[第一章 分布式计算框架与资源调度 6](#_Toc515544216)

[1.1 分布式计算框架 6](#_Toc515544217)

[1.1.1 编程模型 6](#_Toc515544218)

[1.1.2 特殊的组件partitioner与combiner 11](#_Toc515544219)

[1.1.3 用mr进行数据的排序，然后求出topN 14](#_Toc515544220)

[1.1.4 MapReduce 求两个人的共同好友算法 17](#_Toc515544221)

[1.2 分布式资源调度框架 21](#_Toc515544222)

[1.2.1 yarn的概念 21](#_Toc515544223)

[1.2.2 yarn的架构 22](#_Toc515544224)

[1.2.3 yarn的工作流程 22](#_Toc515544225)

[1.2.4 yarn的调度器 Scheduler 24](#_Toc515544226)

[1.3 分布式文件存储系统 26](#_Toc515544227)

[1.3.1 架构 26](#_Toc515544228)

[1.3.2 原理 27](#_Toc515544229)

[1.3.3 API 30](#_Toc515544230)

[1.4 项目 45](#_Toc515544231)

[1.4.1 点击流日志模型 45](#_Toc515544232)

[1.4.2 推荐系统项目 54](#_Toc515544233)

[1.5 相关面试题 57](#_Toc515544234)

[第2章 数据分析及其步骤 59](#_Toc515544235)

[2.1 数据分析定义 59](#_Toc515544236)

[2.2 数据分析的作用 59](#_Toc515544237)

[2.2.1 现状分析 59](#_Toc515544238)

[2.2.2 原因分析 60](#_Toc515544239)

[2.2.3 预测分析 60](#_Toc515544240)

[2.3 数据分析基本步骤（重点） 60](#_Toc515544241)

[2.3.1 明确分析目的和思路 61](#_Toc515544242)

[2.3.2 数据收集 65](#_Toc515544243)

[2.3.3 数据处理 66](#_Toc515544244)

[2.3.4 数据分析 66](#_Toc515544245)

[2.3.5 数据展现（数据可视化） 67](#_Toc515544246)

[2.3.6 报告撰写 67](#_Toc515544247)

[2.4 行业前景 68](#_Toc515544248)

[2.4.1 蓬勃发展的趋势 68](#_Toc515544249)

[2.4.2 数据分析师的职业要求 69](#_Toc515544250)

[2.5 大数据时代 70](#_Toc515544251)

[2.5.1 大数据的含义 70](#_Toc515544252)

[2.5.2 产生背景 70](#_Toc515544253)

[2.5.3 影响 71](#_Toc515544254)

[2.5.4 特征 73](#_Toc515544255)

[2.5.5 思维变革 74](#_Toc515544256)

[2.5.6 第三个思维变革： 74](#_Toc515544257)

[不是所有的事情都必须知道现象背后的原因，而是要让数据自己“发声”，即不是因果关系，而是相关关系。 74](#_Toc515544258)

[2.5.7 科技发展带来的挑战 74](#_Toc515544259)

[2.5.8 分布式系统 75](#_Toc515544260)

[2.6 面试题: 76](#_Toc515544261)

[2.6.1 分布式和集群有啥区别呢？ 76](#_Toc515544262)

[2.6.2 集群 负载均衡 分布式 有什么区别？ 77](#_Toc515544263)

[2.6.3 Web流量日志数据自定义采集 78](#_Toc515544264)

[2.7 Hive 87](#_Toc515544265)

[2.7.1 Hive的基本简介 87](#_Toc515544266)

[2.7.2 Hive数仓开发的基本流程。 88](#_Toc515544267)

[2.7.3 Hive sql 知识点。 90](#_Toc515544268)

[2.8 Hive面试题 94](#_Toc515544269)

[2.8.1 Hive常见 94](#_Toc515544270)

[2.8.2 海量数据处理方法 152](#_Toc515544271)

[2.9 Hadoop HA 165](#_Toc515544272)

[2.9.1 知识点 165](#_Toc515544273)

[2.9.2 上课用图 174](#_Toc515544274)

[2.9.3 3、 面试题 175](#_Toc515544275)

[2.10 Hadoop的联邦机制 176](#_Toc515544276)

[2.10.1 为什么会出现联邦？ 176](#_Toc515544277)

[2.10.2 联邦的实现 176](#_Toc515544278)

[2.10.3 主要优点： 177](#_Toc515544279)

[2.10.4 配置： 178](#_Toc515544280)

[2.10.5 操作 179](#_Toc515544281)

[2.11 项目 180](#_Toc515544282)

[2.11.1 点击流日志模型 180](#_Toc515544283)

[2.11.2 相关面试题 189](#_Toc515544284)

[2.11.3 推荐系统项目 189](#_Toc515544285)

[第3章 Storm 192](#_Toc515544286)

[3.1 架构 193](#_Toc515544287)

[3.1.1 Nimbus 193](#_Toc515544288)

[3.1.2 Zookeeper 193](#_Toc515544289)

[3.1.3 supervisor 193](#_Toc515544290)

[3.1.4 worker 193](#_Toc515544291)

[3.2 编程模型 194](#_Toc515544292)

[3.2.1 Spout 194](#_Toc515544293)

[3.2.2 Bolt 194](#_Toc515544294)

[3.2.3 并行度 194](#_Toc515544295)

[3.2.4 消息不丢失 195](#_Toc515544296)

[3.3 Storm 物联网wifi项目 195](#_Toc515544297)

[3.3.1 wifi项目背景 195](#_Toc515544298)

[3.3.2 2)数据来源(采集) 196](#_Toc515544299)

[3.3.3 集群规模配置 197](#_Toc515544300)

[3.3.4 数据处理流程 197](#_Toc515544301)

[3.3.5 数据计算流程 199](#_Toc515544302)

[3.4 storm实时看板案例 199](#_Toc515544303)

[3.4.1 需求分析 199](#_Toc515544304)

[3.4.2 确定数据源 200](#_Toc515544305)

[3.4.3 确定采集方案 200](#_Toc515544306)

[3.4.4 确定存储 201](#_Toc515544307)

[3.4.5 数据计算 201](#_Toc515544308)

[3.4.6 展现 201](#_Toc515544309)

[3.5 storm日志监控告警系统 202](#_Toc515544310)

[3.5.1 需求分析 202](#_Toc515544311)

[3.5.2 确定数据源 202](#_Toc515544312)

[3.5.3 确定采集方案 202](#_Toc515544313)

[3.5.4 确定存储 203](#_Toc515544314)

[3.5.5 数据计算 203](#_Toc515544315)

[3.6 Storm的框架 203](#_Toc515544316)

[3.7 面试题 204](#_Toc515544317)

[第4章 Kafka 204](#_Toc515544318)

[4.1 kafka的介绍 204](#_Toc515544319)

[4.2 kafka的架构 207](#_Toc515544320)

[4.3 kafka集群的安装与搭建 210](#_Toc515544321)

[4.4 kafkfa的原理 213](#_Toc515544322)

[4.5 kafka的API使用 216](#_Toc515544323)

[4.6 kafka与其他的整合使用 220](#_Toc515544324)

[第5章 HBASE数据库 237](#_Toc515544325)

[5.1 数据库 OLAP、OLTP的介绍和比较 237](#_Toc515544326)

[5.2 Hbase基础 238](#_Toc515544327)

[什么是OLTP 238](#_Toc515544328)

[5.2.1 hbase数据库介绍 241](#_Toc515544329)

[5.2.2 hbase集群结构 244](#_Toc515544330)

[5.2.3 hbase集群搭建 (简易步骤) 245](#_Toc515544331)

[5.2.4 命令行演示 245](#_Toc515544332)

[5.2.5 hbase代码开发（基本，过滤器查询） 246](#_Toc515544333)

[5.2.6 hbase内部原理 247](#_Toc515544334)

[5.2.7 物理存储 249](#_Toc515544335)

[5.2.8 寻址机制 253](#_Toc515544336)

[5.2.9 读写过程 254](#_Toc515544337)

[5.2.10 Region管理 256](#_Toc515544338)

[5.2.11 Master工作机制 257](#_Toc515544339)

[5.2.12 HBase容错性 257](#_Toc515544340)

[5.3 Hbase高级应用 258](#_Toc515544341)

[5.3.1 建表高级属性 258](#_Toc515544342)

[5.3.2 HBase的设计原则 260](#_Toc515544343)

[5.3.3 rowkey长度原则 260](#_Toc515544344)

[5.3.4 什么是热点 261](#_Toc515544345)

[1）加盐 261](#_Toc515544346)

[2）哈希 262](#_Toc515544347)

[3）反转 262](#_Toc515544348)

[4）时间戳反转 262](#_Toc515544349)

[5.4 HBase性能优化完全版 263](#_Toc515544350)

[5.4.1 垃圾回收优化 263](#_Toc515544351)

[5.4.2 启用压缩，详情自行搜索，暂时未曾尝试，后面持续更新。 264](#_Toc515544352)

[5.4.3 优化Region拆分合并以及与拆分Region 264](#_Toc515544353)

[5.4.4 4.客户端入库调优 265](#_Toc515544354)

[5.4.5 HBase配置文件 266](#_Toc515544355)

[5.4.6 HDFS优化部分 270](#_Toc515544356)

[5.5 大数据Hbase 面试题 271](#_Toc515544357)

[第6章 spark 277](#_Toc515544358)

[6.1 什么是Spark 277](#_Toc515544359)

[6.1.1 Spark的特点及相对于MapReduce的优势 277](#_Toc515544360)

[6.1.2 Spark HA高可用部署 279](#_Toc515544361)

[6.1.3 Spark的角色 280](#_Toc515544362)

[6.1.4 Spark程序提交任务模式 281](#_Toc515544363)

[6.1.5 Spark-Shell 282](#_Toc515544364)

[6.1.6 在IDEA中编写WordCount程序 283](#_Toc515544365)

[6.2 Spark Streaming 283](#_Toc515544366)

[6.2.1 Spark Streming的特性 283](#_Toc515544367)

[6.2.2 Spark Streaming 对比 Storm 283](#_Toc515544368)

[6.3 算子操作 283](#_Toc515544369)

[6.3.1 Transformations 283](#_Toc515544370)

[6.3.2 OutputOperations 285](#_Toc515544371)

[6.3.3 Spark Streming编程实战 286](#_Toc515544372)

[6.3.4 SparkStreaming整合flume 286](#_Toc515544373)

[6.3.5 SparkStreaming整合Kafka 286](#_Toc515544374)

[6.3.6 Checkpoint 287](#_Toc515544375)

[6.4 Spark SQL 288](#_Toc515544376)

[6.4.1 几个知识点 288](#_Toc515544377)

[6.4.2 002 以编程方式执行Spark SQL查询 289](#_Toc515544378)

[6.4.3 JDBC数据源 291](#_Toc515544379)

[6.5 SparkRDD 291](#_Toc515544380)

[6.5.1 了解SparkRDD 291](#_Toc515544381)

[1）什么是RDD 291](#_Toc515544382)

[2）RDD的属性 292](#_Toc515544383)

[3.为什么会产生RDD？ 293](#_Toc515544384)

[4） RDD在Spark中的地位及作用 293](#_Toc515544385)

[6.5.2 创建RDD 294](#_Toc515544386)

[6.5.3 RDD编程API 295](#_Toc515544387)

[6.5.4 RDD常用的算子操作 297](#_Toc515544388)

[6.5.5 RDD的依赖关系 300](#_Toc515544389)

[6.5.6 RDD的缓存 301](#_Toc515544390)

[6.5.7 DAG的生成 302](#_Toc515544391)

[6.5.8 Spark任务调度 303](#_Toc515544392)

[6.6 RDD容错机制之checkpoint 305](#_Toc515544393)

[6.6.1 checkpoint是什么 305](#_Toc515544394)

[6.6.2 checkpoint原理机制 305](#_Toc515544395)

[6.6.3 Checkpoint常见面试问题 312](#_Toc515544396)

[6.7 Spark运行架构 316](#_Toc515544397)

[6.7.1 Spark运行基本流程 316](#_Toc515544398)

[6.7.2 Spark运行架构特点 317](#_Toc515544399)

[6.8 开发调优 323](#_Toc515544400)

[6.8.1 调优概述 323](#_Toc515544401)

[6.8.2 原则一：避免创建重复的RDD 323](#_Toc515544402)

[6.8.3 原则二：尽可能复用同一个RDD 324](#_Toc515544403)

