高频面试题库

目录

[第1章 项目涉及技术 12](#_Toc34254624)

[1.1 Linux&Shell相关总结 12](#_Toc34254625)

[1.1.1 Linux常用命令 12](#_Toc34254626)

[1.1.2 Shell常用工具 12](#_Toc34254627)

[1.2 Hadoop相关总结 12](#_Toc34254628)

[1.2.1 Hadoop常用端口号 12](#_Toc34254629)

[1.2.2 Hadoop配置文件以及简单的Hadoop集群搭建 12](#_Toc34254630)

[1.2.3 HDFS读流程和写流程 13](#_Toc34254631)

[1.2.4 MapReduce的Shuffle过程及Hadoop优化（包括：压缩、小文件、集群优化） 14](#_Toc34254632)

[1.2.5 Yarn的Job提交流程 17](#_Toc34254633)

[1.2.6 Yarn的默认调度器、调度器分类、以及他们之间的区别 17](#_Toc34254634)

[1.2.7 项目经验之LZO压缩 18](#_Toc34254635)

[1.2.8 Hadoop参数调优 19](#_Toc34254636)

[1.2.9 项目经验之基准测试 19](#_Toc34254637)

[1.2.10 Hadoop宕机 19](#_Toc34254638)

[1.2.11 Hadoop解决数据倾斜方法 19](#_Toc34254639)

[1.3 Zookeeper相关总结 20](#_Toc34254640)

[1.3.1 选举机制 20](#_Toc34254641)

[1.3.2 常用命令 20](#_Toc34254642)

[1.4 Flume相关总结 21](#_Toc34254643)

[1.4.1 Flume组成，Put事务，Take事务 21](#_Toc34254644)

[1.4.2 Flume拦截器 21](#_Toc34254645)

[1.4.3 Flume Channel选择器 22](#_Toc34254646)

[1.4.4 Flume监控器 22](#_Toc34254647)

[1.4.5 Flume采集数据会丢失吗?（防止数据丢失的机制） 22](#_Toc34254648)

[1.4.6 Flume内存 22](#_Toc34254649)

[1.4.7 FileChannel优化 22](#_Toc34254650)

[1.4.8 HDFS Sink小文件处理 23](#_Toc34254651)

[1.4.9 HDFS Sink小文件处理 23](#_Toc34254652)

[1.5 Kafka相关总结 24](#_Toc34254653)

[1.5.1 Kafka架构 24](#_Toc34254654)

[1.5.2 Kafka压测 24](#_Toc34254655)

[1.5.3 Kafka的机器数量 24](#_Toc34254656)

[1.5.4 Kafka的日志保存时间 24](#_Toc34254657)

[1.5.5 Kafka的硬盘大小 24](#_Toc34254658)

[1.5.6 Kafka监控 24](#_Toc34254659)

[1.5.7 Kakfa分区数 25](#_Toc34254660)

[1.5.8 副本数设定 25](#_Toc34254661)

[1.5.9 多少个Topic 25](#_Toc34254662)

[1.5.10 Kafka丢不丢数据 25](#_Toc34254663)

[1.5.11 Kafka的ISR副本同步队列 25](#_Toc34254664)

[1.5.12 Kafka分区分配策略 25](#_Toc34254665)

[1.5.13 Kafka中数据量计算 26](#_Toc34254666)

[1.5.14 Kafka挂掉 26](#_Toc34254667)

[1.5.15 Kafka消息数据积压，Kafka消费能力不足怎么处理？ 26](#_Toc34254668)

[1.5.16 Kafka幂等性 26](#_Toc34254669)

[1.5.17 Kafka事务 27](#_Toc34254670)

[1.5.18 Kafka数据重复 27](#_Toc34254671)

[1.5.19 Kafka参数优化 27](#_Toc34254672)

[1.5.20 Kafka 高效读写数据 28](#_Toc34254673)

[1.6 Hive总结 30](#_Toc34254674)

[1.6.1 Hive的架构 30](#_Toc34254675)

[1.6.2 Hive和数据库比较 30](#_Toc34254676)

[1.6.3 内部表和外部表 31](#_Toc34254677)

[1.6.4 4个By区别 31](#_Toc34254678)

[1.6.5 窗口函数 31](#_Toc34254679)

[1.6.6 自定义UDF、UDTF 32](#_Toc34254680)

[1.6.7 Hive优化 32](#_Toc34254681)

[1.6.8 Hive解决数据倾斜方法 34](#_Toc34254682)

[1.6.9 用的是动态分区吗？动态分区的底层原理是什么？ 37](#_Toc34254683)

[26. Hive里边字段的分隔符用的什么？为什么用\t？有遇到过字段里边有\t的情况吗，怎么处理的？为什么不用Hive默认的分隔符，默认的分隔符是什么？ 37](#_Toc34254684)

[1.7 HBase总结 37](#_Toc34254685)

[1.7.1 HBase存储结构 37](#_Toc34254686)

[1.7.2 RowKey设计原则 38](#_Toc34254687)

[1.7.3 RowKey如何设计 38](#_Toc34254688)

[1.7.4 Phoenix二级索引（讲原理） 38](#_Toc34254689)

[1.8 Sqoop参数 38](#_Toc34254690)

[1.8.1 Sqoop导入导出Null存储一致性问题 38](#_Toc34254691)

[1.8.2 Sqoop数据导出一致性问题 38](#_Toc34254692)

[1.8.3 Sqoop底层运行的任务是什么 39](#_Toc34254693)

[1.8.4 Sqoop数据导出的时候一次执行多长时间 39](#_Toc34254694)

[1.8.5 Sqoop一天导多少数据 39](#_Toc34254695)

[1.8.6 Sqoop在导入数据的时候数据倾斜 39](#_Toc34254696)

[1.9 Scala 39](#_Toc34254697)

[1.9.1 开发环境 39](#_Toc34254698)

[1.9.2 变量和数据类型 40](#_Toc34254699)

[1.9.3 流程控制 40](#_Toc34254700)

[1.9.4 函数式编程 40](#_Toc34254701)

[1.9.5 面向对象 40](#_Toc34254702)

[1.9.6 集合 40](#_Toc34254703)

[1.9.7 模式匹配 40](#_Toc34254704)

[1.9.8 异常 40](#_Toc34254705)

[1.9.9 隐式转换 40](#_Toc34254706)

[1.9.10 泛型 40](#_Toc34254707)

[1.10 Spark 40](#_Toc34254708)

[1.10.1 Spark有几种部署方式？请分别简要论述 40](#_Toc34254709)

[1.10.2 Spark任务使用什么进行提交，JavaEE界面还是脚本 41](#_Toc34254710)

[1.10.3 Spark提交作业参数（重点） 41](#_Toc34254711)

[1.10.4 简述Spark的架构与作业提交流程（画图讲解，注明各个部分的作用）（重点） 42](#_Toc34254712)

[1.10.5 如何理解Spark中的血统概念（RDD）（笔试重点） 42](#_Toc34254713)

[1.10.6 简述Spark的宽窄依赖，以及Spark如何划分stage，每个stage又根据什么决定task个数? （笔试重点） 43](#_Toc34254714)

[1.10.7 请列举Spark的transformation算子（不少于8个），并简述功能（重点） 43](#_Toc34254715)

[1.10.8 请列举Spark的action算子（不少于6个），并简述功能（重点） 44](#_Toc34254716)

[1.10.9 请列举会引起Shuffle过程的Spark算子，并简述功能。 44](#_Toc34254717)

[1.10.10 简述Spark的两种核心Shuffle（HashShuffle与SortShuffle）的工作流程（包括未优化的HashShuffle、优化的HashShuffle、普通的SortShuffle与bypass的SortShuffle）（重点） 44](#_Toc34254718)

[1.10.11 Spark常用算子reduceByKey与groupByKey的区别，哪一种更具优势？（重点） 46](#_Toc34254719)

[1.10.12 Repartition和Coalesce关系与区别 47](#_Toc34254720)

[1.10.13 分别简述Spark中的缓存机制（cache和persist）与checkpoint机制，并指出两者的区别与联系 47](#_Toc34254721)

[1.10.14 简述Spark中共享变量（广播变量和累加器）的基本原理与用途。（重点） 47](#_Toc34254722)

[1.10.15 当Spark涉及到数据库的操作时，如何减少Spark运行中的数据库连接数？ 48](#_Toc34254723)

[1.10.16 如何使用Spark实现TopN的获取（描述思路或使用伪代码）（重点） 48](#_Toc34254724)

[1.10.17 京东：调优之前与调优之后性能的详细对比（例如调整map个数，map个数之前多少、之后多少，有什么提升） 48](#_Toc34254725)

[1.11 Spark Sql、DataFrames、DataSet 49](#_Toc34254726)

[1.11.1 简述SparkSQL中RDD、DataFrame、DataSet三者的区别与联系? （笔试重点） 49](#_Toc34254727)

[1.11.2 append和overwrite的区别 50](#_Toc34254728)

[1.11.3 coalesce和repartition的区别 50](#_Toc34254729)

[1.11.4 cache缓存级别 50](#_Toc34254730)

[1.11.5 释放缓存和缓存 50](#_Toc34254731)

[1.11.6 Spark Shuffle默认并行度 50](#_Toc34254732)

[1.11.7 kryo序列化 50](#_Toc34254733)

[1.11.8 创建临时表和全局临时表 51](#_Toc34254734)

[1.11.9 BroadCast join 广播join 51](#_Toc34254735)

[1.11.10 控制Spark reduce缓存 调优shuffle 51](#_Toc34254736)

[1.11.11 注册UDF函数 51](#_Toc34254737)

[1.11.12 SparkSQL中join操作与left join操作的区别？ 51](#_Toc34254738)

[1.12 Spark Streaming 52](#_Toc34254739)

[1.12.1 Spark Streaming第一次运行不丢失数据 52](#_Toc34254740)

[1.12.2 Spark Streaming精准一次消费 52](#_Toc34254741)

[1.12.3 Spark Streaming控制每秒消费数据的速度 52](#_Toc34254742)

[1.12.4 Spark Streaming背压机制 52](#_Toc34254743)

[1.12.5 Spark Streaming 一个stage耗时 52](#_Toc34254744)

[1.12.6 Spark Streaming 优雅关闭 52](#_Toc34254745)

[1.12.7 Spark Streaming 默认分区个数 52](#_Toc34254746)

[1.12.8 SparkStreaming有哪几种方式消费Kafka中的数据，它们之间的区别是什么？ 52](#_Toc34254747)

[1.12.9 简述SparkStreaming窗口函数的原理（重点） 54](#_Toc34254748)

[1.13 数据倾斜 54](#_Toc34254749)

[1.13.1 数据倾斜表现 54](#_Toc34254750)

[1.13.2 数据倾斜产生原因 55](#_Toc34254751)

[1.13.3 解决数据倾斜思路 56](#_Toc34254752)

[1.13.4 定位导致数据倾斜代码 57](#_Toc34254753)

[1.13.5 查看导致数据倾斜的key分布情况 59](#_Toc34254754)

[1.13.6 Spark 数据倾斜的解决方案 60](#_Toc34254755)

[1.13.7 Spark数据倾斜处理小结 78](#_Toc34254756)

[1.14 Flink基础 78](#_Toc34254757)

[1.14.1 简单介绍一下 Flink 78](#_Toc34254758)

[1.14.2 Flink相比传统的Spark Streaming区别? 79](#_Toc34254759)

[1.14.3 Flink的组件栈有哪些？ 79](#_Toc34254760)

[1.14.4 Flink 的运行必须依赖 Hadoop组件吗？ 80](#_Toc34254761)

[1.14.5 你们的Flink集群规模多大？ 80](#_Toc34254762)

[1.14.6 Flink的基础编程模型了解吗？ 81](#_Toc34254763)

[1.14.7 Flink集群有哪些角色？各自有什么作用？ 82](#_Toc34254764)

[1.14.8 说说 Flink 资源管理中 Task Slot 的概念 83](#_Toc34254765)

[1.14.9 说说 Flink 的常用算子？ 83](#_Toc34254766)

[1.14.10 说说你知道的Flink分区策略？ 83](#_Toc34254767)

[1.14.11 Flink的并行度了解吗？Flink的并行度设置是怎样的？ 84](#_Toc34254768)

[1.14.12 Flink的Slot和parallelism有什么区别？ 85](#_Toc34254769)

[1.14.13 Flink有没有重启策略？说说有哪几种？ 85](#_Toc34254770)

[1.14.14 用过Flink中的分布式缓存吗？如何使用？ 85](#_Toc34254771)

[1.14.15 说说Flink中的广播变量，使用时需要注意什么？ 86](#_Toc34254772)

[1.14.16 说说Flink中的窗口？ 86](#_Toc34254773)

[1.14.17 说说Flink中的状态存储？ 87](#_Toc34254774)

[1.14.18 Flink中的时间有哪几类 88](#_Toc34254775)

[1.14.19 Flink 中水印是什么概念，起到什么作用？ 88](#_Toc34254776)

[1.14.20 Flink Table & SQL 熟悉吗？TableEnvironment这个类有什么作用 88](#_Toc34254777)

[1.14.20 Flink SQL的实现原理是什么？是如何实现 SQL 解析的呢？ 88](#_Toc34254778)

[1.15 Flink中级 90](#_Toc34254779)

[1.15.1 Flink是如何支持批流一体的？ 90](#_Toc34254780)

[1.15.2 Flink是如何做到高效的数据交换的？ 90](#_Toc34254781)

[1.15.3 Flink是如何做容错的？ 90](#_Toc34254782)

[1.15.4 Flink 分布式快照的原理是什么？ 90](#_Toc34254783)

[1.15.5 Flink是如何保证Exactly-once语义的？ 91](#_Toc34254784)

[1.15.6 Flink 的 kafka 连接器有什么特别的地方？ 91](#_Toc34254785)

[1.15.7 说说 Flink的内存管理是如何做的? 91](#_Toc34254786)

[1.15.8 说说 Flink的序列化如何做的? 92](#_Toc34254787)

[1.15.9 Flink中的Window出现了数据倾斜，你有什么解决办法？ 92](#_Toc34254788)

[1.15.10 Flink中在使用聚合函数 GroupBy、Distinct、KeyBy 等函数时出现数据热点该如何解决？ 93](#_Toc34254789)

[1.15.11 Flink任务延迟高，想解决这个问题，你会如何入手？ 93](#_Toc34254790)

[1.15.12 Flink是如何处理反压的？ 93](#_Toc34254791)

[1.15.13 Flink的反压和Strom有哪些不同？ 93](#_Toc34254792)

[1.15.14 Operator Chains（算子链）这个概念你了解吗？ 94](#_Toc34254793)

[1.15.15 Flink什么情况下才会把Operator chain在一起形成算子链？ 94](#_Toc34254794)

[1.15.16 说说Flink1.9的新特性？ 94](#_Toc34254795)

[1.15.17 消费kafka数据的时候，如何处理脏数据？ 94](#_Toc34254796)

[1.16 Flink高级 95](#_Toc34254797)

[1.16.1 Flink Job的提交流程 95](#_Toc34254798)

[1.16.2 Flink所谓"三层图"结构是哪几个"图"？ 95](#_Toc34254799)

[1.16.3 JobManger在集群中扮演了什么角色？ 95](#_Toc34254800)

[1.16.4 JobManger在集群启动过程中起到什么作用？ 96](#_Toc34254801)

[1.16.5 TaskManager在集群中扮演了什么角色？ 96](#_Toc34254802)

[1.16.6 TaskManager在集群启动过程中起到什么作用？ 96](#_Toc34254803)

[1.16.7 Flink 计算资源的调度是如何实现的？ 97](#_Toc34254804)

[1.16.8 简述Flink的数据抽象及数据交换过程？ 97](#_Toc34254805)

[1.16.9 Flink 中的分布式快照机制是如何实现的？ 98](#_Toc34254806)

[1.16.10 简单说说FlinkSQL的是如何实现的？ 98](#_Toc34254807)

[第2章 项目架构 99](#_Toc34254808)

[2.1 数仓概念 100](#_Toc34254809)

[2.2 系统数据流程设计 101](#_Toc34254810)

[2.3 框架版本选型 101](#_Toc34254811)

[2.4 服务器选型 102](#_Toc34254812)

[2.5 集群规模 102](#_Toc34254813)

[2.6 人员配置参考 103](#_Toc34254814)

[2.6.1 整体架构 103](#_Toc34254815)

[2.6.2 你们部门的职级等级，晋升规则 103](#_Toc34254816)

[2.6.3 人员配置参考 103](#_Toc34254817)

[第3章 用户行为数据分析 104](#_Toc34254818)

[3.1 数仓分层架构表 104](#_Toc34254819)

[3.2 埋点行为数据基本格式(基本字段) 104](#_Toc34254820)

[3.3 项目经验总结 106](#_Toc34254821)

[3.3.1 项目经验之元数据备份 106](#_Toc34254822)

[3.3.2 日期处理函数 106](#_Toc34254823)

[3.3.3 Union与Union all区别 106](#_Toc34254824)

[3.3.4 Shell中单引号和双引号区别 106](#_Toc34254825)

[3.3.5 Tez引擎优点？ 107](#_Toc34254826)

[3.4 需求逻辑（重点） 107](#_Toc34254827)

[3.4.1 如何分析用户活跃？ 107](#_Toc34254828)

[3.4.2 如何分析用户新增？vivo 107](#_Toc34254829)

[3.4.3 如何分析用户1天留存？ 107](#_Toc34254830)

[3.4.4 如何分析沉默用户？ 107](#_Toc34254831)

[3.4.5 如何分析本周回流用户？ 107](#_Toc34254832)

[3.4.6 如何分析流失用户？ 108](#_Toc34254833)

[3.4.7 如何分析最近连续3周活跃用户数？ 108](#_Toc34254834)

[3.4.8 如何分析最近七天内连续三天活跃用户数？ 108](#_Toc34254835)

[第4章 业务交互数据分析 108](#_Toc34254836)

[4.1 电商常识 108](#_Toc34254837)

[4.2 电商业务流程 109](#_Toc34254838)

[4.3 业务表关键字段 109](#_Toc34254839)

[4.3.1 订单表（order\_info） 109](#_Toc34254840)

[4.3.2 订单详情表（order\_detail） 109](#_Toc34254841)

[4.3.3 商品表 110](#_Toc34254842)

[4.3.4 用户表 110](#_Toc34254843)

[4.3.5 商品一级分类表 110](#_Toc34254844)

[4.3.6 商品二级分类表 110](#_Toc34254845)

[4.3.7 商品三级分类表 111](#_Toc34254846)

[4.3.8 支付流水表 111](#_Toc34254847)

[4.4 MySql中表的分类 111](#_Toc34254848)

[4.5 同步策略 112](#_Toc34254849)

[4.6 关系型数据库范式理论 112](#_Toc34254850)

[4.7 数据模型 113](#_Toc34254851)

[4.8 拉链表 113](#_Toc34254852)

[第5章 即席查询数据仓库 115](#_Toc34254853)

[第6章 项目开发经验 115](#_Toc34254854)

[6.1 项目开发中遇到哪些问题 115](#_Toc34254855)

[6.1.1 Hadoop 115](#_Toc34254856)

[6.1.2 Flume 116](#_Toc34254857)

[6.1.3 Kafka 117](#_Toc34254858)

