseListener.java	对象关闭时需要执行的操作接口
FuturePurgatory.java	用于管理追踪期望完成的 future 的一套 API
KafkaRaftMetrics.java	一组度量 Raft 算法性能的方法
MemoryBatchReader.java	用于读取共识日志中批量操作的内存数据
RecordsBatchReader.java	用于读取记录批次的类，支持数据和控制记录，并提供反序列化这些记录的功能
RecordsIterator.java	用于迭代 Raft 算法中的批次信息或快照记录，并反序列化这些消息的迭代器
程序整体功能	Kafka 的 Raft 模块实现了分布式一致性算法 Raft 的各种组件和相关工具。

文件路径	功能简述
org.apache.kafka.raft.internals.StringSerde.java	序列化和反序列化字符串数据
org.apache.kafka.raft.internals.ThresholdPurgatory.java	提供基于阈值的等待机制
org.apache.kafka.raft.internals.TimeRatio.java	维护特定事件的持续时间与总时间的比率

org.apache.kafka.snapshot.FileRawSnapshotReader.java	从文件中读取记录
org.apache.kafka.snapshot.FileRawSnapshotWriter.java	将一系列记录作为快照写入到存储中
org.apache.kafka.snapshot.RawSnapshotReader.java	读取不可变快照
org.apache.kafka.snapshot.RawSnapshotWriter.java	写入快照数据
org.apache.kafka.snapshot.RecordsSnapshotReader.java	从快照中读取记录
org.apache.kafka.snapshot.RecordsSnapshotWriter.java	将 MemoryRecords 或 UnalignedMemoryRecords 写入快照
org.apache.kafka.snapshot.SnapshotPath.java	表示快照在文件系统中的存在情况
org.apache.kafka.snapshot.SnapshotReader.java	读取不可变快照的类型
org.apache.kafka.snapshot.Snapshots.java	用于创建、操作和删除 Kafka 快照
org.apache.kafka.snapshot.SnapshotWriter.java	写入快照数据的接口
org.apache.kafka.ControlRecordTest.java	测试 ControlRecord 类是否符合预期行为
org.apache.kafka.raft.CandidateStateTest.java	测试 CandidateState 类的功能
org.apache.kafka.raft.FileBasedStateStoreTest.java	测试 FileBasedStateStore 的读写功能

该项目实现了 Kafka 的 Raft 模块，用于实现分布式一致性算法 Raft 的各种组件和相关工具。具体实现包括序列化和反序列化、基于阈值的等待、维护事件的持续时间与总时间的比率、从文件中读取记录、将记录作为快照写入到存储中、读取不可变快照、写入快照数据的接口、表示快照在文件系统中的存在情况、读取不可变快照的类型、用于创建、操作和删除 Kafka 快照等功能。

文件名	功能
ResignedStateTest.java	测试 Raft 协议中 follower 节点成为 leader 情况下的 ResignedState 类
UnattachedStateTest.java	测试 Raft 协议中节点尚未加入集群的状态的 UnattachedState 类
ValidOffsetAndEpochTest.java	测试 ValidOffsetAndEpoch 类的构造函数
VotedStateTest.java	测试 VotedState 类的不同方法的行为及输出
\BatchAccumulatorTest.java	测试 BatchAccumulator 类的方法
BatchBuilderTest.java	测试 BatchBuilder 类的功能
BatchMemoryPoolTest.java	测试 BatchMemoryPool 类的内存管理
BlockingMessageQueueTest.java	测试 BlockingMessageQueue 类的方法
KafkaRaftMetricsTest.java	测试 KafkaRaftMetrics 类的指标记录
MemoryBatchReaderTest.java	测试 MemoryBatchReader 类的方法
RecordsBatchReaderTest.java	测试 RecordsBatchReader 类的方法
RecordsIteratorTest.java	测试 RecordsIterator 类的方法及输出
ThresholdPurgatoryTest.java	测试 ThresholdPurgatory 类的行为

TimeRatioTest.java	测试 TimeRatio 类的性能指标
FileRawSnapshotTest.java	测试 FileRawSnapshotWriter 和 FileRawSnapshotReader 类的功能
MockRawSnapshotReader.java	模拟实现 RawSnapshotReader 接口，用于测试

这些文件主要用于测试和验证 Apache Kafka 项目的 Raft 协议的实现和相关工具的正确性和性能。

- 在 KRaft 模式中，Kafka 引入了一个新的服务，称为 quorum controller，它使用一种基于事件的 Raft 协议变体。quorum controllers 使用 KRaft 协议确保元数据在 quorum 中准确地复制。quorum controller 存储其状态使用的是基于事件的存储模型，这确保了可以始终准确地重新创建内部状态机。存储这个状态的事件日志（也被称为元数据主题）会定期通过快照进行简化，以确保日志不会无限增长。quorum 中的其他 controllers 会跟踪活动 controller，并回应它在日志中创建和存储的事件。这种方式大大降低了系统恢复的最坏情况下的时间。
- 另外，与基于 ZooKeeper 的 controller 不同，quorum controller 无需在变为活动状态之前从 ZooKeeper 加载状态。当领导权更改时，新的活动 controller 已经在内存中拥有所有已提交的元数据记录。

4.3 kafka

Kafka 是一个分布式流媒体平台，最初于 2010 年在领英开发，用于处理该公司社交网络生成的大量数据。Kafka 的设计是为了解决实时处理大规模数据处理的问题。Kafka 的设计者 Jay Kreps 选择以著名作家 Franz Kafka 的名字命名，是因为它是“一个用于优化写作的系统”，而且他很喜欢卡夫卡的作品。

Kafka 的开发始于领英的一个黑客马拉松项目，最初被命名为“伏地魔项目”。该项目由 Jay Kreps、Neha Narkhede 和 Jun Rao 领导，他们认为需要一个能够处理领英社交网络生成的大量数据的平台。他们想要一个能够处理实时处理并随着数据量的增长而横向扩展的系统。Kafka 的最初设计受到发布-订阅消息传递模型的影响，它的构建是为了处理数据的实时处理和批处理。该系统设计为具有高度可扩展性和容错性，支持跨多个节点的分布式处理。

2011 年，领英开源了 Kafka，并将其捐赠给了 Apache 软件基金会。该平台能够处理大容量实时数据流，因此很快在其他科技公司中广受欢迎。2012 年，Kafka 项目被转移到 Apache 软件基金会，成为 Apache 基金会的一个 top-level 项目，并于 2012 年 10 月 23 日由 [Apache Incubator](#) 孵化出站。

多年来，Kafka 经历了重大的发展，定期添加新的功能和改进。Kafka 添加的一些关键功能包括支持流处理、身份验证和授权等安全功能，以及改进的监控和管理工具。Kafka 的开发是由社区驱动的，来自世界各地许多开发者的贡献。2014 年，Confluent 公司成立，这是一家由最初的 Kafka 开发者创立的公司，旨在为 Kafka 提供商业支持和服务。

如今，各种规模的公司都广泛使用 Kafka 来处理实时数据处理和流处理。Kafka 已成为大数据生态系统不可或缺的一部分，并被用于许多行业，包括金融、医疗保健、零售和运输。该平台也已成为构建实时数据管道、事件驱动架构和流处理应用程序的热门选择。随着越来越多的公司采用实时数据处理来获得见解和做出更好的商业决策，它的受欢迎程度继续增长。

4.3.1 Kafka 概述



1.消息队列

消息队列是一种常见的通信模式，用于在不同的应用程序或系统之间传递数据。它是一种先进先出（FIFO）的数据结构，可以将消息保存在队列中，并在需要时按顺序处理它们。消息队列通常用于异步通信，其中发送方将消息放入队列中，而接收方则从队列中读取消息。

消息队列通常由一个消息代理或中介程序来管理，该程序充当发送方和接收方之间的桥梁。发送方将消息发送到代理，代理将消息存储在队列中，并确保消息在队列中保持顺序。接收方可以轮询代理以检查是否有新消息，并从队列中获取它们。这种异步通信模式可以帮助应用程序处理高负载和高并发情况，同时降低应用程序之间的依赖性和耦合性。

消息队列还可以提供可靠的消息传递保证，例如消息确认机制、重试机制和消息持久化等。这些功能可以确保即使在出现故障或网络中断的情况下，消息也不会丢失，并且可以重新发送或重新处理。

2.为什么需要消息队列

1. **解耦：**在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。
2. **副本：**有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的“插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。
3. **扩展性：**因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。
4. **灵活性&峰值处理能力：**在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果以为能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。
5. **可恢复性：**系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。
6. **顺序保证：**在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka 保证一个 Partition 内的消息的有序性。
7. **缓冲：**在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行——写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。
8. **异步通信：**很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

3.Kafka 简介

Kafka 是由 [Apache 软件基金会](#) 开发的一个开源流处理平台，由 [Scala](#) 和 [Java](#) 编写。Kafka 是一种高吞吐量的[分布式](#)发布订阅消息系统，它可以处理消费者在网站中的所有动作流数据。这种动作（[网页浏览](#)，搜索和其他用户的行动）是在现代网络上的许多社会功能的一个关键因素。这些数据通常是由于吞吐量的要求而通过处理日志和日志聚合来解决。对于像 [Hadoop](#) 一样的[日志](#)数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka 的目的是通过 [Hadoop](#) 的并行加载机制来统一线上和离线的消息处理，也是为了通过[集群](#)来提供实时的消息。

Kafka 最初是由 LinkedIn 公司开发，并于 2011 年初开源。2012 年 10 月从 Apache Incubator 毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。Kafka 是一个分布式消息队列，Kafka 对消息保存时根据 Topic 进行归类，发送消息者称为 Producer，消息接受者称为 Consumer，此外 kafka 集群有多个 kafka 实例组成，每个实例(server)称为 broker。无论是 kafka 集群，还是 consumer 都依赖于 zookeeper 集群保存一些 meta 信息，来保证系统可用性。

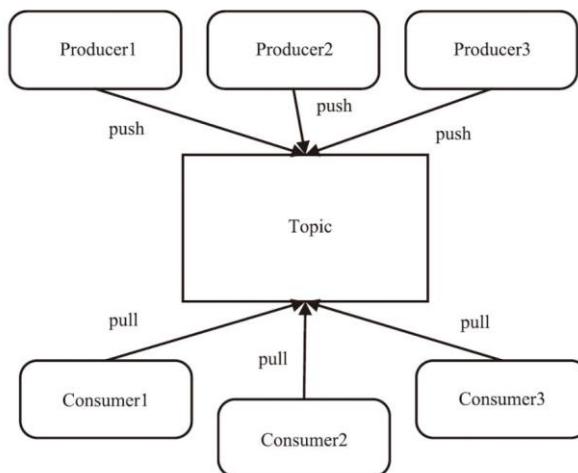
4.3.2 Kafka 优点

Kafka是大数据领域消息传输的杀手锏。这款专为大数据而生的消息中间件以其卓越的百万级TPS吞吐量而声名鹊起，迅速成为大数据领域的宠儿，在数据采集、传输和存储的过程中发挥着举足轻重的作用。Kafka的主要优点如下：

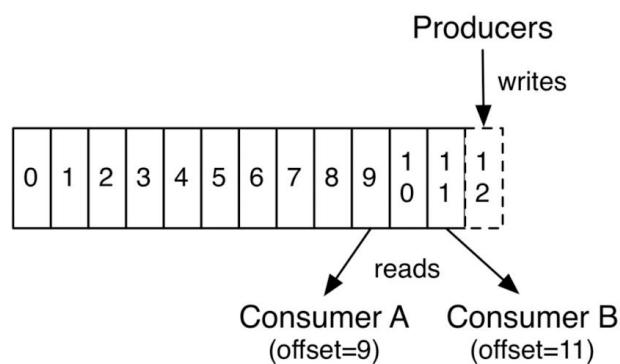
1. 高吞吐量和低延迟：Kafka在设计时就注重了高吞吐量和低延迟，它能够处理大量的数据，并且具有快速的响应时间。这使得Kafka非常适合处理大规模的数据流。
2. 分布式系统：Kafka是一种分布式系统，它可以扩展到多个服务器上，这使得Kafka能够处理大规模的数据流，并且保证了系统的可靠性和容错性。
3. 可靠性：Kafka采用了多种机制来确保消息的可靠性，例如备份、副本和恢复机制等。这些机制能够在系统故障或异常情况下，保证数据的完整性和可靠性。
4. 灵活性：Kafka具有非常高的灵活性，可以适用于各种不同的场景和应用。它可以作为消息队列、事件流处理器、日志聚合器等等，可以满足不同的需求。
5. 开源：Kafka是一个开源项目，它拥有庞大的社区和生态系统。这使得开发者可以通过社区的支持和贡献，得到更好的技术支持和使用体验。

4.3.3 Kafka常用术语

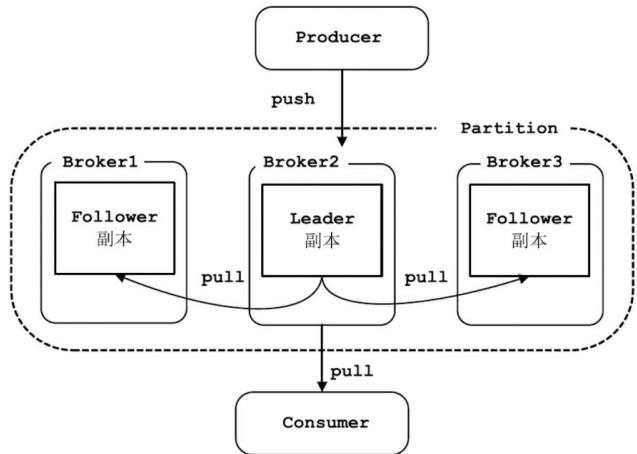
1. Message（消息）：消息是kafka的基本数据单元，代表着一条一条的数据，为了提高网络和存储的利用率，生产者会批量发送消息到Kafka，并在发送之前对消息进行压缩。
2. Topic（主题）：主题是kafka对消息的分类，是一个逻辑概念，可以看作消息集合，用于接收不同业务的消息。



3. Partition（分区）：类似数据库的分区表，通常topic下会有多个分区，每个分区内的数据是有序的，同一个topic的多个分区kafka不保证消息的顺序性，一个分区在逻辑上对应一个Log，对应磁盘上的一个文件夹。
4. Offset（偏移量）：表示消息在分区中的位置，kafka存储的文件是按照offset.log的格式来命名的，便于快速查找。



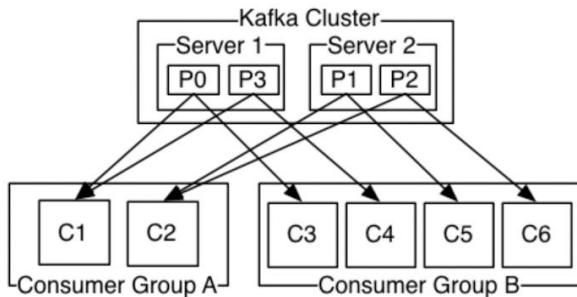
5. Replicas (副本): 该术语针对分区而言的, kafka 对消息做了冗余备份, 目的就是为了容灾, 每个分区可以指定多个副本来冗余数据, 分为 leader partition 和 follower partition, 只有 leader partition 对外提供服务, follower partition 单纯是从 leader partition 同步数据, 因此会存在多份相同的数据。



6. Producer (生产者): 生产者是 kafka 集群的上游, 顾名思义就是往 kafka 输入数据的客户端。

7. Consumer (消费者): 消费者是 kafka 集群的下游, 与生产者相辅相成, kafka 类似一个仓库, 生产者负责生产消息往仓库放, 自然得有消费者从仓库里拿消息, 不然仓库容易爆满。

8. Consumer Group (消费者组): 简称 CG, Kafka 将多个消费者捆绑起来, 组内的消费者共同消费消息, 一个 Consumer 只能属于一个 Consumer Group, Kafka 还通过 Consumer Group 实现了消费者的水平扩展和故障转移。

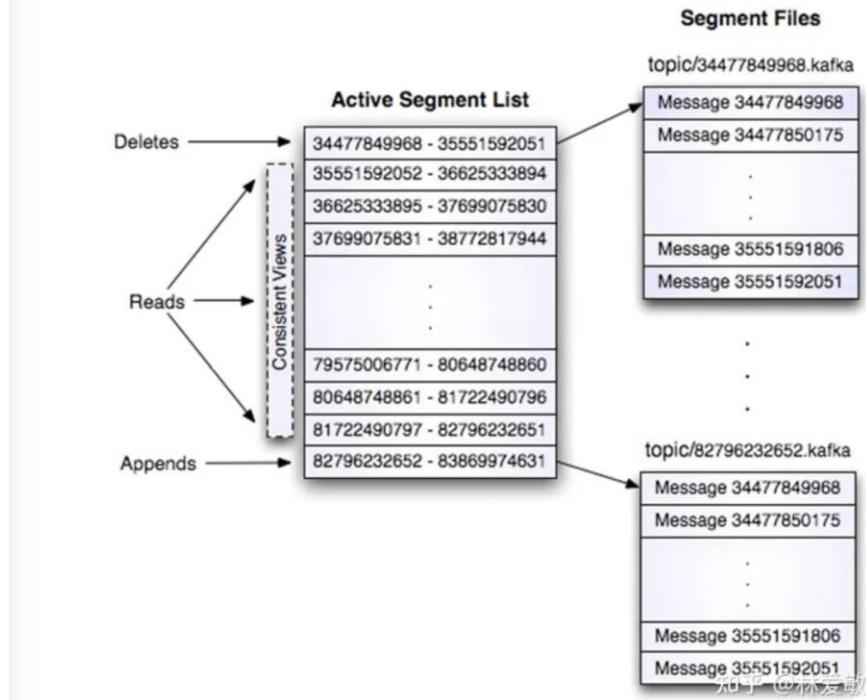


9. Broker (节点): 一个 broker 就是一个 kafka server 实例, 多个 broker 组成 kafka 集群, 主要用于接收生产者发送过来的消息, 写入磁盘, 同时可以接收消费者和其他 broker 的请求。

10. Rebalance (重新负载均衡): 当消费者组的消费者实例出现变化时, 例如新增消费者或者减少消费者, 都会触发 kafka 的 Rebalance 机制, 这个过程比较耗性能, 要尽量避免这个过程被触发。

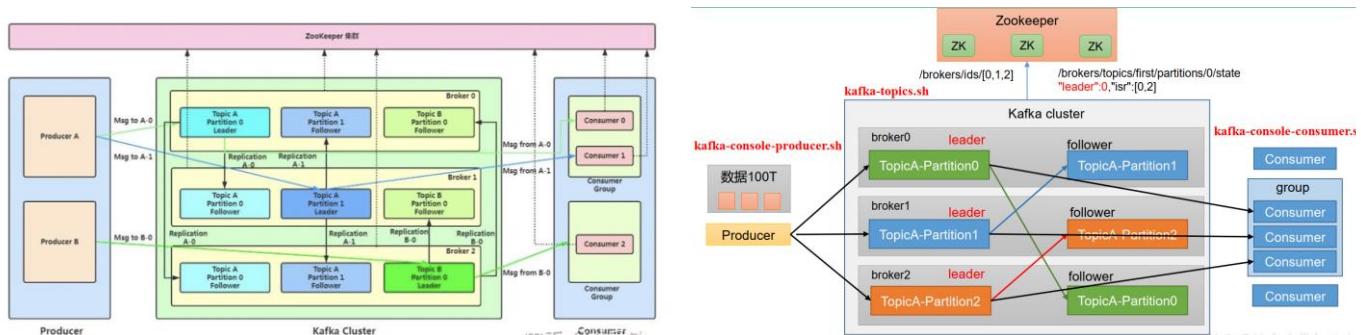
11. Record (记录): 实际写入 Kafka 中并可以被读取的消息记录。每个 record 包含了 key、value 和 timestamp。

Kafka Log Implementation



4.3.4 Kafka 基础架构

Kafka 的分布式架构可以实现高可用、高吞吐量和水平扩展等特性，它的基础架构由 broker、topic、partition、producer、consumer、zookeeper 等核心组件组成。这些组件都可以通过增加节点来进行扩展，以适应不同的负载需求，提供高效、可靠的消息传输和处理能力。



一个典型的 kafka 集群中包含若干 producer，若干 broker（Kafka 支持水平扩展，一般 broker 数量越多，集群吞吐率越高），若干 consumer group，以及一个 zookeeper 集群。kafka 通过 zookeeper 协调管理 kafka 集群，选举分区 leader，以及在 consumer group 发生变化时进行 rebalance。

Kafka 的 topic 被划分为一个或多个分区，多个分区可以分布在一个或多个 broker 节点上，同时为了故障容错，每个分区都会复制多个副本，分别位于不同的 broker 节点，这些分区副本中（不管是 leader 还是 follower 都称为分区副本），一个分区副本会作为 leader，其余的分区副本作为 follower。Leader 负责所有的客户端读写操作，follower 不对外提供服务，仅仅从 leader 上同步数据，当 leader 出现故障时，其中的一个 follower 会顶替成为 leader，继续对外提供服务。

对于传统的 MQ 而言，已经被消费的消息会从队列中删除，但在 Kafka 中被消费的消息也不会立马删除，在 `server.properties` 配置文件中定义了数据的保存时间，当文件到设定的保存时间时才会删除。因为 Kafka 读取消息的时间复杂度为 O(1)，与文件大小无关，所以删除过期文件与提高 Kafka 性能并没有关系，所以选择怎样的删除策略应该考虑磁盘以及具体的需求。

4.3.5 Kafka 工作原理与应用场景

4.3.5.1 Kafka 的工作原理

Kafka 的一个关键特性是其分布式架构，这使它能够处理大量数据并根据需要进行横向扩展。Kafka 通过将数据划分为多个主题来实现可扩展性，然后将这些主题分布在 Kafka 集群中的多个代理中。每个分区都可以被视为一个逻辑数据单元，可以在不同的代理之间独立复制，以实现容错和持久性。Kafka 的容错是通过跨多个代理复制数据来实现的。每个分区都通过可配置数量的代理进行复制，其中一个代理充当领导者，其他代理充当追随者。如果 leader broker 失败，那么其中一个 follower 将接任新的 leader，以确保消息处理可以继续进行而不会中断。

Kafka 的消息系统基于发布-订阅模型，生产者向主题发布消息，消费者订阅主题以接收消息。Kafka 的消息传递系统提供了多种保证，包括至少一次消息传递语义、在分区内对消息进行排序以及回放消息的能力。

Kafka 的分布式、容错和可扩展的消息传递系统为构建实时数据处理管道提供了坚实的基础，使其成为日志聚合、流处理和消息传递等用例的热门选择。具体来说，Kafka 的工作原理涉及以下几个方面：

1. 生产者和消费者

生产者是为 Kafka 主题创建和发布消息的过程。生产者可以用任何有 Kafka 客户端库的语言编写，如 Java、Python 或 Go。当生产者向主题发布消息时，它会将消息发送到集群中的一个代理，代理又将消息分发到适当的分区。

消费者是订阅和消费来自 Kafka 主题的消息的过程。消费者也可以用任何有 Kafka 客户端库的语言编写。当使用者订阅某个主题时，它会指定要从中使用消息的分区。使用者按照消息的写入顺序从分区中读取消息，并可以通过存储其读取的最后一条消息的偏移量来维持其在分区中的当前位置。

在 Kafka 中，数据由生产者生产，消费者消费。生产者将数据发布到 Kafka 集群，而消费者则从集群中读取数据。生产者负责创建一条消息，并将其发送到 Kafka 集群中的特定主题。消费者订阅一个或多个主题并从中读取消息。生产者和消费者通过代理与 Kafka 集群通信，代理是生产者和消费者之间的中介服务器。当生产者向代理发送消息时，代理会将消息附加到相应主题的日志末尾。消费者反过来向代理发送一个请求，请求来自特定主题和分区的消息。

生产者和消费者可以用不同的编程语言编写，并且有适用于大多数流行语言的客户端库。这些客户端库提供了一种简单可靠的方式与 Kafka 集群交互。生产者可以同步或异步发布消息。同步发布意味着生产者在继续之前等待来自代理的确认，而异步发布意味着制作者不等待确认并立即继续处理。

消费者可以通过多种方式读取消息，包括单消息处理、批处理和流处理。当消费者需要一次处理一条消息时，会使用单消息处理。当消费者需要同时处理一组消息时，使用批处理。当消费者需要实时处理无限制的消息流时，会使用流处理。

2. Kafka 代理和集群

代理是存储和管理分区的消息和元数据的服务器。Kafka 集群中的每个代理都可以充当一个或多个分区的领导者或追随者。leader 负责处理分区的所有读写请求，而 follower 则被动地从 leader 复制数据，以保持一致性和容错性。代理程序相互通信，以确保数据在集群中进行复制和同步。

Kafka 集群由一个或多个 Kafka 代理组成，这些代理是管理和存储消息的服务器。代理负责接收来自生产者的消息，在集群中以分布式方式存储消息，并向消费者提供消息。Kafka 集群中的每个代理都维护 Kafka 主题中一部分数据的副本。数据被划分为多个分区，每个分区在多个代理之间进行复制，以提供容错性和可扩展性。可以通过添加或删除代理来扩展或收缩 Kafka 集群。在集群中添加或删除代理时，Kafka 提供了自动分区重新平衡。当添加代理时，它会自动接管一些分区副本，当删除代理时，副本会自动重新分配给其余的代理。

Kafka 代理可以相互通信，以确保数据的一致性和容错性。代理使用分布式一致性算法来商定消息的顺序并维护集群的状态。这个共识算法是使用 Apache ZooKeeper 实现的，这是一个分布式协调服务。

在 Kafka 集群中，代理可以以各种配置进行排列，包括单节点、多代理和多数据中心。单节点配置用于测试和开发，而多代理配置提供可扩展性和容错性。当 Kafka 集群需要分布在多个地理位置以进行灾难恢复或减少延迟时，会使用多数据中心配置。

3. 主题和分区

主题是生产者发布消息的高级类别或渠道。发布到主题的消息可以由一个或多个使用者使用。Kafka 中的每个主题都被进一步划分为一个或多个分区。

分区是 Kafka 中的并行单元，它允许在多个代理之间拆分主题中的消息，以获得更好的吞吐量和可扩展性。每个分区都是一个有序且不可变的消息序列，由主题上下文中唯一的整数 ID 标识。分区在 Kafka 集群中跨多个代理进行复制，以提供容错性和高可用性。分区的复制因子决定了集群中存储的副本数。默认情况下，Kafka 在集群中的不同代理之间对每个分区进行三次复制。

在 Kafka 中，数据被组织成主题，这些主题表示特定的数据流。主题是一个类别或提要名称，生产者向其发布消息，消费者从中消费消息。一个主题可以有一个或多个分区，用于实现可伸缩性和并行性。

分区是 Kafka 中的一个并行单元，表示主题数据的一部分。每个分区都是一个有序且不可变的消息序列，由其主题中的唯一整数 id 标识。每个分区都跨多个代理进行复制，以确保数据的持久性和可用性。

当生产者向主题发布消息时，它会根据分区策略附加到主题的一个分区的末尾。分区策略可以由生产者指定，也可以由 Kafka 提供默认策略。该策略可以基于消息的密钥、循环或自定义逻辑。

当使用者订阅某个主题时，它可以指定要从哪个分区进行消费。消费者可以并行地从多个分区读取消息，这允许更高的吞吐量和并行处理。Kafka 使用者还维护一个称为偏移量的游标，该游标表示每个分区中最后一条消费消息的位置。偏移量存储在 Kafka 中，用于跟踪消费者的进度。

在 Kafka 中添加或删除分区是横向扩展系统的一种方式。分区允许 Kafka 在多个代理之间分发数据，每个分区都可以由一个单独的使用者处理。Kafka 还允许动态分区重新分配，这意味着在系统运行时可以在代理之间移动分区。

4. 发布和订阅消息

在 Kafka 中，生产者向主题发布消息，消费者订阅主题以接收这些消息。Kafka 提供了一个用于分发数据的发布-订阅模型，其中生产者向主题发布消息，消费者订阅这些主题以接收消息。

当生产者向主题发布消息时，消息会根据分区策略写入主题的一个分区。然后在多个代理之间复制消息，以确保持久性和可用性。

消费者可以订阅一个或多个主题以接收消息。当使用者订阅某个主题时，它会被分配到该主题中的一个或多个分区，并且它可以开始从这些分区接收消息。Kafka 消费者维护一个称为偏移量的游标，该游标表示每个分区中最后一条消费消息的位置。使用者可以在首次订阅主题时指定起始偏移量，也可以寻求特定的偏移量来重播消息。

Kafka 还提供了许多用于管理消息消费的功能，例如消费者组，它允许多个消费者共享消费来自一个主题的消息的负载。每个使用者组接收一个主题中消息的单独副本，这样消息就不会在使用者组之间重复。

Kafka 还提供了一个称为消息保留的功能，它允许消息在 Kafka 中存储一段可配置的时间。这对于需要处理历史数据或在发生故障时重播消息的系统来说非常有用。

5. 数据备份和容错

Kafka 被设计为具有高度容错性和可靠性，即使在存在硬件或网络故障的情况下也是如此。数据备份是这种容错的关键组成部分，它是通过使用 Kafka 集群来实现的。

Kafka 集群由一个或多个 Kafka 代理组成。每个代理都包含一个主题的每个分区的一个或多个副本。复制因子决定了集群中应该维护每个分区的复制副本数量。例如，如果复制因子设置为三，那么每个分区在群集中都将有三个副本。

Kafka 集群旨在自动检测代理故障，并启动一个进程，用新的代理替换故障的代理。当代理失败时，集群将自动将分区副本移动到其他代理，以确保数据保持可用。这个过程被称为 leader election，每个分区的一个副本被选为 leader，负责处理该分区的读写请求。

当生产者向 Kafka 写入数据时，它会向分区的前导副本写入数据。然后，引导副本将数据复制到群集中的其他副本。这样可以确保，如果引导副本出现故障，其他副本中的一个可以接任新的引导副本，并且数据仍然可用。

6. 提交日志和偏移

在 Kafka 中，提交日志和偏移是两个重要的概念，用于跟踪消息消耗的进度，并确保以可靠和容错的方式处理消息。

提交日志是一个记录所有已写入 Kafka 主题的消息的日志。它被实现为磁盘上的一组文件，其中每个文件代表日志的一段。当消息被生成到一个主题时，它们被附加到日志的末尾。然后，消费者可以通过从特定偏移量消费来从日志中读取消息。提交日志是一个不可变的数据结构，这意味着一旦将消息写入日志，就不能修改或删除它。

另一方面，偏移本质上是指向 Kafka 主题的提交日志中特定位置的指针。提交日志中的每条消息都被分配了一个唯一的偏移量，用于跟踪消息消耗的进度。当使用者组从主题中读取消息时，它会跟踪它所使用的最后一条消息的偏移量。这个偏移量存储在 Kafka 中，被称为消费者组的“当前偏移量”。当使用者组再次读取消息时，它会从当前偏移量开始读取，从而确保它只使用以前未处理过的新消息。

当使用者组处理完一批消息后，它通常会将当前偏移量提交回 Kafka。此提交操作以原子方式执行，确保只有在成功处理了批处理中的所有消息后，才会更新偏移量。这允许使用者组在发生故障或重新启动时从正确的偏移量恢复读取。

7. 消息保证

在 Kafka 中，消息保证指的是消息的产生、存储和消费的可靠性和一致性水平。Kafka 提供了不同级别的消息保证，以满足各种各样的用例。

Kafka 提供三种消息保证：

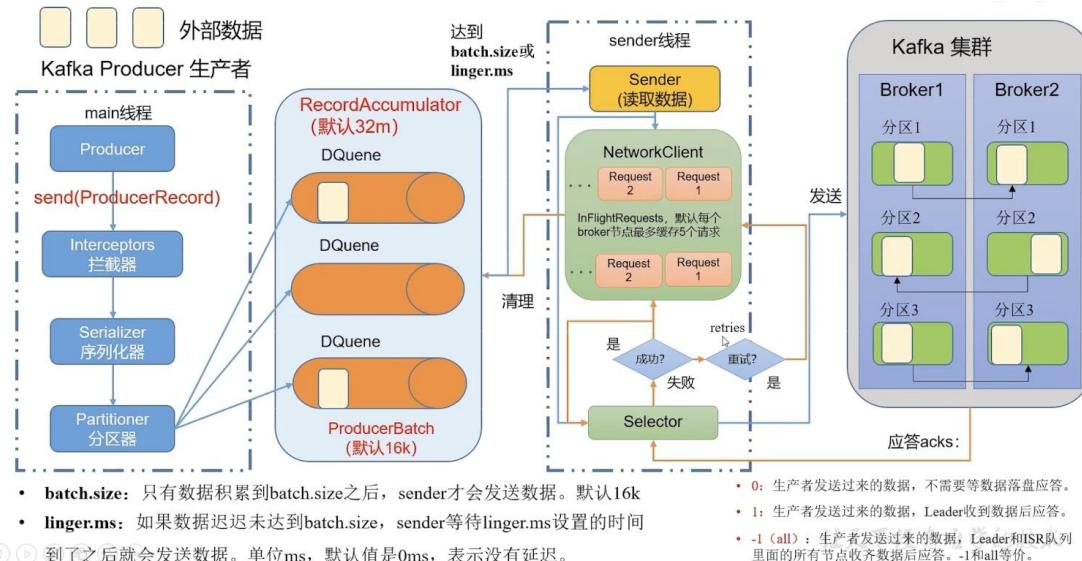
最多一次：这种保证意味着消息可能会丢失，但永远不会重新传递。在这种情况下，生产者在发布消息后不等待来自代理的确认，这使得消息的生成更快，但风险也更大。

至少一次：这种保证可以确保消息不会丢失，但可以重新传递。在这种情况下，生产者在发布消息后等待来自代理的确认，这确保了消息存储在 Kafka 中，但如果生产者没有收到确认，可能会导致消息重复。

正好一次：这一保证确保消息不会丢失，也不会重复。这种保证是最理想的，因为它可以确保每条消息只传递一次，不会有任何数据丢失或重复。然而，实现这一保证比以前的保证更为复杂，需要额外的机制，如交易生产者和幂等生产者。

Kafka 提供了不同的机制来实现这些保证。例如，至少一次保证可以通过在生产者配置中启用 `acks=all` 来实现，消费者可以将他们的偏移提交给 Kafka，以确保他们不会多次读取相同的消息。为了实现一次保证，Kafka 提供了事务生产者和幂等生产者，以确保消息的原子性和唯一性。

4.3.5.2 Kafka 的工作流程



一、生产者的数据发送到 Kafka 集群的流程

1. 生产者的预处理

生产者首先会创建一个 main 线程，在 main 线程中创建 producer 对象，紧接着调用 send 方法发送数据，在发送的数据中根据业务的需求增加拦截器（一般情况下不需要拦截器）。然后，通过自带的序列化器将数据进行序列化，再通过分区器将数据发往不同分区。

2. 数据发送到缓存队列

在记录累加器(RecordAccumulator)中每个分区会创建一个队列，数据发送到分区实际上是将数据发送到缓存队列中。整个过程是在内存里完成的，缓存队列的默认大小是 32mb，其中每一批次大小默认是 16kb。这个缓存队列实际上是一个双端队列，在 RecordAccumulator 中除了双端队列，还有一个内存池的概念。当发送数据时，数据进入队列，会从内存池中取出数据，这些数据发送到 Kafka 集群后，数据会被释放到内存池中。

之后 sender 线程会主动从缓存队列拉取数据，拉取数据需要满足 batch.size 条件或 linger.ms 条件。batch.size 条件即只有数据积累到 batch.size 之后，sender 才会发送数据，默认 16k。linger.ms 条件是指如果数据迟迟未到达 batch.size，sender 等待 linger.ms 设置的时间，到了这个时间之后就会发送数据，单位 ms，默认值是 0ms，表示没有延迟。

3. sender 线程拉取数据

接下来 sender 线程开始拉取数据，以每个节点为 key，后面跟随请求，每个分区的数据以节点的方式发送到队列中。如果发送过程中，broker 中的分区没有及时应答，最多可以发送 5 个请求。Selector 起到一个打通底层链路的作用。链路打通之后，就开始把数据发送到 Kafka 集群。Kafka 集群有一个副本同步的机制。

4. Kafka 集群的应答

Kafka 集群收到数据之后有一个应答的机制。应答 ack 为 0 代表生产者发送过来的数据，不需要等数据落盘应答；为 1 代表生产者发送过来的数据，Leader 收到数据后应答；为 -1(all) 代表生产者发送过来的数据，Leader 和 ISR 队列里面所有节点收齐数据后应答，-1 和 all 等价。

5. 清理已发送的数据

如果 Kafka 集群应答的信息是成功收到数据，那生产者会把请求清理掉，同时清理掉每一个分区缓存队列里对应的数据。如果反馈的信息是没有成功收到数据，那 sender 线程就会重试，直接根据请求再次发送数据，直到 Kafka 集群成功收到数据。

二、Kafka 集群中 Broker 的工作流程

1. Zookeeper 中存储的 Kafka 信息

- (1) /kafka/brokers/ids 记录有哪些服务器
- (2) /kafka/brokers/topics/first/partitions/0/state 记录谁是 leader，有哪些服务器可用
- (3) /kafka/controller 辅助选举 leader

此外 kafka 中还有个 consumer 目录，在 0.9 版本之前用于保存 offset 信息，在 0.9 版本之后 offset 存储在 kafka 主题中。

2. Kafka 副本基本信息

- (1) Kafka 副本作用：提高数据可靠性。
- (2) Kafka 默认副本 1 个，生产环境一般配置为 2 个，保证数据可靠性；太多副本会增加磁盘存储空间，增加网络上数据传输，降低效率。
- (3) Kafka 中副本分为：Leader 和 Follower。Kafka 生产者只会把数据发往 Leader，然后 Follower 找 Leader 进行同步数据。
- (4) Kafka 分区中的所有副本统称为 AR (Assigned Replicas)。

AR = ISR + OSR

ISR，表示和 Leader 保持同步的 Follower 集合。如果 Follower 长时间未向 Leader 发送通信请求或同步数据，则该 Follower 将被踢出 ISR。该时间阈值由 replica.lag.time.max.ms 参数设定，默认 30s。Leader 发生故障之后，就会从 ISR 中选举新的 Leader。

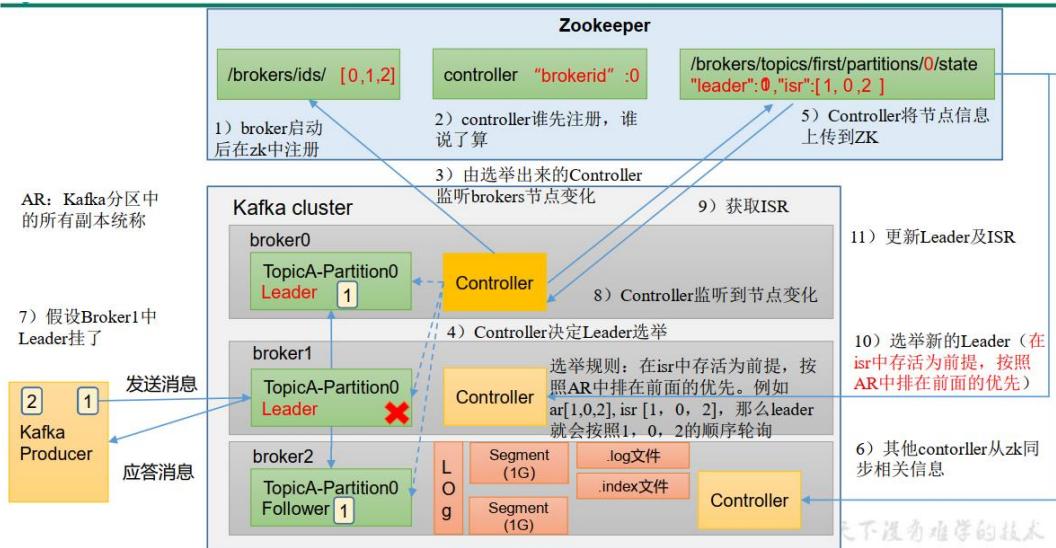
OSR，表示 Follower 与 Leader 副本同步时，延迟过多的副本。

3.Leader 的选举

Kafka 集群中有一个 broker 的 Controller 会被选举为 Controller Leader，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 Leader 选举等工作。

Controller 的信息同步工作是依赖于 Zookeeper 的。

4.Kafka Broker 总体工作流程



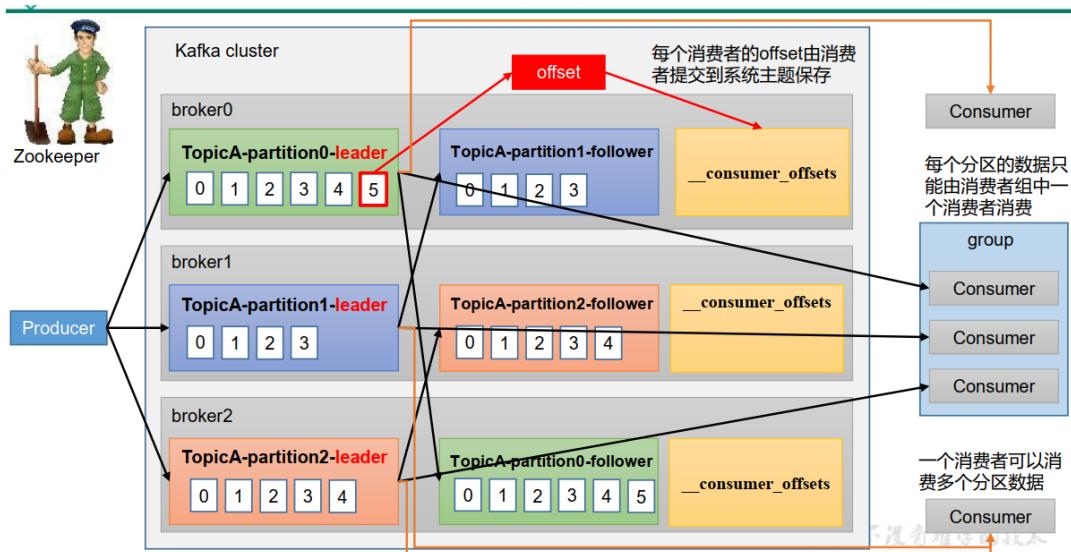
- 1) Broker 启动后再 zookeeper 中注册；2) 开始选择 controller 节点，谁先注册，哪个 controller 节点就负责 leader 选举；3) 由选举出来的 controller 监听 broker 节点的变化；4) controller 决定 leader 选举，选举规则是：在 isr 中存活为前提，按照 AR(Kafka 分区中的所有副本统称) 中排在前面的优先；5) 选举出 leader 后，controller 将节点信息传到 zookeeper 的 /kafka/brokers/topics/first/partitions/0/state 节点；6) 其他 controller 从 zookeeper 同步相关信息，同步的信息在底层是采用 log 进行存储（log 是虚拟的名称，实际底层是用 Segment 存储，每个 Segment 大小是 1 个 G）；7) 假设 Broker 中 leader 挂了，controller 能监控到节点的变化，从 zookeeper 中拉取对应的 leader 信息和 isr 信息，重新按照同样的规则选举新的 leader。

三、消费者的工作流程

1.消费方式

正常来说有两种消费方式：pull(拉)模式和 push(推)模式。Kafka 采取的是主动拉取的方式。Kafka 没有采取 push 模式，是因为由 Broker 决定消息发送速率，很难适应所有消费者的消费速率。但 pull 模式的不足之处是，如果 Kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。

2.消费者总体工作流程



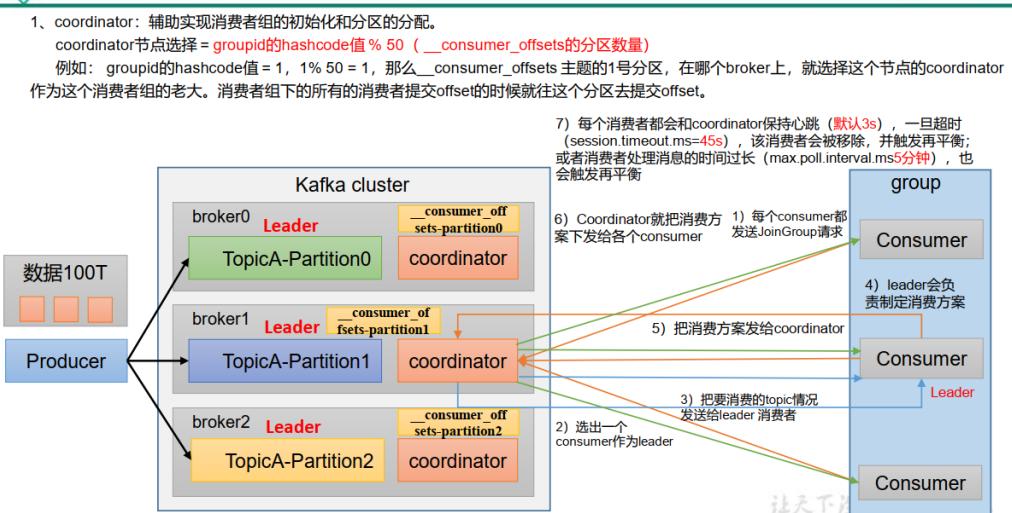
首先生产者主动向 Kafka 集群中的 Leader 节点发送数据，发送到 leader 后，follower 主动进行数据同步，保证数据可靠性。一个消费者可以消费多个分区的数据，在消费时，各个消费者之间是完全独立的，不会产生冲突。但对于消费者组，每个分区的数据只能由消费者组中的一个消费者来消费。

当消费者消费中断时，需要靠 offset(偏移量)在重启时找到中断时的位置。offset 不能保存在 consumer 里，当 consumer 挂掉，内存被回收，offset 就会丢失。在新版本的 Kafka 中，每个消费者的 offset 由消费者提交到系统主题保存(放在`_consumer_offsets`中)，是持久化到硬盘中保存的。在老版本的 Kafka 中，offset 存储在对应的 Zookeeper 中的。这样做的问题是，所有的消费者都把 offset 维护在 Zookeeper 中，那所有的消费者都要主动地跟 Zookeeper 进行大量的交互，就会导致网络数据传输非常频繁，给网络造成压力。把 offset 放在系统主题里更方便管理维护。

3. 消费者组

消费者组由多个 consumer 组成，形成一个消费者组的条件是所有消费者的 groupid 相同。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费。消费者组之间互不影响，所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。如果消费者组中的消费者超过主题分区数量，则有一部分消费者就会闲置，不会接收任何消息。

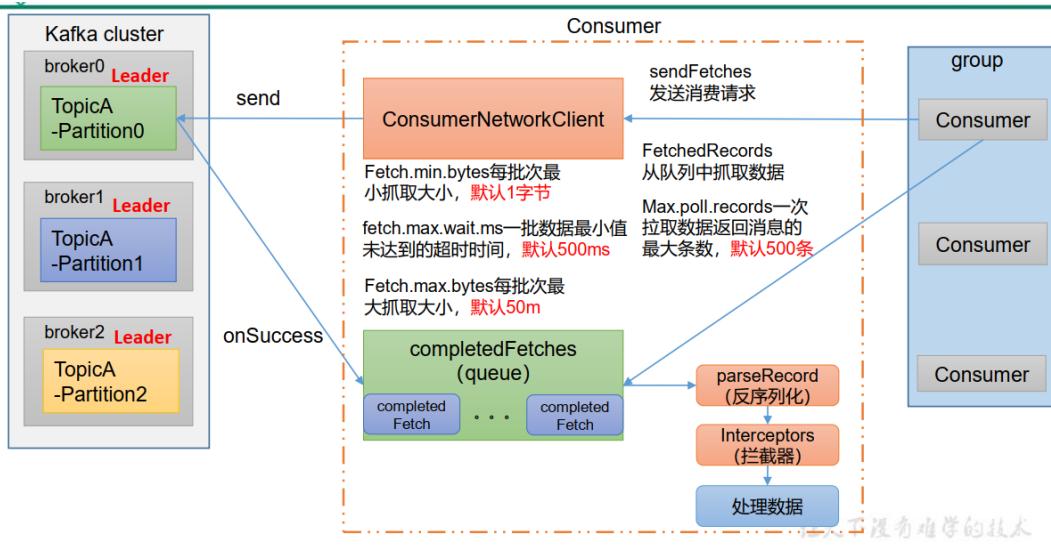
4. 消费者组初始化流程



coordinator 是辅助实现消费者组初始化和分区的分配的一个组件。每一个 broker 节点都有一个对应的 coordinator。消费者组选择哪一个 coordinator 来辅助它进行初始化取决于它的 groupid 的 hash 值。groupid 是用户在代码中手动设置的。coordinator 节点选择 = groupid 的 hash 值 % 50(`_consumer_offsets` 的分区数量)。

选出来对应的 coordinator 后，1) 首先每个 consumer 都发送 JoinGroup 请求；2) 随机选出一个 consumer 作为 leader；3) 把要消费的 topic 情况发送给 leader 消费者；4) leader 会负责制定消费方案，比如哪个消费者消费哪个节点；5) 把消费方案发送给 coordinator；6) coordinator 把消费方案下发给各个 consumer；7) 每个消费者都会和 coordinator 保持心跳（默认 3s），一旦超时（`session.timeout.ms=45s`），消费者会被移除，并触发再平衡；或者消费者处理消息的时间过长（`max.poll.interval.ms=5min`），也会触发再平衡。再平衡就是指把 consumer 未完成的任务再分配给其它 consumer，再平衡会影响 Kafka 的工作性能。

5. 消费者组的详细消费流程



1) 消费者组要想进行工作，首先要创建一个 `ConsumerNetworkClient`（消费者网络连接客户端），主要是用来跟 Kafka 集群进行交互；2) 消费者组调用 `sendFetches` 方法，发送消费请求，用来抓取数据的初始化（几个参数包括：`Fetch.min.bytes` 代表每批次最小抓取大小，默认 1 字节，`fetch.max.wait.ms` 代表一批数据最小值未达到的超时时间，默认 500ms，`Fetch.max.bytes` 代表每批次最大抓取大小，默认 50m）；3) 准备完毕后调用 `send` 方法，发送请求后通过回调方法 `onSuccess` 把对应的结果拉取过来；4) 拉取过来的数据放在 `completedFetches` 里的消息队列里；5) 消费者一批次拉取 500 条消息记录（`Max.poll.records` 代表一次拉取数据返回消息的最大条数，默认 500 条）；6) 对数据进行反序列化处理（`parseRecord`）；7) 数据经过拦截器（`Interceptors`）；8) 消费者根据需求对数据进行处理。

4.3.5.3 Kafka 的应用场景

Kafka 适用于任何需要高性能、可扩展、可靠和实时处理的场景，特别是大数据领域的实时处理和流式处理应用。

1. 消息系统

相较于传统消息系统，Kafka 具有高吞吐量和低延迟特性、拥有内置的分区 Partition、复制备份高容错能力。Kafka 可以作为一种高效的消息队列，将来自一个系统的消息传递到另一个系统。例如，一个 Web 应用程序可以将用户请求发送到 Kafka 集群中，然后由另一个系统（如分布式计算系统）进行处理和回复。

2. 网站行为追踪

Kafka 可以用于用户行为追踪，通过将用户行为数据发送给 Kafka 集群进行实时处理和分析。Kafka 经常被用来记录 web 用户或者 app 用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到 kafka 的 topic 中，然后消费者通过订阅这些 topic 来做实时的监控分析，亦可保存到数据库。基于此，可以实现用户行为的在线与离线分析，例如网站实时监控和异常行为拦截等。

3. 日志收集

Kafka 可以作为日志收集解决方案。日志收集通常是将不同服务器的日志文件收集到一个中心区域，Kafka 实现了对日志文件数据进行抽象，统一了处理接口。Kafka 低延迟，支持不同的日志数据源，分布式消费易于扩展，可同时将数据提供给 hdfs、storm、监控软件等等。

4. 大数据实时计算

kafka 被应用到大数据处理，如与 spark、storm 等整合。可以用于处理大规模数据集，如用户行为数据、网络流量数据等。它可以提供高吞吐量和低延迟的处理能力，并支持分布式处理和计算。

4.3.6 kafka 与其他消息中间件的对比

1. 性能对比

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个量级，同 RabbitMQ	万级，比 RocketMQ、Kafka 低一个量级，同 ActiveMQ	10 万级，支持高吞吐量	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用性	同 ActiveMQ	非常高，分布式架构	非常高，分布式架构，一个数据有多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	基本不丢失	经过参数优化配置，可以做到 0 丢失	基本不丢
topic 对吞吐量的影响	-	-	topic 可以达到几百/几千的级别，吞吐量会有小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个的时候，吞吐量会大幅度下降。在同等机器下，Kafka 尽量保证 topic 数量不会过多，如果要支撑大规模的 topic，需要增加更多的机器资源

2. 功能对比

功能	ActiveMQ	RabbitMQ	RocketMQ	Kafka
安全防护	支持	支持	不支持	不支持
主子账号支持	支持	不支持	不支持	不支持
横向扩展能力	支持	- 集群扩容依赖前端 - LVS 负载均衡调度	支持	支持
Low Latency	支持	不支持	不支持	不支持
消费模型	Push/Pull	Push/Pull	Push/Pull	Pull

定时消息	支持	支持	支持（只支持18个固定Level）	不支持
事务消息	不支持	不支持	不支持	不支持
顺序消息	不支持	不支持	支持	支持
全链路消息轨迹	不支持	不支持	不支持	不支持
消息堆积能力	影响性能	影响性能	百亿级别，影响性能	影响性能
消息堆积查询	不支持	不支持	支持	不支持
消息回溯	不支持	不支持	支持	不支持
消息重试	支持	支持	支持	不支持
死信队列	支持	支持	支持	不支持

4.3.7 kafka 调优

4.3.7.1 Kafka 硬件配置选择

场景说明

100 万日活，每人每天 100 条日志，每天总共的日志条数是 $100 \text{ 万} * 100 \text{ 条} = 1 \text{ 亿条}$ 。

$1 \text{ 亿}/24 \text{ 小时}/60 \text{ 分}/60 \text{ 秒} = 1150 \text{ 条}/\text{每秒钟}$ 。

每条日志大小：0.5k - 2k（取 1k）。

$1150 \text{ 条}/\text{每秒钟} * 1\text{k} \approx 1\text{m/s}$ 。

高峰期每秒钟： $1150 \text{ 条} * 20 \text{ 倍} = 23000 \text{ 条}$ 。

每秒多少数据量：20MB/s。

服务器台式选择

服务器台数 = $2 * (\text{生产者峰值生产速率} * \text{副本} / 100) + 1$

$$= 2 * (20\text{m/s} * 2 / 100) + 1$$

$$= 3 \text{ 台}$$

建议 3 台服务器。

磁盘选择

kafka 底层主要是顺序写，固态硬盘和机械硬盘的顺序写速度差不多。

建议选择普通的机械硬盘。

每天总数据量：1亿条 * 1k \approx 100g

100g * 副本 2 * 保存时间 3 天 / 0.7 \approx 1T

建议三台服务器硬盘总大小，大于等于 1T。

内存选择

Kafka 内存组成：堆内存 + 页缓存

1) Kafka 堆内存建议每个节点：10g ~ 15g

2) 页缓存：页缓存是 Linux 系统服务器的内存。我们只需要保证 1 个 segment (1g) 中 25% 的数据在内存中就好。

每个节点页缓存大小 = (分区数 * 1g * 25%) / 节点数。例如 10 个分区，页缓存大小

$$= (10 * 1g * 25\%) / 3 \approx 1g$$

建议服务器内存大于等于 11G。

CPU 选择

num.io.threads = 8 负责写磁盘的线程数，整个参数值要占总核数的 50%。

num.replica.fetchers = 1 副本拉取线程数，这个参数占总核数的 50% 的 1/3。

num.network.threads = 3 数据传输线程数，这个参数占总核数的 50% 的 2/3。

建议 32 个 cpu core。

网络选择

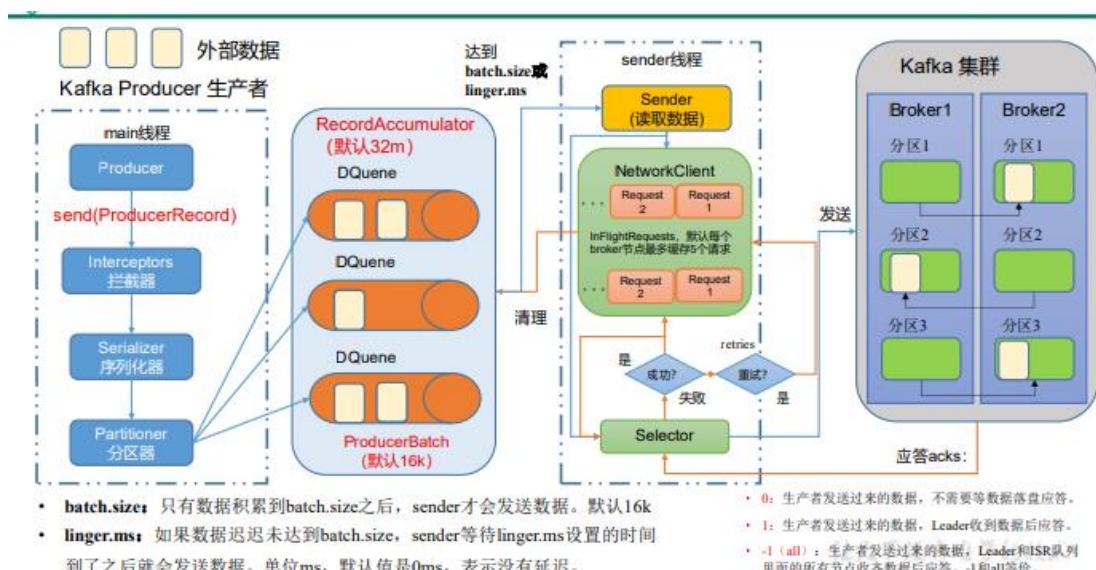
网络带宽 = 峰值吞吐量 \approx 20MB/s 选择千兆网卡即可。

100Mbps 单位是 bit；10M/s 单位是 byte；1byte = 8bit， $100Mbps / 8 = 12.5M/s$ 。

一般百兆的网卡 (100Mbps)、千兆的网卡 (1000Mbps)、万兆的网卡 (10000Mbps)。

4.3.7.2 Kafka 生产者

Kafka 生产者核心参数配置



参数名称	描述
------	----

bootstrap.servers	生产者连接集群所需的 broker 地址清单。例如 hadoop102:9092,hadoop103:9092,hadoop104:9092，可以设置 1 个或者多个，中间用逗号隔开。注意这里并非需要所有的 broker 地址，因为生产者从给定的 broker 里查找到其他 broker 信息。
key.serializer 和 value.serializer	指定发送消息的 key 和 value 的序列化类型。一定要写全类名。
buffer.memory	RecordAccumulator 缓冲区总大小，默认 32m。
batch.size	缓冲区一批数据最大值，默认 16k。适当增加该值，可以提高吞吐量，但是如果该值设置太大，会导致数据传输延迟增加。
linger.ms	如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。单位 ms，默认值是 0ms，表示没有延迟。生产环境建议该值大小为 5-100ms 之间。
acks	0：生产者发送过来的数据，不需要等数据落盘应答。 1：生产者发送过来的数据，Leader 收到数据后应答。 -1 (all)：生产者发送过来的数据，Leader+ 和 isr 队列里面的所有节点收齐数据后应答。默认值是 -1，-1 和 all 是等价的。
max.in.flight.requests.per.connection	允许最多没有返回 ack 的次数，默认为 5，开启幂等性要保证该值是 1-5 的数字。
retries	当消息发送出现错误的时候，系统会重发消息。retries 表示重试次数。默认是 int 最大值，2147483647。如果设置了重试，还想保证消息的有序性，需要设置 MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION=1 否则在重试此失败消息的时候，其他的消息可能发送成功了。
retry.backoff.ms	两次重试之间的时间间隔，默认是 100ms。
enable.idempotence	是否开启幂等性，默认 true，开启幂等性。
compression.type	生产者发送的所有数据的压缩方式。默认是 none，也就是不压缩。

生产者如何提高吞吐量

参数名称	描述
buffer.memory	RecordAccumulator 缓冲区总大小，默认 32m。
batch.size	缓冲区一批数据最大值，默认 16k。适当增加该值，可以提高吞吐量，但是如果该值设置太大，会导致数据传输延迟增加。
linger.ms	如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。单位 ms，默认值是 0ms，表示没有延迟。生产环境建议该值大小为 5-100ms 之间。

compression.type	生产者发送的所有数据的压缩方式。默认是 none，也就是不压缩。 支持压缩类型：none、gzip、snappy、lz4 和 zstd。
------------------	---

数据可靠性

参数名称	描述
acks	0: 生产者发送过来的数据，不需要等数据落盘应答。 1: 生产者发送过来的数据，Leader 收到数据后应答。 -1 (all): 生产者发送过来的数据，Leader+和 ISR 队列里面的所有节点收齐数据后应答。默认值是-1，-1 和 all 是等价的。

- 至少一次 (At Least Once) = ACK 级别设置为-1 + 分区副本大于等于 2 + ISR 里应答的最小副本数量大于等于 2

数据去重

1) 配置参数

参数名称	描述
enable.idempotence	是否开启幂等性，默认 true，表示开启幂等性。

2) Kafka 的事务一共有如下 5 个 API

```
// 1 初始化事务
void initTransactions();

// 2 开启事务
void beginTransaction() throws ProducerFencedException;
// 3 在事务内提交已经消费的偏移量（主要用于消费者）
void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
String consumerGroupId) throws
ProducerFencedException;

// 4 提交事务
void commitTransaction() throws ProducerFencedException;
// 5 放弃事务（类似于回滚事务的操作）
void abortTransaction() throws ProducerFencedException;
```

数据有序

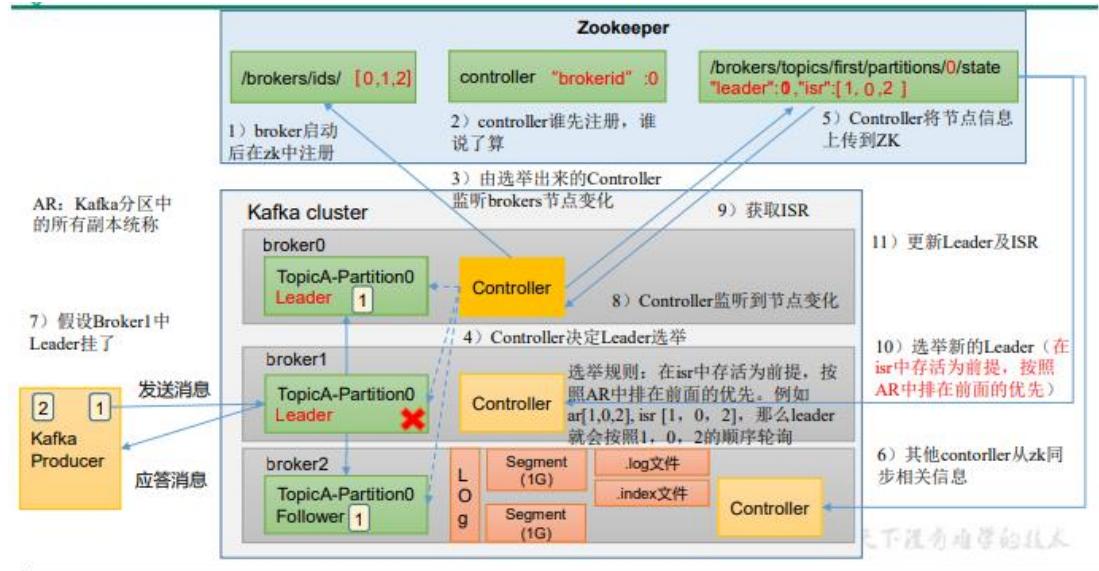
单分区，有序（有条件的，不能乱序）；多分区，分区与分区间无序。

数据乱序

参数名称	描述
enable.idempotence	是否开启幂等性， 默认 true， 表示开启幂等性。
max.in.flight.requests.per.connection	允许最多没有返回 ack 的次数， 默认为 5， 开启幂等性要保证该值是 1-5 的数字。

4.3.7.3 Kafka Broker

Broker 核心参数配置



参数名称	描述
replica.lag.time.max.ms	ISR 中, 如果 Follower 长时间未向 Leader 发送通信请求或同步数据, 则该 Follower 将被踢出 ISR。该时间阈值, 默认 30s。
auto.leader.rebalance.enable	默认是 true。自动 LeaderPartition 平衡。建议关闭。
leader.imbalance.per.broker.percentage	默认是 10%。每个 broker 允许的不平衡的 leader 的比率。如果每个 broker 超过了这个值, 控制器会触发 leader 的平衡。
leader.imbalance.check.interval.seconds	默认值 300 秒。检查 leader 负载是否平衡的间隔时间。
log.segment.bytes	Kafka 中 log 日志是分成一块块存储的, 此配置是指 log 日志划分成块的大小, 默认值 1G。
log.index.interval.bytes	默认 4kb, kafka 里面每当写入了 4kb 大小的日志 (.log), 然后就往 index 文件里面记录一个索引。
log.retention.hours	Kafka 中数据保存的时间, 默认 7 天。
log.retention.minutes	Kafka 中数据保存的时间, 分钟级别, 默认关闭。

log.retention.ms	Kafka 中数据保存的时间，毫秒级别，默认关闭。
log.retention.check.interval.ms	检查数据是否保存超时的间隔，默认是 5 分钟。
log.retention.bytes	默认等于-1，表示无穷大。超过设置的所有日志总大小，删除最早的 segment。
log.cleanup.policy	默认是 delete，表示所有数据启用删除策略；如果设置值为 compact，表示所有数据启用压缩策略。
num.io.threads	默认是 8。负责写磁盘的线程数。整个参数值要占总核数的 50%。
num.replica.fetchers	默认是 1。副本拉取线程数，这个参数占总核数的 50% 的 1/3
num.network.threads	默认是 3。数据传输线程数，这个参数占总核数的 50% 的 2/3。
log.flush.interval.messages	强制页缓存刷写到磁盘的条数，默认是 long 的最大值，9223372036854775807。一般不建议修改，交给系统自己管理。
log.flush.interval.ms	每隔多久，刷数据到磁盘，默认是 null。一般不建议修改，交给系统自己管理。

服役新节点/退役旧节点

(1) 创建一个要均衡的主题。

```
JSON
[myUbuntu@hadoop102 kafka]$ vim topics-to-move.json
```

```
{
  "topics": [
    {"topic": "first"}
  ],
  "version": 1
}
```

(2) 生成一个负载均衡的计划。

```
Shell
[myUbuntu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server
hadoop102:9092 --topics-to-move-json-file topics-to-move.json --broker-list "0,1,2,3" --
generate
```

(3) 创建副本存储计划（所有副本存储在 broker0、broker1、broker2、broker3 中）。

```
JSON
[myUbuntu@hadoop102 kafka]$ vim increase-replication-factor.json
```

(4) 执行副本存储计划。

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh -- bootstrap-server  
hadoop102:9092 --reassignment-json-file increasereplication-factor.json --execute
```

(5) 验证副本存储计划。

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh -- bootstrap-server  
hadoop102:9092 --reassignment-json-file increasereplication-factor.json --verify
```

增加分区

1) 修改分区数（注意：分区数只能增加，不能减少）

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 -  
-alter --topic first --partitions 3
```

增加副本因子

1) 创建 topic

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 -  
-create --partitions 3 --replication-factor 1 -- topic four
```

2) 手动增加副本存储

(1) 创建副本存储计划（所有副本都指定存储在 broker0、broker1、broker2 中）。

JSON

```
[myUbuntu@hadoop102 kafka]$ vim increase-replication-factor.json
```

输入以下内容：

JSON

```
{"version":1,"partitions":[{"topic":"four","partition":0,"replica  
s":[0,1,2]},{"topic":"four","partition":1,"replicas":[0,1,2]},{ "t  
opic":"four","partition":2,"replicas":[0,1,2]}]}
```

(2) 执行副本存储计划。

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh -- bootstrap-server  
hadoop102:9092 --reassignment-json-file increasereplication-factor.json --execute
```

手动调整分区副本存储

(1) 创建副本存储计划（所有副本都指定存储在 broker0、broker1 中）。

JSON

```
[myUbuntu@hadoop102 kafka]$ vim increase-replication-factor.json
```

输入如下内容：

JSON

```
{ "version":1,  
  "partitions":[{"topic":"three","partition":0,"replicas":[0,1]},  
    {"topic":"three","partition":1,"replicas":[0,1]},  
    {"topic":"three","partition":2,"replicas":[1,0]},  
    {"topic":"three","partition":3,"replicas":[1,0]}]  
}
```

(2) 执行副本存储计划。

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh -- bootstrap-server  
hadoop102:9092 --reassignment-json-file increasereplication-factor.json --execute
```

(3) 验证副本存储计划。

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh -- bootstrap-server  
hadoop102:9092 --reassignment-json-file increasereplication-factor.json --verify
```

4.3.7.4 Leader Partition 负载平衡

参数名称	描述
auto.leader.rebalance.enable	默认是 true。自动 Leader Partition 平衡。生产环境中，leader 重选举的代价比较大，可能会带来性能影响，建议设置为 false 关闭。
leader.imbalance.per.broker.percentage	默认是 10%。每个 broker 允许的不平衡的 leader 的比率。如果每个 broker 超过了这个值，控制器会触发 leader 的平衡。
leader.imbalance.check.interval.seconds	默认值 300 秒。检查 leader 负载是否平衡的间隔时间。

自动创建主题

如果 broker 端配置参数 auto.create.topics.enable 设置为 true（默认值是 true），那么当生产者向一个未创建的主题发送消息时，会自动创建一个分区数为 num.partitions（默认值为 1）、副本因子为 default.replication.factor（默认值为 1）的主题。除此之外，当一个消费者开始从未知主题中读取消息时，或者当任意一个客户端向未知主题发送元数据请求时，都会自动创建一个相应主题。这种创建主题的方式是非预期的，增加了主题管理和维护的难度。生产环境建议将该参数设置为 false。

1) 向一个没有提前创建 five 主题发送数据

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-console-producer.sh -- bootstrap-server  
hadoop102:9092 --topic five >hello world
```