[6.8.4 原则三：对多次使用的RDD进行持久化 325](#_Toc515544404)

[6.8.5 原则四：尽量避免使用shuffle类算子 327](#_Toc515544405)

[6.8.6 原则五：使用map-side预聚合的shuffle操作 328](#_Toc515544406)

[6.8.7 原则六：使用高性能的算子 330](#_Toc515544407)

[6.8.8 原则七：广播大变量 331](#_Toc515544408)

[6.8.9 原则八：使用Kryo优化序列化性能 332](#_Toc515544409)

[6.8.10 原则九：优化数据结构 332](#_Toc515544410)

[6.9 资源调优 333](#_Toc515544411)

[6.9.1 调优概述 333](#_Toc515544412)

[6.9.2 资源参数调优 335](#_Toc515544413)

[6.9.3 资源参数参考示例 339](#_Toc515544414)

[6.10 数据倾斜调优 340](#_Toc515544415)

[6.10.1 调优概述 340](#_Toc515544416)

[6.10.2 数据倾斜发生时的现象 340](#_Toc515544417)

[6.10.3 数据倾斜发生的原理 340](#_Toc515544418)

[6.10.4 如何定位导致数据倾斜的代码 341](#_Toc515544419)

[6.10.5 查看导致数据倾斜的key的数据分布情况 344](#_Toc515544420)

[6.10.6 数据倾斜的解决方案 345](#_Toc515544421)

[6.11 shuffle调优 358](#_Toc515544422)

[6.11.1 调优概述 358](#_Toc515544423)

[6.11.2 ShuffleManager发展概述 358](#_Toc515544424)

[6.11.3 HashShuffleManager运行原理 358](#_Toc515544425)

[6.11.4 SortShuffleManager运行原理 360](#_Toc515544426)

[6.11.5 shuffle相关参数调优 362](#_Toc515544427)

[6.12 Spark面试题汇总 365](#_Toc515544428)

[1. spark中的RDD是什么，有哪些特性 366](#_Toc515544429)

[2. 概述一下spark中的常用算子区别（map、mapPartitions、foreach、foreachPartition） 366](#_Toc515544430)

[3. 谈谈spark中的宽窄依赖 366](#_Toc515544431)

[4. spark中如何划分stage 367](#_Toc515544432)

[5. spark-submit的时候如何引入外部jar包 367](#_Toc515544433)

[6. spark 如何防止内存溢出 367](#_Toc515544434)

[7. spark中cache和persist的区别 368](#_Toc515544435)

[8. 简要描述Spark分布式集群搭建的步骤 368](#_Toc515544436)

[9. spark中的数据倾斜的现象、原因、后果 368](#_Toc515544437)

[10. 如何解决spark中的数据倾斜问题 369](#_Toc515544438)

[11. flume整合sparkStreaming问题 370](#_Toc515544439)

[12. kafka整合sparkStreaming问题 372](#_Toc515544440)

[6.12.1 ----简答题 ---- 网上资料 --- 374](#_Toc515544441)

[6.12.2 -------Spark on Yarn面试篇 379](#_Toc515544442)

[6.12.3 -------spark sql 面试篇 383](#_Toc515544443)

[6.12.4 ----选择题 --- 383](#_Toc515544444)

[6.12.5 补充资料: (spark 集群standalone + spark on yarn) 385](#_Toc515544445)

## **第一章 分布式计算框架与资源调度**

## 分布式计算框架

### 编程模型

##### inputformat

在MapReduce程序的开发过程中，往往需要用到FileInputFormat与TextInputFormat，我们会发现TextInputFormat这个类继承自FileInputFormat，FileInputFormat这个类继承自InputFormat，InputFormat这个类会将文件file按照逻辑进行划分，划分成的每一个split切片将会被分配给一个Mapper任务,文件先被切分成split块，而后每一个split切片对应一个Mapper任务

FileInputFormat的划分机制:

A. 简单地按照文件的内容长度进行切片

B. 切片大小，默认等于 block 大小

C. 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

默认情况下， split size =block size,在 hadoop 2.x 中为 128M。

注意:bytesRemaining/splitSize > 1.1 不满足的话，那么最后所有剩余的会作为一个切片。从而不会形成例如 129M 文件规划成两个切片的局面。

##### MaTask端的工作机制

input File 通过 split 被逻辑切分为多个 split 文件，通过 Record 按行读取内容给 map（用户自己实现的）进行处理，数据被 map 处理结束之后交给 OutputCollector 收集器，对其结果 key 进行分区（默认使用 hash 分区），然后写入 buffer，每个 map task 都有一个内存缓冲区，存储着 map 的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个 map task 结束后再对磁盘中这个 map task 产生的所有临时文件做合并，生成最终的正式输出文件，然后等待reduce task 来拉数据。 Map端的输入的(k,v)分别是该行的起始偏移量,以及每一行的数据内容,map端的输出(k,v)可以根据需求进行自定义,但是如果输出的是javabean对象,需要对javabean继承writable

##### shuffle的过程

shuffle的过程是:Map产生输出开始到Reduc取得数据作为输入之前的过程称作shuffle.

1).Collect 阶段：将 MapTask 的结果输出到默认大小为 100M 的环形缓冲区，保存的是 key/value，Partition 分区信息等。

2).Spill 阶段：当内存中的数据量达到一定的阀值的时候，就会将数据写入本地磁盘，在将数据写入磁盘之前需要对数据进行一次排序的操作，如果配置了 combiner，还会将有相同分区号和 key 的数据进行排序。

3).Merge 阶段：把所有溢出的临时文件进行一次合并操作，以确保一个

MapTask 最终只产生一个中间数据文件。

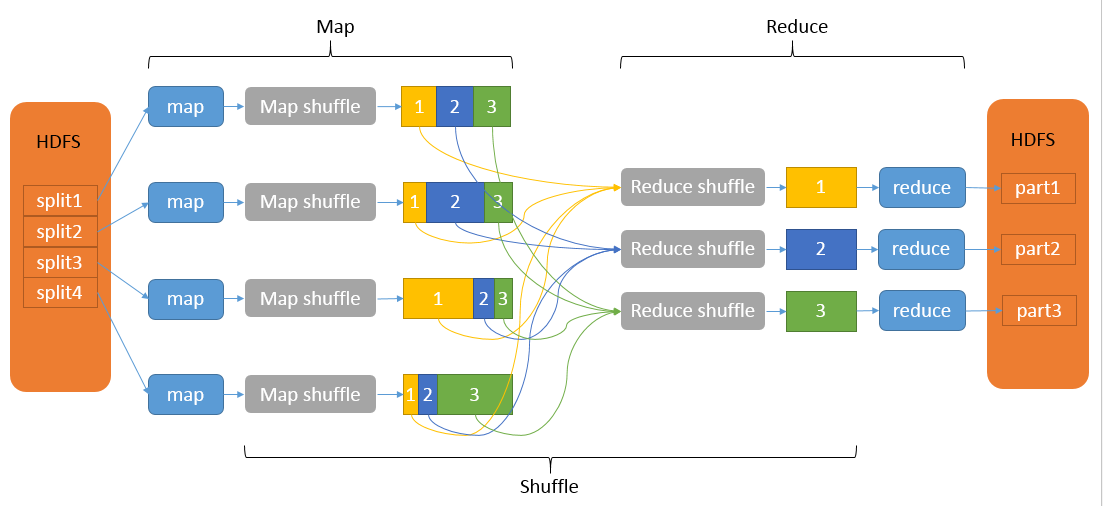
4).Copy 阶段： ReduceTask 启动 Fetcher 线程到已经完成 MapTask 的节点上复制一份属于自己的数据，这些数据默认会保存在内存的缓冲区中，当内存的缓冲区达到一定的阀值的时候，就会将数据写到磁盘之上。

5).Merge 阶段：在 ReduceTask 远程复制数据的同时，会在后台开启两个线

程对内存到本地的数据文件进行合并操作。

6).Sort 阶段：在对数据进行合并的同时，会进行排序操作，由于 MapTask阶段已经对数据进行了局部的排序，ReduceTask 只需保证 Copy 的数据的最终整体有效性即可。

Shuffle 中的缓冲区大小会影响到 mapreduce 程序的执行效率，原则上说，缓冲区越大，磁盘 io 的次数越少，执行速度就越快缓冲区的大小可以通过参数调整, 参数：io.sort.mb 默认 100M



##### reduceTask

  reducer将已经分好组的数据作为输入，并依次为每个键对应分组执行reduce函数。reduce函数的输入是键以及包含与该键对应的所有值的迭代器。

reduce端的输入是map端的输出,它的输出的(k,v)根据需求进行自定义

reducetask 并行度同样影响整个 job 的执行并发度和执行效率，与 maptask

的并发数由切片数决定不同，Reducetask 数量的决定是可以直接手动设置：

job.setNumReduceTasks(4);

如果数据分布不均匀，就有可能在 reduce 阶段产生数据倾斜。

默认的reduceTask的是1

\*Task并行度经验之谈：

最好每个 task 的执行时间至少一分钟。如果 job 的每个 map 或者 reduce task 的运行时间都只有 30-40 秒钟，那么就减少该 job 的 map 或者 reduce 数，每一个 task(map|reduce)的 setup 和加入到调度器中进行调度，这个中间的过程可能都要花费几秒钟，所以如果每个task 都非常快就跑完了，就会在 task 的开始和结束的时候浪费太多的时间。

默认情况下，每一个 task 都是一个新的 JVM 实例，都需要开启和销

毁的开销。在一些情况下，JVM 开启和销毁的时间可能会比实际处理数据的时间

要消耗的长，配置 task 的 M JVM 重用可以改善该问题：

（mapred.job.reuse.jvm.num.tasks，默认是 1，表示一个 JVM 上最多可以

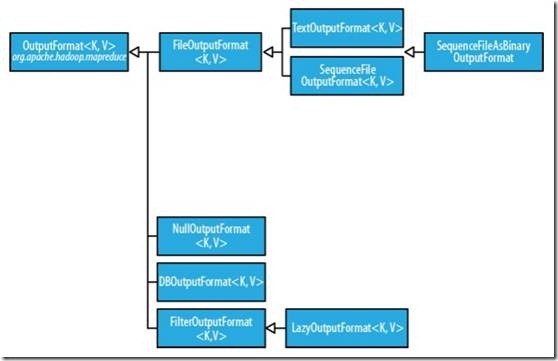
顺序执行的 task 数目（属于同一个 Job）是 1。也就是说一个 task 启一个 JVM）

如果 input 的文件非常的大，比如 1TB，可以考虑将 hdfs 上的每个 block

size 设大，比如设成 256MB 或者 512MB

##### outputformat

OutputFormat主要用于描述输出数据的格式，它能够将用户提供的key/value对写入特定格式的文件中。Hadoop 自带了很多 OutputFormat 的实现，它们与InputFormat实现相对应，足够满足我们业务的需要。 OutputFormat 类的层次结构如下图所示



OutputFormat是MapReduce输出的基类，所有MapReduce输出都实现了 OutputFormat 接口,主要有：

TextInputFormat 、SequenceFileOutputFormat、MultipleOutputs、DBOutputFormat等

### 特殊的组件partitioner与combiner

##### partitioner定义

partitioner的作用是将mapper（如果使用了combiner的话就是combiner）输出的key/value拆分为分片（shard），每个reducer对应一个分片。默认情况下，partitioner先计算key的散列值（通常为md5值）。然后通过reducer个数执行取模运算：key.hashCode%(reducer个数)。这种方式不仅能够随机地将整个key空间平均分发给每个reducer,同时也能确保不同mapper产生的相同key能被分发到同一个reducer。也可以自定义分区去继承partition<key,value>把不同的结果写入不同的文件中

分区Partitioner主要作用在于以下两点

（1）根据业务需要，产生多个输出文件；

（2）多个reduce任务并发运行，提高整体job的运行效率

适用范围：

需要非常注意的是：必须提前知道有多少个分区。比如自定义Partitioner会返回5个不同int值，而reducer number设置了小于5，那就会报错。所以我们可以通过运行分析任务来确定分区数。