[6.1.4 Hive 118](#_Toc34254859)

[6.1.5 MySql 118](#_Toc34254860)

[6.1.6 Tez引擎优点？ 119](#_Toc34254861)

[6.1.7 Sqoop 119](#_Toc34254862)

[6.1.8 Azkaban 120](#_Toc34254863)

[6.1.9 Spark 120](#_Toc34254864)

[6.2 业务经验 121](#_Toc34254865)

[6.2.1 ODS层采用什么压缩方式和存储格式？ 121](#_Toc34254866)

[6.2.2 DWD层做了哪些事？ 121](#_Toc34254867)

[6.2.3 DWS层做了哪些事？ 122](#_Toc34254868)

[6.2.4 分析过哪些指标（一分钟至少说出30个指标） 123](#_Toc34254869)

[6.2.5 分析过最难的两个指标，现场手写 126](#_Toc34254870)

[6.2.6 数据仓库每天跑多少张表，大概什么时候运行，运行多久？ 127](#_Toc34254871)

[6.2.7 数仓中使用的哪种文件存储格式 127](#_Toc34254872)

[6.2.8 数仓中用到过哪些Shell脚本及具体功能 127](#_Toc34254873)

[6.2.9 Shell中提交了一个脚本，进程号已经不知道了，但是需要kill掉这个进程，怎么操作? 127](#_Toc34254874)

[6.2.10 项目中用过的报表工具 128](#_Toc34254875)

[6.2.11 Hive表底层建模用的工具是什么 128](#_Toc34254876)

[6.2.12 活动的话，数据量会增加多少？怎么解决？ 128](#_Toc34254877)

[6.2.13 哪张表最费时间，有没有优化 128](#_Toc34254878)

[6.2.14 用的什么工具做权限管理 128](#_Toc34254879)

[6.2.15并发峰值多少？大概哪个时间点？ 128](#_Toc34254880)

[6.2.16 测试相关 128](#_Toc34254881)

[6.2.17 项目实际工作流程 129](#_Toc34254882)

[6.2.18 项目中实现一个需求大概多长时间 130](#_Toc34254883)

[6.2.19 项目在3年内迭代次数，每一个项目具体是如何迭代的。公司版本迭代多久一次，迭代到哪个版本 130](#_Toc34254884)

[6.2.20 项目开发中每天做什么事 131](#_Toc34254885)

[6.3 元数据管理（Atlas血缘系统） 131](#_Toc34254886)

[6.4 数据质量监控（Griffin） 131](#_Toc34254887)

[6.4.1 为什么要做数据质量监控（2019年下半年） 131](#_Toc34254888)

[6.4.2 建设方法 132](#_Toc34254889)

[6.4.3 监控指标 133](#_Toc34254890)

[6.4.4 Griffin数据质量监控实现 134](#_Toc34254891)

[6.5 数据治理 134](#_Toc34254892)

[6.6 数据中台 135](#_Toc34254893)

[6.7 数据湖 135](#_Toc34254894)

[6.8 埋点 136](#_Toc34254895)

[第7章 电商运营经验 136](#_Toc34254896)

[7.1 电商指标整理 **错误!未定义书签。**](#_Toc34254897)

[7.2 电商8类基本指标 136](#_Toc34254898)

[7.3 五大关键电商指标和三个思路 **错误!未定义书签。**](#_Toc34254899)

[7.4 电商运营数据挖掘 **错误!未定义书签。**](#_Toc34254900)

[7.5 直播指标 140](#_Toc34254901)

[第8章 手写代码 144](#_Toc34254902)

[8.1 基本算法 144](#_Toc34254903)

[8.1.1 冒泡排序 144](#_Toc34254904)

[8.1.2 二分查找 146](#_Toc34254905)

[8.1.3 快排 148](#_Toc34254906)

[8.1.4 归并 149](#_Toc34254907)

[8.1.5 二叉树之Scala实现 150](#_Toc34254908)

[8.2 开发代码 154](#_Toc34254909)

[8.2.1 手写Spark-WordCount 154](#_Toc34254910)

[8.3 手写HQL 155](#_Toc34254911)

[8.3.1 手写HQL 第1题 155](#_Toc34254912)

[8.3.2 手写HQL 第2题 156](#_Toc34254913)

[8.3.3 手写HQL 第3题 158](#_Toc34254914)

[8.3.4 手写HQL 第4题 159](#_Toc34254915)

[8.3.5 手写HQL 第5题 160](#_Toc34254916)

[8.3.6 手写HQL 第6题 164](#_Toc34254917)

[8.3.7 手写HQL 第7题 165](#_Toc34254918)

[8.3.8 手写SQL 第8题 166](#_Toc34254919)

[8.3.9 手写HQL 第9题 167](#_Toc34254920)

[8.3.10 手写HQL 第10题 169](#_Toc34254921)

[第9章 JavaSE 172](#_Toc34254922)

[9.1 HashMap底层源码，数据结构 172](#_Toc34254923)

[9.2 Java自带哪几种线程池？ 175](#_Toc34254924)

[9.3 HashMap和HashTable区别 176](#_Toc34254925)

[9.4 TreeSet和HashSet区别 177](#_Toc34254926)

[9.5 String buffer和String build区别 177](#_Toc34254927)

[9.6 Final、Finally、Finalize 177](#_Toc34254928)

[9.7 ==和Equals区别 178](#_Toc34254929)

[第10章 Redis 178](#_Toc34254930)

[10.1 缓存穿透、缓存雪崩、缓存击穿 178](#_Toc34254931)

[10.2 哨兵模式 179](#_Toc34254932)

[10.3 数据类型 179](#_Toc34254933)

[10.4 持久化 179](#_Toc34254934)

[11.5 悲观锁 180](#_Toc34254935)

[11.6 乐观锁 180](#_Toc34254936)

[第11章 MySql 180](#_Toc34254937)

[11.1 MyISAM与InnoDB的区别 180](#_Toc34254938)

[11.2 索引优化 180](#_Toc34254939)

[11.3 b-tree和b+tree的区别 181](#_Toc34254940)

[11.4 redis是单线程的，为什么那么快 181](#_Toc34254941)

[11.5 MySQL的事务 181](#_Toc34254942)

[第12章 JVM 183](#_Toc34254943)

[12.1 JVM内存分哪几个区，每个区的作用是什么? 183](#_Toc34254944)

[12.2 Java类加载过程? 184](#_Toc34254945)

[12.3 java中垃圾收集的方法有哪些? 185](#_Toc34254946)

[12.4 如何判断一个对象是否存活?(或者GC对象的判定方法) 185](#_Toc34254947)

[12.5 什么是类加载器，类加载器有哪些? 186](#_Toc34254948)

[12.6 简述Java内存分配与回收策略以及Minor GC和Major GC（full GC） 186](#_Toc34254949)

[第13章 JUC 187](#_Toc34254950)

[**13.1 Synchronized与Lock的区别** 187](#_Toc34254951)

[**13.2 Runnable和Callable的区别** 187](#_Toc34254952)

[**13.3 什么是分布式锁** 187](#_Toc34254953)

[**13.4 什么是分布式事务** 187](#_Toc34254954)

[第14章 面试说明 187](#_Toc34254955)

[14.1 面试过程最关键的是什么？ 187](#_Toc34254956)

[14.2 面试时该怎么说？ 187](#_Toc34254957)

[14.3 面试技巧 188](#_Toc34254958)

[14.3.1 六个常见问题 188](#_Toc34254959)

[14.3.2 两个注意事项 189](#_Toc34254960)

[14.3.3 自我介绍（控制在4分半以内，不超过5分钟） 189](#_Toc34254961)

[第15章 LeetCode题目精选 189](#_Toc34254962)

[15.1 两数之和 189](#_Toc34254963)

[15.1.1 问题描述 189](#_Toc34254964)

[15.1.2 参考答案 189](#_Toc34254965)

[15.2 爬楼梯 190](#_Toc34254966)

[15.2.1 问题描述 190](#_Toc34254967)

[15.2.2 参考答案 191](#_Toc34254968)

[15.3 翻转二叉树 191](#_Toc34254969)

[15.3.1 问题描述 191](#_Toc34254970)

[15.3.2 参考答案 192](#_Toc34254971)

[15.4 反转链表 193](#_Toc34254972)

[15.4.1 问题描述 193](#_Toc34254973)

[15.4.2 参考答案 193](#_Toc34254974)

[15.5 LRU缓存机制 193](#_Toc34254975)

[15.5.1 问题描述 194](#_Toc34254976)

[15.5.2 参考答案 194](#_Toc34254977)

[15.6 最长回文子串 195](#_Toc34254978)

[15.6.1 问题描述 196](#_Toc34254979)

[15.6.2 参考答案 196](#_Toc34254980)

[15.7 有效的括号 197](#_Toc34254981)

[15.7.1 问题描述 197](#_Toc34254982)

[15.7.2 参考答案 198](#_Toc34254983)

[15.8 数组中的第K个最大元素 200](#_Toc34254984)

[15.8.1 问题描述 200](#_Toc34254985)

[15.8.2 参考答案 200](#_Toc34254986)

[15.9 实现 Trie (前缀树) 203](#_Toc34254987)

[15.9.1 问题描述 203](#_Toc34254988)

[15.9.2 参考答案 203](#_Toc34254989)

[15.10 编辑距离 205](#_Toc34254990)

[15.10.1 问题描述 205](#_Toc34254991)

[15.10.2 参考答案 206](#_Toc34254992)

# 第1章 项目涉及技术

## 1.1 Linux&Shell相关总结

### 1.1.1 Linux常用命令

|  |  |  |
| --- | --- | --- |
| 序号 | 命令 | 命令解释 |
| 1 | top | 查看内存 |
| 2 | df -h | 查看磁盘存储情况 |
| 3 | iotop | 查看磁盘IO读写(yum install iotop安装） |
| 4 | iotop -o | 直接查看比较高的磁盘读写程序 |
| 5 | netstat -tunlp | grep 端口号 | 查看端口占用情况 |
| 6 | uptime | 查看报告系统运行时长及平均负载 |
| 7 | ps aux | 查看进程 |

### 1.1.2 Shell常用工具

awk、sed、cut、sort

## 1.2 Hadoop相关总结

### 1.2.1 Hadoop常用端口号

* dfs.namenode.http-address:50070
* dfs.datanode.http-address:50075
* SecondaryNameNode辅助名称节点端口号：50090
* dfs.datanode.address:50010
* fs.defaultFS:8020 或者9000
* yarn.resourcemanager.webapp.address:8088
* 历史服务器web访问端口：19888

### 1.2.2 Hadoop配置文件以及简单的Hadoop集群搭建

（1）配置文件：

core-site.xml、hdfs-site.xml、mapred-site.xml、yarn-site.xml

hadoop-env.sh、yarn-env.sh、mapred-env.sh、slaves

（2）简单的集群搭建过程：

JDK安装

配置SSH免密登录

配置hadoop核心文件:

格式化namenode

### 1.2.3 HDFS读流程和写流程

略

### 1.2.4 MapReduce的Shuffle过程及Hadoop优化（包括：压缩、小文件、集群优化）

**一、Shuffle机制**

1）Map方法之后Reduce方法之前这段处理过程叫Shuffle

2）Map方法之后，数据首先进入到分区方法，把数据标记好分区，然后把数据发送到环形缓冲区；环形缓冲区默认大小100m，环形缓冲区达到80%时，进行溢写；溢写前对数据进行排序，排序按照对key的索引进行字典顺序排序，排序的手段快排；溢写产生大量溢写文件，需要对溢写文件进行归并排序；对溢写的文件也可以进行Combiner操作，前提是汇总操作，求平均值不行。最后将文件按照分区存储到磁盘，等待Reduce端拉取。

3）每个Reduce拉取Map端对应分区的数据。拉取数据后先存储到内存中，内存不够了，再存储到磁盘。拉取完所有数据后，采用归并排序将内存和磁盘中的数据都进行排序。在进入Reduce方法前，可以对数据进行分组操作。

**二、Hadoop优化**

0）HDFS小文件影响

（1）影响NameNode的寿命，因为文件元数据存储在NameNode的内存中

（2）影响计算引擎的任务数量，比如每个小的文件都会生成一个Map任务

1）数据输入小文件处理：

（1）合并小文件：对小文件进行归档（Har）、自定义Inputformat将小文件存储成SequenceFile文件。

（2）采用ConbinFileInputFormat来作为输入，解决输入端大量小文件场景。

（3）对于大量小文件Job，可以开启JVM重用。

2）Map阶段

（1）增大环形缓冲区大小。由100m扩大到200m

（2）增大环形缓冲区溢写的比例。由80%扩大到90%

（3）减少对溢写文件的merge次数。（10个文件，一次20个merge）

（4）不影响实际业务的前提下，采用Combiner提前合并，减少 I/O。

3）Reduce阶段

（1）合理设置Map和Reduce数：两个都不能设置太少，也不能设置太多。太少，会导致Task等待，延长处理时间；太多，会导致 Map、Reduce任务间竞争资源，造成处理超时等错误。

（2）设置Map、Reduce共存：调整slowstart.completedmaps参数，使Map运行到一定程度后，Reduce也开始运行，减少Reduce的等待时间。

（3）规避使用Reduce，因为Reduce在用于连接数据集的时候将会产生大量的网络消耗。

（4）增加每个Reduce去Map中拿数据的并行数

（5）集群性能可以的前提下，增大Reduce端存储数据内存的大小。

4）IO传输

（1）采用数据压缩的方式，减少网络IO的的时间。安装Snappy和LZOP压缩编码器。

（2）使用SequenceFile二进制文件

5）整体

（1）MapTask默认内存大小为1G，可以增加MapTask内存大小为4-5g

（2）ReduceTask默认内存大小为1G，可以增加ReduceTask内存大小为4-5g

（3）可以增加MapTask的cpu核数，增加ReduceTask的CPU核数

（4）增加每个Container的CPU核数和内存大小

（5）调整每个Map Task和Reduce Task最大重试次数

**三、压缩**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 压缩格式 | Hadoop自带？ | 算法 | 文件扩展名 | 支持切分 | 换成压缩格式后，原来的程序是否需要修改 |
| DEFLATE | 是，直接使用 | DEFLATE | .deflate | 否 | 和文本处理一样，不需要修改 |
| Gzip | 是，直接使用 | DEFLATE | .gz | 否 | 和文本处理一样，不需要修改 |
| bzip2 | 是，直接使用 | bzip2 | .bz2 | 是 | 和文本处理一样，不需要修改 |
| LZO | 否，需要安装 | LZO | .lzo | 是 | 需要建索引，还需要指定输入格式 |
| Snappy | 否，需要安装 | Snappy | .snappy | 否 | 和文本处理一样，不需要修改 |

提示：如果面试过程问起，我们一般回答压缩方式为Snappy，特点速度快，缺点无法切分（可以回答在链式MR中，Reduce端输出使用bzip2压缩，以便后续的map任务对数据进行split）

**四、切片机制**

1）简单地按照文件的内容长度进行切片

2）切片大小，默认等于Block大小

3）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

提示：切片大小公式：max(0,min(Long\_max,blockSize))

### 1.2.5 Yarn的Job提交流程

略

### 1.2.6 Yarn的默认调度器、调度器分类、以及他们之间的区别

1）Hadoop调度器重要分为三类：

FIFO 、Capacity Scheduler（容量调度器）和Fair Sceduler（公平调度器）。

Hadoop2.7.2默认的资源调度器是 容量调度器

2）区别：

FIFO调度器：先进先出，同一时间队列中只有一个任务在执行。

容量调度器：多队列；每个队列内部先进先出，同一时间队列中只有一个任务在执行。队列的并行度为队列的个数。

公平调度器：多队列；每个队列内部按照缺额大小分配资源启动任务，同一时间队列中有多个任务执行。队列的并行度大于等于队列的个数。

3）一定要强调生产环境中不是使用的FifoScheduler，面试的时侯会发现候选人大概了解这几种调度器的区别，但是问在生产环境用哪种，却说使用的FifoScheduler（企业生产环境一定不会用这个调度的）

### 1.2.7 项目经验之LZO压缩

Hadoop默认不支持LZO压缩，如果需要支持LZO压缩，需要添加jar包，并在hadoop的cores-site.xml文件中添加相关压缩配置。

### 1.2.8 Hadoop参数调优

1）在hdfs-site.xml文件中配置多目录，最好提前配置好，否则更改目录需要重新启动集群

2）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。

dfs.namenode.handler.count=20 \* log2(Cluster Size)，比如集群规模为10台时，此参数设置为60

3）服务器节点上YARN可使用的物理内存总量，默认是8192（MB），注意，如果你的节点内存资源不够8GB，则需要调减小这个值，而YARN不会智能的探测节点的物理内存总量。yarn.nodemanager.resource.memory-mb

4）单个任务可申请的最多物理内存量，默认是8192（MB）。yarn.scheduler.maximum-allocation-mb

### 1.2.9 项目经验之基准测试

搭建完Hadoop集群后需要对HDFS读写性能和MR计算能力测试。测试jar包在hadoop的share文件夹下。

### 1.2.10 Hadoop宕机

1）如果MR造成系统宕机。此时要控制Yarn同时运行的任务数，和每个任务申请的最大内存。调整参数：yarn.scheduler.maximum-allocation-mb（单个任务可申请的最多物理内存量，默认是8192MB）

2）如果写入文件过量造成NameNode宕机。那么调高Kafka的存储大小，控制从Kafka到HDFS的写入速度。高峰期的时候用Kafka进行缓存，高峰期过去数据同步会自动跟上。

### 1.2.11 Hadoop解决数据倾斜方法

**1）提前在map进行combine，减少传输的数据量**

在Mapper加上combiner相当于提前进行reduce，即把一个Mapper中的相同key进行了聚合，减少shuffle过程中传输的数据量，以及Reducer端的计算量。

如果导致数据倾斜的key 大量分布在不同的mapper的时候，这种方法就不是很有效了。

**2）导致数据倾斜的key 大量分布在不同的mapper**

（1）局部聚合加全局聚合。

第一次在map阶段对那些导致了数据倾斜的key 加上1到n的随机前缀，这样本来相同的key 也会被分到多个Reducer中进行局部聚合，数量就会大大降低。

第二次mapreduce，去掉key的随机前缀，进行全局聚合。

思想：二次mr，第一次将key随机散列到不同reducer进行处理达到负载均衡目的。第二次再根据去掉key的随机前缀，按原key进行reduce处理。

这个方法进行两次mapreduce，性能稍差。

（2）增加Reducer，提升并行度  
JobConf.setNumReduceTasks(int)