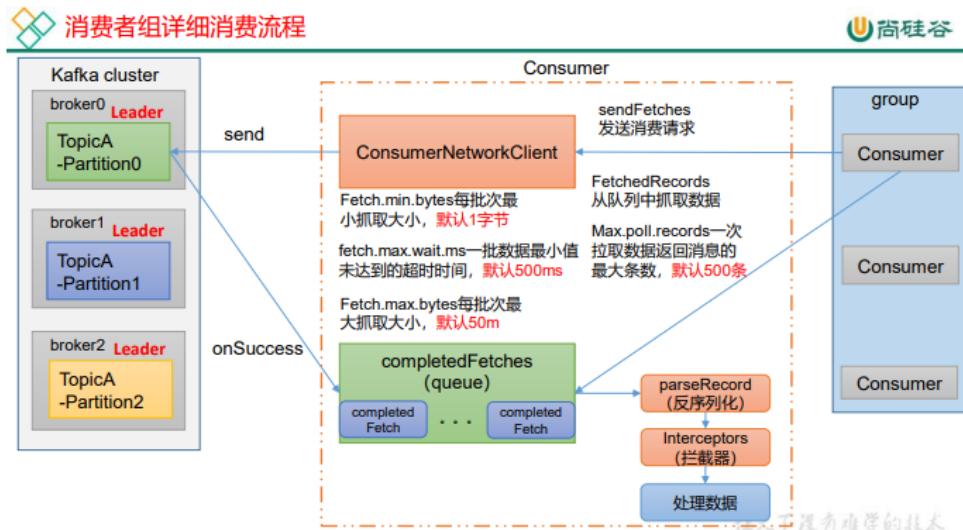
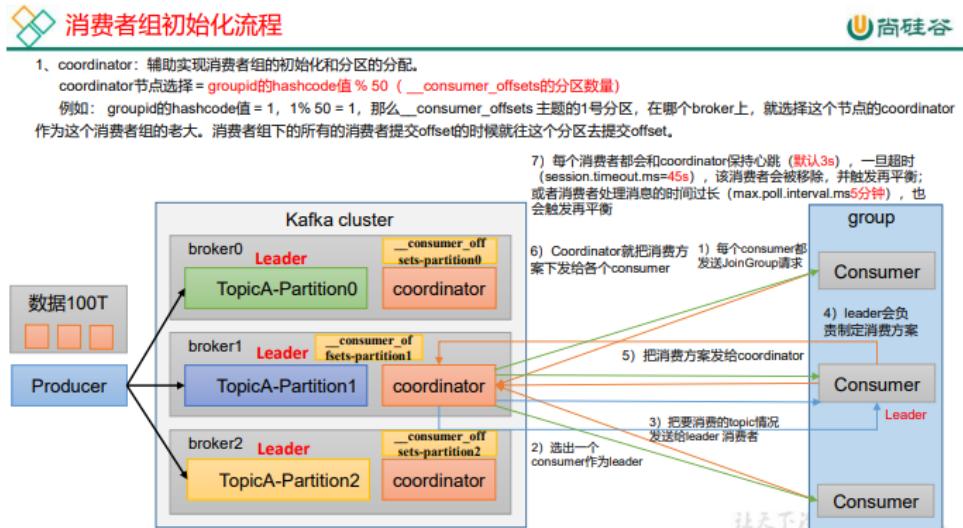
2) 查看 five 主题的详情

Shell

```
[myUbuntu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 -  
-describe --topic five
```

4.3.7.5 Kafka 消费者

Kafka 消费者核心参数配置



参数名称	描述
bootstrap.servers	向 Kafka 集群建立初始连接用到的 host/port 列表。
key.deserializer 和 value.deserializer	指定接收消息的 key 和 value 的反序列化类型。一定要写全类名。
group.id	标记消费者所属的消费者组。
enable.auto.commit	默认值为 true，消费者会自动周期性地向服务器提交偏移量。
auto.commit.interval.ms	如果设置了 enable.auto.commit 的值为 true，则该值定义了消费者偏移量向 Kafka 提交的频率，默认 5s。

auto.offset.reset	当 Kafka 中没有初始偏移量或当前偏移量在服务器中不存在（如数据被删除了该如何处理？）earliest：自动重置偏移量到最早的偏移量。latest：默认，自动重置偏移量为最新的偏移量。none：如果消费组原来的（previous）偏移量不存在，则向消费者抛异常。anything：向消费者抛异常。
offsets.topic.num.partitions	_consumer_offsets 的分区数，默认是 50 个分区。不建议修改。
heartbeat.interval.ms	Kafka 消费者和 coordinator 之间的心跳时间， 默认 3s。 该条目的值必须小于 session.timeout.ms，也不应该高于 session.timeout.ms 的 1/3。不建议修改。
session.timeout.ms	Kafka 消费者和 coordinator 之间连接超时时间， 默认 45s。超过该值，该消费者被移除，消费者组执行再平衡。
max.poll.interval.ms	消费者处理消息的最大时长， 默认是 5 分钟。超过该值，该消费者被移除，消费者组执行再平衡。
fetch.min.bytes	默认 1 个字节。消费者获取服务器端一批消息最小的字节数。
fetch.max.wait.ms	默认 500ms。如果没有从服务器端获取到一批数据的最小字节数。该时间到，仍然会返回数据。
fetch.max.bytes	默认 Default: 52428800 (50 m)。消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值 (50m) 仍然可以拉取回来这批数据，因此，这不是一个绝对最大值。一批次的大小受 message.max.bytes (broker config) or max.message.bytes (topic config) 影响。
max.poll.records	一次 poll 拉取数据返回消息的最大条数， 默认是 500 条。

消费者再平衡

参数名称	描述
heartbeat.interval.ms	Kafka 消费者和 coordinator 之间的心跳时间，

	<p>默认 3s。</p> <p>该条目的值必须小于 session.timeout.ms，也不应该高于 session.timeout.ms 的 1/3。</p>
session.timeout.ms	<p>Kafka 消费者和 coordinator 之间连接超时时间，默认 45s。</p> <p>超过该值，该消费者被移除，消费者组执行再平衡。</p>
max.poll.interval.ms	<p>消费者处理消息的最大时长，默认是 5 分钟。</p> <p>超过该值，该消费者被移除，消费者组执行再平衡。</p>
partition.assignment.strategy	<p>消费者分区分配策略， 默认策略是 Range + CooperativeSticky。Kafka 可以同时使用多个分区分配策略。可以选择的策略包括：Range、RoundRobin、Sticky、CooperativeSticky</p>

指定 Offset 消费

```
Shell
kafkaConsumer.seek(topic, 1000);
```

指定时间消费

```
Shell
HashMap timestampToSearch = new HashMap<>();
timestampToSearch.put(topicPartition, System.currentTimeMillis() - 1 * 24 * 3600 * 1000);
kafkaConsumer.offsetsForTimes(timestampToSearch);
```

消费者如何提高吞吐量

参数名称	描述
fetch.max.bytes	默认 Default: 52428800 (50 m)。消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值 (50m) 仍然可以拉取回来这批数据，因此，这不是一个绝对最大值。一批次的大小受 message.max.bytes (broker config) or max.message.bytes (topic config) 影响。
max.poll.records	一次 poll 拉取数据返回消息的最大条数，默认是 500 条

4.3.7.6 Kafka 总体

如何提升吞吐量

1) 提升生产吞吐量

- (1) `buffer.memory`: 发送消息的缓冲区大小, 默认值是 32m, 可以增加到 64m。
- (2) `batch.size`: 默认是 16k。如果 batch 设置太小, 会导致频繁网络请求, 吞吐量下降; 如果 batch 太大, 会导致一条消息需要等待很久才能被发送出去, 增加网络延时。
- (3) `linger.ms`, 这个值默认是 0, 意思就是消息必须立即被发送。一般设置一个 5-100 毫秒。如果 `linger.ms` 设置的太小, 会导致频繁网络请求, 吞吐量下降; 如果 `linger.ms` 太长, 会导致一条消息需要等待很久才能被发送出去, 增加网络延时。
- (4) `compression.type`: 默认是 `none`, 不压缩, 但是也可以使用 `lz4` 压缩, 效率还是不错的, 压缩之后可以减小数据量, 提升吞吐量, 但是会加大 producer 端的 CPU 开销。

2) 增加分区

3) 消费者提高吞吐量

- (1) 调整 `fetch.max.bytes` 大小, 默认是 50m。
- (2) 调整 `max.poll.records` 大小, 默认是 500 条。

4) 增加下游消费者处理能力

数据精准一次

1) 生产者角度

- `acks` 设置为 -1 (`acks=-1`)。
- 幂等性 (`enable.idempotence = true`) + 事务。

2) broker 服务端角度

- 分区副本大于等于 2 (`--replication-factor 2`)。
- ISR 里应答的最小副本数量大于等于 2 (`min.insync.replicas = 2`)。

3) 消费者

- 事务 + 手动提交 offset (`enable.auto.commit = false`)。
- 消费者输出的目的地必须支持事务 (MySQL、Kafka)。

合理设置分区数

- (1) 创建一个只有 1 个分区的 topic。
- (2) 测试这个 topic 的 producer 吞吐量和 consumer 吞吐量。
- (3) 假设他们的值分别是 T_p 和 T_c , 单位可以是 MB/s。
- (4) 然后假设总的目标吞吐量是 T_t , 那么分区数 = $T_t / \min(T_p, T_c)$ 。

例如: producer 吞吐量 = 20m/s; consumer 吞吐量 = 50m/s, 期望吞吐量 100m/s;

分区数 = $100 / 20 = 5$ 分区

分区数一般设置为: 3-10 个

分区数不是越多越好, 也不是越少越好, 需要搭建完集群, 进行压测, 再灵活调整分区个数。

单条日志大于 1m

参数名称	描述
message.max.bytes	默认 1m， broker 端接收每个批次消息最大值。
max.request.size	默认 1m， 生产者发往 broker 每个请求消息最大值。针对 topic 级别设置消息体的大小。
replica.fetch.max.bytes	默认 1m， 副本同步数据，每个批次消息最大值。
fetch.max.bytes	默认 Default: 52428800 (50 m)。消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值 (50m) 仍然可以拉取回来这批数据，因此，这不是一个绝对最大值。一批次的大小受 message.max.bytes (broker config) or max.message.bytes (topic config) 影响。

服务器挂了

在生产环境中，如果某个 Kafka 节点挂掉。

正常处理办法：

- (1) 先尝试重新启动一下，如果能启动正常，那直接解决。
- (2) 如果重启不行，考虑增加内存、增加 CPU、网络带宽。
- (3) 如果将 kafka 整个节点误删除，如果副本数大于等于 2，可以按照服役新节点的方式重新服役一个新节点，并执行负载均衡。

4.4 Flume

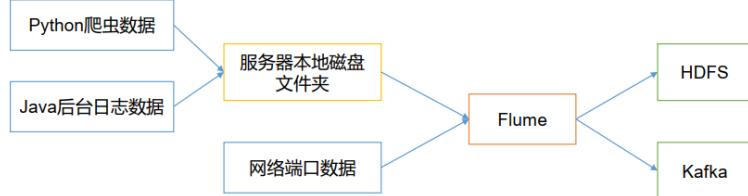
Flume 是由 Cloudera 软件公司产出的可分布式日志收集系统，后于 2009 年被捐赠了 Apache 软件基金会，为 hadoop 相关组件之一。近几年，随着 Flume 的不断被完善，用户在开发的过程中使用的便利性得到很大的改善，目前已演化成支持任何流式数据收集的通用系统，成为 Apache top 项目之一。

Flume 目前存在两个版本：Flume OG (Original generation) 和 Flume NG (Next/New generation)。

其中 Flume OG 对应的是 Apache Flume 0.9.x 之前的版本，早期随着 Flume 功能的扩展，Flume OG 代码工程臃肿、核心组件设计不合理、核心配置不标准等缺点暴露出来，尤其是在 Flume OG 的最后一个发行版本 0.9.4. 中，日志传输不稳定的现象尤为严重，为了解决这些问题，2011 年 10 月 22 号，Cloudera 完成了 Flume-728，对 Flume 进行了里程碑式的改动，重构后的版本统称为 Flume NG (next generation)。同时此次改动后，Flume 也纳入了 apache 旗下。

Flume NG 在 OG 的架构基础上做了调整，去掉了中心化组件 master 以及服务协调组件 Zookeeper，使得架构更加简单和容易部署。Flume NG 和 OG 是完全不兼容的，但沿袭了 OG 中的很多概念，包括 Source, Sink 等。

4.4.1 Flume 概述



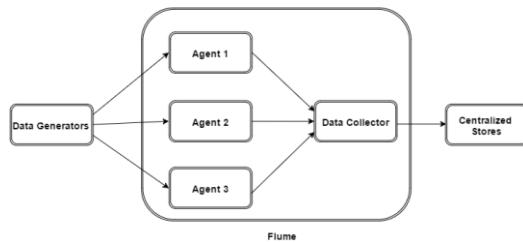
- Flume 是 Cloudera 提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统。Flume 基于流式架构，灵活简单。
 - 不需要配置集群，不需要启动后台进程，对每个采集任务开启一个进程
 - 日志：识别文本文件，不识别图片音频视频等
 - 动态采集，写一条采集一条
- Flume 最主要的作用就是，实时读取服务器本地磁盘的数据，将数据写入到 HDFS/Kafka 之类的分布式文件系统。
- 一般的采集需求，通过对 Flume 的简单配置即可实现；针对特殊场景，Flume 也具备良好的自定义扩展能力。因此，Flume 可以适用于大部分的日常数据采集场景。

4.4.2 Flume 基本架构与工作原理

采集，存储，计算和工具。Flume 属于采集工具。

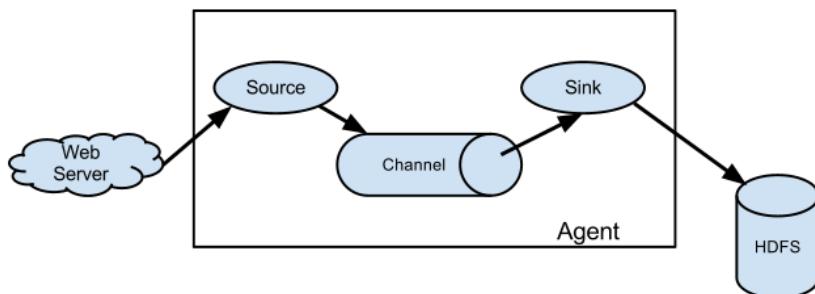
4.4.2.1 Flume 外部结构

数据发生器（如：facebook、twitter）产生的数据被单个的、运行在数据发生器所在服务器上的 agent 所收集，之后数据收容器从各个 agent 上汇集数据，并将采集到的数据存入到 HDFS 或者 HBase 中。



4.4.2.2 Flume 基本架构

Flume 组成架构如下图所示：



Flume 基本组成架构

- **Agent**: Flume 内部有一个或者多个 Agent，每个 Agent 是一个独立的 JVM 进程，它以事件的形式将数据从源头送至目的，是 Flume 数据传输的基本单元。Agent 主要有 3 个部分组成，Source、Channel、Sink。
- **Source**: Source 是负责接收数据到 Flume Agent 的组件。Source 组件可以处理各种类型、各种格式的日

志数据，包括 avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy。

- **Channel:** Channel 是位于 Source 和 Sink 之间的缓冲区。因此，Channel 允许 Source 和 Sink 运作在不同的速率上。Channel 是线程安全的，可以同时处理几个 Source 的写入操作和几个 Sink 的读取操作。Flume 自带两种 Channel：Memory Channel 和 File Channel（kafka 也可用作缓冲）。
 - Memory Channel 是内存中的队列。Memory Channel 在不需要关心数据丢失的情景下适用。如果需要关心数据丢失，那么 Memory Channel 就不应该使用，因为程序死亡、机器宕机或者重启都会导致数据丢失。
 - File Channel 将所有事件写到磁盘。因此在程序关闭或机器宕机的情况下不会丢失数据。
- **Sink:** Sink 不断地轮询 Channel 中的事件且批量地移除它们，并将这些事件批量写入到存储或索引系统、或者被发送到另一个 Flume Agent。Sink 是完全事务性的。在从 Channel 批量删除数据之前，每个 Sink 用 Channel 启动一个事务。批量事件一旦成功写出到存储系统或下一个 Flume Agent，Sink 就利用 Channel 提交事务。事务一旦被提交，该 Channel 从自己的内部缓冲区删除事件。Sink 组件目的地包括 hdfs、logger、avro、thrift、ipc、file、null、HBase、solr、自定义。
- **Event:** 传输单元，Flume 数据传输的基本单元，以 Event 的形式将数据从源头送至目的地。Event 由 Header 和 Body 两部分组成，Header 用来存放该 event 的一些属性，为 K-V 结构，Body 用来存放该条数据，形式为字节数组。



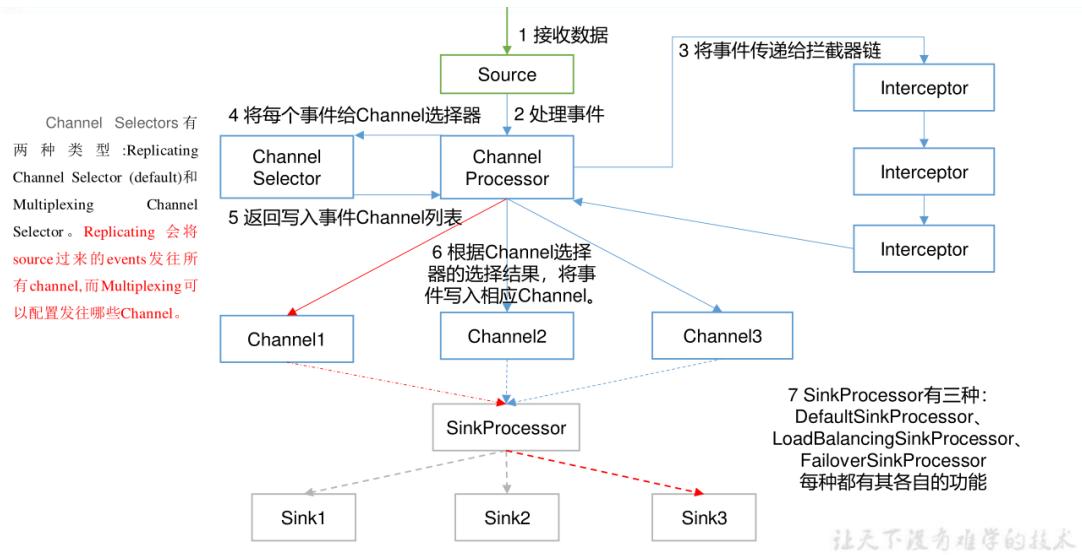
4.4.2.3 Flume 事务



Flume 细节架构

- **Put 事务流程:**
 - `doPut`: 将批数据先写入临时缓冲区 `putList`。
 - `doCommit`: 检查 channel 内存队列是否足够合并。
 - `doRollback`: channel 内存队列空间不足，回滚数据。
- **Take 事务:**
 - `doTake`: 先将数据取到临时缓冲区 `takeList`。
 - `doCommit`: 如果数据全部发送成功，则清除临时缓冲区 `takeList`。
 - `doRollback`: 数据发送过程中如果出现异常，`rollback` 将临时缓冲区 `takeList` 中的数据归还给 channel 内存队列。

4.4.2.4 Flume Agent 内部原理



重要组件:

- Channel 处理器 Channel Processor
 - 拦截器链: 多个 Interceptor, 对数据做调整过滤添加等
 - Channel 选择器 Channel Selector: 两种选择策略
 - . Replicating Channel Selector (默认): 副本, 一份数据发给多个 Channel, 内容完全相同
 - . Multiplexing Channel Selector: 多路复用, 人为控制一条数据发给哪个/哪些 Channel。结合拦截器使用, 根据数据头信息的内容决定将数据发往哪里。
- Sink 处理器 Sink Processor (Sink 组)
 - Channel 和 Sink 可一对多, 共三种类型
 - . Default Sink Processor: 对应单个的 Sink
 - . Load Balancing Sink Processor: 负载均衡, 多个 Sink 以轮询的方式从 Channel 拉取数据, 数据最终分散到多个 Sink
 - . Failover Sink Processor: 故障转移, 可配置优先级, 优先级高的 Sink 单独拉取数据, 若出现故障则由下一优先级的 Sink 继续工作, 恢复后继续由最高优先级的 Sink 工作

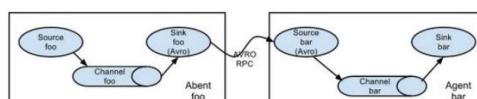
4.4.3 Flume 拓扑结构

将多个 Agent 进行串联。要求使用 Avro 连接上次 Sink 与下层 Source。

Avro: 轻量级的 RPC 框架, 是一个通讯框架, 使用某个端口实现发送数据、接收数据, 类似 nc

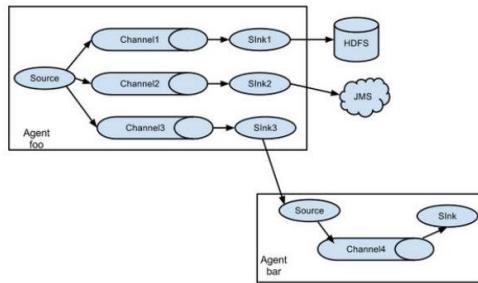
4.4.3.1 简单串联

这种模式是将多个 Flume 顺序连接起来了, 从最初的 Source 开始到最终 Sink 传送的目的存储系统。此模式不建议桥接过多的 Flume 数量, Flume 数量过多不仅会影响传输速率, 而且一旦传输过程中某个节点 Flume 宕机, 会影响整个传输系统。



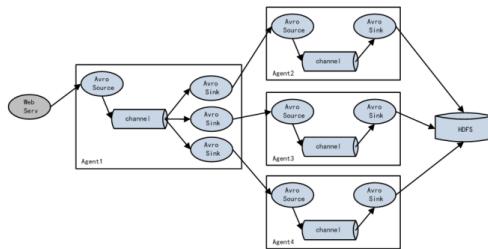
4.4.3.2 复制和多路复用

Flume 支持将事件流向一个或者多个目的地。这种模式可以将相同数据复制到多个 Channel 中，或者将不同数据分发到不同的 Channel 中，Sink 可以选择传送到不同的目的地。



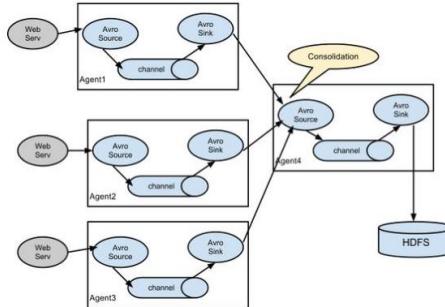
4.4.3.3 负载均衡和故障转移

Flume 支持使用将多个 Sink 逻辑上分到一个 Sink 组，Sink 组配合不同的 SinkProcessor 可以实现负载均衡和错误恢复的功能。



4.4.3.4 聚合

这种模式是我们最常见的，也非常实用，日常 Web 应用通常分布在上百个服务器，大者甚至上千个、上万个服务器。产生的日志，处理起来也非常麻烦。用 Flume 的这种组合方式能很好的解决这一问题，每台服务器部署一个 Flume 采集日志，传送到一个集中收集日志的 Flume，再由此 Flume 上传到 HDFS、Hive、HBase 等，进行日志分析。



4.4.4 Flume 数据流监控

4.4.4.1 监控工具 Ganglia

Ganglia 简介

Ganglia 是一款为 HPC（高性能计算）集群而设计的可扩展的分布式监控系统，它可以监视和显示集群中的节点的各种状态信息，它由运行在各个节点上的 gmond 守护进程来采集 CPU、内存、硬盘利用率、I/O 负载、网络流量情况等方面的数据，然后汇总到 gmetad 守护进程中，使用 rrdtool 存储数据，最后将历史数据以曲线方式通过 PHP 页面呈现。

Ganglia 组成

Ganglia 由 gmond、gmetad 和 gweb 三部分组成。

- gmond (Ganglia Monitoring Daemon) 是一种轻量级服务，安装在每台需要收集指标数据的节点主机上。使用 gmond，你可以很容易收集很多系统指标数据，如 CPU、内存、磁盘、网络和活跃进程的数据等。
- gmetad (Ganglia Meta Daemon) 整合所有信息，并将其以 RRD 格式存储至磁盘的服务。
- gweb (Ganglia Web) Ganglia 可视化工具，gweb 是一种利用浏览器显示 gmetad 所存储数据的 PHP 前端。在 Web 界面中以图表方式展现集群的运行状态下收集的多种不同指标数据。

4.4.4.2 操作 Flume 测试监控

(1) 安装部署 Ganglia

(2) 启动 Ganglia

Bash

```
sudo systemctl start gmond  
sudo systemctl start gmond  
sudo systemctl start gmetad
```

(3) 启动 Flume 任务

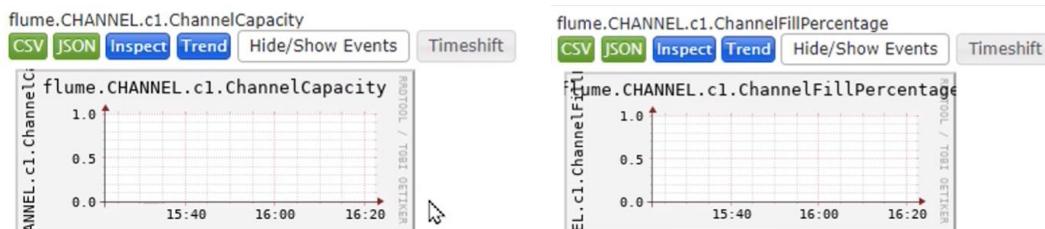
Bash

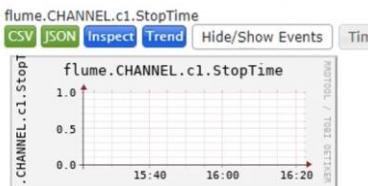
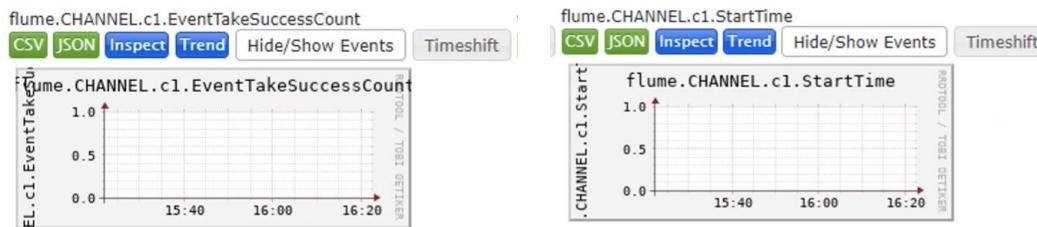
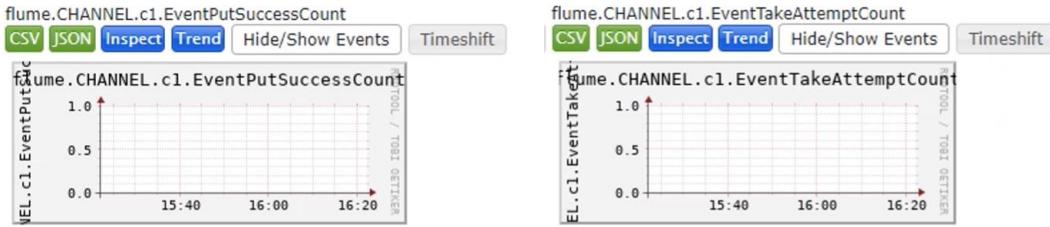
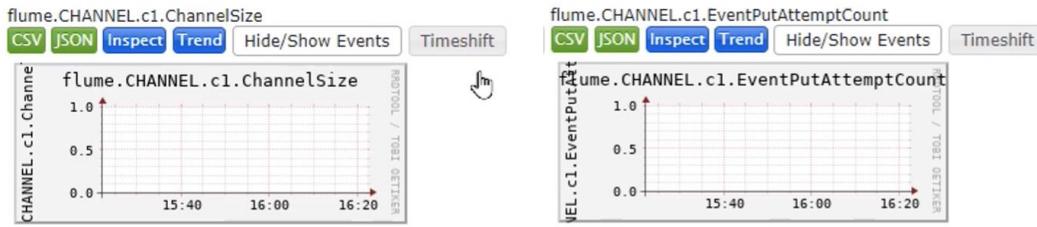
```
bin/flume-ng agent -c conf -f job/flume-netcat-logger.conf -n a1 \  
-Dflume.root.logger==INFO,console \  
-Dflume.monitoring.type=ganglia \  
-Dflume.monitoring.hosts=slave1:8649
```

参数说明：

	Property Name	Description
Ganglia Reporting	flume.monitoring.type	The component type name, has to be ganglia
	flume.monitoring.hosts	Comma-separated list of hostname:port of Ganglia servers

(4) 打开网页观察 Ganglia 监测图





图例说明：

字段（图表名称）	字段含义
EventPutAttemptCount	source 尝试写入 channel 的事件总数量
EventPutSuccessCount	成功写入 channel 且提交的事件总数量
EventTakeAttemptCount	sink 尝试从 channel 拉取事件的总数量
EventTakeSuccessCount	sink 成功读取的事件的总数量
StartTime	channel 启动的时间（毫秒）
StopTime	channel 停止的时间（毫秒）
ChannelSize	目前 channel 中事件的总数量

ChannelFillPercentage	channel 占用百分比
ChannelCapacity	channel 的容量

4.5 Redis

意大利裔工程师 Salvatore Sanfilippo 2009 年开发了一个具有列表结构的内存数据库原型。这个数据库原型支持 O(1) 复杂的推入和弹出操作，并且将数据储存在内存而不是硬盘，所以程序的性能不会受到硬盘 I/O 限制，可以以极快的速度执行针对列表的推入和弹出操作。于是 antirez 使用 C 语言写了这个内存数据库，并给它加上了持久化功能。这就是 Redis。

Redis 版本号第二位如果是奇数，则为非稳定版本如 2.7、2.9、3.1。版本号第二位如果是偶数，则为稳定版本如 2.6、2.8、3.0、3.2。2022 年 4 月正式发布的 Redis 7.0 是目前 Redis 历史版本中变化最大的版本。它有超过 50 个以上新增命令和大量核心特性的新增和改进。



4.5.1 Redis 概述

Redis (Remote Dictionary Server) 是一个基于 C 语言开发的开源数据库 (BSD 许可)，是跨平台的非关系型数据库，也属于一种 nosql 数据库，通常被称为数据结构服务器。与传统数据库不同的是 Redis 的数据是存在内存中的 (内存数据库)，读写速度非常快，被广泛应用于缓存方向。并且，Redis 存储的是 KV 键值对数据。



为了满足不同的业务场景，Redis 内置了多种数据类型实现（比如 String、Hash、Sorted Set、Bitmap）。并且，Redis 还支持事务、持久化、Lua 脚本、多种开箱即用的集群方案（Redis Sentinel、Redis Cluster），通过 Redis Sentinel 提供高可用，通过 Redis Cluster 提供自动分区。

4.5.2 Redis 优势与应用场景

与传统数据库关系(mysql)相比。Redis 是 key-value 数据库(NoSQL 一种)，mysql 是关系数据库；Redis 数据操作主要在内存，而 mysql 主要存储在磁盘；Redis 在某一些场景使用中要明显优于 mysql，比如计数器、排行榜等方面；Redis 通常用于一些特定场景，需要与 Mysql 一起配合使用；两者并不是相互替换和竞争关系，而是共用和配合使用。Redis 优势如下：

- 高性能。** Redis 基于内存，内存的访问速度是磁盘的上千倍；读的速度是 110000 次/秒，写的速度是 81000 次/秒。从硬盘中读取数据比较慢，将用户高频访问的数据存在缓存中，可以使得用户下一次再访问这些数据的时候可以直接从缓存中获取。

2. 持久化存储。Redis 支持内存存储和持久化(RDB+AOF)redis 支持异步将内存中的数据写到硬盘上,同时不影响继续服务,重启的时候可以再次加载进行使用。其也支持数据的备份,即 master-slave 模式的数据备份

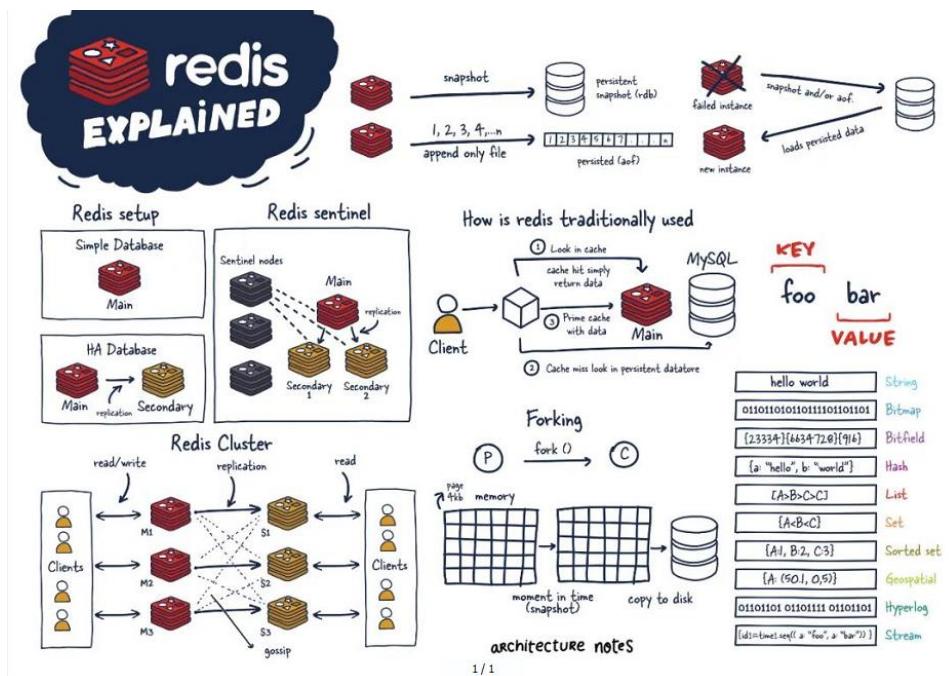
3. 高并发。一般像 MySQL 这类的数据库的 QPS 大概都在 1w 左右 (4 核 8g) , 但是使用 Redis 缓存之后很容易达到 10w+, 甚至最高能达到 30w+ (Redis 集群的话会更高) (QPS (Query Per Second): 服务器每秒可以执行的查询次数:)。Redis 也支持分布式架构和分布式锁。直接操作缓存能够承受的数据库请求数量是远远大于直接访问数据库的, 所以可以考虑把数据库中的部分数据转移到缓存中去, 这样用户的一部分请求会直接到缓存这里而不用经过数据库, 进而也就提高了系统整体的并发。这有助于防止缓存穿透、击穿、雪崩等问题。

4. 数据结构多样。Redis 内置了多种优化过后的数据结构实现, 性能非常高, 不仅仅支持简单的 key-value 类型的数据, 同时还提供 list, set, zset, hash 等数据结构的存储。

5. 消息队列实现。Reids 提供 list 和 set 操作, 这使得 Redis 能作为一个很好的消息队列平台来使用。我们常通过 Reids 的队列功能做购买限制。比如到节假日或者推广期间, 进行一些活动, 对用户购买行为进行限制, 限制今天只能购买几次商品或者一段时间内只能购买一次。

6. 排行榜和点赞功能。在互联网应用中, 有各种各样的排行榜, 如电商网站的月度销量排行榜、社交 APP 的礼物排行榜、小程序的投票排行榜等等。Redis 提供的 zset 数据类型能够快速实现这些复杂的排行榜。

Redis 所能实现的功能总结如下图所示:



4.5.3 Redis 数据类型与基本原理

字符串 (String)

Redis 字符串存储字节序列, 包括文本、序列化对象和二进制数组。因此, 字符串是最基本的 Redis 数据类型。

String 通常用于缓存, 但它们支持额外的功能, 例如也可以实现计数器和执行按位运算。SETNX, SETXX 命令可用于实现分布式锁。INCR 等命令, 可以实现点赞, 例如对于某篇文章, 只要点击了 rest 地址, 直接可以使用 incr key 命令增加一个数字 1, 完成记录数字。

哈希 (Hash)

Redis 哈希是结构化为字段值对集合的记录类型, 是一个 string 类型的 field (字段) 和 value (值) 的映射表。可以使用散列来表示基本对象并存储计数器分组等。简单的应用如使用 hash 实现购物车功能。

列表(List)

Redis 列表是简单的字符串列表, 按照插入顺序排序。你可以添加一个元素到列表的头部 (左边) 或者尾部

(右边)。它的底层实际是个双端链表。

Redis 列表经常用于：实现栈和队列或为后台工作系统构建队列管理。实际的应用如实现微信公众号订阅消息的呈现。

集合(Set)

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据，集合对象的编码可以是 intset 或者 hashtable。Redis 中 Set 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。

可以使用 set 来高效地：追溯非重复项目（例如，跟踪访问给定博客文章的所有唯一 IP 地址）；表示关系（例如，具有给定角色的所有用户的集合）；执行常见的集合运算，例如交集、并集和差集。实际的应用如：

微信抽奖小程序

1 用户 ID，立即参与按钮	sadd key 用户 ID
2 显示已经有多少人参与了	SCARD key
3 抽奖(从 set 中任意选取 N 个中奖人)	SRANDMEMBER key 2 随机抽奖 2 个人，元素不删除 SPOP key 3 随机抽奖 3 个人，元素会删除

朋友圈查看共同点赞好友/社交媒体寻找可能认识的人

1 新增点赞	sadd pub:msgID 点赞用户 ID1 点赞用户 ID2
2 取消点赞	srem pub:msgID 点赞用户 ID
3 展现所有点赞过的用户	SMEMBERS pub:msgID
4 点赞用户数统计，就是常见的点赞红色数字	scard pub:msgID
5 判断某个朋友是否对楼主点赞过	SISMEMBER pub:msgID 用户 ID

有序集合(sorted set)

Redis zset 和 set 一样也是 string 类型元素的集合，且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数，redis 正是通过分数来为集合中的成员进行从小到大的排序。zset 的成员是唯一的，但分数(score)却可以重复。zset 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。

有序集合的一些用例包括：排行榜。例如，您可以使用有序集合轻松维护大型在线游戏中最高分的有序列表。速率限制器。特别是可以使用有序集合来构建滑动窗口速率限制器，以防止过多的 API 请求。具体的应用如根据商品销售对商品进行排序显示：定义商品销售排行榜(sorted set 集合)，key 为 goods:sellsort，分数为商品销售数量。

商品编号 1001 的销量是 9，商品编号 1002 的销量是 15	zadd goods:sellsort 9 1001 15 1002
有一个客户又买了 2 件商品 1001，商品编号 1001 销量加 2	zincrby goods:sellsort 2 1001
求商品销量前 10 名	ZRANGE goods:sellsort 0 9 withscores

地理空间 (GEO)

Redis GEO 主要用于存储地理位置信息，并对存储的信息进行操作，包括：添加地理位置的坐标。获取地理位置的坐标。计算两个位置之间的距离。根据用户给定的经纬度坐标来获取指定范围内的地理位置集合。具体的应用如附近酒店推送或是寻找附近的核酸检测点。

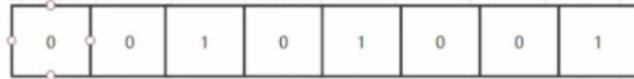
基数统计 (HyperLogLog)

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定且是很小的。在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内

存，就可以计算接近 2^{64} 个不同元素的基数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。具体的应用如统计某个网站/文章的 UV，即 Unique Visitor 独立访客，去重后的真实访客个数（IP 相同的多次访问视为同一个访客）。

位图 (bitmap)

Redis 位图是字符串数据类型的扩展，可让将字符串视为位向量，还可以对一个或多个字符串执行按位运算。下图由许许多多的小格子组成，每一个格子里面只能放 1 或者 0，用它来判断 Y/N 状态，每一个小格子就是一个 bit。1byte = 8bit。



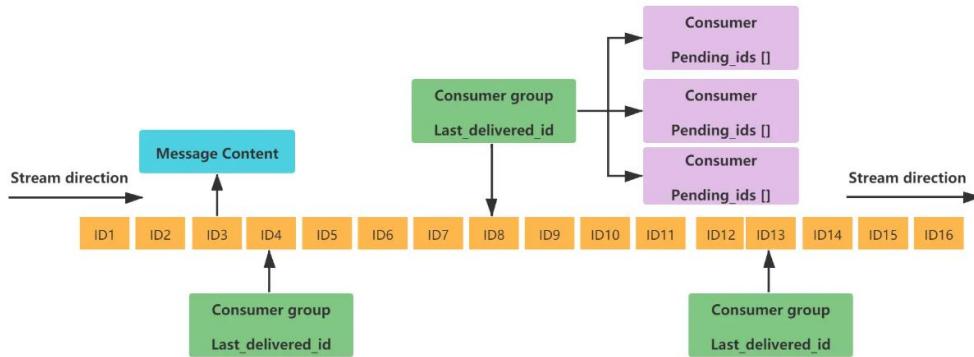
位图用例包括：集合成员对应于整数 0-N 的情况的有效集合表示；对象权限，其中每一位代表一个特定的权限，类似于文件系统存储权限的方式。位图的具体用如：某用户是否登陆过（京东每日签到送京豆）；电影、广告是否被点击播放过；钉钉打卡上下班签到统计。

流 (Stream)

Redis Stream 主要用于消息队列 (MQ, Message Queue)，Redis 本身是有一个 Redis 发布订阅 (pub/sub) 来实现消息队列的功能，但发布订阅有个缺点就是消息无法持久化，如果出现网络断开、Redis 宕机等，消息就会被丢弃。而且也没有 Ack 机制来保证数据的可靠性，假设一个消费者都没有，那消息就直接被丢弃了。

而 Redis Stream 提供了消息的持久化和主备复制功能，可以让任何客户端访问任何时刻的数据，并且能记住每一个客户端的访问位置，还能保证消息不丢失。

Stream 消息队列结构如下所示。Redis 为每个流条目（消息链表）生成一个唯一的 ID。您可以使用这些 ID 稍后检索它们的关联条目，或者读取和处理流中的所有后续条目。Redis 流支持多种分切策略（以防止流无限制地增长）和不止一种消费策略（通过 XREAD、XREADGROUP 和 XRANGE 等命令）。



1	Message Content	消息内容
2	Consumer group	消费组，通过 XGROUP CREATE 命令创建，同一个消费组可以有多个消费者
3	Last_delivered_id	游标，每个消费组会有个游标 last_delivered_id，任意一个消费者读取了消息都会使游标 last_delivered_id 往前移动。
4	Consumer	消费者，消费组中的消费者
5	Pending_ids	消费者会有一个状态变量，用于记录被当前消费已读取但未 ack 的消息 Id，如果客户端没有 ack，这个变量里面的消息 ID 会越来越多，一旦某个消息被 ack 它就开始减少。这个 pending_ids 变量在 Redis 官方被称之为 PEL(Pending Entries List)，记录了当前已经被客户端读取的消息，但是还没有 ack (Acknowledge character: 确认字符)，它用来确保客户端至少消费了消息一次，而不会在网络传输的中途丢失了没处理

Redis 流用例示例包括：事件溯源（例如，跟踪用户操作、点击等）传感器监控（例如，现场设备的读数）通知（例如，将每个用户的日志记录存储在单独的流中）。

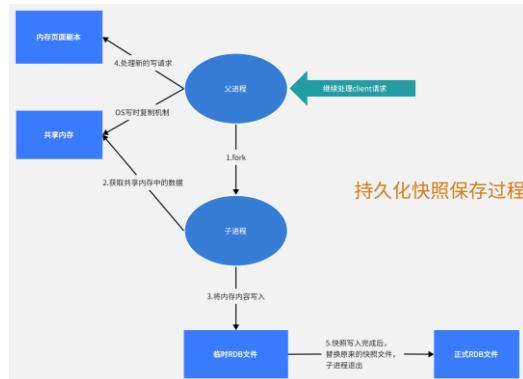
Redis 的基本角色如下图所示：



4.5.4 Redis 持久化

4.5.4.1 RDB

RDB 是 Redis DataBase 的缩写，即内存块照。因为 Redis 的数据时存在内存中的，当服务器宕机时，Redis 中存储的数据就会丢失。这个时候就需要内存快照来恢复 Redis 中的数据了。快照就是在某一时刻，将 Redis 中的所有数据，以文件的形式存储起来。这就类似于照片，当你给朋友拍照时，一张照片就能把朋友一瞬间的形象完全记下来。



在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是 Snapshot 快照，它恢复时是将快照直接读到内存里。

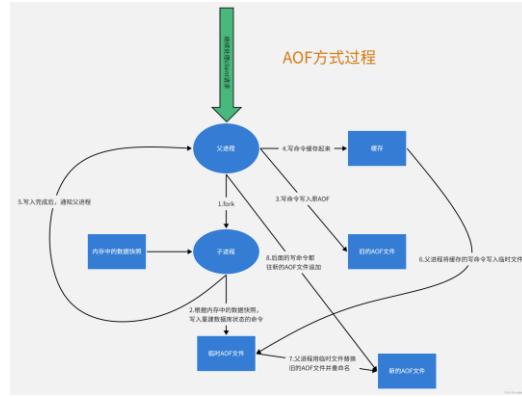
Redis 会单独创建 (fork) 子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何 IO 操作的，这就确保了极高的性能

- 优点：如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那 RDB 方式要比 AOF 方式更加的高效
- 缺点：最后一次持久化后的数据可能丢失

Redis 默认的持久化方式就是 RDB 方式

4.5.4.2 AOF

AOF (Append Only File) 持久化默认是关闭的，通过将 redis.conf 中将 appendonly no，修改为 appendonly yes 来开启 AOF 持久化功能，如果服务器开始了 AOF 持久化功能，服务器会优先使用 AOF 文件来还原数据库状态。只有在 AOF 持久化功能处于关闭状态时，服务器才会使用 RDB 文件来还原数据库状态。



AOF 持久化模式又叫文件追加模式，就是在 Redis 进行写操作的时候，将写命令追加在 AOF 文件的最后，相当于一个历史操作记录文件一般，该操作每秒执行一次，数据的完整度会比较高。

Redis 的 AOF 持久化操作默认是不开启的，通过将 appendonly 设置成 yes 开启，重启 Redis 后生效。

RDB 和 AOF 持久化方式同时开启的话，Redis 启动或恢复数据时，优先使用 AOF 方式。

如果 aof 文件损坏，则启动 Redis 时会由于 AOF 文件损坏而启动失败，这时候可以通过 redis-check-aof 修复 aof 文件，通过命令 redis-check-aof --fix 修复 aof 文件，修复会将错误的命令行去除。

如果在开启 AOF 模式之后将 aof 文件删除，那么 Redis 写命令追加到 aof 失败，不会再创建 aof 文件。

AOF 重写：当 AOF 文件达到 auto-aof-rewrite-percentage 及 auto-aof-rewrite-min-size 两个配置的要求时，会触发重写操作，重写操作实质上是 Redis 创建一个新的 AOF 文件来替代现有的 AOF 文件，新的 AOF 文件的会将可合并的命令合并，减少浪费空间的冗余命令。

- 优点：数据完整性好。
- 缺点：从文件的角度来说，AOF 文件大小远远大于 RDB 文件，修复的速度也比 RDB 慢。AOF 运行效率也要比 RDB 低。

4.6 MongoDB

MongoDB 是一个基于分布式文件存储的开源 NoSQL 数据库系统，由 C++ 编写的。MongoDB 提供了面向文档的存储方式，操作起来比较简单和容易，支持“无模式”的数据建模，可以存储比较复杂的数据类型，是一款非常流行的 文档类型数据库。



在高负载的情况下，MongoDB 天然支持水平扩展和高可用，可以很方便地添加更多的节点/实例，以保证服务性能和可用性。在许多场景下，MongoDB 可以用于代替传统的关系型数据库或键/值存储方式，旨在为 Web 应用提供可扩展的高可用高性能数据存储解决方案。

4.7 Flink

4.7.1 Flink 概述

Apache Flink 是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。Flink 被设计为在所有常见的集群环境中运行，以内存速度和任何规模执行计算。

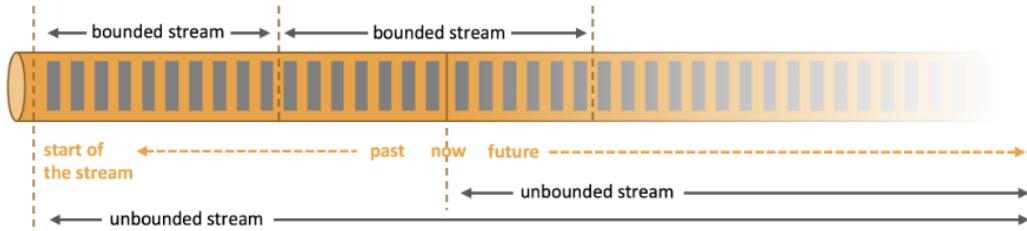
4.7.2 常用名词解释

有界数据

无界流有一个开始但没有定义的结束。它们不会在数据生成时终止并提供数据。无界流必须连续处理，即事件必须在摄取后立即处理。不可能等待所有输入数据到达，因为输入是无界的，不会在任何时间点完成。处理无限数据通常需要按特定顺序摄取事件，例如事件发生的顺序，以便能够推断结果的完整性。

无界数据

有界流有定义的开始和结束。可以通过在执行任何计算之前摄取所有数据来处理有界流。处理有界流不需要有序摄取，因为有界数据集总是可以排序的。有界流的处理也称为批处理。



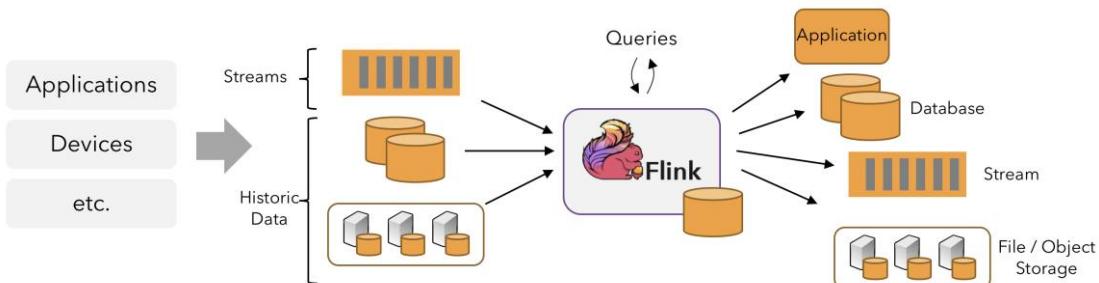
数据流与数据批处理

大数据批处理涉及以批处理或离线方式收集和处理大量数据。在这种方法中，数据在一段时间内收集并存储在数据仓库或分布式文件系统中，例如 Hadoop 分布式文件系统 (HDFS)。一旦积累了足够的数据，就会使用 MapReduce 或 Spark 等技术对其进行批量处理。批处理通常用于不需要实时处理的应用程序，例如数据仓库、ETL（提取、转换、加载）和商业智能。

另一方面，大数据流处理涉及在数据生成或接收时实时处理数据。这种方法涉及将数据作为连续流处理，并且处理可以近乎实时地进行。流处理用于需要实时处理的应用程序，例如欺诈检测、传感器数据处理和点击流分析。

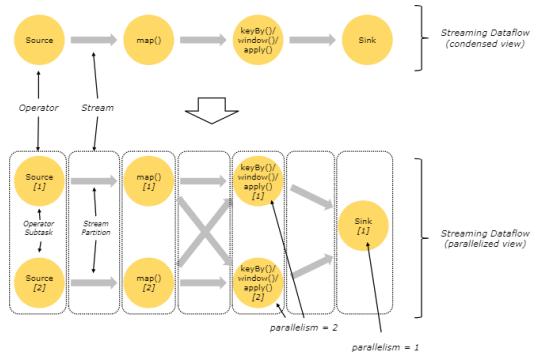
- 数据处理模式：批处理以批处理或离线模式对数据进行操作，而流处理以接近实时的连续流方式对数据进行操作。
- 延迟：批处理的延迟较高，因为数据是批量处理的，而流处理的延迟较低，因为数据是近实时处理的。
- 用例：批处理通常用于不需要实时处理的应用程序，而流处理用于需要实时处理的应用程序。

4.7.3 Flink 基本架构



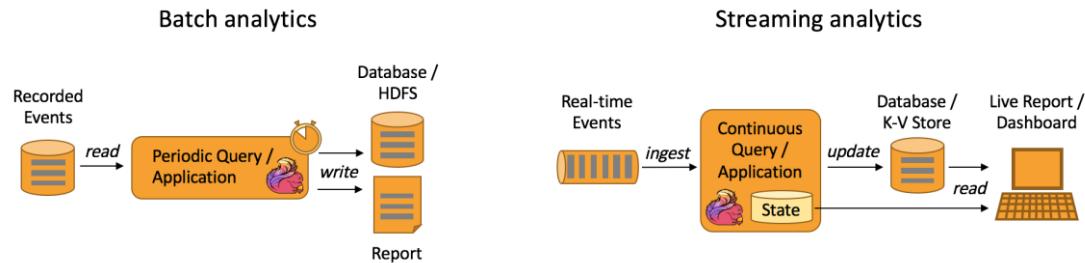
应用程序可能会使用来自消息队列或分布式日志（如 Apache Kafka 或 Kinesis）等流源的实时数据。但是 flink 也可以使用来自各种数据源的有限的历史数据。类似地，Flink 应用程序生成的结果流可以发送到可以作为接收器连接的各种系统。

Flink 中的程序本质上是并行和分布式的。在执行过程中，一个流有一个或多个流分区，每个算子都有一个或多个算子子任务。operator 子任务相互独立，并在不同的线程中执行，并且可能在不同的机器或容器上执行。运算符子任务的数量是该特定运算符的并行度。同一程序的不同运算符可能具有不同级别的并行性。



一对一流（例如上图中的 Source 和 map() 运算符之间）保留元素的分区和排序。这意味着 map() 运算符的 subtask[1] 将以与 Source 运算符的 subtask[1] 生成的顺序相同的顺序看到相同的元素。

重新分配流（如上面的 map() 和 keyBy/window 之间，以及 keyBy/window 和 Sink 之间）改变流的分区。根据所选的转换，每个运算符子任务将数据发送到不同的目标子任务。示例有 keyBy()（通过散列密钥重新分区）、broadcast() 或 rebalance()（随机重新分区）。在重新分配交换中，元素之间的顺序仅保留在每对发送和接收子任务中（例如，map() 的子任务 [1] 和 keyBy/window 的子任务 [2]）。因此，例如，上面显示的 keyBy/window 和 Sink 运算符之间的重新分配引入了关于不同键的聚合结果到达 Sink 的顺序的不确定性。



4.7.4 Flink 理论研究

4.7.4.1 Flink 运行架构



作业管理器 (JobManager)

- 控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的 JobManager 所控制执行。
- JobManager 会先接收到要执行的应用程序，这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的 JAR 包。
- JobManager 会把 JobGraph 转换成一个物理层面的数据流图，这个图被叫做“执行图”（ExecutionGraph），包含了所有可以并发执行的任务。

- JobManager 会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的 TaskManager 上。而在运行过程中，JobManager 会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。

任务管理器（TaskManager）

- Flink 中的工作进程。通常在 Flink 中会有多个 TaskManager 运行，每一个 TaskManager 都包含了一定数量的插槽（slots）。插槽的数量限制了 TaskManager 能够执行的任务数量。
- 启动之后，TaskManager 会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager 就会将一个或者多个插槽提供给 JobManager 调用。JobManager 就可以向插槽分配任务（tasks）来执行了。
- 在执行过程中，一个 TaskManager 可以跟其它运行同一应用程序的 TaskManager 交换数据。

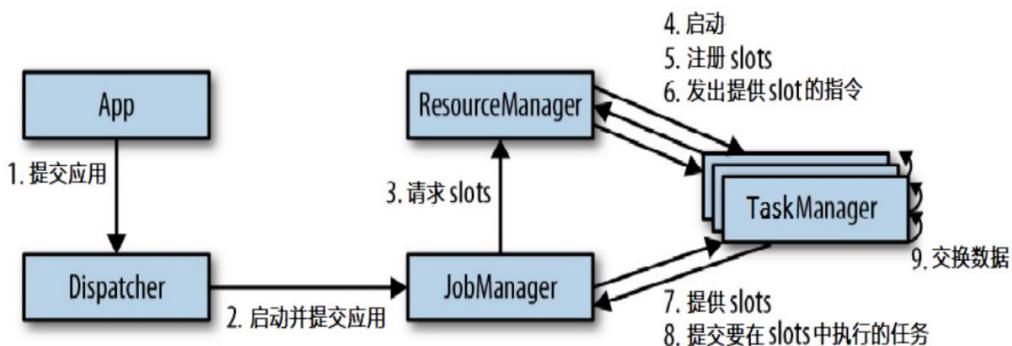
资源管理器（ResourceManager）

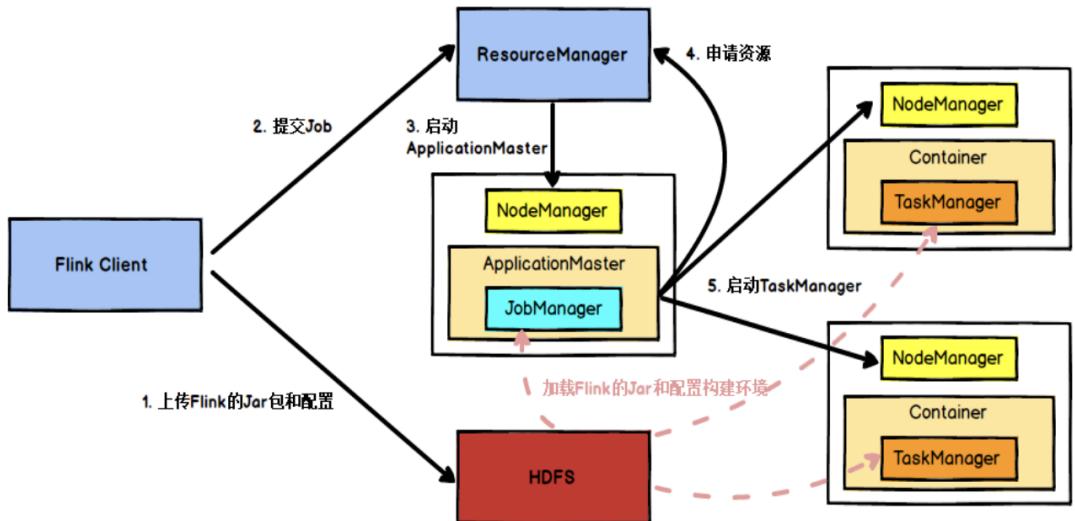
- 主要负责管理任务管理器（TaskManager）的插槽（slot），TaskManager 插槽是 Flink 中定义的处理资源单元。
- Flink 为不同的环境和资源管理工具提供了不同资源管理器，比如 YARN、Mesos、K8s，以及 standalone 部署。
- 当 JobManager 申请插槽资源时，ResourceManager 会将有空闲插槽的 TaskManager 分配给 JobManager。如果 ResourceManager 没有足够的插槽来满足 JobManager 的请求，它还可以向资源提供平台发起会话，以提供启动 TaskManager 进程的容器。

分发器（Dispatcher）

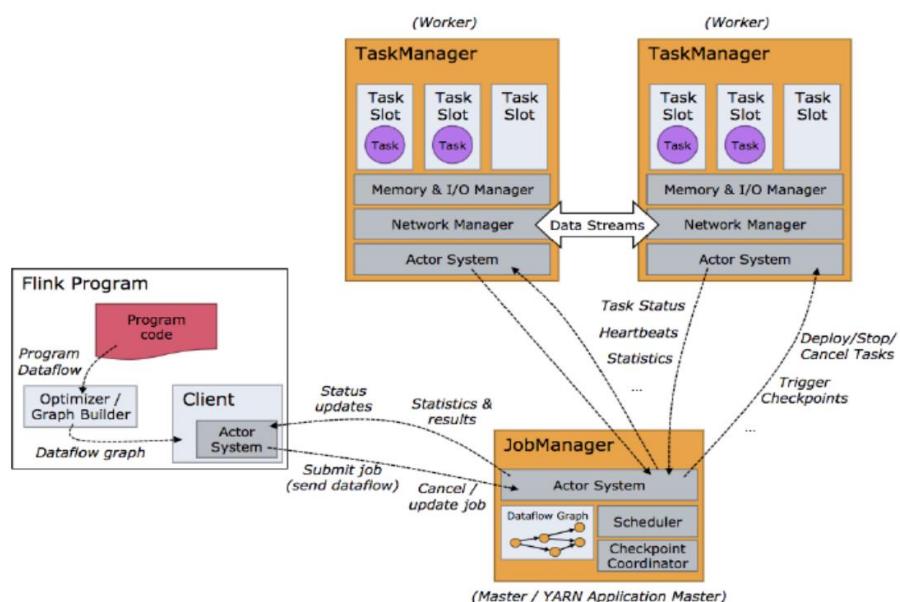
- 可以跨作业运行，它为应用提交提供了 REST 接口。
- 当一个应用被提交执行时，分发器就会启动并将应用移交给一个 JobManager。
- Dispatcher 也会启动一个 Web UI，用来方便地展示和监控作业执行的信息。
- Dispatcher 在架构中可能并不是必需的，这取决于应用提交运行的方式。

任务提交流程

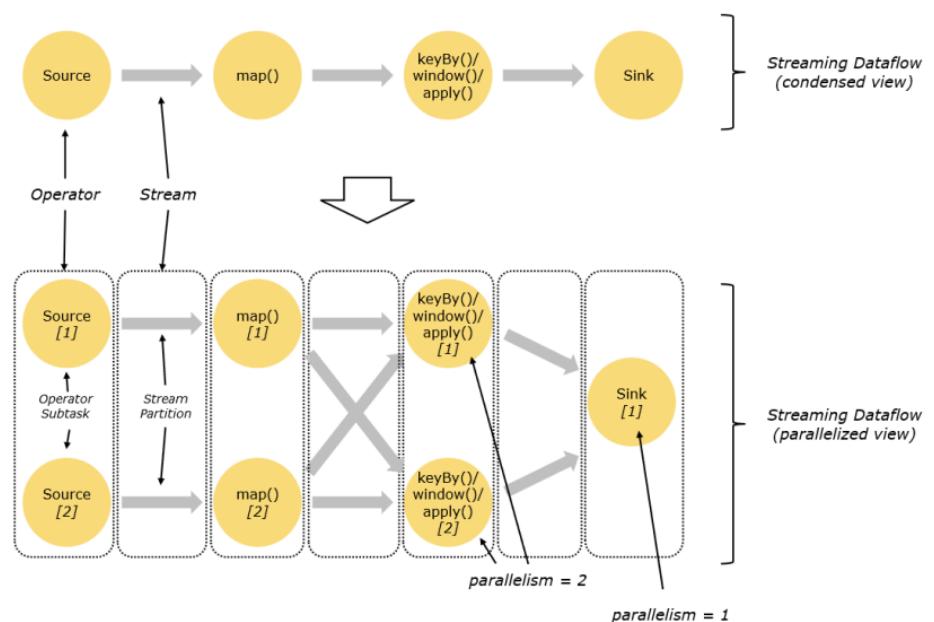




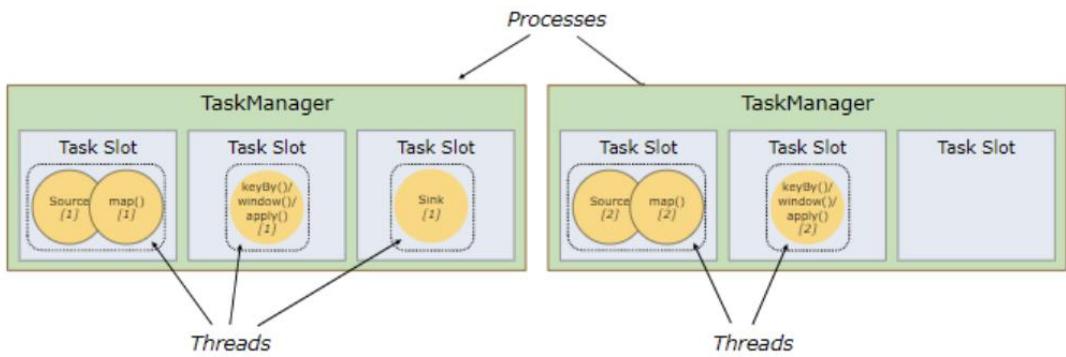
任务调度原理



并行度 (Parallelism)

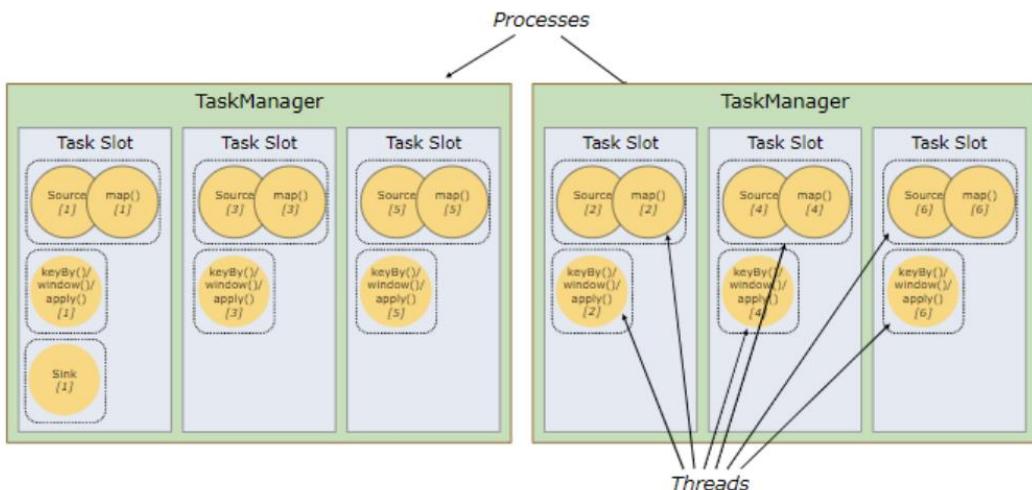


一个特定算子的子任务（subtask）的个数被称之为并行度（parallelism）。一般情况下，一个 stream 的并行度，可以认为就是其所有算子中最大的并行度。



Flink 中每一个 TaskManager 都是一个 JVM 进程，它可能会在独立的线程上执行一个或多个子任务。

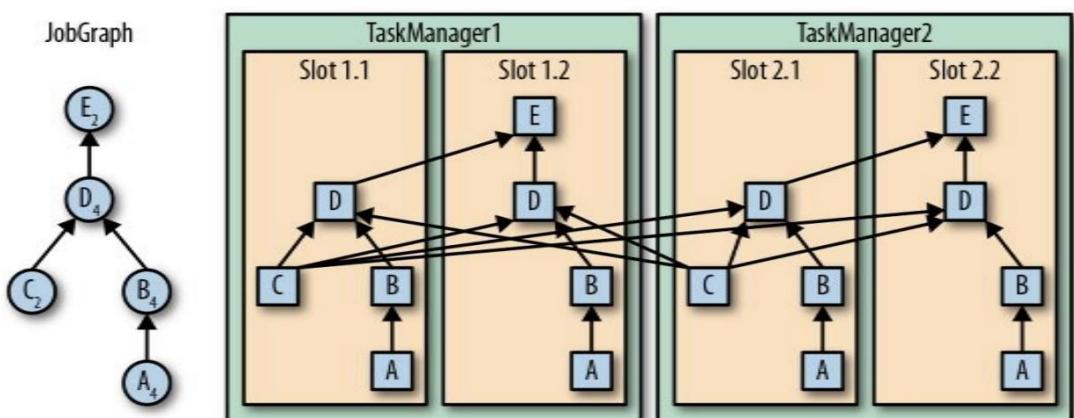
为了控制一个 TaskManager 能接收多少个 task，TaskManager 通过 task slot 来进行控制（一个 TaskManager 至少有一个 slot）。



默认情况下，Flink 允许子任务共享 slot，即使它们是不同任务的子任务。这样的结果是，一个 slot 可以保存作业的整个管道。

Task Slot 是静态的概念，是指 TaskManager 具有的并发执行能力。

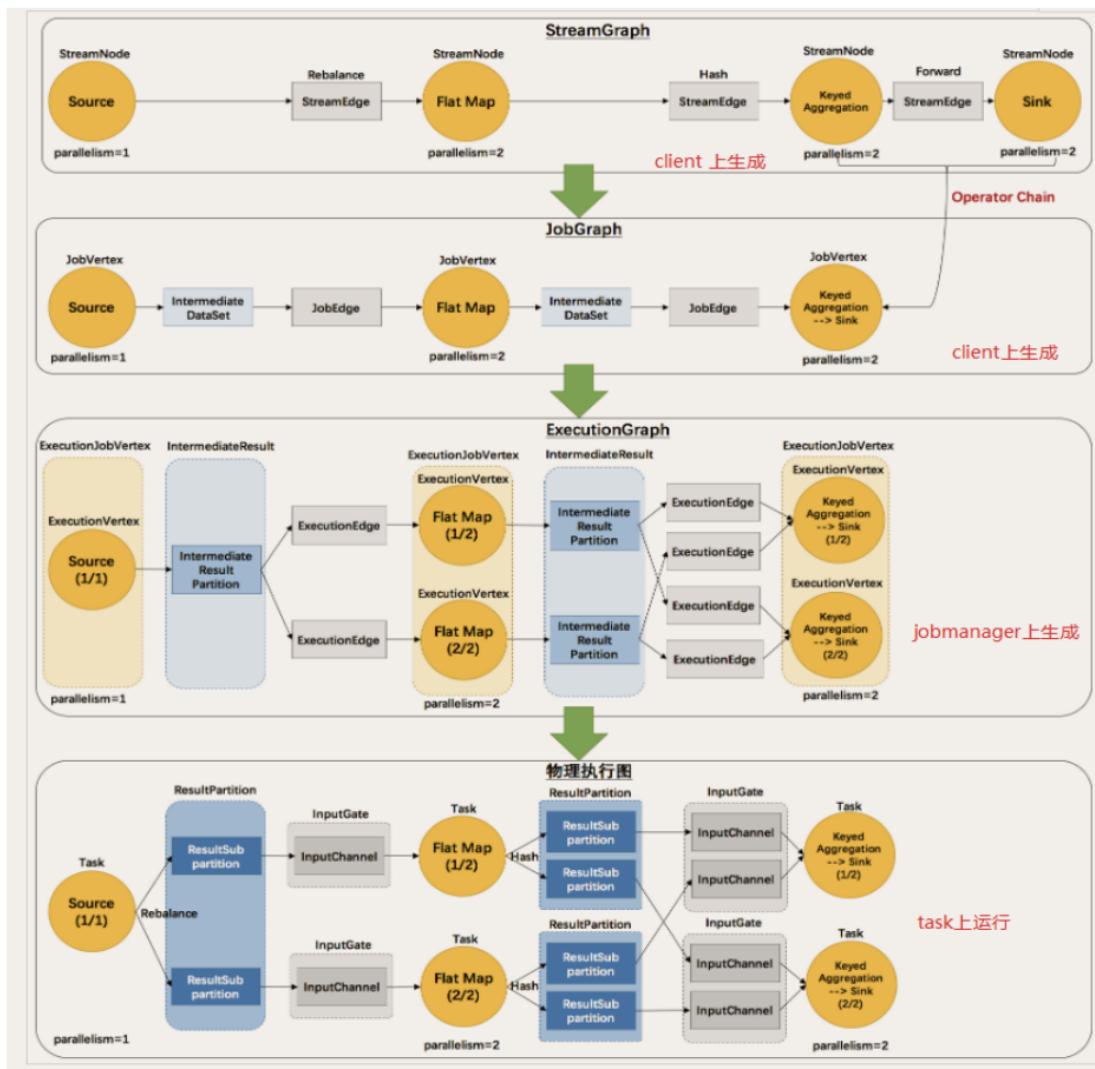
并行子任务的分配



执行图 (ExecutionGraph)