##### map端的combine组件

每一个 map 都可能会产生大量的本地输出，Combiner 的作用就是对 map 端的输出先做一次合并，以减少在 map 和 reduce 节点之间的数据传输量，以提高网络 IO 性能，是 MapReduce 的一种优化手段之一

combiner 是 MR 程序中 Mapper 和 Reducer 之外的一种组件

 combiner 组件的父类就是 Reducer

 combiner 和 reducer 的区别在于运行的位置：

combiner 是在每一个 maptask 所在的节点运行

reducer 是接收全局所有 Mapper 的输出结果；

 combiner 的意义就是对每一个 maptask 的输出进行局部汇总，以减小网络传输量

 具体实现步骤：

1)自定义一个 combiner 继承 Reducer，重写 reduce 方法

2)中设置： job.setCombinerClass(CustomCombiner.class)

 combiner 能够应用的前提是不能影响最终的业务逻辑，而且，combine

输出 kv 应该跟 reducer 的输入 kv 类型要对应起来

Combiner使用需要注意的是：

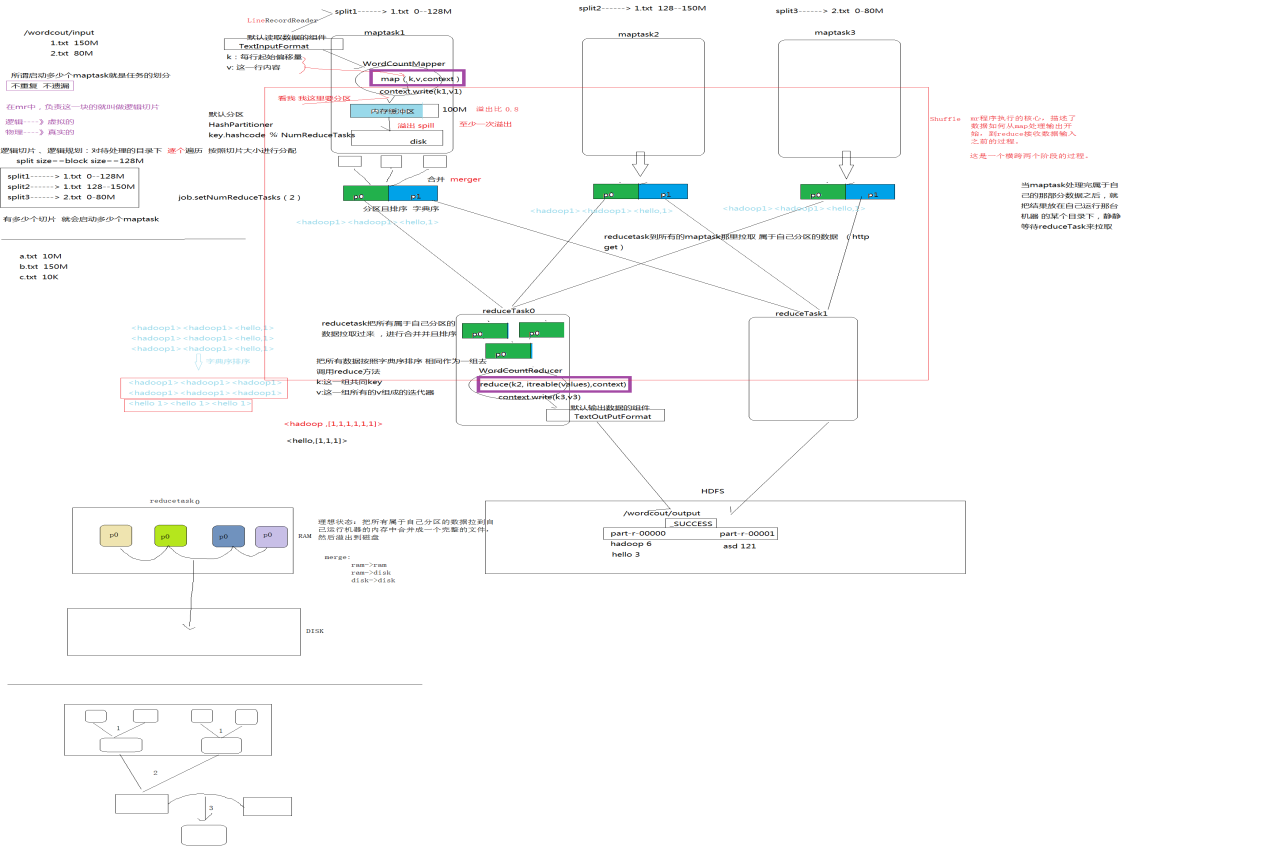
1.有很多人认为这个combiner和map输出的数据合并是一个过程，其实不然，map输出的数据合并只会产生在有数据spill出的时候，即进行merge操作。

2.与mapper与reducer不同的是，combiner没有默认的实现，需要显式的设置在conf中才有作用。

3.并不是所有的job都适用combiner，只有操作满足结合律的才可设置combiner。combine操作类似于：opt(opt(1, 2, 3), opt(4, 5, 6))。如果opt为求和、求最大值的话，可以使用，但是如果是求中值的话，不适用。

4.一般来说，combiner和reducer它们俩进行同样的操作。

分布式计算的整个流程分析如下图所示：



### 用mr进行数据的排序，然后求出topN

1 .需求

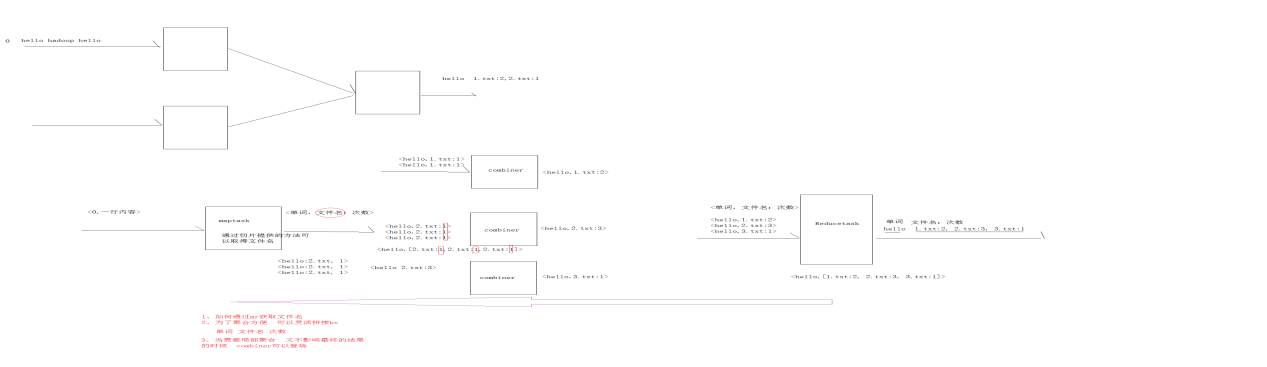
在得出统计每一个用户（手机号）所耗费的总上行流量、下行流量，总流量

结果的基础之上再加一个需求：将统计结果按照总流量倒序排序。

2.分析

基本思路：

实现自定义的 n bean 来封装流量信息，并将 n bean 作为 p map 输出的 y key来传输MR 程序在处理数据的过程中会对数据排序(map 输出的 kv 对传输到 reduce之前，会排序)，排序的依据是 map 输出的 key。所以，我们如果要实现自己需要的排序规则，则可以考虑将排序因素放到 key 中，让 key 实现接口：WritableComparable，然后重写 key 的 compareTo 方法。



3.实现自定义的 bean

public class FlowBean implements WritableComparable<FlowBean>{

private long upFlow;

private long downFlow;

private long sumFlow;

//这里反序列的时候会用到

public FlowBean() {

}

public FlowBean(long upFlow, long downFlow, long sumFlow) {

this.upFlow = upFlow;

this.downFlow = downFlow;

this.sumFlow = sumFlow;

}

public FlowBean(long upFlow, long downFlow) {

this.upFlow = upFlow;

this.downFlow = downFlow;

this.sumFlow = upFlow+downFlow;

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

}

public void set(long upFlow, long downFlow) {

this.upFlow = upFlow;

this.downFlow = downFlow;

this.sumFlow = upFlow+downFlow;

}

@Override

public String to String() {

return upFlow+"\t"+downFlow+"\t"+sumFlow;

}

/这里是序列化方法

@Override

public void write(DataOutput out) throws IOException {

out.writeLong(upFlow);

out.writeLong(downFlow);

out.writeLong(sumFlow);

}

/这里是反序列化方法

@Override

public void readFields(DataInput in) throws IOException {

//注意反序列化的顺序跟序列化的顺序一致

this.upFlow = in.readLong();

this.downFlow = in.readLong();

this.sumFlow = in.readLong();

}

//这里进行 bean 的自定义比较大小

@Override

public int compareTo (FlowBean o) {

//实现按照 sumflow 的大小倒序排序

return this.sumFlow>o.getSumFlow()?-1:1;

}

}

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

public class FlowSumMapper extends Mapper<LongWritable, Text, Text, FlowBean>{

Text k = new Text();

FlowBean v = new FlowBean();

@Override

protected void map(LongWritable key, Text value,Context context)

throws IOException, InterruptedException {

String line = value.toString();

String[] fields = line.split("\t");

String phoneNum = fields[1];

long upFlow = Long.parseLong(fields[fields.length-3]);

long downFlow = Long.parseLong(fields[fields.length-2]);

k.set(phoneNum);

v.set(upFlow, downFlow);

context.write(k, v);

}

}

public class FlowSumReducer extends Reducer<Text, FlowBean, Text, FlowBean> {

FlowBean v = new FlowBean();

@Override

protected void reduce(Text key, Iterable<FlowBean> values, Context context)

throws IOException, InterruptedException {

long upFlowCount = 0;

long downFlowCount = 0;

for (FlowBean bean : values) {

upFlowCount += bean.getUpFlow();

downFlowCount += bean.getDownFlow();

}

v.set(upFlowCount, downFlowCount);

context.write(key, v);

}

}

### MapReduce 求两个人的共同好友算法

需求

以下是qq的好友列表数据，冒号前是一个用户，冒号后是该用户的所有好友（数据中的好友关系是单向的）

A:B,C,D,F,E,O

B:A,C,E,K

C:F,A,D,I

D:A,E,F,L

E:B,C,D,M,L

F:A,B,C,D,E,O,M

G:A,C,D,E,F

H:A,C,D,E,O

I:A,O

J:B,O

K:A,C,D

L:D,E,F

M:E,F,G

O:A,H,I,J

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

思路

需求中给出的传入数据格式为：

用户：该用户拥有的好友们

user1：frined1，friend2，friend3……

user2：frined1，friend2，friend3……

user3：frined1，friend2，friend3……

user4：frined1，friend2，friend3……

……

要求传出的格式为：

两个用户：两个用户的共同好友

user1-user2：friend1，friend2，friend3……

user1-user3：friend1，friend2，friend3……

user1-user4：friend1，friend2，friend3……

user2-user3：friend1，friend2，friend3……

user2-user4：friend1，friend2，friend3……

user3-user4：friend1，friend2，friend3……

……

我们可以先将传入数据格式转换成：

所有用户的好友们：拥有该好友的用户

friend1：user1，user2，user3，user4……

friend2：user1，user2，user3，user4……

friend3：user1，user2，user3，user4……

……

再转换成：

两个用户：两个用户的共同好友

user1-user2：friend1，friend2，friend3……

user1-user3：friend1，friend2，friend3……

user1-user4：friend1，friend2，friend3……

user2-user3：friend1，friend2，friend3……

user2-user4：friend1，friend2，friend3……

user3-user4：friend1，friend2，friend3……

…….

代码实现

第一个Mapper：

package com.xianshun.mapper;

import java.io.IOException;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

public class CommonFriendsStepOneMapper extends Mapper<LongWritable, Text, Text, Text>{

protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)

throws IOException, InterruptedException {

// 传进来的 value: A:B,C,D,F,E,O

// 将 value 转换成字符串,line 表示行数据，为"A:B,C,D,F,E,O"

String line = value.toString();

// 分割字符串,得到用户和好友们，用 userAndFriends表示，为{"A","B,C,D,F,E,O"}

String[] userAndFriends = line.split(":");

// userAndFriends 0位置为用户，user为"A"

String user = userAndFriends[0];

// userAndFriends 1位置为好友们，这里以","分割分割字符串，得到每一个好友

// friend为{"B","C","D","F","E","O"}

String[] friends = userAndFriends[1].split(",");

// 循环遍历，以<B,A>,<C,A>,<D,A>......的形式传给reducer

for (String friend : friends) {

context.write(new Text(friend), new Text(user));

}

}

}

package com.xianshun.reducer;

import java.io.IOException;

import org.apache.hadoop.io.Text;import org.apache.hadoop.mapreduce.Reducer;

public class CommonFriendsStepOneReducer extends Reducer<Text, Text, Text, Text>{

protected void reduce(Text friend, Iterable<Text> users, Context context)

throws IOException, InterruptedException {

// 传进来的数据 <好友，用户> <B,A>,<C,A>,<D,A>,<A,B>,<C,B>,<E,B>,<K,B>......