（3）实现自定义分区

根据数据分布情况，自定义散列函数，将key均匀分配到不同Reducer

### 1.2.12 集群资源分配参数（项目中遇到的问题）

集群有30台机器，跑mr任务的时候发现5个map任务全都分配到了同一台机器上，这个可能是由于什么原因导致的吗？

解决方案：yarn.scheduler.fair.assignmultiple 这个参数 默认是开的，需要关掉

<https://blog.csdn.net/leone911/article/details/51605172>

## 1.3 Zookeeper相关总结

### 1.3.1 选举机制

半数机制：2n+1

10台服务器：3台

20台服务器：5台

100台服务器：11台

台数并不是越多越好。 太多选举时间过长影响性能。

### 1.3.2 常用命令

ls、get、create

## 1.4 Flume相关总结

### 1.4.1 Flume组成，Put事务，Take事务

Taildir Source：断点续传、多目录。Flume1.6以前需要自己自定义Source记录每次读取文件位置，实现断点续传。

File Channel：数据存储在磁盘，宕机数据可以保存。但是传输速率慢。适合对数据传输可靠性要求高的场景，比如，金融行业。

Memory Channel：数据存储在内存中，宕机数据丢失。传输速率快。适合对数据传输可靠性要求不高的场景，比如，普通的日志数据。

Kafka Channel：减少了Flume的Sink阶段，提高了传输效率。

Source到Channel是Put事务

Channel到Sink是Take事务

### 1.4.2 Flume拦截器

（1）拦截器注意事项

项目中自定义了：ETL拦截器和区分类型拦截器。

采用两个拦截器的优缺点：优点，模块化开发和可移植性；缺点，性能会低一些

（2）自定义拦截器步骤

a）实现 Interceptor

b）重写四个方法

* initialize 初始化
* public Event intercept(Event event) 处理单个Event
* public List<Event> intercept(List<Event> events) 处理多个Event，在这个方法中调用Event intercept(Event event)
* close 方法

c）静态内部类，实现Interceptor.Builder

### 1.4.3 Flume Channel选择器

Channel Selectors，可以让不同的项目日志通过不同的Channel到不同的Sink中去。官方文档上Channel Selectors有两种类型:Replicating Channel Selector(default)和Multiplexing Channel selector

这两种Selector的区别是:Replicating会将source过来的events发往所有channel, 而Multiplexing可以选择该发往哪些Channel。

### 1.4.4 Flume监控器

Ganglia

### 1.4.5 Flume采集数据会丢失吗?（防止数据丢失的机制）

不会，Channel存储可以存储在File中，数据传输自身有事务。

### 1.4.6 Flume内存

开发中在flume-env.sh中设置JVM heap为4G或更高，部署在单独的服务器上（4核8线程16G内存）

-Xmx与-Xms最好设置一致，减少内存抖动带来的性能影响，如果设置不一致容易导致频繁fullgc。

-Xms表示JVM Heap(堆内存)最小尺寸，初始分配；-Xmx 表示JVM Heap(堆内存)最大允许的尺寸，按需分配。如果不设置一致，容易在初始化时，由于内存不够，频繁触发fullgc。

### 1.4.7 FileChannel优化

通过配置dataDirs指向多个路径，每个路径对应不同的硬盘，增大Flume吞吐量。

官方说明如下：

Comma separated list of directories for storing log files. Using multiple directories on separate disks can improve file channel peformance

checkpointDir和backupCheckpointDir也尽量配置在不同硬盘对应的目录中，保证checkpoint坏掉后，可以快速使用backupCheckpointDir恢复数据

### 1.4.8 Flume Channel容量

File channel 容量1000000条

Memory channel 容量 100条

### 1.4.9 HDFS Sink小文件处理

（1）HDFS存入大量小文件，有什么影响？

**元数据层面：**每个小文件都有一份元数据，其中包括文件路径，文件名，所有者，所属组，权限，创建时间等，这些信息都保存在Namenode内存中。所以小文件过多，会占用Namenode服务器大量内存，影响Namenode性能和使用寿命

**计算层面：**默认情况下MR会对每个小文件启用一个Map任务计算，非常影响计算性能。同时也影响磁盘寻址时间。

（2）HDFS小文件处理

官方默认的这三个参数配置写入HDFS后会产生小文件，hdfs.rollInterval、hdfs.rollSize、hdfs.rollCount

基于以上hdfs.rollInterval=3600，hdfs.rollSize=134217728，hdfs.rollCount =0，hdfs.roundValue=3600，hdfs.roundUnit= second几个参数综合作用，效果如下：

（1）tmp文件在达到128M时会滚动生成正式文件

（2）tmp文件创建超3600秒时会滚动生成正式文件

举例：在2018-01-01 05:23的时侯sink接收到数据，那会产生如下tmp文件：

/atguigu/20180101/atguigu.201801010520.tmp

即使文件内容没有达到128M，也会在06:23时滚动生成正式文件

## 1.5 Kafka相关总结

### 1.5.1 Kafka架构

略

### 1.5.2 Kafka压测

Kafka官方自带压力测试脚本（kafka-consumer-perf-test.sh、kafka-producer-perf-test.sh）。Kafka压测时，可以查看到哪个地方出现了瓶颈（CPU，内存，网络IO）。一般都是网络IO达到瓶颈。

### 1.5.3 Kafka的机器数量

Kafka机器数量=2\*（峰值生产速度\*副本数/100）+1

### 1.5.4 Kafka的日志保存时间

3天

### 1.5.5 Kafka的硬盘大小

每天的数据量\*3天/70%

### 1.5.6 Kafka监控

公司自己开发的监控器；

开源的监控器：KafkaManager、KafkaMonitor、KafkaEagle

### 1.5.7 Kakfa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Tp和Tc，单位可以是MB/s。

4）然后假设总的目标吞吐量是Tt，那么分区数=Tt / min（Tp，Tc）

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

<https://blog.csdn.net/weixin_42641909/article/details/89294698>

分区数一般设置为：3-10个

### 1.5.8 副本数设定

一般我们设置成2个或3个，很多企业设置为2个。

### 1.5.9 多少个Topic

通常情况：多少个日志类型就多少个Topic。也有对日志类型进行合并的。

### 1.5.10 Kafka丢不丢数据

Ack=0，相当于异步发送，消息发送完毕即offset增加，继续生产。

Ack=1，leader收到leader replica 对一个消息的接受ack才增加offset，然后继续生产。

Ack=-1，leader收到所有replica 对一个消息的接受ack才增加offset，然后继续生产。

### 1.5.11 Kafka的ISR副本同步队列

ISR（In-Sync Replicas），副本同步队列。ISR中包括Leader和Follower。如果Leader进程挂掉，会在ISR队列中选择一个服务作为新的Leader。有replica.lag.max.messages（延迟条数）和replica.lag.time.max.ms（延迟时间）两个参数决定一台服务是否可以加入ISR副本队列，在0.10版本移除了replica.lag.max.messages参数，防止服务频繁的进去队列。

任意一个维度超过阈值都会把Follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的Follower也会先存放在OSR中。

### 1.5.12 Kafka分区分配策略

在 Kafka内部存在两种默认的分区分配策略：Range和 RoundRobin。

Range是默认策略。Range是对每个Topic而言的（即一个Topic一个Topic分），首先对同一个Topic里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。然后用Partitions分区的个数除以消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区。

例如：我们有10个分区，两个消费者（C1，C2），3个消费者线程，10 / 3 = 3而且除不尽。

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6 分区

C2-1 将消费 7, 8, 9 分区

第一步：将所有主题分区组成TopicAndPartition列表，然后对TopicAndPartition列表按照hashCode进行排序，最后按照轮询的方式发给每一个消费线程。

### 1.5.13 Kafka中数据量计算

每天总数据量100g，每天产生1亿条日志， 10000万/24/60/60=1150条/每秒钟

平均每秒钟：1150条

低谷每秒钟：50条

高峰每秒钟：1150条\*（2-20倍）=2300条-23000条

每条日志大小：0.5k-2k（取1k）

每秒多少数据量：2.0M-20MB

### 1.5.14 Kafka挂掉

1）Flume记录

2）日志有记录

3）短期没事

### 1.5.15 Kafka消息数据积压，Kafka消费能力不足怎么处理？

1）如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

2）如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

### 1.5.16 Kafka幂等性

Producer的幂等性指的是当发送同一条消息时，数据在Server端只会被持久化一次，数据不丟不重，但是这里的幂等性是有条件的：

1）**只能保证Producer在单个会话内不丟不重，如果Producer出现意外挂掉再重启是无法保证的**（幂等性情况下，是无法获取之前的状态信息，因此是无法做到跨会话级别的不丢不重）。

2）幂等性不能跨多个Topic-Partition，**只能保证单个Partition内的幂等性**，当涉及多个 Topic-Partition时，这中间的状态并没有同步。

### 1.5.17 Kafka事务

Kafka从0.11版本开始引入了事务支持。事务可以保证Kafka在Exactly Once语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败。

1）Producer事务

为了实现跨分区跨会话的事务，需要引入一个全局唯一的Transaction ID，并将Producer获得的PID和Transaction ID绑定。这样当Producer重启后就可以通过正在进行的Transaction ID获得原来的PID。

为了管理Transaction，Kafka引入了一个新的组件Transaction Coordinator。Producer就是通过和Transaction Coordinator交互获得Transaction ID对应的任务状态。Transaction Coordinator还负责将事务所有写入Kafka的一个内部Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行。

2）Consumer事务

上述事务机制主要是从Producer方面考虑，对于Consumer而言，事务的保证就会相对较弱，尤其时无法保证Commit的信息被精确消费。这是由于Consumer可以通过offset访问任意信息，而且不同的Segment File生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

### 1.5.18 Kafka数据重复

幂等性+ack-1+事务

Kafka数据重复，可以再下一级：SparkStreaming、redis或者hive中dwd层去重，去重的手段：分组、按照id开窗只取第一个值；

### 1.5.19 Kafka参数优化

**1）Broker参数配置（server.properties）**

1、网络和io操作线程配置优化

# broker处理消息的最大线程数（默认为3）

num.network.threads=cpu核数+1

# broker处理磁盘IO的线程数

num.io.threads=cpu核数\*2

2、log数据文件刷盘策略

# 每当producer写入10000条消息时，刷数据到磁盘

log.flush.interval.messages=10000

# 每间隔1秒钟时间，刷数据到磁盘

log.flush.interval.ms=1000

3、日志保留策略配置

# 保留三天，也可以更短 （log.cleaner.delete.retention.ms）

log.retention.hours=72

4、Replica相关配置

offsets.topic.replication.factor:3

# 这个参数指新创建一个topic时，默认的Replica数量,Replica过少会影响数据的可用性，太多则会白白浪费存储资源，一般建议在2~3为宜。

**2）Producer优化（producer.properties）**

buffer.memory:33554432 (32m)

#在Producer端用来存放尚未发送出去的Message的缓冲区大小。缓冲区满了之后可以选择阻塞发送或抛出异常，由block.on.buffer.full的配置来决定。

compression.type:none

#默认发送不进行压缩，推荐配置一种适合的压缩算法，可以大幅度的减缓网络压力和Broker的存储压力。

**4）Kafka内存调整（**kafka-server-start.sh**）**

默认内存1个G，生产环境尽量不要超过6个G。

export KAFKA\_HEAP\_OPTS="-Xms4g -Xmx4g"

### 1.5.20 Kafka高效读写数据

**1）Kafka本身是分布式集群，同时采用分区技术，并发度高。**

**2）顺序写磁盘**

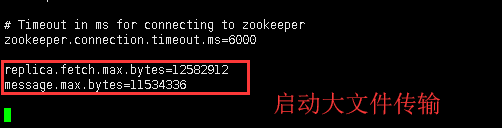
Kafka的producer生产数据，要写入到log文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到600M/s，而随机写只有100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

**3）零复制技术**

略

### 1.5.21 Kafka支持传输

kafka对于消息体的大小默认为单条最大值是1M但是在我们应用场景中, 常常会出现一条消息大于1M，如果不对kafka进行配置。则会出现生产者无法将消息推送到kafka或消费者无法去消费kafka里面的数据, 这时我们就要对kafka进行以下配置：server.properties



### 1.5.22 Kafka过期数据清理

保证数据没有被引用（没人消费他）

日志清理保存的策略只有delete和compact两种

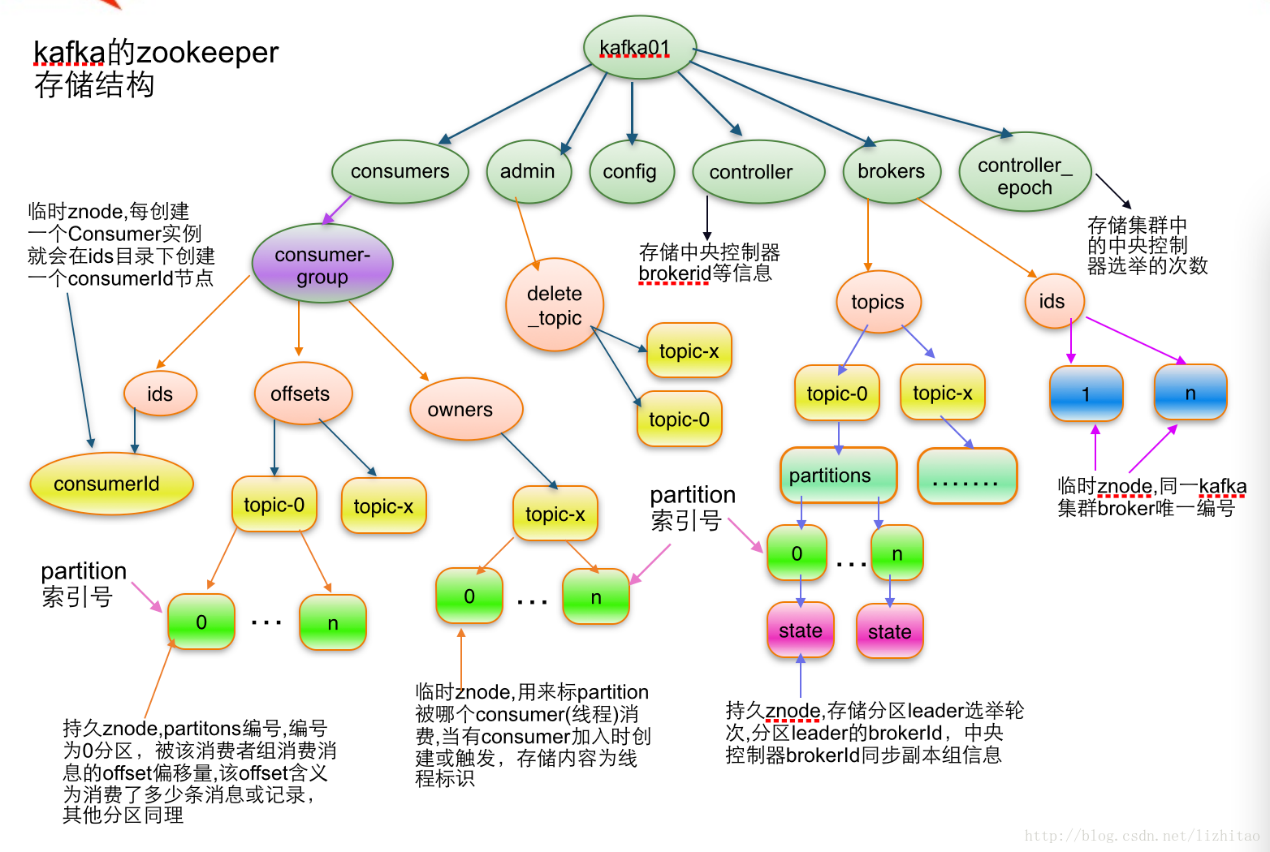
log.cleanup.policy=delete启用删除策略

log.cleanup.policy=compact启用压缩策略

### 1.5.23 Kafka可以按照时间消费数据

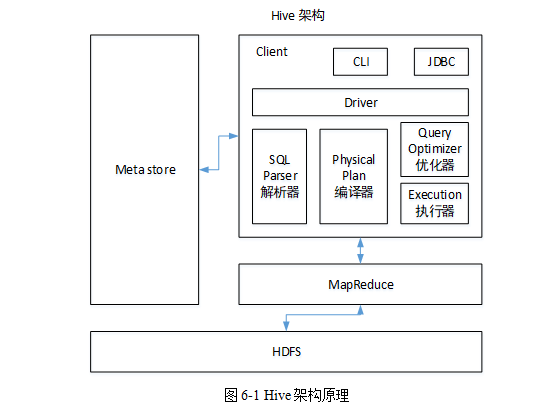
Map<TopicPartition, OffsetAndTimestamp> startOffsetMap = KafkaUtil.fetchOffsetsWithTimestamp(topic, sTime, kafkaProp);

### 1.5.24 Zookeeper中存储kafka哪些信息



## 1.6 Hive总结

### 1.6.1 Hive的架构



### 1.6.2 Hive和数据库比较

Hive 和数据库除了拥有类似的查询语言，再无类似之处。

1）数据存储位置

Hive 存储在 HDFS 。数据库将数据保存在块设备或者本地文件系统中。

2）数据更新

Hive中不建议对数据的改写。而数据库中的数据通常是需要经常进行修改的，

3）执行延迟

Hive 执行延迟较高。数据库的执行延迟较低。当然，这个是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive的并行计算显然能体现出优势。

4）数据规模

Hive支持很大规模的数据计算；数据库可以支持的数据规模较小。

### 1.6.3 内部表和外部表

1）管理表：当我们删除一个管理表时，Hive也会删除这个表中数据。管理表不适合和其他工具共享数据。

2）外部表：删除该表并不会删除掉原始数据，删除的是表的元数据

### 1.6.4 4个By区别

1）Sort By：分区内有序；

2）Order By：全局排序，只有一个Reducer；

3）Distrbute By：类似MR中Partition，进行分区，结合sort by使用。

4） Cluster By：当Distribute by和Sorts by字段相同时，可以使用Cluster by方式。Cluster by除了具有Distribute by的功能外还兼具Sort by的功能。但是排序只能是升序排序，不能指定排序规则为ASC或者DESC。

### 1.6.5 窗口函数

RANK() 排序相同时会重复，总数不会变

DENSE\_RANK() 排序相同时会重复，总数会减少

ROW\_NUMBER() 会根据顺序计算

1） OVER()：指定分析函数工作的数据窗口大小，这个数据窗口大小可能会随着行的变而变化

2）CURRENT ROW：当前行

3）n PRECEDING：往前n行数据

4） n FOLLOWING：往后n行数据

5）UNBOUNDED：起点，UNBOUNDED PRECEDING 表示从前面的起点， UNBOUNDED FOLLOWING表示到后面的终点

6） LAG(col,n)：往前第n行数据

7）LEAD(col,n)：往后第n行数据

8） NTILE(n)：把有序分区中的行分发到指定数据的组中，各个组有编号，编号从1开始，对于每一行，NTILE返回此行所属的组的编号。注意：n必须为int类型。

### 1.6.6 自定义UDF、UDTF

在项目中是否自定义过UDF、UDTF函数，以及用他们处理了什么问题，及自定义步骤？

1）自定义过。

2）用UDF函数解析公共字段；用UDTF函数解析事件字段。

自定义UDF：继承UDF，重写evaluate方法

自定义UDTF：继承自GenericUDTF，重写3个方法：initialize(自定义输出的列名和类型)，process（将结果返回forward(result)），close

为什么要自定义UDF/UDTF，因为自定义函数，可以自己埋点Log打印日志，出错或者数据异常，方便调试.