Flink 中的执行图可以分成四层：StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图

- StreamGraph: 是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。
- JobGraph: StreamGraph 经过优化后生成了 JobGraph, 提交给 JobManager 的数据结构。主要的优化为, 将多个符合条件的节点 chain 在一起作为一个节点
- ExecutionGraph: JobManager 根据 JobGraph 生成 ExecutionGraph。ExecutionGraph 是 JobGraph 的并行化版本, 是调度层最核心的数据结构。
- 物理执行图: JobManager 根据 ExecutionGraph 对 Job 进行调度后, 在各个 TaskManager 上部署 Task 后形成的“图”, 并不是一个具体的数据结构。



数据传输形式

一个程序中, 不同的算子可能具有不同的并行度

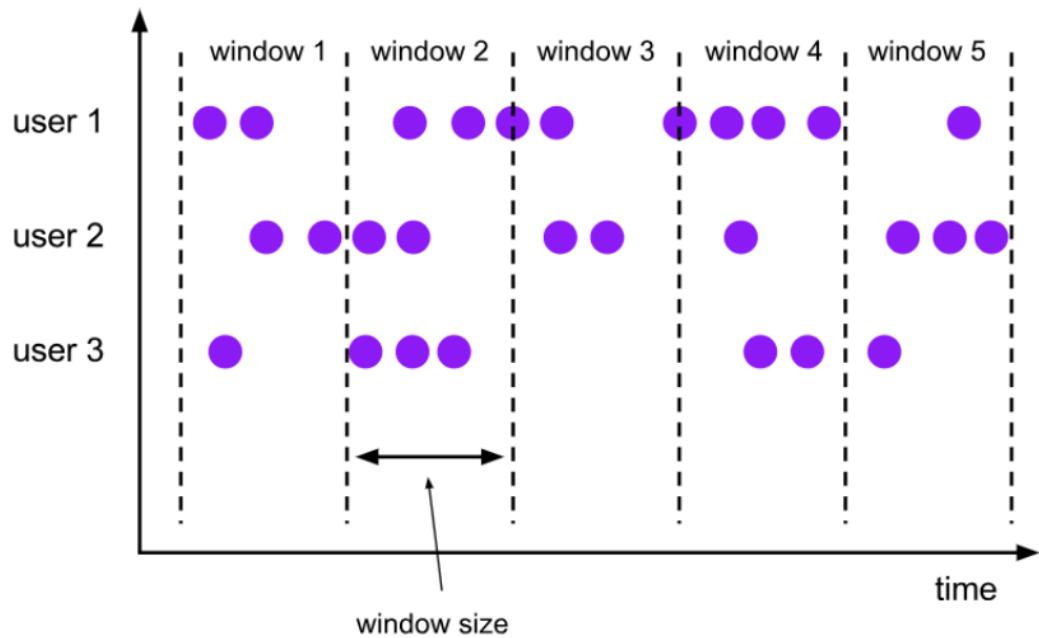
- 算子之间传输数据的形式可以是 one-to-one (forwarding) 的模式也可以是 redistributing 的模式, 具体是哪一种形式, 取决于算子的种类

➤ One-to-one: stream 维护着分区以及元素的顺序 (比如 source 和 map 之间)。这意味着 map 算子的子任务看到的元素的个数以及顺序跟 source 算子的子任务生产的元素的个数、顺序相同。map、filter、flatMap 等算子都是 one-to-one 的对应关系。

➤ Redistributing: stream 的分区会发生改变。每一个算子的子任务依据所选择的 transformation 发送数据到不同的目标任务。例如, keyBy 基于 hashCode 重分区、而 broadcast 和 rebalance 会随机重新分区, 这些算子都会引起 redistribute 过程, 而 redistribute 过程就类似于 Spark 中的 shuffle 过程。

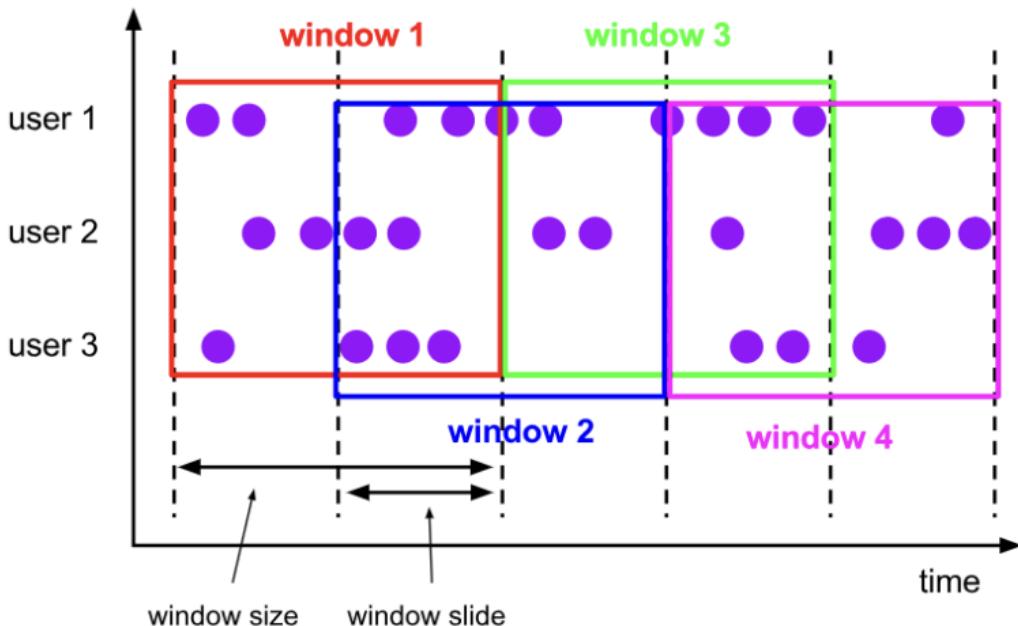
4.7.4.2 Flink Window API

滚动窗口 (Tumbling Windows)



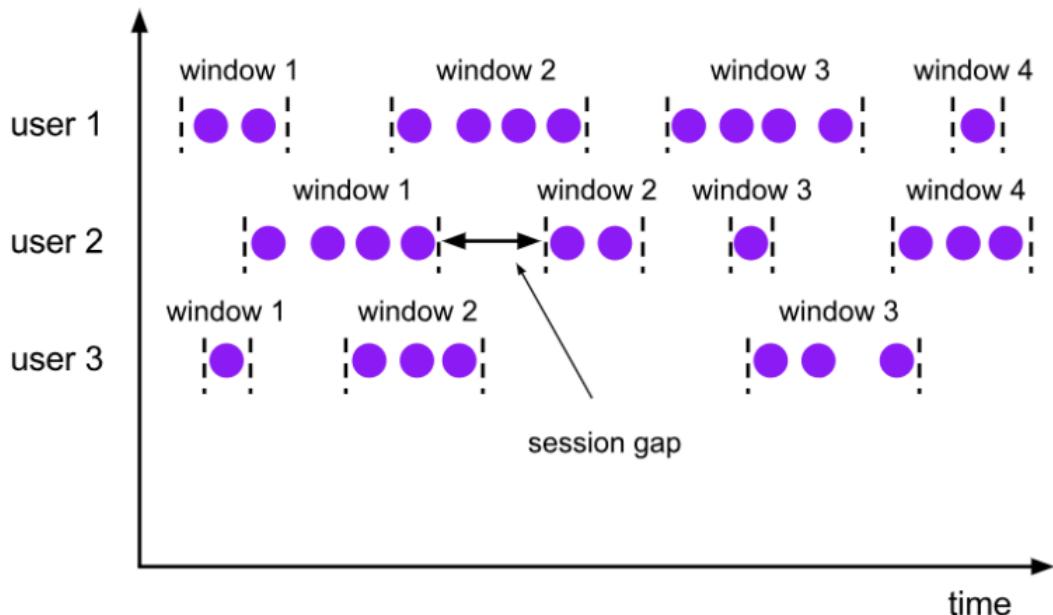
- 将数据依据固定的窗口长度对数据进行切分
- 时间对齐，窗口长度固定，没有重叠

滑动窗口 (Sliding Windows)



- 滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成
- 窗口长度固定，可以有重叠

会话窗口 (Session Windows)



- 由一系列事件组合一个指定时间长度的 timeout 间隙组成，也就是一段时间没有接收到新数据就会生成新的窗口

- 特点：时间无对齐

窗口分配器（window assigner）

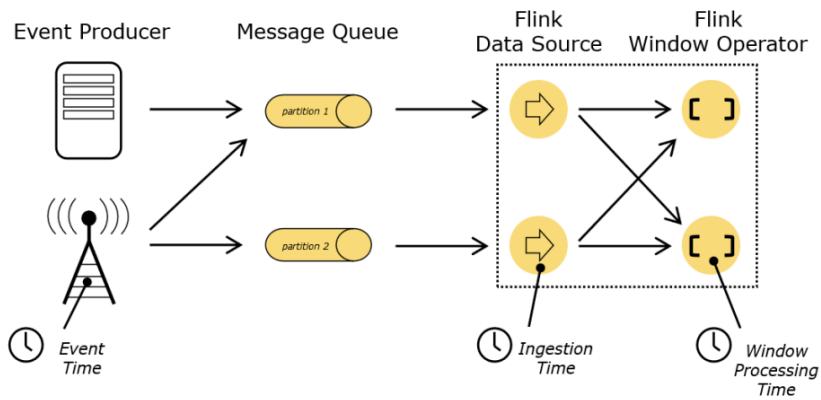
- window() 方法接收的输入参数是一个 WindowAssigner
- WindowAssigner 负责将每条输入的数据分发到正确的 window 中
- Flink 提供了通用的 WindowAssigner
 - 滚动窗口（tumbling window）
 - 滑动窗口（sliding window）
 - 会话窗口（session window）
 - 全局窗口（global window）

窗口函数（window function）

- window function 定义了要对窗口中收集的数据做的计算操作
- 可以分为两类
 - 增量聚合函数（incremental aggregation functions）
 - 每条数据到来就进行计算，保持一个简单状态
 - ReduceFunction, AggregateFunction
 - 全窗口函数（full window functions）
 - 先把窗口所有数据收集起来，等到计算的时候会遍历所有数据
 - ProcessWindowFunction, WindowFunction

4.7.4.3 Flink 中的时间语义和 watermark

时间（Time）语义

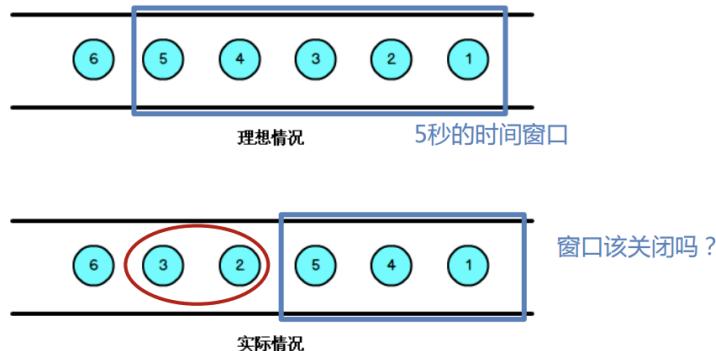


- Event Time: 事件创建的时间
- Ingestion Time: 数据进入 Flink 的时间
- Processing Time: 执行操作算子的本地系统时间，与机器相关

在代码中设置 Event Time

- 我们可以直接在代码中，对执行环境调用 `setStreamTimeCharacteristic` 方法，设置流的时间特性
- 具体的时间，还需要从数据中提取时间戳（timestamp）

乱序数据的影响

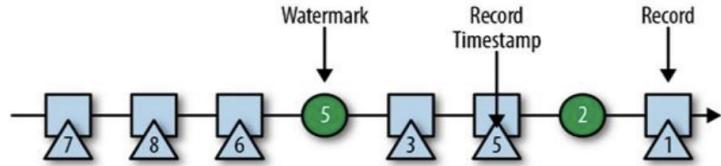


- 当 Flink 以 Event Time 模式处理数据流时，它会根据数据里的时间戳来处理基于时间的算子
- 由于网络、分布式等原因，会导致乱序数据的产生
- 乱序数据会让窗口计算不准确

水位线 (Watermark)

- 怎样避免乱序数据带来计算不正确？
- 遇到一个时间戳达到了窗口关闭时间，不应该立刻触发窗口计算，而是等待一段时间，等迟到的数据来了再关闭窗口
- Watermark 是一种衡量 Event Time 进展的机制，可以设定延迟触发
- Watermark 是用于处理乱序事件的，而正确的处理乱序事件，通常用 Watermark 机制结合 window 来实现；
- 数据流中的 Watermark 用于表示 timestamp 小于 Watermark 的数据，都已经到达了，因此，window 的执行也是由 Watermark 触发的。
- watermark 用来让程序自己平衡延迟和结果正确性

watermark 的特点



- watermark 是一条特殊的数据记录
- watermark 必须单调递增，以确保任务的事件时间时钟在向前推进，而不是在后退
- watermark 与数据的时间戳相关

TimestampAssigner

- 定义了抽取时间戳，以及生成 watermark 的方法，有两种类型

➤ AssignerWithPeriodicWatermarks

- 周期性的生成 watermark：系统会周期性的将 watermark 插入到流中
- 默认周期是 200 毫秒，可以使用 `ExecutionConfig.setAutoWatermarkInterval()` 方法进行设置
- 升序和前面乱序的处理 `BoundedOutOfOrdernessTimestampExtractor`，都是基于周期性 watermark 的。

➤ AssignerWithPunctuatedWatermarks

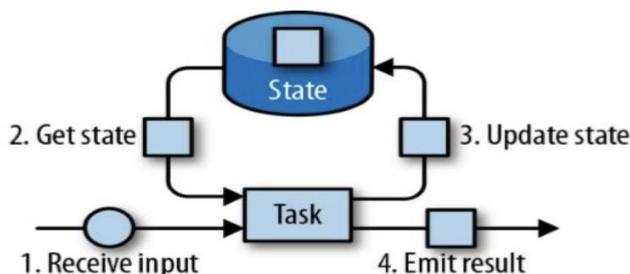
- 没有时间周期规律，可打断的生成 watermark

watermark 的设定

- 在 Flink 中，watermark 由应用程序开发人员生成，这通常需要对相应的领域有一定的了解
- 如果 watermark 设置的延迟太久，收到结果的速度可能就会很慢，解决办法是在水位线到达之前输出一个近似结果
- 而如果 watermark 到达得太早，则可能收到错误结果，不过 Flink 处理迟到数据的机制可以解决这个问题

4.7.4.4 Flink 状态管理

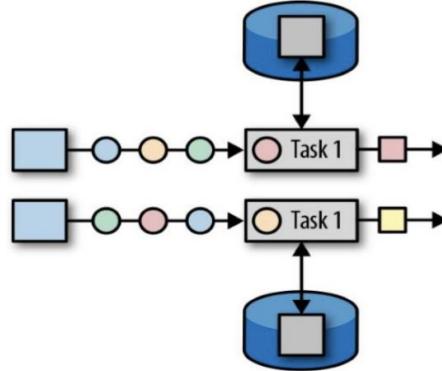
Flink 中的状态



- 由一个任务维护，并且用来计算某个结果的所有数据，都属于这个任务的状态
 - 可以认为状态就是一个本地变量，可以被任务的业务逻辑访问
 - Flink 会进行状态管理，包括状态一致性、故障处理以及高效存储和访问，以便开发人员可以专注于应用程序的逻辑
 - 在 Flink 中，状态始终与特定算子相关联
 - 为了使运行时的 Flink 了解算子的状态，算子需要预先注册其状态
- 总的说来，有两种类型的状态：
- 算子状态（Operator State）

- 算子状态的作用范围限定为算子任务
- 键控状态（Keyed State）
 - 根据输入数据流中定义的键（key）来维护和访问

算子状态（Operator State）



- 算子状态的作用范围限定为算子任务，由同一并行任务所处理的所有数据都可以访问到相同的状态
- 状态对于同一子任务而言是共享的
- 算子状态不能由相同或不同算子的另一个子任务访问

算子状态数据结构

➤ 列表状态（List state）

- 将状态表示为一组数据的列表

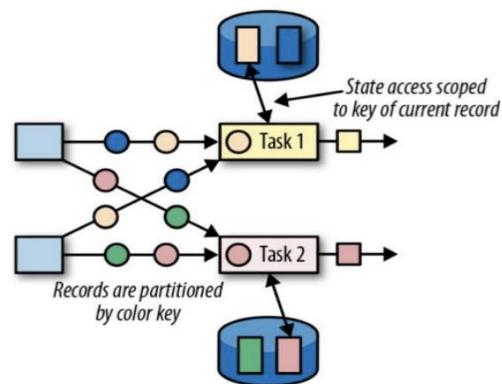
➤ 联合列表状态（Union list state）

- 也将状态表示为数据的列表。它与常规列表状态的区别在于，在发生故障时，或者从保存点（savepoint）启动应用程序时如何恢复

➤ 广播状态（Broadcast state）

- 如果一个算子有多项任务，而它的每项任务状态又都相同，那么这种特殊情况最适合应用广播状态。

键控状态（Keyed State）



- 键控状态是根据输入数据流中定义的键（key）来维护和访问的
- Flink 为每个 key 维护一个状态实例，并将具有相同键的所有数据，都分区到同一个算子任务中，这个任务会维护和处理这个 key 对应的状态
- 当任务处理一条数据时，它会自动将状态的访问范围限定为当前数据的 key

状态后端（State Backends）

- 每传入一条数据，有状态的算子任务都会读取和更新状态

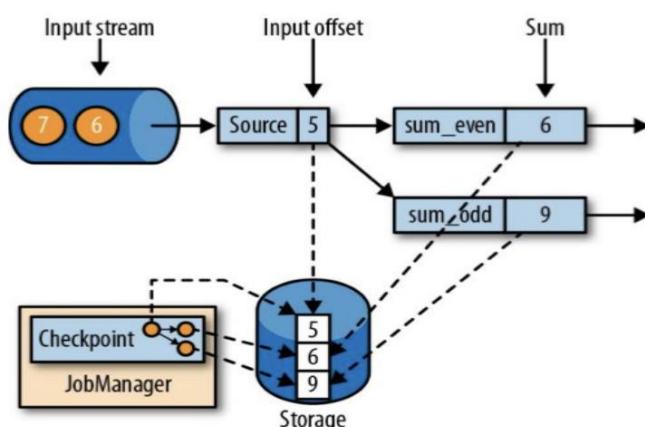
- 由于有效的状态访问对于处理数据的低延迟至关重要，因此每个并行任务都会在本地维护其状态，以确保快速的状态访问
- 状态的存储、访问以及维护，由一个可插入的组件决定，这个组件就叫做状态后端（state backend）
- 状态后端主要负责两件事：本地的状态管理，以及将检查点（checkpoint）状态写入远程存储

➤ MemoryStateBackend

- 内存级的状态后端，会将键控状态作为内存中的对象进行管理，将它们存储在 TaskManager 的 JVM 堆上，而将 checkpoint 存储在 JobManager 的内存中
- 特点：快速、低延迟，但不稳定 ➤ FsStateBackend
- 将 checkpoint 存到远程的持久化文件系统（FileSystem）上，而对于本地状态，跟 MemoryStateBackend 一样，也会存在 TaskManager 的 JVM 堆上
- 同时拥有内存级的本地访问速度，和更好的容错保证
- RocksDBStateBackend
- 将所有状态序列化后，存入本地的 RocksDB 中存储。

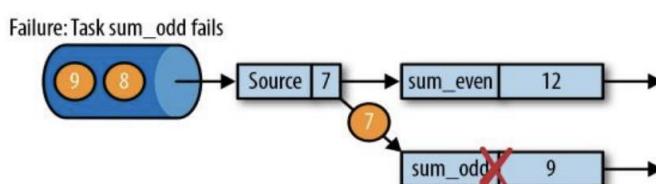
4.7.4.5 Flink 的容错机制

一致性检查点（Checkpoints）



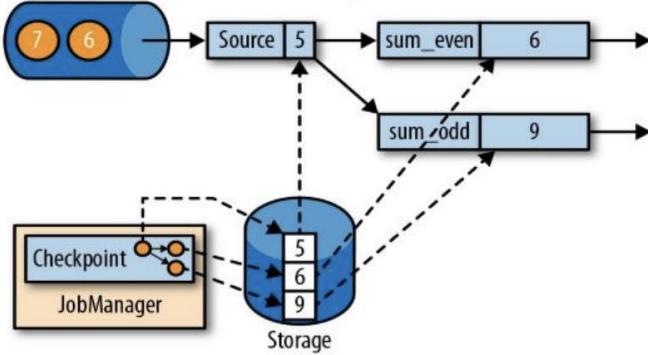
- Flink 故障恢复机制的核心，就是应用状态的一致性检查点
- 有状态流应用的一致检查点，其实就是所有任务的状态，在某个时间点的一份拷贝（一份快照）；这个时间点，应该是所有任务都恰好处理完一个相同的输入数据的时候

从检查点恢复状态



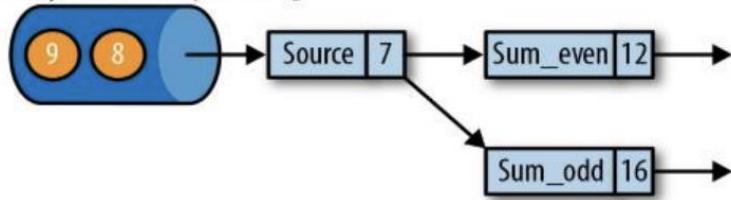
- 在执行流应用程序期间，Flink 会定期保存状态的一致检查点
- 如果发生故障，Flink 将会使用最近的检查点来一致恢复应用程序的状态，并重新启动处理流程
- 遇到故障之后，第一步就是重启应用

Recovery 2: Reset application state from Checkpoint



- 第二步是从 checkpoint 中读取状态，将状态重置
- 从检查点重新启动应用程序后，其内部状态与检查点完成时的状态完全相同

Recovery 3: Continue processing

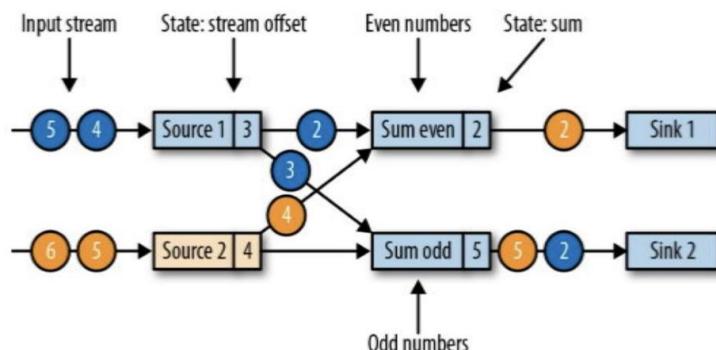


- 第三步：开始消费并处理检查点到发生故障之间的所有数据
- 这种检查点的保存和恢复机制可以为应用程序状态提供“精确一次”（exactly-once）的一致性，因为所有算子都会保存检查点并恢复其所有状态，这样一来所有的输入流就都会被重置到检查点完成时的位置

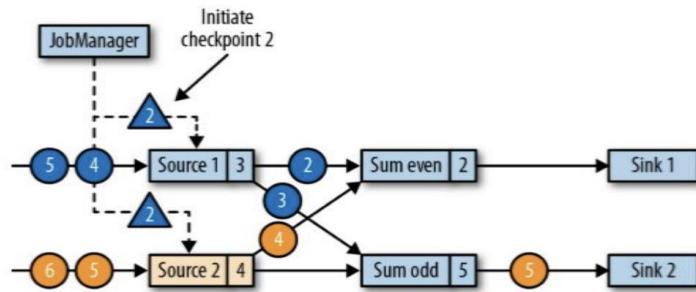
检查点的实现算法

➤ 检查点分界线（Checkpoint Barrier）

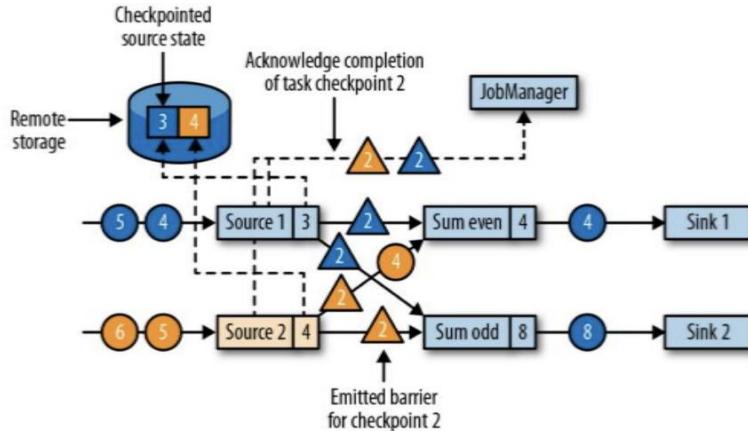
- Flink 的检查点算法用到了一种称为分界线（barrier）的特殊数据形式，用来把一条流上数据按照不同的检查点分开
- 分界线之前到来的数据导致的状态更改，都会被包含在当前分界线所属的检查点中；而基于分界线之后的数据导致的所有更改，就会被包含在之后的检查点中



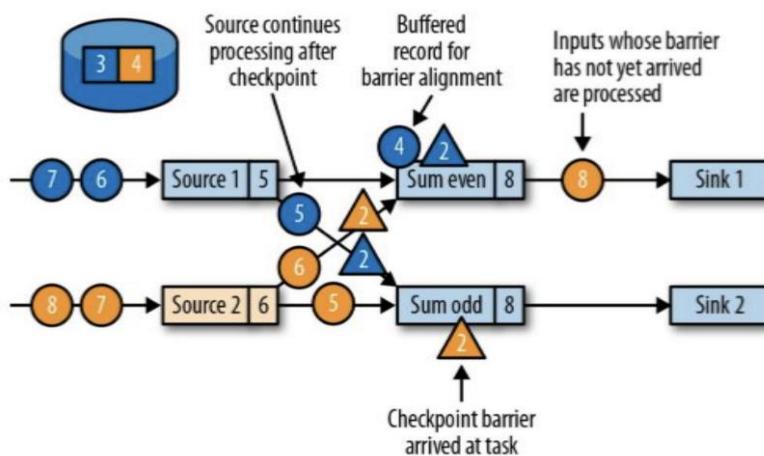
- 现在是一个有两个输入流的应用程序，用并行的两个 Source 任务来读取



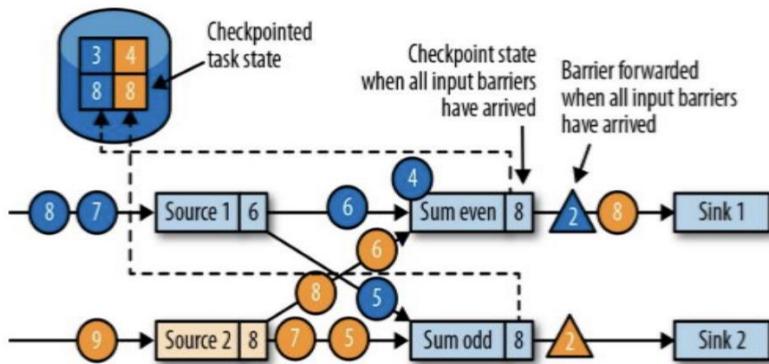
- JobManager 会向每个 source 任务发送一条带有新检查点 ID 的消息，通过这种方式来启动检查点



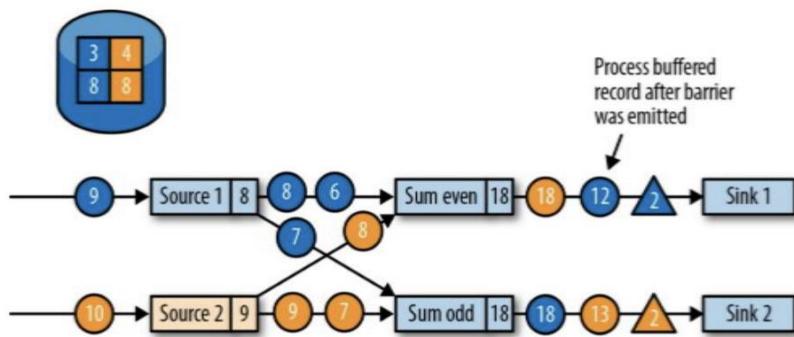
- 数据源将它们的状态写入检查点，并发出一个检查点 barrier
- 状态后端在状态存入检查点之后，会返回通知给 source 任务，source 任务就会向 JobManager 确认检查点完成



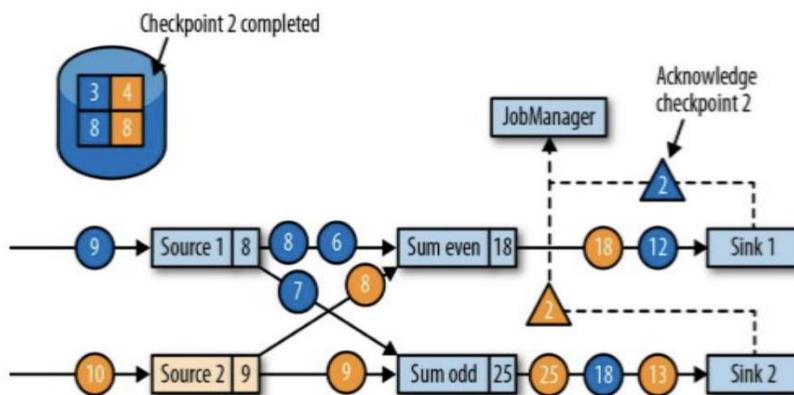
- 分界线对齐：barrier 向下游传递，sum 任务会等待所有输入分区的 barrier 到达
- 对于 barrier 已经到达的分区，继续到达的数据会被缓存
- 而 barrier 尚未到达的分区，数据会被正常处理



- 当收到所有输入分区的 barrier 时，任务就将其状态保存到状态后端的检查点中，然后将 barrier 继续向下游转发



- 向下游转发检查点 barrier 后，任务继续正常的数据处理



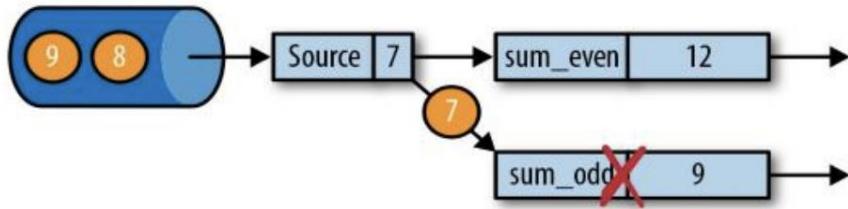
- Sink 任务向 JobManager 确认状态保存到 checkpoint 完毕
- 当所有任务都确认已成功将状态保存到检查点时，检查点就真正完成了

保存点 (Savepoints)

- Flink 还提供了可以自定义的镜像保存功能，就是保存点 (savepoints)
- 原则上，创建保存点使用的算法与检查点完全相同，因此保存点可以认为就是具有一些额外元数据的检查点
- Flink 不会自动创建保存点，因此用户（或者外部调度程序）必须明确地触发创建操作
- 保存点是一个强大的功能。除了故障恢复外，保存点可以用于：有计划的手动备份，更新应用程序，版本迁移，暂停和重启应用，等等

4.7.4.6 Flink 的状态一致性

什么是状态一致性



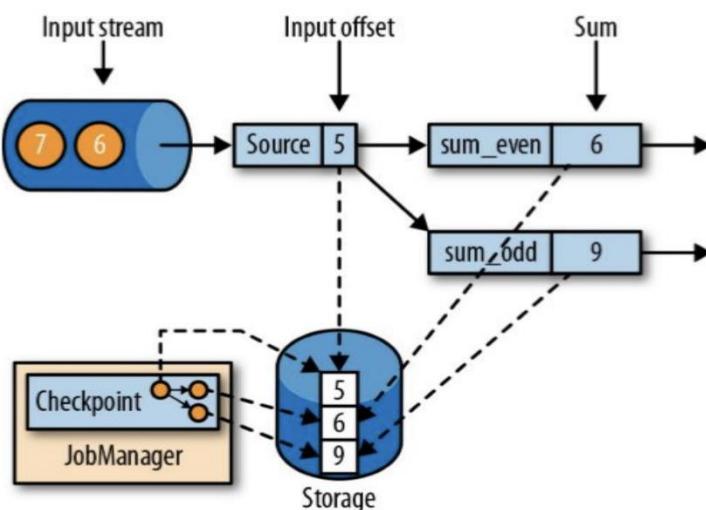
- 有状态的流处理，内部每个算子任务都可以有自己的状态
- 对于流处理器内部来说，所谓的状态一致性，其实就是我们所说的计算结果要保证准确。
- 一条数据不应该丢失，也不应该重复计算
- 在遇到故障时可以恢复状态，恢复以后的重新计算，结果应该也是完全正确的。

状态一致性分类

- AT-MOST-ONCE (最多一次)
 - 当任务故障时，最简单的做法是什么都不干，既不恢复丢失的状态，也不重播丢失的数据。At-most-once语义的含义是最多处理一次事件。
- AT-LEAST-ONCE (至少一次)
 - 在大多数的真实应用场景，我们希望不丢失事件。这种类型的保障称为 atleast-once，意思是所有的事件都得到了处理，而一些事件还可能被处理多次。
- EXACTLY-ONCE (精确一次)
 - 恰好处理一次是最严格的保证，也是最难实现的。恰好处理一次语义不仅仅意味着没有事件丢失，还意味着针对每一个数据，内部状态仅仅更新一次。

一致性检查点 (Checkpoints)

- Flink 使用了一种轻量级快照机制——检查点 (checkpoint) 来保证 exactly-once 语义
- 有状态流应用的一致检查点，其实就是：所有任务的状态，在某个时间点的一份拷贝（一份快照）。而这个时间点，应该是所有任务都恰好处理完一个相同的输入数据的时候。
- 应用状态的一致检查点，是 Flink 故障恢复机制的核心



端到端 (end-to-end) 状态一致性

- 目前我们看到的一致性保证都是由流处理器实现的，也就是说都是在 Flink 流处理器内部保证的；而在真实应用中，流处理应用除了流处理器以外还包含了数据源（例如 Kafka）和输出到持久化系统

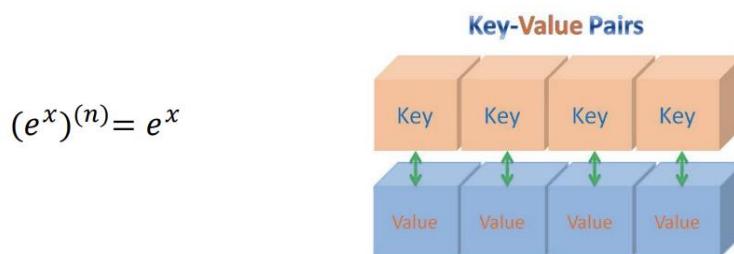
- 端到端的一致性保证，意味着结果的正确性贯穿了整个流处理应用的始终；每一个组件都保证了它自己的一致性
- 整个端到端的一致性级别取决于所有组件中一致性最弱的组件

端到端 exactly-once

- 内部保证 —— checkpoint
- source 端 —— 可重设数据的读取位置
- sink 端 —— 从故障恢复时，数据不会重复写入外部系统
 - 幂等写入
 - 事务写入

幂等写入 (Idempotent Writes)

- 所谓幂等操作，是说一个操作，可以重复执行很多次，但只导致一次结果更改，也就是说，后面再重复执行就不起作用了



事务写入 (Transactional Writes)

- 事务 (Transaction)
 - 应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤销
 - 具有原子性：一个事务中的一系列的操作要么全部成功，要么一个都不做
- 实现思想：构建的事务对应着 checkpoint，等到 checkpoint 真正完成的时候，才把所有对应的结果写入 sink 系统中
- 实现方式
 - 预写日志
 - 两阶段提交

预写日志 (Write-Ahead-Log, WAL)

- 把结果数据先当成状态保存，然后在收到 checkpoint 完成的通知时，一次性写入 sink 系统
- 简单易于实现，由于数据提前在状态后端中做了缓存，所以无论什么 sink 系统，都能用这种方式一批搞定
- DataStream API 提供了一个模板类：GenericWriteAheadSink，来实现这种事务性 sink

两阶段提交 (Two-Phase-Commit, 2PC)

- 对于每个 checkpoint，sink 任务会启动一个事务，并将接下来所有接收的数据添加到事务里
- 然后将这些数据写入外部 sink 系统，但不提交它们 —— 这时只是“预提交”
- 当它收到 checkpoint 完成的通知时，它才正式提交事务，实现结果的真正写入
 - 这种方式真正实现了 exactly-once，它需要一个提供事务支持的外部 sink 系统。Flink 提供了 TwoPhaseCommitSinkFunction 接口。

2PC 对外部 sink 系统的要求

- 外部 sink 系统必须提供事务支持，或者 sink 任务必须能够模拟外部系统上的事务
- 在 checkpoint 的间隔期间里，必须能够开启一个事务并接受数据写入
- 在收到 checkpoint 完成的通知之前，事务必须是“等待提交”的状态。在故障恢复的情况下，这可能需要一些时间。如果这个时候 sink 系统关闭事务（例如超时了），那么未提交的数据就会丢失
- sink 任务必须能够在进程失败后恢复事务
- 提交事务必须是幂等操作

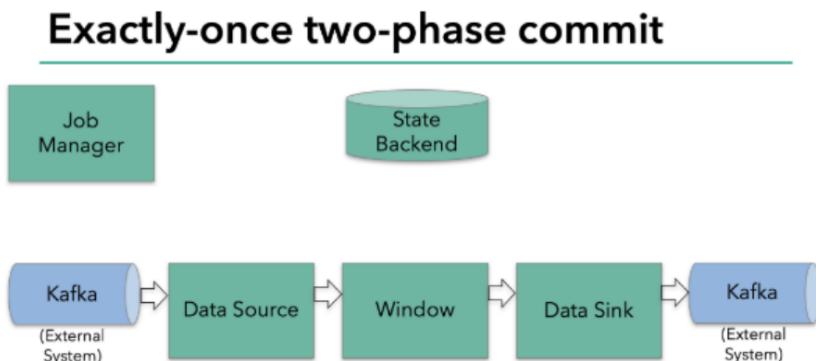
不同 Source 和 Sink 的一致性保证

source sink	不可重置	可重置
任意 (Any)	At-most-once	At-least-once
幂等	At-most-once	Exactly-once (故障恢复时会出现暂时不一致)
预写日志 (WAL)	At-most-once	At-least-once
两阶段提交 (2PC)	At-most-once	Exactly-once

Flink+Kafka 端到端状态一致性的保证

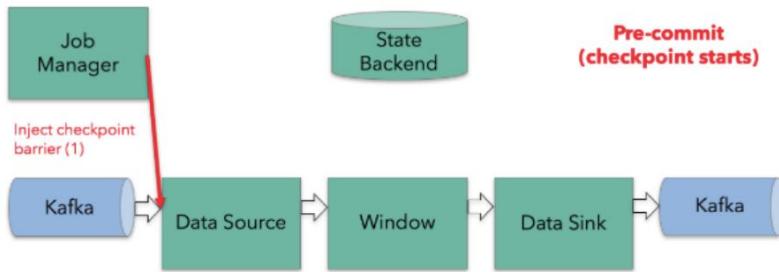
- 内部 —— 利用 checkpoint 机制，把状态存盘，发生故障的时候可以恢复，保证内部的状态一致性
- source —— kafka consumer 作为 source，可以将偏移量保存下来，如果后续任务出现了故障，恢复的时候可以由连接器重置偏移量，重新消费数据，保证一致性
- sink —— kafka producer 作为 sink，采用两阶段提交 sink，需要实现一个 TwoPhaseCommitSinkFunction

Exactly-once 两阶段提交



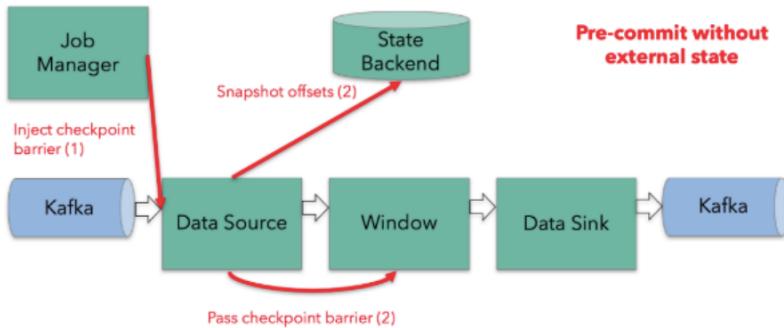
- JobManager 协调各个 TaskManager 进行 checkpoint 存储
- checkpoint 保存在 StateBackend 中，默认 StateBackend 是内存级的，也可以改为文件级的进行持久化保存

Exactly-once two-phase commit



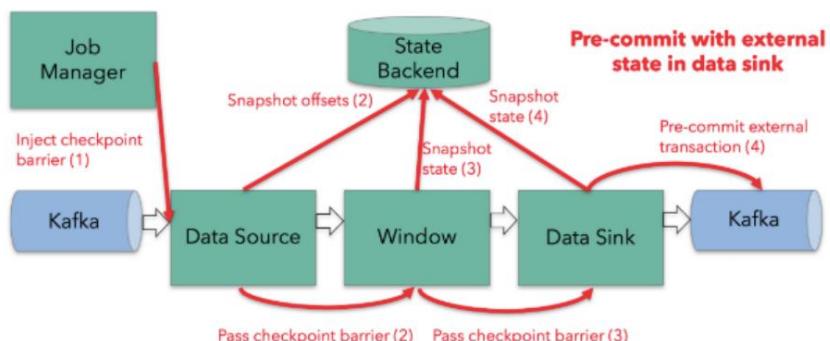
- 当 checkpoint 启动时, JobManager 会将检查点分界线 (barrier) 注入数据流
- barrier 会在算子间传递下去

Exactly-once two-phase commit



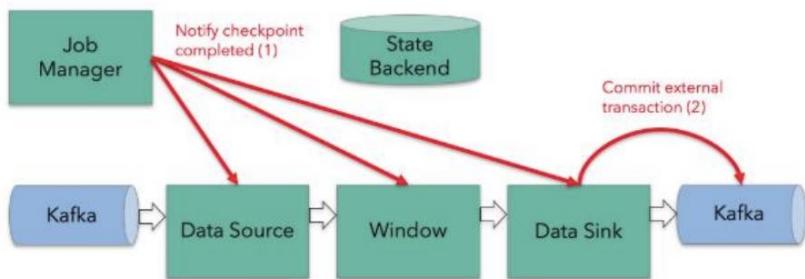
- 每个算子会对当前的状态做个快照, 保存到状态后端
- checkpoint 机制可以保证内部的状态一致性

Exactly-once two-phase commit



- 每个内部的 transform 任务遇到 barrier 时, 都会把状态存到 checkpoint 里
- sink 任务首先把数据写入外部 kafka, 这些数据都属于预提交的事务; 遇到 barrier 时, 把状态保存到状态后端, 并开启新的预提交事务

Exactly-once two-phase commit



- 当所有算子任务的快照完成，也就是这次的 checkpoint 完成时，JobManager 会向所有任务发通知，确认这次 checkpoint 完成
- sink 任务收到确认通知，正式提交之前的事务，kafka 中未确认数据改为“已确认”

Exactly-once 两阶段提交步骤

- 第一条数据来了之后，开启一个 kafka 的事务（transaction），正常写入 kafka 分区日志但标记为未提交，这就是“预提交”
- jobmanager 触发 checkpoint 操作，barrier 从 source 开始向下传递，遇到 barrier 的算子将状态存入状态后端，并通知 jobmanager
- sink 连接器收到 barrier，保存当前状态，存入 checkpoint，通知 jobmanager，并开启下一阶段的事务，用于提交下个检查点的数据
- jobmanager 收到所有任务的通知，发出确认信息，表示 checkpoint 完成
- sink 任务收到 jobmanager 的确认信息，正式提交这段时间的数据
- 外部 kafka 关闭事务，提交的数据可以正常消费了。

4.8 Maxwell

MAXWELL

4.8.1 Maxwell 简介

Maxwell 是由美国 Zendesk 公司开源，用 Java 编写的 MySQL 变更数据抓取软件。它会实时监控 Mysql 数据库的数据变更操作（包括 insert、update、delete），并将变更数据以 JSON 格式发送给 Kafka、Kinesis 等流数据处理平台。官网地址：<http://maxwells-daemon.io/>

Maxwell 输出数据格式

插入	更新	删除
<pre>mysql> insert into gmall.student values(1,'zhangsan');</pre> <p>maxwell 输出:</p> <pre>{ "database": "gmall", "table": "student", "type": "insert", "ts": 1634004537, "xid": 1530970, "commit": true, "data": { "id": 1, "name": "zhangsan" } }</pre>	<pre>mysql> update gmall.student set name='lisi' where id=1;</pre> <p>maxwell 输出:</p> <pre>{ "database": "gmall", "table": "student", "type": "update", "ts": 1634004653, "xid": 1531916, "commit": true, "data": { "id": 1, "name": "lisi" }, "old": { "name": "zhangsan" } }</pre>	<pre>mysql> delete from gmall.student where id=1;</pre> <p>maxwell 输出:</p> <pre>{ "database": "gmall", "table": "student", "type": "delete", "ts": 1634004751, "xid": 1532725, "commit": true, "data": { "id": 1, "name": "lisi" } }</pre>

注: Maxwell 输出的 json 字段说明:

字段	解释
database	变更数据所属的数据库
table	表更数据所属的表
type	数据变更类型
ts	数据变更发生的时间
xid	事务 id
comm it	事务提交标志, 可用于重新组装事务
data	对于 insert 类型, 表示插入的数据; 对于 update 类型, 标识修改之后的数据; 对于 delete 类型, 表示删除的数据
old	对于 update 类型, 表示修改之前的数据, 只包含变更字段

4.8.2 Maxwell 原理

Maxwell 的工作原理是实时读取 MySQL 数据库的二进制日志 (Binlog), 从中获取变更数据, 再将变更数据以 JSON 格式发送至 Kafka 等流处理平台。

4.8.2.1 MySQL 二进制日志

二进制日志 (Binlog) 是 MySQL 服务端非常重要的一种日志, 它会保存 MySQL 数据库的所有数据变更记录。Binlog 的主要作用包括主从复制和数据恢复。Maxwell 的工作原理和主从复制密切相关。

4.8.2.2 MySQL 主从复制

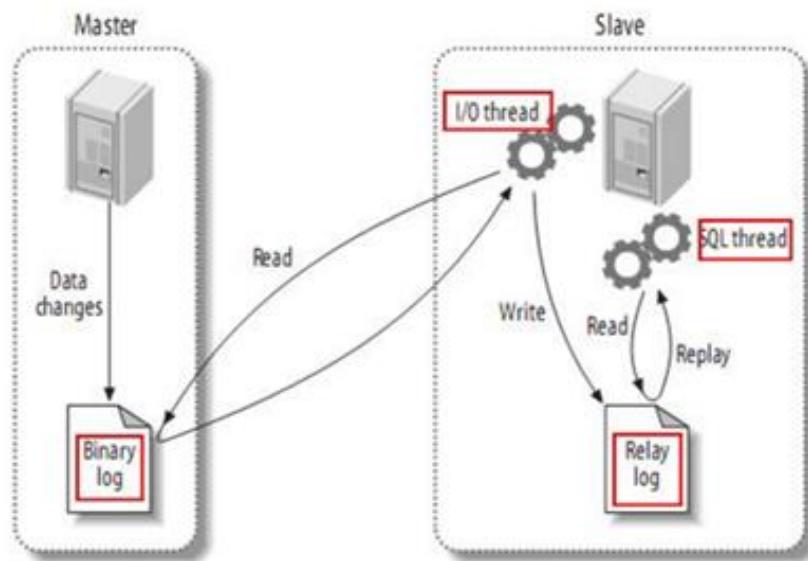
MySQL 的主从复制，就是用来建立一个和主数据库完全一样的数据库环境，这个数据库称为从数据库。

1) 主从复制的应用场景如下：

- (1) 做数据库的热备：主数据库服务器故障后，可切换到从数据库继续工作。
- (2) 读写分离：主数据库只负责业务数据的写入操作，而多个从数据库只负责业务数据的查询工作，在读多写少场景下，可以提高数据库工作效率。

2) 主从复制的工作原理如下：

- (1) Master 主库将数据变更记录，写到二进制日志(binary log)中
- (2) Slave 从库向 mysql master 发送 dump 协议，将 master 主库的 binary log events 拷贝到它的中继日志(relay log)
- (3) Slave 从库读取并回放中继日志中的事件，将改变的数据同步到自己的数据库。



Maxwell 原理：就是将自己伪装成 slave，并遵循 MySQL 主从复制的协议，从 master 同步数据。

4.9 DataX

4.9.1 DataX 简介

DataX 是阿里巴巴开源的一个异构数据源离线同步工具，致力于实现包括关系型数据库(MySQL、Oracle 等)、HDFS、Hive、ODPS、HBase、FTP 等各种异构数据源之间稳定高效的数据同步功能。

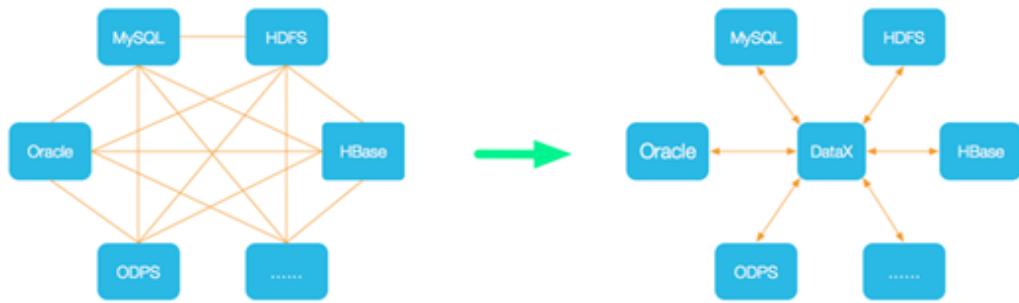
源码地址：<https://github.com/alibaba/DataX>

DataX 支持的数据源

DataX 目前已经有了比较全面的插件体系，主流的 RDBMS 数据库、NOSQL、大数据计算系统都已经接入，

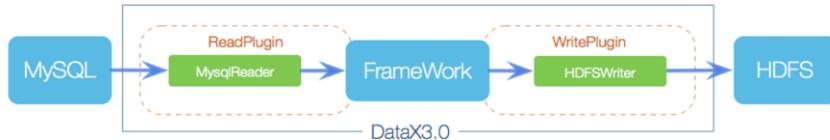
4.9.2 DataX 原理

为了解决异构数据源同步问题，DataX 将复杂的网状的同步链路变成了星型数据链路，DataX 作为中间传输载体负责连接各种数据源。当需要接入一个新的数据源的时候，只需要将此数据源对接到 DataX，便能跟已有的数据源做到无缝数据同步。



DataX 框架设计

DataX 本身作为离线数据同步框架，采用 Framework + plugin 架构构建。将数据源读取和写入抽象成为 Reader/Writer 插件，纳入到整个同步框架中。



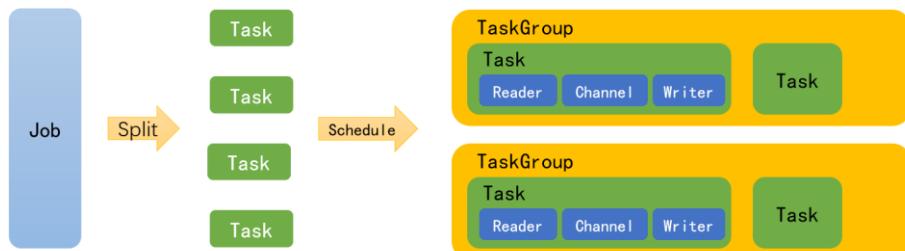
Reader: 数据采集模块，负责采集数据源的数据，将数据发送给**Framework**。

Writer: 数据写入模块，负责不断向**Framework**取数据，并将数据写入到目的端。

Framework: 用于连接**reader**和**writer**，作为两者的数据传输通道，并处理缓冲，流控，并发，数据转换等核心技术问题。

DataX 运行流程

下面用一个 DataX 作业生命周期的时序图说明 DataX 的运行流程、核心概念以及每个概念之间的关系。



Job: 单个数据同步的作业，称为一个Job，一个Job启动一个进程。

Task: 根据不同数据源的切分策略，一个Job会切分为多个Task，Task是DataX作业的最小单元，每个Task负责一部分数据的同步工作。

TaskGroup: Scheduler调度模块会对Task进行分组，每个Task组称为一个Task Group。每个Task Group负责以一定的并发度运行其所分得的Task，单个Task Group的并发度为5。

Reader→Channel→Writer: 每个Task启动后，都会固定启动Reader→Channel→Writer的线程来完成同步工作。

DataX 调度决策思路

举例来说，用户提交了一个 DataX 作业，并且配置了总的并发度为 20，目的是对一个有 100 张分表的 mysql 数据源进行同步。DataX 的调度决策思路是：

- 1) DataX Job 根据分库分表切分策略，将同步工作分成 100 个 Task。
- 2) 根据配置的总的并发度 20，以及每个 Task Group 的并发度 5，DataX 计算共需要分配 4 个 TaskGroup。
- 3) 4 个 TaskGroup 平分 100 个 Task，每一个 TaskGroup 负责运行 25 个 Task。

4.9.3 DataX 与 Sqoop 对比

功能	DataX	Sqoop
运行模式	单进程多线程	MR
分布式	不支持, 可以通过调度系统规避	支持
流控	有流控功能	需要定制
统计信息	已有一些统计, 上报需定制	没有, 分布式的数据收集不方便
数据校验	在 core 部分有校验功能	没有, 分布式的数据收集不方便
监控	需要定制	需要定制

Sqoop 依赖于 Hadoop 生态, 充分利用了 map-reduce 计算框架, 在 Hadoop 的框架中运行, 对 HDFS、Hive 支持友善, 在处理数仓大表的速度相对较快, 但不具备统计和校验能力。

DataX 无法分布式部署, 需要依赖调度系统实现多客户端, 可以在传输过程中进行过滤, 并且可以统计传输数据的信息, 因此在业务场景复杂(表结构变更)更适用, 同时对于不同的数据源支持更好, 同时不支持自动创建表和分区。支持流量控制, 支持运行信息收集, 及时跟踪数据同步情况。

Sqoop 采用命令行的方式调用, 比如容易与我们的现有的调度监控方案相结合, DataX 采用 xml 配置文件的方式, 在开发运维上还是有点不方便。

Sqoop 只可以在关系型数据库和 Hadoop 组件之间进行数据迁移, 而在 Hadoop 相关组件之间, 比如 hive 和 hbase 之间就无法使用 sqoop 互相导入导出数据, 同时在关系型数据库之间, 比如 mysql 和 oracle 之间也无法通过 sqoop 导入导出数据。与之相反, DataX 能够分别实现关系型数据库 Hadoop 组件之间、关系型数据库之间、Hadoop 组件之间的数据迁移。

4.10 sqoop

4.10.1 sqoop 简介

Apache Sqoop 是在 Hadoop 生态体系和 RDBMS 体系之间传送数据的一种工具。(Apache 基金会已经不再维护)

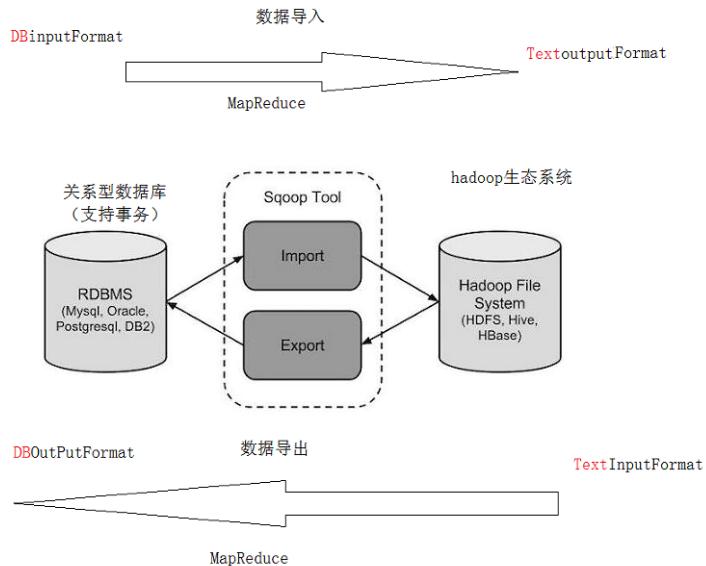
Sqoop 具有以下主要功能:

1. 从关系数据库中导入数据到 Hadoop 的 HDFS、Hive 和 HBase 中。
2. 从 Hadoop 的 HDFS、Hive 和 HBase 中导出数据到关系数据库中。
3. 在关系数据库中执行 SQL 查询, 并将结果导入到 Hadoop 中。
4. 在 Hadoop 中对数据进行转换后, 将结果导出到关系数据库中。
5. 支持增量导入与导出, 只传输新增或更新的数据。
6. 支持对数据进行压缩, 加快数据传输速度。
7. 支持创建和编辑元数据, 用于定义数据传输任务。
8. 支持并行数据传输, 以提高效率。

Sqoop 通过关系数据库的 JDBC 接口来实现与数据库的连接与交互。它可以高效并行地在 Hadoop 与关系数据库之间进行大容量数据传输。

这使得我们可以利用 Hadoop 进行大数据存储与离线计算, 利用关系数据库进行 OLTP 交易处理, 并通过 Sqoop 实现两者之间的数据交互, 构建统一的数据采集与分析系统。

4.10.2 sqoop 原理



Sqoop 工作机制是将导入或导出命令翻译成 mapreduce 程序来实现。在翻译出的 mapreduce 中主要是对 inputformat 和 outputformat 进行定制。

Hadoop 生态系统包括：HDFS、Hive、Hbase 等

RDBMS 体系包括：Mysql、Oracle、DB2 等

Sqoop 可以理解为：“SQL 到 Hadoop 和 Hadoop 到 SQL”。

4.11 Hbase



4.11.1 Hbase 简介

Apache HBase™ 是以 hdfs 为数据存储的，一种分布式、可扩展的 NoSQL 数据库。

HBase 的设计理念依据 Google 的 BigTable 论文，论文中对于数据模型的首句介绍。Bigtable 是一个稀疏的、分布式的、持久的多维排序 map。

之后对于映射的解释如下：该映射由行键、列键和时间戳索引；映射中的每个值都是一个未解释的字节数组。最终 HBase 关于数据模型和 BigTable 的对应关系如下：

HBase 使用与 Bigtable 非常相似的数据模型。用户将数据行存储在带标签的表中。数据行具有可排序的键和任意数量的列。该表存储稀疏，因此如果用户喜欢，同一表中的行可以具有疯狂变化的列。

最终理解 HBase 数据模型的关键在于稀疏、分布式、多维、排序的映射。其中映射 map 指代非关系型数据库的 key-Value 结构。

4.11.2 Hbase 逻辑与物理结构

逻辑结构

存储数据稀疏，数据存储多维，不同的行具有不同的列。

数据存储整体有序，按照RowKey的字典序排列，RowKey为Byte数组

列	personal_info			office_info		列族
	Row Key	name	city	phone	tel	
row_key1	张三	北京	131*****	010-11111111	atguigu	
row_key2	王五	广州		010-11111111	atguigu	
row_key3		深圳	187*****		atguigu	
row_key4	横七	大连	134*****		atguigu	
row_key5	竖八		139*****		atguigu	
row_key6	金九	武汉		010-11111111	atguigu	
row_key7	银十	保定	158*****	010-11111111	atguigu	

物理结构 物理存储结构即为数据映射关系，而在概念视图的空单元格，底层实际根本不存储。

StoreFile	personal_info			Timestamp		
	Row Key	name	city	phone	不同版本（version）的数据根据timestamp进行区分 读取数据默认读取最新的版本	
	row_key1	张三	北京	131*****		
	row_key11		上海	132*****		
	row_key2	王五	广州			

Row Key	Column Family	Column Qualifier	Timestamp	Type	Value
row_key1	personal_info	name	t1	Put	张三
row_key1	personal_info	city	t2	Put	北京
row_key1	personal_info	phone	t3	Put	131*****
row_key1	personal_info	phone	t4	Put	177*****

数据模型

1) Name Space

命名空间，类似于关系型数据库的 database 概念，每个命名空间下有多个表。HBase 两个自带的命名空间，分别是 hbase 和 default，hbase 中存放的是 HBase 内置的表，default 表是用户默认使用的命名空间。

2) Table

类似于关系型数据库的表概念。不同的是，HBase 定义表时只需要声明列族即可，不需要声明具体的列。因为数据存储是稀疏的，所有往 HBase 写入数据时，字段可以动态、按需指定。因此，和关系型数据库相比，HBase 能够轻松应对字段变更的场景。

3) Row

HBase 表中的每行数据都由一个 RowKey 和多个 Column（列）组成，数据是按照 RowKey 的字典顺序存储的，并且查询数据时只能根据 RowKey 进行检索，所以 RowKey 的设计十分重要。

4) Column

HBase 中的每个列都由 Column Family(列族)和 Column Qualifier (列限定符) 进行限定，例如 info: name, info: age。建表时，只需指明列族，而列限定符无需预先定义。

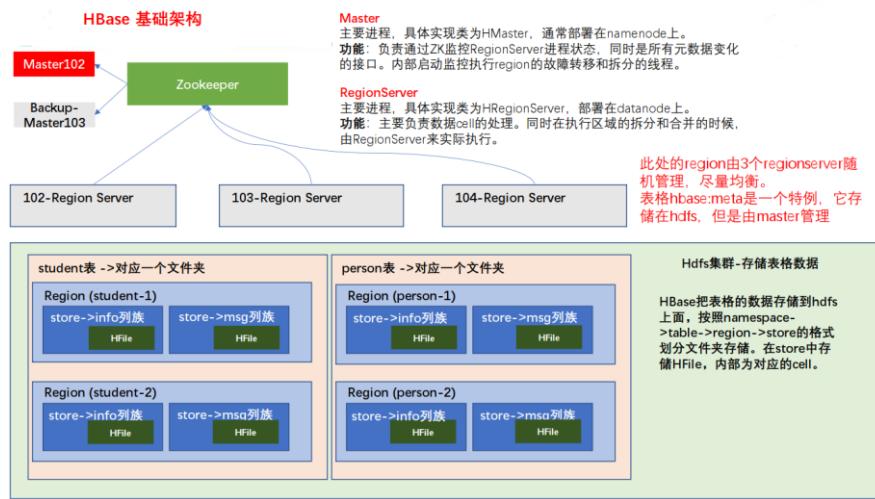
5) Time Stamp

用于标识数据的不同版本 (version)，每条数据写入时，系统会自动为其加上该字段，其值为写入 HBase 的时间。

6) Cell

由{rowkey, column Family: column Qualifier, timestamp} 唯一确定的单元。cell 中的数据全部是字节码形式存储。

4.11.3 Hbase 基本架构



架构角色：

Master

实现类为 HMaster，负责监控集群中所有的 RegionServer 实例。主要作用如下：

- (1) 管理元数据表格 hbase:meta，接收用户对表格创建修改删除的命令并执行
- (2) 监控 region 是否需要进行负载均衡，故障转移和 region 的拆分。通过启动多个后台线程监控实现上述功能：
 - ①LoadBalancer 负载均衡器周期性监控 region 分布在 regionServer 上面是否均衡，由参数 hbase.balancer.period 控制周期时间，默认 5 分钟。
 - ②CatalogJanitor 元数据管理器定期检查和清理 hbase:meta 中的数据。 meta 表内容在进阶中介绍。
 - ③MasterProcWAL master 预写日志处理器把 master 需要执行的任务记录到预写日志 WAL 中，如果 master 宕机，让 backupMaster 读取日志继续干。

Region Server

Region Server 实现类为 HRegionServer，主要作用如下：

- (1) 负责数据 cell 的处理，例如写入数据 put，查询数据 get 等
- (2) 拆分合并 region 的实际执行者，有 master 监控，有 regionServer 执行。

Zookeeper

HBase 通过 Zookeeper 来做 master 的高可用、记录 RegionServer 的部署信息、并且存储有 meta 表的位置信息。HBase 对于数据的读写操作时直接访问 Zookeeper 的，在 2.3 版本推出 Master Registry 模式，客户端可以直接访问 master。使用此功能，会加大对 master 的压力，减轻对 Zookeeper 的压力。

HDFS

HDFS 为 Hbase 提供最终的底层数据存储服务，同时为 HBase 提供高容错的支持。

4.11.4 生产调优

RowKey 设计

一条数据的唯一标识就是 rowkey，那么这条数据存储于哪个分区，取决于 rowkey 处于哪个一个预分区的区间内，设计 rowkey 的主要目的，就是让数据均匀的分布于所有的 region 中，在一定程度上防止数据倾斜。rowkey 常用的设计方案。

- 1) 生成随机数、hash、散列值
- 2) 时间戳反转

3) 字符串拼接

参数优化

1) Zookeeper 会话超时时间

hbase-site.xml

属性: zookeeper.session.timeout

解释: 默认值为 90000 毫秒 (90s)。当某个 RegionServer 挂掉, 90s 之后 Master 才能察觉到。可适当减小此值, 尽可能快地检测 regionserver 故障, 可调整至 20-30s。

看你能有都能忍耐超时, 同时可以调整重试时间和重试次数 hbase.client.pause (默认值 100ms)

hbase.client.retries.number (默认 15 次)

2) 设置 RPC 监听数量

hbase-site.xml

属性: hbase.regionserver.handler.count

解释: 默认值为 30, 用于指定 RPC 监听的数量, 可以根据客户端的请求数进行调整, 读写请求较多时, 增加此值。

3) 手动控制 Major Compaction

hbase-site.xml

属性: hbase.hregion.majorcompaction

解释: 默认值: 604800000 秒 (7 天), Major Compaction 的周期, 若关闭自动 Major Compaction, 可将其设为 0。如果关闭一定记得自己手动合并, 因为大合并非常有意义

4) 优化 HStore 文件大小

hbase-site.xml

属性: hbase.hregion.maxfilesize

解释: 默认值 10737418240 (10GB), 如果需要运行 HBase 的 MR 任务, 可以减小此值, 因为一个 region 对应一个 map 任务, 如果单个 region 过大, 会导致 map 任务执行时间过长。该值的意思就是, 如果 Hfile 的大小达到这个数值, 则这个 region 会被切分为两个 Hfile。

5) 优化 HBase 客户端缓存

hbase-site.xml

属性: hbase.client.write.buffer

解释: 默认值 2097152bytes (2M) 用于指定 HBase 客户端缓存, 增大该值可以减少 RPC 调用次数, 但是会消耗更多内存, 反之则反之。一般我们需要设定一定的缓存大小, 以达到减少 RPC 次数的目的。

6) 指定 scan.next 扫描 HBase 所获取的行数

hbase-site.xml

属性: hbase.client.scanner.caching

解释: 用于指定 scan.next 方法获取的默认行数, 值越大, 消耗内存越大。

7) BlockCache 占用 RegionServer 堆内存的比例

hbase-site.xml

属性: hfile.block.cache.size

解释: 默认 0.4, 读请求比较多的情况下, 可适当调大

8) MemStore 占用 RegionServer 堆内存的比例

hbase-site.xml

属性: hbase.regionserver.global.memstore.size 解释: 默认 0.4, 写请求较多的情况下, 可适当调大 Lars Hofhansl (拉斯·霍夫汉斯) 大神推荐 Region 设置 20G, 刷写大小设置 128M, 其它默认。

HBase 使用经验法则

官方给出了权威的使用法则:

- (1) Region 大小控制 10-50G
- (2) cell 大小不超过 10M (性能对应小于 100K 的值有优化), 如果使用 mob (Mediumsized Objects 一种特殊用法) 则不超过 50M。
- (3) 1 张表有 1 到 3 个列族, 不要设计太多。最好就 1 个, 如果使用多个尽量保证不会同时读取多个列族。
- (4) 1 到 2 个列族的表格, 设计 50-100 个 Region。
- (5) 列族名称要尽量短, 不要去模仿 RDBMS (关系型数据库) 具有准确的名称和描述。
- (6) 如果 RowKey 设计时间在最前面, 会导致有大量的旧数据存储在不活跃的 Region 中, 使用的时候, 仅会操作少数的活动 Region, 此时建议增加更多的 Region 个数。
- (7) 如果只有一个列族用于写入数据, 分配内存资源的时候可以做出调整, 即写缓存不会占用太多的内存。

4.12 Hive



4.12.1 Hive 简介

Hive 是由 Facebook 开源, 基于 Hadoop 的一个数据仓库工具, 可以将结构化的数据文件映射为一张表, 并提供类 SQL 查询功能。

例如: 需求, 统计单词出现个数。

- (1) 在 Hadoop 课程中我们用 MapReduce 程序实现的, 当时需要写 Mapper、Reducer 和 Driver 三个类, 并实现对应逻辑, 相对繁琐。

```
JavaScript
```

```
test 表
```

```
id 列
```

```
myUbuntu
```

```
myUbuntu
```

```
ss
```

```
ss
```

```
jiao
```

```
banzhang
```

```
xue
```

```
hadoop
```