// 新建 stringBuffer, 用于存放 拥有该好友的用户们

StringBuffer stringBuffer = new StringBuffer();

// 遍历所有的用户，并将用户放在stringBuffer中，以","分隔

for (Text user : users) {

stringBuffer.append(user).append(",");

}

// 以好友为key,用户们为value传给下一个mapper

context.write(friend, new Text(stringBuffer.toString()));

}

}

第一个MR结果：

A I,K,C,B,G,F,H,O,D,

B A,F,J,E,C A,E,B,H,F,G,K,D G,C,K,A,L,F,E,H,E G,M,L,H,A,F,B,D,

F L,M,D,C,G,A,

G M,

H O,I O,C,

J O,K B,

L D,E,

M E,F,O A,H,I,J,F,

第二个Mapper：

package com.xianshun.mapper;

import java.io.IOException;import java.util.Arrays;

import org.apache.hadoop.io.LongWritable;import org.apache.hadoop.io.Text;import org.apache.hadoop.mapreduce.Mapper;

public class CommonFriendsStepTwoMapper extends Mapper<LongWritable, Text, Text, Text>{

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

// 传进来的数据 A I,K,C,B,G,F,H,O,D,

// 将传进来的数据转换成字符串并以"\t"分割，得到fiendsAndUsers

String[] friendAndUsers = value.toString().split("\t");

// fiendsAndUsers 0位置为好友 "A"

String friend = friendAndUsers[0];

// fiendsAndUsers 1位置为拥有上面好友用户们

// 以 ","进行分割字符串，得到每一个用户,{"I","K",....}

String[] users = friendAndUsers[1].split(",");

// 将user进行排序，避免重复

Arrays.sort(users); // 以用户-用户为key，好友们做value传给reducer

for(int i=0; i<users.length-2; i++) {

for(int j=i+1; j<users.length-1; j++) {

context.write(new Text(users[i] + "-" + users[j]), new Text(friend));

}

}

}

}

第二个Reducer：

package com.xianshun.reducer;

import java.io.IOException;

import org.apache.hadoop.io.Text;import org.apache.hadoop.mapreduce.Reducer;

public class CommonFriendsStepTwoReducer extends Reducer<Text, Text, Text, Text>{

protected void reduce(Text user\_user, Iterable<Text> friends, Context context)

throws IOException, InterruptedException {

// 传进来的数据 <用户1-用户2，好友们>

// 新建 stringBuffer, 用于用户的共同好友们

StringBuffer stringBuffer = new StringBuffer();

// 遍历所有的好友，并将这些好友放在stringBuffer中，以" "分隔

for (Text friend : friends) {

stringBuffer.append(friend).append(" ");

}

// 以好友为key,用户们为value传给下一个mapper

context.write(user\_user, new Text(stringBuffer.toString()));

}

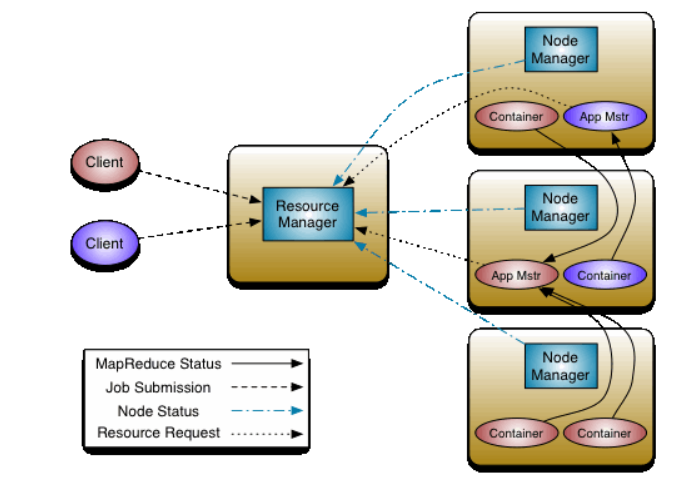
}

## 分布式资源调度框架

### yarn的概念

Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统和调度平台，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。可以把 yarn 理解为相当于一个分布式的操作系统平台，而 mapreduce 等运算程序则相当于运行于操作系统之上的应用程序，Yarn 为这些程序提供运算所需的资源（内存、cpu）。

### yarn的架构



YARN 是一个资源管理、任务调度的框架，主要包含三大模块：ResourceManager（RM）、

NodeManager（NM）、ApplicationMaster（AM）.

1).ResourceManager 负责所有资源的监控、分配和管理；

2).ApplicationMaster 负责每一个具体应用程序的调度和协调；

3).NodeManager 负责每一个节点的维护。对于所有的 applications，RM 拥有绝对的控制权和对资源的分配权。而每个 AM 则会和RM 协商资源，同时和 NodeManager 通信来执行和监控 task。

### yarn的工作流程

1）client 向 RM 提交应用程序，其中包括启动该应用的 ApplicationMaster 的必须信息，例如 ApplicationMaster 程序、启动 ApplicationMaster 的命令、用户程序等。

2）ResourceManager 启动一个 container 用于运行 ApplicationMaster。启动中的 ApplicationMaster 向 ResourceManager 注册自己，启动成功后与 RM 保持心

跳。

3）ApplicationMaster 向 ResourceManager 发送请求，申请相应数目的 container。

4)ResourceManager 返回 ApplicationMaster 的申请的 containers 信息。申请成功的

container，由 ApplicationMaster 进行初始化。container 的启动信息初始化后，AM

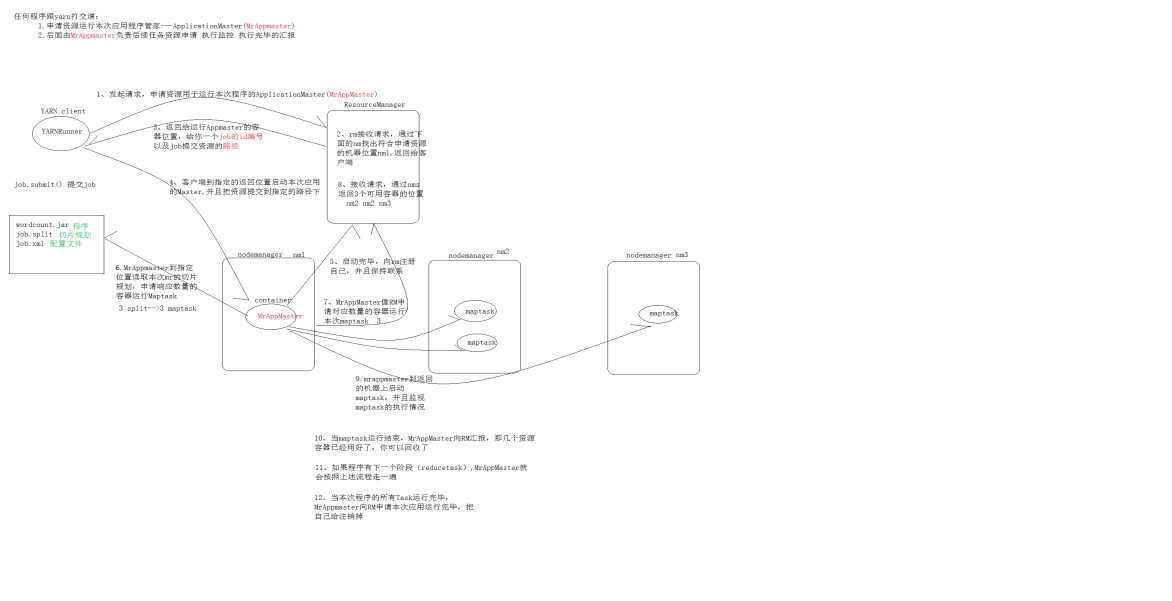
与对应的 NodeManager 通信，要求 NM 启动 container。AM 与 NM 保持心跳，从而对 NM上运行的任务进行监控和管理。

5） container 运行期间，ApplicationMaster 对 container 进行监控。container 通过 RPC

协议向对应的 AM 汇报自己的进度和状态等信息。

6) 应用运行期间，client 直接与 AM 通信获取应用的状态、进度更新等信息。

7) 应用运行结束后，ApplicationMaster 向 ResourceManager 注销自己，并允许属于它的container 被收回。



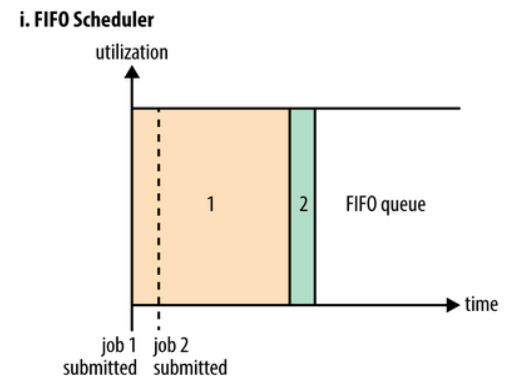
### yarn的调度器 Scheduler

Yarn中,负责给应用分配资源的就是 Scheduler,三种调度器可以选择:FIFO Scheduler ,Capacity Scheduler,FairScheduler。

##### 1. FIFO Scheduler

FIFO Scheduler 把应用按提交的顺序排成一个队列，这是一个 先进先出队列，在进行

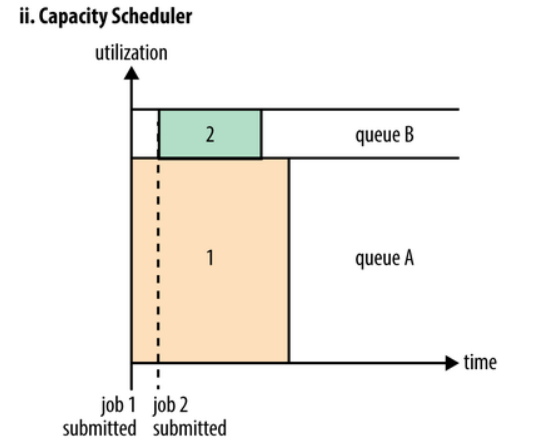
资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。



##### 2. Capacity Scheduler

Capacity 调度器允许多个组织共享整个集群，每个组织可以获得集群的一部分计算能

力。通过为每个组织分配专门的队列，然后再为每个队列分配一定的集群资源，这样整个集群就可以通过设置多个队列的方式给多个组织提供服务了。除此之外，队列内部又可以垂直划分，这样一个组织内部的多个成员就可以共享这个队列资源了，在一个队列内部，资源的调度是采用的是先进先出(FIFO)策略。