### 1.6.7 Hive优化

**1）MapJoin**

如果不指定MapJoin或者不符合MapJoin的条件，那么Hive解析器会将Join操作转换成Common Join，即：在Reduce阶段完成join。容易发生数据倾斜。可以用MapJoin把小表全部加载到内存在map端进行join，避免reducer处理。

**2）行列过滤**

列处理：在SELECT中，只拿需要的列，如果有，尽量使用分区过滤，少用SELECT \*。

行处理：在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在Where后面，那么就会先全表关联，之后再过滤。

**3）列式存储**

**4）采用分区技术**

**5）合理设置Map数**

**（1）通常情况下，作业会通过input的目录产生一个或者多个map任务。**

主要的决定因素有：input的文件总个数，input的文件大小，集群设置的文件块大小。

**（2）是不是map数越多越好？**

答案是否定的。如果一个任务有很多小文件（远远小于块大小128m），则每个小文件也会被当做一个块，用一个map任务来完成，而一个map任务启动和初始化的时间远远大于逻辑处理的时间，就会造成很大的资源浪费。而且，同时可执行的map数是受限的。

**（3）是不是保证每个map处理接近128m的文件块，就高枕无忧了？**

答案也是不一定。比如有一个127m的文件，正常会用一个map去完成，但这个文件只有一个或者两个小字段，却有几千万的记录，如果map处理的逻辑比较复杂，用一个map任务去做，肯定也比较耗时。

针对上面的问题2和3，我们需要采取两种方式来解决：即减少map数和增加map数；

**6）小文件进行合并**

在Map执行前合并小文件，减少Map数：CombineHiveInputFormat具有对小文件进行合并的功能（系统默认的格式）。HiveInputFormat没有对小文件合并功能。

**7）合理设置Reduce数**

Reduce个数并不是越多越好

（1）过多的启动和初始化Reduce也会消耗时间和资源；

（2）另外，有多少个Reduce，就会有多少个输出文件，如果生成了很多个小文件，那么如果这些小文件作为下一个任务的输入，则也会出现小文件过多的问题；

在设置Reduce个数的时候也需要考虑这两个原则：处理大数据量利用合适的Reduce数；使单个Reduce任务处理数据量大小要合适；

**8）常用参数**

// 输出合并小文件

SET hive.merge.mapfiles = true; -- 默认true，在map-only任务结束时合并小文件

SET hive.merge.mapredfiles = true; -- 默认false，在map-reduce任务结束时合并小文件

SET hive.merge.size.per.task = 268435456; -- 默认256M

SET hive.merge.smallfiles.avgsize = 16777216; -- 当输出文件的平均大小小于16m该值时，启动一个独立的map-reduce任务进行文件merge

**9）开启map端combiner（不影响最终业务逻辑）**

set hive.map.aggr=true；

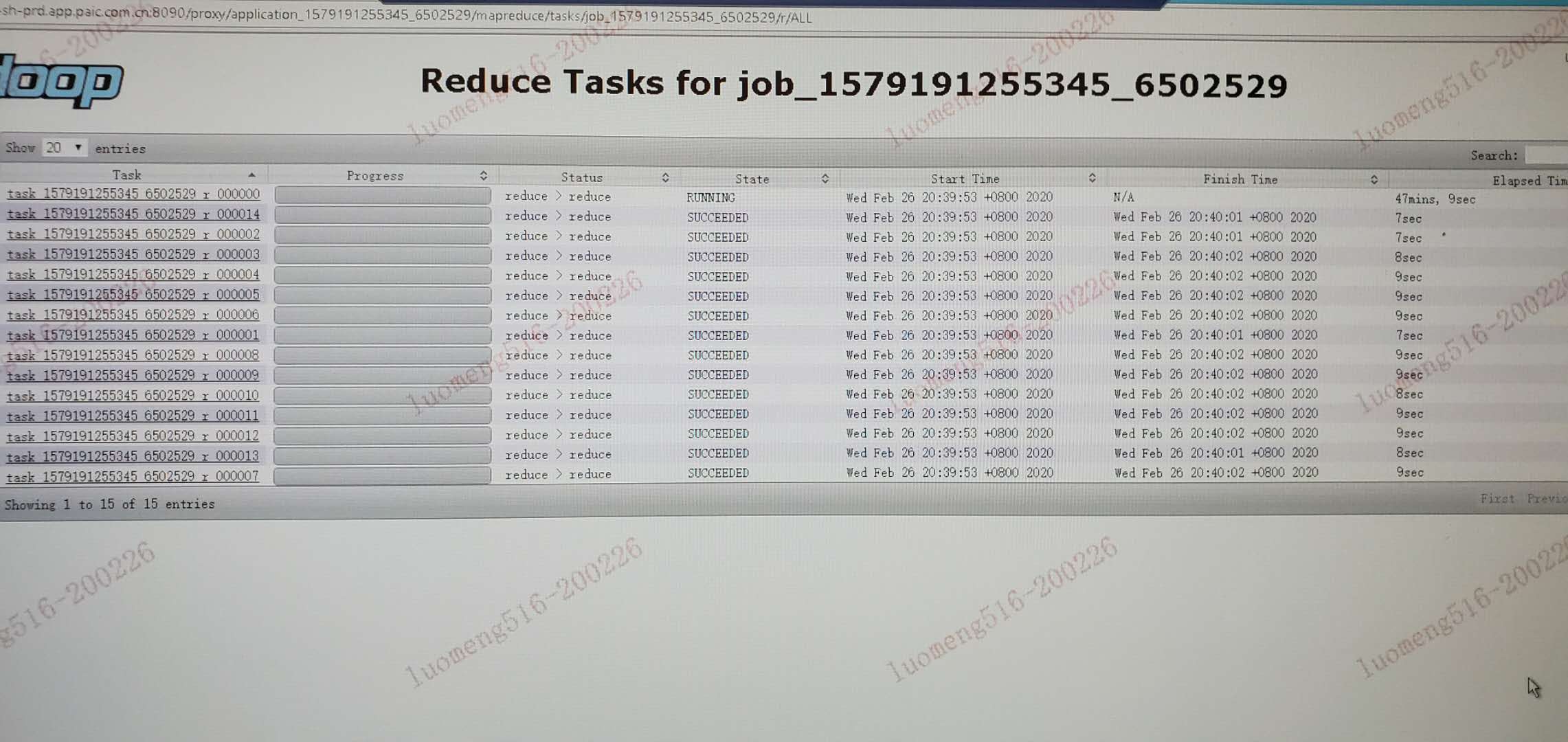
**10）压缩（选择快的）**

设置map端输出、中间结果压缩。（不完全是解决数据倾斜的问题，但是减少了IO读写和网络传输，能提高很多效率）

**11）开启JVM重用**

### 1.6.8 Hive解决数据倾斜方法

**1）数据倾斜长啥样？**



**2）怎么产生的数据倾斜？**

**不同数据类型关联产生数据倾斜**

情形：比如用户表中user\_id字段为int，log表中user\_id字段既有string类型也有int类型。当按照user\_id进行两个表的Join操作时。

后果：处理此特殊值的reduce耗时；只有一个reduce任务  
默认的Hash操作会按int型的id来进行分配，这样会导致所有string类型id的记录都分配到一个Reducer中。

解决方式：把数字类型转换成字符串类型

select \* from users a

left outer join logs b

on a.usr\_id = cast(b.user\_id as string)

**3）解决数据倾斜的方法？**

**（1）group by**

注：group by 优于distinct group

解决方式：采用sum() group by的方式来替换count(distinct)完成计算。

**（2）mapjoin**

**（3）开启数据倾斜时负载均衡**

set hive.groupby.skewindata=true;

思想：就是先随机分发并处理，再按照key group by来分发处理。

操作：当选项设定为true，生成的查询计划会有两个MRJob。

第一个MRJob 中，Map的输出结果集合会随机分布到Reduce中，每个Reduce做部分聚合操作，并输出结果，这样处理的结果是相同的GroupBy Key有可能被分发到不同的Reduce中，从而达到负载均衡的目的；

第二个MRJob再根据预处理的数据结果按照GroupBy Key分布到Reduce中（这个过程可以保证相同的原始GroupBy Key被分布到同一个Reduce中），最后完成最终的聚合操作。

点评：它使计算变成了两个mapreduce，先在第一个中在 shuffle 过程 partition 时随机给 key 打标记，使每个key 随机均匀分布到各个 reduce 上计算，但是这样只能完成部分计算，因为相同key没有分配到相同reduce上。

所以需要第二次的mapreduce,这次就回归正常 shuffle,但是数据分布不均匀的问题在第一次mapreduce已经有了很大的改善，因此基本解决数据倾斜。因为大量计算已经在第一次mr中随机分布到各个节点完成。

**（4）控制空值分布**

将为空的key转变为字符串加随机数或纯随机数，将因空值而造成倾斜的数据分不到多个Reducer。

注：对于异常值如果不需要的话，最好是提前在where条件里过滤掉，这样可以使计算量大大减少

实践中，可以使用case when对空值赋上随机值。此方法比直接写is not null更好，因为前者job数为1，后者为2.

使用case when实例1：

select userid, name from user\_info a

join (  
select case when userid is null then cast (rand(47)\* 100000 as int )  
else userid end from user\_read\_log  
) b on a.userid = b.userid

使用case when实例2：

select '${date}' as thedate,  
 a.search\_type,  
 a.query,  
 a.category,  
 a.cat\_name,  
 a.brand\_id,  
 a.brand\_name,  
 a.dir\_type,  
 a.rewcatid,  
 a.new\_cat\_name,  
 a.new\_brand\_id,  
 f.brand\_name as new\_brand\_name,  
 a.pv,  
 a.uv,  
 a.ipv,  
 a.ipvuv,  
 a.trans\_amt,  
 a.trans\_num,  
 a.alipay\_uv  
from fdi\_search\_query\_cat\_qp\_temp a  
left outer join brand f  
on f.pt='${date}000000' and case when a.new\_brand\_id is null then concat('hive',rand() ) else a.new\_brand\_id end = f.brand\_id;

如果上述的方法还不能解决，比如当有多个JOIN的时候，建议建立临时表，然后拆分HIVE SQL语句。

### 1.6.9 用的是动态分区吗？动态分区的底层原理是什么？

a. 静态分区与动态分区的主要区别在于静态分区是手动指定，而动态分区是通过数据来进行判断。

b. 详细来说，静态分区的列实在编译时期，通过用户传递来决定的；动态分区只有在 SQL 执行时才能决定。

c. 动态分区是基于查询参数的位置去推断分区的名称，从而建立分区

### 26. Hive里边字段的分隔符用的什么？为什么用\t？有遇到过字段里边有\t的情况吗，怎么处理的？

hive 默认的字段分隔符为ascii码的控制符\001（^A）,建表的时候用fields terminated by '\001'

遇到过字段里边有\t的情况，自定义InputFormat，替换为其他分隔符再做后续处理

## 1.7 HBase总结

### 1.7.1 HBase存储结构

略

### 1.7.2 RowKey设计原则

1）rowkey长度原则

2）rowkey散列原则

3）rowkey唯一原则

### 1.7.3 RowKey如何设计

1）生成随机数、hash、散列值

2）字符串反转

### 1.7.4 Phoenix二级索引（讲原理）



## 1.8 Sqoop参数

/opt/module/sqoop/bin/sqoop import \

--connect \

--username \

--password \

--target-dir \

--delete-target-dir \

--num-mappers \

--fields-terminated-by   \

--query   "$2" ' and $CONDITIONS;'

### 1.8.1 Sqoop导入导出Null存储一致性问题

Hive中的Null在底层是以“\N”来存储，而MySQL中的Null在底层就是Null，为了保证数据两端的一致性。在导出数据时采用--input-null-string和--input-null-non-string两个参数。导入数据时采用--null-string和--null-non-string。

### 1.8.2 Sqoop数据导出一致性问题

1）场景1：如Sqoop在导出到Mysql时，使用4个Map任务，过程中有2个任务失败，那此时MySQL中存储了另外两个Map任务导入的数据，此时老板正好看到了这个报表数据。而开发工程师发现任务失败后，会调试问题并最终将全部数据正确的导入MySQL，那后面老板再次看报表数据，发现本次看到的数据与之前的不一致，这在生产环境是不允许的。

官网：http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html

Since Sqoop breaks down export process into multiple transactions, it is possible that a failed export job may result in partial data being committed to the database. This can further lead to subsequent jobs failing due to insert collisions in some cases, or lead to duplicated data in others. You can overcome this problem by specifying a staging table via the --staging-table option which acts as an auxiliary table that is used to stage exported data. The staged data is finally moved to the destination table in a single transaction.

–staging-table方式

sqoop export --connect jdbc:mysql://192.168.137.10:3306/user\_behavior --username root --password 123456 --table app\_cource\_study\_report --columns watch\_video\_cnt,complete\_video\_cnt,dt --fields-terminated-by "\t" --export-dir "/user/hive/warehouse/tmp.db/app\_cource\_study\_analysis\_${day}" --staging-table app\_cource\_study\_report\_tmp --clear-staging-table --input-null-string '\N'

2）场景2：设置map数量为1个（不推荐，面试官想要的答案不只这个）

多个Map任务时，采用–staging-table方式，仍然可以解决数据一致性问题。

### 1.8.3 Sqoop底层运行的任务是什么

只有Map阶段，没有Reduce阶段的任务。

### 1.8.4 Sqoop数据导出的时候一次执行多长时间

Sqoop任务一般情况40 -50分钟的都有。取决于数据量（11：11,6:18）。

### 1.8.5 Sqoop一天导多少数据

业务数据每天1G，就导1G。

### 1.8.6 Sqoop在导入数据的时候数据倾斜

<https://blog.csdn.net/lizhiguo18/article/details/103969906>

sqoop 抽数的并行化主要涉及到两个参数：num-mappers：启动N个map来并行导入数据，默认4个；split-by：按照某一列来切分表的工作单元。

通过ROWNUM() 生成一个严格均匀分布的字段，然后指定为分割字段

## 1.9 Scala

### 1.9.1 开发环境

要求掌握必要的scala开发环境搭建技能。

### 1.9.2 变量和数据类型

掌握var和val的区别

掌握数值类型（Byte、Short、Int、Long、Float、Double、Char）之间的转换关系

### 1.9.3 流程控制

掌握if-else、for、while等必要的流程控制结构，掌握如何实现break、continue的功能。

### 1.9.4 函数式编程

掌握高阶函数、匿名函数、函数柯里化、函数参数以及函数至简原则。

### 1.9.5 面向对象

掌握Scala与Java继承方面的区别、单例对象（伴生对象）、特质的用法及功能。

### 1.9.6 集合

掌握常用集合的使用、集合常用的计算函数。

### 1.9.7 模式匹配

掌握模式匹配的用法

### 1.9.8 异常

掌握异常常用操作即可

### 1.9.9 隐式转换

掌握隐式方法、隐式参数、隐式类，以及隐式解析机制

### 1.9.10 泛型

掌握泛型语法

## 1.10 Spark

### 1.10.1 Spark有几种部署方式？请分别简要论述

1）Local:运行在一台机器上，通常是练手或者测试环境。

2）Standalone:构建一个基于Mster+Slaves的资源调度集群，Spark任务提交给Master运行。是Spark自身的一个调度系统。

3）Yarn: Spark客户端直接连接Yarn，不需要额外构建Spark集群。有yarn-client和yarn-cluster两种模式，主要区别在于：Driver程序的运行节点。

4）Mesos：国内大环境比较少用。

### 1.10.2 Spark任务使用什么进行提交，JavaEE界面还是脚本

Shell 脚本。

### 1.10.3 Spark提交作业参数（重点）

参考答案：

https://blog.csdn.net/gamer\_gyt/article/details/79135118

1）在提交任务时的几个重要参数

executor-cores —— 每个executor使用的内核数，默认为1，官方建议2-5个，我们企业是4个

num-executors —— 启动executors的数量，默认为2

executor-memory —— executor内存大小，默认1G

driver-cores —— driver使用内核数，默认为1

driver-memory —— driver内存大小，默认512M

2）边给一个提交任务的样式

spark-submit \

--master local[5] \

--driver-cores 2 \

--driver-memory 8g \

--executor-cores 4 \

--num-executors 10 \

--executor-memory 8g \

--class PackageName.ClassName XXXX.jar \

--name "Spark Job Name" \

InputPath \

OutputPath

### 1.10.4 简述Spark的架构与作业提交流程（画图讲解，注明各个部分的作用）（重点）

略，参考课程内容作答

### 1.10.5 如何理解Spark中的血统概念（RDD）（笔试重点）

RDD在Lineage依赖方面分为两种Narrow Dependencies与Wide Dependencies用来解决数据容错时的高效性以及划分任务时候起到重要作用。

### 1.10.6 简述Spark的宽窄依赖，以及Spark如何划分stage，每个stage又根据什么决定task个数? （笔试重点）

Stage：根据RDD之间的依赖关系的不同将Job划分成不同的Stage，遇到一个宽依赖则划分一个Stage。

Task：Stage是一个TaskSet，将Stage根据分区数划分成一个个的Task。

### 1.10.7 请列举Spark的transformation算子（不少于8个），并简述功能（重点）

1）map（func）：返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成.