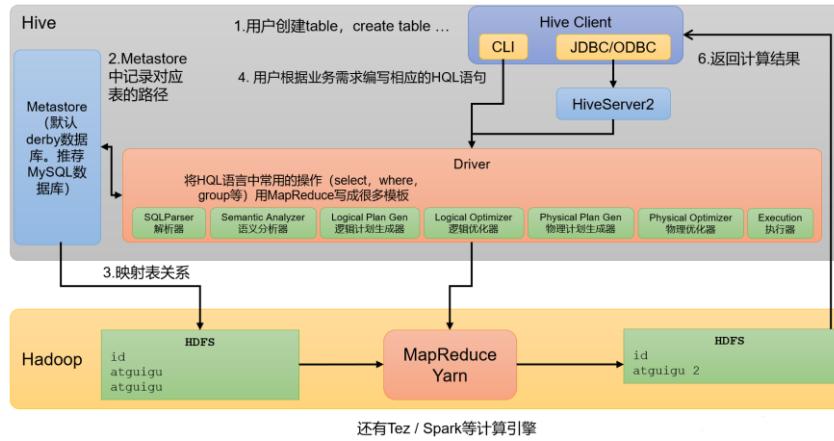
- (2) 如果通过 Hive SQL 实现, 一行就搞定了, 简单方便, 容易理解。select count(*) from test group by id;

Hive 本质

Hive 是一个 Hadoop 客户端, 用于将 HQL (Hive SQL) 转化成 MapReduce 程序。

- (1) Hive 中每张表的数据存储在 HDFS
- (2) Hive 分析数据底层的实现是 MapReduce (也可配置为 Spark 或者 Tez)
- (3) 执行程序运行在 Yarn 上

4.12.2 Hive 架构原理



1) 用户接口: Client

CLI (command-line interface)、JDBC/ODBC。

说明: JDBC 和 ODBC 的区别。

(1) JDBC 的移植性比 ODBC 好; (通常情况下, 安装完 ODBC 驱动程序之后, 还需要经过确定的配置才能够应用。而不相同的配置在不相同数据库服务器之间不能够通用。所以, 安装一次就需要再配置一次。JDBC 只需要选取适当的 JDBC 数据库驱动程序, 就不需要额外的配置。在安装过程中, JDBC 数据库驱动程序会自己完成有关的配置。)

(2) 两者使用的语言不同, JDBC 在 Java 编程时使用, ODBC 一般在 C/C++ 编程时使用。

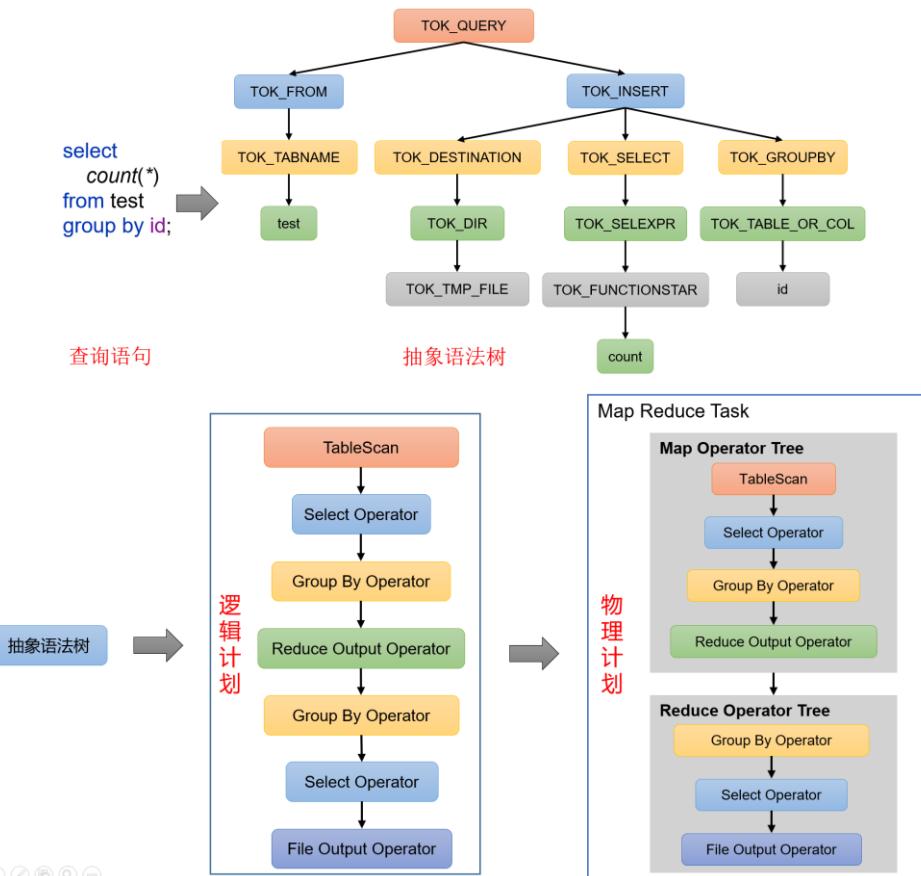
2) 元数据: Metastore

元数据包括: 数据库 (默认是 default)、表名、表的拥有者、列/分区字段、表的类型 (是否是外部表)、表的数据所在目录等。

默认存储在 **自带的 derby 数据库中**, 由于 derby 数据库只支持单客户端访问, 生产环境中为了多人开发, 推荐使用 MySQL 存储 Metastore。

3) 驱动器: Driver

- (1) 解析器 (SQLParser): 将 SQL 字符串转换成抽象语法树 (AST)
- (2) 语义分析 (Semantic Analyzer): 将 AST 进一步划分为 QueryBlock
- (3) 逻辑计划生成器 (Logical Plan Gen): 将语法树生成逻辑计划
- (4) 逻辑优化器 (Logical Optimizer): 对逻辑计划进行优化
- (5) 物理计划生成器 (Physical Plan Gen): 根据优化后的逻辑计划生成物理计划
- (6) 物理优化器 (Physical Optimizer): 对物理计划进行优化
- (7) 执行器 (Execution): 执行该计划, 得到查询结果并返回给客户端



4) Hadoop

使用 HDFS 进行存储，可以选择 MapReduce/Tez/Spark 进行计算。

4.12.3 Hive 生产调优

假设计算环境为 Hive on MR。计算资源的调整主要包括 Yarn 和 MR。

4.12.3.1 Yarn 资源配置

1) Yarn 配置说明

需要调整的 Yarn 参数均与 CPU、内存等资源有关，核心配置参数如下

(1) `yarn.nodemanager.resource.memory-mb`

该参数的含义是，一个 NodeManager 节点分配给 Container 使用的内存。该参数的配置，取决于 NodeManager 所在节点的总内存容量和该节点运行的其他服务的数量。

考虑上述因素，此处可将该参数设置为 64G，如下：

```
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>65536</value>
</property>
```

(2) `yarn.nodemanager.resource.cpu-vcores`

该参数的含义是，一个 NodeManager 节点分配给 Container 使用的 CPU 核数。该参数的配置，同样取决于 NodeManager 所在节点的总 CPU 核数和该节点运行的其他服务。

考虑上述因素，此处可将该参数设置为 16。

```
<property>
```

```
<name>yarn.nodemanager.resource.cpu-vcores</name>
<value>16</value>
</property>
```

(3) yarn.scheduler.maximum-allocation-mb

该参数的含义是，单个 Container 能够使用的最大内存。推荐配置如下：

```
<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>16384</value>
</property>
```

(4) yarn.scheduler.minimum-allocation-mb

该参数的含义是，单个 Container 能够使用的最小内存，推荐配置如下：

```
<property>
  <name>yarn.scheduler.minimum-allocation-mb</name>
  <value>512</value>
</property>
```

2) Yarn 配置实操

(1) 修改\$HADOOP_HOME/etc/hadoop/yarn-site.xml 文件

(2) 修改如下参数

```
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>65536</value>
</property>
<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>16</value>
</property>
<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>16384</value>
</property>
<property>
  <name>yarn.scheduler.minimum-allocation-mb</name>
  <value>512</value>
</property>
```

(3) 分发该配置文件

(4) 重启 Yarn。

4.12.3.2 MapReduce 资源配置

MapReduce 资源配置主要包括 Map Task 的内存和 CPU 核数，以及 Reduce Task 的内存和 CPU 核数。核心配

置参数如下：

1) mapreduce.map.memory.mb

该参数的含义是，单个 Map Task 申请的 container 容器内存大小，其默认值为 1024。该值不能超出 yarn.scheduler.maximum-allocation-mb 和 yarn.scheduler.minimum-allocation-mb 规定的范围。

该参数需要根据不同的计算任务单独进行配置，在 hive 中，可直接使用如下方式为每个 SQL 语句单独进行配置：

```
set mapreduce.map.memory.mb=2048;
```

2) mapreduce.map.cpu.vcores

该参数的含义是，单个 Map Task 申请的 container 容器 cpu 核数，其默认值为 1。该值一般无需调整。

3) mapreduce.reduce.memory.mb

该参数的含义是，单个 Reduce Task 申请的 container 容器内存大小，其默认值为 1024。该值同样不能超出 yarn.scheduler.maximum-allocation-mb 和 yarn.scheduler.minimum-allocation-mb 规定的范围。

该参数需要根据不同的计算任务单独进行配置，在 hive 中，可直接使用如下方式为每个 SQL 语句单独进行配置：

```
set mapreduce.reduce.memory.mb=2048;
```

4) mapreduce.reduce.cpu.vcores

该参数的含义是，单个 Reduce Task 申请的 container 容器 cpu 核数，其默认值为 1。该值一般无需调整。

4.13 Spark



4.13.1 Spark 概述

Spark 是一种由 Scala 语言开发的快速、通用、可扩展的大数据分析引擎

Spark Core 中提供了 Spark 最基础与最核心的功能

Spark SQL 是 Spark 用来操作结构化数据的组件。通过 Spark SQL，用户可以使用 SQL 或者 Apache Hive 版本的 SQL 方言（HQL）来查询数据。

Spark Streaming 是 Spark 平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的 API。

4.13.2 Hadoop Mapreduce 与 Spark 对比

Hadoop 的 MR 框架和 Spark 框架都是数据处理框架

Hadoop MapReduce 由于其设计初衷并不是为了满足循环迭代式数据流处理，因此在多并行运行的数据可复用场景（如：机器学习、图挖掘算法、交互式数据挖掘算法）中存在诸多计算效率等问题。所以 Spark 应运而生，Spark 就是在传统的 MapReduce 计算框架的基础上，利用其计算过程的优化，从而大大加快了数据分析、挖掘的运行和读写速度，并将计算单元缩小到更适合并行计算和重复使用的 RDD 计算模型。

- 机器学习中 ALS、凸优化梯度下降等。这些都需要基于数据集或者数据集的衍生数据反复查询反复操作。MR 这种模式不太合适，即使多 MR 串行处理，性能和时间也是一个问题。数据的共享依赖于磁盘。另外一种是交互式数据挖掘，MR 显然不擅长。而 Spark 所基于的 scala 语言恰恰擅长函数的处理。

- Spark 是一个分布式数据快速分析项目。它的核心技术是弹性分布式数据集（Resilient Distributed Datasets），提供了比 MapReduce 丰富的模型，可以快速在内存中对数据集进行多次迭代，来支持复杂的数据挖掘算法和图形计算算法。
- Spark 和 Hadoop 的根本差异是多个作业之间的数据通信问题：Spark 多个作业之间数据通信是基于内存，而 Hadoop 是基于磁盘。
- Spark Task 的启动时间快。Spark 采用 fork 线程的方式，而 Hadoop 采用创建新的进程的方式。
- Spark 只有在 shuffle 的时候将数据写入磁盘，而 Hadoop 中多个 MR 作业之间的数据交互都要依赖于磁盘交互
- Spark 的缓存机制比 HDFS 的缓存机制高效。

经过上面的比较，我们可以看出在绝大多数的数据计算场景中，Spark 确实会比 MapReduce 更有优势。但是 Spark 是基于内存的，所以在实际的生产环境中，由于内存的限制，可能会由于内存资源不够导致 Job 执行失败，此时，MapReduce 其实是一个更好的选择，所以 Spark 并不能完全替代 MR

4.13.3 Spark 核心模块



➤ Spark Core

Spark Core 中提供了 Spark 最基础与最核心的功能，Spark 其他的功能如：Spark SQL，Spark Streaming，GraphX，MLlib 都是在 Spark Core 的基础上进行扩展的

➤ Spark SQL

Spark SQL 是 Spark 用来操作结构化数据的组件。通过 Spark SQL，用户可以使用 SQL 或者 Apache Hive 版本的 SQL 方言（HQL）来查询数据。

➤ Spark Streaming

Spark Streaming 是 Spark 平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的 API。

➤ Spark MLlib

MLlib 是 Spark 提供的一个机器学习算法库。MLlib 不仅提供了模型评估、数据导入等额外的功能，还提供了一些更底层的机器学习原语。

➤ Spark GraphX

GraphX 是 Spark 面向图计算提供的框架与算法库。

4.13.4 Spark 运行环境

Spark 作为一个数据处理框架和计算引擎，被设计在所有常见的集群环境中运行，在国内工作中主流的环境为 Yarn。以下是一些部署模式的对比：

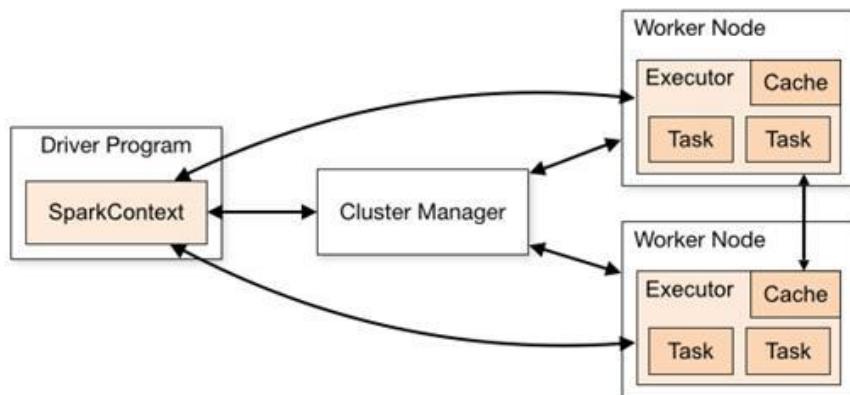
模式	Spark 安装机器数	需启动的进程	所属者	应用场景
Local	1	无	Spark	测试
Standalone	3	Master 及 Worker	Spark	单独部署
Yarn	1	Yarn 及 HDFS	Hadoop	混合部署

独立部署（Standalone）模式由 Spark 自身提供计算资源，无需其他框架提供资源。这种方式降低了和其他第三方资源框架的耦合性，独立性非常强。但是你也要记住，Spark 主要是计算框架，而不是资源调度框架，所以本身提供的资源调度并不是它的强项，所以还是和其他专业的资源调度框架集成会更靠谱一些。在国内工作中，Yarn 使用的非常多。

4.13.5 Spark 运行架构

Spark 框架的核心是一个计算引擎，整体来说，它采用了标准 master-slave 的结构。

如下图所示，它展示了一个 Spark 执行时的基本结构。图形中的 Driver 表示 master，负责管理整个集群中的作业任务调度。图形中的 Executor 则是 slave，负责实际执行任务。



由上图可以看出，对于 Spark 框架有两个核心组件：

Driver

Spark 驱动器节点，用于执行 Spark 任务中的 main 方法，负责实际代码的执行工作。Driver 在 Spark 作业执行时主要负责：

- 将用户程序转化为作业（job）
- 在 Executor 之间调度任务(task)
- 跟踪 Executor 的执行情况
- 通过 UI 展示查询运行情况

实际上，我们无法准确地描述 Driver 的定义，因为在整个的编程过程中没有看到任何有关 Driver 的字眼。所以简单理解，所谓的 Driver 就是驱使整个应用运行起来的程序，也称之为 Driver 类。

Executor

Spark Executor 是集群中工作节点（Worker）中的一个 JVM 进程，负责在 Spark 作业中运行具体任务（Task），任务彼此之间相互独立。Spark 应用启动时，Executor 节点被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有 Executor 节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他 Executor 节点上继续运行。

Executor 有两个核心功能：

- 负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程
- 它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在 Executor 进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

Master & Worker

Spark 集群的独立部署环境中，不需要依赖其他的资源调度框架，自身就实现了资源调度的功能，所以环境中还有其他两个核心组件：Master 和 Worker，这里的 Master 是一个进程，主要负责资源的调度和分配，并进行集群的监控等职责，类似于 Yarn 环境中的 RM，而 Worker 呢，也是进程，一个 Worker 运行在集群中的一台服务器上，由 Master 分配资源对数据进行并行的处理和计算，类似于 Yarn 环境中 NM。

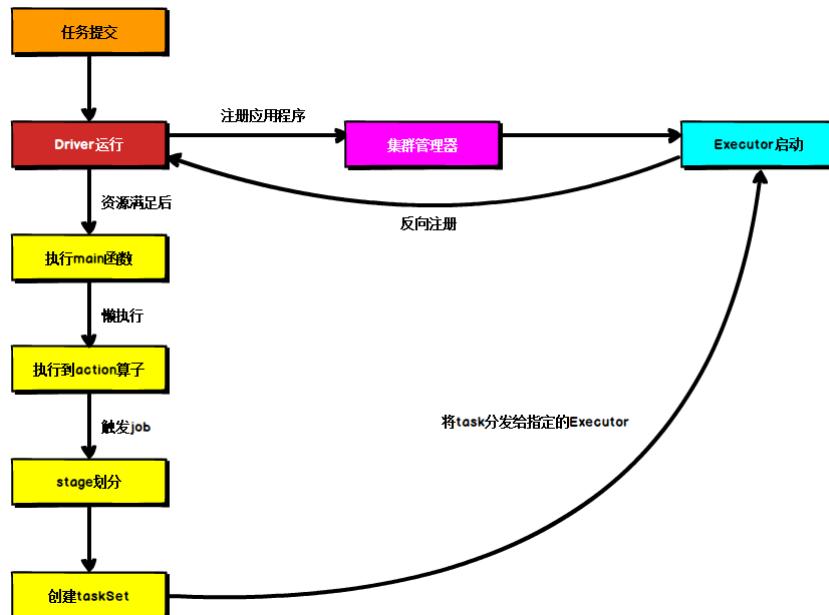
ApplicationMaster

Hadoop 用户向 YARN 集群提交应用程序时,提交程序中应该包含 ApplicationMaster, 用于向资源调度器申请执行任务的资源容器 Container, 运行用户自己的程序任务 job, 监控整个任务的执行, 跟踪整个任务的状态, 处理任务失败等异常情况。

说的简单点就是, ResourceManager (资源) 和 Driver (计算) 之间的解耦合靠的就是 ApplicationMaster。

4.13.6 提交流程

所谓的提交流程, 其实就是我们开发人员根据需求写的应用程序通过 Spark 客户端提交给 Spark 运行环境执行计算的流程。在不同的部署环境中, 这个提交过程基本相同, 但是又有细微的区别, 我们这里不进行详细的比较, 但是因为国内工作中, 将 Spark 引用部署到 Yarn 环境中会更多一些, 所以本课程中的提交流程是基于 Yarn 环境的。



Spark 应用程序提交到 Yarn 环境中执行的时候, 一般会有两种部署执行的方式: Client 和 Cluster。两种模式主要区别在于: **Driver 程序的运行节点位置**。

Yarn Client 模式

Client 模式将用于监控和调度的 Driver 模块在客户端执行, 而不是在 Yarn 中, 所以一般用于测试。

- Driver 在任务提交的本地机器上运行
- Driver 启动后会和 ResourceManager 通讯申请启动 ApplicationMaster
- ResourceManager 分配 container, 在合适的 NodeManager 上启动 ApplicationMaster, 负责向 ResourceManager 申请 Executor 内存
- ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container, 然后 ApplicationMaster 在资源分配指定的 NodeManager 上启动 Executor 进程
- Executor 进程启动后会向 Driver 反向注册, Executor 全部注册完成后 Driver 开始执行 main 函数
- 之后执行到 Action 算子时, 触发一个 Job, 并根据宽依赖开始划分 stage, 每个 stage 生成对应的 TaskSet, 之后将 task 分发到各个 Executor 上执行。

Yarn Cluster 模式

Cluster 模式将用于监控和调度的 Driver 模块启动在 Yarn 集群资源中执行。一般应用于实际生产环境。

- 在 YARN Cluster 模式下, 任务提交后会和 ResourceManager 通讯申请启动 ApplicationMaster,
- 随后 ResourceManager 分配 container, 在合适的 NodeManager 上启动 ApplicationMaster, 此时的

ApplicationMaster 就是 Driver。

- Driver 启动后向 ResourceManager 申请 Executor 内存， ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container，然后在合适的 NodeManager 上启动 Executor 进程
- Executor 进程启动后会向 Driver 反向注册， Executor 全部注册完成后 Driver 开始执行 main 函数，
- 之后执行到 Action 算子时，触发一个 Job，并根据宽依赖开始划分 stage，每个 stage 生成对应的 TaskSet，之后将 task 分发到各个 Executor 上执行。

4.13.7 生产调优

最优资源配置

可以进行分配的资源如表所示：

名称	说明
--num-executors	配置 Executor 的数量
--driver-memory	配置 Driver 内存（影响不大）
--executor-memory	配置每个 Executor 的内存大小
--executor-cores	配置每个 Executor 的 CPU core 数量

调节原则：尽量将任务分配的资源调节到可以使用的资源的最大限度。对于具体资源的分配，我们分别讨论 Spark 的两种 Cluster 运行模式：

- 第一种是 Spark Standalone 模式，你在提交任务前，一定知道或者可以从运维部门获取到你可以使用的资源情况，在编写 submit 脚本的时候，就根据可用的资源情况进行资源的分配，比如说集群有 15 台机器，每台机器为 8G 内存， 2 个 CPU core，那么就指定 15 个 Executor，每个 Executor 分配 8G 内存， 2 个 CPU core。
- 第二种是 Spark Yarn 模式，由于 Yarn 使用资源队列进行资源的分配和调度，在编写 submit 脚本的时候，就根据 Spark 作业要提交到的资源队列，进行资源的分配，比如资源队列有 400G 内存， 100 个 CPU core，那么指定 50 个 Executor，每个 Executor 分配 8G 内存， 2 个 CPU core。

对各项资源进行了调节后，得到的性能提升会有如下表现：

名称	解析
增加 Executor 个数	在资源允许的情况下,增加 Executor 的个数可以提高执行 task 的并行度。比如有 4 个 Executor, 每个 Executor 有 2 个 CPU core, 那么可以并行执行 8 个 task, 如果将 Executor 的个数增加到 8 个(资源允许的情况下), 那么可以并行执行 16 个 task, 此时的并行能力提升了一倍。
增加每个 Executor 的 CPU core 个数	在资源允许的情况下,增加每个 Executor 的 Cpu core 个数, 可以提高执行 task 的并行度。比如有 4 个 Executor, 每个 Executor 有 2 个 CPU core, 那么可以并行执行 8 个 task, 如果将每个 Executor 的 CPU core 个数增加到 4 个(资源允许的情况下), 那么可以并行执行 16 个 task, 此时的并行能力提升了一倍。
增加每个 Executor 的内存量	在资源允许的情况下,增加每个 Executor 的内存量以后, 对性能的提升有三点: <ol style="list-style-type: none"> 1. 可以缓存更多的数据(即对 RDD 进行 cache), 写入磁盘的数据相应减少, 甚至可以不写入磁盘, 减少了可能的磁盘 IO; 2. 可以为 shuffle 操作提供更多内存, 即有更多空间来存放 reduce 端拉取的数据, 写入磁盘的数据相应减少, 甚至可以不写入磁盘, 减少了可能的磁盘 IO; 3. 可以为 task 的执行提供更多内存, 在 task 的执行过程中可能创建很多对象, 内存较少时会引发频繁的 GC, 增加内存后, 可以避免频繁的 GC, 提升整体性能。

RDD 优化

1) RDD 复用

在对 RDD 进行算子时, 要避免相同的算子和计算逻辑之下对 RDD 进行重复的计算

2) RDD 持久化

在 Spark 中, 当多次对同一个 RDD 执行算子操作时, 每一次都会对这个 RDD 以前的父 RDD 重新计算一次, 这种情况是必须要避免的, 对同一个 RDD 的重复计算是对资源的极大浪费, 因此, 必须对多次使用的 RDD 进行持久化, 通过持久化将公共 RDD 的数据缓存到内存/磁盘中, 之后对于公共 RDD 的计算都会从内存/磁盘中直接获取 RDD 数据。

3) RDD 尽可能早的 filter 操作

获取到初始 RDD 后, 应该考虑尽早地过滤掉不需要的数据, 进而减少对内存的占用, 从而提升 Spark 作业的运行效率。

并行度调节

Spark 作业中的并行度指各个 stage 的 task 的数量。

如果并行度设置不合理而导致并行度过低, 会导致资源的极大浪费, 例如, 20 个 Executor, 每个 Executor 分

配 3 个 CPU core，而 Spark 作业有 40 个 task，这样每个 Executor 分配到的 task 个数是 2 个，这就使得每个 Executor 有一个 CPU core 空闲，导致资源的浪费。

理想的并行度设置，应该是让并行度与资源相匹配，简单来说就是在资源允许的前提下，并行度要设置的尽可能大，达到可以充分利用集群资源。合理的设置并行度，可以提升整个 Spark 作业的性能和运行速度。

Spark 官方推荐，task 数量应该设置为 Spark 作业总 CPU core 数量的 2~3 倍。

广播大变量

默认情况下，task 中的算子中如果使用了外部的变量，每个 task 都会获取一份变量的复本，这就造成了内存的极大消耗。一方面，如果后续对 RDD 进行持久化，可能就无法将 RDD 数据存入内存，只能写入磁盘，磁盘 IO 将会严重消耗性能；另一方面，task 在创建对象的时候，也许会发现堆内存无法存放新创建的对象，这就会导致频繁的 GC，GC 会导致工作线程停止，进而导致 Spark 暂停工作一段时间，严重影响 Spark 性能。

Kryo 序列化

默认情况下，Spark 使用 Java 的序列化机制。Java 的序列化机制使用方便，不需要额外的配置，在算子中使用的变量实现 Serializable 接口即可，但是，Java 序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。

Kryo 序列化机制比 Java 序列化机制性能提高 10 倍左右，Spark 之所以没有默认使用 Kryo 作为序列化类库，是因为它不支持所有对象的序列化，同时 Kryo 需要用户在使用前注册需要序列化的类型，不够方便，但从 Spark 2.0.0 版本开始，**简单类型、简单类型数组、字符串类型的 Shuffling RDDs 已经默认使用 Kryo 序列化方式了。**

4.14 Hadoop 生产调优

4.14.1 HDFS—核心参数

4.14.1.1 NameNode 内存生产配置

1) NameNode 内存计算

每个文件块大概占用 150byte，一台服务器 128G 内存为例，能存储的文件块数大约有：

$$128 * 1024 * 1024 * 1024 / 150\text{Byte} \approx 9.1 \text{ 亿}$$

2) Hadoop2.x 系列，配置 NameNode 内存

NameNode 内存默认 2000m，如果服务器内存 4G，NameNode 内存可以配置 3g。

在 hadoop-env.sh 文件中配置如下。

Shell

```
HADOOP_NAMENODE_OPTS=-Xmx3072m
```

3) Hadoop3.x 系列，配置 NameNode 内存

(1) hadoop-env.sh 中描述 Hadoop 的内存是动态分配的

Shell

```
# The maximum amount of heap to use (Java -Xmx). If no unit
# is provided, it will be converted to MB. Daemons will
# prefer any Xmx setting in their respective _OPT variable.
# There is no default; the JVM will autoscale based upon machine
# memory size.
# export HADOOP_HEAPSIZE_MAX=
# The minimum amount of heap to use (Java -Xms). If no unit
# is provided, it will be converted to MB. Daemons will
# prefer any Xms setting in their respective _OPT variable.
```

```
# There is no default; the JVM will autoscale based upon machine
# memory size.
# export HADOOP_HEAPSIZE_MIN=
HADOOP_NAMENODE_OPTS=-Xmx102400m
```

(2) 查看 NameNode 占用内存

```
Shell
[myUbuntu@hadoop102 ~]$ jps
3088 NodeManager
2611 NameNode
3271 JobHistoryServer
2744 DataNode
MaxHeapSize = 1031798784 (984.0MB)
```

查看发现 hadoop102 上的 NameNode 和 DataNode 占用内存都是自动分配的，且相等。并不是很合理。

NameNode 和 DataNode 的内存占用应该根据集群规模和使用情况进行调整，并不一定相等。如果 NameNode 和 DataNode 的内存占用相等且自动分配，可能会出现以下不合理之处：

1. 性能瓶颈：NameNode 和 DataNode 所需要的内存大小是不同的，如果它们占用的内存相等，可能会导致性能瓶颈和资源浪费。
2. 系统不稳定：如果 NameNode 和 DataNode 的内存占用相等，并且都是自动分配的，可能会导致系统不稳定，因为它们需要的内存大小是根据不同的任务和使用情况来确定的。
3. 容易导致故障：如果 NameNode 和 DataNode 的内存占用相等，并且都是自动分配的，可能会导致某个节点因为内存不足而宕机，从而导致数据丢失或者系统崩溃。

因此，合理的做法是根据集群规模和使用情况来调整 NameNode 和 DataNode 所需要的内存大小，以确保系统的性能和稳定性。

查阅相关文档，得到的经验性参考为：

<p>NameNode</p> <p>namenode最小值 1G, 每增加1000000个block, 增加1G内存</p> <ul style="list-style-type: none"> Minimum: 1 GB (for proof-of-concept deployments) Add an additional 1 GB for each additional 1,000,000 blocks Snapshots and encryption can increase the required heap memory. <p>See Sizing NameNode Heap Memory</p> <p>Set this value using the Java Heap Size of NameNode in Bytes HDFS configuration property.</p>	<p>DataNode</p> <p>datanode最小值 4G, block数, 或者副本数升高, 都应该调大datanode的值。</p> <p>一个datanode上的副本总数低于4000000, 调为4G, 超过4000000, 每增加1000000, 增加1G</p> <p>Minimum: 4 GB</p> <p>Increase the memory for higher replica counts or a higher number of blocks per DataNode. When increasing the memory, Cloudera recommends an additional 1 GB of memory for every 1 million replicas above 4 million on the DataNodes. For example, 5 million replicas require 5 GB of memory.</p> <p>Set this value using the Java Heap Size of DataNode in Bytes HDFS configuration property.</p>
--	---

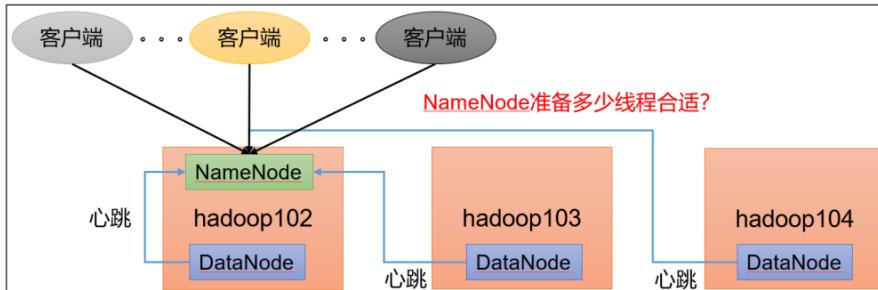
对 hadoop-env.sh 作如下修改：

```

Shell
export HDFS_NAMENODE_OPTS="-Dhadoop.security.logger=INFO,RFAS -Xmx1024m"
export HDFS_DATANODE_OPTS="-Dhadoop.security.logger=ERROR,RFAS -Xmx1024m"

```

4.14.1.2 NameNode 心跳并发配置



1) hdfs-site.xml

XML

The number of Namenode RPC server threads that listen to requests from clients. If `dfs.namenode.servicerpc-address` is not configured then Namenode RPC server threads listen to requests from all nodes.

NameNode 有一个工作线程池，用来处理不同 DataNode 的并发心跳以及客户端并发的元数据操作。

对于大集群或者有大量客户端的集群来说，通常需要增大该参数。默认值是 10。

```

<property>
<name>dfs.namenode.handler.count</name>
<value>21</value>
</property>

```

根据企业经验： $\text{dfs.namenode.handler.count} = 20 \times \log_{\text{Cluster Size}}$ ，比如集群规模（DataNode 台数）为 3 台时，此参数设置为 21。可通过简单的 python 代码计算该值，代码如下：

```

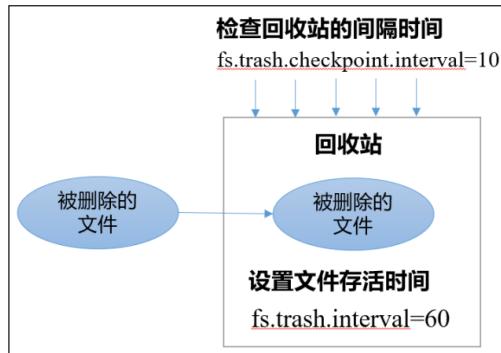
Shell
[myUbuntu@hadoop102 ~]$ sudo yum install -y python
[myUbuntu@hadoop102 ~]$ python
Python 2.7.5 (default, Apr 11 2018, 07:36:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import math
>>> print int(20*math.log(3))
21
>>> quit()

```

4.14.1.3 开启回收站配置

开启回收站功能，可以将删除的文件在不超时的情况下，恢复原数据，起到防止误删除、备份等作用。

1) 回收站工作机制



2) 开启回收站功能参数说明

- (1) 默认值 `fs.trash.interval = 0`, 0 表示禁用回收站；其他值表示设置文件的存活时间。
- (2) 默认值 `fs.trash.checkpoint.interval = 0`, 检查回收站的间隔时间。如果该值为 0，则该值设置和 `fs.trash.interval` 的参数值相等。
- (3) 要求 `fs.trash.checkpoint.interval <= fs.trash.interval`。

3) 启用回收站

修改 `core-site.xml`, 配置垃圾回收时间为 1 分钟。

XML

```
<property>
<name>fs.trash.interval</name>
<value>1</value>
</property>
```

4) 查看回收站

回收站目录在 HDFS 集群中的路径： /user/myUbuntu/.Trash/....

- 5) 注意：通过网页上直接删除的文件也不会走回收站。
- 6) 通过程序删除的文件不会经过回收站，需要调用 `moveToTrash()` 才进入回收站

Shell

```
Trash trash = New Trash(conf);
trash.moveToTrash(path);
```

- 7) 只有在命令行利用 `hadoop fs -rm` 命令删除的文件才会走回收站。

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -rm -r/user/myUbuntu/input
2021-07-14 16:13:42,643 INFO fs.TrashPolicyDefault:
Moved:'hdfs://hadoop102:9820/user/myUbuntu/input' to trash at:
hdfs://hadoop102:9820/user/myUbuntu/.Trash/Current/user/myUbuntu/input
```

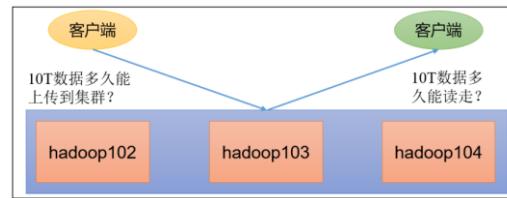
8) 恢复回收站数据

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -mv
/user/myUbuntu/.Trash/Current/user/myUbuntu/input/user/myUbuntu/input
```

4.14.2 HDFS—集群压测

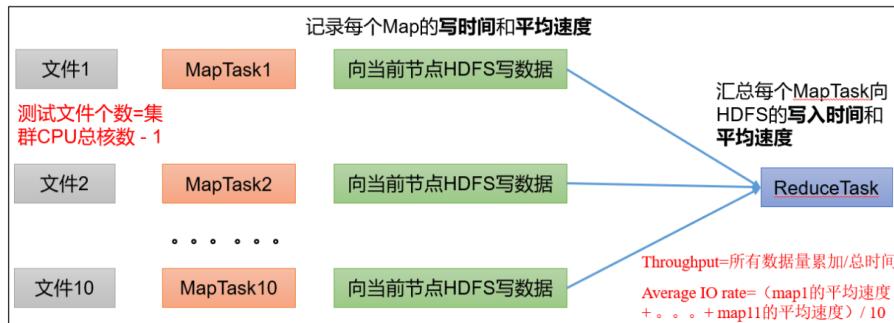
在企业中非常关心每天从 Java 后台拉取过来的数据，需要多久能上传到集群？消费者关心多久能从 HDFS 上拉取需要的数据？为了搞清楚 HDFS 的读写性能，生产环境上非常需要对集群进行压测。



HDFS 的读写性能主要受网络和磁盘影响比较大。为了方便测试，将 hadoop102、hadoop103、hadoop104 虚拟机网络都设置为 100mbps。

4.14.2.1 测试 HDFS 写性能

0) 写测试底层原理



1) 测试内容：向 HDFS 集群写 10 个 128M 的文件

```
Shell
[myUbuntu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-clientjobclient-3.1.3-tests.jar
TestDFSIO -write -nrFiles 10 -
fileSize 128MB
2021-02-09 10:43:16,853 INFO fs.TestDFSIO: ----- TestDFSIO ----- : write
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Date & time: Tue Feb
09 10:43:16 CST 2021
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Number of files: 10
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Total MBytes processed: 1280
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Throughput mb/sec: 1.61
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Average IO rate mb/sec: 1.9
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: IO rate std deviation: 0.76
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Test exec time sec: 133.05
2021-02-09 10:43:16,854 INFO fs.TestDFSIO:
```

- Number of files: 生成 mapTask 数量，一般是集群中 (CPU 核数-1)，我们测试虚拟机就按照实际的物理内存-1 分配即可
- Total MBytes processed: 单个 map 处理的文件大小
- Throughput mb/sec: 单个 mapTak 的吞吐量
计算方式: 处理的总文件大小/每一个 mapTask 写数据的时间累加
- 集群整体吞吐量: 生成 mapTask 数量*单个 mapTak 的吞吐量
- Average IO rate mb/sec:: 平均 mapTak 的吞吐量
计算方式: 每个 mapTask 处理文件大小/每一个 mapTask 写数据的时间全部相加除以 task 数量
- IO rate std deviation: 方差、反映各个 mapTask 处理的差值，越小越均衡

3) 测试结果分析

- (1) 由于副本 1 就在本地，所以该副本不参与测试



一共参与测试的文件: 10 个文件 * 2 个副本 = 20 个

压测后的速度: 1.61

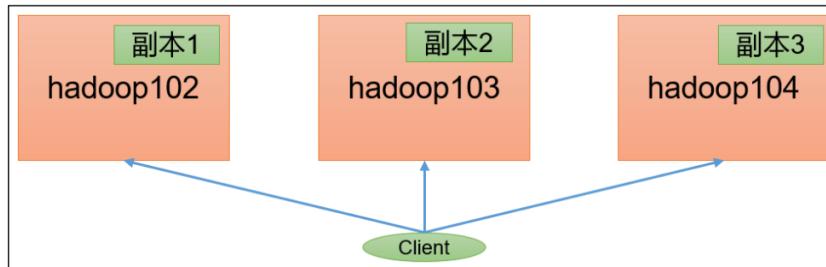
实测速度: 1.61M/s * 20 个文件 \approx 32M/s

三台服务器的带宽: $12.5 + 12.5 + 12.5 \approx 30\text{m/s}$

所有网络资源都已经用满。

如果实测速度远远小于网络，并且实测速度不能满足工作需求，可以考虑采用固态硬盘或者增加磁盘个数。

(2) 如果客户端不在集群节点，那就三个副本都参与计算



4.14.2.2 测试 HDFS 读性能

1) 测试内容: 读取 HDFS 集群 10 个 128M 的文件

Shell

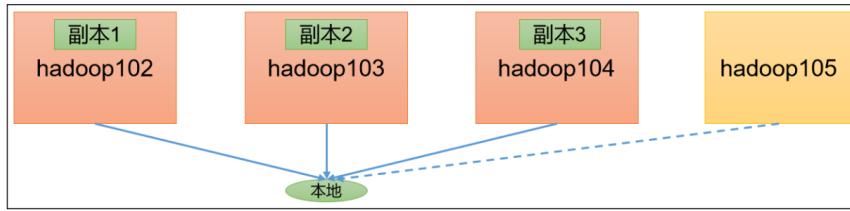
```
[myUbuntu@hadoop102 mapreduce]$ hadoop jar
/opt/module/hadoop3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-clientjobclient-
3.1.3-tests.jar TestDFSIO -read -nrFiles 10 -fileSize
128MB
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: Date & time: Tue Feb
09 11:34:15 CST 2021
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: Number of files: 10
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: Total MBytes processed: 1280
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: Throughput mb/sec: 200.28
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: Average IO rate mb/sec: 266.74
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: IO rate std deviation: 143.12
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: Test exec time sec: 20.83
```

2) 删除测试生成数据

Shell

```
[myUbuntu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-clientjobclient-3.1.3-tests.jar
TestDFSIO -clean
```

3) 测试结果分析: 为什么读取文件速度大于网络带宽——由于目前只有三台服务器，且有三个副本，数据读取就近原则，相当于都是读取的本地磁盘数据，没有走网络



4.14.3 HDFS—多目录

4.14.3.1 NameNode 多目录配置

- 1) NameNode 的本地目录可以配置成多个，且每个目录存放内容相同，增加了可靠性



- 2) 具体配置如下

- (1) 在 hdfs-site.xml 文件中添加如下内容

XML

```
<property>
<name>dfs.namenode.name.dir</name>
<value>file://${hadoop.tmp.dir}/dfs/name1,file://${hadoop.tmp.dir}/dfs/name2</value>
</property>
```

- (2) 停止集群，删除三台节点的 data 和 logs 中所有数据。

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ rm -rf data/ logs/
[myUbuntu@hadoop103 hadoop-3.1.3]$ rm -rf data/ logs/
[myUbuntu@hadoop104 hadoop-3.1.3]$ rm -rf data/ logs/
```

- (3) 格式化集群并启动。

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ bin/hdfs namenode -format
[myUbuntu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
```

- 3) 查看结果

Shell

```
[myUbuntu@hadoop102 dfs]$ ll
总用量 12
drwx----- 3 myUbuntu myUbuntu 4096 12 月 11 08:03 data
drwxrwxr-x 3 myUbuntu myUbuntu 4096 12 月 11 08:03 name1
drwxrwxr-x 3 myUbuntu myUbuntu 4096 12 月 11 08:03 name2
```

检查 name1 和 name2 里面的内容，发现一模一样。

4.14.3.2 DataNode 多目录配置

1) DataNode 可以配置成多个目录，每个目录存储的数据不一样（数据不是副本）



2) 具体配置如下

在 hdfs-site.xml 文件中添加如下内容

XML

```
<property>
<name>dfs.datanode.data.dir</name>
<value>file://${hadoop.tmp.dir}/dfs/data1,file://${hadoop.tmp.dir}/dfs/data2</value>
</property>
```

3) 查看结果

Shell

```
[myUbuntu@hadoop102 dfs]$ ll
总用量 12
drwx----- 3 myUbuntu myUbuntu 4096 4 月 4 14:22 data1
drwx----- 3 myUbuntu myUbuntu 4096 4 月 4 14:22 data2
drwxrwxr-x 3 myUbuntu myUbuntu 4096 12 月 11 08:03 name1
drwxrwxr-x 3 myUbuntu myUbuntu 4096 12 月 11 08:03 name2
```

4) 向集群上传一个文件，再次观察两个文件夹里面的内容发现不一致（一个有数一个没有）

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -put
wcinput/word.txt /
```

4.14.3.3 集群数据均衡之磁盘间数据均衡

生产环境，由于硬盘空间不足，往往需要增加一块硬盘。刚加载的硬盘没有数据时，可以执行磁盘数据均衡命令。



(1) 生成均衡计划（我们只有一块磁盘，不会生成计划）

Shell

```
hdfs diskbalancer -plan hadoop103
```

(2) 执行均衡计划

Shell

```
hdfs diskbalancer -execute hadoop103.plan.json
```

(3) 查看当前均衡任务的执行情况

```
Shell  
hdfs diskbalancer -query hadoop103
```

(4) 取消均衡任务

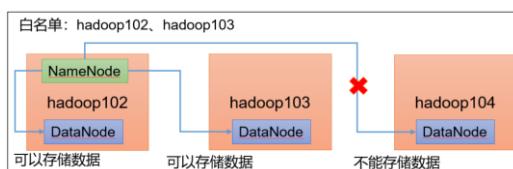
```
Shell  
hdfs diskbalancer -cancel hadoop103.plan.json
```

4.14.4 HDFS—集群扩容及缩容

4.14.4.1 添加白名单

白名单：表示在白名单的主机 IP 地址可以，用来存储数据。

企业中：配置白名单，可以尽量防止黑客恶意访问攻击。



配置白名单步骤如下：

- 1) 在 NameNode 节点的 /opt/module/hadoop-3.1.3/etc/hadoop 目录下分别创建 whitelist 和 blacklist 文件
- (1) 创建白名单

```
Shell  
[myUbuntu@hadoop102 hadoop]$ vim whitelist
```

在 whitelist 中添加如下主机名称，假如集群正常工作的节点为 102 103

```
Shell  
hadoop102  
hadoop103
```

(2) 创建黑名单

```
Shell  
[myUbuntu@hadoop102 hadoop]$ touch blacklist
```

保持空的就可以

- 2) 在 hdfs-site.xml 配置文件中增加 dfs.hosts 配置参数

```
XML  
<!-- 白名单 -->  
<property>  
<name>dfs.hosts</name>  
<value>/opt/module/hadoop-3.1.3/etc/hadoop/whitelist</value>  
</property>  
<!-- 黑名单 -->  
<property>  
<name>dfs.hosts.exclude</name>  
<value>/opt/module/hadoop-3.1.3/etc/hadoop/blacklist</value>  
</property>
```

- 3) 分发配置文件 whitelist, hdfs-site.xml

Shell

```
[myUbuntu@hadoop104 hadoop]$ xsync hdfs-site.xml whitelist
```

4) 第一次添加白名单必须重启集群，不是第一次，只需要刷新 NameNode 节点即可

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ myhadoop.sh stop  
[myUbuntu@hadoop102 hadoop-3.1.3]$ myhadoop.sh start
```

5) 在 web 浏览器上查看 DN， <http://hadoop102:9870/dfshealth.html#tab-datanode>

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓ hadoop102:9866 (192.168.10.102:9866)	http://hadoop102:9864	0s	2m	89.95 GB	3	61.53 KB (0%)	3.1.3
✓ hadoop103:9866 (192.168.10.103:9866)	http://hadoop103:9864	2s	2m	89.95 GB	3	61.53 KB (0%)	3.1.3

6) 在 hadoop104 上执行上传数据数据失败

Shell

```
[myUbuntu@hadoop104 hadoop-3.1.3]$ hadoop fs -put NOTICE.txt /
```

7) 二次修改白名单，增加 hadoop104

Shell

```
[myUbuntu@hadoop102 hadoop]$ vim whitelist
```

修改为如下内容

```
hadoop102  
hadoop103  
hadoop104
```

8) 刷新 NameNode

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -refreshNodes  
Refresh nodes successful
```

9) 在 web 浏览器上查看 DN， <http://hadoop102:9870/dfshealth.html#tab-datanode>

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓ hadoop102:9866 (192.168.10.102:9866)	http://hadoop102:9864	1s	0m	89.95 GB	2	40 KB (0%)	3.1.3
✓ hadoop103:9866 (192.168.10.103:9866)	http://hadoop103:9864	0s	0m	89.95 GB	2	40 KB (0%)	3.1.3
✓ hadoop104:9866 (192.168.10.104:9866)	http://hadoop104:9864	0s	0m	89.95 GB	2	40 KB (0%)	3.1.3

4.14.4.2 服役新服务器

1) 需求

随着公司业务的增长，数据量越来越大，原有的数据节点的容量已经不能满足存储数据的需求，需要在原有集群基础上动态添加新的数据节点。

2) 环境准备

(1) 在 hadoop100 主机上再克隆一台 hadoop105 主机

(2) 修改 IP 地址和主机名称

Shell

```
[root@hadoop105 ~]# vim /etc/sysconfig/network-scripts/ifcfgens33
```

```
[root@hadoop105 ~]# vim /etc/hostname
```

(3) 拷贝 hadoop102 的/opt/module 目录和/etc/profile.d/my_env.sh 到 hadoop105

Shell

```
[myUbuntu@hadoop102 opt]$ scp -r module/*
```

```
myUbuntu@hadoop105:/opt/module/
```

```
[myUbuntu@hadoop102 opt]$ sudo scp /etc/profile.d/my_env.sh
```

```
root@hadoop105:/etc/profile.d/my_env.sh
```

```
[myUbuntu@hadoop105 hadoop-3.1.3]$ source /etc/profile
```

(4) 删除 hadoop105 上 Hadoop 的历史数据， data 和 log 数据

Shell

```
[myUbuntu@hadoop105 hadoop-3.1.3]$ rm -rf data/ logs/
```

(5) 配置 hadoop102 和 hadoop103 到 hadoop105 的 ssh 无密登录

Shell

```
[myUbuntu@hadoop102 .ssh]$ ssh-copy-id hadoop105
```

```
[myUbuntu@hadoop103 .ssh]$ ssh-copy-id hadoop105
```

3) 服役新节点具体步骤

(1) 直接启动 DataNode，即可关联到集群

Shell

```
[myUbuntu@hadoop105 hadoop-3.1.3]$ hdfs --daemon start datanode
```

```
[myUbuntu@hadoop105 hadoop-3.1.3]$ yarn --daemon start
```

```
nodemanager
```

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
hadoop102:9866 (192.168.10.102:9866)	http://hadoop102:9864	2s	1m	89.95 GB	3	68 KB (0%)	3.1.3
hadoop103:9866 (192.168.10.103:9866)	http://hadoop103:9864	0s	33m	89.95 GB	3	68 KB (0%)	3.1.3
hadoop104:9866 (192.168.10.104:9866)	http://hadoop104:9864	0s	23m	89.95 GB	3	68 KB (0%)	3.1.3
hadoop105:9866 (192.168.10.105:9866)	http://hadoop105:9864	2s	0m	89.95 GB	0	8 KB (0%)	3.1.3

4) 在白名单中增加新服役的服务器

(1) 在白名单 whitelist 中增加 hadoop104、hadoop105，并重启集群

Shell

```
[myUbuntu@hadoop102 hadoop]$ vim whitelist
```

修改为如下内容

Shell

```
hadoop102  
hadoop103  
hadoop104  
hadoop105
```

(2) 分发

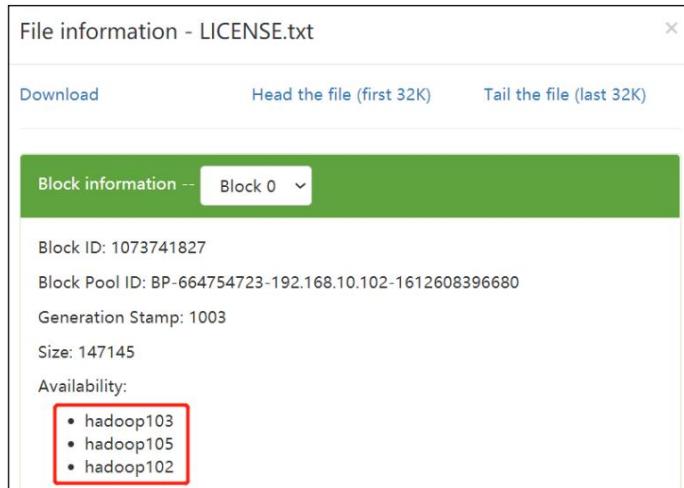
```
Shell  
[myUbuntu@hadoop102 hadoop]$ xsync whitelist
```

(3) 刷新 NameNode

```
Shell  
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -refreshNodes  
Refresh nodes successfu
```

5) 在 hadoop105 上上传文件

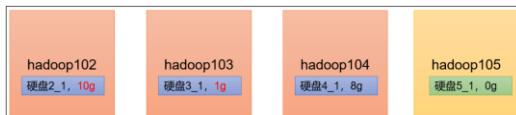
```
Shell  
[myUbuntu@hadoop105 hadoop-3.1.3]$ hadoop fs -put /opt/module/hadoop-  
3.1.3/LICENSE.txt /
```



4.14.4.3 服务器间数据均衡

1) 企业经验:

在企业开发中，如果经常在 hadoop102 和 hadoop104 上提交任务，且副本数为 2，由于数据本地性原则，就会导致 hadoop102 和 hadoop104 数据过多，hadoop103 存储的数据量小。另一种情况，就是新服役的服务器数据量比较少，需要执行集群均衡命令。



2) 开启数据均衡命令:

```
[myUbuntu@hadoop105 hadoop-3.1.3]$ sbin/start-balancer.sh -  
threshold 10
```

对于参数 10，代表的是集群中各个节点的磁盘空间利用率相差不超过 10%，可根据实际情况进行调整。

3) 停止数据均衡命令:

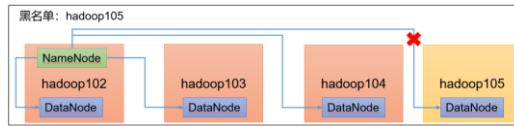
```
[myUbuntu@hadoop105 hadoop-3.1.3]$ sbin/stop-balancer.sh
```

注意: 由于 HDFS 需要启动单独的 Rebalance Server 来执行 Rebalance 操作, 所以尽量不要在 NameNode 上执行 start-balancer.sh, 而是找一台比较空闲的机器。

4.14.4.4 黑名单退役服务器

黑名单: 表示在黑名单的主机 IP 地址不可以, 用来存储数据。

企业中: 配置黑名单, 用来退役服务器。



黑名单配置步骤如下:

1) 编辑 /opt/module/hadoop-3.1.3/etc/hadoop 目录下的 blacklist 文件

Shell

```
[myUbuntu@hadoop102 hadoop] vim blacklist
```

添加如下主机名称 (要退役的节点)

Shell

```
hadoop105
```

注意: 如果白名单中没有配置, 需要在 hdfs-site.xml 配置文件中增加 dfs.hosts 配置参数

Shell

```
<!-- 黑名单 -->
<property>
<name>dfs.hosts.exclude</name>
<value>/opt/module/hadoop-3.1.3/etc/hadoop/blacklist</value>
</property>
```

2) 分发配置文件 blacklist, hdfs-site.xml

Shell

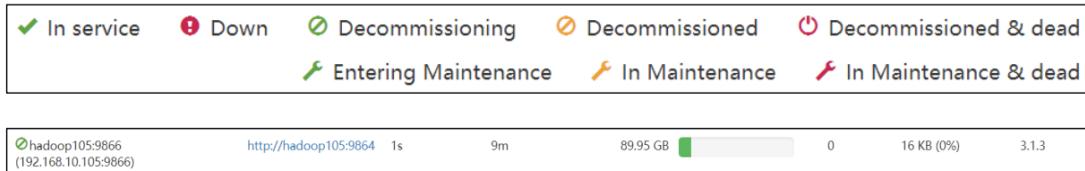
```
[myUbuntu@hadoop104 hadoop]$ xsync hdfs-site.xml blacklist
```

3) 第一次添加黑名单必须重启集群, 不是第一次, 只需要刷新 NameNode 节点即可

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
```

4) 检查 Web 浏览器, 退役节点的状态为 decommission in progress (退役中), 说明数据节点正在复制块到其他节点



5) 等待退役节点状态为 decommissioned (所有块已经复制完成), 停止该节点及节点资源管理器。注意: 如果副本数是 3, 服役的节点小于等于 3, 是不能退役成功的, 需要修改

副本数后才能退役



Shell

```
[myUbuntu@hadoop105 hadoop-3.1.3]$ hdfs --daemon stop datanode  
stopping datanode  
[myUbuntu@hadoop105 hadoop-3.1.3]$ yarn --daemon stop  
nodemanager  
stopping nodemanager
```

6) 如果数据不均衡, 可以用命令实现集群的再平衡

Shell

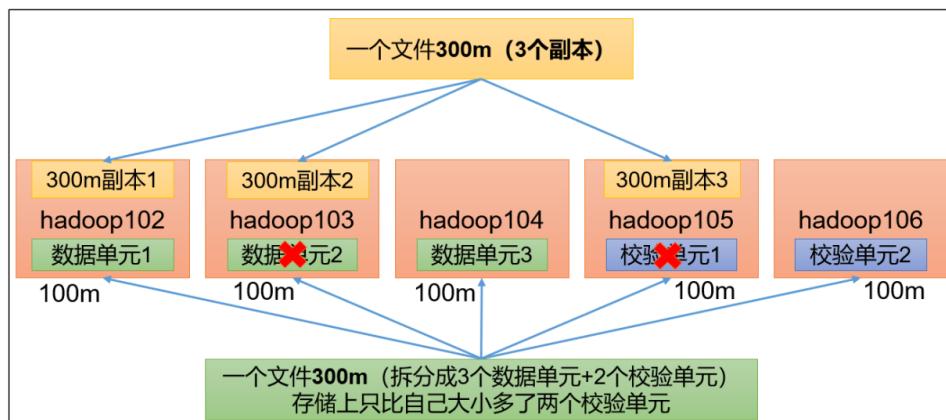
```
[myUbuntu@hadoop102 hadoop-3.1.3]$ sbin/start-balancer.sh -  
threshold 10
```

4.14.5 HDFS—存储优化

4.14.5.1 纠删码

纠删码原理

HDFS 默认情况下, 一个文件有 3 个副本, 这样提高了数据的可靠性, 但也带来了 2 倍的冗余开销。Hadoop3.x 引入了纠删码, 采用计算的方式, 可以节省约 50% 左右的存储空间。



1) 纠删码操作相关的命令

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs ec  
Usage: bin/hdfs ec [COMMAND]  
[-listPolicies]  
[-addPolicies -policyFile <file>]  
[-getPolicy -path <path>]  
[-removePolicy -policy <policy>]  
[-setPolicy -path <path> [-policy <policy>] [-replicate]]  
[-unsetPolicy -path <path>]  
[-listCodecs]  
[-enablePolicy -policy <policy>]  
[-disablePolicy -policy <policy>]  
[-help <command-name>].
```

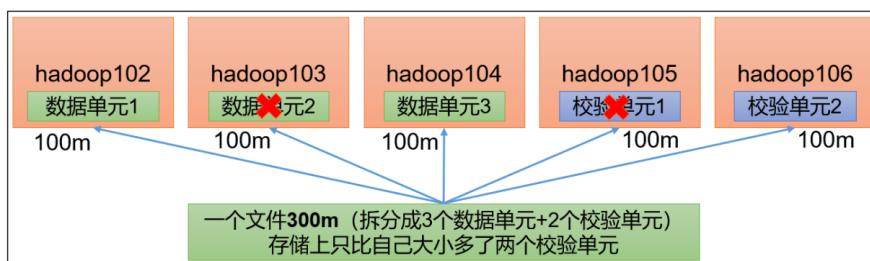
2) 查看当前支持的纠删码策略

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3] hdfs ec -listPolicies  
Erasure Coding Policies:  
ErasureCodingPolicy=[Name=RS-10-4-1024k, Schema=[ECSchema=[Codec=rs,  
numDataUnits=10, numParityUnits=4]], CellSize=1048576, Id=5],  
State=DISABLED  
ErasureCodingPolicy=[Name=RS-3-2-1024k, Schema=[ECSchema=[Codec=rs,  
numDataUnits=3, numParityUnits=2]], CellSize=1048576, Id=2],  
State=DISABLED  
ErasureCodingPolicy=[Name=RS-6-3-1024k, Schema=[ECSchema=[Codec=rs,  
numDataUnits=6, numParityUnits=3]], CellSize=1048576, Id=1],  
State=ENABLED  
ErasureCodingPolicy=[Name=RS-LEGACY-6-3-1024k,  
Schema=[ECSchema=[Codec=rs-legacy, numDataUnits=6, numParityUnits=3]],  
CellSize=1048576, Id=3], State=DISABLED  
ErasureCodingPolicy=[Name=XOR-2-1-1024k, Schema=[ECSchema=[Codec=xor,  
numDataUnits=2, numParityUnits=1]], CellSize=1048576, Id=4],  
State=DISABLED
```

3) 纠删码策略解释:

RS-3-2-1024k: 使用 RS 编码, 每 3 个数据单元, 生成 2 个校验单元, 共 5 个单元, 也就是说: 这 5 个单元中, 只要有任意的 3 个单元存在 (不管是数据单元还是校验单元, 只要总数=3), 就可以得到原始数据。每个单元的大小是 $1024\text{k}=1024*1024=1048576$ 。



RS-10-4-1024k: 使用 RS 编码, 每 10 个数据单元 (cell), 生成 4 个校验单元, 共 14 个单元, 也就是说: 这 14 个单元中, 只要有任意的 10 个单元存在 (不管是数据单元还是校验单元, 只要总数=10), 就可以得到原始数据。每个单元的大小是 $1024\text{k}=1024*1024=1048576$ 。

RS-6-3-1024k: 使用 RS 编码, 每 6 个数据单元, 生成 3 个校验单元, 共 9 个单元, 也就是说: 这 9 个单元中, 只要有任意的 6 个单元存在 (不管是数据单元还是校验单元, 只要总数=6), 就可以得到原始数据。每个单元的大小是 $1024\text{k}=1024*1024=1048576$ 。

RS-LEGACY-6-3-1024k: 策略和上面的 RS-6-3-1024k 一样, 只是编码的算法用的是 rslegacy。

XOR-2-1-1024k: 使用 XOR 编码 (速度比 RS 编码快), 每 2 个数据单元, 生成 1 个校验单元, 共 3 个单元, 也就是说: 这 3 个单元中, 只要有任意的 2 个单元存在 (不管是数据单元还是校验单元, 只要总数=2), 就可以得到原始数据。每个单元的大小是 $1024\text{k}=1024*1024=1048576$ 。

$1024\text{k}=1024*1024=1048576$ 。

纠删码案例实操

纠删码策略是给具体一个路径设置。所有往此路径下存储的文件, 都会执行此策略。

默认只开启对 RS-6-3-1024k 策略的支持, 如要使用别的策略需要提前启用。



1) 需求: 将/input 目录设置为 RS-3-2-1024k 策略

2) 具体步骤

(1) 开启对 RS-3-2-1024k 策略的支持

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs ec -enablePolicy - policy RS-3-2-1024k  
Erasure coding policy RS-3-2-1024k is enabled
```

(2) 在 HDFS 创建目录，并设置 RS-3-2-1024k 策略

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs dfs -mkdir /input  
[myUbuntu@hadoop202 hadoop-3.1.3]$ hdfs ec -setPolicy -path  
/input -policy RS-3-2-1024k
```

(3) 上传文件，并查看文件编码后的存储情况

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs dfs -put web.log /input
```

注：你所上传的文件需要大于 2M 才能看出效果。（低于 2M，只有一个数据单元和两个校验单元）

(4) 查看存储路径的数据单元和校验单元，并作破坏实验

4.14.5.2 异构存储（冷热数据分离）

异构存储主要解决，不同的数据，存储在不同类型的硬盘中，达到最佳性能的问题。



1) 关于存储类型

RAM_DISK: (内存镜像文件系统)

SSD: (SSD固态硬盘)

DISK: (普通磁盘，在HDFS中，如果没有主动声明数据目录存储类型默认都是DISK)

ARCHIVE: (没有特指哪种存储介质，主要的指的是计算能力比较弱而存储密度比较高的存储介质，用来解决数据量的容量扩增的问题，一般用于归档)

2) 关于存储策略 说明：从Lazy_Persist到Cold，分别代表了设备的访问速度从快到慢

策略ID	策略名称	副本分布	
15	Lazy_Persist	RAM_DISK:1 , DISK:n-1	一个副本保存在内存RAM_DISK中，其余副本保存在磁盘中。
12	All_SSD	SSD:n	所有副本都保存在SSD中。
10	One_SSD	SSD:1 , DISK:n-1	一个副本保存在SSD中，其余副本保存在磁盘中。
7	Hot(default)	DISK:n	Hot: 所有副本保存在磁盘中，这也是默认的存储策略。
5	Warm	DISK:1 , ARCHIVE:n-1	一个副本保存在磁盘上，其余副本保存在归档存储上。
2	Cold	ARCHIVE:n	所有副本都保存在归档存储上。

异构存储 Shell 操作

(1) 查看当前有哪些存储策略可以用

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies - listPolicies
```

(2) 为指定路径（数据存储目录）设置指定的存储策略

Shell

```
hdfs storagepolicies -setStoragePolicy -path xxx -policy xxx
```

(3) 获取指定路径（数据存储目录或文件）的存储策略

Shell

```
hdfs storagepolicies -getStoragePolicy -path xxx
```

(4) 取消存储策略；执行改命令之后该目录或者文件，以其上级的目录为准，如果是根目录，那么就是 HOT

Shell

```
hdfs storagepolicies -unsetStoragePolicy -path xxx
```

(5) 查看文件块的分布

Shell

```
bin/hdfs fsck xxx -files -blocks -locations
```

(6) 查看集群节点

Shell

```
hadoop dfsadmin -report
```

测试环境准备

1) 测试环境描述

服务器规模： 5 台

集群配置： 副本数为 2， 创建好带有存储类型的目录（提前创建）

集群规划：

节点	存储类型分配
hadoop102	RAM_DISK, SSD
hadoop103	SSD, DISK
hadoop104	DISK, RAM_DISK
hadoop105	ARCHIVE
hadoop106	ARCHIVE

2) 配置文件信息

(1) 为 hadoop102 节点的 hdfs-site.xml 添加如下信息

XML

```
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
<name>dfs.storage.policy.enabled</name>
<value>true</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>[SSD]file:///opt/module/hadoop-
3.1.3/hdfsdata/ssd,[RAM_DISK]file:///opt/module/hadoop-
3.1.3/hdfsdata/ram_disk</value>
</property>
```

(2) 为 hadoop103 节点的 hdfs-site.xml 添加如下信息

```
XML
<property>
<name>dfs.replication</name> <value>2</value>
</property>
<property>
<name>dfs.storage.policy.enabled</name>
<value>true</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>[SSD]file:///opt/module/hadoop-
3.1.3/hdfsdata/ssd,[DISK]file:///opt/module/hadoop-3.1.3/hdfsdata/disk</value>
</property>
```

(3) 为 hadoop104 节点的 hdfs-site.xml 添加如下信息

```
XML
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
<name>dfs.storage.policy.enabled</name>
<value>true</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>[RAM_DISK]file:///opt/module/hdfsdata/ram_disk,[DISK]file:///o
pt/module/hadoop-3.1.3/hdfsdata/disk</value>
</property>
```

(4) 为 hadoop105 节点的 hdfs-site.xml 添加如下信息

```
XML
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
<name>dfs.storage.policy.enabled</name>
<value>true</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>[ARCHIVE]file:///opt/module/hadoop-3.1.3/hdfsdata/archive</value>
</property>
```

(5) 为 hadoop106 节点的 hdfs-site.xml 添加如下信息

```
XML
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
```

```
<name>dfs.storage.policy.enabled</name>
<value>true</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>[ARCHIVE]file:///opt/module/hadoop-3.1.3/hdfsdata/archive</value>
</property>
```

3) 数据准备

(1) 启动集

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs namenode -format
[myUbuntu@hadoop102 hadoop-3.1.3]$ myhadoop.sh start
```

(1) 并在 HDFS 上创建文件目录

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -mkdir /hdfsdata
```

(2) 并将文件资料上传

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -put /opt/module/hadoop-
3.1.3/NOTICE.txt /hdfsdata
```

HOT 存储策略案例

(1) 最开始我们未设置存储策略的情况下，我们获取该目录的存储策略

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -getStoragePolicy-path
/hdfsdata
```

(2) 我们查看上传的文件块分布

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -locations
[DatanodeInfoWithStorage[192.168.10.104:9866,DS-0b133854-7f9e-48df-939b-
5ca6482c5afb,DISK], DatanodeInfoWithStorage[192.168.10.103:9866,DSca1bd3b9-d9a5-
4101-9f92-3da5f1baa28b,DISK]]
```

未设置存储策略，所有文件块都存储在 DISK 下。所以，默认存储策略为 HOT。

WARM 存储策略测试

(1) 接下来我们为数据降温

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy-path
/hdfsdata -policy WARM
```

(2) 再次查看文件块分布，我们可以看到文件块依然放在原处。

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -locations
```

(3) 我们需要让他 HDFS 按照存储策略自行移动文件块

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(4) 再次查看文件块分布

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -locations  
[DatanodeInfoWithStorage[192.168.10.105:9866,DS-d46d08e1-80c6-4fca-b0a2-  
4a3dd7ec7459,ARCHIVE], DatanodeInfoWithStorage[192.168.10.103:9866,DSca1bd3b9-  
d9a5-4101-9f92-3da5f1baa28b,DISK]]
```

文件块一半在 DISK，一半在 ARCHIVE，符合我们设置的 WARM 策略

COLD 策略测试

(1) 我们继续将数据降温为 cold

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy-path  
/hdfsdata -policy COLD
```

注意：当我们将目录设置为 COLD 并且我们未配置 ARCHIVE 存储目录的情况下，不可以向该目录直接上传文件，会报出异常。

(2) 手动转移

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 检查文件块的分布

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ bin/hdfs fsck /hdfsdata -files -blocks-locations  
[DatanodeInfoWithStorage[192.168.10.105:9866,DS-d46d08e1-80c6-4fca-b0a2-  
4a3dd7ec7459,ARCHIVE], DatanodeInfoWithStorage[192.168.10.106:9866,DS-  
827b3f8b-84d7-47c6-8a14-0166096f919d,ARCHIVE]]
```

所有文件块都在 ARCHIVE，符合 COLD 存储策略。

ONE_SSD 策略测试

(1) 接下来我们将存储策略从默认的 HOT 更改为 One SSD

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy-path  
/hdfsdata -policy One_SSD
```

(2) 手动转移文件块

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 转移完成后，我们查看文件块分布，

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ bin/hdfs fsck /hdfsdata -files -blocks-locations  
[DatanodeInfoWithStorage[192.168.10.104:9866,DS-0b133854-7f9e-48df-939b-
```

```
5ca6482c5afb,DISK], DatanodeInfoWithStorage[192.168.10.103:9866,DS-2481a204-59dd-46c0-9f87-ec4647ad429a,SSD]]
```

文件块分布为一半在 SSD，一半在 DISK，符合 One_SSD 存储策略。

ALL_SSD 策略测试

(1) 接下来，我们再将存储策略更改为 All_SSD

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy-path /hdfsdata -policy All_SSD
```

(2) 手动转移文件块

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 查看文件块分布，我们可以看到，

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ bin/hdfs fsck /hdfsdata -files -blocks-locations [DatanodeInfoWithStorage[192.168.10.102:9866,DS-c997cfb4-16dc-4e69-a0c4-9411a1b0c1eb,SSD], DatanodeInfoWithStorage[192.168.10.103:9866,DS-2481a204-59dd-46c0-9f87-ec4647ad429a,SSD]]
```