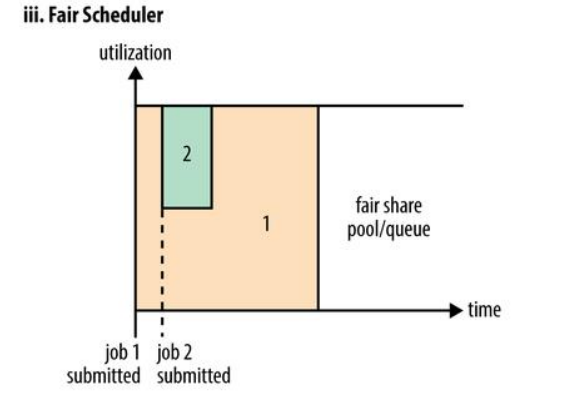
##### 3.Fair Scheduler

在 Fair 调度器中，我们不需要预先占用一定的系统资源，Fair 调度器会为所有运行的

job 动态的调整系统资源。如下图所示，当第一个大 job 提交时，只有这一个 job 在运行，

此时它获得了所有集群资源；当第二个小任务提交后，Fair 调度器会分配一半资源给这个

小任务，让这两个任务公平的共享集群资源。



## 分布式文件存储系统

### IMG_256 架构

　如上图所示，HDFS也是按照Master和Slave的结构。分NameNode、SecondaryNameNode、DataNode这几个角色。

**NameNode**：是Master节点，是大领导。管理数据块映射；处理客户端的读写请求；配置副本策略；管理HDFS的名称空间；

**SecondaryNameNode**：是一个小弟，分担大哥namenode的一部分工作量；是NameNode的冷备份；合并fsimage和fsedits然后再发给namenode。

**DataNode**：Slave节点，奴隶，干活的。负责存储client发来的数据块block；执行数据块的读写操作。

**热备份**：b是a的热备份，如果a坏掉。那么b马上运行代替a的工作。

**冷备份**：b是a的冷备份，如果a坏掉。那么b不能马上代替a工作。但是b上存储a的一些信息，减少a坏掉之后的损失。

**fsimage**:元数据镜像文件（文件系统的目录树。）

**edits**：元数据的操作日志（针对文件系统做的修改操作记录）

**namenode内存中存储的是=fsimage+edits。**

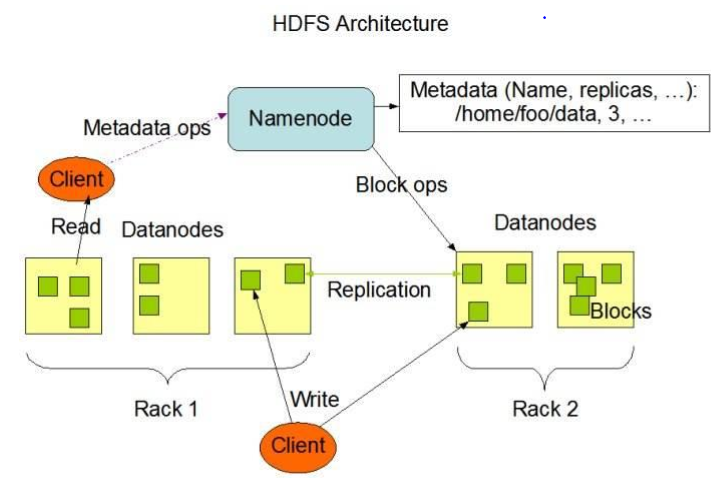
SecondaryNameNode负责定时默认1小时，从namenode上，获取fsimage和edits来进行合并，然后再发送给namenode。减少namenode的工作量。

### 原理

**1.工作机制**

NameNode 负责管理整个文件系统元数据；DataNode 负责管理具体文件数据块存储；Secondary NameNode 协助 NameNode 进行元数据的备份。

HDFS 的内部工作机制对客户端保持透明，客户端请求访问 HDFS 都是通过向NameNode 申请来进行。



**2.读写流程**

**\* HDFS 写数据流程**

1、 client 发起文件上传请求，通过 RPC 与 NameNode 建立通讯，NameNode

检查目标文件是否已存在，父目录是否存在，返回是否可以上传；

2、 client 请求第一个 block 该传输到哪些 DataNode 服务器上；

3、 NameNode 根据配置文件中指定的备份数量及机架感知原理进行文件分

配，返回可用的 DataNode 的地址如：A，B，C；

注： Hadoop 在设计时考虑到数据的安全与高效，数据文件默认在 HDFS 上存放

三份， 存储策略 为本地一份，同机架内其它某一节点上一份，不同机架的某一

节点上一份。

4、 client 请求 3 台 DataNode 中的一台 A 上传数据（本质上是一个 RPC 调

用，建立 pipeline）， A 收到请求会继续调用 B，然后 B 调用 C，将整个

pipeline 建立完成，后逐级返回 client；

5、 client 开始往 A 上传第一个 block（先从磁盘读取数据放到一个本地内

存缓存），以 packet 为单位（默认 64K），A 收到一个 packet 就会传给 B，

B 传给 C；A 每传一个 packet 会放入一个应答队列等待应答。

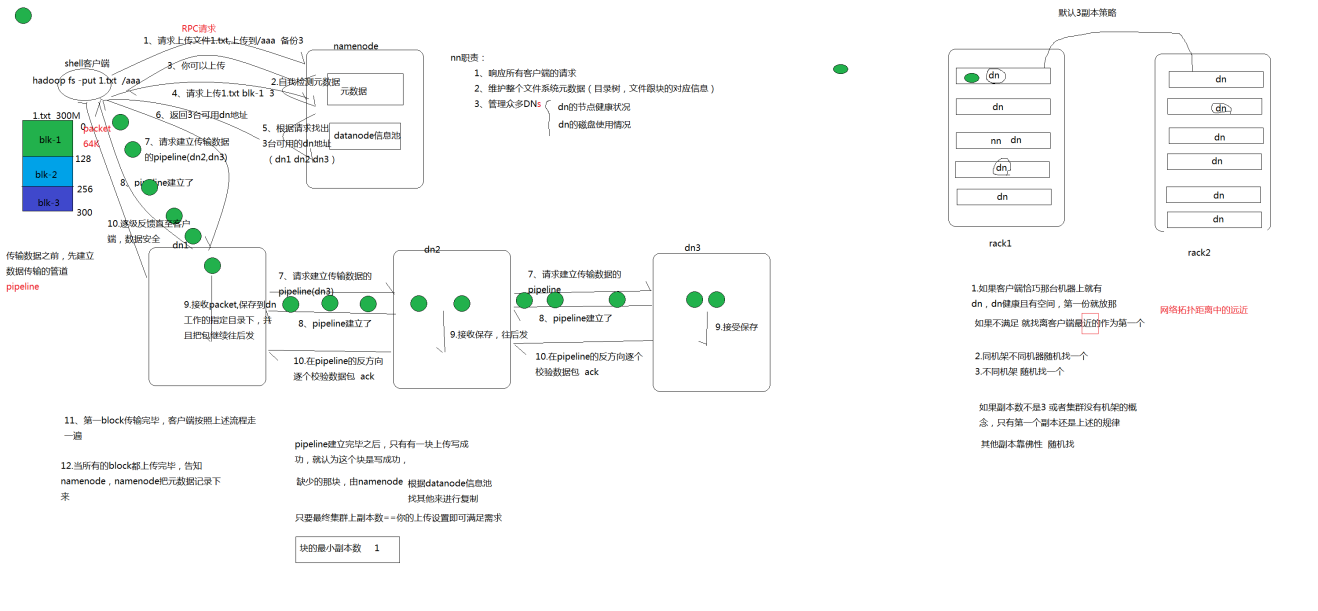
6、 数据被分割成一个个 packet 数据包在 pipeline 上依次传输，在

pipeline 反方向上，逐个发送 ack（命令正确应答），最终由 pipeline

中第一个 DataNode 节点 A 将 pipeline ack 发送给 client;

7、 当一个 block 传输完成之后，client 再次请求 NameNode 上传第二个

block 到服务器。



**\*HDFS 读数据流程**

1、 Client 向 NameNode 发起 RPC 请求，来确定请求文件 block 所在的位置；

2、 NameNode会视情况返回文件的部分或者全部block列表，对于每个block，

NameNode 都会返回含有该 block 副本的 DataNode 地址；

3、 这些返回的 DN 地址，会按照集群拓扑结构得出 DataNode 与客户端的距

离，然后进行排序，排序两个规则：网络拓扑结构中距离 Client 近的排

靠前；心跳机制中超时汇报的 DN 状态为 STALE，这样的排靠后；

4、 Client 选取排序靠前的 DataNode 来读取 block，如果客户端本身就是

DataNode,那么将从本地直接获取数据；

5、 底层上本质是建立 Socket Stream（FSDataInputStream），重复的调用

父类 DataInputStream 的 read 方法，直到这个块上的数据读取完毕；

6、 当读完列表的 block 后，若文件读取还没有结束，客户端会继续向

NameNode 获取下一批的 block 列表；

7、 读取完一个 block 都会进行 checksum 验证，如果读取 DataNode 时出现

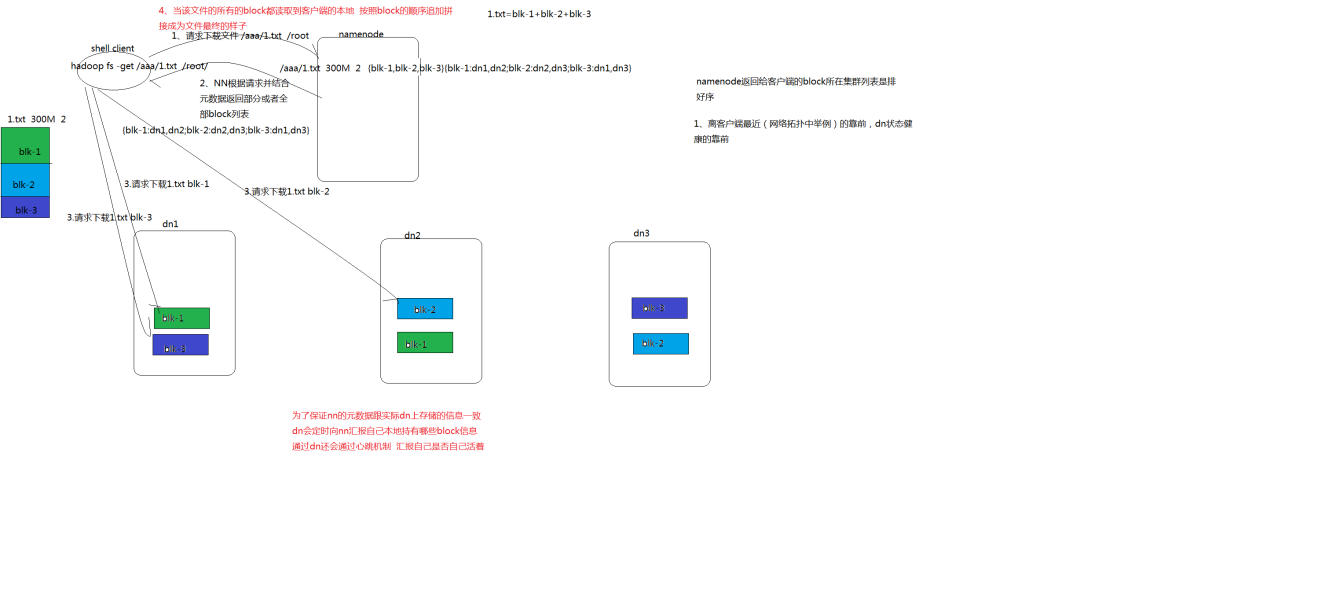
错误，客户端会通知 NameNode，然后再从下一个拥有该 block 副本的

DataNode 继续读。

8、 read 方法是并行的读取 block 信息，不是一块一块的读取；NameNode 只

是返回Client请求包含块的DataNode地址，并不是返回请求块的数据；

1. 最终读取来所有的 block 会合并成一个完整的最终文件。



### API

**1.shell 定时采集数据至 HDFS**

**\* 技术分析**

HDFS SHELL:

hadoop fs –put // 满足上传 文件，不能满足定时、周期性传入。

Linux crontab:

crontab -e

0 0 \* \* \* /shell/ uploadFile2Hdfs.sh //每天凌晨 12：00 执行一次

**\* 实现流程**

一般日志文件生成的逻辑由业务系统决定，比如每小时滚动一次，或者一定

大小滚动一次，避免单个日志文件过大不方便操作。

比如滚动后的文件命名为 access.log.x,其中 x 为数字。正在进行写的日志

文件叫做 access.log。这样的话，如果日志文件后缀是 1\2\3 等数字，则该文

件满足需求可以上传，就把该文件移动到准备上传的工作区间目录。工作区间有

文件之后，可以使用 hadoop put 命令将文件上传。

**\* 代码实现**



**2.JavaAPI操作**

**HDFS 的 JAVA API 操作**

HDFS 在生产应用中主要是客户端的开发，其核心步骤是从 HDFS 提供的 api

中构造一个 HDFS 的访问客户端对象，然后通过该客户端对象操作（增删改查）

HDFS 上的文件。

**1）搭建开发环境**

创建 Maven 工程，引入 pom 依赖

<dependencies>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-common</artifactId>

<version>2.7.4</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.7.4</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>2.7.4</version>

</dependency>

</dependencies>

配置 windows 平台 Hadoop 环境

在 windows 上做 HDFS 客户端应用开发，需要设置 Hadoop 环境,而且要求是windows 平台编译的 Hadoop,不然会报以下的错误:

Failed to locate the winutils binary in the hadoop binary path java.io.IOException: Could not

locate executable null\bin\winutils.exe in the Hadoop binaries.