2）mapPartitions(func)：类似于map，但独立地在RDD的每一个分片上运行，因此在类型为T的RD上运行时，func的函数类型必须是Iterator[T] => Iterator[U]。假设有N个元素，有M个分区，那么map的函数的将被调用N次,而mapPartitions被调用M次,一个函数一次处理所有分区。

3）reduceByKey（func，[numTask]）：在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用定的reduce函数，将相同key的值聚合到一起，reduce任务的个数可以通过第二个可选的参数来设置。

4）aggregateByKey (zeroValue:U,[partitioner: Partitioner]) (seqOp: (U, V) => U,combOp: (U, U) => U: 在kv对的RDD中，，按key将value进行分组合并，合并时，将每个value和初始值作为seq函数的参数，进行计算，返回的结果作为一个新的kv对，然后再将结果按照key进行合并，最后将每个分组的value传递给combine函数进行计算（先将前两个value进行计算，将返回结果和下一个value传给combine函数，以此类推），将key与计算结果作为一个新的kv对输出。

5）combineByKey(createCombiner: V=>C, mergeValue: (C, V) =>C, mergeCombiners: (C, C) =>C):

对相同K，把V合并成一个集合。

1.createCombiner: combineByKey() 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。如果这是一个新的元素,combineByKey()会使用一个叫作createCombiner()的函数来创建那个键对应的累加器的初始值

2.mergeValue: 如果这是一个在处理当前分区之前已经遇到的键，它会使用mergeValue()方法将该键的累加器对应的当前值与这个新的值进行合并

3.mergeCombiners: 由于每个分区都是独立处理的， 因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器， 就需要使用用户提供的 mergeCombiners() 方法将各个分区的结果进行合并。

…

根据自身情况选择比较熟悉的算子加以介绍。

### 1.10.8 请列举Spark的action算子（不少于6个），并简述功能（重点）

1）reduce：

2）collect:

3）first：

4）take：

5）aggregate：

6）countByKey：

7）foreach：

8）saveAsTextFile：

### 1.10.9 请列举会引起Shuffle过程的Spark算子，并简述功能。

reduceBykey：

groupByKey：

…ByKey:

### 1.10.10 简述Spark的两种核心Shuffle（HashShuffle与SortShuffle）的工作流程（包括未优化的HashShuffle、优化的HashShuffle、普通的SortShuffle与bypass的SortShuffle）（重点）

参考课程内容作答

### 1.10.11 Spark常用算子reduceByKey与groupByKey的区别，哪一种更具优势？（重点）

reduceByKey：按照key进行聚合，在shuffle之前有combine（预聚合）操作，返回结果是RDD[k,v]。

groupByKey：按照key进行分组，直接进行shuffle。

开发指导：reduceByKey比groupByKey，建议使用。但是需要注意是否会影响业务逻辑。

### 1.10.12 Repartition和Coalesce关系与区别

1）关系：

两者都是用来改变RDD的partition数量的，repartition底层调用的就是coalesce方法：coalesce(numPartitions, shuffle = true)

2）区别：

repartition一定会发生shuffle，coalesce根据传入的参数来判断是否发生shuffle

一般情况下增大rdd的partition数量使用repartition，减少partition数量时使用coalesce

### 1.10.13 分别简述Spark中的缓存机制（cache和persist）与checkpoint机制，并指出两者的区别与联系

都是做RDD持久化的

cache:内存，不会截断血缘关系，使用计算过程中的数据缓存。

checkpoint：磁盘，截断血缘关系，在ck之前必须没有任何任务提交才会生效，ck过程会额外提交一次任务。

### 1.10.14 简述Spark中共享变量（广播变量和累加器）的基本原理与用途。（重点）

累加器（accumulator）是Spark中提供的一种分布式的变量机制，其原理类似于mapreduce，即分布式的改变，然后聚合这些改变。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。而广播变量用来高效分发较大的对象。

共享变量出现的原因：

通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。

Spark的两个共享变量，累加器与广播变量，分别为结果聚合与广播这两种常见的通信模式突破了这一限制。

### 1.10.15 当Spark涉及到数据库的操作时，如何减少Spark运行中的数据库连接数？

使用foreachPartition代替foreach，在foreachPartition内获取数据库的连接。

### 1.10.16 如何使用Spark实现TopN的获取（描述思路或使用伪代码）（重点）

方法1：

（1）按照key对数据进行聚合（groupByKey）

（2）将value转换为数组，利用scala的sortBy或者sortWith进行排序（mapValues）数据量太大，会OOM。

方法2：

（1）取出所有的key

（2）对key进行迭代，每次取出一个key利用spark的排序算子进行排序

方法3：

（1）自定义分区器，按照key进行分区，使不同的key进到不同的分区

（2）对每个分区运用spark的排序算子进行排序

### 1.10.17 京东：调优之前与调优之后性能的详细对比（例如调整map个数，map个数之前多少、之后多少，有什么提升）

这里举个例子。比如我们有几百个文件，会有几百个map出现，读取之后进行join操作，会非常的慢。这个时候我们可以进行coalesce操作，比如240个map，我们合成60个map，也就是窄依赖。这样再shuffle，过程产生的文件数会大大减少。提高join的时间性能。

## 1.11 Spark Sql、DataFrames、DataSet

### 1.11.1 简述SparkSQL中RDD、DataFrame、DataSet三者的区别与联系? （笔试重点）

**1）RDD**

优点:

编译时类型安全

编译时就能检查出类型错误

面向对象的编程风格

直接通过类名点的方式来操作数据

缺点:

序列化和反序列化的性能开销

无论是集群间的通信, 还是IO操作都需要对对象的结构和数据进行序列化和反序列化。

GC的性能开销，频繁的创建和销毁对象, 势必会增加GC

**2）DataFrame**

DataFrame引入了schema和off-heap

schema : RDD每一行的数据, 结构都是一样的，这个结构就存储在schema中。 Spark通过schema就能够读懂数据, 因此在通信和IO时就只需要序列化和反序列化数据, 而结构的部分就可以省略了。

**3）DataSet**

DataSet结合了RDD和DataFrame的优点，并带来的一个新的概念Encoder。

当序列化数据时，Encoder产生字节码与off-heap进行交互，能够达到按需访问数据的效果，而不用反序列化整个对象。Spark还没有提供自定义Encoder的API，但是未来会加入。

三者之间的转换：

参考课程内容作答

### 1.11.2 append和overwrite的区别

append在原有分区上进行追加，overwrite在原有分区上进行全量刷新

### 1.11.3 coalesce和repartition的区别

coalesce和repartition都用于改变分区，coalesce用于缩小分区且不会进行shuffle，repartition用于增大分区（提供并行度）会进行shuffle,在spark中减少文件个数会使用coalesce来减少分区来到这个目的。但是如果数据量过大，分区数过少会出现OOM所以coalesce缩小分区个数也需合理

### 1.11.4 cache缓存级别

DataFrame的cache默认采用 MEMORY\_AND\_DISK 这和RDD 的默认方式不一样RDD cache 默认采用MEMORY\_ONLY

### 1.11.5 释放缓存和缓存

缓存：(1)dataFrame.cache (2)sparkSession.catalog.cacheTable(“tableName”)

释放缓存：(1)dataFrame.unpersist (2)sparkSession.catalog.uncacheTable(“tableName”)

### 1.11.6 Spark Shuffle默认并行度

参数spark.sql.shuffle.partitions 决定 默认并行度200

### 1.11.7 kryo序列化

kryo序列化比java序列化更快更紧凑，但spark默认的序列化是java序列化并不是spark序列化，因为spark并不支持所有序列化类型，而且每次使用都必须进行注册。注册只针对于RDD。在DataFrames和DataSet当中自动实现了kryo序列化。

### 1.11.8 创建临时表和全局临时表

DataFrame.createTempView() 创建普通临时表

DataFrame.createGlobalTempView() DataFrame.createOrReplaceTempView() 创建全局临时表

### 1.11.9 BroadCast join 广播join

原理：先将小表数据查询出来聚合到driver端，再广播到各个executor端，使表与表join时

进行本地join，避免进行网络传输产生shuffle。

使用场景：大表join小表 只能广播小表

### 1.11.10 控制Spark reduce缓存 调优shuffle

spark.reducer.maxSizeInFilght 此参数为reduce task能够拉取多少数据量的一个参数默认48MB，当集群资源足够时，增大此参数可减少reduce拉取数据量的次数，从而达到优化shuffle的效果，一般调大为96MB,资源够大可继续往上跳。

spark.shuffle.file.buffer 此参数为每个shuffle文件输出流的内存缓冲区大小，调大此参数可以减少在创建shuffle文件时进行磁盘搜索和系统调用的次数，默认参数为32k 一般调大为64k。

### 1.11.11 注册UDF函数

SparkSession.udf.register 方法进行注册

### 1.11.12 SparkSQL中join操作与left join操作的区别？

join和sql中的inner join操作很相似，返回结果是前面一个集合和后面一个集合中匹配成功的，过滤掉关联不上的。

leftJoin类似于SQL中的左外关联left outer join，返回结果以第一个RDD为主，关联不上的记录为空。

部分场景下可以使用left semi join替代left join：

因为 left semi join 是 in(keySet) 的关系，**遇到右表重复记录，左表会跳过,性能更高**，而 left join 则会一直遍历。但是left semi join 中最后 select 的结果中只许出现左表中的列名，因为右表只有 join key 参与关联计算了

## 1.12 Spark Streaming

### 1.12.1 Spark Streaming第一次运行不丢失数据

kafka参数 auto.offset.reset 参数设置成earliest 从最初始偏移量开始消费数据

### 1.12.2 Spark Streaming精准一次消费

1. 手动维护偏移量
2. 处理完业务数据后，再进行提交偏移量操作

极端情况下，如在提交偏移量时断网或停电会造成spark程序第二次启动时重复消费问题，所以在涉及到金额或精确性非常高的场景会使用事物保证精准一次消费

### 1.12.3 Spark Streaming控制每秒消费数据的速度

通过spark.streaming.kafka.maxRatePerPartition参数来设置Spark Streaming从kafka分区每秒拉取的条数

### 1.12.4 Spark Streaming背压机制

把spark.streaming.backpressure.enabled 参数设置为ture,开启背压机制后Spark Streaming会根据延迟动态去kafka消费数据,上限由spark.streaming.kafka.maxRatePerPartition参数控制，所以两个参数一般会一起使用

### 1.12.5 Spark Streaming 一个stage耗时

Spark Streaming stage耗时由最慢的task决定,所以数据倾斜时某个task运行慢会导致整个Spark Streaming都运行非常慢。

### 1.12.6 Spark Streaming 优雅关闭

把spark.streaming.stopGracefullyOnShutdown参数设置成ture,Spark会在JVM关闭时正常关闭StreamingContext,而不是立马关闭

Kill 命令：yarn application -kill 后面跟 applicationid

### 1.12.7 Spark Streaming 默认分区个数

Spark Streaming默认分区个数与所对接的kafka topic分区个数一致，Spark Streaming里一般不会使用repartition算子增大分区，因为repartition会进行shuffle增加耗时

### 1.12.8 SparkStreaming有哪几种方式消费Kafka中的数据，它们之间的区别是什么？

**一、基于Receiver的方式**

这种方式使用Receiver来获取数据。Receiver是使用Kafka的高层次Consumer API来实现的。receiver从Kafka中获取的数据都是存储在Spark Executor的内存中的（如果突然数据暴增，大量batch堆积，很容易出现内存溢出的问题），然后Spark Streaming启动的job会去处理那些数据。

然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用Spark Streaming的预写日志机制（Write Ahead Log，WAL）。该机制会同步地将接收到的Kafka数据写入分布式文件系统（比如HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

**二、基于Direct的方式**

这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。

**优点如下：**

**简化并行读取：**如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。

**高性能：**如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复。

**一次且仅一次的事务机制**。

**三、对比：**

基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。

基于direct的方式，使用kafka的简单api，Spark Streaming自己就负责追踪消费的offset，并保存在checkpoint中。Spark自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

在实际生产环境中大都用Direct方式

### 1.12.9 简述SparkStreaming窗口函数的原理（重点）

窗口函数就是在原来定义的SparkStreaming计算批次大小的基础上再次进行封装，每次计算多个批次的数据，同时还需要传递一个滑动步长的参数，用来设置当次计算任务完成之后下一次从什么地方开始计算。

图中time1就是SparkStreaming计算批次大小，虚线框以及实线大框就是窗口的大小，必须为批次的整数倍。虚线框到大实线框的距离（相隔多少批次），就是滑动步长。

## 1.13 数据倾斜

公司一：总用户量1000万，5台64G内存的服务器。

公司二：总用户量10亿，1000台64G内存的服务器。

1.公司一的数据分析师在做join的时候发生了数据倾斜，会导致有几百万用户的相关数据集中到了一台服务器上，几百万的用户数据，说大也不大，正常字段量的数据的话64G还是能轻松处理掉的。

2.公司二的数据分析师在做join的时候也发生了数据倾斜，可能会有1个亿的用户相关数据集中到了一台机器上了（相信我，这很常见）。这时候一台机器就很难搞定了，最后会很难算出结果。

### 1.13.1 数据倾斜表现

1）hadoop中的数据倾斜表现：

* 有一个多几个Reduce卡住，卡在99.99%，一直不能结束。
* 各种container报错OOM
* 异常的Reducer读写的数据量极大，至少远远超过其它正常的Reducer
* 伴随着数据倾斜，会出现任务被kill等各种诡异的表现。

2）hive中数据倾斜

一般都发生在Sql中group by和join on上，而且和数据逻辑绑定比较深。

3）Spark中的数据倾斜

Spark中的数据倾斜，包括Spark Streaming和Spark Sql，表现主要有下面几种：

* Executor lost，OOM，Shuffle过程出错；
* Driver OOM；
* 单个Executor执行时间特别久，整体任务卡在某个阶段不能结束；
* 正常运行的任务突然失败；

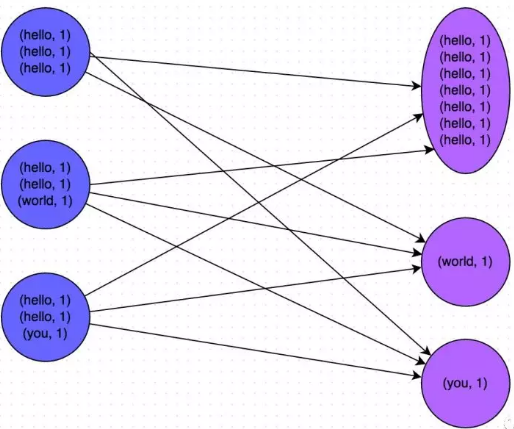
### 1.13.2 数据倾斜产生原因

我们以Spark和Hive的使用场景为例。

他们在做数据运算的时候会涉及到，count distinct、group by、join on等操作，这些都会触发Shuffle动作。一旦触发Shuffle，所有相同key的值就会被拉到一个或几个Reducer节点上，容易发生单点计算问题，导致数据倾斜。

一般来说，数据倾斜原因有以下几方面：

**1）key分布不均匀；**



**2）建表时考虑不周**

我们举一个例子，就说数据默认值的设计吧，假设我们有两张表：

user（用户信息表）：userid，register\_ip

ip（IP表）：ip，register\_user\_cnt

这可能是两个不同的人开发的数据表。如果我们的数据规范不太完善的话，会出现一种情况：

user表中的register\_ip字段，如果获取不到这个信息，我们默认为null；

但是在ip表中，我们在统计这个值的时候，为了方便，我们把获取不到ip的用户，统一认为他们的ip为0。

两边其实都没有错的，但是一旦我们做关联了，这个任务会在做关联的阶段，也就是sql的on的阶段卡死。

**3）业务数据激增**

比如订单场景，我们在某一天在北京和上海两个城市多了强力的推广，结果可能是这两个城市的订单量增长了10000%，其余城市的数据量不变。

然后我们要统计不同城市的订单情况，这样，一做group操作，可能直接就数据倾斜了。

### 1.13.3 解决数据倾斜思路

很多数据倾斜的问题，都可以用和平台无关的方式解决，比如更好的**数据预处理**，**异常值的过滤**等。因此，解决数据倾斜的重点在于对数据设计和业务的理解，这两个搞清楚了，数据倾斜就解决了大部分了。

**1）业务逻辑**

我们从业务逻辑的层面上来优化数据倾斜，比如上面的两个城市做推广活动导致那两个城市数据量激增的例子，我们可以单独对这两个城市来做count，单独做时可用两次MR，第一次打散计算，第二次再最终聚合计算。完成后和其它城市做整合。

**2）程序层面**

比如说在Hive中，经常遇到count(distinct)操作，这样会导致最终只有一个Reduce任务。

我们可以先group by，再在外面包一层count，就可以了。比如计算按用户名去重后的总用户量：

（1）优化前 只有一个reduce，先去重再count负担比较大：

select name,count(distinct name)from user;

（2）优化后

// 设置该任务的每个job的reducer个数为3个。Hive默认-1，自动推断。

set mapred.reduce.tasks=3;

// 启动两个job，一个负责子查询(可以有多个reduce)，另一个负责count(1)：

select count(1) from (select name from user group by name) tmp;

**3）调参方面**

Hadoop和Spark都自带了很多的参数和机制来调节数据倾斜，合理利用它们就能解决大部分问题。

**4）从业务和数据上解决数据倾斜**

很多数据倾斜都是在数据的使用上造成的。我们举几个场景，并分别给出它们的解决方案。

* 有损的方法：找到异常数据，比如ip为0的数据，过滤掉
* 无损的方法：对分布不均匀的数据，单独计算
* 先对key做一层hash，先将数据随机打散让它的并行度变大，再汇集
* 数据预处理

### 1.13.4 定位导致数据倾斜代码

Spark数据倾斜只会发生在shuffle过程中。

这里给大家罗列一些常用的并且可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。

出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

#### 1.13.4.1 某个task执行特别慢的情况

首先要看的，就是数据倾斜发生在第几个stage中：

如果是用yarn-client模式提交，那么在提交的机器本地是直接可以看到log，可以在log中找到当前运行到了第几个stage；

如果是用yarn-cluster模式提交，则可以通过Spark Web UI来查看当前运行到了第几个stage。

此外，无论是使用yarn-client模式还是yarn-cluster模式，我们都可以在Spark Web UI上深入看一下当前这个stage各个task分配的数据量，从而进一步确定是不是task分配的数据不均匀导致了数据倾斜。

看task运行时间和数据量

task运行时间

比如下图中，倒数第三列显示了每个task的运行时间。明显可以看到，有的task运行特别快，只需要几秒钟就可以运行完；而有的task运行特别慢，需要几分钟才能运行完，此时单从运行时间上看就已经能够确定发生数据倾斜了。

task数据量

此外，倒数第一列显示了每个task处理的数据量，明显可以看到，运行时间特别短的task只需要处理几百KB的数据即可，而运行时间特别长的task需要处理几千KB的数据，处理的数据量差了10倍。此时更加能够确定是发生了数据倾斜。

推断倾斜代码

知道数据倾斜发生在哪一个stage之后，接着我们就需要根据stage划分原理，推算出来发生倾斜的那个stage对应代码中的哪一部分，这部分代码中肯定会有一个shuffle类算子。

精准推算stage与代码的对应关系，需要对Spark的源码有深入的理解，这里我们可以介绍一个相对简单实用的推算方法：只要看到Spark代码中出现了一个shuffle类算子或者是Spark SQL的SQL语句中出现了会导致shuffle的语句（比如group by语句），那么就可以判定，以那个地方为界限划分出了前后两个stage。

这里我们就以如下单词计数来举例。

val conf = new SparkConf()val sc = new SparkContext(conf)val lines = sc.textFile("hdfs://...")val words = lines.flatMap(\_.split(" "))val pairs = words.map((\_, 1))val wordCounts = pairs.reduceByKey(\_ + \_)wordCounts.collect().foreach(println(\_))

在整个代码中只有一个reduceByKey是会发生shuffle的算子，也就是说这个算子为界限划分出了前后两个stage：

stage0，主要是执行从textFile到map操作，以及shuffle write操作（对pairs RDD中的数据进行分区操作，每个task处理的数据中，相同的key会写入同一个磁盘文件内）。

stage1，主要是执行从reduceByKey到collect操作，以及stage1的各个task一开始运行，就会首先执行shuffle read操作（会从stage0的各个task所在节点拉取属于自己处理的那些key，然后对同一个key进行全局性的聚合或join等操作，在这里就是对key的value值进行累加）

stage1在执行完reduceByKey算子之后，就计算出了最终的wordCounts RDD，然后会执行collect算子，将所有数据拉取到Driver上，供我们遍历和打印输出。

123456789

通过对单词计数程序的分析，希望能够让大家了解最基本的stage划分的原理，以及stage划分后shuffle操作是如何在两个stage的边界处执行的。然后我们就知道如何快速定位出发生数据倾斜的stage对应代码的哪一个部分了。

比如我们在Spark Web UI或者本地log中发现，stage1的某几个task执行得特别慢，判定stage1出现了数据倾斜，那么就可以回到代码中，定位出stage1主要包括了reduceByKey这个shuffle类算子，此时基本就可以确定是是该算子导致了数据倾斜问题。

此时，如果某个单词出现了100万次，其他单词才出现10次，那么stage1的某个task就要处理100万数据，整个stage的速度就会被这个task拖慢。

#### 1.13.4.2 某个task莫名其妙内存溢出的情况

这种情况下去定位出问题的代码就比较容易了。我们建议直接看yarn-client模式下本地log的异常栈，或者是通过YARN查看yarn-cluster模式下的log中的异常栈。一般来说，通过异常栈信息就可以定位到你的代码中哪一行发生了内存溢出。然后在那行代码附近找找，一般也会有shuffle类算子，此时很可能就是这个算子导致了数据倾斜。