所有的文件块都存储在 SSD，符合 All_SSD 存储策略。

LAZY_PERSIST 策略测试

(1) 继续改变策略，将存储策略改为 lazy_persist

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy-path /hdfsdata -policy lazy_persist
```

(2) 手动转移文件块

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 查看文件块分布

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -locations [DatanodeInfoWithStorage[192.168.10.104:9866,DS-0b133854-7f9e-48df-939b-5ca6482c5afb,DISK], DatanodeInfoWithStorage[192.168.10.103:9866,DSca1bd3b9-d9a5-4101-9f92-3da5f1baa28b,DISK]]
```

这里我们发现所有的文件块都是存储在 DISK，按照理论一个副本存储在 RAM_DISK，其他副本存储在 DISK 中，这是因为，我们还需要配置“dfs.datanode.max.locked.memory”，“dfs.block.size”参数。那么出现存储策略为 LAZY_PERSIST 时，文件块副本都存储在 DISK 上的原因有如下两点：

(1) 当客户端所在的 DataNode 节点没有 RAM_DISK 时，则会写入客户端所在的 DataNode 节点的 DISK 磁盘，其余副本会写入其他节点的 DISK 磁盘。

(2) 当客户端所在的 DataNode 有 RAM_DISK，但“dfs.datanode.max.locked.memory”参数值未设置或者设置过小（小于“dfs.block.size”参数值）时，则会写入客户端所在的 DataNode 节点的 DISK 磁盘，其余副本会写入其他节点的 DISK 磁盘。但是由于虚拟机的“max locked memory”为 64KB，所以，如果参数配置过大，还会报出错误：

Shell

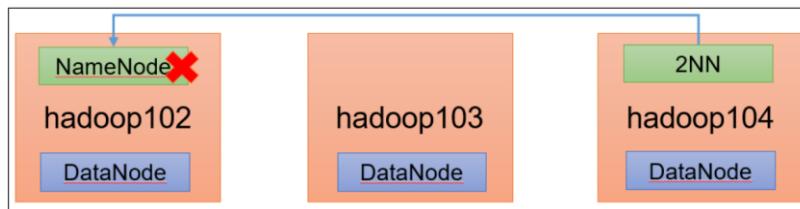
```
ERROR org.apache.hadoop.hdfs.server.datanode.DataNode: Exception in secureMain
java.lang.RuntimeException: Cannot start datanode because the configured max locked
memory size (dfs.datanode.max.locked.memory) of 209715200 bytes is more than the
datanode's available RLIMIT_MEMLOCK ulimit of 65536 bytes.
```

我们可以通过该命令查询此参数的内存

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ ulimit -a
max locked memory (kbytes, -l) 64
```

4.14.6 HDFS—故障排除



- 1) 需求：NameNode 进程挂了并且存储的数据也丢失了，如何恢复 NameNode
- 2) 故障模拟
 - (1) kill -9 NameNode 进程

Shell

```
[myUbuntu@hadoop102 current]$ kill -9 19886
```

- (2) 删除 NameNode 存储的数据（/opt/module/hadoop-3.1.3/data/tmp/dfs/name）

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ rm -rf /opt/module/hadoop-3.1.3/data/dfs/name/*
```

- 3) 问题解决

- (1) 拷贝 SecondaryNameNode 中数据到原 NameNode 存储数据目录

Shell

```
[myUbuntu@hadoop102 dfs]$ scp -r
myUbuntu@hadoop104:/opt/module/hadoop-3.1.3/data/dfs/namesecondary/* ./name/
```

- (2) 重新启动 NameNode

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs --daemon start namenode
```

- (3) 向集群上传一个文件

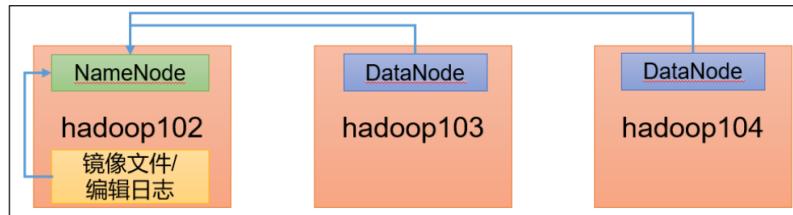
4.14.6.1 集群安全模式&磁盘修复

- 1) 安全模式：文件系统只接受读数据请求，而不接受删除、修改等变更请求

- 2) 进入安全模式场景

➤ NameNode 在加载镜像文件和编辑日志期间处于安全模式；

➤ NameNode 再接收 DataNode 注册时，处于安全模式



3) 退出安全模式条件

dfs.namenode.safemode.min.datanodes: 最小可用 datanode 数量, 默认 0

dfs.namenode.safemode.threshold-pct: 副本数达到最小要求的 block 占系统总 block 数的百分比, 默认 0.999f。
(只允许丢一个块)

dfs.namenode.safemode.extension: 稳定时间, 默认值 30000 毫秒, 即 30 秒

4) 基本语法

集群处于安全模式, 不能执行重要操作 (写操作)。集群启动完成后, 自动退出安全模式。

- (1) bin/hdfs dfsadmin -safemode get (功能描述: 查看安全模式状态)
- (2) bin/hdfs dfsadmin -safemode enter (功能描述: 进入安全模式状态)
- (3) bin/hdfs dfsadmin -safemode leave (功能描述: 离开安全模式状态)
- (4) bin/hdfs dfsadmin -safemode wait (功能描述: 等待安全模式状态)

5) 案例 1: 启动集群进入安全模式

- (1) 重新启动集群

```
Shell
[myUbuntu@hadoop102 subdir0]$ myhadoop.sh stop
[myUbuntu@hadoop102 subdir0]$ myhadoop.sh start
```

- (2) 集群启动后, 立即来到集群上删除数据, 提示集群处于安全模式

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	atguigu	supergroup	21.35 KB	Feb 14 10:58	3	128 MB	NOTICE.txt

6) 案例 2: 磁盘修复

需求: 数据块损坏, 进入安全模式, 如何处理

(1) 分别进入 hadoop102、hadoop103、hadoop104 的 /opt/module/hadoop-3.1.3/data/dfs/data/current/BP-1015489500-192.168.10.102-1611909480872/current/finalized/subdir0/subdir0 目录, 统一删除某 2 个块信息

```
Shell
[myUbuntu@hadoop102 subdir0]$ pwd
/opt/module/hadoop-3.1.3/data/dfs/data/current/BP-1015489500-192.168.10.102-
1611909480872/current/finalized/subdir0/subdir0
[myUbuntu@hadoop102 subdir0]$ rm -rf blk_1073741847
blk_1073741847_1023.meta
```

```
[myUbuntu@hadoop102 subdir0]$ rm -rf blk_1073741865  
blk_1073741865_1042.meta
```

说明：hadoop103/hadoop104 重复执行以上命令

(2) 重新启动集群

Shell

```
[myUbuntu@hadoop102 subdir0]$ myhadoop.sh stop  
[myUbuntu@hadoop102 subdir0]$ myhadoop.sh start
```

(3) 观察 <http://hadoop102:9870/dfshealth.html#tab-overview>

Summary

Security is off.

Safe mode is ON. The reported blocks 5 needs additional 1 blocks to reach the threshold 0.9990 of total blocks 7. The minimum number of live datanodes is not required. Safe mode will be turned off automatically once the thresholds have been reached.

说明：安全模式已经打开，块的数量没有达到要求。

(4) 离开安全模式

Shell

```
[myUbuntu@hadoop102 subdir0]$ hdfs dfsadmin -safemode get  
Safe mode is ON  
[myUbuntu@hadoop102 subdir0]$ hdfs dfsadmin -safemode leave  
Safe mode is OFF
```

(5) 观察 <http://hadoop102:9870/dfshealth.html#tab-overview>

There are 2 missing blocks. The following files may be corrupted:

blk_1073741847 /tmp/logs/atguigu/logs-tfile/application_1611912355782_0001/hadoop103_45020
blk_1073741865 /input/word.txt

Please check the logs or run fsck in order to identify the missing blocks. See the Hadoop FAQ for common causes and potential solutions.

(6) 将元数据删除

Browse Directory

/tmp/logs/atguigu/logs-tfile/application_1611912355782_0001

Show 25 entries

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r-----	atguigu	atguigu	96.69 KB	Jan 29 17:27	3	128 MB	hadoop103_45020

Showing 1 to 1 of 1 entries

Browse Directory

/input

Show 25 entries

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	atguigu	supergroup	36 B	Jan 30 18:34	3	128 MB	word.txt

Showing 1 to 1 of 1 entries

(7) 观察 <http://hadoop102:9870/dfshealth.html#tab-overview>, 集群已经正常

7) 案例 3:

需求: 模拟等待安全模式

(1) 查看当前模式

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -safemode get  
Safe mode is OFF
```

(2) 先进入安全模式

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ bin/hdfs dfsadmin -safemode enter
```

(3) 创建并执行下面的脚本

在/opt/module/hadoop-3.1.3 路径上, 编辑一个脚本 safemode.sh

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ vim safemode.sh  
#!/bin/bash  
hdfs dfsadmin -safemode wait  
hdfs dfs -put /opt/module/hadoop-3.1.3/README.txt /  
[myUbuntu@hadoop102 hadoop-3.1.3]$ chmod 777 safemode.sh  
[myUbuntu@hadoop102 hadoop-3.1.3]$ ./safemode.sh
```

(4) 再打开一个窗口, 执行

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ bin/hdfs dfsadmin -safemode leave
```

(5) 再观察上一个窗口

Shell

```
Safe mode is OFF
```

(6) HDFS 集群上已经有上传的数据了

4.14.6.2 慢磁盘监控

“慢磁盘”指的时写入数据非常慢的一类磁盘。其实慢性磁盘并不少见, 当机器运行时间长了, 上面跑的任务多了, 磁盘的读写性能自然会退化, 严重时就会出现写入数据延时的问题。如何发现慢磁盘?

正常在 HDFS 上创建一个目录, 只需要不到 1s 的时间。如果你发现创建目录超过 1 分钟及以上, 而且这个现象并不是每次都有。只是偶尔慢了一下, 就很有可能存在慢磁盘。

可以采用如下方法找出是哪块磁盘慢:

1) 通过心跳未联系时间。

一般出现慢磁盘现象, 会影响到 DataNode 与 NameNode 之间的心跳。正常情况心跳时间间隔是 3s。超过 3s 说明有异常。

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓ hadoop102:9866 (192.168.202.102:9866)	http://hadoop102:9864	1s	2m	88.34 GB	19	1.98 GB (2.24%)	3.1.3
✓ hadoop103:9866 (192.168.202.103:9866)	http://hadoop103:9864	1s	2m	88.34 GB	19	1.98 GB (2.24%)	3.1.3
✓ hadoop104:9866 (192.168.202.104:9866)	http://hadoop104:9864	1s	2m	88.34 GB	19	1.9 GB (2.16%)	3.1.3

2) fio 命令，测试磁盘的读写性能

(1) 顺序读测试

Shell

```
[myUbuntu@hadoop102 ~]# sudo yum install -y fio
[myUbuntu@hadoop102 ~]# sudo fio -filename=/home/myUbuntu/test.log -direct=1 -
iodepth 1 -thread -
rw=read -ioengine=psync -bs=16k -size=2G -numjobs=10 -runtime=60 -group_reporting -
-name=test_r
Run status group 0 (all jobs):
READ: bw=360MiB/s (378MB/s), 360MiB/s-360MiB/s (378MB/s-378MB/s),
io=20.0GiB (21.5GB), run=56885-56885msec
```

结果显示，磁盘的总体顺序读速度为 360MiB/s。

(2) 顺序写测试

Shell

```
[myUbuntu@hadoop102 ~]# sudo fio -filename=/home/myUbuntu/test.log -direct=1 -
iodepth 1 -thread -
rw=write -ioengine=psync -bs=16k -size=2G -numjobs=10 -runtime=60 -group_reporting -
-name=test_w
Run status group 0 (all jobs):
WRITE: bw=341MiB/s (357MB/s), 341MiB/s-341MiB/s (357MB/s-357MB/s),
io=19.0GiB (21.4GB), run=60001-60001msec -
```

结果显示，磁盘的总体顺序写速度为 341MiB/s。

(3) 随机写测试

Shell

```
[myUbuntu@hadoop102 ~]# sudo fio -filename=/home/myUbuntu/test.log -direct=1 -
iodepth 1 -thread -
rw=randwrite -ioengine=psync -bs=16k -size=2G -numjobs=10 -runtime=60 -
group_reporting -name=test_randw
Run status group 0 (all jobs):
WRITE: bw=309MiB/s (324MB/s), 309MiB/s-309MiB/s (324MB/s-324MB/s),
io=18.1GiB (19.4GB), run=60001-60001msec -
```

结果显示，磁盘的总体随机写速度为 309MiB/s。

(4) 混合随机读写：

Shell

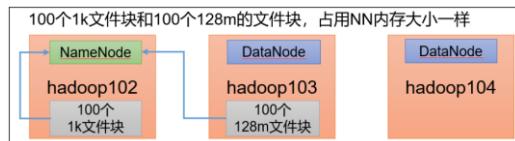
```
[myUbuntu@hadoop102 ~]# sudo fio -filename=/home/myUbuntu/test.log -direct=1 -
iodepth 1 -thread -
rw=randrw -rwmixread=70 -ioengine=psync -bs=16k -size=2G -numjobs=10 -runtime=60
-group_reporting -name=test_r_w -ioscheduler=noop
Run status group 0 (all jobs):
READ: bw=220MiB/s (231MB/s), 220MiB/s-220MiB/s (231MB/s-231MB/s), io=12.9GiB
(13.9GB), run=60001-60001msec
WRITE: bw=94.6MiB/s (99.2MB/s), 94.6MiB/s-94.6MiB/s(99.2MB/s-99.2MB/s),
io=5674MiB (5950MB), run=60001-60001msec
```

结果显示，磁盘的总体混合随机读写，读速度为 220MiB/s，写速度 94.6MiB/s。

4.14.6.3 小文件归档

1) HDFS 存储小文件弊端

每个文件均按块存储，每个块的元数据存储在 NameNode 的内存中，因此 HDFS 存储小文件会非常低效。因为大量的小文件会耗尽 NameNode 中的大部分内存。但注意，存储小文件所需要的磁盘容量和数据块的大小无关。例如，一个 1MB 的文件设置为 128MB 的块存储，实际使用的是 1MB 的磁盘空间，而不是 128MB。



2) 解决存储小文件办法之一

HDFS 存档文件或 HAR 文件，是一个更高效的文件存档工具，它将文件存入 HDFS 块，在减少 NameNode 内存使用的同时，允许对文件进行透明的访问。具体说来，HDFS 存档文件对内还是一个一个独立文件，对 NameNode 而言却是一个整体，减少了 NameNode 的内存。

3) 案例实操

(1) 需要启动 YARN 进程

```
Shell  
[myUbuntu@hadoop102 hadoop-3.1.3]$ start-yarn.sh
```

(2) 归档文件

把 /input 目录里面的所有文件归档成一个叫 input.har 的归档文件，并把归档后文件存储到 /output 路径下。

```
Shell  
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop archive -archiveName  
input.har -p /input /output
```

(3) 查看归档

```
Shell  
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls  
/output/input.har  
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls  
har://output/input.har
```

(4) 解归档文件

```
Shell  
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop fs -cp  
har://output/input.har/* /
```

4.14.7 Hadoop 综合调优

4.14.7.1 Hadoop 小文件优化方法

Hadoop 小文件弊端

HDFS 上每个文件都要在 NameNode 上创建对应的元数据，这个元数据的大小约为 150byte，这样当小文件比较多的时候，就会产生很多的元数据文件，一方面会大量占用 NameNode 的内存空间，另一方面就是元数据

文件过多，使得寻址索引速度变慢。

小文件过多，在进行 MR 计算时，会生成过多切片，需要启动过多的 MapTask。每个 MapTask 处理的数据量小，导致 MapTask 的处理时间比启动时间还小，白白消耗资源。

Hadoop 小文件解决方案

1) 在数据采集的时候，就将小文件或小批数据合成大文件再上传 HDFS（数据源头）

2) Hadoop Archive (存储方向)

是一个高效的将小文件放入 HDFS 块中的文件存档工具，能够将多个小文件打包成一个 HAR 文件，从而达到减少 NameNode 的内存使用

3) CombineTextInputFormat (计算方向)

CombineTextInputFormat 用于将多个小文件在切片过程中生成一个单独的切片或者少量的切片。

4) 开启 uber 模式，实现 JVM 重用 (计算方向)

默认情况下，每个 Task 任务都需要启动一个 JVM 来运行，如果 Task 任务计算的数据量很小，我们可以让同一个 Job 的多个 Task 运行在一个 JVM 中，不必为每个 Task 都开启一个 JVM。

(1) 未开启 uber 模式，在 /input 路径上上传多个小文件并执行 wordcount 程序

Shell

```
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar wordcount /input /output2
```

(2) 观察控制台

Shell

```
2021-02-14 16:13:50,607 INFO mapreduce.Job: Job job_1613281510851_0002  
running in uber mode : false
```

(3) 观察 <http://hadoop103:8088/cluster>

Show 20 entries			
ID	User	Name	Application Type
application_1613281510851_0002	atguigu	word count	MAPREDUCE
Show 20 entries			
Attempt ID		Started	
appattempt_1613281510851_0002_000001		Sun Feb 14 16:13:45 +0800 2021	

Total Allocated Containers: 5
Each table cell represents the number of NodeLocal/RackLocal/OffSwitch containers satisfied by NodeLocal/RackLocal/OffSwitch resource requests.

(4) 开启 uber 模式，在 mapred-site.xml 中添加如下配置

XML

```
<!-- 开启 uber 模式， 默认关闭 -->  
<property>  
<name>mapreduce.job.ubertask.enable</name>  
<value>true</value>  
</property>  
<!-- uber 模式中最大的 mapTask 数量， 可向下修改 -->  
<property>  
<name>mapreduce.job.ubertask.maxmaps</name>  
<value>9</value>  
</property>  
<!-- uber 模式中最大的 reduce 数量， 可向下修改 -->  
<property>  
<name>mapreduce.job.ubertask.maxreduces</name>
```

```
<value>1</value>
</property>
<!-- uber 模式中最大的输入数据量， 默认使用 dfs.blocksize 的值，可向下修改 -->
<property>
<name>mapreduce.job.ubertask.maxbytes</name>
<value></value>
</property>
```

(5) 分发配置

```
Shell
[myUbuntu@hadoop102 hadoop]$ xsync mapred-site.xml
```

(6) 再次执行 wordcount 程序

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop jar share/hadoop/mapreduce/hadoop-
mapreduce-examples-3.1.3.jar wordcount /input /output2
```

(7) 观察控制台

```
Shell
2021-02-14 16:28:36,198 INFO mapreduce.Job: Job job_1613281510851_0003 running in
uber mode : true
```

(8) 观察 <http://hadoop103:8088/cluster>

Total Allocated Containers: 1
Each table cell represents the number of NodeLocal/RackLocal/OffSwitch containers satisfied by NodeLocal/RackLocal/OffSwitch resource requests.

4.14.7.2 测试 MapReduce 计算性能

使用 Sort 程序评测 MapReduce

1) 使用 RandomWriter 来产生随机数，每个节点运行 10 个 Map 任务，每个 Map 产生大约 1G 大小的二进制随机数

```
Shell
[myUbuntu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar randomwriter
random-data
```

(2) 执行 Sort 程序

```
Shell
[myUbuntu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar sort random-data
sorted-data
```

(3) 验证数据是否真正排好序了

```
Shell
[myUbuntu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-clientjobclient-3.1.3-tests.jar
testmapredsort -sortInput random-data-sortOutput sorted-data
```

4.14.7.3 企业开发场景案例

需求

- (1) 需求：从 1G 数据中，统计每个单词出现次数。服务器 3 台，每台配置 4G 内存，4 核 CPU，4 线程。
- (2) 需求分析：

$1G / 128m = 8$ 个 MapTask； 1 个 ReduceTask； 1 个 mrAppMaster

平均每个节点运行 10 个 / 3 台 ≈ 3 个任务 (4 3 3)

HDFS 参数调优

- (1) 修改：hadoop-env.sh

```
Shell
export HDFS_NAMENODE_OPTS="-Dhadoop.security.logger=INFO,RFAS -Xmx1024m"
export HDFS_DATANODE_OPTS="-Dhadoop.security.logger=ERROR,RFAS-Xmx1024m"
```

- (2) 修改 hdfs-site.xml

```
Shell
<!-- NameNode 有一个工作线程池， 默认值是 10 -->
<property>
<name>dfs.namenode.handler.count</name>
<value>21</value>
</property>
```

- (3) 修改 core-site.xml

```
Shell
<!-- 配置垃圾回收时间为 60 分钟 -->
<property>
<name>fs.trash.interval</name>
<value>60</value>
</property>
```

- (4) 分发配置

```
Shell
[myUbuntu@hadoop102 hadoop]$ xsync hadoop-env.sh hdfs-site.xml core-site.xml
```

MapReduce 参数调优

- (1) 修改 mapred-site.xml

```
XML
<!-- 环形缓冲区大小， 默认 100m -->
<property>
<name>mapreduce.task.io.sort.mb</name>
<value>100</value>
</property>
<!-- 环形缓冲区溢写阈值， 默认 0.8 -->
<property>
<name>mapreduce.map.sort.spill.percent</name>
<value>0.80</value>
```

```
</property>
<!-- merge 合并次数， 默认 10 个 -->
<property>
<name>mapreduce.task.io.sort.factor</name>
<value>10</value>
</property>
<!-- maptask 内存， 默认 1g; maptask 堆内存大小默认和该值大小一致
mapreduce.map.java.opts -->
<property>
<name>mapreduce.map.memory.mb</name>
<value>-1</value>
<description>The amount of memory to request from the
scheduler for each map task. If this is not specified or is
non-positive, it is inferred from mapreduce.map.java.opts and
mapreduce.job.heap.memory-mb.ratio. If java-opts are also not
specified, we set it to 1024.
</description>
</property>
<!-- matask 的 CPU 核数， 默认 1 个 -->
<property>
<name>mapreduce.map.cpu.vcores</name>
<value>1</value>
</property>
<!-- matask 异常重试次数， 默认 4 次 -->
<property>
<name>mapreduce.map.maxattempts</name>
<value>4</value>
</property>
<!-- 每个 Reduce 去 Map 中拉取数据的并行数。默认值是 5 -->
<property>
<name>mapreduce.reduce.shuffle.parallelcopies</name>
<value>5</value>
</property>
<!-- Buffer 大小占 Reduce 可用内存的比例， 默认值 0.7 -->
<property>
<name>mapreduce.reduce.shuffle.input.buffer.percent</name>
<value>0.70</value>
</property>
<!-- Buffer 中的数据达到多少比例开始写入磁盘， 默认值 0.66。 -->
<property>
<name>mapreduce.reduce.shuffle.merge.percent</name>
<value>0.66</value>
</property>
<!-- reducetask 内存， 默认 1g; reducetask 堆内存大小默认和该值大小一致
mapreduce.reduce.java.opts -->
<property>
<name>mapreduce.reduce.memory.mb</name>
<value>-1</value>
<description>The amount of memory to request from the scheduler for each reduce task. If
this is not specified or is non-positive, it is inferred from mapreduce.reduce.java.opts and
mapreduce.job.heap.memory-mb.ratio. If java-opts are also not specified, we set it to 1024.
</description>
</property>
<!-- reducetask 的 CPU 核数， 默认 1 个 -->
<property>
<name>mapreduce.reduce.cpu.vcores</name>
<value>2</value>
```

```

</property>
<!-- reducetask 失败重试次数， 默认 4 次 -->
<property>
<name>mapreduce.reduce.maxattempts</name>
<value>4</value>
</property>
<!-- 当 MapTask 完成的比例达到该值后才会为 ReduceTask 申请资源。默认是 0.05
-->
<property>
<name>mapreduce.job.reduce.slowstart.completedmaps</name>
<value>0.05</value>
</property>
<!-- 如果程序在规定的默认 10 分钟内没有读到数据，将强制超时退出 -->
<property>
<name>mapreduce.task.timeout</name>
<value>600000</value>
</property>

```

(2) 分发配置

Shell
[myUbuntu@hadoop102 hadoop]\$ xsync mapred-site.xml

Yarn 参数调优

(1) 修改 yarn-site.xml 配置参数如下：

XML

```

<!-- 选择调度器， 默认容量 -->
<property>
<description>The class to use as the resource scheduler.</description>
<name>yarn.resourcemanager.scheduler.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>
</property>
<!-- ResourceManager 处理调度器请求的线程数量，默认 50；如果提交的任务数大于
50，可以增加该值，但是不能超过 3 台 * 4 线程 = 12 线程（去除其他应用程序实际
不能超过 8） -->
<property>
<description>Number of threads to handle scheduler interface.</description>
<name>yarn.resourcemanager.scheduler.client.thread-count</name>
<value>8</value>
</property>
<!-- 是否让 yarn 自动检测硬件进行配置， 默认是 false，如果该节点有很多其他应用
程序，建议手动配置。如果该节点没有其他应用程序，可以采用自动 -->
<property>
<description>Enable auto-detection of node capabilities such as memory and CPU.
</description>
<name>yarn.nodemanager.resource.detect-hardware-capabilities</name>
<value>false</value>
</property>
<!-- 是否将虚拟核数当作 CPU 核数， 默认是 false，采用物理 CPU 核数 -->
<property>
<description>Flag to determine if logical processors(such as hyperthreads) should be
counted as cores. Only applicable on Linux when yarn.nodemanager.resource.cpu-vcores
is set to -1 and yarn.nodemanager.resource.detect-hardware-capabilities is true.
</description>

```

```

</description>
<name>yarn.nodemanager.resource.count-logical-processors-ascores</name>
<value>false</value>
</property>
<!-- 虚拟核数和物理核数乘数， 默认是 1.0 -->
<property>
<description>Multiplier to determine how to convert phyiscal cores to vcores. This value is used if yarn.nodemanager.resource.cpu-vcores is set to -1(which implies auto-calculate vcores) and yarn.nodemanager.resource.detect-hardware-capabilities is set to true. The number of vcores will be calculated as number of CPUs * multiplier.
</description>
<name>yarn.nodemanager.resource.pcores-vcores-multiplier</name>
<value>1.0</value>
</property>
<!-- NodeManager 使用内存数， 默认 8G， 修改为 4G 内存 -->
<property>
<description>Amount of physical memory, in MB, that can be allocated for containers. If set to -1 and yarn.nodemanager.resource.detect-hardware-capabilities is true, it is automatically calculated(in case of Windows and Linux). In other cases, the default is 8192MB.
</description>
<name>yarn.nodemanager.resource.memory-mb</name>
<value>4096</value>
</property>
<!-- nodemanager 的 CPU 核数， 不按照硬件环境自动设定时默认是 8 个， 修改为 4 个 -->
<property>
<description>Number of vcores that can be allocated for containers. This is used by the RM scheduler when allocating resources for containers. This is not used to limit the number of CPUs used by YARN containers. If it is set to -1 and yarn.nodemanager.resource.detect-hardware-capabilities is true, it is automatically determined from the hardware in case of Windows and Linux. In other cases, number of vcores is 8 by default.</description>
<name>yarn.nodemanager.resource.cpu-vcores</name>
<value>4</value>
</property>
<!-- 容器最小内存， 默认 1G -->
<property>
<description>The minimum allocation for every container request at the RM in MBs. Memory requests lower than this will be set to the value of this property. Additionally, a node manager that is configured to have less memory than this value will be shut down by the resource manager.
</description>
<name>yarn.scheduler.minimum-allocation-mb</name>
<value>1024</value>
</property>
<!-- 容器最大内存， 默认 8G， 修改为 2G -->
<property>
<description>The maximum allocation for every container request at the RM in MBs. Memory requests higher than this will throw an InvalidResourceRequestException.
</description>
<name>yarn.scheduler.maximum-allocation-mb</name>
<value>2048</value>
</property>
<!-- 容器最小 CPU 核数， 默认 1 个 -->
<property>
<description>The minimum allocation for every container request at the RM in terms of

```

virtual CPU cores. Requests lower than this will be set to the value of this property. Additionally, a node manager that is configured to have fewer virtual cores than this value will be shut down by the resource manager.

```
</description>
<name>yarn.scheduler.minimum-allocation-vcores</name>
<value>1</value>
</property>
<!-- 容器最大 CPU 核数， 默认 4 个， 修改为 2 个 -->
<property>
<description>The maximum allocation for every container request at the RM in terms of virtual CPU cores. Requests higher than this will throw an InvalidResourceRequestException.</description>
<name>yarn.scheduler.maximum-allocation-vcores</name>
<value>2</value>
</property>
<!-- 虚拟内存检查， 默认打开， 修改为关闭 -->
<property>
<description>Whether virtual memory limits will be enforced for containers.</description>
<name>yarn.nodemanager.vmem-check-enabled</name>
<value>false</value>
</property>
<!-- 虚拟内存和物理内存设置比例,默认 2.1 -->
<property>
<description>Ratio between virtual memory to physical memory when setting memory limits for containers. Container allocations are expressed in terms of physical memory, and virtual memory usage is allowed to exceed this allocation by this ratio.</description>
<name>yarn.nodemanager.vmem-pmem-ratio</name>
<value>2.1</value>
</property>
```

(2) 分发配置

```
Shell
[myUbuntu@hadoop102 hadoop]$ xsync yarn-site.xml
```

执行程序

(1) 重启集群

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ sbin/stop-yarn.sh
[myUbuntu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

(2) 执行 WordCount 程序

```
Shell
[myUbuntu@hadoop102 hadoop-3.1.3]$ hadoop jar share/hadoop/mapreduce/hadoop-
mapreduce-examples-3.1.3.jar wordcount /input /output
```

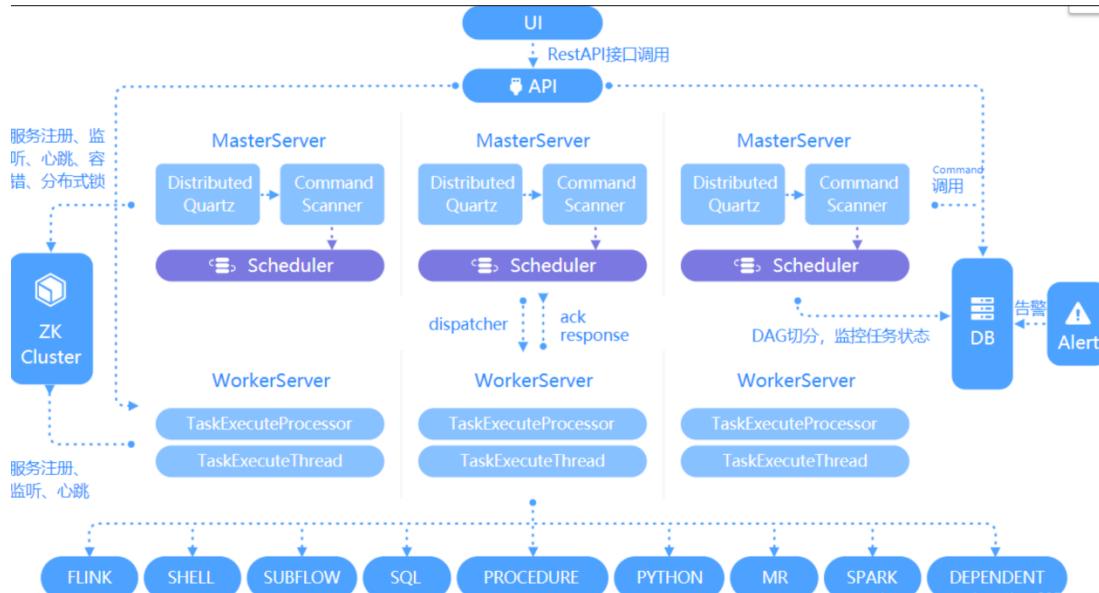
4.15 DolphinScheduler

概念

Apache DolphinScheduler 是一个分布式易扩展的可视化 DAG 工作流任务调度开源系统。适用于企业级场景，提供了一个可视化操作任务、工作流和全生命周期数据处理过程的解决方案。

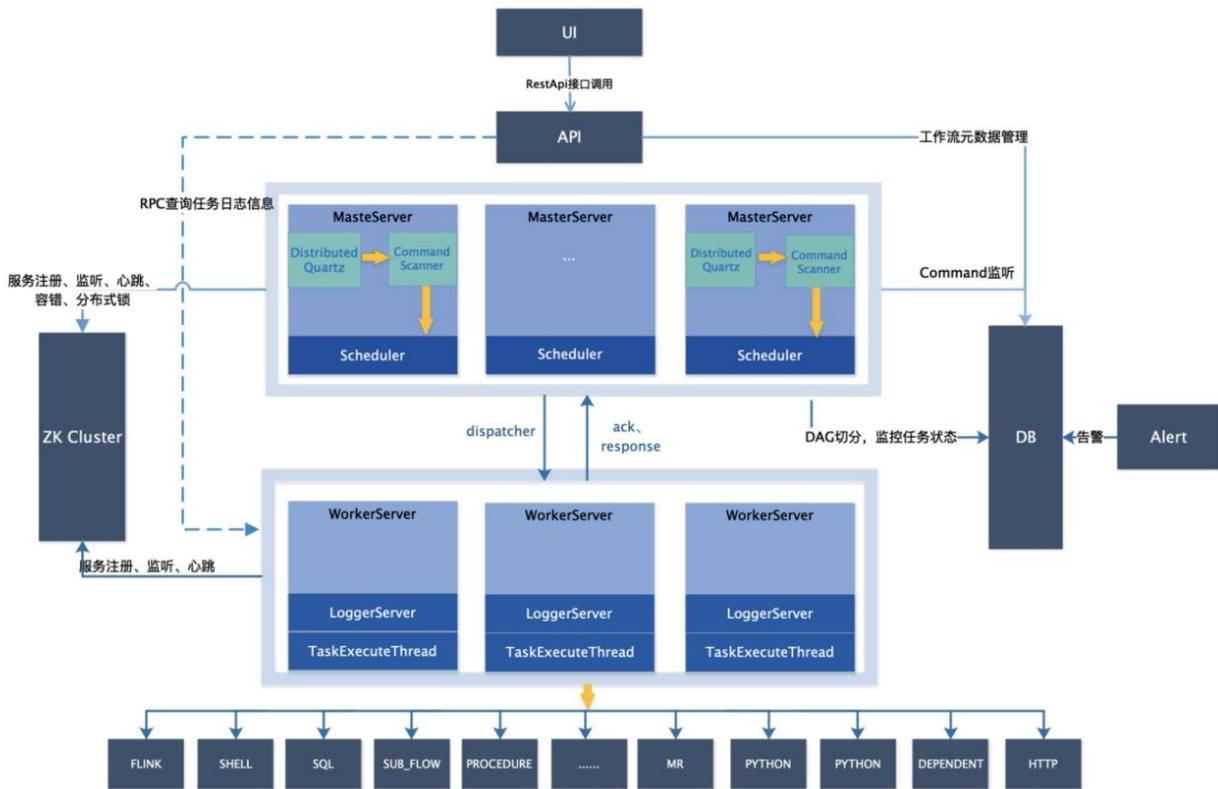
Apache DolphinScheduler 旨在解决复杂的大数据任务依赖关系，并为应用程序提供数据和各种 OPS 编排中的关系。解决数据研发 ETL 依赖错综复杂，无法监控任务健康状态的问题。DolphinScheduler 以 DAG (Directed Acyclic Graph, DAG) 流式方式组装任务，可以及时监控任务的执行状态，支持重试、指定节点恢复失败、暂停、恢复、终止任务等操作。

特性



- 分布式：就是可以这个组件不同的部分可以分布在不同的机器上，整体的协调工作完成任务调度；
- 去中心化：DolphinScheduler 有多台 master，假设有 m₁,m₂,m₃……，也有多台 worker，假设有 w₁,w₂,w₃……，而这些机器都注册在 zookeeper，每次任务 A 来的时候，都同时需要 m 和 w，但是根据配置的不同任务 A 将分到不同的 m 和 w 上，而不是单纯在一个单点上运行，提高容错性；
- 易扩展：DolphinScheduler 是支持扩容的，若需要回收一些机器，也可以支持缩容；
- 可视化 DAG 工作流任务调度：除了提供查看日志来查看任务和工作流的运行情况，还有个有序无环图可查看任务和工作流的运行情况；
- 处理流程中错综复杂的依赖关系：现在有要跑的项目 A,B; 项目 A 有工作流 a₁(内有任务一 s₁, 任务二 s₂), a₂(内有任务一 s₃, 任务二 s₄), a₃(内有任务一 s₅, 任务二 s₆)；项目 B 里面有工作流 b₁(内有任务一 s₇, 任务二 s₈), b₂(内有任务一 s₉, 任务二 s₁₀)；一般的任务调度器，都能保证一个工作流内的任务如 s₁, s₂ 是有序往下运行的，也可并行；但是有些调度器就实现不了工作流 a₁ 和 a₂ 以及 a₃ 之间有依赖关系了，比如某些场景下必须工作流 b₁ 先跑完，才可以跑 a₁, a₂; 甚至必须先跑完整个项目 B 才能跑，那项目 A 就更无法支持了，即项目之间有壁垒，但是 DolphinScheduler 弥补了这些缺陷，即以上的这些复杂依赖关系，不论是工作流之间，还是项目之间都可以得到支持；

架构详解



MasterServer: MasterServer 采用分布式无中心设计理念，MasterServer 主要负责 DAG 任务切分、任务提交监控，并同时监听其它 MasterServer 和 WorkerServer 的健康状态。MasterServer 服务启动时向 Zookeeper 注册临时节点，通过监听 Zookeeper 临时节点变化来进行容错处理。MasterServer 基于 netty 提供监听服务，该服务内主要包含：

1. Distributed Quartz 分布式调度组件，主要负责定时任务的启停操作，当 quartz 调起任务后，Master 内部会有线程池具体负责处理任务的后续操作；
2. MasterSchedulerThread 是一个扫描线程，定时扫描数据库中的 command 表，根据不同的命令类型进行不同的业务操作；
3. MasterExecThread 主要是负责 DAG 任务切分、任务提交监控、各种不同命令类型的逻辑处理；
4. MasterTaskExecThread 主要负责任务的持久化。

WorkerServer: WorkerServer 也采用分布式无中心设计理念，WorkerServer 主要负责任务的执行和提供日志服务。WorkerServer 服务启动时向 Zookeeper 注册临时节点，并维持心跳。Server 基于 netty 提供监听服务，该服务包含：FetchTaskThread 主要负责不断从 Task Queue 中领取任务，并根据不同任务类型调用 TaskScheduleThread 对应执行器。

LoggerServer: 是一个 RPC 服务，提供日志分片查看、刷新和下载等功能；

ZooKeeper: ZooKeeper 服务，系统中的 MasterServer 和 WorkerServer 节点都通过 ZooKeeper 来进行集群管理和容错。另外系统还基于 ZooKeeper 进行事件监听和分布式锁。我们也曾经基于 Redis 实现过队列，不过我们希望 DolphinScheduler 依赖到的组件尽量地少，所以最后还是去掉了 Redis 实现。

Task Queue: 提供任务队列的操作，目前队列也是基于 Zookeeper 来实现。由于队列中存的信息较少，不必担心队列里数据过多的情况，实际上我们压测过百万级数据存队列，对系统稳定性和性能没影响。

Alert: 提供告警相关接口，接口主要包括告警两种类型的告警数据的存储、查询和通知功能。其中通知功能又有邮件通知和 **SNMP(暂未实现)** 两种。

API: API 接口层，主要负责处理前端 UI 层的请求。该服务统一提供 RESTful api 向外部提供请求服务。接口包括工作流的创建、定义、查询、修改、发布、下线、手工启动、停止、暂停、恢复、从该节点开始执行等等。

UI: 系统的前端页面，提供系统的各种可视化操作界面。

5. 项目与组件实践

5.1 raft 算法的 go 语言实现

```
Go
func (r *Raft) run() {
    for {
        // Check if we are doing a shutdown select {
        case <-r.shutdownCh:
            // Clear the leader to prevent forwarding
            r.setLeader("")
            return default:
    }

    // Enter into a sub-FSM switch r.getState() {
    case Follower:
        r.runFollower()
    case Candidate:
        r.runCandidate()
    case Leader:
        r.runLeader()
    }
}

// 跟随者、候选人、领导者 3 种节点状态都有分别对应的功能函数
// 创建 Raft 节点
// 每次服务起来的时候，都是创建一个 raft 节点：
func NewRaft(conf *Config, fsm FSM, logs LogStore, stable StableStore, snaps
SnapshotStore, trans Transport) (*Raft, error) {
    // Validate the configuration.
    if err := ValidateConfig(conf); err != nil {
        return nil, err
    }

    // Ensure we have a LogOutput.
    var logger hclog.Logger
    if conf.Logger != nil {
        logger = conf.Logger
    } else {
        if conf.LogOutput == nil {
            conf.LogOutput = os.Stderr
        }
    }

    logger = hclog.New(&hclog.LoggerOptions{
        Name: "raft",
        Level: hclog.LevelFromString(conf.LogLevel),
        Output: conf.LogOutput,
    })
}

// Try to restore the current term.
currentTerm, err := stable.GetUint64(keyCurrentTerm)
if err != nil && err.Error() != "not found" {
    return nil, fmt.Errorf("failed to load current term: %v", err)
```

```

}

// Read the index of the last log entry.
lastIndex, err := logs.LastIndex()
if err != nil {
    return nil, fmt.Errorf("failed to find last log: %v", err)
}

// Get the last log entry.
var lastLog Log
if lastIndex > 0 {
    if err = logs.GetLog(lastIndex, &lastLog); err != nil {
        return nil, fmt.Errorf("failed to get last log at index %d: %v", lastIndex, err)
    }
}

// Make sure we have a valid server address and ID.
protocolVersion := conf.ProtocolVersion
localAddr := ServerAddress(trans.LocalAddr())
localID := conf.LocalID// TODO (slackpad) - When we deprecate protocol version 2,
remove this// along with the AddPeer() and RemovePeer() APIs.if protocolVersion < 3 &&
string(localID) != string(localAddr) {
    return nil, fmt.Errorf("when running with ProtocolVersion < 3, LocalID must be set to
the network address")
}

// Create Raft struct.
r := &Raft{
    protocolVersion: protocolVersion,
    applyCh: make(chan *logFuture),
    conf: *conf,
    fsm: fsm,
    fsmMutateCh: make(chan interface{}, 128),
    fsmSnapshotCh: make(chan *reqSnapshotFuture),
    leaderCh: make(chan bool),
    localID: localID,
    localAddr: localAddr,
    logger: logger,
    logs: logs,
    configurationChangeCh: make(chan *configurationChangeFuture),
    configurations: configurations{},
    rpcCh: trans.Consumer(),
    snapshots: snaps,
    userSnapshotCh: make(chan *userSnapshotFuture),
    userRestoreCh: make(chan *userRestoreFuture),
    shutdownCh: make(chan struct{}),
    stable: stable,
    trans: trans,
    verifyCh: make(chan *verifyFuture, 64),
    configurationsCh: make(chan *configurationsFuture, 8),
    bootstrapCh: make(chan *bootstrapFuture),
    observers: make(map[uint64]*Observer),
    leadershipTransferCh: make(chan *leadershipTransferFuture, 1),
}

// Initialize as a follower.
r.setState(Follower)

```

```

// Start as leader if specified. This should only be used// for testing purposes.if
conf.StartAsLeader {
    r.setState(Leader)
    r.setLeader(r.localAddr)
}

// Restore the current term and the last log.
r.setCurrentTerm(currentTerm)
r.setLastLog(lastLog.Index, lastLog.Term)

// Attempt to restore a snapshot if there are any.if err := r.restoreSnapshot(); err != nil {
    return nil, err
}

// Scan through the log for any configuration change entries.
snapshotIndex, _ := r.getLastSnapshot()
for index := snapshotIndex + 1; index <= lastLog.Index; index++ {
    var entry Log
    if err := r.logs.GetLog(index, &entry); err != nil {
        r.logger.Error("failed to get log", "index", index, "error", err)
        panic(err)
    }
    r.processConfigurationLogEntry(&entry)
}
r.logger.Info("initial configuration",
    "index", r.configurations.latestIndex,
    "servers", hclog.Fmt("%+v", r.configurations.latest.Servers))

// Setup a heartbeat fast-path to avoid head-of-line// blocking where possible. It MUST be
safe for this// to be called concurrently with a blocking RPC.
trans.SetHeartbeatHandler(r.processHeartbeat)

if conf.skipStartup {
    return r, nil
}
// Start the background work.
r.goFunc(r.run)
r.goFunc(r.runFSM)
r.goFunc(r.runSnapshots)
return r, nil
}

```

以上代码实现了如下功能：

- 恢复选举轮次 term
- 读取最后一个事务日志的索引
- 获取所有的事务日志
- 创建一个 raft 数据结构
- 设置该 raft 节点为 follower 状态
- 更新该 raft 节点的 term 和事务日志
- 恢复快照，如果有的话
- 加载从最后一个快照索引之后的所有事务日志
- 开始心跳

- 启动一个 raft 后台运行协程
- 启动一个状态机后台运行协程
- 启动一个快照处理后台运行协程

```

Go
func (r *Raft) restoreSnapshot() error {
    snapshots, err := r.snapshots.List()
    if err != nil {
        r.logger.Error("failed to list snapshots", "error", err)
        return err
    }

    // Try to load in order of newest to oldest for _, snapshot := range snapshots {
    if !r.conf.NoSnapshotRestoreOnStart {
        _, source, err := r.snapshots.Open(snapshot.ID)
        if err != nil {
            r.logger.Error("failed to open snapshot", "id", snapshot.ID, "error", err)
            continue
        }

        err = r.fsm.Restore(source)
        // Close the source after the restore has completed
        source.Close()
        if err != nil {
            r.logger.Error("failed to restore snapshot", "id", snapshot.ID, "error", err)
            continue
        }

        r.logger.Info("restored from snapshot", "id", snapshot.ID)
    }
    // Update the lastApplied so we don't replay old logs
    r.setLastApplied(snapshot.Index)

    // Update the last stable snapshot info
    r.setLastSnapshot(snapshot.Index, snapshot.Term)

    // Update the configuration
    var conf Configuration
    var index uint64
    if snapshot.Version > 0 {
        conf = snapshot.Configuration
        index = snapshot.ConfigurationIndex
    } else {
        conf = decodePeers(snapshot.Peers, r.trans)
        index = snapshot.Index
    }
    r.setCommittedConfiguration(conf, index)
    r.setLatestConfiguration(conf, index)

    // Success! return nil
}

// If we had snapshots and failed to load them, its an error if len(snapshots) > 0 {
//     return fmt.Errorf("failed to load any existing snapshots")
}
return nil

```

```
}
```

- 遍历所有的快照（从新到旧）
- 恢复每个快照到内存中
- 更新 raft 快照信息
- 更新 raft 配置实例 Configuration 信息

raft 在启动的时候会初始化自己为 follower 状态，然后最后会执行 raft.run 方法：

```
Go
func (r *Raft) run() {
    for {
        // Check if we are doing a shutdownselect {
        case <-r.shutdownCh:
            // Clear the leader to prevent forwarding
            r.setLeader("")
            returndefault:
        }

        // Enter into a sub-FSMswitch r.getState() {
        case Follower:
            r.runFollower()
        case Candidate:
            r.runCandidate()
        case Leader:
            r.runLeader()
        }
    }
}
```

raft 在启动的时候会初始化自己为 follower 状态，然后最后会执行 raft.run 方法：

```
Go
func (r *Raft) run() {
    for {
        // Check if we are doing a shutdownselect {
        case <-r.shutdownCh:
            // Clear the leader to prevent forwarding
            r.setLeader("")
            returndefault:
        }

        // Enter into a sub-FSMswitch r.getState() {
        case Follower:
            r.runFollower()
        case Candidate:
            r.runCandidate()
        case Leader:
            r.runLeader()
        }
    }
}
```

这是一个轮询方法，会一直获取 raft 的状态，然后进入对应的状态处理流程中，这里我们先来看下 follower 流程

```

Go
func (r *Raft) runFollower() {
    didWarn := false
    r.logger.Info("entering follower state", "follower", r, "leader", r.Leader())
    metrics.IncrCounter([]string{ "raft", "state", "follower" }, 1)
    heartbeatTimer := randomTimeout(r.conf.HeartbeatTimeout)

    for r.getState() == Follower {
        select {
        case rpc := <-r.rpcCh:
            r.processRPC(rpc)

        case c := <-r.configurationChangeCh:
            // Reject any operations since we are not the leader
            c.respond(ErrNotLeader)

        case a := <-r.applyCh:
            // Reject any operations since we are not the leader
            a.respond(ErrNotLeader)

        case v := <-r.verifyCh:
            // Reject any operations since we are not the leader
            v.respond(ErrNotLeader)

        case r := <-r.userRestoreCh:
            // Reject any restores since we are not the leader
            r.respond(ErrNotLeader)

        case r := <-r.leadershipTransferCh:
            // Reject any operations since we are not the leader
            r.respond(ErrNotLeader)

        case c := <-r.configurationsCh:
            c.configurations = r.configurations.Clone()
            c.respond(nil)

        case b := <-r.bootstrapCh:
            b.respond(r.liveBootstrap(b.configuration))

        case <-heartbeatTimer:
            // Restart the heartbeat timer
            heartbeatTimer = randomTimeout(r.conf.HeartbeatTimeout)

            // Check if we have had a successful contact
            lastContact := r.LastContact()
            if time.Now().Sub(lastContact) < r.conf.HeartbeatTimeout {
                continue
            }

            // Heartbeat failed! Transition to the candidate state
            lastLeader := r.Leader()
            r.setLeader("")
}

if r.configurations.latestIndex == 0 {
    if !didWarn {
        r.logger.Warn("no known peers, aborting election")
        didWarn = true
}

```

```

        }
    } else if r.configurations.latestIndex == r.configurations.committedIndex &&
        !hasVote(r.configurations.latest, r.localID) {
        if !didWarn {
            r.logger.Warn("not part of stable configuration, aborting election")
            didWarn = true
        }
    } else {
        r.logger.Warn("heartbeat timeout reached, starting election", "last-leader", lastLeader)
        metrics.IncrCounter([]string{"raft", "transition", "heartbeat_timeout"}, 1)
        r.setState(Candidate)
        return
    }

case <-r.shutdownCh:
    return
}
}
}

```

在状态维护过程中，会根据收到的不同的标识进入不同的处理流程，这里只看下心跳这一块：如果与 leader 最后一次的联系时间间隔大于了心跳超时时间，则将自己的状态置为 candidate，也就是观察者，然后就会进入观察者流程中。

Plain Text

```

2020-11-30T11:45:37.238-0500 [INFO] agent.server.raft: entering follower state:
follower="Node at 51.6.196.201:8300 [Follower]" leader=
2020-11-30T11:45:37.240-0500 [WARN] agent.server.memberlist.wan: memberlist:
Binding to public address without encryption!
2020-11-30T11:45:37.240-0500 [INFO] agent.server.serf.wan: serf: EventMemberJoin:
node2_201.dc1 51.6.196.2012020-11-30T11:45:37.240-0500 [INFO]
agent.server.serf.wan: serf: Attempting re-join to previously known node: node3_211.dc1:
51.6.196.211:83022020-11-30T11:45:37.240-0500 [WARN] agent.server.memberlist.lan:
memberlist: Binding to public address without encryption!
2020-11-30T11:45:37.240-0500 [INFO] agent.server.serf.lan: serf: EventMemberJoin:
node2_201 51.6.196.2012020-11-30T11:45:37.240-0500 [INFO] agent.server.serf.lan:
serf: Attempting re-join to previously known node: node3_211: 51.6.196.211:83012020-
11-30T11:45:37.241-0500 [INFO] agent.server: Handled event for server in area:
event=member-join server=node2_201.dc1 area=wan
2020-11-30T11:45:37.241-0500 [INFO] agent.server: Adding LAN server:
server="node2_201 (Addr: tcp/51.6.196.201:8300) (DC: dc1)"2020-11-30T11:45:37.241-
0500 [WARN] agent: Service name will not be discoverable via DNS due to invalid
characters. Valid characters include all alpha-numerics and dashes.: service=java_register
2020-11-30T11:45:37.242-0500 [INFO] agent.server.serf.lan: serf: EventMemberJoin:
node3_211 51.6.196.2112020-11-30T11:45:37.242-0500 [INFO] agent.server: Adding
LAN server: server="node3_211 (Addr: tcp/51.6.196.211:8300) (DC: dc1)"2020-11-
30T11:45:37.242-0500 [INFO] agent.server.serf.lan: serf: Re-joined to previously known
node: node3_211: 51.6.196.211:83012020-11-30T11:45:37.242-0500 [INFO]
agent.server.serf.wan: serf: EventMemberJoin: node3_211.dc1 51.6.196.2112020-11-
30T11:45:37.243-0500 [INFO] agent.server: Handled event for server in area:
event=member-join server=node3_211.dc1 area=wan
2020-11-30T11:45:37.243-0500 [INFO] agent: Started DNS server:
address=51.6.196.201:8600 network=udp
2020-11-30T11:45:37.243-0500 [INFO] agent: Started DNS server:
address=51.6.196.201:8600 network=tcp
2020-11-30T11:45:37.243-0500 [INFO] agent.server.serf.wan: serf: Re-joined to
previously known node: node3_211.dc1: 51.6.196.211:83022020-11-30T11:45:37.244-

```

```

0500 [INFO] agent: Started HTTP server: address=51.6.196.201:8500 network=tcp
2020-11-30T11:45:37.244-0500 [INFO] agent: started state syncer
==> Consul agent running!
2020-11-30T11:45:38.954-0500 [INFO] agent.server: New leader elected:
payload=node3_211

```

- 首先发送一个选票消息给所有的节点：推选自己为 leader，等候其他所有的节点的回复
- 进入 candidate 状态轮询流程：
- 如果收到的其他节点的 vote 选票信息中，选举轮次 term 大于自己的选举轮次 term，表明自己的选举落后了，自己没有资格在竞争 leader，于是设置自己的状态为 follower，同时更新自己的选举轮询为最新的。最后退出候选者流程。
- 如果收到的其他节点的 vote 选票，同意自己的提案，也就是同意自己成为 leader，则自己的选票加一。如果自己的选票数大于了半数节点数，则表明自己成功当选为 leader，更新自己的状态为 leader。最后退出候选者流程。
- 如果选举请求超时，则直接退出候选者流程，等待下一次再次进入候选流程发起选票。

```

Go
func (r *Raft) runCandidate() {
    r.logger.Info("entering candidate state", "node", r, "term", r.getCurrentTerm()+1)
    metrics.IncrCounter([]string{"raft", "state", "candidate"}, 1)

    // Start vote for us, and set a timeout
    voteCh := r.electSelf()

    // Make sure the leadership transfer flag is reset after each run. Having this// flag will set
    // the field LeadershipTransfer in a RequestVoteRequest to true,// which will make other
    // servers vote even though they have a leader already// It is important to reset that flag,
    // because this privilege could be abused// otherwise.defer func()
    { r.candidateFromLeadershipTransfer = false }()

    electionTimer := randomTimeout(r.conf.ElectionTimeout)

    // Tally the votes, need a simple majority
    grantedVotes := 0
    votesNeeded := r.quorumSize()
    r.logger.Debug("votes", "needed", votesNeeded)

    for r.getState() == Candidate {
        select {
        case rpc := <-r.rpcCh:
            r.processRPC(rpc)

        case vote := <-voteCh:
            // Check if the term is greater than ours, bailif vote.Term > r.getCurrentTerm() {
            r.logger.Debug("newer term discovered, fallback to follower")
            r.setState(Follower)
            r.setCurrentTerm(vote.Term)
            return
        }

        // Check if the vote is grantedif vote.Granted {
        grantedVotes++
        r.logger.Debug("vote granted", "from", vote.voterID, "term", vote.Term, "tally",
        grantedVotes)
        }
    }
}

```

```

// Check if we've become the leader if grantedVotes >= votesNeeded {
    r.logger.Info("election won", "tally", grantedVotes)
    r.setState(Leader)
    r.setLeader(r.localAddr)
    return
}

case c := <-r.configurationChangeCh:
    // Reject any operations since we are not the leader
    c.respond(ErrNotLeader)

case a := <-r.applyCh:
    // Reject any operations since we are not the leader
    a.respond(ErrNotLeader)

case v := <-r.verifyCh:
    // Reject any operations since we are not the leader
    v.respond(ErrNotLeader)

case r := <-r.userRestoreCh:
    // Reject any restores since we are not the leader
    r.respond(ErrNotLeader)

case c := <-r.configurationsCh:
    c.configurations = r.configurations.Clone()
    c.respond(nil)

case b := <-r.bootstrapCh:
    b.respond(ErrCantBootstrap)

case <-electionTimer:
    // Election failed! Restart the election. We simply return, // which will kick us back into
    runCandidate
    r.logger.Warn("Election timeout reached, restarting election")
    return
    case <-r.shutdownCh:
        return
    }
}
}

```

在上面的 candidate 流程中，我们说到了，在自己当选为 leader 之后，就会设置自己的状态为 leader，此时就会进入到 leader 流程中：

- 向外通知自己为 leader
- 设置领导者状态
- 设置上一次联系时间：由于我们以前是领导者，当我们下线时更新了上次联系时间，因此在我们从下线到再次成为领导者之间是没有更新最后联系时间的。因此，对于 follower 而言，他们就会认为我们的数据非常陈旧。

Go

```

func (r *Raft) runLeader() {
    r.logger.Info("entering leader state", "leader", r)
    metrics.IncrCounter([]string{"raft", "state", "leader"}, 1)

    // Notify that we are the leader

```

```

asyncNotifyBool(r.leaderCh, true)

// Push to the notify channel if givenif notify := r.conf.NotifyCh; notify != nil {
select {
case notify <- true:
case <-r.shutdownCh:
}
}

// setup leader state. This is only supposed to be accessed within the// leaderloop.
r.setupLeaderState()

// Cleanup state on step down
defer func() {
    // Since we were the leader previously, we update our// last contact time when we step
    down, so that we are not// reporting a last contact time from before we were the// leader.
    Otherwise, to a client it would seem our data// is extremely stale.
    r.setLastContact()

    // Stop replicationfor _, p := range r.leaderState.replState {
        close(p.stopCh)
    }

    // Respond to all inflight operationsfor e := r.leaderState.inflight.Front(); e != nil; e =
e.Next() {
        e.Value.(*logFuture).respond(ErrLeadershipLost)
    }

    // Respond to any pending verify requestsfor future := range r.leaderState.notify {
        future.respond(ErrLeadershipLost)
    }

    // Clear all the state
    r.leaderState.commitCh = nil
    r.leaderState.commitment = nil
    r.leaderState.inflight = nil
    r.leaderState.replState = nil
    r.leaderState.notify = nil
    r.leaderState.stepDown = nil// If we are stepping down for some reason, no known
    leader// We may have stepped down due to an RPC call, which would// provide the leader,
    so we cannot always blank this out.
    r.leaderLock.Lock()
    if r.leader == r.localAddr {
        r.leader = ""
    }
    r.leaderLock.Unlock()

    // Notify that we are not the leader
    asyncNotifyBool(r.leaderCh, false)

    // Push to the notify channel if givenif notify := r.conf.NotifyCh; notify != nil {
    select {
    case notify <- false:
    case <-r.shutdownCh:
        // On shutdown, make a best effort but do not block
        select {
        case notify <- false:
        default:
    }
}
}

```

```

        }
    }
}

// Start a replication routine for each peer
r.startStopReplication()

// Dispatch a no-op log entry first. This gets this leader up to the latest// possible commit
index, even in the absence of client commands. This used// to append a configuration entry
instead of a noop. However, that permits// an unbounded number of uncommitted
configurations in the log. We now// maintain that there exists at most one uncommitted
configuration entry in// any log, so we have to do proper no-ops here.
noop := &logFuture{
    log: Log{
        Type: LogNoop,
    },
}
r.dispatchLogs([]*logFuture{noop})

// Sit in the leader loop until we step down
r.leaderLoop()
}

```

leader 向 follower 发送日志时，会顺带邻近的前一条日志，follower 接收日志时，会在相同任期号和索引位置找前一条日志，如果存在且匹配，则接收日志；否则拒绝，leader 会减少日志索引位置并进行重试，直到某个位置与 follower 达成一致。

然后 follower 删除索引后的所有日志，并追加 leader 发送的日志，一旦日志追加成功，则 follower 和 leader 的所有日志就保持一致。只有在多数派的 follower 都响应接收到日志后，表示事务可以提交，才能返回客户端提交成功。

follower 接收快照流程

- 1. 如果 leaderTermId < currentTerm, 则返回
- 2. 如果是第一个块，创建快照
- 3. 在指定的偏移，将数据写入快照
- 4. 如果不是最后一块，等待更多的块
- 5. 接收完毕后，丢掉以前旧的快照
- 6. 删除掉不需要的日志

Go

```

func (r *Raft) startStopReplication() {
    inConfig := make(map[ServerID]bool, len(r.configurations.latest.Servers))
    lastIdx := r.getLastIndex()

    // Start replication goroutines that need starting for _, server := range
    r.configurations.latest.Servers {
        if server.ID == r.localID {
            continue
        }
        inConfig[server.ID] = true
        if _, ok := r.leaderState.replState[server.ID]; !ok {
            r.logger.Info("added peer, starting replication", "peer", server.ID)
            s := &followerReplication{
                peer:           server,

```

```

commitment:      r.leaderState.commitment,
stopCh:         make(chan uint64, 1),
triggerCh:       make(chan struct{}, 1),
triggerDeferErrorCh: make(chan *deferError, 1),
currentTerm:    r.getCurrentTerm(),
nextIndex:      lastIdx + 1,
lastContact:    time.Now(),
notify:          make(map[*verifyFuture]struct{} {}),
notifyCh:        make(chan struct{}, 1),
stepDown:        r.leaderState.stepDown,
}
r.leaderState.replState[server.ID] = s
r.goFunc(func() { r.replicate(s) })
asyncNotifyCh(s.triggerCh)
r.observe(PeerObservation{Peer: server, Removed: false})
}

// Stop replication goroutines that need stopping
for serverID, repl := range r.leaderState.replState {
if inConfig[serverID] {
  continue
}
// Replicate up to lastIdx and stop
r.logger.Info("removed peer, stopping replication", "peer", serverID, "last-index",
lastIdx)
repl.stopCh <- lastIdx
close(repl.stopCh)
delete(r.leaderState.replState, serverID)
r.observe(PeerObservation{Peer: repl.peer, Removed: true})
}
}

func (r *Raft) replicate(s *followerReplication) {
// Start an async heartbeating routing
stopHeartbeat := make(chan struct{})
defer close(stopHeartbeat)
r.goFunc(func() { r.heartbeat(s, stopHeartbeat) })
}

RPC:
shouldStop := false for !shouldStop {
select {
case maxIndex := <-s.stopCh:
  // Make a best effort to replicate up to this index if maxIndex > 0 {
    r.replicateTo(s, maxIndex)
  }
  return
case deferErr := <-s.triggerDeferErrorCh:
  lastLogIdx, _ := r.getLastLog()
  shouldStop = r.replicateTo(s, lastLogIdx)
  if !shouldStop {
    deferErr.respond(nil)
  } else {
    deferErr.respond(fmt.Errorf("replication failed"))
  }
case <-s.triggerCh:
  lastLogIdx, _ := r.getLastLog()
  shouldStop = r.replicateTo(s, lastLogIdx)
  // This is _not_ our heartbeat mechanism but is to ensure
}
}

```

```

// followers quickly learn the leader's commit index when
// raft commits stop flowing naturally. The actual heartbeats
// can't do this to keep them unblocked by disk IO on the
// follower. See https://github.com/hashicorp/raft/issues/282.case <-
randomTimeout(r.conf.CommitTimeout):
    lastLogIdx, _ := r.getLastLog()
    shouldStop = r.replicateTo(s, lastLogIdx)
}

// If things looks healthy, switch to pipeline mode if !shouldStop && s.allowPipeline {
    goto PIPELINE
}
}
return
}

PIPELINE:
// Disable until re-enabled
s.allowPipeline = false

// Replicates using a pipeline for high performance. This method
// is not able to gracefully recover from errors, and so we fall back
// to standard mode on failure.
if err := r.pipelineReplicate(s); err != nil {
    if err != ErrPipelineReplicationNotSupported {
        r.logger.Error("failed to start pipeline replication to", "peer", s.peer, "error", err)
    }
}
goto RPC
}

```

- 检查是否需要生成快照：比较上一次生成快照的日志 id 跟当前最新的日志 id，如果他们之间的差值达到了设置的快照阈值，则表示需要生成快照。
- 向 FSM 发起创建快照的请求
- 本地持久化快照，同时将本地所有的旧的快照文件删除掉
- 更新最新的生成快照的日志 id
- 截断最新快照日志 id 之前的事务日志，减少事务日志文件的大小。

注：如果 leader 发现 follower 日志落后太远(超过阈值)，则触发发送快照流程

备注：快照不能太频繁，否则会导致磁盘 IO 压力较大；但也需要定期做，清理非必要的日志，缓解日志的空间压力，另外可以提高 follower 追赶的速度。

```

Go
func (r *Raft) runSnapshots() {
    for {
        select {
        case <-randomTimeout(r.conf.SnapshotInterval):
            // Check if we should snapshot if !r.shouldSnapshot()
            continue
        }

        // Trigger a snapshot if _, err := r.takeSnapshot(); err != nil {
        r.logger.Error("failed to take snapshot", "error", err)
        }

        case future := <-r.userSnapshotCh:
    }
}

```

```

// User-triggered, run immediately
if err != nil {
    r.logger.Error("failed to take snapshot", "error", err)
} else {
    future.opener = func() (*SnapshotMeta, io.ReadCloser, error) {
        return r.snapshots.Open(id)
    }
}
future.respond(err)

case <-r.shutdownCh:
    return
}
}
}

```

5.2 Kafka 相关实践

5.2.1 Kafka-Eagle 监控

Kafka-Eagle 框架可以监控 Kafka 集群的整体运行情况，在生产环境中经常使用。

5.2.1.1 MySQL 环境准备

Kafka-Eagle 的安装依赖于 MySQL，MySQL 主要用来存储可视化展示的数据。如果集群中之前安装过 MySQL 可以跨过该步。

5.2.1.2 Kafka 环境准备

1) 关闭 Kafka 集群

```

Shell
[myUbuntu@hadoop102 kafka]$ kf.sh stop

```

2) 修改/opt/module/kafka/bin/kafka-server-start.sh 命令中

```

Shell
[myUbuntu@hadoop102 kafka]$ vim bin/kafka-server-start.sh

```

修改如下参数值：

```

Shell
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi

```

为

```

Shell
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-server -Xms2G -Xmx2G -XX:PermSize=128m -
XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=8 -
XX:ConcGCThreads=5 -XX:InitiatingHeapOccupancyPercent=70"
    export JMX_PORT="9999"
#export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"

```

注意：修改之后在启动 Kafka 之前要分发之其他节点

Shell

```
[myUbuntu@hadoop102 bin]$ xsync kafka-server-start.sh
```

5.2.1.3 Kafka-Eagle 安装

0) 官网：<https://www.kafka-eagle.org/>

1) 上传压缩包 **kafka-eagle-bin-2.0.8.tar.gz** 到集群/opt/software 目录

2) 解压到本地

Shell

```
[myUbuntu@hadoop102 software]$ tar -zxvf kafka-eagle-bin-2.0.8.tar.gz
```

3) 进入刚才解压的目录

Shell

```
[myUbuntu@hadoop102 kafka-eagle-bin-2.0.8]$ ll
总用量 79164
-rw-rw-r--. 1 myUbuntu myUbuntu 81062577 10月 13 00:00 efak-web-2.0.8-bin.tar.gz
```

4) 将 **efak-web-2.0.8-bin.tar.gz** 解压至/**opt/module**

Shell

```
[myUbuntu@hadoop102 kafka-eagle-bin-2.0.8]$ tar -zxvf efak-web-2.0.8-bin.tar.gz -C
/opt/module/
```

5) 修改名称

Shell

```
[myUbuntu@hadoop102 module]$ mv efak-web-2.0.8/ efak
```

6) 修改配置文件 /**opt/module/efak/conf/system-config.properties**

Shell

```
[myUbuntu@hadoop102 conf]$ vim system-config.properties
#####
multi zookeeper & kafka cluster list
Settings prefixed with 'kafka.eagle.' will be deprecated, use 'efak.' instead
#####
efak.zk.cluster.alias=cluster1
cluster1.zk.list=hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka

#####
zookeeper enable acl
#####
cluster1.zk.acl.enable=false
cluster1.zk.acl.schema=digest
cluster1.zk.acl.username=test
cluster1.zk.acl.password=test123

#####
```

```
broker size online list
#####
cluster1.efak.broker.size=20

#####
zk client thread limit
#####
kafka.zk.limit.size=32
#####

EFAK webui port
#####
efak.webui.port=8048

#####
kafka jmx acl and ssl authenticate
#####
cluster1.efak.jmx.acl=false
cluster1.efak.jmx.user=keadmin
cluster1.efak.jmx.password=keadmin123
cluster1.efak.jmx.ssl=false
cluster1.efak.jmx.truststore.location=/data/ssl/certificates/kafka.truststore
cluster1.efak.jmx.truststore.password=ke123456

#####
kafka offset storage
#####
# offset 保存在 kafka
cluster1.efak.offset.storage=kafka

#####
kafka jmx uri
#####
cluster1.efak.jmx.uri=service:jmx:rmi://jndi/rmi://%s/jmxrmi

#####
kafka metrics, 15 days by default
#####
efak.metrics.charts=true
efak.metrics.retain=15

#####
kafka sql topic records max
#####
efak.sql.topic.records.max=5000
efak.sql.topic.preview.records.max=10

#####
delete kafka topic token
#####
efak.topic.token=keadmin

#####
kafka sasl authenticate
#####
cluster1.efak.sasl.enable=false
cluster1.efak.sasl.protocol=SASL_PLAINTEXT
cluster1.efak.sasl.mechanism=SCRAM-SHA-256
```

```

cluster1.efak.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
e required username="kafka" password="kafka-eagle";
cluster1.efak.sasl.client.id=
cluster1.efak.blacklist.topics=
cluster1.efak.sasl.cgroup.enable=false
cluster1.efak.sasl.cgroup.topics=
cluster2.efak.sasl.enable=false
cluster2.efak.sasl.protocol=SASL_PLAINTEXT
cluster2.efak.sasl.mechanism=PLAIN
cluster2.efak.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required username="kafka" password="kafka-eagle";
cluster2.efak.sasl.client.id=
cluster2.efak.blacklist.topics=
cluster2.efak.sasl.cgroup.enable=false
cluster2.efak.sasl.cgroup.topics=

#####
kafka ssl authenticate
#####
cluster3.efak.ssl.enable=false
cluster3.efak.ssl.protocol=SSL
cluster3.efak.ssl.truststore.location=
cluster3.efak.ssl.truststore.password=
cluster3.efak.ssl.keystore.location=
cluster3.efak.ssl.keystore.password=
cluster3.efak.ssl.key.password=
cluster3.efak.ssl.endpoint.identification.algorithm=https
cluster3.efak.blacklist.topics=
cluster3.efak.ssl.cgroup.enable=false
cluster3.efak.ssl.cgroup.topics=

#####
kafka sqlite jdbc driver address
#####
# 配置 mysql 连接
efak.driver=com.mysql.jdbc.Driver
efak.url=jdbc:mysql://hadoop102:3306/ke?useUnicode=true&characterEncoding=UTF-
8&zeroDateTimeBehavior=convertToNull
efak.username=root
efak.password=000000

#####
kafka mysql jdbc driver address
#####
efak.driver=com.mysql.cj.jdbc.Driver
efak.url=jdbc:mysql://127.0.0.1:3306/ke?useUnicode=true&characterEncoding=UTF-
8&zeroDateTimeBehavior=convertToNull
#efak.username=root
#efak.password=123456