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

为此我们需要进行如下的操作：

A、在 windows 平台下编译 Hadoop 源码（可以参考资料编译，但不推荐）

B、使用已经编译好的 Windows 版本 Hadoop：

hadoop-2.7.4-with-windows.tar.gz

C、解压一份到 windows 的任意一个目录下

D、在 windows 系统中配置 HADOOP\_HOME 指向你解压的安装包目录

E、在 windows 系统的 path 变量中加入 HADOOP\_HOME 的 bin 目录

**2）构造客户端对象**

在 java 中操作 HDFS，主要涉及以下 Class：

Configuration：该类的对象封转了客户端或者服务器的配置;

FileSystem：该类的对象是一个文件系统对象，可以用该对象的一些方法来

对文件进行操作，通过 FileSystem 的静态方法 get 获得该对象。

FileS ystem fs = FileSystem.get(conf)

get 方法从 conf 中的一个参数 fs.defaultFS 的配置值判断具体是什么类

型的文件系统。如果我们的代码中没有指定 fs.defaultFS，并且工程 classpath

下也没有给定相应的配置，conf中的默认值就来自于hadoop的jar包中的core

default.xml ， 默 认 值 为 ： file:/// ， 则 获 取 的 将 不 是 一 个

DistributedFileSystem 的实例，而是一个本地文件系统的客户端对象。

**3）示例代码**

Configuration conf = new Configuration();

//这里指定使用的是 hdfs 文件系统

conf.set("fs.defaultFS", "hdfs://node-21:9000");

//通过如下的方式进行客户端身份的设置

System.setProperty("HADOOP\_USER\_NAME", "root");

//通过 FileSystem 的静态方法获取文件系统客户端对象

FileSystem fs = FileSystem.get(conf);

//也可以通过如下的方式去指定文件系统的类型 并且同时设置用户身份

//FileSystem fs = FileSystem.get(new URI("hdfs://node-21:9000"), conf, "root");

//创建一个目录

fs.create(new Path("/hdfsbyjava-ha"), false);

//上传一个文件

fs.copyFromLocalFile(new Path("e:/hello.sh"), new Path("/hdfsbyjava-ha"));

//关闭我们的文件系统

fs.close();

**\*Hbase使用javaapi存储数据到HDFS**

public class HbaseDemo {

private Configuration conf = null;

@Before

public void init(){

conf = HBaseConfiguration.create();

conf.set("hbase.zookeeper.quorum", "itcast01:2181,itcast02:2181,itcast03:2181");

}

@Test

public void testDrop() throws Exception{

HBaseAdmin admin = new HBaseAdmin(conf);

admin.disableTable("account");

admin.deleteTable("account");

admin.close();

}

@Test

public void testPut() throws Exception{

HTable table = new HTable(conf, "person\_info");

Put p = new Put(Bytes.toBytes("person\_rk\_bj\_zhang\_000002"));

p.add("base\_info".getBytes(), "name".getBytes(), "zhangwuji".getBytes());

table.put(p);

table.close();

}

@Test

public void testDel() throws Exception{

HTable table = new HTable(conf, "user");

Delete del = new Delete(Bytes.toBytes("rk0001"));

del.deleteColumn(Bytes.toBytes("data"), Bytes.toBytes("pic"));

table.delete(del);

table.close();

}

@Test

public void testGet() throws Exception{

HTable table = new HTable(conf, "person\_info");

Get get = new Get(Bytes.toBytes("person\_rk\_bj\_zhang\_000001"));

get.setMaxVersions(5);

Result result = table.get(get);

List<Cell> cells = result.listCells();

for(Cell c:cells){

}

//result.getValue(family, qualifier); 可以从result中直接取出一个特定的value

//遍历出result中所有的键值对

List<KeyValue> kvs = result.list();

//kv ---> f1:title:superise.... f1:author:zhangsan f1:content:asdfasldgkjsldg

for(KeyValue kv : kvs){

String family = new String(kv.getFamily());

System.out.println(family);

String qualifier = new String(kv.getQualifier());

System.out.println(qualifier);

System.out.println(new String(kv.getValue()));

}

table.close();

}

**\*Storm使用javaapi存储数据HDFS**

第一步：导入整合的jar包

*com/artifact/org.apache.storm/storm-hdfs -->* <**dependency**>  
 <**groupId**>org.apache.storm</**groupId**>  
 <**artifactId**>storm-hdfs</**artifactId**>  
 <**version**>1.1.1</**version**>  
 <**exclusions**>  
 <**exclusion**>  
 <**groupId**>org.apache.hadoop</**groupId**>  
 <**artifactId**>hadoop-client</**artifactId**>  
 </**exclusion**>  
 <**exclusion**>  
 <**groupId**>org.apache.hadoop</**groupId**>  
 <**artifactId**>hadoop-auth</**artifactId**>  
 </**exclusion**>  
 <**exclusion**>  
 <**groupId**>org.apache.hadoop</**groupId**>  
 <**artifactId**>hadoop-common</**artifactId**>  
 </**exclusion**>  
 <**exclusion**>  
 <**groupId**>org.apache.hadoop</**groupId**>  
 <**artifactId**>hadoop-hdfs</**artifactId**>  
 </**exclusion**>  
 </**exclusions**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.apache.storm</**groupId**>  
 <**artifactId**>storm-jdbc</**artifactId**>  
 <**version**>1.1.1</**version**>  
 </**dependency**>  
 *<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->* <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.1.38</**version**>  
 </**dependency**>  
</**dependencies**>  
  
  
  
<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>3.1</**version**>  
 <**configuration**>  
 <**source**>1.8</**source**>  
 <**target**>1.8</**target**>  
 <**encoding**>utf-8</**encoding**>  
 </**configuration**>  
 </**plugin**>  
  
  
 *<!-- storm与hdfs的整合，请使用这种打包方式 -->* <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-shade-plugin</**artifactId**>  
 <**version**>1.4</**version**>  
 <**configuration**>  
 <**createDependencyReducedPom**>true</**createDependencyReducedPom**>  
 </**configuration**>  
 <**executions**>  
 <**execution**>  
 <**phase**>package</**phase**>  
 <**goals**>  
 <**goal**>shade</**goal**>  
 </**goals**>  
 <**configuration**>  
 <**transformers**>  
 <**transformer  
 implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"**/>  
 <**transformer  
 implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"**>  
 <**mainClass**>cn.itcast.hadoop.stormhadoop.MainTopology</**mainClass**>  
 </**transformer**>  
 </**transformers**>  
 </**configuration**>  
 </**execution**>  
 </**executions**>  
 </**plugin**>  
  
 *<!--  
 注意 storm与hdfs的整合已经不能使用这种打包方式，使用这种打包方式会出错 Error preparing HdfsBolt: No FileSystem for scheme: hdfs at org.apache.storm.hdfs.bolt.AbstractHdfs  
 storm与mysql的整合请使用这种打包方式  
 -->  
 <!-- <plugin>  
 <artifactId> maven-assembly-plugin </artifactId>  
 <configuration>  
 <descriptorRefs>  
 <descriptorRef>jar-with-dependencies</descriptorRef>  
 </descriptorRefs>  
 <archive>  
 <manifest>  
 <mainClass>cn.itcast.storm.demo2.JdbcTopo</mainClass>  
 </manifest>  
 </archive>  
 </configuration>  
 <executions>  
 <execution>  
 <id>make-assembly</id>  
 <phase>package</phase>  
 <goals>  
 <goal>single</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin> -->* </**plugins**>  
</**build**>

第二步：开发我们的RandomOrder

**public class** RandomOrderSpout **extends** BaseRichSpout{  
  
 **private** SpoutOutputCollector **collector**;  
 **private** PaymentInfo **paymentInfo**;  
  
  
 */\*\*  
 \* 初始化的方法只会被调用一次  
 \** ***@param conf*** *\** ***@param context*** *\** ***@param collector*** *\*/* @Override  
 **public void** open(Map conf, TopologyContext context, SpoutOutputCollector collector) {  
 **this**.**collector** = collector;  
 **paymentInfo** = **new** PaymentInfo();  
 }  
  
 */\*\*  
 \* 这个方法会反复的不断被调用  
 \*/* @Override  
 **public void** nextTuple() {  
 *//发送一个json格式的字符串出去  
 //collector.emit(new Values(paymentInfo.random()));  
 //注意我们这里发送数据多带了一个参数是我们的msgId,如果发送数据的时候，带上了msgid，就表明我们开启了storm的ack机制* String random = **paymentInfo**.random();  
 **collector**.emit(**new** Values(random), random);  
  
 }  
  
 @Override  
 **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
 declarer.declare(**new** Fields(**"randomOrder"**));  
 }  
  
  
 */\*\*  
 \* 消息发送失败的时候需要调用这个fail方法  
 \** ***@param msgId*** *\*/* @Override  
 **public void** fail(Object msgId) {  
 **collector**.emit(**new** Values(msgId),msgId);  
 }  
  
  
 */\*\*  
 \* 消息发送成功时候调用的方法  
 \** ***@param msgId*** *\*/* @Override  
 **public void** ack(Object msgId) {  
 *// super.ack(msgId);* System.***out***.println(**"我是消息发送成功了"**);  
 }  
}

第三步：实现我们的countMoneyBolt

**public class** CountMoneyBolt **extends** BaseRichBolt{  
*//OOM out of memory exception* **private** OutputCollector **collector**;  
 **private** JSONObject **jsonObject**;  
  
 **private static** ConcurrentHashMap<String,Long> *concurrentHashMap* = **new** ConcurrentHashMap<String,Long>();  
  
  
 */\*\*  
 \* 初始化的方法，只会被调用一次  
 \** ***@param stormConf*** *\** ***@param context*** *\** ***@param collector*** *\*/* @Override  
 **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
 **this**.**collector** = collector;  
 **jsonObject** = **new** JSONObject();  
  
 }  
  
 */\*\*  
 \* 反复不断的被调用  
 \** ***@param input*** *\*/* @Override  
 **public void** execute(Tuple input) {  
 *//获取上游发送的数据过来* String randomOrder = input.getStringByField(**"randomOrder"**);  
 PaymentInfo paymentInfo = **jsonObject**.*parseObject*(randomOrder, PaymentInfo.**class**);  
 **long** payPrice = paymentInfo.getPayPrice();  
 **if**(*concurrentHashMap*.containsKey(**"totalMoney"**)){  
 *concurrentHashMap*.put(**"totalMoney"**,*concurrentHashMap*.get(**"totalMoney"**)+payPrice);  
 }**else**{  
 *concurrentHashMap*.put(**"totalMoney"**,payPrice);  
 }  
 System.***out***.println(**"当前的总金额是"**+*concurrentHashMap*.toString());  
 **collector**.emit(**new** Values(randomOrder));  
 *//如果继承的是BaseRichBolt，一定要记得最后调用一下ack方法，绝对不会错，如果不调，有可能会错误* **collector**.ack(input);  
 }  
  
 @Override  
 **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
 declarer.declare(**new** Fields(**"moneyOrder"**));  
 }  
}

第四步：主代码实现

**public class** OrderMain {  
  
 **public static void** main(String[] args) **throws** InvalidTopologyException, AuthorizationException, AlreadyAliveException {  
 *// use "|" instead of "," for field delimiter* RecordFormat format = **new** DelimitedRecordFormat()  
 .withFieldDelimiter(**"|"**);  
  
*// sync the filesystem after every 1k tuples 数据同步到hdfs的策略，每执行一千条数据的时候，同步一次到hdfs上面去* SyncPolicy syncPolicy = **new** CountSyncPolicy(1000);  
  
*// rotate files when they reach 5MB 每当文件大小达到5kb的时候，同步到HDFS上面去* FileRotationPolicy rotationPolicy = **new** FileSizeRotationPolicy(5.0f, FileSizeRotationPolicy.Units.***KB***);  
  
  
 *//指定我们的数据上传到hdfs的哪个路劲下面去* FileNameFormat fileNameFormat = **new** DefaultFileNameFormat()  
 .withPath(**"/test/"**);  
*//FileSystem* HdfsBolt bolt = **new** HdfsBolt()  
 .withFsUrl(**"hdfs://node01:9000"**)  
 .withFileNameFormat(fileNameFormat)  
 .withRecordFormat(format)  
 .withRotationPolicy(rotationPolicy)  
 .withSyncPolicy(syncPolicy);  
  
  
 TopologyBuilder builder = **new** TopologyBuilder();  
 builder.setSpout(**"randomOrderSpout"**,**new** RandomOrderSpout());  
 builder.setBolt(**"countMoneyBolt"**,**new** CountMoneyBolt()).localOrShuffleGrouping(**"randomOrderSpout"**);  
 builder.setBolt(**"hdfsBolt"**,bolt).localOrShuffleGrouping(**"countMoneyBolt"**);  
  