但是大家要注意的是，不能单纯靠偶然的内存溢出就判定发生了数据倾斜。因为自己编写的代码的bug，以及偶然出现的数据异常，也可能会导致内存溢出。因此还是要按照上面所讲的方法，通过Spark Web UI查看报错的那个stage的各个task的运行时间以及分配的数据量，才能确定是否是由于数据倾斜才导致了这次内存溢出。

### 1.13.5 查看导致数据倾斜的key分布情况

先对pairs采样10%的样本数据，然后使用countByKey算子统计出每个key出现的次数，最后在客户端遍历和打印样本数据中各个key的出现次数。

val sampledPairs = pairs.sample(false, 0.1)

val sampledWordCounts = sampledPairs.countByKey()

sampledWordCounts.foreach(println(\_))

### 1.13.6 Spark 数据倾斜的解决方案

#### 1.13.6.1 使用Hive ETL预处理数据

##### 1.13.6.1.1 适用场景

导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且业务场景需要频繁使用Spark对Hive表执行某个分析操作，那么比较适合使用这种技术方案。

##### 1.13.6.1.2 实现思路

此时可以评估一下，是否可以通过Hive来进行数据预处理（即通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join），然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。

##### 1.13.6.1.3 方案实现原理

这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

##### 1.13.6.1.4 方案优缺点

优点：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

缺点：治标不治本，Hive ETL中还是会发生数据倾斜。

##### 1.13.6.1.5 方案实践经验

在一些Java系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验。

##### 1.13.6.1.6 项目实践经验

在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过Java Web系统提交数据分析统计任务，后端通过Java提交Spark作业进行数据分析统计。要求Spark作业速度必须要快，尽量在10分钟以内，否则速度太慢，用户体验会很差。所以我们将有些Spark作业的shuffle操作提前到了Hive ETL中，从而让Spark直接使用预处理的Hive中间表，尽可能地减少Spark的shuffle操作，大幅度提升了性能，将部分作业的性能提升了6倍以上。

#### 1.13.6.2 过滤少数导致倾斜的key

##### 1.13.6.2.1 方案适用场景

如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜。

##### 1.13.6.2.2 方案实现思路

如果我们判断那少数几个数据量特别多的key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个key。

比如，在Spark SQL中可以使用where子句过滤掉这些key或者在Spark Core中对RDD执行filter算子过滤掉这些key。

如果需要每次作业执行时，动态判定哪些key的数据量最多然后再进行过滤，那么可以使用sample算子对RDD进行采样，然后计算出每个key的数量，取数据量最多的key过滤掉即可。

##### 1.13.6.2.3 方案实现原理

将导致数据倾斜的key给过滤掉之后，这些key就不会参与计算了，自然不可能产生数据倾斜。

##### 1.13.6.2.4 方案优缺点

优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

缺点：适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。

##### 1.13.6.2.5 方案实践经验

在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉。

#### 1.13.6.3 提高shuffle操作的并行度

##### 1.13.6.3.1 方案适用场景

如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。

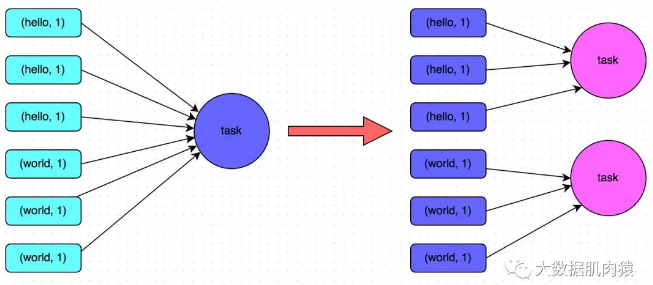
##### 1.13.6.3.2 方案实现思路

在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，默认是200，对于很多场景来说都有点过小。

##### 1.13.6.3.3 方案实现原理

增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。

而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。具体原理如下图所示。



##### 1.13.6.3.4 方案优缺点

优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

缺点：只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

##### 1.13.6.3.5 方案实践经验

该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用最简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

#### 1.13.6.4 两阶段聚合（局部聚合+全局聚合）

##### 1.13.6.4.1 方案适用场景

对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案。

##### 1.13.6.4.2 方案实现思路

这个方案的核心实现思路就是进行两阶段聚合：

第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1\_hello, 1) (1\_hello, 1) (2\_hello, 1) (2\_hello, 1)。

接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1\_hello, 2) (2\_hello, 2)。

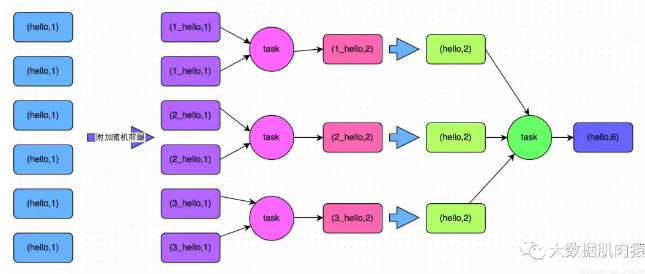
然后将各个key的前缀给去掉，就会变成(hello,2)(hello,2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

示例代码如下：

// 第一步，给RDD中的每个key都打上一个随机前缀。  
JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair(  
 new PairFunction<Tuple2<Long,Long>, String, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<String, Long> call(Tuple2<Long, Long> tuple)  
 throws Exception {  
 Random random = new Random();  
 int prefix = random.nextInt(10);  
 return new Tuple2<String, Long>(prefix + "\_" + tuple.\_1, tuple.\_2);  
 }  
 });  
   
// 第二步，对打上随机前缀的key进行局部聚合。  
JavaPairRDD<String, Long> localAggrRdd = randomPrefixRdd.reduceByKey(  
 new Function2<Long, Long, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Long call(Long v1, Long v2) throws Exception {  
 return v1 + v2;  
 }  
 });  
   
// 第三步，去除RDD中每个key的随机前缀。  
JavaPairRDD<Long, Long> removedRandomPrefixRdd = localAggrRdd.mapToPair(  
 new PairFunction<Tuple2<String,Long>, Long, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<Long, Long> call(Tuple2<String, Long> tuple)  
 throws Exception {  
 long originalKey = Long.valueOf(tuple.\_1.split("\_")[1]);  
 return new Tuple2<Long, Long>(originalKey, tuple.\_2);  
 }  
 });  
   
// 第四步，对去除了随机前缀的RDD进行全局聚合。  
JavaPairRDD<Long, Long> globalAggrRdd = removedRandomPrefixRdd.reduceByKey(  
 new Function2<Long, Long, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Long call(Long v1, Long v2) throws Exception {  
 return v1 + v2;  
 }  
 });

##### 1.13.6.4.3 方案实现原理

将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。



##### 1.13.6.4.4 方案优缺点

优点  
对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上。

缺点  
仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案。

#### 1.13.6.5 将reduce join转为map join

##### 1.13.6.5.1 方案适用场景

在对RDD使用join类操作，或者是在Spark SQL中使用join语句时，而且join操作中的一个RDD或表的数据量比较小（比如几百M或者一两G），比较适用此方案。

##### 1.13.6.5.2 方案实现思路

不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量，广播给其他Executor节点；

接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

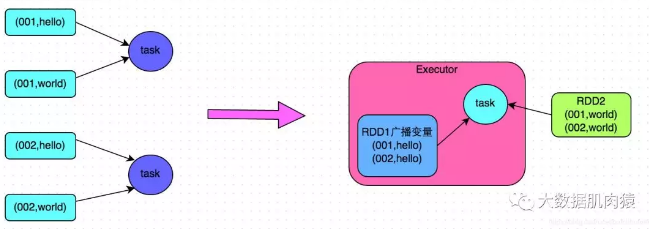
示例如下：

// 首先将数据量比较小的RDD的数据，collect到Driver中来。  
List<Tuple2<Long, Row>> rdd1Data = rdd1.collect()  
// 然后使用Spark的广播功能，将小RDD的数据转换成广播变量，这样每个Executor就只有一份RDD的数据。  
// 可以尽可能节省内存空间，并且减少网络传输性能开销。  
final Broadcast<List<Tuple2<Long, Row>>> rdd1DataBroadcast = sc.broadcast(rdd1Data);  
   
// 对另外一个RDD执行map类操作，而不再是join类操作。  
JavaPairRDD<String, Tuple2<String, Row>> joinedRdd = rdd2.mapToPair(  
 new PairFunction<Tuple2<Long,String>, String, Tuple2<String, Row>>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<String, Tuple2<String, Row>> call(Tuple2<Long, String> tuple)  
 throws Exception {  
 // 在算子函数中，通过广播变量，获取到本地Executor中的rdd1数据。  
 List<Tuple2<Long, Row>> rdd1Data = rdd1DataBroadcast.value();  
 // 可以将rdd1的数据转换为一个Map，便于后面进行join操作。  
 Map<Long, Row> rdd1DataMap = new HashMap<Long, Row>();  
 for(Tuple2<Long, Row> data : rdd1Data) {  
 rdd1DataMap.put(data.\_1, data.\_2);  
 }  
 // 获取当前RDD数据的key以及value。  
 String key = tuple.\_1;  
 String value = tuple.\_2;  
 // 从rdd1数据Map中，根据key获取到可以join到的数据。  
 Row rdd1Value = rdd1DataMap.get(key);  
 return new Tuple2<String, String>(key, new Tuple2<String, Row>(value, rdd1Value));  
 }  
 });  
   
// 这里得提示一下。  
// 上面的做法，仅仅适用于rdd1中的key没有重复，全部是唯一的场景。  
// 如果rdd1中有多个相同的key，那么就得用flatMap类的操作，在进行join的时候不能用map，而是得遍历rdd1所有数据进行join。  
// rdd2中每条数据都可能会返回多条join后的数据。

##### 1.13.6.5.3 方案实现原理

普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。

但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜。具体原理如下图所示。



##### 1.13.6.5.4 方案优缺点

优点：对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜。

缺点：适用场景较少，因为这个方案只适用于一个大表和一个小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver和每个Executor内存中都会驻留一份小RDD的全量数据。如果我们广播出去的RDD数据比较大，比如10G以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。

#### 1.13.6.6 采样倾斜key并分拆join操作

##### 1.13.6.6.1 方案适用场景

两个RDD/Hive表进行join的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个RDD/Hive表中的key分布情况。

如果出现数据倾斜，是因为其中某一个RDD/Hive表中的少数几个key的数据量过大，而另一个RDD/Hive表中的所有key都分布比较均匀，那么采用这个解决方案是比较合适的。

##### 1.13.6.6.2 方案实现思路

对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key。

然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀；

而不会导致倾斜的大部分key形成另外一个RDD。

接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个单独的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀；

不会导致倾斜的大部分key也形成另外一个RDD。

再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了。

而另外两个普通的RDD就照常join即可。

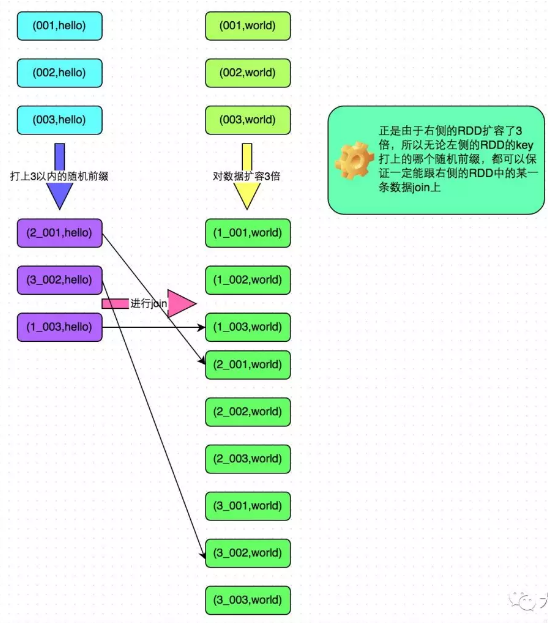
最后将两次join的结果使用union算子合并起来即可，就是最终的join结果。

示例如下：

// 首先从包含了少数几个导致数据倾斜key的rdd1中，采样10%的样本数据。  
JavaPairRDD<Long, String> sampledRDD = rdd1.sample(false, 0.1);  
   
// 对样本数据RDD统计出每个key的出现次数，并按出现次数降序排序。  
// 对降序排序后的数据，取出top 1或者top 100的数据，也就是key最多的前n个数据。  
// 具体取出多少个数据量最多的key，由大家自己决定，我们这里就取1个作为示范。  
  
// 每行数据变为<key,1>  
JavaPairRDD<Long, Long> mappedSampledRDD = sampledRDD.mapToPair(  
 new PairFunction<Tuple2<Long,String>, Long, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<Long, Long> call(Tuple2<Long, String> tuple)  
 throws Exception {  
 return new Tuple2<Long, Long>(tuple.\_1, 1L);  
 }  
 });  
   
// 按key累加行数  
JavaPairRDD<Long, Long> countedSampledRDD = mappedSampledRDD.reduceByKey(  
 new Function2<Long, Long, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Long call(Long v1, Long v2) throws Exception {  
 return v1 + v2;  
 }  
 });  
   
// 反转key和value,变为<value,key>  
JavaPairRDD<Long, Long> reversedSampledRDD = countedSampledRDD.mapToPair(  
 new PairFunction<Tuple2<Long,Long>, Long, Long>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<Long, Long> call(Tuple2<Long, Long> tuple)  
 throws Exception {  
 return new Tuple2<Long, Long>(tuple.\_2, tuple.\_1);  
 }  
 });  
  
// 以行数排序key，取最多行数的key  
final Long skewedUserid = reversedSampledRDD.sortByKey(false).take(1).get(0).\_2;  
   
// 从rdd1中分拆出导致数据倾斜的key，形成独立的RDD。  
JavaPairRDD<Long, String> skewedRDD = rdd1.filter(  
 new Function<Tuple2<Long,String>, Boolean>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Boolean call(Tuple2<Long, String> tuple) throws Exception {  
 return tuple.\_1.equals(skewedUserid);  
 }  
 });  
   
// 从rdd1中分拆出不导致数据倾斜的普通key，形成独立的RDD。  
JavaPairRDD<Long, String> commonRDD = rdd1.filter(  
 new Function<Tuple2<Long,String>, Boolean>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Boolean call(Tuple2<Long, String> tuple) throws Exception {  
 return !tuple.\_1.equals(skewedUserid);  
 }  
 });  
   
// rdd2，就是那个所有key的分布相对较为均匀的rdd。  
// 这里将rdd2中，前面获取到的key对应的数据，过滤出来，分拆成单独的rdd，并对rdd中的数据使用flatMap算子都扩容100倍。  
// 对扩容的每条数据，都打上0～100的前缀。  
JavaPairRDD<String, Row> skewedRdd2 = rdd2.filter(  
 new Function<Tuple2<Long,Row>, Boolean>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Boolean call(Tuple2<Long, Row> tuple) throws Exception {  
 return tuple.\_1.equals(skewedUserid);  
 }  
 }).flatMapToPair(new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Iterable<Tuple2<String, Row>> call(  
 Tuple2<Long, Row> tuple) throws Exception {  
 Random random = new Random();  
 List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();  
 for(int i = 0; i < 100; i++) {  
 list.add(new Tuple2<String, Row>(i + "\_" + tuple.\_1, tuple.\_2));  
 }  
 return list;  
 }  
   
 });  
   
// 将rdd1中分拆出来的导致倾斜的key的独立rdd，每条数据都打上100以内的随机前缀。  
// 然后将这个rdd1中分拆出来的独立rdd，与上面rdd2中分拆出来的独立rdd，进行join。  
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 = skewedRDD.mapToPair(  
 new PairFunction<Tuple2<Long,String>, String, String>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<String, String> call(Tuple2<Long, String> tuple)  
 throws Exception {  
 Random random = new Random();  
 int prefix = random.nextInt(100);  
 return new Tuple2<String, String>(prefix + "\_" + tuple.\_1, tuple.\_2);  
 }  
 })  
 .join(skewedUserid2infoRDD)  
 .mapToPair(new PairFunction<Tuple2<String,Tuple2<String,Row>>, Long, Tuple2<String, Row>>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<Long, Tuple2<String, Row>> call(  
 Tuple2<String, Tuple2<String, Row>> tuple)  
 throws Exception {  
 long key = Long.valueOf(tuple.\_1.split("\_")[1]);  
 return new Tuple2<Long, Tuple2<String, Row>>(key, tuple.\_2);  
 }  
 });  
   
// 将rdd1中分拆出来的包含普通key的独立rdd，直接与rdd2进行join。  
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(rdd2);  
   
// 将倾斜key join后的结果与普通key join后的结果，uinon起来。  
// 就是最终的join结果。  
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);

##### 1.13.6.6.3 方案实现原理

对于join导致的数据倾斜，如果只是某几个key导致了倾斜，可以将少数几个key分拆成独立RDD，并附加随机前缀打散成n份去进行join，此时这几个key对应的数据就不会集中在少数几个task上，而是分散到多个task进行join了。具体原理见下图。



##### 1.13.6.6.4 方案优缺点

优点：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

缺点：如果导致倾斜的key特别多的话，比如成千上万个key都导致数据倾斜，那么这种方式也不适合。

#### 1.13.6.7 使用随机前缀和扩容RDD进行join

##### 1.13.6.7.1 方案适用场景

如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义，此时就只能使用最后一种方案来解决问题了。

##### 1.13.6.7.2 方案实现思路

该方案的实现思路基本和“解决方案六”类似，首先查看RDD/Hive表中的数据分布情况，找到那个造成数据倾斜的RDD/Hive表，比如有多个key都对应了超过1万条数据。

然后将该RDD的每条数据都打上一个n以内的随机前缀。

同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。

最后将两个处理后的RDD进行join即可。

示例代码如下：

// 首先将其中一个key分布相对较为均匀的RDD膨胀100倍。  
JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair(  
 new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Iterable<Tuple2<String, Row>> call(Tuple2<Long, Row> tuple)  
 throws Exception {  
 List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();  
 for(int i = 0; i < 100; i++) {  
 list.add(new Tuple2<String, Row>(0 + "\_" + tuple.\_1, tuple.\_2));  
 }  
 return list;  
 }  
 });  
   
// 其次，将另一个有数据倾斜key的RDD，每条数据都打上100以内的随机前缀。  
JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair(  
 new PairFunction<Tuple2<Long,String>, String, String>() {  
 private static final long serialVersionUID = 1L;  
 @Override  
 public Tuple2<String, String> call(Tuple2<Long, String> tuple)  
 throws Exception {  
 Random random = new Random();  
 int prefix = random.nextInt(100);  
 return new Tuple2<String, String>(prefix + "\_" + tuple.\_1, tuple.\_2);  
 }  
 });  
   
// 将两个处理后的RDD进行join即可。  
JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD);

##### 1.13.6.7.3 方案实现原理

将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。

该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；

而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。

##### 1.13.6.7.4 方案优缺点

优点：对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。

##### 1.13.6.7.5 方案实践经验

曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。

#### 1.13.6.8 多种方案组合使用

在实践中发现，很多情况下，如果只是处理较为简单的数据倾斜场景，那么使用上述方案中的某一种基本就可以解决。但是如果要处理一个较为复杂的数据倾斜场景，那么可能需要将多种方案组合起来使用。