```

7) 添加环境变量

```

Shell
[myUbuntu@hadoop102 conf]$ sudo vim /etc/profile.d/my_env.sh
# kafkaEFAK
export KE_HOME=/opt/module/efak
export PATH=$PATH:$KE_HOME/bin

```

注意：source /etc/profile

Shell

```
[myUbuntu@hadoop102 conf]$ source /etc/profile
```

8) 启动

(1) 注意：启动之前需要先启动 ZK 以及 KAFKA。

Shell

```
[myUbuntu@hadoop102 kafka]$ kf.sh start
```

(2) 启动 efak

Shell

```
[myUbuntu@hadoop102 efak]$ bin/ke.sh start
Version 2.0.8 -- Copyright 2016-2021
*****
EFAK Service has started success.
Welcome, Now you can visit 'http://192.168.10.102:8048'!
Account:admin ,Password:123456
*****
<Usage> ke.sh [start|status|stop|restart|stats] </Usage>
<Usage> https://www.kafka-eagle.org/ </Usage>
*****
```

说明：如果停止 efak，执行命令。

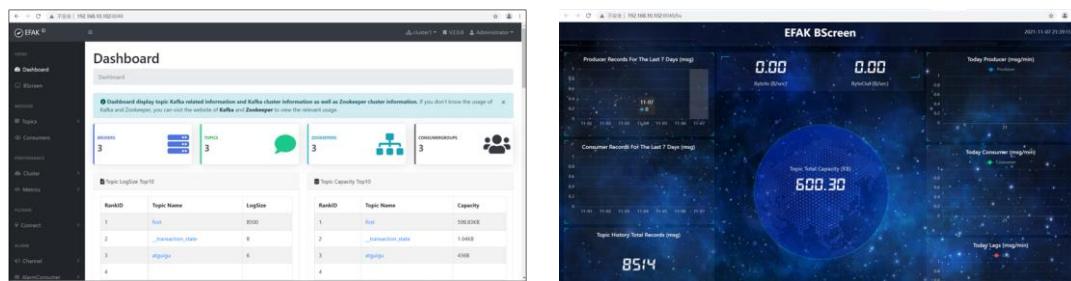
Shell

```
[myUbuntu@hadoop102 efak]$ bin/ke.sh stop
```

5.2.1.4 Kafka-Eagle 页面操作

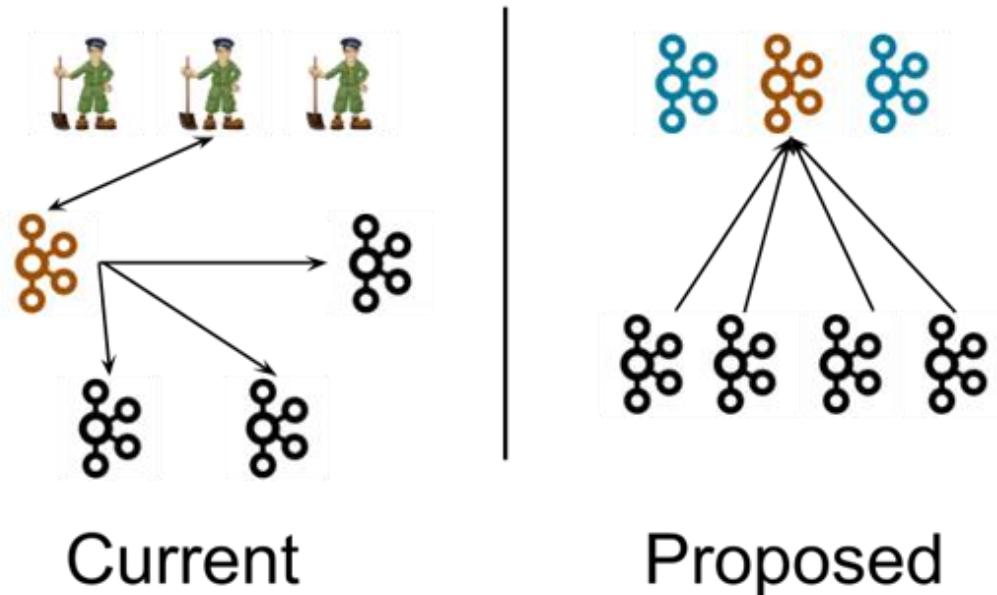
1) 登录页面查看监控数据

<http://192.168.10.102:8048/>



5.2.2 Kafka-Kraft 模式部署

5.2.2.1 Kafka-Kraft 架构



左图为 Kafka 现有架构，元数据在 zookeeper 中，运行时动态选举 controller，由 controller 进行 Kafka 集群管理。右图为 kraft 模式架构（实验性），不再依赖 zookeeper 集群，而是用三台 controller 节点代替 zookeeper，元数据保存在 controller 中，由 controller 直接进行 Kafka 集群管理。

这样做的好处有以下几个：

- Kafka 不再依赖外部框架，而是能够独立运行；
- controller 管理集群时，不再需要从 zookeeper 中先读取数据，集群性能上升；
- 由于不依赖 zookeeper，集群扩展时不再受到 zookeeper 读写能力限制；
- controller 不再动态选举，而是由配置文件规定。这样我们可以有针对性的加强 controller 节点的配置，而不是像以前一样对随机 controller 节点的高负载束手无策。

5.2.2.2 Kafka-Kraft 集群部署

1) 再次解压一份 kafka 安装包

```
Shell
[myUbuntu@hadoop102 software]$ tar -zvxf kafka_2.12-3.0.0.tgz -C /opt/module/
```

2) 重命名为 kafka2

```
Shell
[myUbuntu@hadoop102 module]$ mv kafka_2.12-3.0.0/ kafka2
```

3) 在 hadoop102 上修改 /opt/module/kafka2/config/kraft/server.properties 配置文件

```
Shell
[myUbuntu@hadoop102 kraft]$ vim server.properties

#kafka 的角色 (controller 相当于主机、broker 节点相当于从机，主机类似 zk 功能)
process.roles=broker, controller
#节点 ID
node.id=2
#controller 服务协议别名
controller.listener.names=CONTROLLER
#全 Controller 列表
```

```

controller.quorum.voters=2@hadoop102:9093,3@hadoop103:9093,4@hadoop104:9093
#不同服务器绑定的端口
listeners=PLAINTEXT://:9092,CONTROLLER://:9093
#broker 服务协议别名
inter.broker.listener.name=PLAINTEXT
#broker 对外暴露的地址
advertised.Listeners=PLAINTEXT://hadoop102:9092
#协议别名到安全协议的映射
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,
SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
#kafka 数据存储目录
log.dirs=/opt/module/kafka2/data

```

```

#kafka的角色 (controller相当于主机、broker节点相当于从机，主机类似zk功能)
process.roles=broker, controller
#节点ID
node.id=2
#controller服务协议别名
controller.listener.names=CONTROLLER
#全Controller列表
controller.quorum.voters=2@hadoop102:9093,3@hadoop103:9093,4@hadoop104:9093
#不同服务器绑定的端口
listeners=PLAINTEXT://:9092,CONTROLLER://:9093
#broker服务协议别名
inter.broker.listener.name=PLAINTEXT
#broker对外暴露的地址
advertised.Listeners=PLAINTEXT://hadoop102:9092
#协议别名到安全协议的映射
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAI
NTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
#kafka数据存储目录
log.dirs=/opt/module/kafka2/data
~
```

4) 分发 kafka2

Shell
[myUbuntu@hadoop102 module]\$ xsync kafka2/

- 在 hadoop103 和 hadoop104 上需要对 node.id 相应改变，值需要和 controller.quorum.voters 对应。
- 在 hadoop103 和 hadoop104 上需要根据各自的主机名称，修改相应的 advertised.Listeners 地址。

5) 初始化集群数据目录

- (1) 首先生成存储目录唯一 ID。

Shell
[myUbuntu@hadoop102 kafka2]\$ bin/kafka-storage.sh random-uuid
J7s9e8PPTKOO47PxzI39VA

- (2) 用该 ID 格式化 kafka 存储目录（三台节点）。

Shell
[myUbuntu@hadoop102 kafka2]\$ bin/kafka-storage.sh format -t
J7s9e8PPTKOO47PxzI39VA -c /opt/module/kafka2/config/kraft/server.properties
[myUbuntu@hadoop103 kafka2]\$ bin/kafka-storage.sh format -t
J7s9e8PPTKOO47PxzI39VA -c /opt/module/kafka2/config/kraft/server.properties
[myUbuntu@hadoop104 kafka2]\$ bin/kafka-storage.sh format -t
J7s9e8PPTKOO47PxzI39VA -c /opt/module/kafka2/config/kraft/server.properties

```
[myUbuntu@hadoop102 kafka2]$ bin/kafka-storage.sh format -t J7s9e8PPTK0047Pxzi39  
VA -c /opt/module/kafka2/config/kraft/server.properties  
[myUbuntu@hadoop103 kafka2]$ bin/kafka-storage.sh format -t J7s9e8PPTK0047Pxzi39  
VA -c /opt/module/kafka2/config/kraft/server.properties  
[myUbuntu@hadoop104 kafka2]$ bin/kafka-storage.sh format -t J7s9e8PPTK0047Pxzi39  
VA -c /opt/module/kafka2/config/kraft/server.properties
```

6) 启动 kafka 集群

Shell

```
[myUbuntu@hadoop102 kafka2]$ bin/kafka-server-start.sh -daemon  
config/kraft/server.properties  
[myUbuntu@hadoop103 kafka2]$ bin/kafka-server-start.sh -daemon  
config/kraft/server.properties  
[myUbuntu@hadoop104 kafka2]$ bin/kafka-server-start.sh -daemon  
config/kraft/server.properties
```

```
[myUbuntu@hadoop102 kafka2]$ bin/kafka-server-start.sh -daemon config/kraft/server.properties  
[myUbuntu@hadoop103 kafka2]$ bin/kafka-server-start.sh -daemon config/kraft/server.properties  
[myUbuntu@hadoop104 kafka2]$ bin/kafka-server-start.sh -daemon config/kraft/server.properties
```

7) 停止 kafka 集群

Shell

```
[myUbuntu@hadoop102 kafka2]$ bin/kafka-server-stop.sh  
[myUbuntu@hadoop103 kafka2]$ bin/kafka-server-stop.sh  
[myUbuntu@hadoop104 kafka2]$ bin/kafka-server-stop.sh
```

```
[myUbuntu@hadoop102 kafka2]$ bin/kafka-server-stop.sh  
[myUbuntu@hadoop103 kafka2]$ bin/kafka-server-stop.sh  
[myUbuntu@hadoop104 kafka2]$ bin/kafka-server-stop.sh
```

5.2.2.3 Kafka-Kraft 集群启动停止脚本

1) 在/home/myUbuntu/bin 目录下创建文件 kf2.sh 脚本文件

Shell

```
[myUbuntu@hadoop102 bin]$ vim kf2.sh
```

脚本如下：

```
#!/bin/bash

case $1 in
"start"){
    for i in hadoop102 hadoop103 hadoop104
    do
        echo " -----启动 $i Kafka2-----"
        ssh $i "/opt/module/kafka2/bin/kafka-server-start.sh -daemon  
/opt/module/kafka2/config/kraft/server.properties"
        done
} ;;
"stop"){
    for i in hadoop102 hadoop103 hadoop104
    do
```

```
echo " ----- 停止 $i Kafka2 -----"
ssh $i "/opt/module/kafka2/bin/kafka-server-stop.sh"
done
} ;;
esac
```

2) 添加执行权限

```
Shell
[myUbuntu@hadoop102 bin]$ chmod +x kf2.sh
```

3) 启动集群命令

```
Shell
[myUbuntu@hadoop102 ~]$ kf2.sh start
```

4) 停止集群命令

```
Shell
[myUbuntu@hadoop102 ~]$ kf2.sh stop
```

5.2.3 案例 1：Flume+Kafka+SparkStreaming 实现词频统计

任务描述

配置 Kafka 和 Flume，把 Flume Source 类别设置为 netcat，绑定到 localhost 的 33333 端口，通过“telnet localhost 33333”命令向 Flume Source 发送消息，然后，让 Flume 把消息发送给 Kafka，并让 Kafka 发送消息到 Spark Streaming，Spark Streaming 组件收到各种单词消息后，对单词进行词频统计，在屏幕上打印出每个单词出现了几次。

5.2.3.1 Flume 的安装和准备

Flume 是非常流行的日志采集系统，可以发送消息给 Kafka，这里我们介绍如何配置将消息发送给 Kafka。

首先下载安装 Flume。关于 Flume 的概念和安装方法，参考 4.3 中 Flume 的相关内容。

本次实验要使用 netcat source 发送消息，在实验之前，要先搞懂 Flume 是如何配置 source、sink 和 channels 的。之后测试 Flume 是否可用。

5.2.3.2 Spark 准备工作

要通过 Kafka 连接 Spark 来进行 Spark Streaming 操作，Kafka 和 Flume 等高级输入源，需要依赖独立的库（jar 文件）。也就是说 Spark 需要 jar 包让 Kafka 和 Spark streaming 相连。按照前面安装好的 Spark 版本，这些 jar 包都不在里面。打开一个新的终端，输入以下命令启动 spark-shell：

```
Shell
cd /usr/local/spark
./bin/spark-shell
```

启动成功后，在 spark-shell 中执行下面 import 语句：

```
Shell
import org.apache.spark.streaming.kafka._
```

可以看到，马上会报错如下，因为找不到相关的 jar 包。

Plaintext

```
<console>:23: error: object kafka is not a member of package org.apache.spark.streaming
      import org.apache.spark.streaming.kafka._  
                           ^
```

根据 Spark 官网的说明，对于 Spark2.1.0 版本，如果要使用 Kafka，则需要下载 spark-streaming-kafka-0-8_2.11 相关 jar 包。

现在在 Linux 系统中，访问 http://mvnrepository.com/artifact/org.apache.spark/spark-streaming-kafka-0-8_2.11/2.1.0，里面有提供 spark-streaming-kafka-0-8_2.11-2.1.0jar 文件的下载，其中，2.11 表示 scala 的版本，2.1.0 表示 Spark 版本号。下载后的文件会被默认保存在当前 Linux 登录用户 hadoop 的下载目录下（“/home/hadoop/下载”）。

把这个文件复制到 Spark 目录的 jars 目录下。请新打开一个终端，输入下面命令：

Shell

```
cd /usr/local/spark/jars
mkdir kafka
cd ~
cd 下载
cp ./spark-streaming-kafka-0-8_2.11-2.1.0.jar /usr/local/spark/jars/kafka
```

这样，就把 spark-streaming-kafka-0-8_2.11-2.1.0.jar 文件拷贝到了“/usr/local/spark/jars/kafka”目录下。

下面还要继续把 Kafka 安装目录的 libs 目录下的所有 jar 文件复制到“/usr/local/spark/jars/kafka”目录下输入以下命令：

Shell

```
cd /usr/local/kafka/libs
ls
cp ./* /usr/local/spark/jars/kafka
```

5.2.3.3 实验过程

1. 编写 Flume 配置文件 flume_to_kafka.conf

输入命令：

Shell

```
cd /usr/local/flume
cd conf
vim flume_to_kafka.conf
```

内容如下：

Plaintext

```
a1.sources=r1
a1.channels=c1
a1.sinks=k1
#Describe/configure the source
a1.sources.r1.type=netcat
a1.sources.r1.bind=localhost
a1.sources.r1.port=33333
#Describe the sink
a1.sinks.k1.type=org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic=test
a1.sinks.k1.kafka.bootstrap.servers=localhost:9092
a1.sinks.k1.kafka.producer.acks=1
```

```
a1.sinks.k1.flumeBatchSize=20
#Use a channel which buffers events in memory
a1.channels.c1.type=memory
a1.channels.c1.capacity=1000000
a1.channels.c1.transactionCapacity=1000000
#Bind the source and sink to the channel
a1.sources.r1.channels=c1
a1.sinks.k1.channel=c1
```

2. 编写 Spark Streaming 程序(进行词频统计的程序)

首先创建 scala 代码的目录结构。

输入命令：

```
Shell
cd /usr/local/spark/mycode
mkdir flume_to_kafka
cd flume_to_kafka
mkdir -p src/main/scala
cd src/main/scala
```

输入命令：

```
Shell
vim KafkaWordCounter.scala
```

KafkaWordCounter.scala 是用于单词词频统计，它会把从 kafka 发送过来的单词进行词频统计，代码内容如下：

```
Plaintext
package org.apache.spark.examples.streaming
import org.apache.spark._
import org.apache.spark.SparkConf
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.kafka.KafkaUtils

object KafkaWordCounter{
def main(args:Array[String]){
StreamingExamples.setStreamingLogLevels()
val sc=new SparkConf().setAppName("KafkaWordCounter").setMaster("local[2]")
val ssc=new StreamingContext(sc,Seconds(10))
ssc.checkpoint("file:///usr/local/spark/mycode/flume_to_kafka/checkpoint") //设置检查点
val zkQuorum="localhost:2181" //Zookeeper 服务器地址
val group="1" //topic 所在的 group，可以设置为自己想要的名称，比如不用 1，而是
val group = "test-consumer-group"
val topics="test" //topics 的名称
val numThreads=1 //每个 topic 的分区数
val topicMap=topics.split(",").map(_.numThreads.toInt)).toMap
val lineMap=KafkaUtils.createStream(ssc,zkQuorum,group,topicMap)
val lines=lineMap.map(_._2)
val words=lines.flatMap(_.split(" "))
val pair=words.map(x => (x,1))
val wordCounts=pair.reduceByKeyAndWindow(_ + _, _ - _,Minutes(2),Seconds(10),2)
wordCounts.print
ssc.start
ssc.awaitTermination
```

```
}
```

reduceByKeyAndWindow 函数作用解释如下：

reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks]) 更加高效的

reduceByKeyAndWindow，每个窗口的 reduce 值，是基于先前窗口的 reduce 值进行增量计算得到的；它会对进入滑动窗口的新数据进行 reduce 操作，并对离开窗口的老数据进行“逆向 reduce”操作。但是，只能用于“可逆 reduce 函数”，即那些 reduce 函数都有一个对应的“逆向 reduce 函数”（以 InvFunc 参数传入）；

此代码中就是一个窗口转换操作 reduceByKeyAndWindow，其中，Minutes(2)是滑动窗口长度，Seconds(10)是滑动窗口时间间隔（每隔多长时间滑动一次窗口）。reduceByKeyAndWindow 中就使用了加法和减法这两个 reduce 函数，加法和减法这两种 reduce 函数都是“可逆的 reduce 函数”，也就是说，当滑动窗口到达一个新的位置时，原来之前被窗口框住的部分数据离开了窗口，又有新的数据被窗口框住，但是，这时计算窗口内单词的词频时，不需要对当前窗口内的所有单词全部重新执行统计，而是只要把窗口内新增进来的元素，增量加入到统计结果中，把离开窗口的元素从统计结果中减去，这样，就大大提高了统计的效率。尤其对于窗口长度较大时，这种“逆函数”带来的效率的提高是很明显的。

3. 创建 StreamingExamples.scala

继续在当前目录(/usr/local/spark/mycode/flume_to_kafka/src/main/scala)下创建 StreamingExamples.scala 代码文件，用于设置 log4j:

输入命令：

```
Shell
```

```
vim StreamingExamples.scala
```

代码内容如下：

```
Plaintext
```

```
package org.apache.spark.examples.streaming
import org.apache.spark.internal.Logging
import org.apache.log4j.{Level, Logger}
//Utility functions for Spark Streaming examples.
object StreamingExamples extends Logging {
//Set reasonable logging levels for streaming if the user has not configured log4j.
  def setStreamingLogLevels() {
    val log4jInitialized = Logger.getRootLogger.getAllAppenders.hasMoreElements
    if (!log4jInitialized) {
      // We first log something to initialize Spark's default logging, then we override the
      // logging level.
      logInfo("Setting log level to [WARN] for streaming example." + " To override add a
custom log4j.properties to the classpath.")
      Logger.getRootLogger.setLevel(Level.WARN)
    }
  }
}
```

4. 打包文件 simple.sbt

输入命令：

```
Shell
```

```
cd /usr/local/spark/mycode/flume_to_kafka
```

输入命令：

```
Shell
```

```
vim simple.sbt
```

内容如下：

```
Plaintext
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.spark" % "spark-streaming_2.11" % "2.1.0"
libraryDependencies += "org.apache.spark" % "spark-streaming-kafka-0-8_2.11" %
"2.1.0"
```

要注意版本号一定要设置正确。

在/usr/local/spark/mycode/flume_to_kafka 目录下输入命令：

```
Shell
cd /usr/local/spark/mycode/flume_to_kafka
find .
```

打包之前，这条命令用来查看代码结构，目录结构如下所示：

```
Plaintext
.
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/KafkaWordCounter.scala
./src/main/scala/StreamingExamples.scala
```

5. 打包编译

一定要在/usr/local/spark/mycode/flume_to_kafka 目录下运行打包命令
输入命令：

```
Shell
cd /usr/local/spark/mycode/flume_to_kafka
/usr/local/sbt/sbt package
```

第一次打包的过程可能会很慢，请耐心等待几分钟。打包成功后，会看到如下 SUCCESS 的提示。

```
Plaintext
OpenJDK 64-Bit Server VM warning: ignoring option MaxPermSize=256M; support was
removed in 8.0
[info] Set current project to Simple Project (in build
file:/usr/local/spark/mycode/flume_to_kafka/)
[info] Compiling 2 Scala sources to /usr/local/spark/mycode/flume_to_kafka/target/scala-
2.11/classes...
[info] Packaging /usr/local/spark/mycode/flume_to_kafka/target/scala-2.11/simple-
project_2.11-1.0.jar ...
[info] Done packaging.
[success] Total time: 7 s, completed 2018-9-5 10:03:08
```

6. 启动 zookeeper 和 kafka

先启动 zookeeper

输入命令：

```
Shell  
cd /usr/local/kafka  
.bin/zookeeper-server-start.sh config/zookeeper.properties
```

屏幕显示如下，不要关闭这个终端。

打开第二个终端，然后输入下面命令启动 Kafka 服务：

输入命令：

```
Shell  
cd /usr/local/kafka  
bin/kafka-server-start.sh config/server.properties
```

屏幕显示如下，不要关闭这个终端

7.运行程序 KafkaWordCounter

打开第三个终端，我们已经创建过 topic，名为 test（这是你之前在 flume_to_kafka.conf 中设置的 topic 名字），端口号 2181。在第三个终端运行“KafkaWordCounter”程序，进行词频统计，由于现在没有启动输入，所以只有提示信息，没有结果。

输入命令：

```
Shell  
cd /usr/local/spark  
/usr/local/spark/bin/spark-submit --driver-class-path  
/usr/local/spark/jars/*:/usr/local/spark/jars/kafka/* --class  
"org.apache.spark.examples.streaming.KafkaWordCounter"  
/usr/local/spark/mycode/flume_to_kafka/target/scala-2.11/simple-project_2.11-1.0.jar
```

其中”/usr/local/spark/jars/“和”/usr/local/spark/jars/kafka/“用来指明引用的 jar 包，

“org.apache.spark.examples.streaming.KafkaWordCounter”代表包名和类名，这是编写 KafkaWordCounter.scala 里面的包名和类名，最后一个参数用来说明打包文件的位置。

执行该命令后，屏幕上会显示程序运行的相关信息，并会每隔 10 秒钟刷新一次信息，用来输出词频统计的结果，此时还只有提示信息，如下所示：

Plaintext

Time: 1687318063000 ms

Time: 1687318073000 ms

Time: 1687318083000 ms

Time: 1687318093000 ms

Time: 1687318103000 ms

按 Ctrl+z 可以退出程序运行

在启动 Flume 之前，Zookeeper 和 Kafka 要先启动成功，不然启动 Flume 会报连不上 Kafka 的错误。

8.启动 flume agent

打开第四个终端，在这个新的终端中启动 Flume Agent

输入命令：

```
Shell  
cd /usr/local/flume  
bin/flume-ng agent --conf ./conf --conf-file ./conf/flume_to_kafka.conf --name a1 -  
Dflume.root.logger=INFO,console
```

启动 agent 以后，该 agent 就会一直监听 localhost 的 33333 端口，这样，我们下面就可以通过“telnet localhost 33333”命令向 Flume Source 发送消息。这个终端也不要关闭，让它一直处于监听状态。

9.打开第五个终端，发送消息

输入命令：

```
Shell  
telnet localhost 33333
```

这个端口 33333 是在 flume conf 文件中设置的 source

在这个窗口里面随便敲入若干个字符和若干个回车，这些消息都会被 Flume 监听到，Flume 把消息采集到以后汇集到 Sink，然后由 Sink 发送给 Kafka 的 topic(test)。因为 spark Streaming 程序不断地在监控 topic，在输入终端和前面运行词频统计程序那个终端窗口内看到类似如下的统计结果：

Flume 发送之前，KafkaWordCounter 检测不到消息，没有统计结果，只有提示信息。

在该窗口输入数据：(每一行结尾加一个空格再按回车发送)

```
Plaintext  
hello world  
hello pku  
hello spark  
大数据
```

可以看到统计结果如下，其中“，4)”是对换行符的统计

```
Plaintext  
-----  
Time: 1687318113000 ms  
-----  
(hello,3)  
(pku,1)  
.4)  
(spark,1)  
(大数据,1)  
(world,1)
```

```
-----  
Time: 1687318113000 ms  
-----  
(hello,3)  
(pku,1)  
,4)  
(spark,1)  
(大数据,1)  
(world,1)
```

输入数据：只输入一个回车

统计结果如下：

Plaintext

```
Time: 1687318140000 ms
```

```
(hello,3)  
(pku,1)  
.5)  
(spark,1)  
(大数据,1)  
(world,1)
```

```
-----  
Time: 1687318189000 ms  
-----  
(hello,4)  
(pku,1)  
,1)ark  
,5)  
(spark,1)  
(大数据,1)  
(world,1)
```

可以看到如上所示的统计结果，其他的都没变，“4”“变成“，5）”，这是换行符的数量加一。

可以看到，统计结果在不断的累积，但是随着运行屏幕显示其实还和窗口设置有关。如果输完最后一个单词直接敲回车发送，但是接收统计端会把换行符和最后一个单词组合起来形成错误的格式，导致统计结果有误。所以在每行输入完成后先多敲一个空格再按换行符发送数据。

比如输入“hello spark”(spark 末尾不加空格)

统计结果如下：

Plaintext

```
Time: 1687318189000 ms
```

```
(hello,4)  
(pku,1)  
,1)ark  
,5)  
(spark,1)
```

(大数据,1)
(world,1)

```
-----  
Time: 1687318189000 ms  
-----
```

```
(hello,4)  
(pku,1)  
,1)ark  
,5)  
(spark,1)  
(大数据,1)  
(world,1)
```

可以看到"hello"被正确的统计了，数量加一，然而“spark”却和换行符结合变成不能识别的格式“,1)ark”。所以，这里在发送每一行数据时都要在末尾输入一个空格再按下回车。

运行时应该有 5 个终端处于开启状态，分别是

Zookeeper 服务\Kafka 服务\Spark Streaming(KafkaWordCounter)\flume agent\telnet 发送端至此，就完成了 Flume、Kafka 和 Spark Streaming 整合进行词频统计的任务。

注意：

由于用 netcatsource 作为 Flume 的输入源统计时总会加上格式不正确的换行符，这里试验了使用 Avro source 作为 Flume 的输入源进行 Flume_Kafka_SparkStreaming 词频统计。

只需修改 flume_to_kafka.conf 文件为以下内容，并在启动时有所不同。

```
Shell  
cd /usr/local/flume/conf  
vim flume_to_kafka2.conf
```

配置 flume 的 sources 为 avro

```
Plaintext  
a1.sources=r1  
a1.channels=c1  
a1.sinks=k1  
# Describe/configure the source  
a1.sources.r1.type=avro  
a1.sources.r1.bind=0.0.0.0  
a1.sources.r1.port=4141  
# Describe the sink  
a1.sinks.k1.type=org.apache.flume.sink.kafka.KafkaSink  
a1.sinks.k1.kafka.topic=test  
a1.sinks.k1.kafka.bootstrap.servers=localhost:9092  
a1.sinks.k1.kafka.producer.acks=1  
a1.sinks.k1.flumeBatchSize=20  
# Use a channel which buffers events in memory  
a1.channels.c1.type=memory  
a1.channels.c1.capacity=1000000  
a1.channels.c1.transactionCapacity=1000000  
#Bind the source and sink to the channel  
a1.sources.r1.channels=c1
```

```
a1.sinks.k1.channel=c1
```

启动时，顺序如下：

1.启动 zookeeper

```
Shell  
cd /usr/local/kafka  
.bin/zookeeper-server-start.sh config/zookeeper.properties
```

2.启动 kafka

```
Shell  
cd /usr/local/kafka  
.bin/kafka-server-start.sh config/server.properties
```

3.启动 Flume Agent

这里指定我们新编写的配置文件 flume_to_kafka2.conf

```
Shell  
cd /usr/local/flume  
.usr/local/flume/bin/flume-ng agent -c . -f /usr/local/flume/conf/flume_to_kafka2.conf -n  
a1 -Dflume.root.logger=INFO,console #启动日志控制台
```

4.启动 Spark Streaming 应用程序

```
Shell  
cd /usr/local/spark  
.usr/local/spark/bin/spark-submit --driver-class-path  
.usr/local/spark/jars/*:/usr/local/spark/jars/kafka/* --class  
"org.apache.spark.examples.streaming.KafkaWordCounter"  
.usr/local/spark/mycode/flume_to_kafka/target/scala-2.11/simple-project_2.11-1.0.jar
```

此时只显示提示信息

Plaintext

```
-----  
Time: 1687318371000 ms  
-----
```

```
-----  
Time: 1687318381000 ms  
-----
```

```
-----  
Time: 1687318391000 ms  
-----
```

5.创建指定文件

先打开另外一个终端，在/usr/local/flume 下创建一个文件 log.00

```
Shell  
cd /usr/local/flume  
sudo vim log.00
```

文件内容随意，这里输入

```
Plaintext
hello world
hello pku
hello success
大数据
```

6.再打开另外一个终端，执行以下命令，4141 是 avro.conf 文件里的端口名

```
Shell
cd /usr/local/flume
bin/flume-ng avro-client --conf conf -H localhost -p 4141 -F /usr/local/flume/log.00
```

SparkStreaming 程序统计显示如下，可以看到不存在换行符格式的问题，正确统计结果。

```
Plaintext
-----
Time: 1687318461000 ms
-----
(hello,3)
(pku,1)
(大数据,1)
(success,1)
(world,1)
```

```
-----
Time: 1687318461000 ms
-----
(hello,3)
(pku,1)
(大数据,1)
(success,1)
(world,1)
```

至此，使用 Avro source 作为 Flume 的输入源实现 Flume+Kafka+SparkStreaming 词频统计完成。

5.2.4 案例 2：Spark+Kafka 构建实时分析 Dashboard

5.2.4.1 实验环境准备

Python 安装

Ubuntu16.04 系统自带 Python2.7 和 Python3.5，本案例使用 Anaconda 中创建的 python3.7.13 环境。

Python 依赖库

本案例主要使用了两个 Python 库，Flask 和 Flask-SocketIO，启动进入 Ubuntu 系统，打开一个命令行终端，然后开启 Anaconda 环境：

```
Shell
conda activate env_py37
```

Anaconda 之所以强大，其中一个原因是其便捷的第三方库管理。可以使用如下 Shell 命令完成 Flask 和 Flask-SocketIO 这两个 Python 第三方库的安装以及与 Kafka 相关的 Python 库的安装：

```
Shell
```

```
conda install flask  
conda install flask-socketio  
conda install kafka-python
```

这些安装好的库在程序文件的开头可以直接用来引用。比如：

```
Python  
from flask import Flask  
from flask_socketio import SocketIO  
from kafka import KafkaConsumer
```

PyCharm 安装

PyCharm 是一款 Python 开发 IDE，可以极大方便工程管理以及程序开发。在 Ubuntu 系统中打开自带的火狐浏览器，前往 [PyCharm 官网](#) 下载免费的 Community 版本，下载后默认会被保存到当前登录用户的主目录下的“下载”目录中。比如，如果当前使用 hadoop 用户名登录了 Ubuntu 系统，那么，就会被保存在“/home/hadoop/下载”这个目录下。然后执行如下命令对安装文件进行解压缩：

```
Shell  
cd ~ #进入当前 hadoop 用户的主目录  
sudo tar -zxvf ~/下载/pycharm-community-2016.3.2.tar.gz -C /usr/local #把 pycharm 解  
压缩到 /usr/local 目录下  
cd /usr/local  
sudo mv pycharm-community-2016.3.2 pycharm #重命名  
sudo chown -R hadoop ./pycharm #把 pycharm 目录权限赋予给当前登录 Ubuntu 系统  
的 hadoop 用户
```

然后，执行如下命令启动 PyCharm：

```
Shell  
cd /usr/local/pycharm  
./bin/pycharm.sh #启动 PyCharm
```

执行上述命令之后，即可开启 PyCharm。

Python 工程目录结构

这里先给出本案例 Python 工程的目录结构，后续的操作可以根据这个目录进行操作。

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
labproject > app.py
Project consumer
labproject ~/mydir/labproject
data
  user_log.csv
scripts
  consumer.py
  producer.py
static
  js
    exporting.js
    highcharts.js
    jquery-3.1.1.min.js
    socket.io.js
    socket.io.js.map
templates
  index.html
app.py
External Libraries
Python 3.5.2 (/usr/bin/python)
background_thread() for msg in consumer:
    import json
    from flask import Flask, render_template
    from flask_socketio import SocketIO
    from kafka import KafkaConsumer
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'secret!'
    socketio = SocketIO(app, engineio_logger=True)
    thread = None
    consumer = KafkaConsumer('result')

    def background_thread():
        for msg in consumer:
            data_json = msg.value.decode('utf8')
            data_list = json.loads(data_json)
            girl = 0
            boy = 0
            for data in data_list:
                if '0' in data.keys():
                    girl = data['0']
                elif '1' in data.keys():
                    boy = data['1']
                else:
                    continue
            result = str(girl) + ',' + str(boy)
            print(result)
            socketio.emit('test_message', {'data': result})

```

1. data 目录存放的是用户日志数据；
2. scripts 目录存放的是 Kafka 生产者和消费者；
3. static/js 目录存放的是前端所需要的 js 框架；
4. templates 目录存放的是 html 页面；
5. app.py 为 web 服务器，接收 Structured Streaming 处理后的结果，并推送实时数据给浏览器；
6. External Libraries 是本项目所依赖的 Python 库，是 PyCharm 自动生成。

此外，Spark 自带 Scala，因此如果是开发 Spark 应用程序，则没必要单独安装 Scala。

5.2.4.2 数据处理和 Python 操作 Kafka

数据预处理

数据集介绍

本案例采用的数据集压缩包为 data_format.zip，来自于：<https://pan.baidu.com/s/1cs02Nc>，该数据集压缩包是淘宝 2015 年双 11 前 6 个月(包含双 11)的交易数据(交易数据有偏移，但是不影响实验的结果)，里面包含 3 个文件，分别是用户行为日志文件 user_log.csv、回头客训练集 train.csv、回头客测试集 test.csv。在这个案例中只是用 user_log.csv 这个文件，下面列出文件 user_log.csv 的数据格式定义：

用户行为日志 user_log.csv，日志中的字段定义如下：

1. user_id | 买家 id
2. item_id | 商品 id
3. cat_id | 商品类别 id
4. merchant_id | 卖家 id
5. brand_id | 品牌 id
6. month | 交易时间:月
7. day | 交易事件:日
8. action | 行为,取值范围{0,1,2,3},0 表示点击, 1 表示加入购物车, 2 表示购买, 3 表示关注商品
9. age_range | 买家年龄分段: 1 表示年龄=50,0 和 NULL 则表示未知
10. gender | 性别:0 表示女性, 1 表示男性, 2 和 NULL 表示未知

11. province| 收获地址省份

数据具体格式如下：

Plaintext

```
user_id,item_id,cat_id,merchant_id,brand_id,month,day,action,age_range,gender,province  
328862,323294,833,2882,2661,08,29,0,0,1,内蒙古  
328862,844400,1271,2882,2661,08,29,0,1,1,山西  
328862,575153,1271,2882,2661,08,29,0,2,1,山西  
328862,996875,1271,2882,2661,08,29,0,1,1,内蒙古  
328862,1086186,1271,1253,1049,08,29,0,0,2,浙江  
328862,623866,1271,2882,2661,08,29,0,0,2,黑龙江  
328862,542871,1467,2882,2661,08,29,0,5,2,四川  
328862,536347,1095,883,1647,08,29,0,7,1,吉林
```

这个案例实时统计每秒中男女生购物人数，因此针对每条购物日志，我们只需要获取 gender 即可，然后发送给 Kafka，接下来 Structured Streaming 再接收 gender 进行处理。

数据预处理

本案例使用 Python 对数据进行预处理，并将处理后的数据直接通过 Kafka 生产者发送给 Kafka，这里需要先安装 Python 操作 Kafka 的代码库，请在 Ubuntu 中打开一个命令行终端，执行如下 Shell 命令来安装 Python 操作 Kafka 的代码库（注：如果之前已经安装过，则这里不需要安装）：

Shell

```
conda activate env_py37  
conda install kafka-python
```

接着可以写如下 Python 代码，文件名为 producer.py：(具体的工程文件结构参照步骤一)

Python

```
# coding: utf-8  
import csv  
import time  
from kafka import KafkaProducer  
  
# 实例化一个 KafkaProducer 示例，用于向 Kafka 投递消息  
producer = KafkaProducer(bootstrap_servers='localhost:9092')  
# 打开数据文件  
csvfile = open("../data/user_log.csv","r")  
# 生成一个可用于读取 csv 文件的 reader  
reader = csv.reader(csvfile)  
  
for line in reader:  
    gender = line[9] # 性别在每行日志代码的第 9 个元素  
    if gender == 'gender':  
        continue # 去除第一行表头 time.sleep(0.1) # 每隔 0.1 秒发送一行数据  
    # 发送数据，topic 为'sex'  
    producer.send('sex',line[9].encode('utf8'))
```

上述代码首先是先实例化一个 Kafka 生产者。然后读取用户日志文件，每次读取一行，接着每隔 0.1 秒发送给 Kafka，这样 1 秒发送 10 条购物日志。这里发送给 Kafka 的 topic 为'sex'。

Python 操作 Kafka

这里可以写一个 KafkaConsumer 测试数据是否投递成功，代码如下，文件名为 consumer.py

```
Python
from kafka import KafkaConsumer

consumer = KafkaConsumer('sex')
for msg in consumer:
    print((msg.value).decode('utf8'))
```

在开启上述 KafkaProducer 和 KafkaConsumer 之前，需要先开启 Kafka，命令如下：

```
Shell
cd /usr/local/kafka
bin/zookeeper-server-start.sh config/zookeeper.properties
```

打开一个新的命令行窗口，输入命令如下：

```
Shell
cd /usr/local/kafka
bin/kafka-server-start.sh config/server.properties
```

在 Kafka 开启之后，即可开启 KafkaProducer 和 KafkaConsumer。开启方法如下：
请在 Ubuntu 中，打开一个命令行终端窗口，执行如下命令：

```
Shell
cd /home/hadoop/mydir/labproject/scripts #进入到代码目录
python3 producer.py #启动生产者发送消息给 Kafka
```

然后，在 Ubuntu 中，打开另外一个命令行终端窗口，执行如下命令：

```
Shell
cd /home/hadoop/mydir/labproject/scripts #进入到代码目录
python3 consumer.py #启动消费者从 Kafka 接收消息
```

运行上面这条命令以后，会看到屏幕上会输出一行又一行的数字，如：

```
Plaintext
2
1
1
1
2
0
2
1
.....
```

但也可以不用上面这种命令行方式来启动生产者和消费者，如果目前使用的是开发工具 PyCharm，则也可以直接在代码区域右键，点击 Run 'producer' 以及 Run 'consumer'，来运行生产者和消费者。如果生产者和消费者运行成功，则在 consumer 窗口会输出如下信息：

```
Terminal: Local × Local (2) × +  
(py37) hadoop@ubuntu:~/PycharmProjects/lab/scripts$ python3 consumer.py  
1  
1  
1  
1  
2  
2  
2
```

The screenshot shows a PyCharm interface with a terminal window titled "Local". The command "python3 consumer.py" is run, and the output is displayed as a series of numbers: 1, 1, 1, 1, 2, 2, 2.

5.2.4.3 Structured Streaming 实时处理数据

编程思路

本案例在于实时统计每秒中男女生购物人数，而 Structured Streaming 接收的数据为 1,1,0,2...，其中 0 代表女性，1 代表男性，所以对于 2 或者 null 值，则不考虑。其实通过分析，可以发现这个就是典型的 wordcount 问题，而且是基于 Spark 流计算。女生的数量，即为 0 的个数，男生的数量，即为 1 的个数。

因此利用 Structured Streaming 的 groupBy 接口，设置窗口大小为 1，滑动步长为 1，这样统计出的 0 和 1 的个数即为每秒男生女生的人数。

编程实现

配置 Spark 开发 Kafka 环境

点击 <https://pan.baidu.com/s/12JCw8L0kt4x0rZFoTWIGAw?pwd=b5wl>，下载 Spark 连接 Kafka 的代码库。然后把下载的代码库放到目录/usr/local/spark/jars 目录下，命令如下：

Shell

```
sudo mv ~/下载/spark-streaming_2.12-3.2.0.jar /usr/local/spark/jars  
sudo mv ~/下载/spark-streaming-kafka-0-10_2.12-3.2.0.jar /usr/local/spark/jars
```

然后在/usr/local/spark/jars 目录下新建 kafka 目录，把/usr/local/kafka/libs 下所有函数库复制到 /usr/local/spark/jars/kafka 目录下，命令如下

Shell

```
cd /usr/local/spark/jars  
mkdir kafka  
cd kafka  
cp /usr/local/kafka/libs/* .
```

然后，修改 Spark 配置文件，命令如下

Shell

```
cd /usr/local/spark/conf  
sudo vim spark-env.sh
```

把 Kafka 相关 jar 包的路径信息增加到 spark-env.sh，修改后的 spark-env.sh 类似如下：

Plaintext

```
export  
SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoopclasspath):/usr/local/spark/ja  
rs/kafka/*:/usr/local/kafka/libs/*
```

因为我使用的是 anaconda 中创建的 python 环境，所以介绍一下，怎么为 spark 设置 python 环境。需要修改 conf 目录下的 spark_env.sh：在这个文件的开头添加：

Plaintext

```
export PYSPARK_PYTHON=/home/hadoop/anaconda3/envs/py37/bin/python
```

建立 pySpark 项目

首先在/usr/local/spark/mycode 新建项目目录

Shell

```
cd /usr/local/spark/mycode  
mkdir kafka
```

然后在 kafka 这个目录下创建一个 kafka_test.py 文件。

Python

```
from kafka import KafkaProducer  
from pyspark.streaming import StreamingContext  
#from pyspark.streaming.kafka import KafkaUtils  
from pyspark import SparkConf, SparkContext  
import json  
import sys  
from pyspark.sql import DataFrame  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import window  
from pyspark.sql.types import StructType, StructField  
from pyspark.sql.types import TimestampType, StringType  
from pyspark.sql.functions import col, column, expr  
  
def KafkaWordCount(zkQuorum, group, topics, numThreads):  
  
    spark = SparkSession \  
        .builder \  
        .appName("KafkaWordCount") \  
        .getOrCreate()  
  
    spark.sparkContext.setLogLevel("ERROR")  
  
    topicAry = topics.split(",")  
    # 将 topic 转换为 hashmap 形式，而 python 中字典就是一种 hashmap  
    topicMap = {}  
    for topic in topicAry:  
        topicMap[topic] = numThreads  
    #lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicMap).map(lambda x :  
    #x[1])  
  
    df = spark \  
        .readStream \  
        .format("kafka") \  
        .option("kafka.bootstrap.servers", "localhost:9092") \  
        .option("subscribe", "sex") \  
        .load()  
    df.selectExpr( "CAST(timestamp AS timestamp)", "CAST(value AS STRING)")  
    #lines = df.selectExpr("CAST(value AS STRING)")  
  
    windowedCounts = df \  
        .withWatermark("timestamp", "1 seconds") \  
        .groupBy(  
            window(col("timestamp"), "1 seconds", "1 seconds"),
```

```

    col("value")) \
    .count()
wind = windowedCounts.selectExpr( "CAST(value AS STRING)", "CAST(count AS
STRING)")

query = wind.writeStream.option("checkpointLocation",
"/check").outputMode("append").foreach(sendmsg).start()

query.awaitTermination()
query.stop()

# 格式转化，将格式变为[{"1": 3}]
def Get_dic(row):
    res = []
    #for elm in row:
    tmp = {row[0]: row[1]}
    res.append(tmp)
    print(res)
    return json.dumps(res)

def sendmsg(row):
    print(row)
    if row.count != 0:
        msg = Get_dic(row)
        # 实例化一个 KafkaProducer 示例，用于向 Kafka 投递消息
        producer = KafkaProducer(bootstrap_servers='localhost:9092')
        producer.send("result", msg.encode('utf8'))
        # 很重要，不然不会更新
        producer.flush()

if __name__ == '__main__':
    # 输入的四个参数分别代表着
    # 1.zkQuorum 为 zookeeper 地址
    # 2.group 为消费者所在的组
    # 3.topics 该消费者所消费的 topics
    # 4.numThreads 开启消费 topic 线程的个数
    if (len(sys.argv) < 5):
        print("Usage: KafkaWordCount <zkQuorum> <group> <topics> <numThreads>")
        exit(1)
    zkQuorum = sys.argv[1]
    group = sys.argv[2]
    topics = sys.argv[3]
    numThreads = int(sys.argv[4])
    print(group, topics)
    KafkaWordCount(zkQuorum, group, topics, numThreads)

```

上述代码做了以下工作：

1. 首先按每秒的频率读取 Kafka 消息；
2. 然后对每秒的数据执行 wordcount 算法，统计出 0 的个数，1 的个数，2 的个数；
3. 最后将上述结果封装成 json 发送给 Kafka。

另外需要注意，上面代码中有一段如下代码：

```
Plaintext  
.option("checkpointLocation", "/check")
```

这行代码表示把检查点文件写入分布式文件系统 HDFS，所以一定要事先启动 Hadoop。如果没有启动 Hadoop，则后面运行时会出现“拒绝连接”的错误提示。如果还没有启动 Hadoop，则可以现在在 Ubuntu 终端中，使用如下 Shell 命令启动 Hadoop：

```
Shell  
cd /usr/local/hadoop #这是 hadoop 的安装目录  
.sbin/start-dfs.sh
```

运行项目

编写好程序之后，接下来编写运行脚本，在 /usr/local/spark/mycode/kafka 目录下新建 startup.sh 文件，输入如下内容：

```
Shell  
/usr/local/spark/bin/spark-submit --packages org.apache.spark:spark-sql-kafka-0-  
10_2.12:3.2.0 /usr/local/spark/mycode/kafka/kafka_test.py 127.0.0.1:2181 1 sex 1
```

其中最后四个为输入参数，含义如下

1. 127.0.0.1:2181 为 Zookeeper 地址
2. 1 为 consumer group 标签
3. sex 为消费者接收的 topic
4. 1 为消费者线程数

最后在 /usr/local/spark/mycode/kafka 目录下，运行如下命令即可执行刚编写好的 Structured Streaming 程序

```
Shell  
sh startup.sh
```

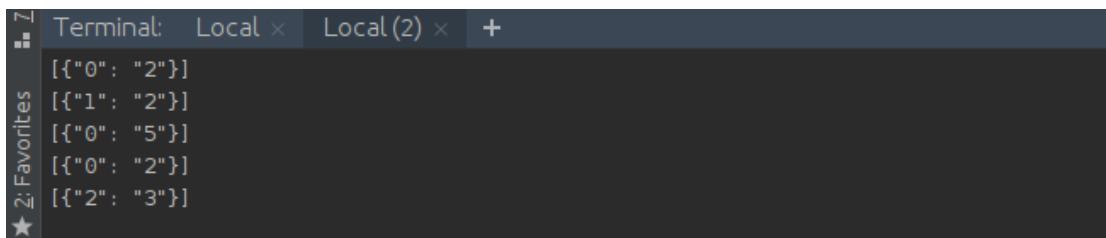
程序运行成功之后，下面通过步骤二的 KafkaProducer 和 KafkaConsumer 来检测程序。

测试程序

下面开启之前编写的 KafkaProducer 投递消息，然后将 KafkaConsumer 中接收的 topic 改为 result，验证是否能接收 topic 为 result 的消息，更改之后的 KafkaConsumer 为

```
Shell  
from kafka import KafkaConsumer  
  
consumer = KafkaConsumer('result')  
for msg in consumer:  
    print((msg.value).decode('utf8'))
```

在同时开启 Structured Streaming 项目，KafkaProducer 以及 KafkaConsumer 之后，可以在 KafkaConsumer 运行窗口看到如下输出：



The screenshot shows a terminal window with two tabs: 'Local' and 'Local (2)'. The 'Local (2)' tab is active and displays the following JSON data:

```
[{"0": "2"}, {"1": "2"}, {"0": "5"}, {"0": "2"}, {"2": "3"}]
```

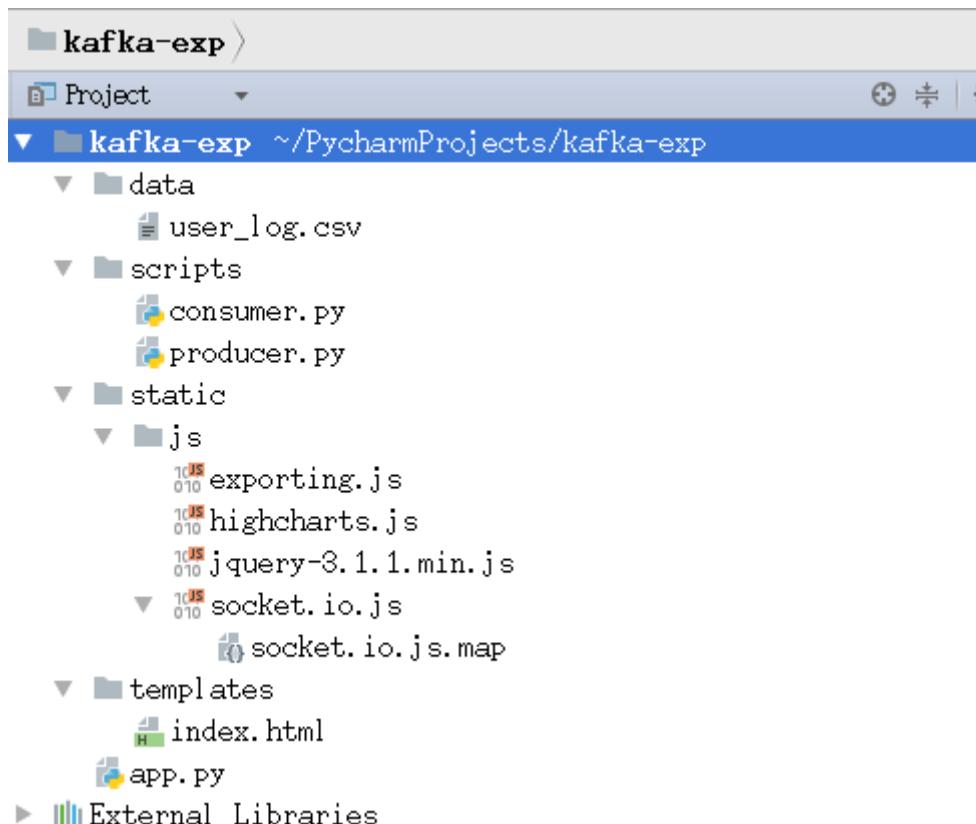
5.2.4.4 结果展示

Flask-SocketIO 实时推送数据

Structured Streaming 实时接收 Kafka 中 topic 为'sex'发送的日志数据，然后 Structured Streaming 进行实时处理，统计好每秒中男女生购物人数之后，将结果发送至 Kafka，topic 为'result'。在本章节，将介绍如何利用 Flask-SocketIO 将结果实时推送到浏览器。

接下来需要使用 Flask-SocketIO，相关文档可以查看 Flask-SocketIO 文档：<https://flask-socketio.readthedocs.io/en/latest/>

项目工程结构图如下



首先创建图中的 app.py 文件,app.py 的功能就是作为一个简易的服务器，处理连接请求，以及处理从 kafka 接收的数据，并实时推送到浏览器。app.py 的代码如下：

```
Python
import json
from flask import Flask, render_template
from flask_socketio import SocketIO
from kafka import KafkaConsumer

# 因为第一步骤安装好了flask，所以这里可以引用

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)
thread = None
# 实例化一个 consumer，接收 topic 为 result 的消息
consumer = KafkaConsumer('result')

# 一个后台线程，持续接收 Kafka 消息，并发送给客户端浏览器
def background_thread():
```

```

girl = 0
boy = 0
for msg in consumer:
    data_json = msg.value.decode('utf8')
    data_list = json.loads(data_json)
    for data in data_list:
        if '0' in data.keys():
            girl = data['0']
        elif '1' in data.keys():
            boy = data['1']
        else:
            continue
    result = str(girl) + ',' + str(boy)
    print(result)
    socketio.emit('test_message', {'data': result})

# 客户端发送 connect 事件时的处理函数
@socketio.on('test_connect')
def connect(message):
    print(message)
    global thread
    if thread is None:
        # 单独开启一个线程给客户端发送数据
        thread = socketio.start_background_task(target=background_thread)
    socketio.emit('connected', {'data': 'Connected'})

# 通过访问 http://127.0.0.1:5000/ 访问 index.html
@app.route("/")
def handle_mes():
    return render_template("index.html")

# main 函数
if __name__ == '__main__':
    socketio.run(app, debug=True)

```

这段代码最重要就是 background_thread 函数，该函数从 Kafka 接收消息，并进行处理，获得男女生每秒钟人数，然后将结果通过函数 socketio.emit 实时推送至浏览器。

浏览器获取数据并展示

index.html 文件负责获取数据并展示效果，该文件中的代码内容如下：

```

HTML
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>DashBoard</title>
    <script src="static/js/socket.io.js"></script>
    <script src="static/js/jquery-3.1.1.min.js"></script>
    <script src="static/js/highcharts.js"></script>
    <script src="static/js/exporting.js"></script>
    <script type="text/javascript" charset="utf-8">
        var socket = io.connect('http://' + document.domain + ':' + location.port);
        socket.on('connect', function() {

```

```

socket.emit('test_connect', {data: 'I\'m connected!'});

});

socket.on('test_message',function(message){
  console.log(message);
  var obj = eval(message);
  var result = obj["data"].split(",");
  $('#girl').html(result[0]);
  $('#boy').html(result[1]);
});

socket.on('connected',function(){
  console.log('connected');
});

socket.on('disconnect', function () {
  console.log('disconnect');
});
</script>
</head>
<body>
<div>
  <b>Girl: </b><b id="girl"></b>
  <b>Boy: </b><b id="boy"></b>
</div>
<div id="container" style="width: 600px;height:400px;"></div>

<script type="text/javascript">
$(document).ready(function () {
  Highcharts.setOptions({
    global: {
      useUTC: false
    }
  });

  Highcharts.chart('container', {
    chart: {
      type: 'spline',
      animation: Highcharts.svg, // don't animate in old IE
      marginRight: 10,
      events: {
        load: function () {

          // set up the updating of the chart each second
          var series1 = this.series[0];
          var series2 = this.series[1];
          setInterval(function () {
            var x = (new Date()).getTime(), // current time
            count1 = $('#girl').text();
            y = parseInt(count1);
            series1.addPoint([x, y], true, true);

            count2 = $('#boy').text();
            z = parseInt(count2);
            series2.addPoint([x, z], true, true);
          }, 1000);
        }
      }
    }
  });
});

```

```

},
title: {
    text: '男女生购物人数实时分析'
},
xAxis: {
    type: 'datetime',
    tickPixelInterval: 50
},
yAxis: {
    title: {
        text: '数量'
    },
    plotLines: [{
        value: 0,
        width: 1,
        color: '#808080'
    }]
},
tooltip: {
    formatter: function () {
        return '<b>' + this.series.name + '</b><br/>' +
            Highcharts.dateFormat('%Y-%m-%d %H:%M:%S', this.x) + '<br/>' +
            Highcharts.numberFormat(this.y, 2);
    }
},
legend: {
    enabled: true
},
exporting: {
    enabled: true
},
series: [{
    name: '女生购物人数',
    data: (function () {
        // generate an array of random data
        var data = [],
            time = (new Date()).getTime(),
            i;

        for (i = -19; i <= 0; i += 1) {
            data.push({
                x: time + i * 1000,
                y: Math.random()
            });
        }
        return data;
    }())
},
{
    name: '男生购物人数',
    data: (function () {
        // generate an array of random data
        var data = [],
            time = (new Date()).getTime(),
            i;

        for (i = -19; i <= 0; i += 1) {

```

```

        data.push({
            x: time + i * 1000,
            y: Math.random()
        });
    }
    return data;
})()
])
});
);
</script>
</body>
</html>

```

可以看到，在上面给出的 index.html 文件的开头部分，包含如下几行代码：

HTML

```

<script src="static/js/socket.io.js"></script>
<script src="static/js/jquery-3.1.1.min.js"></script>
<script src="static/js/highcharts.js"></script>
<script src="static/js/exporting.js"></script>

```

也就是说，在 index.html 中，需要用到几个 js 库，包括 socket.io.js、jquery-3.1.1.min.js、highcharts.js 和 exporting.js。

socket.io.js

客户端浏览器需要使用 js 框架 socket.io.js 来实时接收服务端的消息，该 js 框架用法和 Flask-SocketIO 使用类似。首先我们可以看到在工程目录中有两个 js 库文件跟 socket.io.js 相关，一个就是 socket.io.js，下载链接：<https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js>；还有一个文件就是 socket.io.js.map，下载链接：<https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js.map>。

下载好 socket.io.js 和 socket.io.js.map 这两个 js 库文件后，按照上面给出的工程文件目录结构，把这两个文件复制到 js 文件夹下。

然后，可以看到在 index.html 中是如何调用 socket.io.js 和 socket.io.js.map 这两个 js 库文件，在 index.html 中包含了如下一段代码，就是用来调用这两个库文件的：

```

JavaScript
<script type="text/javascript" charset="utf-8">
    // 创建连接服务器的链接
    var socket = io.connect('http://' + document.domain + ':' + location.port);
    socket.on('connect', function() { // 连上服务器后的回调函数
        socket.emit('connect', {data: 'I\'m connected!'});
    });
    // 接收服务器实时发送的数据
    socket.on('test_message', function(message){
        console.log(message);
        var obj = eval(message);
        var result = obj["data"].split(",");
        // 将男生和女生人数展示在 html 标签内
        $('#girl').html(result[0]);
        $('#boy').html(result[1]);
    });

    socket.on('connected', function(){
        console.log('connected');
    });

```

```
// 链接断开时的回调函数
socket.on('disconnect', function () {
  console.log('disconnect');
});
</script>
```

上面这段代码就是使用 socket.io.js 库来实时地接收服务端发送过来的消息，并将消息数据实时地设置在 html 标签内，交给 highcharts.js 进行实时获取和展示。

highchart.js

highcharts.js 是一个用纯 JavaScript 编写的一个图表库。能够很简单便捷地在 Web 网站或是 Web 应用程序中添加有交互性的图表。可以到官网下载最新版本的 highchart.js 库文件(下载链接:

<https://www.highcharts.com/download>)。注意，到官网下载 highchart.js 库文件时，下载到本地的是一个类似 Highcharts-6.0.7.zip 这样的压缩文件，对这个文件进行解压缩，可以看到里面有个 code 子目录，在 code 子目录下面就可以找到 highchart.js 库文件，按照上面给出的工程文件目录结构，把这个 highchart.js 库文件复制到 IntelliJIDEA 工程目录中的 js 文件夹下。

在 index.html 中包含如下一段代码，就是调用 highcharts.js 库，来实时地从 html 标签内获取数据并展示在网页中。

```
JavaScript
<script type="text/javascript">
$(document).ready(function () {
Highcharts.setOptions({
  global: {
    useUTC: false
  }
});

Highcharts.chart('container', {
  chart: {
    type: 'spline',
    animation: Highcharts.svg, // 这个在 ie 浏览器可能不支持
    marginRight: 10,
    events: {
      load: function () {

        //设置图表每秒更新一次
        var series1 = this.series[0];
        var series2 = this.series[1];
        setInterval(function () {
          var x = (new Date()).getTime(); // 获取当前时间
          count1 = $('#girl').text();
          y = parseInt(count1);
          series1.addPoint([x, y], true, true);

          count2 = $('#boy').text();
          z = parseInt(count2);
          series2.addPoint([x, z], true, true);
        }, 1000);
      }
    },
    title: { // 设置图表名
      text: '男女生购物人数实时分析'
    },
  }
});
```

```

xAxis: { //x 轴设置为实时时间
    type: 'datetime',
    tickPixelInterval: 50
},
yAxis: {
    title: {
        text: '数量'
    },
    plotLines: [{ //设置坐标线颜色粗细
        value: 0,
        width: 1,
        color: '#808080'
    }]
},
tooltip: {
    //规范显示时间的格式
    formatter: function () {
        return '<b>' + this.series.name + '</b><br/>' +
            Highcharts.dateFormat('%Y-%m-%d %H:%M:%S', this.x) + '<br/>' +
            Highcharts.numberFormat(this.y, 2);
    }
},
legend: {
    enabled: true
},
exporting: {
    enabled: true
},
series: [
    {
        name: '女生购物人数',
        data: (function () {
            // 随机方式生成初始值填充图表
            var data = [],
                time = (new Date()).getTime(),
                i;

            for (i = -19; i <= 0; i += 1) {
                data.push({
                    x: time + i * 1000,
                    y: Math.random()
                });
            }
            return data;
        }())
    },
    {
        name: '男生购物人数',
        data: (function () {
            // 随机方式生成初始值填充图表
            var data = [],
                time = (new Date()).getTime(),
                i;

            for (i = -19; i <= 0; i += 1) {
                data.push({
                    x: time + i * 1000,
                    y: Math.random()
                });
            }
            return data;
        }())
    }
]
}

```

```
    });
  }
  return data;
}
]);
});
});
```

这段代码是在设置表格的格式，因为 highchart.js 这个框架，让在网页中显示数据图表的工作变得很轻松。

exporting.js

在上面，到官网下载 highchart.js 库文件时，下载到本地的是一个类似 Highcharts-6.0.7.zip 这样的压缩文件，对这个文件进行解压缩，可以看到里面有个 code 子目录，在 code 子目录下面就可以找到一个 js 子目录，在 js 子目录下可以找到一个 modules 子目录，在 modules 子目录中就可以找到库文件 exporting.js。然后，按照上面给出的工程文件目录结构，把这个 exporting.js 库文件复制到 IntelliJIDEA 工程目录中的 js 文件夹下。

jquery.js

除了上述三个 js 库文件外，还有一个 jquery.js 库文件。jQuery 是一个快速、简洁的 JavaScript 框架，它封装 JavaScript 常用的功能代码，提供一种简便的 JavaScript 设计模式，优化 HTML 文档操作、事件处理、动画设计和 Ajax 交互。[点击这里下载 jquery.js](#)，然后，按照上面给出的工程文件目录结构，把这个 jquery.js 库文件复制到 IntelliJIDEA 工程目录中的 js 文件夹下。

效果展示

经过以上步骤，就可以启动程序来看看最后的效果。

- 1.确保 kafka 开启。
- 2.开启 producer.py 模拟数据流。
- 3.启动 Structured Streaming 实时处理数据。可以在实时处理数据启动之后，把 consumer.py 的 topic 改成 result，运行 consumer.py 就可以看到数据处理后的输出结果。
- 4.启动 app.py。如果你是使用 pycharm 客户端，那右键就可以了。当然也可以使用终端命令：

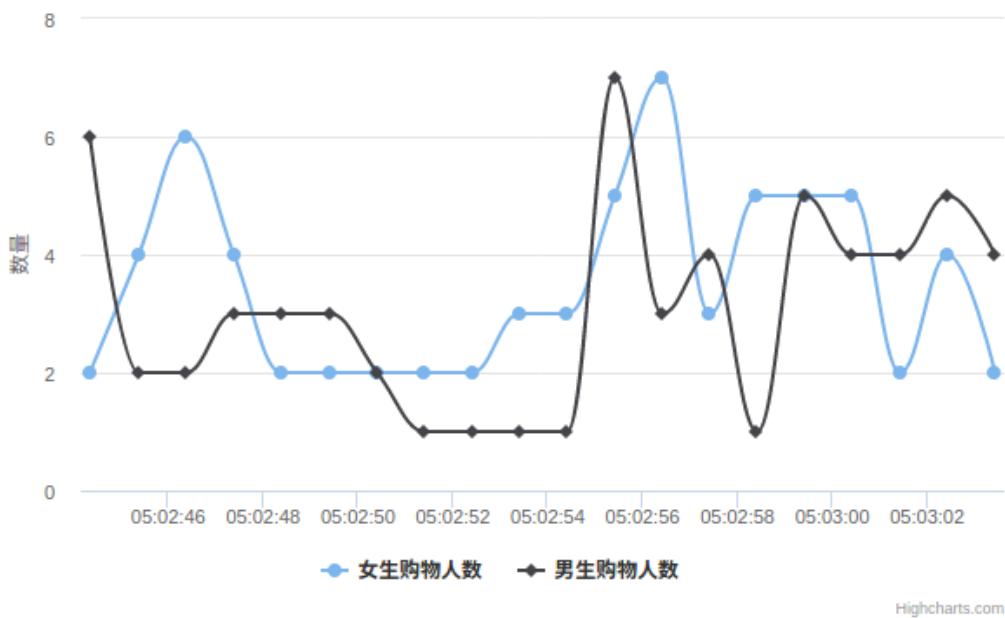
```
Shell
python app.py
```

这时候你可以用浏览器访问 <http://127.0.0.1:5000/>，就可以看到最终效果图了。

本案例的最终效果如下所示：

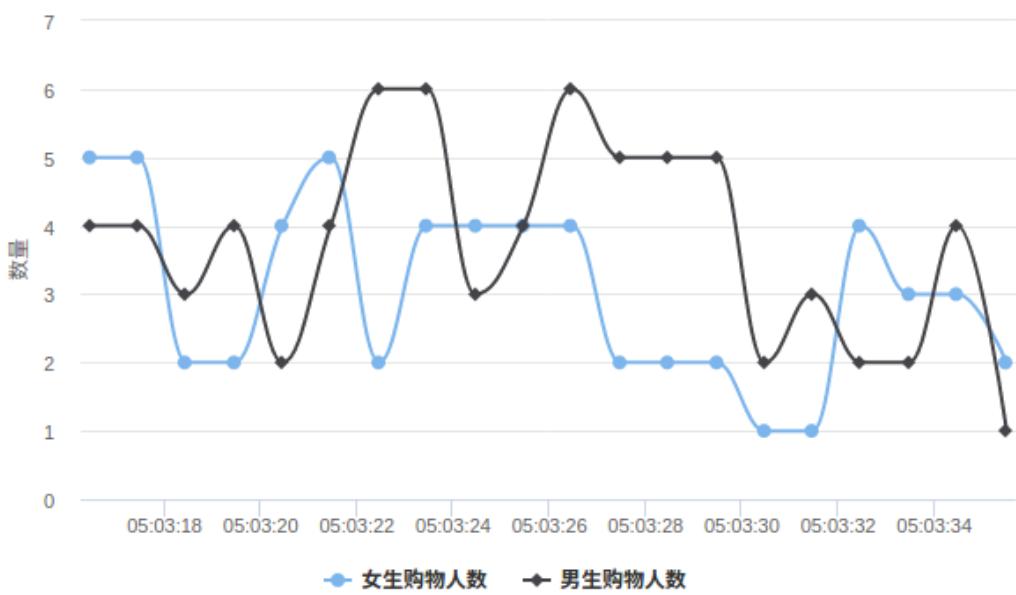
Girl: 2 Boy: 5

男女生购物人数实时分析



Girl: 3 Boy: 5

男女生购物人数实时分析



5.3 Flume：高可用拓扑结构实践

5.3.1 Flume 拓扑部署案例

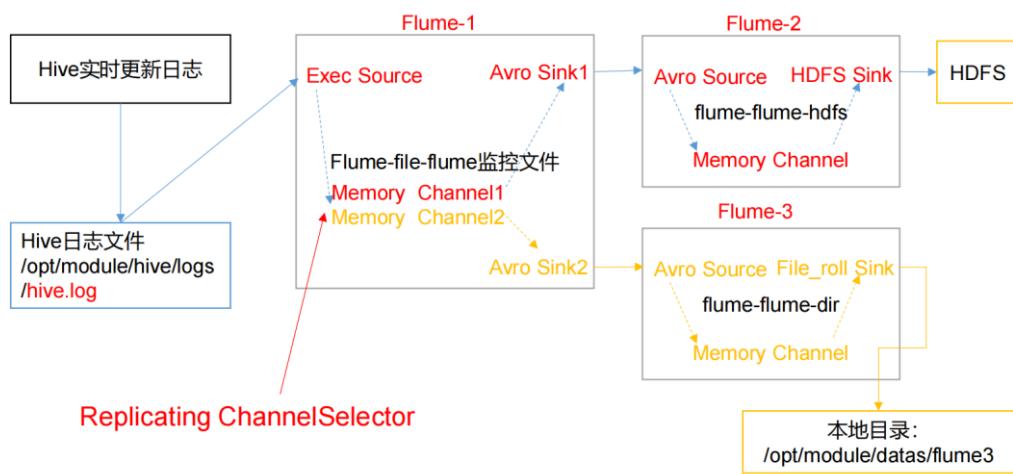
5.3.1.1 Flume 复制案例

1) 案例需求

使用 Flume1 监控文件变动，Flume1 将变动内容传递给 Flume2，Flume2 负责存储到 HDFS。同时 Flume1 将

变动内容传递给 Flume3，Flume3 负责输出到 Local FileSystem。

2) 需求分析



3) 实现步骤

(1) 准备工作

在/opt/module/flume/job 目录下创建 group1 文件夹，存放配置文件