 Config config = **new** Config();  
 *//设置我们ack的线程的数量，默认是一个，如果数据量比较大，需要将ack的线程数调大，用于调优* config.setNumAckers(3);  
 *//设置我们的内存池里面有多少个状态没有清掉的时候就不要再继续发送数据了* config.setMaxSpoutPending(5000);  
  
 **if**(args !=**null** && args.**length** > 0){  
 config.setDebug(**false**);  
 StormSubmitter.*submitTopology*(args[0],config,builder.createTopology());  
 }**else**{  
 config.setDebug(**true**);  
 LocalCluster cluster = **new** LocalCluster();  
 cluster.submitTopology(**"stormtoHdfs"**,config,builder.createTopology());  
 }  
 }  
}

**\*Flume使用javaapi存储数据到HDFS**

**1，配置flume代理文件**

配置一个flume agent代理,在此名称为shaman。配置文件（netcat-memory-hdfs.conf）如下：

# Identify the components on agent shaman:

shaman.sources = netcat\_s1

shaman.sinks = hdfs\_w1

shaman.channels = **in**-mem\_c1# Configure the source:

shaman.sources.netcat\_s1.type = netcat

shaman.sources.netcat\_s1.bind = localhost

shaman.sources.netcat\_s1.port = 44444# Describe the sink:

shaman.sinks.hdfs\_w1.type = hdfs

shaman.sinks.hdfs\_w1.hdfs.path = hdfs://localhost:8020/user/root/test

shaman.sinks.hdfs\_w1.hdfs.writeFormat = Text

shaman.sinks.hdfs\_w1.hdfs.fileType = DataStream

# Configure a channel that buffers events in memory:

shaman.channels.**in**-mem\_c1.type = memory

shaman.channels.**in**-mem\_c1.capacity = 20000

shaman.channels.**in**-mem\_c1.transactionCapacity = 100# Bind the source and sink to the channel:

shaman.sources.netcat\_s1.channels = **in**-mem\_c1

shaman.sinks.hdfs\_w1.channel = **in**-mem\_c1

备注：  
hdfs://localhost:8020/user/root/test，其中hdfs://localhost:8020为hadoop配置文件core-site.xml中  
fs.defaultFS属性的值，root为hadoop的登陆用户。

**2，启动flume代理**

bin/flume-ng agent -f agent/netcat-memory-hdfs.conf -n shaman -Dflume.root.logger=DEBUG,console -Dorg.apache.flume.log.printconfig=true -Dorg.apache.flume.log.rawdata=true

**3，打开telnet客户端，输入字母测试**

telnet localhost 44444

然后输入文字

**4，查看hdfs test目录**

hdfs dfs -**ls** /user/root/**test**

会发现有新的文件出现，文件里面的内容即是通过telent输入的字母

## 项目

### 点击流日志模型

#### 1.分析什么

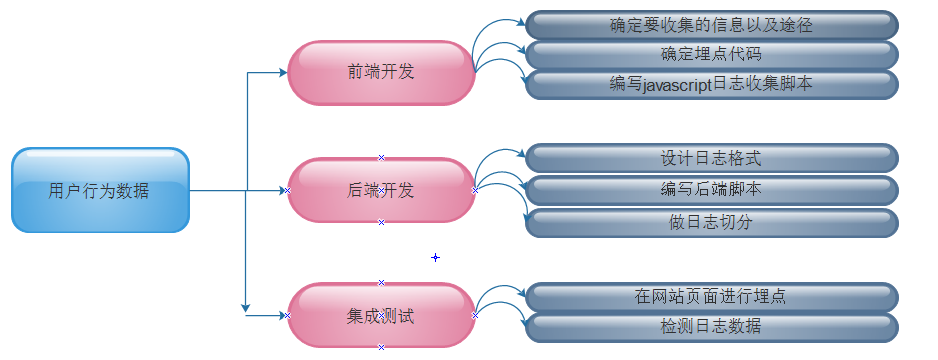
点击流日志模型主要是分析用户在网站上持续访问的轨迹，查看网站的访问量，分析关键路径得到漏斗模型，以此来优化网站，提高用户的网站停留时间。

#### 2.确定数据源

在本次案例中，使用埋点的方式收集数据，埋点是指在网页中插入小段javascript代码，这个代码片段会动态创建一个script标签，并将src属性指向一个单独的js文件，此时这个单独的js文件会被浏览器请求到并执行，这个js往往就是真正的数据收集脚本。

数据收集完成后，js会请求一个后端的数据收集脚本，这个脚本一般是一个伪装成图片的动态脚本程序，js会将收集的数据通过http参数的方式传递给后端脚本，后端脚本解析参数并按固定格式记录到访问日志，同时可能会在http响应中给客户端种植一些用于追踪的cookie。

设计实现



##### 确定收集信息

##### C:\Users\sqq\AppData\Local\Temp\1527313067(1).png

###### 1）确定埋点代码

<script type=”text/javascript”>

val \_map=\_map || [];

\_map.push([‘\_setAccount’],’ UA-XXXXX-X’);

//自调用匿名函数只会在运行时执行一次，一般用于初始化

(function() {

//引入一个外部的js文件ma

var ma = document.createElement('script');

ma.type ='text/javascript';

//异步调用外部js文件

ma.async = true;

//根据协议将src指向对应的ma.js

ma.src = ('https:' == document.location.protocol ?

'https://ssl' : 'http://www') + '.google-analytics.com/ma.js';

//将这个元素插入到dom树上

var s = document.getElementsByTagName('script')[0];

s.parentNode.insertBefore( ma, s);

})();

</script>

###### 2）前端收集脚本

在第二步配置中的ma.js被请求后会被执行，一般要做如下几件事：

1. 通过浏览器内置js对象收集信息
2. 解析\_map数组，收集配置信息，这里面可能会包括用户自定义的事件跟踪，业务数据等
3. 将上面两步收集的数据按预定义格式解析并拼接
4. 请求一个后端脚本，将信息放在http request参数中携带给后端脚本

示例代码

(function(){

var params={};

//document对象数据，实现了A步骤

if(document){

params.domain=document.domain||’’;

params.url=document.URL||’’;

params.title=document.title||’’;

params.referrer=document.referrer||’’;

}

//window对象数据

if(window&&window.screen){

params.sh=window.screen.height||0;

params.sw=window.screen.width||0;

params.cd=window.screen.colorDepth||0;

}

//nevigator对象数据

if(nevigator){

params.lang=navigator.language||’’;

}

//接卸map数组。收集配置信息

if(\_maq){

for(var i in\_maq){

switch(\_map[i][0]){

case ‘\_setAccount’:

params.account=\_maq[i][1];

break;

default:

break;

}

}

}

//拼接参数串

var args=’’;

for(var i in params){

if(agrs!=’’)args+=’&’;

args += i + '=' + encodeURIComponent(params[i]);

}

//通过image对象请求后端脚本

var img=new Image(1,1);

//请求后端脚本常用的是ajax，但是ajax是不能跨域请求的，通用的方

//法是js脚本创建一个image对象，将image对象的src属性指向后端脚//本并携带参数,就实现了跨域请求。

img.src=’http://xx.xxx.xxx/log.gif?’+args;

})();

###### 3)后端脚本

log.gif是后端脚本，是一个伪装成gif图片的脚本，后端脚本一般需要完成以下几件事情

1. 解析http请求参数得到信息
2. 从web服务器获取一些客户端无法获取的信息
3. 将信息按格式写入log
4. 生成一副1\*1的gif图片作为响应内容并将响应头的content-type设置为image/gif
5. 在响应头中通过set-cookie设置一些需要的cookie信息

我们使用nginx的access\_log做日志收集

首先，需要在 nginx 的配置文件中定义日志格式：

log\_format tick

"$msec||$remote\_addr||$status||$body\_bytes\_sent||$u\_domain||$u\_url|

|$u\_title||$u\_referrer||$u\_sh||$u\_sw||$u\_cd||$u\_lang||$http\_user\_ag

ent||$u\_account";

注意这里以 u\_开头的是我们待会会自己定义的变量，其它的是 nginx 内置变

量。然后是核心的两个 location：

location / log.gif {

#伪装成 gif 文件

default\_type image/gif;

#本身关闭 access\_log，通过 subrequest 记录 log

access\_log off;

access\_by\_lua "

-- 用户跟踪 cookie 名为\_\_utrace

local uid = ngx.var.cookie\_\_\_utrace

if not uid then

-- 如果没有则生成一个跟踪 cookie，算法为

md5(时间戳+IP+客户端信息)

uid = ngx.md5(ngx.now() ..

ngx.var.remote\_addr .. ngx.var.http\_user\_agent)

end

ngx.header['Set-Cookie'] = {'\_\_utrace=' .. uid ..

'; path=/'}

if ngx.var.arg\_domain then

-- 通过 subrequest 子请求 到/i-log 记录日志，

将参数和用户跟踪 cookie 带过去

ngx.location.capture('/i-log?' ..

ngx.var.args .. '&utrace=' .. uid)

end

";

#此请求资源本地不缓存

add\_header Expires "Fri, 01 Jan 1980 00:00:00 GMT";

add\_header Pragma "no-cache";

add\_header Cache-Control "no-cache, max-age=0, must-

revalidate";

#返回一个 1×1 的空 gif 图片

empty\_gif;

}

location /i-log {

#内部 location，不允许外部直接访问

internal;

#设置变量，注意需要 unescape，来自 ngx\_set\_misc 模块

set\_unescape\_uri $u\_domain $arg\_domain;

set\_unescape\_uri $u\_url $arg\_url;

set\_unescape\_uri $u\_title $arg\_title;

set\_unescape\_uri $u\_referrer $arg\_referrer;

set\_unescape\_uri $u\_sh $arg\_sh;

set\_unescape\_uri $u\_sw $arg\_sw;

set\_unescape\_uri $u\_cd $arg\_cd;

set\_unescape\_uri $u\_lang $arg\_lang;

set\_unescape\_uri $u\_account $arg\_account;

#打开日志

log\_subrequest on;

#记录日志到 ma.log 格式为 tick

access\_log /path/to/logs/directory/ma.log tick;

#输出空字符串

echo '';

}

###### 4）日志切分

日志收集系统访问日志时间一长文件变得很大，而且日志放在一个文件不便

于管理。通常要按时间段将日志切分，例如每天或每小时切分一个日志。通过

crontab 定时调用一个 shell 脚本实现，如下：

\_prefix="/path/to/nginx"

time=`date +%Y%m%d%H`

mv ${\_prefix}/logs/ma.log ${\_prefix}/logs/ma/ma-${time}.log

kill -USR1 `cat ${\_prefix}/logs/nginx.pid `

这个脚本将 ma.log 移动到指定文件夹并重命名为 ma-{yyyymmddhh}.log，

然后向 nginx 发送 USR1 信号令其重新打开日志文件。

USR1 通常被用来告知应用程序重载配置文件, 向服务器发送一个 USR1 信号

将导致以下步骤的发生：停止接受新的连接，等待当前连接停止，重新载入配置

文件，重新打开日志文件，重启服务器，从而实现相对平滑的不关机的更改。

cat ${\_prefix}/logs/nginx.pid 取 nginx 的进程号

然后再/etc/crontab 里加入一行：

59 \* \* \* \* root /path/to/directory/rotatelog.sh

在每个小时的 59 分启动这个脚本进行日志轮转操作。

#### 3.确定采集方案

在本次案例中，使用Flume日志采集系统来采集数据到HDFS系统。在服务器上部署agent节点，修改配置文件

配置文件如下

a1.sources=r1

a1.sinks=k1

a1.channels=c1

//监视指定的一些文件，将近实时的tail这些文件，获取这些文件的新追加的行

a1.sources.r1.type= TAILDIR

a1.sources.r1.channels=c1

//配置检查点文件的路径，检查点文件会以json格式保存已经tail文件的位置，解决了断点不能续传的缺陷

a1.sources.r1.positionFile=/export/log/flume/taildir\_position.json

//可以指定要tail的组

a1.sources.r1.filegroups=f1 f2

//指定每个文件的全路径

a1.sources.r1.filegroups.f1=/export/log/test1/example.log

a1.sources.r1.filegroups.f2=/export/log/test2/.\*log.\*

//配置channels

a1.channels.c1.type=memory

a1.channels.c1.capacity=1000

a1.channels.c1.transactionCapacity=100

//配置下沉点，将文件写到hdfs中去

a1.sinks.k1.type=hdfs

a1.sinks.k1.channel=c1

a1.sinks.k1.hdfs.path=/flume/events/%y-%m-%d/

a1.sinks.k1.hdfs.filePrefix=events-

a1.sinks.k1.hdfs.round=true

a1.sinks.k1.hdfs.roundValue=10

a1.sinks.k1.hdfs.roundUnit=minute

#### 4. 确定三层架构

ETL工作的实质就是从各个数据源提取数据，对数据进行转换，并最终加载填充数据到数据仓库纬度建模后的表中。

##### 1)创建ODS层数据表

1. 创建原始数据表

drop table if exists ods\_weblog\_origin;

create table ods\_weblog\_origin(

valid string,

remote\_addr string,

remote\_user string,

time\_local string,

request string,

status string,

body\_bytes\_sent string,

http\_referer string,

http\_user\_agent string)

partitioned by (datestr string)

row format delimited

fields terminated by '\001';