比如说，我们针对出现了多个数据倾斜环节的Spark作业，可以先运用解决方案一HiveETL预处理和过滤少数导致倾斜的k，预处理一部分数据，并过滤一部分数据来缓解；

其次可以对某些shuffle操作提升并行度，优化其性能；

最后还可以针对不同的聚合或join操作，选择一种方案来优化其性能。

大家需要对这些方案的思路和原理都透彻理解之后，在实践中根据各种不同的情况，灵活运用多种方案，来解决自己的数据倾斜问题。

### 1.13.7 Spark数据倾斜处理小结



## 1.14 Flink基础

### 1.14.1 简单介绍一下 Flink

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。并且 Flink 提供了数据分布、容错机制以及资源管理等核心功能。Flink提供了诸多高抽象层的API以便用户编写分布式任务：

DataSet API， 对静态数据进行批处理操作，将静态数据抽象成分布式的数据集，用户可以方便地使用Flink提供的各种操作符对分布式数据集进行处理，支持Java、Scala和Python。

DataStream API，对数据流进行流处理操作，将流式的数据抽象成分布式的数据流，用户可以方便地对分布式数据流进行各种操作，支持Java和Scala。

Table API，对结构化数据进行查询操作，将结构化数据抽象成关系表，并通过类SQL的DSL对关系表进行各种查询操作，支持Java和Scala。

此外，Flink 还针对特定的应用领域提供了领域库，例如： Flink ML，Flink 的机器学习库，提供了机器学习Pipelines API并实现了多种机器学习算法。 Gelly，Flink 的图计算库，提供了图计算的相关API及多种图计算算法实现。根据官网的介绍，Flink 的特性包含：

### 1.14.2 Flink相比传统的Spark Streaming区别?

这个问题是一个非常宏观的问题，因为两个框架的不同点非常之多。但是在面试时有非常重要的一点一定要回答出来：**Flink 是标准的实时处理引擎，基于事件驱动。而 Spark Streaming 是微批（Micro-Batch）的模型。**

下面我们就分几个方面介绍两个框架的主要区别：

1. 架构模型Spark Streaming 在运行时的主要角色包括：Master、Worker、Driver、Executor，Flink 在运行时主要包含：Jobmanager、Taskmanager和Slot。

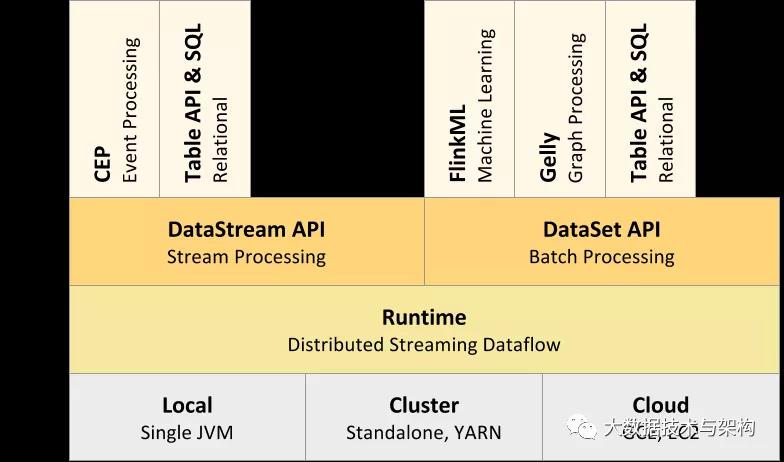
2. 任务调度Spark Streaming 连续不断的生成微小的数据批次，构建有向无环图DAG，Spark Streaming 会依次创建 DStreamGraph、JobGenerator、JobScheduler。Flink 根据用户提交的代码生成 StreamGraph，经过优化生成 JobGraph，然后提交给 JobManager进行处理，JobManager 会根据 JobGraph 生成 ExecutionGraph，ExecutionGraph 是 Flink 调度最核心的数据结构，JobManager 根据 ExecutionGraph 对 Job 进行调度。

3. 时间机制Spark Streaming 支持的时间机制有限，只支持处理时间。 Flink 支持了流处理程序在时间上的三个定义：处理时间、事件时间、注入时间。同时也支持 watermark 机制来处理滞后数据。

4. 容错机制对于 Spark Streaming 任务，我们可以设置 checkpoint，然后假如发生故障并重启，我们可以从上次 checkpoint 之处恢复，但是这个行为只能使得数据不丢失，可能会重复处理，不能做到恰好一次处理语义。Flink 则使用两阶段提交协议来解决这个问题。

### 1.14.3 Flink的组件栈有哪些？

根据 Flink 官网描述，Flink 是一个分层架构的系统，每一层所包含的组件都提供了特定的抽象，用来服务于上层组件。



自下而上，每一层分别代表：Deploy 层：该层主要涉及了Flink的部署模式，在上图中我们可以看出，Flink 支持包括local、Standalone、Cluster、Cloud等多种部署模式。Runtime 层：Runtime层提供了支持 Flink 计算的核心实现，比如：支持分布式 Stream 处理、JobGraph到ExecutionGraph的映射、调度等等，为上层API层提供基础服务。API层：API 层主要实现了面向流（Stream）处理和批（Batch）处理API，其中面向流处理对应DataStream API，面向批处理对应DataSet API，后续版本，Flink有计划将DataStream和DataSet API进行统一。Libraries层：该层称为Flink应用框架层，根据API层的划分，在API层之上构建的满足特定应用的实现计算框架，也分别对应于面向流处理和面向批处理两类。面向流处理支持：CEP（复杂事件处理）、基于SQL-like的操作（基于Table的关系操作）；面向批处理支持：FlinkML（机器学习库）、Gelly（图处理）。

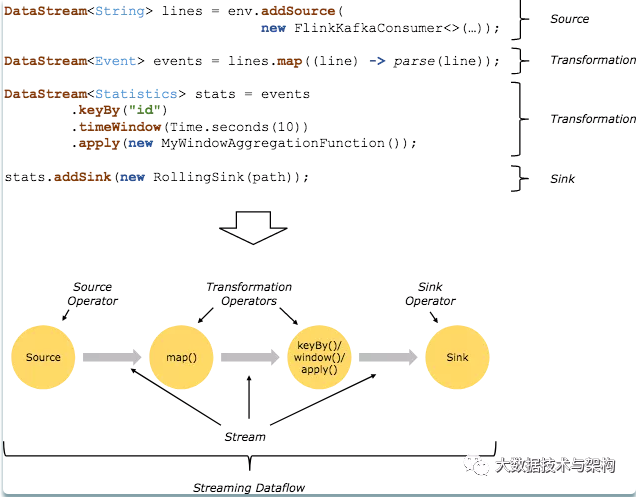
### 1.14.4 Flink 的运行必须依赖 Hadoop组件吗？

Flink可以完全独立于Hadoop，在不依赖Hadoop组件下运行。但是做为大数据的基础设施，Hadoop体系是任何大数据框架都绕不过去的。Flink可以集成众多Hadooop 组件，例如Yarn、Hbase、HDFS等等。例如，Flink可以和Yarn集成做资源调度，也可以读写HDFS，或者利用HDFS做检查点。

### 1.14.5 你们的Flink集群规模多大？

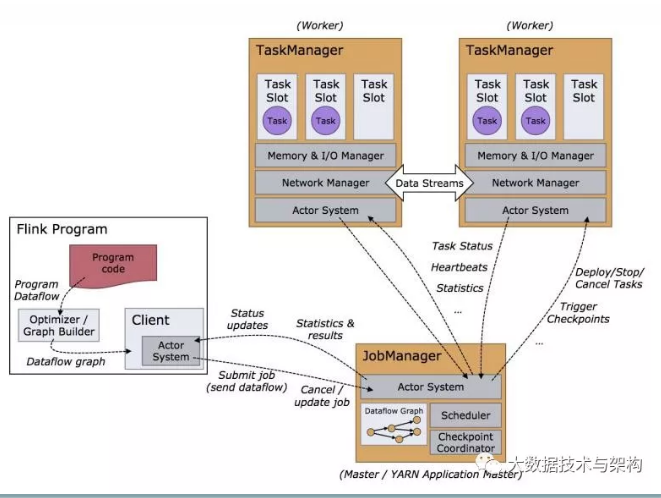
大家注意，这个问题看起来是问你实际应用中的Flink集群规模，其实还隐藏着另一个问题：Flink可以支持多少节点的集群规模？在回答这个问题时候，可以将自己生产环节中的集群规模、节点、内存情况说明，同时说明部署模式（一般是Flink on Yarn），除此之外，用户也可以同时在小集群（少于5个节点）和拥有 TB 级别状态的上千个节点上运行 Flink 任务。

### 1.14.6 Flink的基础编程模型了解吗？



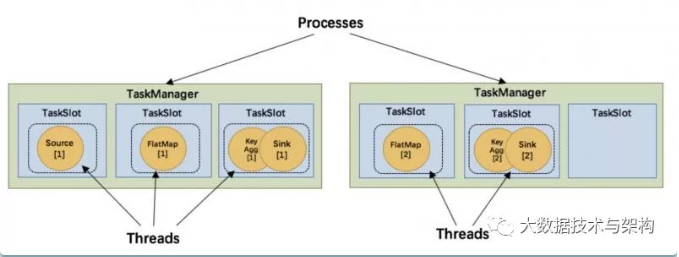
上图是来自Flink官网的运行流程图。通过上图我们可以得知，Flink 程序的基本构建是数据输入来自一个 Source，Source 代表数据的输入端，经过 Transformation 进行转换，然后在一个或者多个Sink接收器中结束。数据流（stream）就是一组永远不会停止的数据记录流，而转换（transformation）是将一个或多个流作为输入，并生成一个或多个输出流的操作。执行时，Flink程序映射到 streaming dataflows，由流（streams）和转换操作（transformation operators）组成。

### 1.14.7 Flink集群有哪些角色？各自有什么作用？



Flink 程序在运行时主要有 TaskManager，JobManager，Client三种角色。其中JobManager扮演着集群中的管理者Master的角色，它是整个集群的协调者，负责接收Flink Job，协调检查点，Failover 故障恢复等，同时管理Flink集群中从节点TaskManager。TaskManager是实际负责执行计算的Worker，在其上执行Flink Job的一组Task，每个TaskManager负责管理其所在节点上的资源信息，如内存、磁盘、网络，在启动的时候将资源的状态向JobManager汇报。Client是Flink程序提交的客户端，当用户提交一个Flink程序时，会首先创建一个Client，该Client首先会对用户提交的Flink程序进行预处理，并提交到Flink集群中处理，所以Client需要从用户提交的Flink程序配置中获取JobManager的地址，并建立到JobManager的连接，将Flink Job提交给JobManager。

### 1.14.8 说说 Flink 资源管理中 Task Slot 的概念



在Flink架构角色中我们提到，TaskManager是实际负责执行计算的Worker，TaskManager 是一个 JVM 进程，并会以独立的线程来执行一个task或多个subtask。为了控制一个 TaskManager 能接受多少个 task，Flink 提出了 Task Slot 的概念。简单的说，TaskManager会将自己节点上管理的资源分为不同的Slot：固定大小的资源子集。这样就避免了不同Job的Task互相竞争内存资源，但是需要主要的是，Slot只会做内存的隔离。没有做CPU的隔离。

### 1.14.9 说说 Flink 的常用算子？

Flink 最常用的常用算子包括：Map：DataStream → DataStream，输入一个参数产生一个参数，map的功能是对输入的参数进行转换操作。Filter：过滤掉指定条件的数据。KeyBy：按照指定的key进行分组。Reduce：用来进行结果汇总合并。Window：窗口函数，根据某些特性将每个key的数据进行分组（例如：在5s内到达的数据）

### 1.14.10 说说你知道的Flink分区策略？

什么要搞懂什么是分区策略。分区策略是用来决定数据如何发送至下游。目前 Flink 支持了8中分区策略的实现。



上图是整个Flink实现的分区策略继承图：**GlobalPartitioner** 数据会被分发到下游算子的第一个实例中进行处理。**ShufflePartitioner** 数据会被随机分发到下游算子的每一个实例中进行处理。**RebalancePartitioner** 数据会被循环发送到下游的每一个实例中进行处理。**RescalePartitioner** 这种分区器会根据上下游算子的并行度，循环的方式输出到下游算子的每个实例。这里有点难以理解，假设上游并行度为2，编号为A和B。下游并行度为4，编号为1，2，3，4。那么A则把数据循环发送给1和2，B则把数据循环发送给3和4。假设上游并行度为4，编号为A，B，C，D。下游并行度为2，编号为1，2。那么A和B则把数据发送给1，C和D则把数据发送给2。**BroadcastPartitioner** 广播分区会将上游数据输出到下游算子的每个实例中。适合于大数据集和小数据集做Jion的场景。**ForwardPartitioner** ForwardPartitioner 用于将记录输出到下游本地的算子实例。它要求上下游算子并行度一样。简单的说，ForwardPartitioner用来做数据的控制台打印。**KeyGroupStreamPartitioner** Hash分区器。会将数据按 Key 的 Hash 值输出到下游算子实例中。**CustomPartitionerWrapper** 用户自定义分区器。需要用户自己实现Partitioner接口，来定义自己的分区逻辑。例如：

static classCustomPartitionerimplementsPartitioner<String> {  
 @Override  
 publicintpartition(String key, int numPartitions) {  
 switch (key){  
 case "1":  
 return 1;  
 case "2":  
 return 2;  
 case "3":  
 return 3;  
 default:  
 return 4;  
 }  
 }  
 }

### 1.14.11 Flink的并行度了解吗？Flink的并行度设置是怎样的？

Flink中的任务被分为多个并行任务来执行，其中每个并行的实例处理一部分数据。这些并行实例的数量被称为并行度。我们在实际生产环境中可以从四个不同层面设置并行度：

操作算子层面(Operator Level)

执行环境层面(Execution Environment Level)

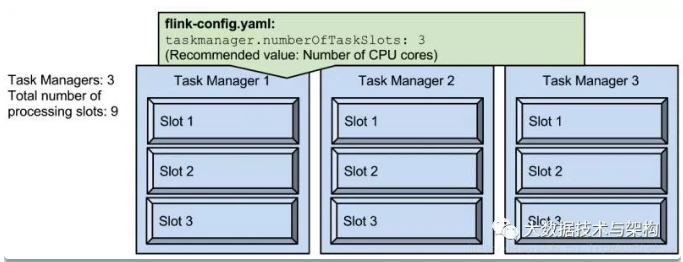
客户端层面(Client Level)

系统层面(System Level)

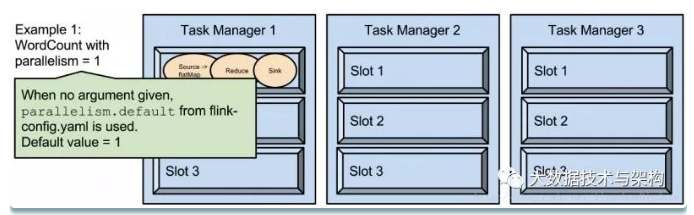
需要注意的优先级：算子层面>环境层面>客户端层面>系统层面。

### 1.14.12 Flink的Slot和parallelism有什么区别？

官网上十分经典的图：



slot是指taskmanager的并发执行能力，假设我们将 taskmanager.numberOfTaskSlots 配置为3 那么每一个 taskmanager 中分配3个 TaskSlot, 3个 taskmanager 一共有9个TaskSlot。



parallelism是指taskmanager实际使用的并发能力。假设我们把 parallelism.default 设置为1，那么9个 TaskSlot 只能用1个，有8个空闲。

### 1.14.13 Flink有没有重启策略？说说有哪几种？

Flink 实现了多种重启策略。

固定延迟重启策略（Fixed Delay Restart Strategy）

故障率重启策略（Failure Rate Restart Strategy）

没有重启策略（No Restart Strategy）

Fallback重启策略（Fallback Restart Strategy）

### 1.14.14 用过Flink中的分布式缓存吗？如何使用？

Flink实现的分布式缓存和Hadoop有异曲同工之妙。目的是在本地读取文件，并把他放在 taskmanager 节点中，防止task重复拉取。

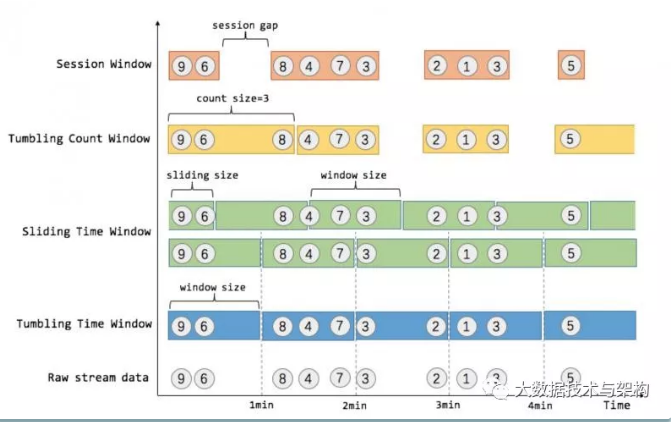
val env = ExecutionEnvironment.getExecutionEnvironment  
  
// register a file from HDFS  
env.registerCachedFile("hdfs:///path/to/your/file", "hdfsFile")  
  
// register a local executable file (script, executable, ...)  
env.registerCachedFile("file:///path/to/exec/file", "localExecFile", true)  
  
// define your program and execute  
...  
val input: DataSet[String] = ...  
val result: DataSet[Integer] = input.map(new MyMapper())  
...  
env.execute()

### 1.14.15 说说Flink中的广播变量，使用时需要注意什么？

我们知道Flink是并行的，计算过程可能不在一个 Slot 中进行，那么有一种情况即：当我们需要访问同一份数据。那么Flink中的广播变量就是为了解决这种情况。我们可以把广播变量理解为是一个公共的共享变量，我们可以把一个dataset 数据集广播出去，然后不同的task在节点上都能够获取到，这个数据在每个节点上只会存在一份。

### 1.14.16 说说Flink中的窗口？

来一张官网经典的图：



Flink 支持两种划分窗口的方式，按照time和count。如果根据时间划分窗口，那么它就是一个time-window 如果根据数据划分窗口，那么它就是一个count-window。flink支持窗口的两个重要属性（size和interval）如果size=interval,那么就会形成tumbling-window(无重叠数据) 如果size>interval,那么就会形成sliding-window(有重叠数据) 如果size< interval, 那么这种窗口将会丢失数据。比如每5秒钟，统计过去3秒的通过路口汽车的数据，将会漏掉2秒钟的数据。通过组合可以得出四种基本窗口：

time-tumbling-window 无重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5))

time-sliding-window 有重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5), Time.seconds(3))

count-tumbling-window无重叠数据的数量窗口，设置方式举例：countWindow(5)

count-sliding-window 有重叠数据的数量窗口，设置方式举例：countWindow(5,3)

### 1.14.17 说说Flink中的状态存储？

Flink在做计算的过程中经常需要存储中间状态，来避免数据丢失和状态恢复。选择的状态存储策略不同，会影响状态持久化如何和 checkpoint 交互。Flink提供了三种状态存储方式：MemoryStateBackend、FsStateBackend、RocksDBStateBackend。