```
Bash  
cd /opt/module/flume/job  
mkdir group1
```

在/opt/module/datas 目录下创建 flume3 文件夹，存放 Flume3 的输出

```
Bash  
cd /opt/module/datas  
mkdir flume3
```

(2) 配置 Flume1

在 /opt/module/flume/job/group1 目录下创建 Flume1 的配置文件 flume-file-flume.conf。编辑配置文件，为 Flume1 配置 1 个接收日志文件的 source 和两个 channel、两个 sink，分别输送给 Flume2 和 Flume3。

```
Bash  
cd /opt/module/flume/job/group1  
touch flume-file-flume.conf  
vim flume-file-flume.conf
```

配置文件内容如下：

```
Bash  
# Name the components on this agent  
a1.sources = r1  
a1.sinks = k1 k2  
a1.channels = c1 c2  
# 将数据流复制给所有 channel  
a1.sources.r1.selector.type = replicating  
  
# Describe/configure the source  
a1.sources.r1.type = exec
```

```

a1.sources.r1.command = tail -F /opt/module/hive/logs/hive.log
a1.sources.r1.shell = /bin/bash -c

# Describe the sink
# sink 端的 avro 是一个数据发送者
a1.sinks.k1.type = avro
a1.sinks.k1.hostname = hadoop102
a1.sinks.k1.port = 4141
a1.sinks.k2.type = avro
a1.sinks.k2.hostname = hadoop102
a1.sinks.k2.port = 4142

# Describe the channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
a1.channels.c2.type = memory
a1.channels.c2.capacity = 1000
a1.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1 c2
a1.sinks.k1.channel = c1
a1.sinks.k2.channel = c2

```

属性说明：

	Property Name	Description
Exec Source	type	The component type name, needs to be exec
	command	The command to execute
Avro Sink	type	The component type name, needs to be avro.
	hostname	The hostname or IP address to bind to.
	port	The port # to listen on.

(3) 配置 Flume2

在 /opt/module/flume/job/group1 目录下创建 Flume2 的配置文件 flume-flume-hdfs.conf。编辑配置文件，配置上级 Flume 输出的 Source，输出是到 HDFS 的 Sink。

```

Bash
touch flume-flume-hdfs.conf
vim flume-flume-hdfs.conf

```

配置文件内容如下：

```

Bash
# Name the components on this agent
a2.sources = r1
a2.sinks = k1
a2.channels = c1

```

```

# Describe/configure the source
# source 端的 avro 是一个数据接收服务
a2.sources.r1.type = avro
a2.sources.r1.bind = hadoop102
a2.sources.r1.port = 4141

# Describe the sink
a2.sinks.k1.type = hdfs
a2.sinks.k1.hdfs.path = hdfs://hadoop102:9820/flume2/%Y%m%d%H
#上传文件的前缀
a2.sinks.k1.hdfs.filePrefix = flume2-
#是否按照时间滚动文件夹
a2.sinks.k1.hdfs.round = true
#多少时间单位创建一个新的文件夹
a2.sinks.k1.hdfs.roundValue = 1
#重新定义时间单位
a2.sinks.k1.hdfs.roundUnit = hour
#是否使用本地时间戳
a2.sinks.k1.hdfs.useLocalTimeStamp = true
#积攒多少个 Event 才 flush 到 HDFS 一次
a2.sinks.k1.hdfs.batchSize = 100
#设置文件类型，可支持压缩
a2.sinks.k1.hdfs fileType = DataStream
#多久生成一个新的文件
a2.sinks.k1.hdfs.rollInterval = 30
#设置每个文件的滚动大小大概是 128M
a2.sinks.k1.hdfs.rollSize = 134217700
#文件的滚动与 Event 数量无关
a2.sinks.k1.hdfs.rollCount = 0

# Describe the channel
a2.channels.c1.type = memory
a2.channels.c1.capacity = 1000
a2.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a2.sources.r1.channels = c1
a2.sinks.k1.channel = c1

```

属性说明：

	Property Name	Description
Avro Source	type	The component type name, needs to be avro
	bind	hostname or IP address to listen on
	port	Port # to bind to
HDFS Sink	type	The component type name, needs to be hdfs
	hdfs.path	HDFS directory path (eg hdfs://namenode/flume/webdata/)

	hdfs.filePrefix	Name prefixed to files created by Flume in hdfs directory
	hdfs.round	Should the timestamp be rounded down (if true, affects all time based escape sequences except %t)
	hdfs.roundValue	Rounded down to the highest multiple of this (in the unit configured using hdfs.roundUnit), less than current time.
	hdfs.roundUnit	The unit of the round down value - second, minute or hour.
	hdfs.useLocalTimeStamp	Use the local time (instead of the timestamp from the event header) while replacing the escape sequences.
	hdfs.batchSize	number of events written to file before it is flushed to HDFS
	hdfs.fileType	File format: currently SequenceFile, DataStream or CompressedStream (1)DataStream will not compress output file and please don't set codeC (2)CompressedStream requires set hdfs.codeC with an available codeC
	hdfs.rollInterval	Number of seconds to wait before rolling current file (0 = never roll based on time interval)
	hdfs.rollSize	File size to trigger roll, in bytes (0: never roll based on file size)
	hdfs.rollCount	Number of events written to file before it rolled (0 = never roll based on number of events)

(4) 配置 Flume3

在 /opt/module/flume/job/group1 目录下创建 Flume3 的配置文件 flume-flume-dir.conf。编辑配置文件，配置上级 Flume 输出的 Source，输出是到本地目录的 Sink。

```
Bash
touch flume-flume-dir.conf
vim flume-flume-dir.conf
```

配置文件内容如下：

```
Bash
# Name the components on this agent
a3.sources = r1
a3.sinks = k1
```

```

a3.channels = c2

# Describe/configure the source
a3.sources.r1.type = avro
a3.sources.r1.bind = hadoop102
a3.sources.r1.port = 4142

# Describe the sink
a3.sinks.k1.type = file_roll
a3.sinks.k1.sink.directory = /opt/module/data/flume3

# Describe the channel
a3.channels.c2.type = memory
a3.channels.c2.capacity = 1000
a3.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel
a3.sources.r1.channels = c2
a3.sinks.k1.channel = c2

```

属性说明：

	Property Name	Description
Avro Source	type	The component type name, needs to be avro
	bind	hostname or IP address to listen on
	port	Port # to bind to
File Roll Sink	type	The component type name, needs to be file_roll.
	sink.directory	The directory where files will be stored

注意： 输出的本地目录必须是已经存在的目录，如果该目录不存在，并不会创建新的目录。我们在准备阶段已提前创建好 /opt/module/data/flume3 作为输出目录。

(5) 执行配置文件

在 flume 根目录下分别启动对应的 flume 进程：flume-flume-dir, flume-flume-hdfs, flume-file-flume

Bash

```

bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group1/flume-flume-dir.conf
bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group1/flume-flume-hdfs.conf
bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group1/flume-file-flume.conf

```

(6) 启动 Hadoop 和 Hive

Bash

```

[myUbuntu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh
[myUbuntu@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh
[myUbuntu@hadoop102 hive]$ bin/hive
hive (default)>

```

(7) 检查 HDFS 上的数据

Browse Directory

/flume2/20180522/00								Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
-rw-r--r--	atguigu	supergroup	1.13 KB	2018/5/22 上午12:08:32	3	128 MB	flume2-1526918911367.tmp	

(8) 检查/opt/module/datas/flume3 目录中数据

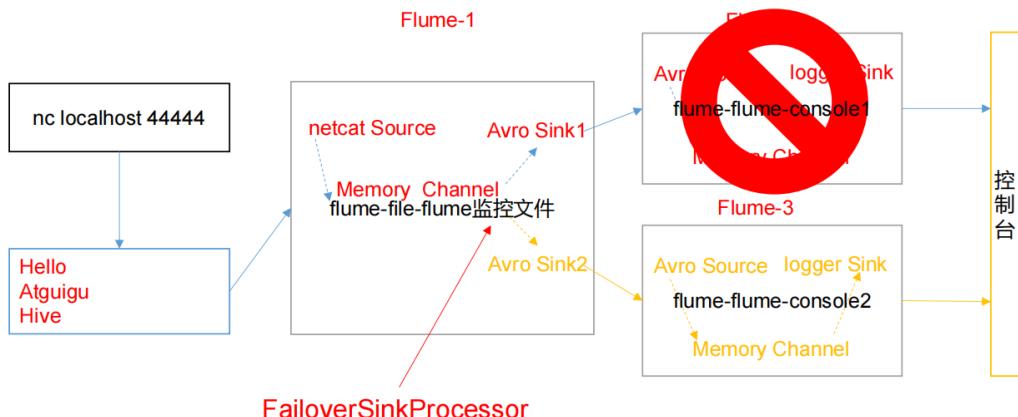
```
Bash
[myUbuntu@hadoop102 flume3]$ ll
总用量 8
-rw-rw-r--. 1 myUbuntu myUbuntu 5942 5 月 22 00:09 1526918887550-3
```

5.3.1.2 Flume 负载均衡和故障转移

1) 案例需求

使用 Flume1 监控一个端口，其 sink 组中的 sink 分别对接 Flume2 和 Flume3，采用 FailoverSinkProcessor，实现故障转移的功能。

2) 需求分析



3) 实现步骤

(一) 故障转移

(1) 准备工作

在/opt/module/flume/job 目录下创建 group2 文件夹，存放配置文件

(2) 配置 Flume1

在 /opt/module/flume/job/group2 目录下创建 Flume1 的配置文件 flume-netcat-flume.conf。编辑配置文件，为 Flume1 配置 1 个 netcat source 和 1 个 channel、1 个 sink group（2 个 sink），分别输送给 Flume2 和 Flume3。

配置文件内容如下：

```
Bash
# Name the components on this agent
```

```

a1.sources = r1
a1.channels = c1
a1.sinkgroups = g1
a1.sinks = k1 k2

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Sink Group
a1.sinkgroups.g1.processor.type = failover
a1.sinkgroups.g1.processor.priority.k1 = 5
a1.sinkgroups.g1.processor.priority.k2 = 10
a1.sinkgroups.g1.processor.maxpenalty = 10000

# Describe the sink
a1.sinks.k1.type = avro
a1.sinks.k1.hostname = hadoop102
a1.sinks.k1.port = 4141
a1.sinks.k2.type = avro
a1.sinks.k2.hostname = hadoop102
a1.sinks.k2.port = 4142

# Describe the channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinks.k1.channel = c1
a1.sinks.k2.channel = c1

```

	Property Name	Description
NetCat TCP Source	type	The component type name, needs to be netcat
	bind	Host name or IP address to bind to
	port	Port # to bind to
Failover Sink Processor	sinks	Space-separated list of sinks that are participating in the group
	processor.type	The component type name, needs to be failover
	processor.priority.<sinkName>	Priority value. <sinkName> must be one of the sink instances associated with the current sink group A higher priority value Sink gets activated earlier. A larger

		absolute value indicates higher priority
	processor.maxpenalty	The maximum backoff period for the failed Sink (in millis)
Avro Sink	type	The component type name, needs to be avro.
	hostname	The hostname or IP address to bind to.
	port	The port # to listen on.

(3) 配置 Flume2

在 /opt/module/flume/job/group2 目录下创建 Flume2 的配置文件 flume-flume-console1.conf。编辑配置文件，配置上级 Flume 输出的 Source，输出是到本地控制台。

配置文件内容如下：

```
Bash
# Name the components on this agent
a2.sources = r1
a2.sinks = k1
a2.channels = c1

# Describe/configure the source
a2.sources.r1.type = avro
a2.sources.r1.bind = hadoop102
a2.sources.r1.port = 4141

# Describe the sink
a2.sinks.k1.type = logger

# Describe the channel
a2.channels.c1.type = memory
a2.channels.c1.capacity = 1000
a2.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a2.sources.r1.channels = c1
a2.sinks.k1.channel = c1
```

(4) 配置 Flume3

在 /opt/module/flume/job/group2 目录下创建 Flume3 的配置文件 flume-flume-console2.conf。编辑配置文件，配置上级 Flume 输出的 Source，输出是到本地控制台。

配置文件内容如下：

```
Bash
# Name the components on this agent
a3.sources = r1
a3.sinks = k1
a3.channels = c2

# Describe/configure the source
a3.sources.r1.type = avro
```

```

a3.sources.r1.bind = hadoop102
a3.sources.r1.port = 4142

# Describe the sink
a3.sinks.k1.type = logger

# Describe the channel
a3.channels.c2.type = memory
a3.channels.c2.capacity = 1000
a3.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel
a3.sources.r1.channels = c2
a3.sinks.k1.channel = c2

```

(5) 执行配置文件

```

hadoop@ubuntu:/usr/local/flume/job/group2$ vim flume11-netcat-flume.conf
hadoop@ubuntu:/usr/local/flume/job/group2$ vim flume2-flume-console1.conf
hadoop@ubuntu:/usr/local/flume/job/group2$ vim flume3-flume-console2.conf

```

在 flume 根目录下分别启动对应的 flume 进程： flume-flume-console2， flume-flume-console1， flume-netcat-flume。

Bash

```

bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group2/flume-flume-
console2.conf -Dflume.root.logger=INFO,console
bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group2/flume-flume-
console1.conf -Dflume.root.logger=INFO,console
bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group2/flume-netcat-flume.conf

```

```

hadoop@ubuntu:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group2/flume3-flume-console2.conf
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access

```

```

hadoop@ubuntu:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group2/flume2-flume-console1.conf
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access

```

```

hadoop@ubuntu:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group2/flume11-netcat-flume.conf
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access

```

(6) 使用 netcat 工具向本机的 44444 端口发送内容

Bash

```
nc localhost 44444
```

```

hadoop@ubuntu:~$ nc localhost 44444
hi
OK
failover
OK
|

```

(7) 查看 Flume2 及 Flume3 的控制台打印日志

Flume2 无输出：

```

2023-06-20 08:53:51,512 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:207)] Starting new configuration: { sourceRunners:{r1=EventDrivenSourceRunner: { source:Avro source r1: { bindAddress: ubuntu, port: 4141 } }} sinkRunners:{k1=SinkRunner: { policy:org.apache.flume.sink.DefaultSinkProcessor@lebea008 counterGroup:{ name:null counters:{} } } } channels:{c1=org.apache.flume.channel.MemoryChannel{name: c1} }
2023-06-20 08:53:51,513 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:214)] Starting Channel c1
2023-06-20 08:53:51,517 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:229)] Waiting for channel: c1 to start. Sleeping for 500 ms
2023-06-20 08:53:51,519 [lifecycleSupervisor-1-0] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.register(MonitoredCounterGroup.java:119)] Monitored counter group for type: CHANNEL, name: c1: Successfully registered new MBean.
2023-06-20 08:53:51,519 [lifecycleSupervisor-1-0] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.start(MonitoredCounterGroup.java:95)] Component type: CHANNEL, name: c1 started
2023-06-20 08:53:52,018 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:241)] Starting Sink k1
2023-06-20 08:53:52,019 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:252)] Starting Source r1
2023-06-20 08:53:52,019 [lifecycleSupervisor-1-2] [INFO - org.apache.flume.source.AvroSource.start(AvroSource.java:184)] Starting Avro source r1: { bindAddress: ubuntu, port: 4141 }...
2023-06-20 08:53:52,512 [lifecycleSupervisor-1-2] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.register(MonitoredCounterGroup.java:119)] Monitored counter group for type: SOURCE, name: r1: Successfully registered new MBean.
2023-06-20 08:53:52,512 [lifecycleSupervisor-1-2] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.start(MonitoredCounterGroup.java:95)] Component type: SOURCE, name: r1 started
2023-06-20 08:53:52,517 [lifecycleSupervisor-1-2] [INFO - org.apache.flume.source.AvroSource.start(AvroSource.java:223)] Avro source r1 started.

```

Flume3:

```

2023-06-20 08:50:55,307 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:241)] Starting Sink k1
2023-06-20 08:50:55,308 [main] [INFO - org.apache.flume.node.Application.startAllComponents(Application.java:252)] Starting Source r1
2023-06-20 08:50:55,308 [lifecycleSupervisor-1-1] [INFO - org.apache.flume.source.AvroSource.start(AvroSource.java:184)] Starting Avro source r1: { bindAddress: ubuntu, port: 4142 }...
2023-06-20 08:50:55,921 [lifecycleSupervisor-1-1] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.register(MonitoredCounterGroup.java:119)] Monitored counter group for type: SOURCE, name: r1: Successfully registered new MBean.
2023-06-20 08:50:55,922 [lifecycleSupervisor-1-1] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.start(MonitoredCounterGroup.java:95)] Component type: SOURCE, name: r1 started
2023-06-20 08:50:55,925 [lifecycleSupervisor-1-1] [INFO - org.apache.flume.source.AvroSource.start(AvroSource.java:223)] Avro source r1 started.
2023-06-20 08:55:10,601 [SinkRunner-PollingRunner-DefaultSinkProcessor] [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 69
hi }
2023-06-20 08:57:00,623 [SinkRunner-PollingRunner-DefaultSinkProcessor] [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 66 61 69 6C 6F 76 65 72
failover }

```

(8) 将 Flume3 kill，观察 Flume2 的控制台打印情况

Kill Flume3:

```

hadoop@ubuntu:/usr/local/flume$ jobs
[1]+ Stopped bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group2/flume3-flume-console2.conf
hadoop@ubuntu:/usr/local/flume$ kill %1
[1]+ Stopped bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group2/flume3-flume-console2.conf
hadoop@ubuntu:/usr/local/flume$ 2023-06-20 09:04:25,062 (agent-shutdown-hook) [INFO - org.apache.flume.node.Application.stopAllComponents(Application.java:140)] Shutting down configuration: { sourceRunners:{r1:EventDrivenSourceRunner: { source:Avro source r1: { bindAddress: u
buntu, port: 4142 } }} sinkRunners:{k1=SinkRunner: { policy:org.apache.flume.sink.DefaultSinkProcessor@lebea008 counterGroup:{ name:null counters:{runner.backoffs.consecutive=53, runner.backoffs=101} } } } channels:{c2=org.apache.flume.channel.MemoryChannel{name: c2} }
2023-06-20 09:04:25,063 (agent-shutdown-hook) [INFO - org.apache.flume.node.Application.stopAllComponents(Application.java:144)] Stopping Source r1

```

使用 netcat 工具向本机的 44444 端口发送内容：

```

hadoop@ubuntu:~$ nc localhost 44444
hi
OK
failover
OK
hello
OK
failover
OK

```

Flume2:

```

2023-06-20 08:53:52,512 [lifecycleSupervisor-1-2] [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.start(MonitoredCounterGroup.java:95)] Component type: SOURCE, name: r1 started
2023-06-20 08:53:52,517 [lifecycleSupervisor-1-2] [INFO - org.apache.flume.source.AvroSource.start(AvroSource.java:223)] Avro source r1 started.
2023-06-20 09:06:14,155 [SinkRunner-PollingRunner-DefaultSinkProcessor] [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 65 6C 6F
hello }
2023-06-20 09:06:30,888 [SinkRunner-PollingRunner-DefaultSinkProcessor] [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 66 61 69 6C 6F 76 65 72
failover }

```

(二) 负载均衡

只需更改 Flume1 的 Sink Group 配置，其余与故障转移流程相同。

```

hadoop@ubuntu:/usr/local/flume/job/group2$ vim flume12-netcat-flume.conf
hadoop@ubuntu:/usr/local/flume/job/group2$ cd ..
hadoop@ubuntu:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group2/flume12-netcat-flume.conf
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access

```

```

Bash
# Sink Group
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = load_balance
a1.sinkgroups.g1.processor.backoff = true
a1.sinkgroups.g1.processor.selector = random

```

	Property Name	Description
Load balancing Sink Processor	processor.sinks	Space-separated list of sinks that are participating in the group
	processor.type	The component type name, needs to be load_balance
	processor.backoff	Should failed sinks be backed off exponentially.
	processor.selector	Selection mechanism. Must be either round_robin, random or FQCN of custom class that inherits from AbstractSinkSelector

使用 netcat 工具向本机的 44444 端口发送内容:

```

hadoop@ubuntu:~$ nc localhost 44444
hello1
OK
hello2
OK
hello3
OK
hello4
OK
hello5
OK
hello6
OK
hello7
OK
hello8
OK
hello9
OK
hello10
OK

```

Flume2:

```

2023-06-20 09:20:50,255 (nioEventLoopGroup-3-1) [INFO - org.apache.avro.ipc.netty.NettyServer$NettyServerAvroHandler.channelInactive(Netty
Server.java:181)] Connection to /127.0.0.1:5424 disconnected.
2023-06-20 09:21:32,963 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 31 hello1 }
2023-06-20 09:21:32,963 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 32 hello2 }
2023-06-20 09:21:32,964 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 33 hello3 }
2023-06-20 09:21:32,964 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 34 hello4 }
2023-06-20 09:21:32,964 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 35 hello5 }
2023-06-20 09:21:49,937 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 36 hello6 }
2023-06-20 09:22:11,943 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 38 hello8 }
2023-06-20 09:22:11,943 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 39 hello9 }
2023-06-20 09:22:11,943 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 31 30 hello10 }

```

Flume3:

```

2023-06-20 09:20:50,285 (nioEventLoopGroup-3-1) [INFO - org.apache.avro.ipc.netty.NettyServer$NettyServerAvroHandler.channelInactive(Netty
Server.java:181)] Connection to /127.0.0.1:45616 disconnected.
2023-06-20 09:21:55,161 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:9
5)] Event: { headers:{} body: 68 65 6C 6F 37 hello7 }

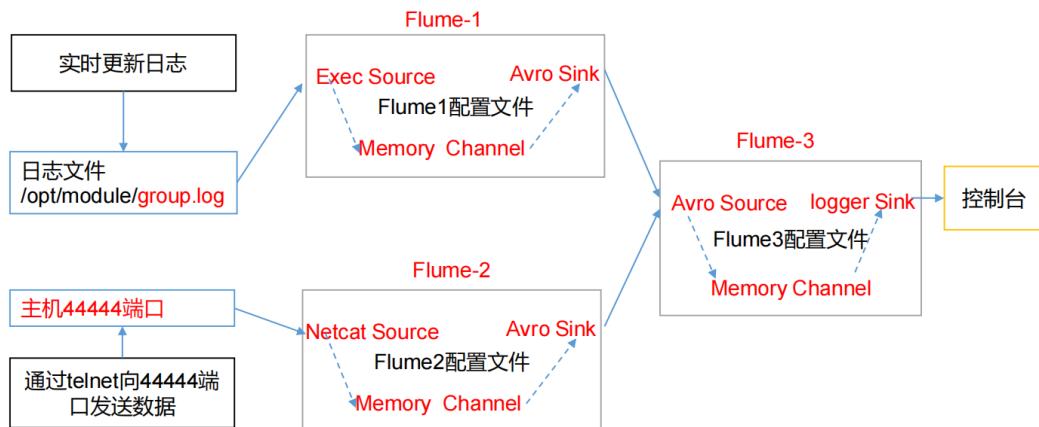
```

5.3.1.3 Flume 聚合案例

1) 案例需求

- hadoop102 上的 Flume1 监控文件/opt/module/group.log
- hadoop103 上的 Flume2 监控某一个端口的数据流，
- Flume1 与 Flume2 将数据发送给 hadoop104 上的 Flume3，Flume3 将最终数据打印到控制台

2) 需求分析



3) 实现步骤

(1) 准备工作

分发 flume

```

Bash
[myUbuntu@hadoop102 module]$ xsync flume

```

分别在 hadoop102、hadoop103 以及 hadoop104 的 /opt/module/flume/job 目录下创建 group3 文件夹

```

Bash
[myUbuntu@hadoop102 job]$ mkdir group3
[myUbuntu@hadoop103 job]$ mkdir group3
[myUbuntu@hadoop104 job]$ mkdir group3

```

(2) 配置 Flume1

在 hadoop102 的 /opt/module/flume/job/group3 目录下创建 Flume1 的配置文件 flume1-logger-flume.conf 。编辑配置文件，为 Flume1 配置 Source 用于监控 hive.log 文件，配置 Sink 输出数据到下一级 Flume。

Bash

```
[myUbuntu@hadoop102 group3]$ touch flume1-logger-flume.conf  
[myUbuntu@hadoop102 group3]$ vim flume1-logger-flume.conf
```

```
hadoop@hadoop1:/usr/local/flume/job/group3$ touch flume1-logger-flume.conf  
hadoop@hadoop1:/usr/local/flume/job/group3$ vim flume1-logger-flume.conf
```

配置文件内容如下：

Bash

```
# Name the components on this agent  
a1.sources = r1  
a1.sinks = k1  
a1.channels = c1  
  
# Describe/configure the source  
a1.sources.r1.type = exec  
a1.sources.r1.command = tail -F /opt/module/group.log  
a1.sources.r1.shell = /bin/bash -c  
  
# Describe the sink  
a1.sinks.k1.type = avro  
a1.sinks.k1.hostname = hadoop104  
a1.sinks.k1.port = 4141  
  
# Describe the channel  
a1.channels.c1.type = memory  
a1.channels.c1.capacity = 1000  
a1.channels.c1.transactionCapacity = 100  
  
# Bind the source and sink to the channel  
a1.sources.r1.channels = c1  
a1.sinks.k1.channel = c1
```

(3) 配置 Flume2

在 hadoop103 的 /opt/module/flume/job/group3 目录下创建 Flume2 的配置文件 flume2-netcat-flume.conf 。编辑配置文件，为 Flume2 配置 Source 监控端口 44444 数据流，配置 Sink 数据到下一级 Flume。

Bash

```
[myUbuntu@hadoop103 group3]$ touch flume2-netcat-flume.conf  
[myUbuntu@hadoop103 group3]$ vim flume2-netcat-flume.conf
```

```
hadoop@hadoop2:/usr/local/flume/job/group3$ touch flume2-netcat-flume.conf  
hadoop@hadoop2:/usr/local/flume/job/group3$ vim flume2-netcat-flume.conf
```

配置文件内容如下：

Bash

```
# Name the components on this agent  
a2.sources = r1  
a2.sinks = k1  
a2.channels = c1
```

```

# Describe/configure the source
a2.sources.r1.type = netcat
a2.sources.r1.bind = hadoop103
a2.sources.r1.port = 44444

# Describe the sink
a2.sinks.k1.type = avro
a2.sinks.k1.hostname = hadoop104
a2.sinks.k1.port = 4141

# Use a channel which buffers events in memory
a2.channels.c1.type = memory
a2.channels.c1.capacity = 1000
a2.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a2.sources.r1.channels = c1
a2.sinks.k1.channel = c1

```

(4) 配置 Flume3

在 hadoop104 的 /opt/module/flume/job/group3 目录下创建 Flume3 的配置文件 flume3-flume-logger.conf。编辑配置文件，为 Flume3 配置 source 用于接收 flume1 与 flume2 发送过来的数据流，最终合并后 sink 到控制台。

Bash

```
[myUbuntu@hadoop104 group3]$ touch flume3-flume-logger.conf
[myUbuntu@hadoop104 group3]$ vim flume3-flume-logger.conf
```

```
hadoop@hadoop3:/usr/local/flume/job/group3$ touch flume3-flume-logger.conf
hadoop@hadoop3:/usr/local/flume/job/group3$ vim flume3-flume-logger.conf
```

配置文件内容如下：

```

Bash
# Name the components on this agent
a3.sources = r1
a3.sinks = k1
a3.channels = c1

# Describe/configure the source
a3.sources.r1.type = avro
a3.sources.r1.bind = hadoop104
a3.sources.r1.port = 4141

# Describe the sink
a3.sinks.k1.type = logger

# Describe the channel
a3.channels.c1.type = memory
a3.channels.c1.capacity = 1000
a3.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a3.sources.r1.channels = c1
a3.sinks.k1.channel = c1

```

(5) 执行配置文件

分别开启对应配置文件： flume3-flume-logger.conf, flume2-netcat-flume.conf, flume1-logger-flume.conf。

Bash

```
[myUbuntu@hadoop104 flume]$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group3/flume3-flume-logger.conf -Dflume.root.logger=INFO,console  
[myUbuntu@hadoop103 flume]$ bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group3/flume2-netcat-flume.conf  
[myUbuntu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group3/flume1-logger-flume.conf
```

```
hadoop@hadoop1:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group3/flume1-logger-flume.conf  
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh  
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access  
  
hadoop@hadoop2:/usr/local/flume$ cd ..  
hadoop@hadoop2:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group3/flume1-logger-flume.conf  
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh  
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access  
  
hadoop@hadoop3:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group3/flume3-flume-logger.conf  
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh  
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access
```

(6) 在 hadoop103 上向/opt/module 目录下的 group.log 追加内容

Bash

```
[myUbuntu@hadoop103 module]$ echo 'hello' > group.log
```

```
hadoop@hadoop1:/usr/local/flume$ echo 'hello' > group.log  
hadoop@hadoop1:/usr/local/flume$ echo 'hadoop1' > group.log
```

(7) 在 hadoop102 上向 44444 端口发送数据

Bash

```
[myUbuntu@hadoop102 flume]$ telnet hadoop102 44444
```

```
hadoop@hadoop2:~$ telnet hadoop2 44444  
Trying 192.168.10.102...  
Connected to hadoop2.  
Escape character is '^]'.  
hi  
OK  
hadoop2  
OK
```

(8) 检查 hadoop104 上数据

```
2018-06-12 10:28:44,097 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 65 6C 6C 6F hello }  
2018-06-12 10:28:48,479 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 65 6C 6C 6F hello }  
  
2023-06-20 23:06:55,432 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 65 6C 6C 6F hello }  
2023-06-20 23:07:49,442 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 69 0D hi. }  
2023-06-20 23:07:53,833 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 68 61 64 6F 70 32 0D hadoop2. }  
2023-06-20 23:08:11,458 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 31 1 }
```

5.3.1.4 Flume 多路复用和自定义拦截器

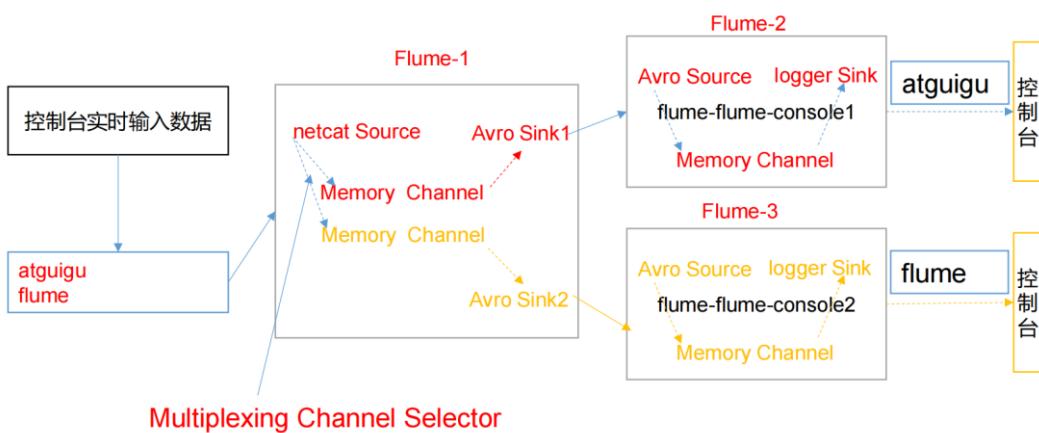
1) 案例需求

使用 Flume 采集服务器本地日志，需要按照日志类型的不同，将不同种类的日志发往不同的分析系统。

2) 需求分析

在实际的开发中，一台服务器产生的日志类型可能有很多种，不同类型的日志可能需要发送到不同的分析系统。此时会用到 Flume 拓扑结构中的 Multiplexing 结构，Multiplexing 的原理是，根据 event 中 Header 的某个 key 的值，将不同的 event 发送到不同的 Channel 中，所以我们需要自定义一个 Interceptor，为不同类型的 event 的 Header 中的 key 赋予不同的值。

在该案例中，我们以端口数据模拟日志，以是否包含“myUbuntu”模拟不同类型的日志，我们需要自定义 interceptor 区分数据中是否包含“myUbuntu”，将其分别发往不同的分析系统（Channel）。



3) 实现步骤

(1) 创建一个 Maven 项目，并引入以下依赖。

```
XML
<dependency>
    <groupId>org.apache.flume</groupId>
    <artifactId>flume-ng-core</artifactId>
    <version>1.9.0</version>
</dependency>
```

(2) 定义 CustomInterceptor 类并实现 Interceptor 接口。

1. 继承 flume 的拦截器接口
2. 重写 4 个抽象方法
 - a. 初始化方法 initialize：加载拦截器，获取连接
 - b. 两个处理方法 intercept，返回值与参数不同
 - i. Event：处理单个 Event，用于解耦

```

public Event intercept(Event event) {
    // 1. 获取事件中的头信息
    Map<String, String> headers = event.getHeaders();
    // 2. 获取事件中的body信息
    String body = new String(event.getBody());
    // 3. 判断body的开头第一个字符
    // 如果是字母发送到channel1, 如果是数字发送到channel2
    char c = body.charAt(0);
    if (c >= '0' && c <= '9') {
        headers.put("type", "number");
    } else if((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
        headers.put("type", "letter");
    }
    return event;
}

```

ii. List<Event>: 处理多个 Event, 系统实际使用的方法

c. 关闭方法 close

3. 编写静态内部类 Builder, 继承拦截器接口的 Builder, 重写 build 方法和配置方法 configure

Java

```

package com.neu.interceptor;

import org.apache.flume.Context;
import org.apache.flume.Event;
import org.apache.flume.interceptor.Interceptor;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class CustomInterceptor implements Interceptor {

    // 声明一个存放事件的集合
    private List<Event> addHeaderEvents;

    public void initialize() {
        addHeaderEvents = new ArrayList<Event>();
    }

    // 单个事件拦截
    public Event intercept(Event event) {
        // 1. 获取事件中的头信息
        Map<String, String> headers = event.getHeaders();
        // 2. 获取事件中的 body 信息
        String body = new String(event.getBody());
        // 3. 根据 body 中是否有“hello”来决定添加怎样的头信息
        if (body.contains("myUbuntu")) {
            headers.put("type", "first");
        } else {
            headers.put("type", "second");
        }
    }
}

```

```

        return event;
    }

// 批量事件拦截
public List<Event> intercept(List<Event> events) {
    // 1.清空集合
    addHeaderEvents.clear();
    // 2.遍历 events
    for (Event event : events) {
        // 3.为每个事件添加头信息
        addHeaderEvents.add(intercept(event));
    }
    return addHeaderEvents;
}

public void close() {

}

public static class Builder implements Interceptor.Builder {

    public Interceptor build() {
        return new CustomInterceptor();
    }

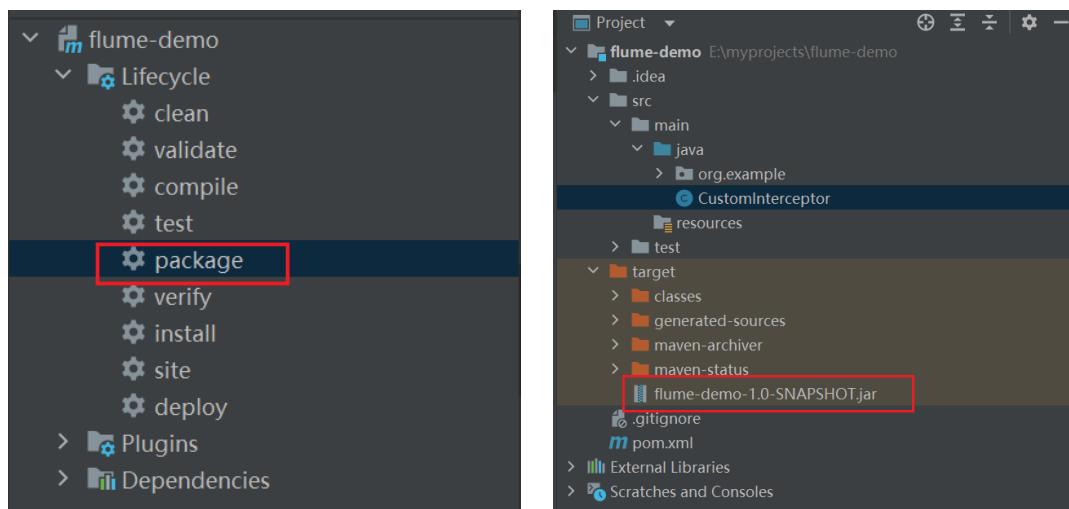
    public void configure(Context context) {

    }
}
}

```

之后将 Maven 项目打成 jar 包上传到 /opt/module/flume/lib 目录下。

打包：



使用共享文件夹上传：

```

hadoop@hadoop1:/mnt/hgfs/share$ ls
apache-flume-1.11.0-bin.tar.gz  hadoop-3.3.5.tar.gz
flume-demo-1.0-SNAPSHOT.jar    jdk-8u361-linux-x64.tar.gz
hadoop@hadoop1:/mnt/hgfs/share$ cp flume-demo-1.0-SNAPSHOT.jar /usr/local/flume/lib/
hadoop@hadoop1:/mnt/hgfs/share$ cd /usr/local/flume
hadoop@hadoop1:/usr/local/flume$ cd lib
hadoop@hadoop1:/usr/local/flume/lib$ ls
async-1.4.0.jar
asyncbase-1.8.2.jar
audience-annotations-0.5.0.jar
avro-1.11.0.jar
avro-ipc-1.11.0.jar
avro-ipc-jetty-1.11.0.jar
avro-ipc-netty-1.11.0.jar
commons-cli-1.5.0.jar
commons-codec-1.15.jar
commons-collections-3.2.2.jar
commons-compress-1.21.jar
commons-dbcp-1.4.jar
commons-io-2.11.0.jar
commons-lang-2.6.jar
commons-lang3-3.11.jar
commons-logging-1.2.jar
commons-pool-1.5.4.jar
commons-text-1.9.jar
curator-client-5.1.0.jar
curator-framework-5.1.0.jar
curator-recipes-5.1.0.jar
derby-10.14.2.0.jar
flume-avro-source-1.11.0.jar
flume-demo-1.0-SNAPSHOT.jar
flume-file-channel-1.11.0.jar

```

(3) 编辑 Flume 配置文件

为 hadoop102 上的 Flume1 配置 1 个 netcat source，1 个 sink group (2 个 avro sink)，并配置相应的 ChannelSelector 和 interceptor。

```

hadoop@hadoop1:/usr/local/flume/job/group4$ touch flume1-netcat-flume.conf
hadoop@hadoop1:/usr/local/flume/job/group4$ vim flume1-netcat-flume.conf

```

Bash

```

# Name the components on this agent
a1.sources = r1
a1.sinks = k1 k2
a1.channels = c1 c2

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Interceptor
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type =
com.myUbuntu.flume.interceptor.CustomInterceptor$Builder

# Channel Selector
a1.sources.r1.selector.type = multiplexing
a1.sources.r1.selector.header = type
a1.sources.r1.selector.mapping.first = c1
a1.sources.r1.selector.mapping.second = c2

# Describe the sink
a1.sinks.k1.type = avro
a1.sinks.k1.hostname = hadoop103

```

```

a1.sinks.k1.port = 4141
a1.sinks.k2.type=avro
a1.sinks.k2.hostname = hadoop104
a1.sinks.k2.port = 4242

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Use a channel which buffers events in memory
a1.channels.c2.type = memory
a1.channels.c2.capacity = 1000
a1.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1 c2
a1.sinks.k1.channel = c1
a1.sinks.k2.channel = c2

```

	Property Name	Default	Description
Multiplexing Channel Selector	selector.type	replicating	The component type name, needs to be multiplexing
	selector.header	flume.selector.header	
	selector.default	—	
	selector.mapping.*	—	

为 hadoop103 上的 Flume4 配置一个 avro source 和一个 logger sink。

```

hadoop@hadoop2:/usr/local/flume/job/group3$ touch flume2-netcat-flume.conf
hadoop@hadoop2:/usr/local/flume/job/group3$ vim flume2-netcat-flume.conf

```

```

Bash
a1.sources = r1
a1.sinks = k1
a1.channels = c1

a1.sources.r1.type = avro
a1.sources.r1.bind = hadoop103
a1.sources.r1.port = 4141

a1.sinks.k1.type = logger

a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

```

```
a1.sinks.k1.channel = c1  
a1.sources.r1.channels = c1
```

为 hadoop104 上的 Flume3 配置一个 avro source 和一个 logger sink。

```
hadoop@hadoop3:/usr/local/flume/job/group3$ touch flume3-flume-logger.conf  
hadoop@hadoop3:/usr/local/flume/job/group3$ vim flume3-flume-logger.conf
```

Bash

```
a1.sources = r1  
a1.sinks = k1  
a1.channels = c1  
  
a1.sources.r1.type = avro  
a1.sources.r1.bind = hadoop104  
a1.sources.r1.port = 4242
```

```
a1.sinks.k1.type = logger
```

```
a1.channels.c1.type = memory  
a1.channels.c1.capacity = 1000  
a1.channels.c1.transactionCapacity = 100
```

```
a1.sinks.k1.channel = c1  
a1.sources.r1.channels = c1
```

(4) 分别在 hadoop102, hadoop103, hadoop104 上启动 flume 进程，注意先后顺序。

```
hadoop@hadoop3:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group4/flume3-flume-logger.conf  
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh  
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access  
  
hadoop@hadoop2:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group4/flume2-flume-logger.conf  
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh  
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access  
  
hadoop@hadoop1:/usr/local/flume$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group4/flume1-netcat-flume.conf  
Info: Sourcing environment configuration script /usr/local/flume/conf/flume-env.sh  
Info: Including Hadoop libraries found via (/usr/local/hadoop/bin/hadoop) for HDFS access
```

(5) 在 hadoop102 使用 netcat 向 localhost:44444 发送字母和数字。

```
hadoop@hadoop1:/usr/local/flume$ netcat localhost 44444  
asdd  
OK  
dsfasdfa  
OK  
1231321  
OK  
5454  
OK  
sdgads  
OK  
5436sdgsf  
OK  
asdfa3435  
OK
```

(6) 观察 hadoop103 和 hadoop104 打印的日志。

hadoop2: 仅接收字母开头的消息

```

2023-06-21 00:38:29,765 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=letter} body: 61 73 64 64 asdd }
2023-06-21 00:38:29,766 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=letter} body: 64 73 66 61 73 64 66 61 dsfasdfa }
2023-06-21 00:38:30,620 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=letter} body: 73 64 67 61 64 73 sdgads }
2023-06-21 00:38:35,818 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=letter} body: 61 73 64 66 61 33 34 33 35 asdfa3435 }

```

hadoop3: 仅接收数字开头的消息

```

2023-06-21 00:38:35,719 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=number} body: 31 32 33 31 33 32 31 1231321 }
2023-06-21 00:38:35,720 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=number} body: 35 34 35 34 5454 }
2023-06-21 00:38:35,720 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{type=number} body: 35 34 33 36 73 64 67 73 66 5436sdgsf }

```

5.4 Redis 实操与分布式高可用部署

5.4.1 Redis 安装

首先在 ubuntu 中安装 redis:

ping 通说明 Redis 安装成功。

下表是一些常见的 Redis 配置文件:

```

master@Master:~$ redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>

```

	A	B
1	配置	解释
2	bind 127.0.0.1	绑定的ip
3	protected-mode yes	保护模式
4	port	端口设置
5	daemonize yes	以守护进程的方式运行, 默认为no, 即后台运行
6	pidfile /var/run/redis_6379.pid	如果以后台的方式运行, 我们就需要指定一个pid文件
7	loglevel notice	日志打印级别
8	logfile	日志的文件位置名
9	database 16	数据库的数量

点击图片可查看完整电子表格

5.4.2 Redis 事务实践

Redis 事务是一组命令的集合, 这些命令要么全部被执行, 要么全部都不执行。

Redis 事务具有以下主要特征:

- 单独的隔离操作: 事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中, 不会被其他客户端发送来的命令请求所打断。
- 没有锁机制: Redis 的事务并不具备类似关系型数据库事务那样的锁机制, 所以不能避免并发产生的冲突。
- 不保证原子性: Redis 中的事务并不保证原子性, 也就是说, 事务执行过程中出错的话, 已经执行成功的命令不会被回滚。 (但是 Redis 单条命令保证原子性)

	A	B
1	命令	操作
2	Redis Exec	执行所有事务块内的命令
3	Redis Watch	监视一个(或多个) key , 如果在事务执行之前这个(或这些) key 被其他命令所改动, 那么事务将被打断
4	Redis Discard	取消事务, 放弃执行事务块内的所有命令
5	Redis Unwatch	取消 WATCH 命令对所有 key 的监视
6	Redis Multi	标记一个事务块的开始

点击图片可查看完整电子表格

5.4.2.1 正常执行事务

首先用 Multi 命令开启事务，接着命令入队，输入 Exec 执行事务。可以看到入队的时候命令并没有被执行，只有真正进行执行语句的时候，这些命令才在队列中依次执行。

```
master@Master:~$ redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> get k2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> exec
1) OK
2) OK
3) "v2"
4) OK
127.0.0.1:6379>
```

5.4.2.2 放弃执行事务（Discard）

首先清空数据库

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379>
```

删除成功接着重新编写事务，只不过放弃执行（Discard）

可以看到放弃执行以后 set k2 命令并没有执行，所以 k2 中为空（nil）

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> get k2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> Discard
OK
127.0.0.1:6379> get k2
(nil)
127.0.0.1:6379> █
```

5.4.2.3 事务异常

- 编译异常

gett~~t~~ 这里由于拼写错误导致编译异常，整个事务中的所有命令都不执行。

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> gettt k3
(error) ERR unknown command 'gettt'
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get k1
(nil)
127.0.0.1:6379> █
```

- 运行异常

在事务中首先设置 k1 的内容为字符串“v1”，由于字符串不支持 incr 加一的语法所以在运行时会出现问题（由于语法格式没错所以编译时不会出现问题）。这里的命令会直到执行的时候才会发生报错，但是整个事务当中并没有影响到其他的命令。所以 redis 是不会保证整个事务中的原子性。

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k1 "v1"
QUEUED
127.0.0.1:6379> incr k1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR value is not an integer or out of range
3) OK
127.0.0.1:6379> get k2
"v2"
127.0.0.1:6379> █
```

所以,Redis 事务可以通过 MULTI 和 EXEC 命令将一组命令包裹起来原子性地执行,属于 Redis 的一种对数据进行批量操作的功能机制。

但是,Redis 事务也存在一定限制,不保证完全的原子性,且只能包含修改数据库的命令,而不能包含查询数据库的命令。

理解 Redis 事务的特点,可以更好地在实际应用中对 Redis 数据进行批量操作与控制,但也需要注意其限制,防止产生不必要的并发问题。

5.4.3 Redis 实现锁（乐观锁）

- 悲观锁:悲观锁的基本假设是,每次去拿数据的时候都认为可能会产生锁冲突,因此每次去拿数据的时候都会上锁,这样可以保证每次只有一个线程在操作数据。

特点:

- 读取数据时会上锁,可能会产生较长时间的锁等待,从而影响性能。
- 可以保证每次只有一个线程进行写操作,保证数据完整性。
- 仅适合写操作较少的场景。

- 乐观锁:乐观锁的基本假设是,每次去拿数据的时候都认为不会产生锁冲突,所以不会上锁。只在更新提交时去判断一下在此期间是否有人修改过这个数据。如果没有修改过,则更新。如果修改过,则重新读取数据,再对新数据做更新操作。

特点:

- 读取数据时不加锁,提高了读操作的性能。
- 只在更新提交时才会有锁冲突判断,可以提高吞吐量。
- 适合写操作较多的场景。

Redis 应用场景常常是对性能吞吐量要求很高的场景, 因此会用到乐观锁。

我们利用 Redis Watch 命令来监视 key, 如果在事务执行之前这个(或这些) key 被其他命令所改动, 那么事务将被打断。

采用两个 shell 分别执行事务:

	A	B	C
1	时间点	事务A	事务B
2	1	watch money	
3	2	multi	
4	3	decrby money 100	
5	4	incrby out 100	multi
6	5		incrby money 100
7	6		exec

点击图片可查看完整电子表格

首先模拟初始环境：

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> set money 1000
OK
127.0.0.1:6379> set out 0
OK
127.0.0.1:6379>
```

shell1:

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> set money 1000
OK
127.0.0.1:6379> set out 0
OK
127.0.0.1:6379> watch money
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> decrby money 100
QUEUED
127.0.0.1:6379> incrby out 100
QUEUED
127.0.0.1:6379> exec
(nil)
127.0.0.1:6379> 
```

shell2:

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incrby money 100
QUEUED
127.0.0.1:6379> exec
1) (integer) 1100
127.0.0.1:6379> 
```

可以看到 shell1 中事务 A 提交失败，在事务执行之前，Redis watch 会先检查数据是否被修改。借助 watch 的监视功能实现了乐观锁。

5.4.4 Redis 发布订阅

Redis 发布订阅（pub/sub）是一种消息通信模式：发送者（pub）发送消息，订阅者（sub）接收消息。微信、微博、关注系统等。Redis 客户端可以订阅任意数量的频道

常见的相关命令如下：

	xwW9Wm	Z7eGQ9	qBL9TD	RVDrpy	Ogtjc7	pTgM1g	w2	◀	▶
A									
1	命令及描述								
2	PSUBSCRIBE pattern [pattern ...]	订阅一个或多个符合给定模式的频道。							
3	PUBSUB subcommand [argument [argument ...]]	查看订阅与发布系统状态。							
4	PUBLISH channel message	将信息发送到指定的频道。							
5	PUNSUBSCRIBE [pattern [pattern ...]]	退订所有给定模式的频道。							
6	SUBSCRIBE channel [channel ...]	订阅给定的一个或多个频道的信息。							
7	UNSUBSCRIBE [channel [channel ...]]	指退订给定的频道。							

点击图片可查看完整电子表格

下面模拟 2 个客户端实践一下:

客户端 1:

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> subscribe imperfect
```

客户端 2:

```
127.0.0.1:6379> publish imperfect "hello,imperfect"
(integer) 1
127.0.0.1:6379> publish imperfect "hello,redis"
(integer) 1
127.0.0.1:6379> □
```

客户端 1 订阅结果:

```
127.0.0.1:6379> subscribe imperfect
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "imperfect"
3) (integer) 1
1) "message"
2) "imperfect"
3) "hello,imperfect"
1) "message"
2) "imperfect"
3) "hello,redis"
```

订阅发布接受成功。

Redis 通过 PUBLISH、SUBSCRIBE 和 PSUBSCRIBE 等命令实现发布和订阅功能

通过 SUBSCRIBE 命令订阅某频道后，redis-server 里维护了一个字典，字典就是一个 channel，而字典的值则是一个链表，链表中保存了所有订阅这个 channel 的客户端。SUBSCRIBE 命令的关键，就是将客户端添加到给定 channel 的订阅链表中

通过 PUBLISH 命令向订阅者发送消息，redis-server 会使用给定的频道作为键，在它所维护的 channel 字典中查找记录了订阅这个频道的所有客户端的链表，遍历这个链表，将消息发布给所有订阅者

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以针对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。

5.4.5 Redis 集群环境搭建

Redis 集群是将多个 Redis 节点组织在一起，实现数据的共享和管理。这里利用端口号实现伪集群的环境搭建。

打开 shell，修改端口、pid 名字、log 文件名字、dump.rdb 的名字：

```
master@Master:/etc/redis/cluster/6380
# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
daemonize yes

# When running daemonized, Redis writes a pid file in /var/run/redis.pid by
# default. You can specify a custom pid file location here.
pidfile /var/run/redis/redis-server6380.pid

# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket.
port 6380

# Specify the log file name. Also the empty string can be used to force
# Redis to log on the standard output. Note that if you use standard
# output for logging but daemonize, logs will be sent to /dev/null
logfile /var/log/redis/redis-server6380.log

# The filename where to dump the DB
dbfilename dump6380.rdb
```

依次类推在 cluster 目录下修改 6379、6380、6381 端口的配置文件。

Shell

```
sudo redis-server conf/redis6379.conf  
sudo redis-server conf/redis6380.conf  
sudo redis-server conf/redis6381.conf
```

使用命令查看进程

Shell

```
ps -ef | grep redis
```

结果如下：

```
master@Master:/etc/redis$ sudo redis-server cluster/6379/redis.conf  
master@Master:/etc/redis$ ps -ef | grep redis  
redis    937      1  0 6月20 ?          00:00:38 /usr/bin/redis-server 127  
.0.0.1:6379  
master   10389  10175  0 10:40 pts/4    00:00:00 grep --color=auto redis  
master@Master:/etc/redis$ sudo redis-server cluster/6380/redis.conf  
master@Master:/etc/redis$ sudo redis-server cluster/6381/redis.conf  
master@Master:/etc/redis$ ps -ef | grep redis  
redis    937      1  0 6月20 ?          00:00:38 /usr/bin/redis-server 127  
.0.0.1:6379  
root    10394  1593   0 10:40 ?          00:00:00 redis-server 127.0.0.1:6  
380  
root    10399  1593   0 10:40 ?          00:00:00 redis-server 127.0.0.1:6  
381  
master   10403  10175  0 10:41 pts/4    00:00:00 grep --color=auto redis  
master@Master:/etc/redis$
```

可以看到有 3 个 redis 进程在开启，说明集群搭建成功。

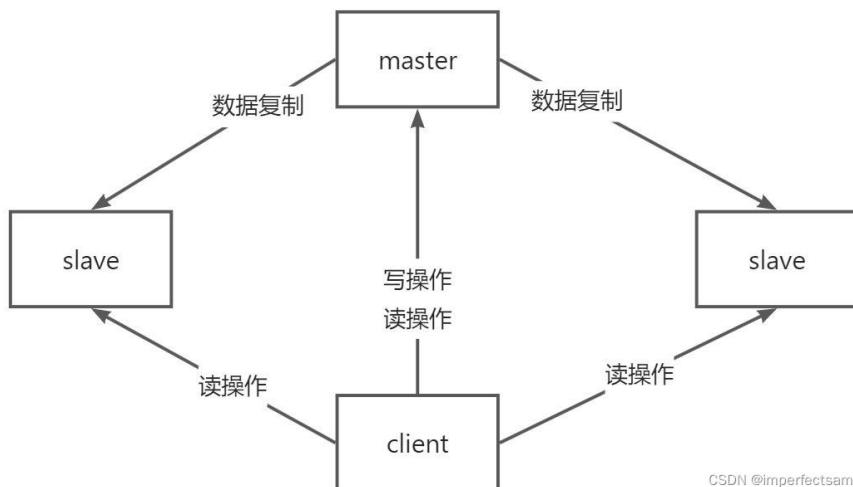
5.4.6 Redis 高可用

高可用 HA (High Availability) 是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间（比如 server 宕机、网络断联等）。

Redis 主要提供一主多从下的主从复制和哨兵集群机制保证高可用。

单机的 Redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑读高并发的。

因此架构做成主从(master-slave)架构，一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支撑读高并发。



CSDN @imperfectsam

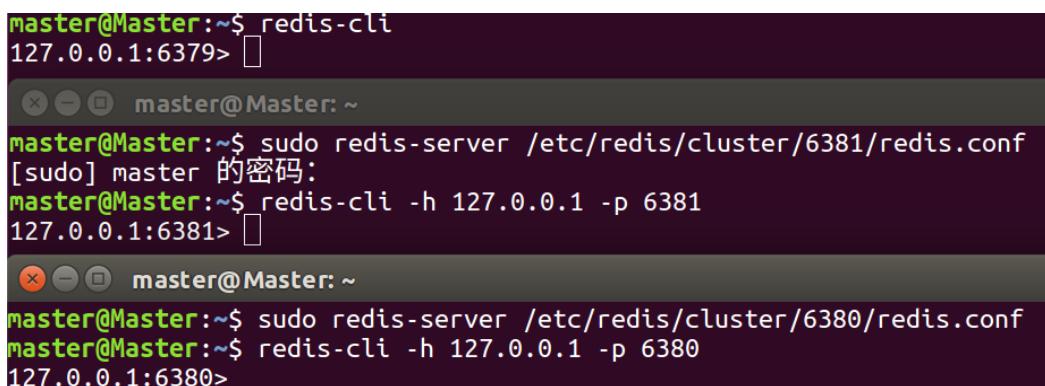
5.4.6.1 主从复制

主从复制，是将一台 Redis 服务器的数据，复制到其他的 Redis 服务器。前者称为主节点(master)，后者称为从节点(slave)。

数据的复制是单向的，只能由主节点到从节点。

- 数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
- 故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。
- 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写 Redis 数据时应用连接主节点，读 Redis 数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高 Redis 服务器的并发量。
- 高可用基石：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是 Redis 高可用的基础。

分别在 3 个 shell 打开 redis 服务：



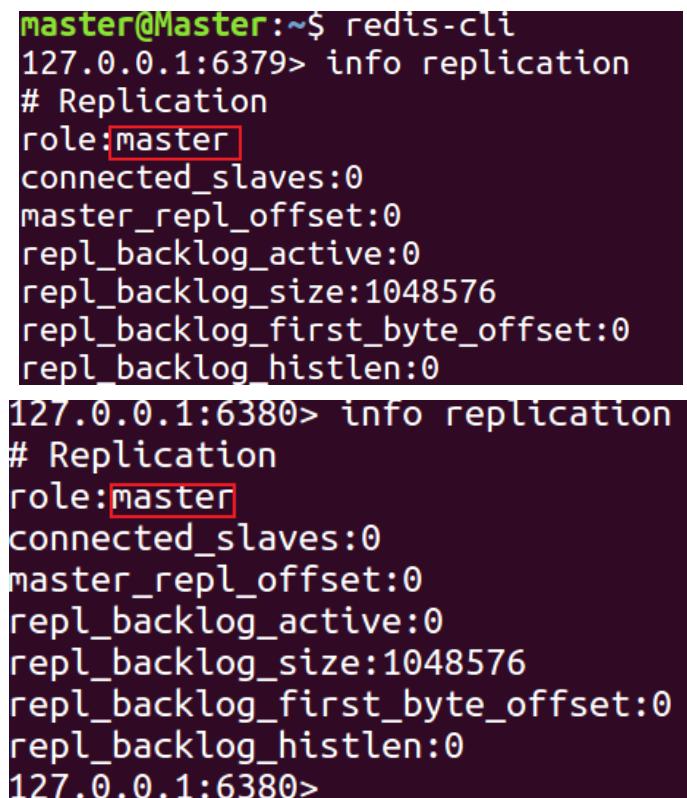
The screenshot shows three terminal windows on a Linux desktop. The top window shows the redis-cli connected to port 6379. The middle window shows the redis-server process starting at port 6381, requiring a password. The bottom window shows the redis-server process starting at port 6380, also requiring a password.

```
master@Master:~$ redis-cli
127.0.0.1:6379> 

master@Master:~$ sudo redis-server /etc/redis/cluster/6381/redis.conf
[sudo] master 的密码:
master@Master:~$ redis-cli -h 127.0.0.1 -p 6381
127.0.0.1:6381> 

master@Master:~$ sudo redis-server /etc/redis/cluster/6380/redis.conf
master@Master:~$ redis-cli -h 127.0.0.1 -p 6380
127.0.0.1:6380>
```

使用 info replication 查看信息：



The screenshot shows two terminal windows. The top window shows the redis-cli connected to port 6379, displaying the 'info replication' command output. The bottom window shows the redis-cli connected to port 6380, also displaying the 'info replication' command output. Both outputs show 'role:master' for both nodes.