1. 点击流模型pageviews表

drop table if exits ods\_weblog\_origin;

create table ods\_weblog\_origin(

valid string,

remote\_addr string,

remote\_user string,

time\_local string,

request string,

visit\_step string,

page\_staylong string,

http\_referer string,

http\_user\_agent string,

body\_bytes\_sent string,

status string)

partitioned by(datestr string)

row format delimited fields terminated by ‘\001’;

1. 点击流visit模型

drop table if exits ods\_click\_stream\_visit;

create table ods\_click\_stream\_click(

session string,

remote\_addr string,

inTime string,

outTime string,

inPage string,

outPage string,

referal string,

pageVisits int)

partitioned by (datastr string)

row format delimited fields terminated by ‘\001’;

##### 2)导入ODS层数据

load data inpath ‘/weblog/preprocessed/’ overwrite into table ods\_weblog\_origin partition(datastr=’20180526’);--数据导入

show partitions ods\_weblog\_origin;--查看分区

##### 3)生成ODS层明细宽表

#### 5.模块开发---统计分析

##### 1)流量分析

1. 多维度统计PV总量

按时间维度

1. 计算每小时的pv数

select count(\*) as pvs ,month,day,hour from ods\_weblog\_detail group by month,day,hour;

1. 计算该处理批次一天中的各个小时pvs

drop table dw\_pvs\_everyhour\_oneday;

create table dw\_pvs\_everyhour\_oneday(month string,day string,hour string ,pvs bigint) partitioned by(datestr);

insert into table dw\_pvs\_everyhour\_oneday partition(datestr=”20180526”) select a.month as month,a.day as a.day,a.hour as hour,a.count(\*) as pvs from ods\_weblog\_detail where a.datastr=’20180526’ group by a.month,a.day,a.hour;

1. 计算每天的pvs

drop table dw\_pvs\_everyday;

create table dw\_pvs\_everyday(pvs bigint,month string,day string);

insert into table dw\_pvs\_everyday select count(\*) as pvs,a.month as month,a.day as day from ods\_weblog\_detail a group by a.month,a.day;

1. 人均浏览量

需求：统计今日所有来访者平均请求的页面数

计算方式：总页面请求数/去重总人数

remote\_addr表示不同的用户，可以先统计出不同的remote\_addr的pv量，然后累加所有pv作为总的页面请求数，再count所有remote\_addr作为总的去重总人数。

--总页面请求数/去重总人数

drop table dw\_avgpv\_user\_everyday;

create table dw\_avgpv\_user\_everyday(

day string,

avgpv string

);

Insert into table dw\_avgpv\_user\_everyday

select ‘20180526’,sum(b.pvs)/count(b.remote\_addr)from (select remote\_addr, count(1) as pvs from ods\_weblog\_detail where datestr=’20180526’ group by remote\_addr)b;

1. 统计pv总量最大的来源

统计每个小时各来访host的产生pv数最多的前N个（topN）

drop table dw\_pvs\_refhost\_topn\_everyhour;

create table dw\_pvs\_refhost\_topn\_everyhour(

hour string,toporder string,ref\_host string,ref\_host\_cnts string)

partitioned by (datestr string);

insert into dw\_pvs\_refhost\_topn\_everyhour

select ref\_host,ref\_host\_cnts,concat(month,day,hour),row\_number() over(partition by concat(month,day.hour) order by ref\_host\_cnts desc)as od from dw\_pvs\_refererhost\_everyhour where od<=3;

##### 2)关键路径转化率分析（漏斗模型）

A、需求分析

在一条指定的业务流程中，各个步骤的完成人数及相对上一个步骤的百分比

B、模型设计

定义好业务流程中的页面标识，下例中的步骤为：

C、模型设计

查询每一个步骤的总访问人数

Create table dw\_oute\_numbs as

select ‘step1’ as step ,count(distinct remote\_addr) as numbs from ods\_clik\_pageviews where datestr=’20180526’ and request like ‘/item%’

union

select ‘step2’ as step,count(distinct remote\_addr) as numbs from ods\_clik\_pageviews where datestr=’20180526’ and request like ‘/category%’

union

select ‘step3’ as step,count(distinct remote\_addr) as numbs from ods\_clik\_pageviews where datestr=’20180526’ and request like ‘/order%’

union

select ‘step4’ as step,count(distinct remote\_addr) as numbs from ods\_clik\_pageviews where datestr=’20180526’ and request like ‘/index%’

查询每一步相对于路径起点人数的比例

select tmp.rnstep,tmp.rnnumbs/tmp.rrnumbs as ratio from (select rn.step as rnstep,rn.rumbs as rnnumbs,rr.step as rrstep,rr.numbs as rrnumbs from dw\_oute\_numbs rn inner join dw\_out\_numbs rr)tmp where tmp.rrstep=’step1’;

查询每一步骤相对于上一步骤的漏出率

Select tmp.rnstep,tmp.rnnumbs/tmp.rrnumbs from (select rn.step as rnstep,rn,rumbs as rnnumbs,rr.step as rrstep,rr.numbs as rrnumbs from dw\_oute\_numbs rn inner join dw\_out\_numbs rr )where cast(substr(tmp.rnstep,5,1),int)=cast(substr(tmp.rrstep,5,1),int)-1;

#### 6 模块开发—结果导出

采用apache sqoop将数据导出

#### 7 模块开发—数据可视化

采用Echarts对数据进行可视化展示。

#### 相关面试题

1. 电商总体运营指标
2. 网站流量指标

### 推荐系统项目

推荐系统：本质上是商品售卖系统，和电商网站的目的一样，用来售卖商品。

会收集用户的所有行为信息（网站浏览信息、订单信息、关注信息、收藏商品、评论系统、外部信息（微博信息、联盟网站）），通过分析分析用户的历史行为给用户的兴趣建模，从而主动给用户推荐能够满足他们兴趣和需求的信息。

**数据源：**

基础数据的来源有很多维度，包括用户的访问、浏览、下单、收藏、用户的历史订单数据，评价信息等很多数据。

基础数据主要包括：

1. 要推荐物品或内容的元数据。例如关键字、属性描述等
2. 系统用户的基本信息，例如性别，年龄等
3. 用户对物品或者信息的偏好，包括用户对物品的评分，用户查看物品的记录，用户购买记录等

可以将这些用户的偏好信息分为两类：

**显示的用户反馈**：这类是用户在网站上自然浏览或者使用网站以外，显式的提供反馈信息，例如用户对物品的评分，或者对物品的评论。

**隐式的用户信息：**用户在使用网站是产生的数据，隐式的反应了用户对物品的喜好，例如用户查看了某物品的信息等等用户反馈。

**数据的采集方案：**

数据收集模块**：**

点击流模块：flume+kafka+storm+redis处理用户当前浏览的信息，将信息计算出来，保存到redis中（比如用户对一个品类的偏好）

订单支付（AMQ）：开发一个消费者程序（storm、JavaAPP），用来计算用户的偏好

外部数据：通过爬虫技术爬取用户的社交网站数据。

合作数据：比如京东和腾讯合作，可以获取一些用户在腾讯上的信息。

1. 将用户产生的数据（浏览商品、关注商品、收藏商品、加入购物车、下订单、评论等）保存到消息队列（kafka）中，将不同类型的数据保存到不同的topic中，以作分类。
2. 另一种方式就是将用户产生的数据保存到分库分表的数据库中

**数据的存储与计算：**

1. **离线推荐**

**离线推荐计算数据的周期可以是一个月或者15天**

1. 由FTP服务对推送的数据进行校验，将数据库中校验通过的数据定时的上传到HDFS文件系统中。
2. 基于HDFS建立hive数据仓库，将HDFS中的数据映射成一张张的表（用户表、收藏表、购物车表、订单表等）保存到hive数据仓库中。
3. 将hive数据仓库中的数据通过HSQL计算和导入到算法中（协同过滤算法）进行计算，得到一些用户的偏好数据。
4. 将计算出来的用户偏好数据定时导入到hbase/redis中。
5. 用户通过点击浏览商品，在推荐引擎（Javaweb）中获取用户id，根据用户的id从redis或hbase中拿去已经计算好的推荐结果（离线计算）推荐给用户。在此过程中，还要对结果数据进行过滤，比如商品上下线状态的判定、推荐结果种类丰富性的判定、推荐结果数量的补足等。最后将计算过滤之后的商品展示给用户。
6. **实时推荐**
7. 通过storm，实时的消费消息队列kafka的topic中的数据，实时的计算一些用户的偏好，同样将计算好的偏好数据实时的导出到hbase或redis中，等待用户点击浏览商品，实时的将计算好的结果展示给用户。

**大数据处理平台：**

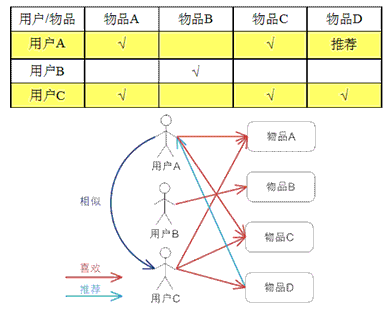
离线计算和实时计算

**数据的计算：**

协同过滤算法：也就是计算相似度，关于相似度的计算，就是计算像个向量的距离，距离越近相似度越大。

1. 基于用户的协同过滤算法

原理：基于用户对物品的偏好找到相邻邻居用户，然后将邻居用户喜欢的推荐给当前用户。



假如：用户A喜欢物品A、物品C

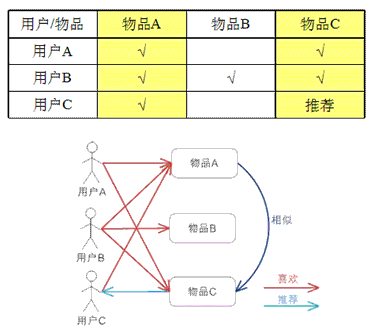
用户B喜欢物品B

用户C喜欢物品A、物品C、物品D

通过计算发现用户A和用户C相似，所以将用户C喜欢的物品D推荐给用户A。

1. 基于物品的协同过滤算法

原理：与基于用户的类似，只是在计算邻居时采用物品本身，而不是从用户角度。基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。



如图：根据用户历史数据的偏好计算出喜欢物品A的用户大部分都喜欢物品C，计算出物品A和物品C比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C，所以将物品C推荐给用户C。

**数据展现：**

Web网站

面试题：

1. 推荐系统的本质是什么
2. 推荐系统的指标是什么
3. 推荐系统的y和x
4. 推荐系统的样本构造和数据拼接
5. 推荐系统的场景思考
6. 推荐系统相关组件

## 相关面试题

1、hadoop运行原理

2、mapreduce原理

3、mapreduce的优化

4、举一个简单的例子说下mapreduce是怎么运行的

5、hadoop中combiner的作用

6、简述hadoop的安装

7、请列出hadoop的进程名

8、简述hadoop的调度器

9、列出你开发mapreduce的语言

10、我们开发job时是否可以去掉reduce阶段

11、datanode在什么情况下不会备份

12、combiner出现在哪个过程

13、hdfs的体系结构

14、3个datanode中有一个datanode出现错误会怎么样

15、描述一下hadoop中，有哪些地方用了缓存机制，作用分别是什么？

16、如何确定hadoop集群的健康状况

17、shuffe阶段，你怎么理解

18、mapreduce的map数量和reduce数量怎么确定，怎么配置

19、简单说一下mapreduce的编程模型

20、hadoop的TextInputFormatter作用是什么，如何自定义实现

21、hadoop和spark都是并行计算，他们有什么相同和区别

22、为什么要用flume导入hdfs，hdfs的架构是怎样的

23、简单说一下hadoop和spark的shuffle过程

24、hadoop高并发

25、map-reduce程序运行的时候会有什么比较常见的问题