```
master@Master:~$ redis-cli
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
replication_backlog_active:0
replication_backlog_size:1048576
replication_backlog_first_byte_offset:0
replication_backlog_histlen:0

127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
replication_backlog_active:0
replication_backlog_size:1048576
replication_backlog_first_byte_offset:0
replication_backlog_histlen:0
```

发现三个 redis 进程都认为自己是主机。

现在配置主机和从机的关系

主机: 6379

从机: 6380、6381

从机执行命令

```
Shell  
SLAVEOF 127.0.0.1 6379
```

结果如下:

主机:

```
127.0.0.1:6379> info replication  
# Replication  
role:master  
connected_slaves:2  
slave0:ip=127.0.0.1,port=6380,state=online,offset=141,lag=0  
slave1:ip=127.0.0.1,port=6381,state=online,offset=141,lag=0  
master_repl_offset:141  
repl_backlog_active:1  
repl_backlog_size:1048576  
repl_backlog_first_byte_offset:2  
repl_backlog_histlen:140  
127.0.0.1:6379> █
```

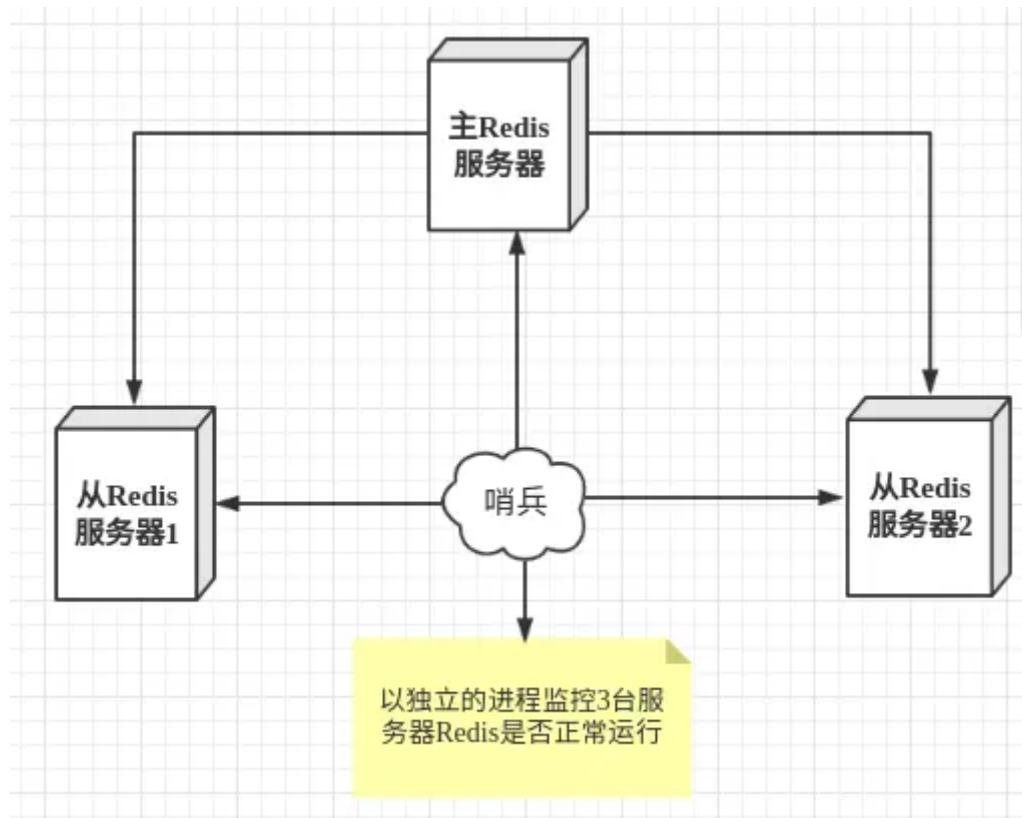
从机:

```
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379  
OK  
127.0.0.1:6380> info replication  
# Replication  
role:slave  
master_host:127.0.0.1  
master_port:6379  
master_link_status:up  
master_last_io_seconds_ago:5  
master_sync_in_progress:0  
slave_repl_offset:1  
slave_priority:100  
slave_read_only:1  
connected_slaves:0  
master_repl_offset:0  
repl_backlog_active:0  
repl_backlog_size:1048576  
repl_backlog_first_byte_offset:0  
repl_backlog_histlen:0  
127.0.0.1:6380> █
```

在 config 中配置则可以实现永久的主从配置。

5.4.6.2 哨兵模式

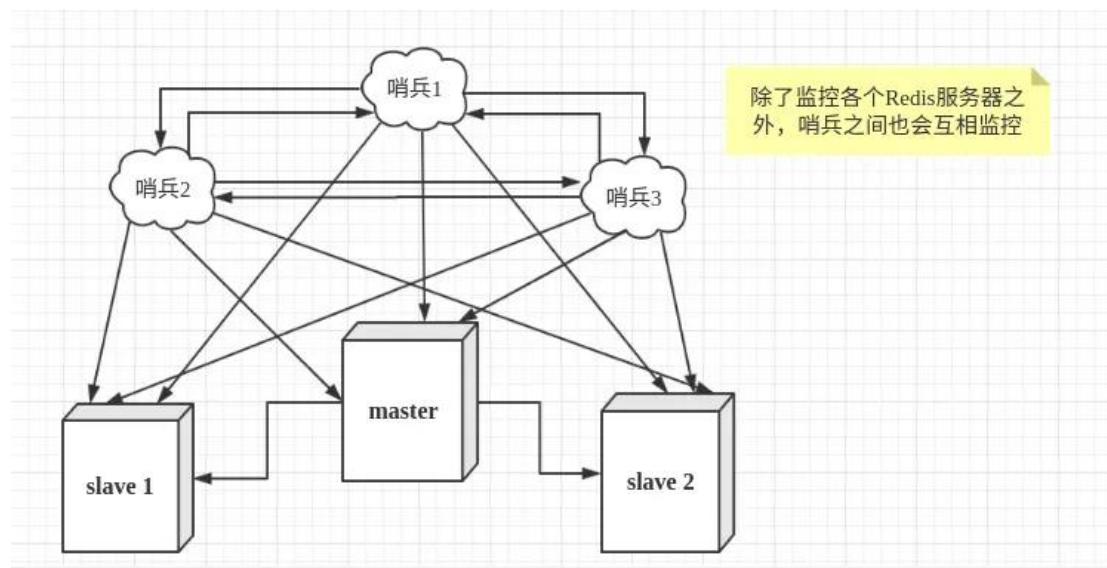
哨兵模式是一种特殊的模式，Redis 为其提供了专属的哨兵命令，它是一个独立的进程，能够独立运行。下面使用 Sentinel 搭建 Redis 集群，基本结构图如下所示：



在上图过程中，哨兵主要有两个重要作用：

- 哨兵节点会以每秒一次的频率对每个 Redis 节点发送 PING 命令，并通过 Redis 节点的回复来判断其运行状态。
- 当哨兵监测到主服务器发生故障时，会自动在从节点中选择一台将机器，并将其提升为主服务器，然后使用 PubSub 发布订阅模式，通知其他的从节点，修改配置文件，跟随新的主服务器。

在实际生产情况中，Redis Sentinel 是集群的高可用的保障，为避免 Sentinel 发生意外，它一般是由 3~5 个节点组成，这样就算挂了个别节点，该集群仍然可以正常运转。其结构图如下所示：



5.5 Flink：基于 Flink 的 CVPR2022 论文数据的分析与处理

5.5.1 Flink 集群搭建

将 flink 文件解压

```
Shell  
hadoop@master:~$ sudo tar -zxvf flink-1.17.1-bin-scala_2.12.tgz -C /opt/module/
```

修改 conf 文件

```
Shell  
hadoop@master:/opt/module/flink-1.17.1/conf$ vim flink-conf.yaml
```

修改条件如下：

```
Shell  
# JobManager 节点地址.  
jobmanager.rpc.address: master  
jobmanager.bind-host: 0.0.0.0  
rest.address: master  
rest.bind-address: 0.0.0.0  
# TaskManager 节点地址.需要配置为当前机器名  
taskmanager.bind-host: 0.0.0.0  
taskmanager.host: master
```

分别修改 masters 和 workers

```
Shell  
hadoop@master:/opt/module/flink-1.17.1/conf$ sudo vim workers  
hadoop@master:/opt/module/flink-1.17.1/conf$ sudo vim masters
```

```
Shell  
# masters  
hadoop102  
hadoop103  
hadoop104  
# workers  
master:8081
```

分发至另外的节点，并修改配置文件，随后启动 flink

```
Shell  
hadoop@master:/opt/module/flink-1.17.1$ ./bin/start-cluster.sh  
Starting cluster.  
Starting standalonesession daemon on host master.  
Starting taskexecutor daemon on host master.  
[INFO] 1 instance(s) of taskexecutor are already running on slave1.  
Starting taskexecutor daemon on host slave1.  
Starting taskexecutor daemon on host slave2.
```

Path, ID	Data Port	Last Heartbeat	All Slots	Free Slots	CPU	Flink Managed MEM
slave2:41937-f5bbc9 akka.tcp://flink@slave2:41937/user/rpc/taskmanager_0	45911	2023-06-21 11:07:52	1	1	2	512 MB
slave1:41193-1e8c5c akka.tcp://flink@slave1:41193/user/rpc/taskmanager_0	39347	2023-06-21 11:07:52	1	1	2	512 MB
master:33483-0fd1d3 akka.tcp://flink@master:33483/user/rpc/taskmanager_0	44307	2023-06-21 11:07:52	1	1	2	512 MB

随后在 Flink 程序的 pom.xml 中添加配置

```

XML
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.2.4</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <artifactSet>
              <excludes>
                <exclude>com.google.code.findbugs:jsr305</exclude>
                <exclude>org.slf4j:*</exclude>
                <exclude>log4j:*</exclude>
                <exclude>org.apache.hadoop:*</exclude>
              </excludes>
            </artifactSet>
            <filters>
              <filter>
                <!-- Do not copy the signatures in the META-INF folder.
                    Otherwise, this might cause SecurityExceptions when using the JAR. --
->
                <artifact>*:*</artifact>
                <excludes>
                  <exclude>META-INF/*.SF</exclude>
                  <exclude>META-INF/*.DSA</exclude>
                  <exclude>META-INF/*.RSA</exclude>
                </excludes>
              </filter>
            </filters>
            <transformers combine.children="append">
              <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"
">
                </transformer>
              </transformers>
            </configuration>

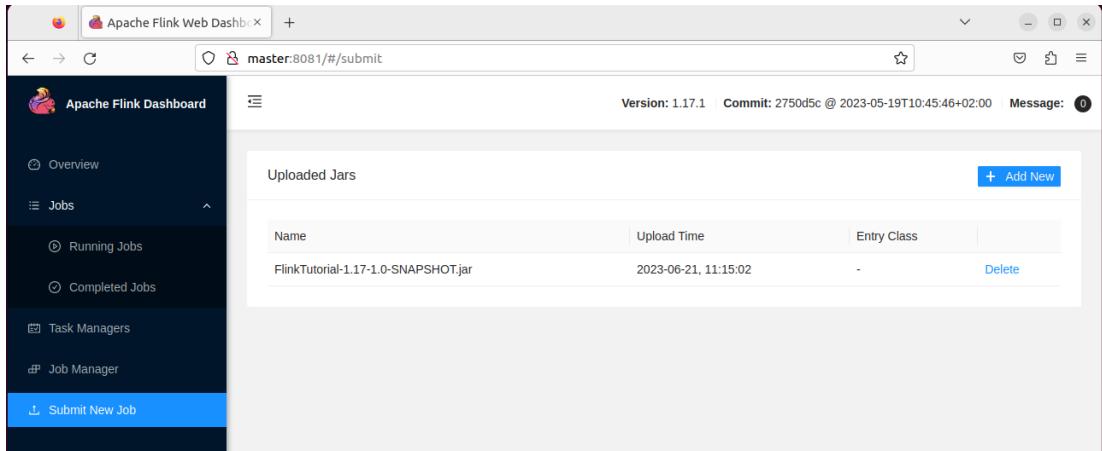
```

```
</execution>
</executions>
</plugin>
</plugins>
</build>
```

5.5.2 Flink 作业提交

将 Flink Java 项目打包成为 jar 文件上传至虚拟机

在 WebUI 界面提交作业



Python 爬虫源码

```
Python
import requests
from bs4 import BeautifulSoup
import pandas as pd
import numpy as np
headers = {'user-agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/74.0.3729.131 Safari/537.36'}#创建头部信息
urls=[

    'https://openaccess.thecvf.com/CVPR2023?day=all',
]
alllist=[]
dayCount = 0

for url in urls:
    r=requests.get(url,headers=headers)
    content=r.content.decode('utf-8')
    soup = BeautifulSoup(content, 'html.parser')
    dts=soup.find_all('dt',class_='ptitle')
    hts='http://openaccess.thecvf.com/'
    #数据爬取
    for i in range(len(dts)):
        try:
            title=dts[i].a.text.strip()
            print('这是第'+str(i+dayCount)+',篇文章:', title)
            href=hts+dts[i].a['href']
            r = requests.get(href, headers=headers)
            content = r.content.decode('utf-8')
            soup = BeautifulSoup(content, 'html.parser')
            #print(title,href)
```

```

div_author=soup.find(name='div', attrs={"id": "authors"})
authors = div_author.text.strip().split(';')[0]
print('第'+str(i+dayCount)+'篇文章的作者：', authors)
value=(title, authors)
alllist.append(value)
except:
    continue
dayCount+=len(dts)

name = ['title', 'authors']
papers = pd.DataFrame(columns=name, data=allist)
print(papers.head())
papers.to_csv('CVPR2022.csv', encoding='utf-8')

```

Python 数据清洗源码

```

Python
import pandas as pd
import re

data = pd.read_csv('CVPR_cat.csv')
data[u'authors'] = data[u'authors'].astype(str)
data[u'authors'] = data[u'authors'].apply(lambda x:re.sub(',\s+;',';',x))
data[u'title'] = data[u'title'].astype(str)
d = data[u'title'].apply(lambda x:re.sub(',\s+',x))
data[u'title'] = d.apply(lambda x:re.sub('\\"\\(\')',' ',x))
data = data.iloc[:,1:3]
data.to_csv('CVPR_cat.csv', index=True, encoding='utf-8')

```

5.5.3 通过爬虫爬取 CVPR2022 的论文数据

```

(base) C:\Users\test\Desktop>python temp.py
这是第0篇文章: Dual Cross-Attention Learning for Fine-Grained Visual Categorization and Object Re-Identification
第0篇文章的作者: Haowei Zhu, Wenjing Ke, Dong Li, Ji Liu, Lu Tian, Yi Shan
这是第1篇文章: SimAN: Exploring Self-Supervised Representation Learning of Scene Text via Similarity-Aware Normalization
第1篇文章的作者: Canjie Luo, Lianwen Jin, Jingdong Chen
这是第2篇文章: Weakly Supervised Semantic Segmentation by Pixel-to-Prototype Contrast
第2篇文章的作者: Ye Du, Zehua Fu, Qingjie Liu, Yunhong Wang
这是第3篇文章: Controllable Animation of Fluid Elements in Still Images
第3篇文章的作者: Aniruddha Mahapatra, Kuldeep Kulkarni
这是第4篇文章: Recurrent Dynamic Embedding for Video Object Segmentation
第4篇文章的作者: Mingxing Li, Li Hu, Zhiwei Xiong, Bang Zhang, Pan Pan, Dong Liu
这是第5篇文章: Deep Hierarchical Semantic Segmentation
第5篇文章的作者: Liulei Li, Tianfei Zhou, Wenguan Wang, Jianwu Li, Yi Yang

import requests
from bs4 import BeautifulSoup
import pandas as pd
import numpy as np
headers = {'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.131 Safari'}
urls=[
    'https://openaccess.thecvf.com/CVPR2022?day=2022-06-21',
    'https://openaccess.thecvf.com/CVPR2022?day=2022-06-22',
    'https://openaccess.thecvf.com/CVPR2022?day=2022-06-23',
    'https://openaccess.thecvf.com/CVPR2022?day=2022-06-24'
]

```

针对某些论文信息缺失采用 try-except 块捕获错误

```

#数据爬取
for i in range(len(dts)):
    try:
        title=dts[i].a.text.strip()
        print('这是第'+str(i+dayCount)+'篇文章:', title)
        href=hts+dts[i].a['href']
        r = requests.get(href, headers=headers)
        content = r.content.decode('utf-8')
        soup = BeautifulSoup(content, 'html.parser')
        #print(title, href)
        div_author=soup.find(name='div', attrs={"id":"authors"})
        authors = div_author.text.strip().split(';')[0]
        print('第'+str(i+dayCount)+'篇文章的作者: ', authors)
        value=(title, authors)
        alllist.append(value)
    except:
        continue
    dayCount+=len(dts)

```

最终爬取结果如下

		authors
这是第2072篇文章的作者:	Liangdong Qiu, Chongjie Ye, Pei Chen, Yunbi Liu, Xiaoguang Han, Shuguang Cui	
这是第2073篇文章:	Globetrotter: Connecting Languages by Connecting Images	
第2073篇文章的作者:	Dídac Surís, Dave Epstein, Carl Vondrick	
0	Dual Cross-Attention Learning for Fine-Grained...	Haowei Zhu, Wenjing Ke, Dong Li, Ji Liu, Lu Ti...
1	SimAN: Exploring Self-Supervised Representatio...	Canjie Luo, Lianwen Jin, Jingdong Chen
2	Weakly Supervised Semantic Segmentation by Pix...	Ye Du, Zehua Fu, Qingjie Liu, Yunhong Wang
3	Controllable Animation of Fluid Elements in St...	Aniruddha Mahapatra, Kuldeep Kulkarni
4	Recurrent Dynamic Embedding for Video Object S...	Mingxing Li, Li Hu, Zhiwei Xiong, Bang Zhang, ...

5.5.4 使用 Flink 的批处理功能统计关键词词频

maven 包的配置信息如下所示:

```

XML
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Flink</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <flink.version>1.12.4</flink.version>
        <scala.version>2.12</scala.version>
    </properties>

    <dependencies>
        <!-- flink 的 scala 的 api -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-scala_${scala.version}</artifactId>
            <version>${flink.version}</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>

```

```

<!-- flink streaming 的 scala 的 api -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_${scala.version}</artifactId>
    <version>${flink.version}</version>
    <scope>compile</scope>
</dependency>
</dependencies>
</project>

```

使用 Scala 语言对清洗过的数据进行 mapreduce 操作

```

Scala
import org.apache.flink.api.common.operators.Order
import org.apache.flink.api.scala.ExecutionEnvironment
import org.apache.flink.api.scala._
import scala.io.Source
object compute_cvpr {
    def main(args: Array[String]): Unit = {
        val bEnv = ExecutionEnvironment.getExecutionEnvironment
        val filePath="F:\\pyProject\\spider\\CVPR_cat.csv"
        val csv = bEnv.readCsvFile[PapersLog](filePath,ignoreFirstLine = true)
        //println(csv.hashCode)
        val stopEnword = Source.fromFile("src/main/scala/stopwords.txt").getLines()
        val stopWordList = stopEnword.toList
        val counts = csv.flatMap(_.title.split(" "))
            .filter(_.nonEmpty)
            .filter(stopword(_, stopWordList))
            .map((_,1))
            .groupByKey(0)
            .sum(1)
            .sortPartition(field = 1, Order.DESCENDING)
            .setParallelism(1)
        counts.writeAsCsv("src/main/scala/CVPR/hotWords22.csv").setParallelism(1)
        bEnv.execute("batch wordCount")
    }
    def stopword(string: String, stopWordList: List[String]): Boolean = {
        !stopWordList.contains(string.toLowerCase())
    }
    case class
    PapersLog(index:Int,title:String,authors:String)
}

```

```

PapersLog(259,Dual-AI: Dual-Path Actor Interaction Learning for Group Activity Recognition,"Mingfei Han")
PapersLog(260,A Brand New Dance Partner: Music-Conditioned Pluralistic Dancing Controlled by Multiple Dance Genres,"Jinwoo Kim")
PapersLog(261,Adaptive Early-Learning Correction for Segmentation From Noisy Annotations,"Sheng Liu")
PapersLog(262,Multi-Scale Memory-Based Video Deblurring,"Bo Ji")
PapersLog(263,A Scalable Combinatorial Solver for Elastic Geometrically Consistent 3D Shape Matching,"Paul Roetzer")
PapersLog(264,Geometric Structure Preserving Warp for Natural Image Stitching,"Peng Du")
PapersLog(265,Focal Length and Object Pose Estimation via Render and Compare,"Georgy Ponomatkin")
PapersLog(266,Dynamic 3D Gaze From Afar: Deep Gaze Estimation From Temporal Eye-Head-Body Coordination,"Soma Nonaka")

Process finished with exit code 0

```

scala 源码

```

Scala
import org.apache.flink.api.common.operators.Order
import org.apache.flink.api.scala.ExecutionEnvironment
import org.apache.flink.api.scala._

```

```

object compute_author {
  def main(args: Array[String]): Unit = {
    val bEnv = ExecutionEnvironment.getExecutionEnvironment
    val filePath="C:\\Users\\test\\CVPR_cat.csv"
    val csv = bEnv.readCsvFile[PapersLog](filePath,ignoreFirstLine = true)
    csv.print()

    val counts = csv.flatMap(_.authors.split(","))
      .filter(_.nonEmpty)
      .map((_,1))
      .groupBy(0)
      .sum(1)
      .sortPartition(field = 1, Order.DESCENDING)
      .setParallelism(1)

    counts.writeAsCsv("src/main/scala/CVPR/authors_all_DES.csv").setParallelism(1)
    bEnv.execute("batch wordCount")
  }

  case class
  PapersLog(Index:Int, title:String,authors:String)
}

```

```

Scala
import org.apache.flink.api.common.operators.Order
import org.apache.flink.api.scala.ExecutionEnvironment
import org.apache.flink.api.scala._
import scala.io.Source

object compute_cvpr {
  def main(args: Array[String]): Unit = {
    val bEnv = ExecutionEnvironment.getExecutionEnvironment
    val filePath="C:\\Users\\test\\CVPR_cat2.csv"
    val csv = bEnv.readCsvFile[PapersLog](filePath,ignoreFirstLine = true)
    //println(csv.hashCode)
    val stopEnword = Source.fromFile("src/main/scala/stopwords.txt").getLines()
    val stopWordList = stopEnword.toList

    val counts = csv.flatMap(_.title.split(" "))
      .filter(_.nonEmpty)
      .filter(stopword(_, stopWordList))
      .map((_,1))
      .groupBy(0)
      .sum(1)
      .sortPartition(field = 1, Order.DESCENDING)
      .setParallelism(1)
    counts.writeAsCsv("src/main/scala/CVPR/hotWords16-20_des.csv").setParallelism(1)

    bEnv.execute("batch wordCount")
  }

  def stopword(string: String, stopWordList: List[String]): Boolean = {
    !stopWordList.contains(string.toLowerCase())
  }
}

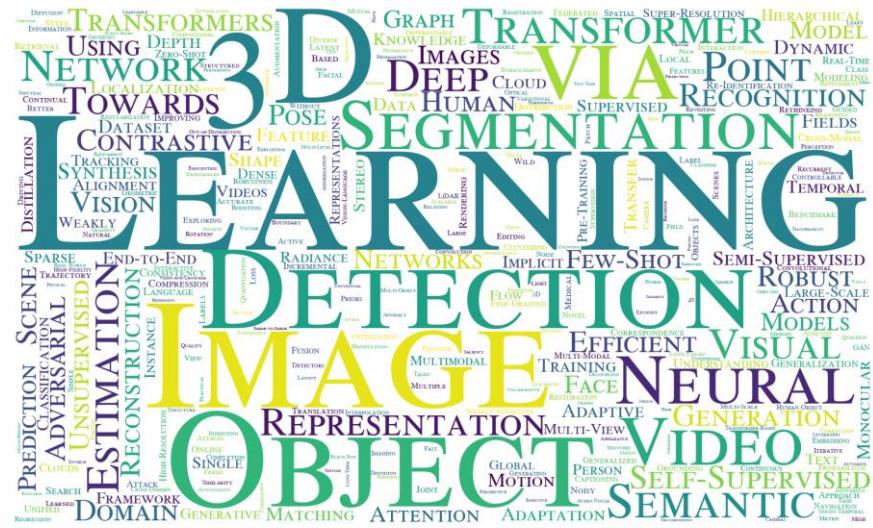
```

```
case class  
PapersLog(index:Int,title:String,authors:String)  
}
```

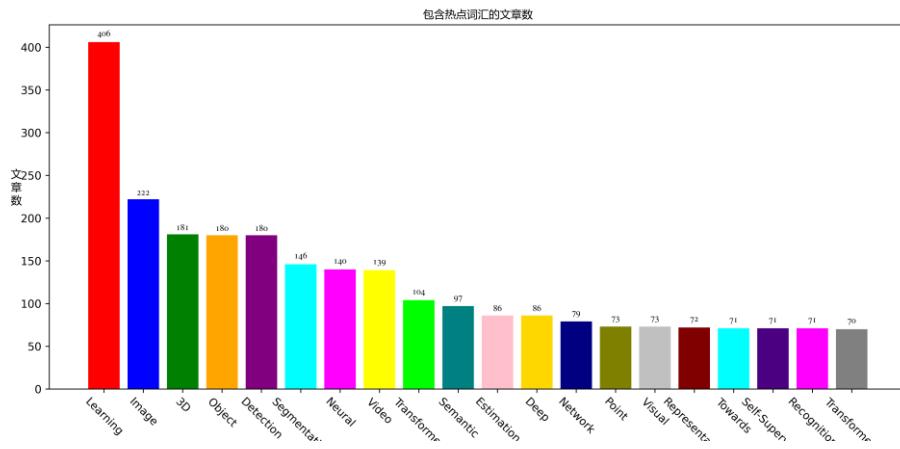
python 统计结果

```
hotword_all.info  
✓ 0.0s  
<bound method DataFrame.info of  
  word  num  
0    Learning  406  
1      Image  222  
2        3D  181  
3     Object  180  
4   Detection  180  
...    ...  ...  
2981   X-Pool:    1  
2982  XMP-Font:    1  
2983 Yourself:    1  
2984   ZZ-Net:    1  
2985      sRGB    1  
  
[ 2986 rows x 2 columns]>
```

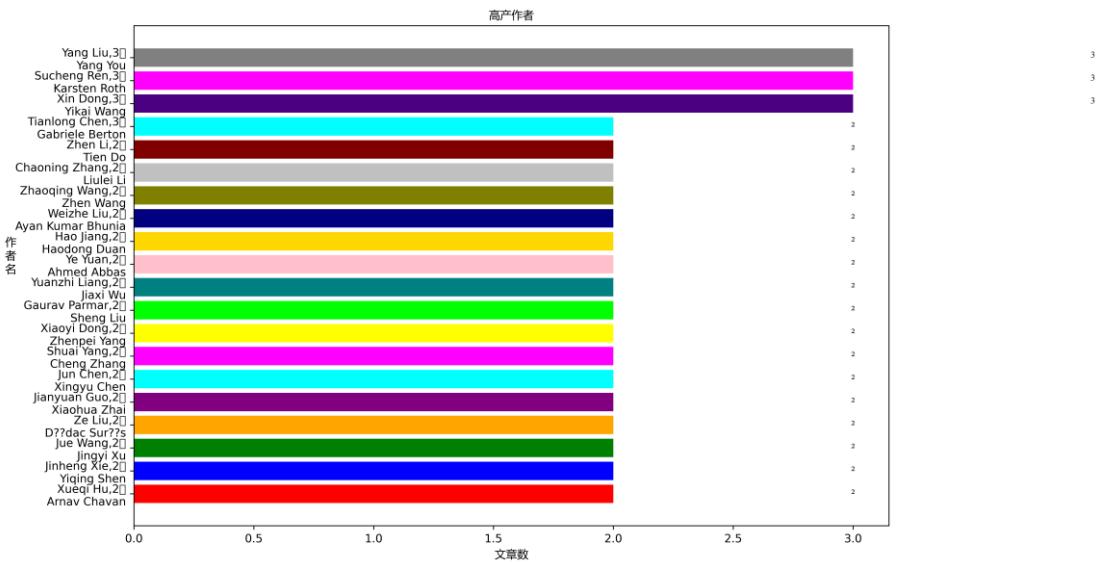
5.5.5 关键词云图谱统计



5.5.6 关键词柱状图统计



5.5.7 高产作者统计



5.6 Sqoop：基础操作与通道实践

5.6.1 Sqoop 安装

在安装和使用 Sqoop 环境前需要先具备 Java 和 Hadoop 环境。

安装版本：1.4.6（Apache 已经停止维护）

修改配置文件

```
Shell
cd $SQOOP_HOME/conf
mv sqoop-env-template.sh sqoop-env.sh
vi sqoop-env.sh
export HADOOP_COMMON_HOME=/export/servers/hadoop-2.7.5
export HADOOP_MAPRED_HOME=/export/servers/hadoop-2.7.5
export HIVE_HOME=/export/servers/hive
```

```
#Set path to where bin/hadoop is available
export HADOOP_COMMON_HOME=/usr/local/hadoop

#Set path to where hadoop-*-*-core.jar is available
export HADOOP_MAPRED_HOME=/usr/local/hadoop
```

加入 mysql 的 jdbc 驱动包

```
Shell
cp /hive/lib/mysql-connector-java-5.1.32.jar $SQOOP_HOME/lib/
```

验证启动

```
Shell
bin/sqoop list-databases \
--connect jdbc:mysql://localhost:3306/ \
--username root --password hadoop
```

到这里，整个 Sqoop 安装工作完成。

5.6.2 Sqoop 导入数据

在 DataGrip 中新建数据表

SQL

```
SET FOREIGN_KEY_CHECKS=0;
```

```
-- -----
```

```
DROP TABLE IF EXISTS `emp`;
CREATE TABLE `emp` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(100) DEFAULT NULL,
  `deg` varchar(100) DEFAULT NULL,
  `salary` int(11) DEFAULT NULL,
  `dept` varchar(10) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
-- -----
```

```
INSERT INTO `emp` VALUES ('1201', 'gopal', 'manager', '50000', 'TP');
INSERT INTO `emp` VALUES ('1202', 'manisha', 'Proof reader', '50000', 'TP');
INSERT INTO `emp` VALUES ('1203', 'khalil', 'php dev', '30000', 'AC');
INSERT INTO `emp` VALUES ('1204', 'prasanth', 'php dev', '30000', 'AC');
INSERT INTO `emp` VALUES ('1205', 'kranthy', 'admin', '20000', 'TP');
```

```
-- -----
```

```
DROP TABLE IF EXISTS `emp_add`;
CREATE TABLE `emp_add` (
  `id` int(11) DEFAULT NULL,
  `hno` varchar(100) DEFAULT NULL,
  `street` varchar(100) DEFAULT NULL,
  `city` varchar(100) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
-- -----
```

```
INSERT INTO `emp_add` VALUES ('1201', '288A', 'vgiri', 'jubilee');
INSERT INTO `emp_add` VALUES ('1202', '108I', 'aoc', 'sec-bad');
INSERT INTO `emp_add` VALUES ('1203', '144Z', 'pgutta', 'hyd');
INSERT INTO `emp_add` VALUES ('1204', '78B', 'old city', 'sec-bad');
INSERT INTO `emp_add` VALUES ('1205', '720X', 'hitec', 'sec-bad');
```

```
-- -----
```

```
DROP TABLE IF EXISTS `emp_conn`;
CREATE TABLE `emp_conn` (
  `id` int(100) DEFAULT NULL,
  `phno` varchar(100) DEFAULT NULL,
  `email` varchar(100) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```

INSERT INTO `emp_conn` VALUES ('1201', '2356742', 'gopal@tp.com');
INSERT INTO `emp_conn` VALUES ('1202', '1661663', 'manisha@tp.com');
INSERT INTO `emp_conn` VALUES ('1203', '8887776', 'khalil@ac.com');
INSERT INTO `emp_conn` VALUES ('1204', '9988774', 'prasanth@ac.com');
INSERT INTO `emp_conn` VALUES ('1205', '1231231', 'kranthi@tp.com');

```

emp:

	id	name	deg	salary	dept
1	1201	gopal	manager	50000	TP
2	1202	manisha	Proof reader	50000	TP
3	1203	khalil	php dev	30000	AC
4	1204	prasanth	php dev	30000	AC
5	1205	kranthi	admin	20000	TP

emp_add:

	id	hno	street	city
1	1201	288A	vgiri	jubilee
2	1202	108I	aoc	sec-bad
3	1203	144Z	pgutta	hyd
4	1204	78B	old city	sec-bad
5	1205	720X	hitec	sec-bad

emp_conn:

	id	phno	email
1	1201	2356742	gopal@tp.com
2	1202	1661663	manisha@tp.com
3	1203	8887776	khalil@ac.com
4	1204	9988774	prasanth@ac.com
5	1205	1231231	kranthi@tp.com

“导入工具”导入单个表从 RDBMS 到 HDFS。表中的每一行被视为 HDFS 的记录。所有记录都存储为文本文件的文本数据。

Shell

```
$ sqoop import (generic-args) (import-args)
```

5.6.2.1 导入 mysql 数据到 HDFS

Shell

```

bin/sqoop import \
--connect jdbc:mysql://node-1:3306/userdb \
--username root \
--password hadoop \

```

```
--delete-target-dir \
--target-dir /sqoopresult \
--table emp --m 1
```

```
23/06/11 15:24:54 INFO sqoop.Sqoop: Running Sqoop version: 1.4.5
23/06/11 15:24:56 INFO manager.MySQLManager: Preparing to use a MySQL streaming resultset.
INFO orm.CompilationManager: Writing jar file: /tmp/sqoop-hadoop/compile/c
ebe706d23ebb1fd99c1f063ad51ebd7/emp.jar
-----
0 mapreduce.Job: map 0% reduce 0%
23/06/11 15:28:08 INFO mapreduce.Job: map 100% reduce 0%
23/06/11 15:28:16 INFO mapreduce.Job: Job job_1419242001831_0001 completed
successfully
-----
23/06/11 15:28:17 INFO mapreduce.ImportJobBase: Transferred 145 bytes in 1
77.5849 seconds (0.8165 bytes/sec)
23/06/11 15:28:17 INFO mapreduce.ImportJobBase: Retrieved 5 records.
```

使用命令查看导入的数据:

```
Shell
$ $HADOOP_HOME/bin/hadoop fs -cat /user/hadoop/emp/part-m-00000
```

```
1201,gopal,manager,50000,TP
1202,manisha,preader,50000,TP
1203,kalil,php dev,30000,AC
1204,prasanth,php dev,30000,AC
1205,kranthi,admin,20000,TP
```

导入成功。

5.6.2.2 导入 mysql 数据到 Hive

- 先复制表结构到 hive 中再导入数据

将 mysql 中 emp_add 表结构复制到 hive 并命名为 emp_add_sp

```
Shell
bin/sqoop create-hive-table \
--connect jdbc:mysql://node-1:3306/sqoopdb \
--table emp_add \
--username root \
--password hadoop \
--hive-table test.emp_add_sp
```

将 mysql 数据导入 hive

```
Shell
bin/sqoop import \
--connect jdbc:mysql://node-1:3306/sqoopdb \
--username root \
--password hadoop \
--table emp_add \
--hive-table test.emp_add_sp \
--hive-import \
```

```
--m 1
```

- 直接复制到 hive

Shell

```
bin/sqoop import \
--connect jdbc:mysql://node-1:3306/userdb \
--username root \
--password hadoop \
--table emp_conn \
--hive-import \
--m 1 \
--hive-database test;
```

5.6.2.3 借助 where 语法实现 mysql 数据子集的导入

Shell

```
bin/sqoop import \
--connect jdbc:mysql://node-1:3306/sqoopdb \
--username root \
--password hadoop \
--where "city ='sec-bad'" \
--target-dir /wherequery \
--table emp_add --m 1
```

5.6.2.4 增量导入

Sqoop 支持增量导入数据，可以增量导入新添加的行数据。这样可以避免频繁导入完整的数据表。

Sqoop 增量导入主要有两种方式：基于时间戳的 Lastmodified 模式，基于数据库主键的模式。

下面以主键模式增量导入为例：

- 首次需要全量数据导入

Shell

```
bin/sqoop import \
--connect jdbc:mysql://node-1:3306/userdb \
--username root \
--password hadoop \
--target-dir /appendresult \
--table emp --m 1
```

- 在 mysql 中插入新信息

SQL

```
insert into `userdb`.`emp` (`id`, `name`, `deg`, `salary`, `dept`) values ('1206', 'allen', 'admin', '30000', 'tp');
insert into `userdb`.`emp` (`id`, `name`, `deg`, `salary`, `dept`) values ('1207', 'woon', 'admin', '40000', 'tp');
```

- 实现增量导入

Shell

```
bin/sqoop import \
--connect jdbc:mysql://node-1:3306/userdb \
```

```
--username root --password hadoop \
--table emp --m 1 \
--target-dir /appendresult \
--incremental append \
--check-column id \
--last-value 1205
```

导入成功。

5.6.3 Sqoop 导出数据

类似导入操作， Sqoop 导出主要也有默认方式和增量导出的方式

- 默认模式导出

默认情况下， Sqoop 会将每一条输入记录转换成 SQL 的 INSERT 语句。默认模式主要用于全表数据的导出。

- 增量导出

增量模式下可以只导出 Hadoop 中上次导出后新添加或更新的数据到关系数据库中,实现差异化的数据导出。这可以避免覆盖关系数据库中的数据,提高导出效率。

导出命令语法:

```
Shell
$ sqoop export (generic-args) (export-args)
```

在 HDFS 中“EMP/”目录下新建文件 emp_data，输入以下内容:

```
1201, gopal, manager, 50000, TP
1202, manisha, preader, 50000, TP
1203, kalil, php dev, 30000, AC
1204, prasanth, php dev, 30000, AC
1205, kranthi, admin, 20000, TP
1206, satish p. aru des. 20000, GR
```

在导出数据之前需要先在 mysql 中创建表头:

```
mysql> USE db;
mysql> CREATE TABLE employee (
    id INT NOT NULL PRIMARY KEY,
    name VARCHAR(20),
    deg VARCHAR(20),
    salary INT,
    dept VARCHAR(10));
```

导出命令:

```
Shell
bin/sqoop export \
--connect jdbc:mysql://node-1:3306/userdb \
--username root \
--password hadoop \
--table emp \
```

```
--export-dir /user/hadoop/emp/
```

在 mysql 中验证是否导出成功：

SQL

```
select * from employee;
```

Id	Name	Designation	Salary	Dept
1201	gopal	manager	50000	TP
1202	manisha	preader	50000	TP
1203	kalil	php dev	30000	AC
1204	prasanth	php dev	30000	AC
1205	kranthi	admin	20000	TP
1206	satish p	grp des	20000	GR

说明导出成功。

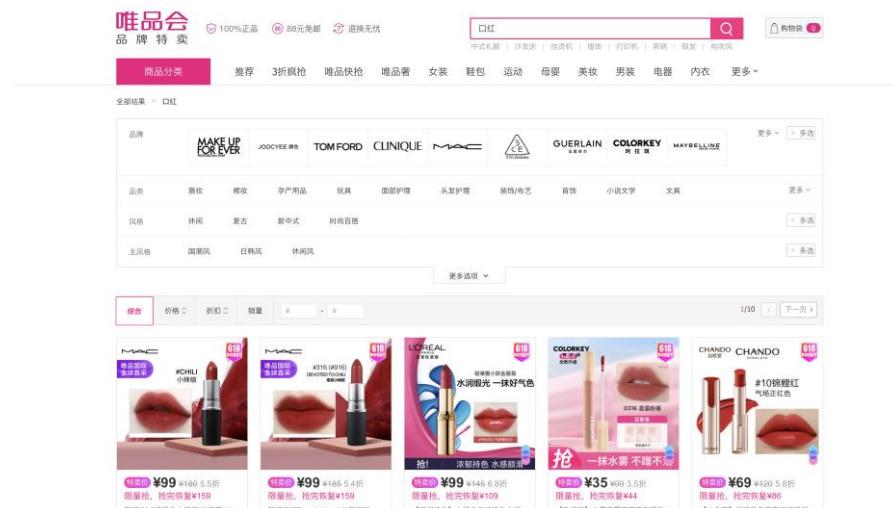
5.7 爬虫应用项目： Python 爬取唯品会数据→Flume→Kafka

5.7.1 唯品会口红商品数据爬虫

5.7.1.1 数据来源分析

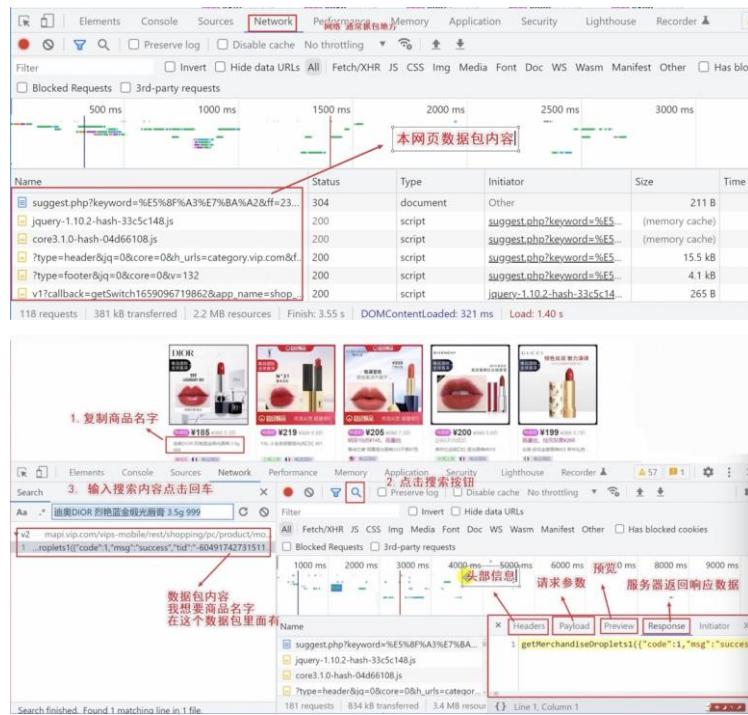
1. 采集数据内容

爬虫的数据是唯品会网站上口红商品信息，包括口红的原价、折扣、品牌、标题、描述等



2. 分析数据来源

通过网页分析找到数据包。网页中每一页共包含 120 条商品数据，但是一个数据包中只有该页面的部分商品数据。通过进一步分析数据包，发现在页面中共三个数据包，分别对应前 50 条商品数据、中间 50 条商品数据和后 20 条商品数据。



5.7.1.2 代码实现

1. 导入库

```
SQL
#导入数据请求模块 --
import requests
#导入格式化输出模块
from pprint import pprint
#导入 csv 模块
import csv
import time
```

2. 创建接收数据的 csv 文件

```
SQL
#创建文件
#flume 配置监督文件夹后会给加入文件夹中的文件自动加入 CSV 后缀
f = open('data', mode='a', encoding='utf-8', newline="")
csv_writer = csv.DictWriter(f, fieldnames=['标题', '品牌', '原价', '折扣', '售价'])
#写入表头
csv_writer.writeheader()
```

3. 抓取页面信息方法编写

```
SQL
#HTTP 请求的头部信息
headers = {
    'cookie': 'cps=adp%3Ag1o71nr0%3A%3A%3A%3A; vip_first_visitor=1;
    vip_address=%257B%2522pid%2522%253A%2522104103%2522%252C%2522cid%252
    2%253A%2522104103101%2522%252C%2522pname%2522%253A%2522%255Cu6e56
    %255Cu5357%255Cu7701%2522%252C%2522cname%2522%253A%2522%255Cu957%
    %255Cu6c99%255Cu5e02%2522%257D; vip_province=104103;
```

```

vip_province_name=%E6%B9%96%E5%8D%97%E7%9C%81;
vip_city_name=%E9%95%BF%E6%B2%99%E5%B8%82; vip_city_code=104103101;
vip_wh=VIP_HZ; vip_ipver=31; user_class=a;
mars_sid=b5bbf072cb8ed88296d9329d7665a548;
VipUINFO=luc%3Aa%7Csuc%3Aa%7Cbct%3Ac_new%7Chct%3Ac_new%7Cbdts%3A
0%7Cbcts%3A0%7Ckfts%3A0%7Cc10%3A0%7Crcabt%3A0%7Cp2%3A0%7Cp3%3A1
%7Cp4%3A0%7Cp5%3A1%7Cul%3A3105;
PHPSESSID=urihb7npn87saub2bq02c8i864; mars_pid=0;
visit_id=64A4526890CE0069861940120BF1C8B1;
vip_access_times=%7B%22list%22%3A17%7D; pg_session_no=31;
vip_tracker_source_from=;
mars_cid=1616063423670_791ead13d4c9db3d47f30231840db66e',
'referer': 'https://category.vip.com/',
'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/89.0.4389.90 Safari/537.36'
}

#发送 HTTP 请求获取商品信息
def get_info(pid):
    html_url = 'https://mapi.vip.com/vips-mobile/rest/shopping/pc/product/module/list/v2'
    params_1 = {
        'app_name': 'shop_pc',
        'app_version': '4.0',
        'warehouse': 'VIP_HZ',
        'fdc_area_id': '104103101',
        'client': 'pc',
        'mobile_platform': '1',
        'province_id': '104103',
        'api_key': '70f71280d5d547b2a7bb370a529aee1',
        'user_id': '',
        'mars_cid': '1616063423670_791ead13d4c9db3d47f30231840db66e',
        'wap_consumer': 'a',
        'productIds': pid,
        'scene': 'search',
        'standby_id': 'nature',
        'extParams':
        '{"stdSizeVids":"","preheatTipsVer":"3","couponVer":"v2","exclusivePrice":"1","iconSpec
":"2x","ic2label":1}',
        'context': '',
        '_': '1616070446929',
    }
    response_1 = requests.get(url=html_url, params=params_1, headers=headers)

#解析数据
products_list = response_1.json()['data']['products']
for i in products_list:
    # 标题
    title = i['title']
    # 品牌
    show_name = i['brandShowName']
    # 原价
    market_Price = i['price']['marketPrice']
    # 折扣
    discount = i['price']['saleDiscount']
    # 售价
    sale_Price = i['price']['salePrice']
    dit = {

```

```

        '标题': title,
        '品牌': show_name,
        '原价': market_Price,
        '折扣': discount,
        '售价': sale_Price,
    }
    csv_writer.writerow(dict)
#控制台查看
print(title, show_name, market_Price, discount, sale_Price, sep=' | ')

```

4. 运行爬取

SQL

```

#通过 python 代码模拟浏览器发送请求
for page in range(0, 1921, 120):
    time.sleep(1)
    url = 'https://mapi.vip.com/vips-mobile/rest/shopping/pc/search/product/rank'
    params = {
        # 'callback': 'getMerchandiseIds',
        'app_name': 'shop_pc',
        'app_version': '4.0',
        'warehouse': 'VIP_HZ',
        'fdc_area_id': '104103101',
        'client': 'pc',
        'mobile_platform': '1',
        'province_id': '104103',
        'api_key': '70f71280d5d547b2a7bb370a529aeea1',
        'user_id': '',
        'mars_cid': '1616063423670_791ead13d4c9db3d47f30231840db66e',
        'wap_consumer': 'a',
        'standby_id': 'nature',
        'keyword': '口红',
        'lv3CatIds': '',
        'lv2CatIds': '',
        'lv1CatIds': '',
        'brandStoreSns': '',
        'props': '',
        'priceMin': '',
        'priceMax': '',
        'vipService': '',
        'sort': '0',
        'pageOffset': str(page),
        'channelId': '1',
        'gPlatform': 'PC',
        'batchSize': '120',
        '_': '1616070446924',
    }

```

```

#发送请求，url/params/headers 都是 get 函数里面的参数
response = requests.get(url=url, params=params, headers=headers)

```

获取数据

```

# response.text 文本数据 response.json()字典数据 response.content 二进制数据
#print(response.json())
#格式化输出的效果
#pprint(response.json())

```

```
#键值对取值，根据冒号左边的内容，提取冒号右边的内容
```

```
"""
```

```
print(products)
```

```
lis = []
```

```
for index in products:
```

```
    pid = index['pid']
```

```
    # 把商品 ID 添加到 lis 列表里面
```

```
    lis.append(pid)
```

```
"""
```

```
#列表推导式
```

```
products = response.json()['data']['products']
```

```
#列表切片【因为整个网页分成前面 50 中间 50 后面 20】
```

```
#列表转字符串
```

```
string_1 = ','.join(products[0:50])
```

```
string_2 = ','.join(products[50:100])
```

```
string_3 = ','.join(products[100:])
```

```
get_info(pid=string_1)
```

```
get_info(pid=string_2)
```

```
get_info(pid=string_3)
```

5.7.1.3 爬虫结果

1. 将爬虫结果将控制台中打印结果如下

```
雕花口红/浮雕唇膏女半哑光持色正红烂番茄色杜鹃定制款 | 花西子 | 130 | 9.9折 | 129
变色润唇膏保湿滋润女淡化唇纹口红不沾杯防水持久护唇多用 | 薇润姿 | 39 | 5.9折 | 23
【多色可选】阿玛尼红管臻致丝绒哑光唇釉#405 206哑光复古 | Armani | 330 | 6.8折 | 225
【心动挚礼】小灯管口红唇膏滋润显白保湿豆沙色水红色奶茶色 | 美宝莲 | 109 | 5.0折 | 54
魅可口红Devoted to chili 新旧版随机发 | MAC | 185 | 7.5折 | 139
滋润口红2g保湿饱满显色唇彩怀孕期孕妇可用彩妆化妆品提升气色 | 植物主义 | 58 | 3.1折 | 18
【520礼物】小丝缎唇膏口红女水润百搭元气缎面大牌 | 卡姿兰 | 129 | 4.5折 | 58
YSL圣罗兰方管口红滋润1966 | 圣罗兰 | 350 | 6.3折 | 219
古驰倾色绒雾唇膏#505 3.5g | GUCCI | 370 | 5.9折 | 219
唇釉唇泥雾面丝绒哑光口红轻薄持久显白丝滑唇彩学生小众品牌女 | AKF | 117 | 3.3折 | 39
【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色 | 欧莱雅 | 145 | 6.8折 | 99
【孕妇可用】变色唇膏润唇膏女持久滋润不易掉色不易沾杯唇膏口红 | Dr. Hztone+ | 99 | 3.7折 | 37
兰蔻22年全新粉金小蛮腰唇膏口红 196 羊绒朱砂 | Lancome | 310 | 6.1折 | 189
【精选潮流】哑光口红南瓜豆沙橘红多色顺滑轻柔雾感聚会必备 | 3CE | 129 | 5.0折 | 64
【多色可选】YSL小金条细管哑光口红1966 复古显白 | 圣罗兰 | 390 | 7.5折 | 293
【520礼物】兔年迷你空气唇釉礼盒6支套装丝绒雾面镜面白口红 | 珂拉琪 | 199 | 4.5折 | 89
雕花口红/浮雕唇膏女半哑光持色正红烂番茄色杜鹃定制款 | 花西子 | 130 | 9.9折 | 129
【心动挚礼】黑魔方 绝色持久唇膏口红滋润丝绒哑光高级显白 | 美宝莲 | 109 | 5.9折 | 64
阿玛尼权力唇膏 #405 权能番茄 持久滋润口红 | Armani | 370 | 6.5折 | 239
```

2. 爬虫生成的文件 data.csv

- 1 标题,品牌,原价,折扣,售价
- 2 魅可MAC子弹头小辣椒/烂番茄chili口红#602,MAC,180,5.5折,99
- 3 菁纯丝绒雾面口红 #196 新款小蛮腰唇膏口红胡萝卜色,兰蔻,300,6.3折,189
- 4 【滋润持久】小妖金口红持久水润显色豆沙色161开挂红666,欧莱雅,145,6.8折,99
- 5 【小金笔】润泽显色唇膏口红滋润显色显白保湿细腻顺滑黄皮适用,自然堂,120,5.8折,69
- 6 【小金笔】润泽显色唇膏口红滋润显色显白保湿细腻顺滑黄皮适用,自然堂,120,5.8折,69
- 7 【滋润持久】小妖金口红持久水润显色豆沙色161开挂红666,欧莱雅,145,6.8折,99
- 8 【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色,欧莱雅,145,6.8折,99
- 9 【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色,欧莱雅,145,6.8折,99
- 10 【特色百搭】纷泽滋润细跟口红唇膏611裸色细管绒雾唇霜女,欧莱雅,175,5.7折,99
- 11 【持久滋润】印迹唇釉口红女轻薄持久滋润显色大牌女全新,欧莱雅,145,7.5折,109
- 12 【持久滋润】印迹唇釉口红女轻薄持久滋润显色大牌女全新,欧莱雅,145,7.5折,109
- 13 【特色百搭】纷泽滋润细跟口红唇膏611裸色细管绒雾唇霜女,欧莱雅,175,5.7折,99
- 14 【哑光持久】免年限定版纷泽滋润细跟口红绒雾唇霜高级显白必备,欧莱雅,175,7.9折,139
- 15 【哑光持久】纷泽滋润细跟口红绒雾唇霜高级显白必备,欧莱雅,175,8.5折,149
- 16 虫草焕颜恒色凝润唇膏 135 牛血红 3.9g,羽西,150,7.2折,108

5.7.2 flume 监控数据并传输到到 kafka

5.7.2.1 flume 配置文件

a. 配置文件：Flume 代理监视指定目录下的 CSV 文件变化，并将数据从源读取到内存通道中，然后将数据从内存通道发送到 Kafka 消息队列

SQL

```
# 定义代理名称和组件
#csv_source 作为源
agent.sources = csv_source
#memory_channel 作为通道
agent.channels = memory_channel
#kafka-sink 作为目标
agent.sinks = kafka-sink

# 配置 CSV 源
#指定源的类型为 spooldir, 即监视指定目录中的文件变化
agent.sources.csv_source.type = spooldir
#spoolDir 指定 CSV 文件所在的目录路径
agent.sources.csv_source.spoolDir = /home/ljz-ha/桌面/project/csv_directory
#fileHeader 设置为 true 表示 CSV 文件包含头部信息
agent.sources.csv_source.fileHeader = true
#fileSuffix 指定文件的后缀名为.csv
agent.sources.csv_source.fileSuffix = .csv

# 配置内存通道
#指定通道的类型为 memory, 数据在内存中进行传输和存储
agent.channels.memory_channel.type = memory

# 配置日志输出目标
#指定目标的类型, 将数据发送到 Kafka 消息队列
agent.sinks.kafka-sink.type = org.apache.flume.sink.kafka.KafkaSink
#指定 Kafka 主题的名称为 streamingtopic
agent.sinks.kafka-sink.topic = streamingtopic
```

```

#指定 Kafka 集群的地址和端口
agent.sinks.kafka-sink.brokerList = hadoop102:9092
#表示发送消息后需要等待 Kafka 的确认
agent.sinks.kafka-sink.requiredAcks = 1
#batchSize 指定了每次批量发送的消息数量为 5
agent.sinks.kafka-sink.batchSize = 5

# 绑定源、通道和目标
agent.sources.csv_source.channels = memory_channel
agent.sinks.kafka-sink.channel = memory_channel

```

5.7.2.2 数据传输流程

爬虫生成的数据文件通过 Flume 传输到 Kafka 的流程：

爬虫并生成数据文件——启动 Zookeeper——启动 Kafka（监听，topic 是 streamingtopic）——执行 kafka-console-consumer 命令（将 Kafka 接收到的数据打印到控制台）——用配置好的 Flume 代理监控文件夹——将爬虫生成的文件拉入文件夹中——Flume 将爬虫数据传到 Kafka

流程细节

1. 爬虫并生成数据文件

运行程序爬取唯品会口红商品数据，并将数据采集进 data 文件中，文件中的部分数据截图如下：

1 标题,品牌,原价,折扣,售价
2 魅可MAC子弹头小辣椒/烂番茄chili口红#602,MAC,180,5.5折,99
3 菁纯丝绒雾面口红 #196 新款小蛮腰唇膏口红胡萝卜色,兰蔻,300,6.3折,189
4 【滋润持久】小妖金口红持久水润显色豆沙色161开挂红666,欧莱雅,145,6.8折,99
5 【小金笔】润泽显色唇膏口红滋润显色显白保湿细腻顺滑黄皮适用,自然堂,120,5.8折,69
6 【小金笔】润泽显色唇膏口红滋润显色显白保湿细腻顺滑黄皮适用,自然堂,120,5.8折,69
7 【滋润持久】小妖金口红持久水润显色豆沙色161开挂红666,欧莱雅,145,6.8折,99
8 【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色,欧莱雅,145,6.8折,99
9 【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色,欧莱雅,145,6.8折,99
10 【持色百搭】纷泽滋润细跟口红唇膏611裸色细管绒雾唇霜女,欧莱雅,175,5.7折,99
11 【持久滋润】印迹唇釉口红女轻薄持久滋润显色大牌女全新,欧莱雅,145,7.5折,109
12 【持久滋润】印迹唇釉口红女轻薄持久滋润显色大牌女全新,欧莱雅,145,7.5折,109
13 【持色百搭】纷泽滋润细跟口红唇膏611裸色细管绒雾唇霜女,欧莱雅,175,5.7折,99
14 【哑光持久】兔年限定版纷泽滋润细跟口红绒雾唇霜高级显白必备,欧莱雅,175,7.9折,139
15 【哑光持久】纷泽滋润细跟口红绒雾唇霜高级显白必备,欧莱雅,175,8.5折,149
16 虫草焕颜恒色凝润唇膏 135 牛血红 3.9g,羽西,150,7.2折,108

2. 启动 zookeeper

启动 Zookeeper 服务，作为 Kafka 集群的协调和管理节点：

```

ljk-ha@hadoop102:~/module/kafka_2.11-0.9.0.0/bin$ cd $ZK_HOME/bin
ljk-ha@hadoop102:~/module/zookeeper-3.4.5/bin$ ./zk
zkCleanup.sh zkCli.sh zkEnv.sh zkServer.sh
zkCli.cmd zkEnv.cmd zkServer.cmd
ljk-ha@hadoop102:~/module/zookeeper-3.4.5/bin$ ./zkServer.sh start
JMX enabled by default
Using config: /home/ljk-ha/module/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED

```

3. 启动 kafka

启动 Kafka Broker，配置监听端口和主题（streamingtopic）：

```

ljk-ha@hadoop102:~/module/zookeeper-3.4.5/bin$ cd $KAFKA_HOME/bin
ljk-ha@hadoop102:~/module/kafka_2.11-0.9.0.0/bin$ kafka-server-start.sh $KAFKA_HOME/config/server.properties

```

```
[2023-06-02 16:3] liz-ha@hadoop102: ~/module/kafka_2.11-0.9.0.0/bin ls.jar is not in the  
classpath (kafka)  
[2023-06-02 16:33:42,055] INFO Creating /brokers/ids/0 (is it secure? false) (ka  
fka.utils.ZKCheckedEphemeral)  
[2023-06-02 16:33:42,075] INFO Result of znode creation is: OK (kafka.utils.ZKCh  
eckedEphemeral)  
[2023-06-02 16:33:42,083] INFO Registered broker 0 at path /brokers/ids/0 with a  
ddresses: PLAINTEXT -> EndPoint(hadoop102,9092,PLAINTEXT) (kafka.utils.ZkUtils)  
[2023-06-02 16:33:42,113] INFO Kafka version : 0.9.0.0 (org.apache.kafka.common.  
utils.AppInfoParser)  
[2023-06-02 16:33:42,119] INFO Kafka commitId : fc7243c2af4b2b4a (org.apache.kaf  
ka.common.utils.AppInfoParser)  
[2023-06-02 16:33:42,121] INFO [Kafka Server 0], started (kafka.server.KafkaServ  
er)  
[2023-06-02 16:33:42,485] INFO [ReplicaFetcherManager on broker 0] Removed fetch  
er for partitions [test1,0],[test2,0],[streamingtopic,0] (kafka.server.ReplicaFe  
tcherManager)  
[2023-06-02 16:33:42,559] INFO [ReplicaFetcherManager on broker 0] Removed fetch  
er for partitions [test1,0],[test2,0],[streamingtopic,0] (kafka.server.ReplicaFe  
tcherManager)
```

4. 执行 kafka-console-consumer 命令

通过执行 kafka-console-consumer 命令，验证 Kafka 是否能正确接收到数据，并将接收到的数据打印到控制台：

```
liz-ha@hadoop102:~/module/kafka_2.11-0.9.0.0/bin$ ./kafka-console-consumer.sh --  
zookeeper hadoop102:2181 --topic streamingtopic
```

5. 启动 flume 代理

使用 Flume 的配置文件，来监控指定的文件夹：

```
liz-ha@hadoop102:~/module/kafka_2.11-0.9.0.0/bin$ flume-ng agent --name agent -  
-conf $FLUME_HOME/conf --conf-file /home/liz-ha/桌面/project/spider2.conf -Dfl  
ume.root.logger=INFO,console
```

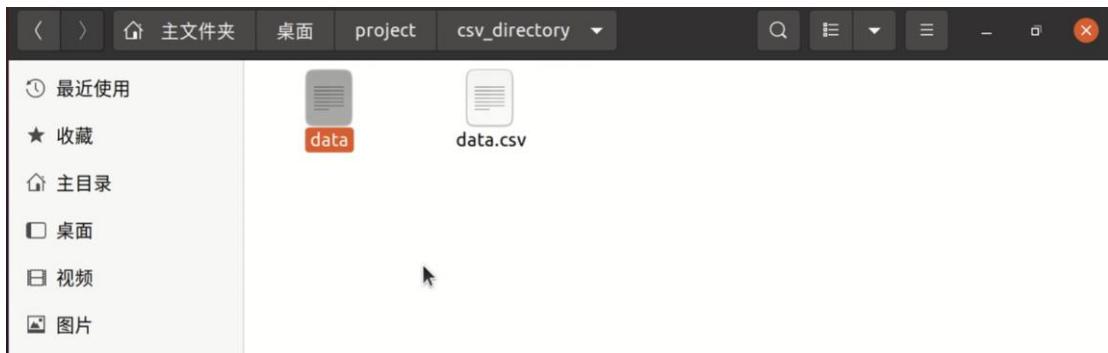
Kafka Sink 启动，开始将数据从 Flume 发送到 Kafka 的操作：

```
2023-06-02 16:34:51,763 (lifecycleSupervisor-1-1) [INFO - kafka.utils.Logging$cl  
ass.info(Logging.scala:68)] Property request.required.acks is overridden to 1  
2023-06-02 16:34:51,763 (lifecycleSupervisor-1-1) [INFO - kafka.utils.Logging$cl  
ass.info(Logging.scala:68)] Property serializer.class is overridden to kafka.ser  
ializer.DefaultEncoder  
2023-06-02 16:34:51,859 (lifecycleSupervisor-1-1) [INFO - org.apache.flume.instr  
umentation.MonitoredCounterGroup.register(MonitoredCounterGroup.java:120)] Monit  
ored counter group for type: SINK, name: kafka-sink: Successfully registered new  
MBean.  
2023-06-02 16:34:51,859 (lifecycleSupervisor-1-1) [INFO - org.apache.flume.instr  
umentation.MonitoredCounterGroup.start(MonitoredCounterGroup.java:96)] Component  
type: SINK, name: kafka-sink started
```

6. 将爬虫生成的文件拉入文件夹中

Flume 监控文件夹，实时读取被移动到文件夹里的文件，因此爬虫程序生成的数据文件不能直接生成在 Flume 文件夹中。需要在数据文件完整生成后，手动将文件拖入被监控的文件夹中。

将爬虫程序生成的数据文件放入 Flume 所监控的文件夹中：



7. Flume 将新增文件中的数据传输到 Kafka

Flume 会检测到文件夹中的新增文件，将新增文件内容作为消息发送到 Kafka 的指定 topic (streamingtopic)

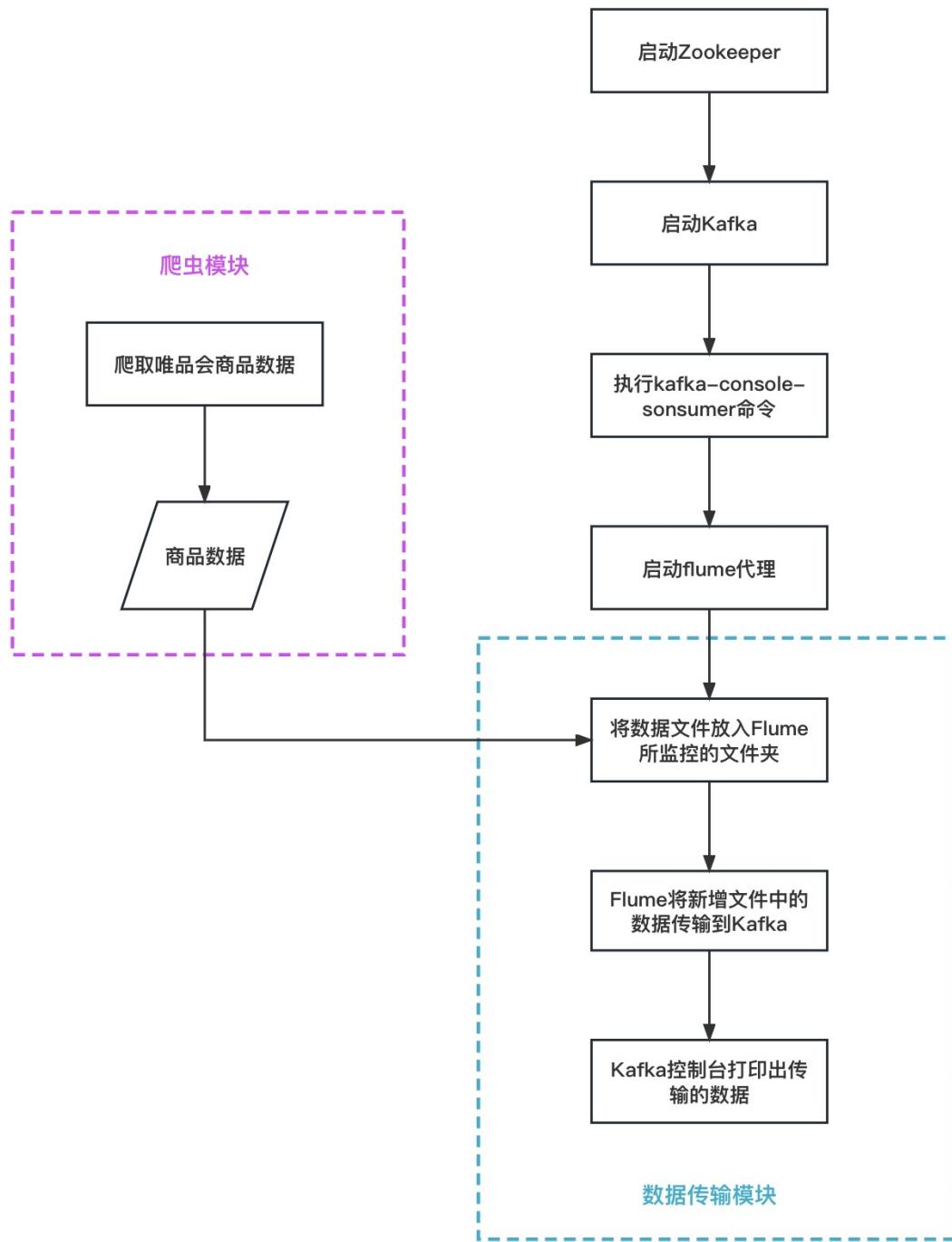
- a. Flume 检测到文件夹中的新增文件
- b. Flume 将新增 csv 文件中的数据传输到 Kafka
- c. Kafka 控制台打印出传输的数据

```
ljjz-ha@hadoop102:~/module/kafka_2.11-0.9.0.0/bin$ ./kafka-console-consumer.sh --  
zookeeper hadoop102:2181 --topic streamingtopic  
标题,品牌,原价,折扣,售价  
魅可MAC子弹头小辣椒/炫番茄chili口红#602,MAC,180,5.5折,99  
菁纯丝绒雾面口红 #196 新款小蛮腰唇膏口红胡萝卜色,兰蔻,300,6.3折,189  
【滋润持久】小妖金口红持久水润显色豆沙色161开挂红666,欧莱雅,145,6.8折,99  
【小金笔】润泽显色唇膏口红滋润显色显白保湿细腻顺滑黄皮适用,自然堂,120,5.8折,69  
【小金笔】润泽显色唇膏口红滋润显色显白保湿细腻顺滑黄皮适用,自然堂,120,5.8折,69  
【滋润持久】小妖金口红持久水润显色豆沙色161开挂红666,欧莱雅,145,6.8折,99  
【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色,欧莱雅,145,6.8折,99  
【丝绒哑光】纷泽滋润口红女高级浓郁666开挂红显色,欧莱雅,145,6.8折,99  
【持色百搭】纷泽滋润细跟口红唇膏611裸色细管绒雾唇霜女,欧莱雅,175,5.7折,99  
【持久滋润】印迹唇釉口红女轻薄持久滋润显色大牌女全新,欧莱雅,145,7.5折,109  
【持久滋润】印迹唇釉口红女轻薄持久滋润显色大牌女全新,欧莱雅,145,7.5折,109  
【持色百搭】纷泽滋润细跟口红唇膏611裸色细管绒雾唇霜女,欧莱雅,175,5.7折,99  
【哑光持久】免年限定版纷泽滋润细跟口红绒雾唇霜高级显白必备,欧莱雅,175,7.9折,139  
【哑光持久】纷泽滋润细跟口红绒雾唇霜高级显白必备,欧莱雅,175,8.5折,149  
  
2023-06-02 16:36:58,948 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO -  
kafka.utils.Logging$class.info(Logging.scala:68)] Fetching metadata from broker  
id:0,host:hadoop102,port:9092 with correlation id 0 for 1 topic(s) Set(streaming  
topic)  
2023-06-02 16:36:58,967 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO -  
kafka.utils.Logging$class.info(Logging.scala:68)] Connected to hadoop102:9092 fo  
r producing
```

通过以上流程，爬虫生成的数据文件会通过 Flume 传输到 Kafka 中，实现了将爬虫数据同步至 Kafka 的功能

流程图

SpiderFlowKafkaSync
爬虫数据通过Flume同步到Kafka
流程图



5.8 DolphinScheduler 任务调度系统实操

5.8.1 部署

安装进程管理工具包 psmisc

```
[hadoop@Master ~]$ sudo apt-get install psmisc
```

下载并解压安装包到目标目录，并改名

```
[hadoop@Slave2 ~]$ wget https://dlcdn.apache.org/dolphinscheduler/3.1.7/apache-dolphinscheduler-3.1.7-bin.tar.gz  
[hadoop@Master module]$ sudo tar -zxvf apache-dolphinscheduler-3.1.7-bin.tar.gz -C /opt/module/  
[hadoop@Master module]$ sudo mv apache-dolphinscheduler-3.1.7-bin/ dolphinscheduler-3.1.7
```

DolphinScheduler 元数据存储在关系型数据库中，故需创建相应的数据库和用户，创建数据库和用户并赋予相应权限。

```
mysql> CREATE DATABASE dolphinscheduler DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;  
mysql> CREATE USER 'dolphinscheduler'@'%' IDENTIFIED BY 'dolphinscheduler';  
mysql> GRANT ALL PRIVILEGES ON dolphinscheduler.* TO 'dolphinscheduler'@'%';  
mysql> flush privileges;
```

修改数据源配置文件，修改 conf 目录下的 datasource.properties 文件

```
[hadoop@Master:/opt/module/dolphinscheduler-3.1.7]$ vim conf/datasource.properties
```

修改内容如下：

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://Master:3306/dolphinscheduler?useUnicode=true&characterEncoding=UTF-8  
spring.datasource.username=dolphinscheduler  
spring.datasource.password=dolphinscheduler
```

拷贝 MySQL 驱动到 DolphinScheduler 的解压目录下的 lib 中

```
[hadoop@Master:/opt/module/dolphinscheduler-3.1.7]$ cp /opt/module/mysql-connector-java-5.1.27-bin.jar lib/
```

执行数据库初始化脚本

```
[hadoop@Master:/opt/module/dolphinscheduler-3.1.7]$ ./script/create-dolphinscheduler.sh
```

配置一键部署脚本，修改 conf/config 目录下的 install_config.conf 文件

```
[hadoop@Master:/opt/module/dolphinscheduler-3.1.7]$ vim conf/config/install_config.conf
```

做如下修改：

```

# postgresql or mysql
dbtype="mysql"

# db config
# db address and port
dbhost="Master:3306"

# db username
username="dolphinscheduler"

# database name
dbname="dolphinscheduler"

# db passwprd
# NOTICE: if there are special characters, please use the \ to escape, for example, `[' escape to `\\['
password="dolphinscheduler"

# zk cluster
zkQuorum="Master:2181,Slave1:2181,Slave2:2181"

# Note: the target installation path for dolphinscheduler, please not config as the same as the current path (pwd)
installPath="/opt/module/dolphinscheduler"

# deployment user
# Note: the deployment user needs to have sudo privileges and permissions to operate hdfs. If hdfs is enabled, the root directory needs to be created by itself
deployUser="hadoop"

# resource storage type: HDFS, S3, NONE
resourceStorageType="HDFS"

# resource store on HDFS/S3 path, resource file will store to this hadoop hdfs path, self configuration, please make sure the directory exists on hdfs and have read write permissions. "/dolphinscheduler" is recommended
resourceUploadPath="/dolphinscheduler"

# if resourceStorageType is HDFS, defaultFS write namenode address, HA you need to put core-site.xml and hdfs-site.xml in the conf directory.
# if S3, write S3 address, HA, for example : s3a://dolphinscheduler,
# Note, s3 be sure to create the root directory /dolphinscheduler
defaultFS="hdfs://Master:8020"

# resourcemanager port, the default value is 8088 if not specified
resourceManagerHttpAddressPort="8088"

# postgresql or mysql
dbtype="mysql"

# db config
# db address and port
dbhost="Master:3306"

# db username
username="dolphinscheduler"

# database name
dbname="dolphinscheduler"

# db passwprd

```

```

# NOTICE: if there are special characters, please use the \ to escape, for example, `\[` escape to `\\\[`
password="dolphinscheduler"

# zk cluster
zkQuorum="Master:2181,Slave1:2181,Slave2:2181"

# Note: the target installation path for dolphinscheduler, please not config as the same as the current path
(pwd)
installPath="/opt/module/dolphinscheduler"

# deployment user
# Note: the deployment user needs to have sudo privileges and permissions to operate hdfs. If hdfs is
enabled, the root directory needs to be created by itself
deployUser="hadoop"

# resource storage type: HDFS, S3, NONE
resourceStorageType="HDFS"

# resource store on HDFS/S3 path, resource file will store to this hadoop hdfs path, self configuration, please
make sure the directory exists on hdfs and have read write permissions. "/dolphinscheduler" is
recommended
resourceUploadPath="/dolphinscheduler"

# if resourceStorageType is HDFS, defaultFS write namenode address, HA you need to put core-site.xml
and hdfs-site.xml in the conf directory.
# if S3, write S3 address, HA, for example : s3a://dolphinscheduler,
# Note, s3 be sure to create the root directory /dolphinscheduler
defaultFS="hdfs://Master:8020"

# resourcemanager port, the default value is 8088 if not specified
resourceManagerHttpAddressPort="8088"

# if resourcemanager HA is enabled, please set the HA IPs; if resourcemanager is single, keep this value
empty
yarnHaIps=

# if resourcemanager HA is enabled or not use resourcemanager, please keep the default value; If
resourcemanager is single, you only need to replace ds1 to actual resourcemanager hostname
singleYarnIp="Slave1"

# who have permissions to create directory under HDFS/S3 root path
# Note: if kerberos is enabled, please config hdfsRootUser=
hdfsRootUser="hadoop"

# api server port
apiServerPort="12345"

# install hosts
# Note: install the scheduled hostname list. If it is pseudo-distributed, just write a pseudo-distributed
hostname
ips="Master,Slave1,Slave2"

```

```
# ssh port, default 22
# Note: if ssh port is not default, modify here
sshPort="22"

# run master machine
# Note: list of hosts hostname for deploying master
masters="Master"

# run worker machine
# note: need to write the worker group name of each worker, the default value is "default"
workers="Master:default,Slave1:default,Slave2:default"

# run alert machine
# note: list of machine hostnames for deploying alert server
alertServer="Master"

# run api machine
# note: list of machine hostnames for deploying api server
apiServers="Master"
```

启动 Zookeeper 集群

```
[hadoop@Master:/opt/module/dolphinscheduler-3.1.7]$ zk.sh start
```

一键部署并启动 DolphinScheduler

```
[hadoop@Master:/opt/module/dolphinscheduler-3.1.7]$ bin/install.sh
```

查看 DolphinScheduler 进程

```
----- Master -----
29139 ApiApplicationServer
28963 WorkerServer
3332 QuorumPeerMain
2100 DataNode
28902 MasterServer
29081 AlertServer
1978 NameNode
29018 LoggerServer
2493 NodeManager
29551 Jps
----- Slave1 -----
29568 Jps
29315 WorkerServer
2149 NodeManager
1977 ResourceManager
2969 QuorumPeerMain
29372 LoggerServer
1903 DataNode
----- Slave2 -----
1905 SecondaryNameNode
27074 WorkerServer
2050 NodeManager
2630 QuorumPeerMain
1817 DataNode
27354 Jps
27133 LoggerServer
```

5.8.2 网页端查看

进入 <http://master:12345/dolphinscheduler/ui/login> 访问 DolphinScheduler UI，初始用户的用户名为：admin，密码为 dolphinScheduler123



首页模块

This screenshot displays the main dashboard of the DolphinScheduler application. It includes two large circular donut charts in the center, each with a red ring. To the left of the charts is a table titled '任务状态统计' (Task Status Statistics) with data from June 26, 2023, 00:00:00 to 21:39:57. To the right is a similar table titled '流程状态统计' (Process Status Statistics) with the same time range. Both tables show counts for various states: 提交成功 (Submitted successfully), 正在运行 (Running), 暂停 (Paused), 失败 (Failed), 成功 (Successful), 需要容错 (Requires tolerance), KILL, 延时执行 (Delayed execution), 强制成功 (Forced success), and 派发 (Dispatched).

#	数量	状态
1	0	提交成功
2	0	正在运行
3	0	暂停
4	0	失败
5	2	成功
6	0	需要容错
7	0	KILL
8	0	延时执行
9	0	强制成功
10	0	派发

#	数量	状态
1	0	提交成功
2	0	正在运行
3	0	暂停
4	0	失败
5	2	成功
6	0	需要容错
7	0	KILL
8	0	延时执行
9	0	派发



项目管理模块



数据源中心模块

用户管理

#	用户名	用户类型	租户	队列	邮件	手机	状态	操作
1	topage	普通用户	dolphinschduler	default	li@gmail.com		启用	
2	admin	管理员	dolphinschduler	default	xxx@qq.com		启用	

右侧工具栏：项目、资源、数据源、UDF函数、命名空间。

资源中心模块

文件管理

#	名称	所属用户	是否文件夹	文件名称	描述	大小	操作
1	test.py	admin	否	test.py		15 B	

项目工作流定义

DolphinScheduler 首页 项目... 资... 数... 数据... 监... 安... 深色 中文 UTC admin

项目概览 工作流 工作流关系 工作流定义 工作流实例 任务 任务定义 任务实例

工作流定义

创建工作流 导入工作流 搜索框

工作流定义

#	工作流名称	状态	操作
1	shell工作流_copy_202301...	下线	
2	shell工作流	下线	
3	echo2	上线	
4	echo1	上线	

删除 导出 批量复制 1 / 10 / 页 跳至

DolphinScheduler 首页 项目管理 资源中心 数据质量 数据源中心 监控中心 安全中心 深色 中文 UTC admin

项目概览 工作流 工作流关系 工作流定义 工作流实例 任务 任务定义 任务实例

工作流定义

shell工作流 (已上线) 搜索框

SHELL SUB_PROCESS PROCEDURE SQL SPARK FUNK MapReduce PYTHON DEPENDENT HTTP DataX PIGEON SQOOP CONDITIONS

监控中心模块

DolphinScheduler 首页 项目管理 资源中心 数据质量 数据源中心 监控中心 安全中心 深色 中文 UTC admin

服务管理 Master Worker DB 统计管理

主机: 192.168.0.158 目录详情 创建时间: 2023-06-26 03:14:15 最后心跳时间: 2023-06-26 17:30:17

处理器使用量	内存使用量	磁盘可用容量	平均负载量
			27.68 0.08