### 1.14.18 Flink中的时间有哪几类

Flink 中的时间和其他流式计算系统的时间一样分为三类：事件时间，摄入时间，处理时间三种。如果以 EventTime 为基准来定义时间窗口将形成EventTimeWindow,要求消息本身就应该携带EventTime。如果以 IngesingtTime 为基准来定义时间窗口将形成 IngestingTimeWindow,以 source 的systemTime为准。如果以 ProcessingTime 基准来定义时间窗口将形成 ProcessingTimeWindow，以 operator 的systemTime 为准。

### 1.14.19 Flink 中水印是什么概念，起到什么作用？

Watermark 是 Apache Flink 为了处理 EventTime 窗口计算提出的一种机制, 本质上是一种时间戳。 一般来讲Watermark经常和Window一起被用来处理乱序事件。

### 1.14.20 Flink Table & SQL 熟悉吗？TableEnvironment这个类有什么作用

TableEnvironment是Table API和SQL集成的核心概念。这个类主要用来：

在内部catalog中注册表

注册外部catalog

执行SQL查询

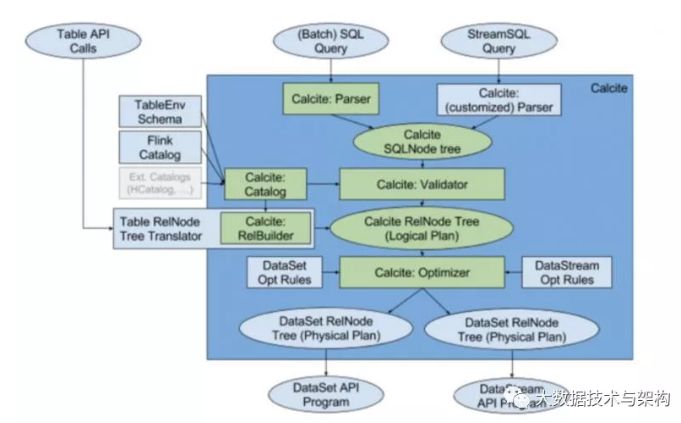
注册用户定义（标量，表或聚合）函数

将DataStream或DataSet转换为表

持有对ExecutionEnvironment或StreamExecutionEnvironment的引用

### 1.14.20 Flink SQL的实现原理是什么？是如何实现 SQL 解析的呢？

首先大家要知道 Flink 的SQL解析是基于Apache Calcite这个开源框架。



基于此，一次完整的SQL解析过程如下：

用户使用对外提供Stream SQL的语法开发业务应用

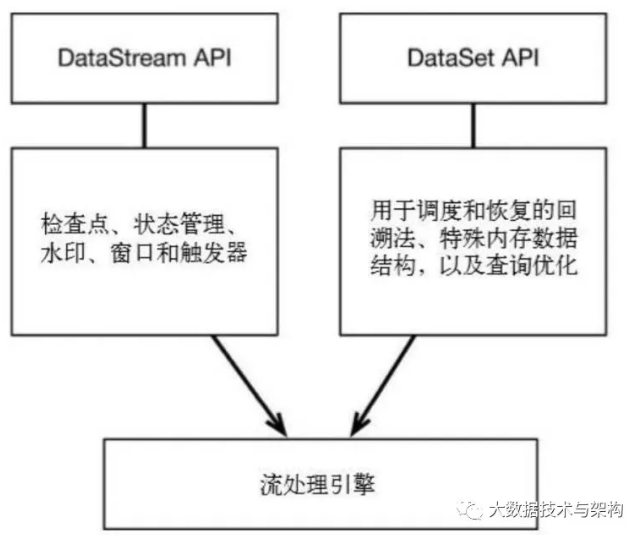
用calcite对StreamSQL进行语法检验，语法检验通过后，转换成calcite的逻辑树节点；最终形成calcite的逻辑计划

采用Flink自定义的优化规则和calcite火山模型、启发式模型共同对逻辑树进行优化，生成最优的Flink物理计划

对物理计划采用janino codegen生成代码，生成用低阶API DataStream 描述的流应用，提交到Flink平台执行

## 1.15 Flink中级

### 1.15.1 Flink是如何支持批流一体的？



本道面试题考察的其实就是一句话：Flink的开发者认为批处理是流处理的一种特殊情况。批处理是有限的流处理。Flink 使用一个引擎支持了DataSet API 和 DataStream API。

### 1.15.2 Flink是如何做到高效的数据交换的？

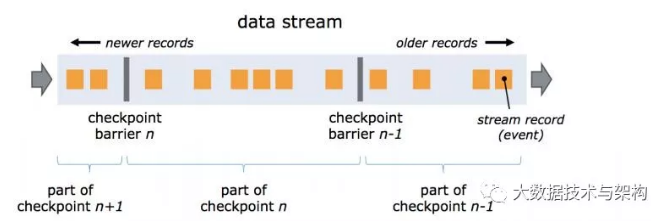
在一个Flink Job中，数据需要在不同的task中进行交换，整个数据交换是有 TaskManager 负责的，TaskManager 的网络组件首先从缓冲buffer中收集records，然后再发送。Records 并不是一个一个被发送的，二是积累一个批次再发送，batch 技术可以更加高效的利用网络资源。

### 1.15.3 Flink是如何做容错的？

Flink 实现容错主要靠强大的CheckPoint机制和State机制。Checkpoint 负责定时制作分布式快照、对程序中的状态进行备份；State 用来存储计算过程中的中间状态。

### 1.15.4 Flink 分布式快照的原理是什么？

Flink的分布式快照是根据Chandy-Lamport算法量身定做的。简单来说就是持续创建分布式数据流及其状态的一致快照。



核心思想是在 input source 端插入 barrier，控制 barrier 的同步来实现 snapshot 的备份和 exactly-once 语义。

### 1.15.5 Flink是如何保证Exactly-once语义的？

Flink通过实现两阶段提交和状态保存来实现端到端的一致性语义。 分为以下几个步骤：

开始事务（beginTransaction）创建一个临时文件夹，来写把数据写入到这个文件夹里面

预提交（preCommit）将内存中缓存的数据写入文件并关闭

正式提交（commit）将之前写完的临时文件放入目标目录下。这代表着最终的数据会有一些延迟

丢弃（abort）丢弃临时文件

若失败发生在预提交成功后，正式提交前。可以根据状态来提交预提交的数据，也可删除预提交的数据。

### 1.15.6 Flink 的 kafka 连接器有什么特别的地方？

Flink源码中有一个独立的connector模块，所有的其他connector都依赖于此模块，Flink 在1.9版本发布的全新kafka连接器，摒弃了之前连接不同版本的kafka集群需要依赖不同版本的connector这种做法，只需要依赖一个connector即可。

### 1.15.7 说说 Flink的内存管理是如何做的?

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上。此外，Flink大量的使用了堆外内存。如果需要处理的数据超出了内存限制，则会将部分数据存储到硬盘上。Flink 为了直接操作二进制数据实现了自己的序列化框架。理论上Flink的内存管理分为三部分：

Network Buffers：这个是在TaskManager启动的时候分配的，这是一组用于缓存网络数据的内存，每个块是32K，默认分配2048个，可以通过“taskmanager.network.numberOfBuffers”修改

Memory Manage pool：大量的Memory Segment块，用于运行时的算法（Sort/Join/Shuffle等），这部分启动的时候就会分配。下面这段代码，根据配置文件中的各种参数来计算内存的分配方法。（heap or off-heap，这个放到下节谈），内存的分配支持预分配和lazy load，默认懒加载的方式。

User Code，这部分是除了Memory Manager之外的内存用于User code和TaskManager本身的数据结构。

### 1.15.8 说说 Flink的序列化如何做的?

Java本身自带的序列化和反序列化的功能，但是辅助信息占用空间比较大，在序列化对象时记录了过多的类信息。Apache Flink摒弃了Java原生的序列化方法，以独特的方式处理数据类型和序列化，包含自己的类型描述符，泛型类型提取和类型序列化框架。TypeInformation 是所有类型描述符的基类。它揭示了该类型的一些基本属性，并且可以生成序列化器。TypeInformation 支持以下几种类型：

BasicTypeInfo: 任意Java 基本类型或 String 类型

BasicArrayTypeInfo: 任意Java基本类型数组或 String 数组

WritableTypeInfo: 任意 Hadoop Writable 接口的实现类

TupleTypeInfo: 任意的 Flink Tuple 类型(支持Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的Java Tuple实现

CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)

PojoTypeInfo: 任意的 POJO (Java or Scala)，例如，Java对象的所有成员变量，要么是 public 修饰符定义，要么有 getter/setter 方法

GenericTypeInfo: 任意无法匹配之前几种类型的类

针对前六种类型数据集，Flink皆可以自动生成对应的TypeSerializer，能非常高效地对数据集进行序列化和反序列化。

### 1.15.9 Flink中的Window出现了数据倾斜，你有什么解决办法？

window产生数据倾斜指的是数据在不同的窗口内堆积的数据量相差过多。本质上产生这种情况的原因是数据源头发送的数据量速度不同导致的。出现这种情况一般通过两种方式来解决：

在数据进入窗口前做预聚合

重新设计窗口聚合的key

### 1.15.10 Flink中在使用聚合函数 GroupBy、Distinct、KeyBy 等函数时出现数据热点该如何解决？

数据倾斜和数据热点是所有大数据框架绕不过去的问题。处理这类问题主要从3个方面入手：

在业务上规避这类问题

例如一个假设订单场景，北京和上海两个城市订单量增长几十倍，其余城市的数据量不变。这时候我们在进行聚合的时候，北京和上海就会出现数据堆积，我们可以单独数据北京和上海的数据。

Key的设计上

把热key进行拆分，比如上个例子中的北京和上海，可以把北京和上海按照地区进行拆分聚合。

参数设置

Flink 1.9.0 SQL(Blink Planner) 性能优化中一项重要的改进就是升级了微批模型，即 MiniBatch。原理是缓存一定的数据后再触发处理，以减少对State的访问，从而提升吞吐和减少数据的输出量。

### 1.15.11 Flink任务延迟高，想解决这个问题，你会如何入手？

在Flink的后台任务管理中，我们可以看到Flink的哪个算子和task出现了反压。最主要的手段是资源调优和算子调优。资源调优即是对作业中的Operator的并发数（parallelism）、CPU（core）、堆内存（heap\_memory）等参数进行调优。作业参数调优包括：并行度的设置，State的设置，checkpoint的设置。

### 1.15.12 Flink是如何处理反压的？

Flink 内部是基于 producer-consumer 模型来进行消息传递的，Flink的反压设计也是基于这个模型。Flink 使用了高效有界的分布式阻塞队列，就像 Java 通用的阻塞队列（BlockingQueue）一样。下游消费者消费变慢，上游就会受到阻塞。

### 1.15.13 Flink的反压和Strom有哪些不同？

Storm 是通过监控 Bolt 中的接收队列负载情况，如果超过高水位值就会将反压信息写到 Zookeeper ，Zookeeper 上的 watch 会通知该拓扑的所有 Worker 都进入反压状态，最后 Spout 停止发送 tuple。Flink中的反压使用了高效有界的分布式阻塞队列，下游消费变慢会导致发送端阻塞。二者最大的区别是Flink是逐级反压，而Storm是直接从源头降速。

### 1.15.14 Operator Chains（算子链）这个概念你了解吗？

为了更高效地分布式执行，Flink会尽可能地将operator的subtask链接（chain）在一起形成task。每个task在一个线程中执行。将operators链接成task是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量。这就是我们所说的算子链。

### 1.15.15 Flink什么情况下才会把Operator chain在一起形成算子链？

两个operator chain在一起的的条件：

上下游的并行度一致

下游节点的入度为1 （也就是说下游节点没有来自其他节点的输入）

上下游节点都在同一个 slot group 中（下面会解释 slot group）

下游节点的 chain 策略为 ALWAYS（可以与上下游链接，map、flatmap、filter等默认是ALWAYS）

上游节点的 chain 策略为 ALWAYS 或 HEAD（只能与下游链接，不能与上游链接，Source默认是HEAD）

两个节点间数据分区方式是 forward（参考理解数据流的分区）

用户没有禁用 chain

### 1.15.16 说说Flink1.9的新特性？

支持hive读写，支持UDF

Flink SQL TopN和GroupBy等优化

Checkpoint跟savepoint针对实际业务场景做了优化

Flink state查询

### 1.15.17 消费kafka数据的时候，如何处理脏数据？

可以在处理前加一个fliter算子，将不符合规则的数据过滤出去。

## 1.16 Flink高级

### 1.16.1 Flink Job的提交流程

用户提交的Flink Job会被转化成一个DAG任务运行，分别是：StreamGraph、JobGraph、ExecutionGraph，Flink中JobManager与TaskManager，JobManager与Client的交互是基于Akka工具包的，是通过消息驱动。整个Flink Job的提交还包含着ActorSystem的创建，JobManager的启动，TaskManager的启动和注册。

### 1.16.2 Flink所谓"三层图"结构是哪几个"图"？

一个Flink任务的DAG生成计算图大致经历以下三个过程：

StreamGraph 最接近代码所表达的逻辑层面的计算拓扑结构，按照用户代码的执行顺序向StreamExecutionEnvironment添加StreamTransformation构成流式图。

JobGraph 从StreamGraph生成，将可以串联合并的节点进行合并，设置节点之间的边，安排资源共享slot槽位和放置相关联的节点，上传任务所需的文件，设置检查点配置等。相当于经过部分初始化和优化处理的任务图。

ExecutionGraph 由JobGraph转换而来，包含了任务具体执行所需的内容，是最贴近底层实现的执行图。

### 1.16.3 JobManger在集群中扮演了什么角色？

JobManager 负责整个 Flink 集群任务的调度以及资源的管理，从客户端中获取提交的应用，然后根据集群中 TaskManager 上 TaskSlot 的使用情况，为提交的应用分配相应的 TaskSlot 资源并命令 TaskManager 启动从客户端中获取的应用。JobManager 相当于整个集群的 Master 节点，且整个集群有且只有一个活跃的 JobManager ，负责整个集群的任务管理和资源管理。JobManager 和 TaskManager 之间通过 Actor System 进行通信，获取任务执行的情况并通过 Actor System 将应用的任务执行情况发送给客户端。同时在任务执行的过程中，Flink JobManager 会触发 Checkpoint 操作，每个 TaskManager 节点 收到 Checkpoint 触发指令后，完成 Checkpoint 操作，所有的 Checkpoint 协调过程都是在 Fink JobManager 中完成。当任务完成后，Flink 会将任务执行的信息反馈给客户端，并且释放掉 TaskManager 中的资源以供下一次提交任务使用。

### 1.16.4 JobManger在集群启动过程中起到什么作用？

JobManager的职责主要是接收Flink作业，调度Task，收集作业状态和管理TaskManager。它包含一个Actor，并且做如下操作：

RegisterTaskManager: 它由想要注册到JobManager的TaskManager发送。注册成功会通过AcknowledgeRegistration消息进行Ack。

SubmitJob: 由提交作业到系统的Client发送。提交的信息是JobGraph形式的作业描述信息。

CancelJob: 请求取消指定id的作业。成功会返回CancellationSuccess，否则返回CancellationFailure。

UpdateTaskExecutionState: 由TaskManager发送，用来更新执行节点(ExecutionVertex)的状态。成功则返回true，否则返回false。

RequestNextInputSplit: TaskManager上的Task请求下一个输入split，成功则返回NextInputSplit，否则返回null。

JobStatusChanged： 它意味着作业的状态(RUNNING, CANCELING, FINISHED,等)发生变化。这个消息由ExecutionGraph发送。

### 1.16.5 TaskManager在集群中扮演了什么角色？

TaskManager 相当于整个集群的 Slave 节点，负责具体的任务执行和对应任务在每个节点上的资源申请和管理。客户端通过将编写好的 Flink 应用编译打包，提交到 JobManager，然后 JobManager 会根据已注册在 JobManager 中 TaskManager 的资源情况，将任务分配给有资源的 TaskManager节点，然后启动并运行任务。TaskManager 从 JobManager 接收需要部署的任务，然后使用 Slot 资源启动 Task，建立数据接入的网络连接，接收数据并开始数据处理。同时 TaskManager 之间的数据交互都是通过数据流的方式进行的。可以看出，Flink 的任务运行其实是采用多线程的方式，这和 MapReduce 多 JVM 进行的方式有很大的区别，Flink 能够极大提高 CPU 使用效率，在多个任务和 Task 之间通过 TaskSlot 方式共享系统资源，每个 TaskManager 中通过管理多个 TaskSlot 资源池进行对资源进行有效管理。

### 1.16.6 TaskManager在集群启动过程中起到什么作用？

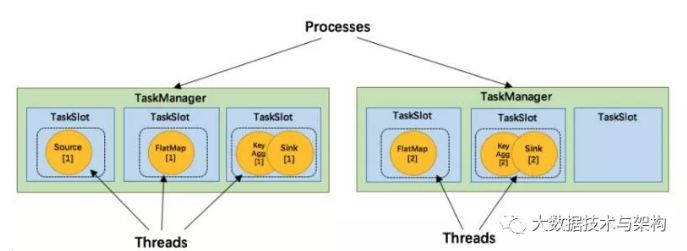
TaskManager的启动流程较为简单： 启动类：org.apache.flink.runtime.taskmanager.TaskManager 核心启动方法 ： selectNetworkInterfaceAndRunTaskManager 启动后直接向JobManager注册自己，注册完成后，进行部分模块的初始化。

### 1.16.7 Flink 计算资源的调度是如何实现的？

TaskManager中最细粒度的资源是Task slot，代表了一个固定大小的资源子集，每个TaskManager会将其所占有的资源平分给它的slot。

通过调整 task slot 的数量，用户可以定义task之间是如何相互隔离的。每个 TaskManager 有一个slot，也就意味着每个task运行在独立的 JVM 中。每个 TaskManager 有多个slot的话，也就是说多个task运行在同一个JVM中。

而在同一个JVM进程中的task，可以共享TCP连接（基于多路复用）和心跳消息，可以减少数据的网络传输，也能共享一些数据结构，一定程度上减少了每个task的消耗。 每个slot可以接受单个task，也可以接受多个连续task组成的pipeline，如下图所示，FlatMap函数占用一个taskslot，而key Agg函数和sink函数共用一个taskslot：

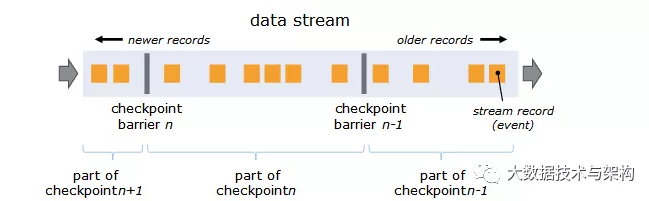


### 1.16.8 简述Flink的数据抽象及数据交换过程？

Flink 为了避免JVM的固有缺陷例如java对象存储密度低，FGC影响吞吐和响应等，实现了自主管理内存。MemorySegment就是Flink的内存抽象。默认情况下，一个MemorySegment可以被看做是一个32kb大的内存块的抽象。这块内存既可以是JVM里的一个byte[]，也可以是堆外内存（DirectByteBuffer）。在MemorySegment这个抽象之上，Flink在数据从operator内的数据对象在向TaskManager上转移，预备被发给下个节点的过程中，使用的抽象或者说内存对象是Buffer。对接从Java对象转为Buffer的中间对象是另一个抽象StreamRecord。

### 1.16.9 Flink 中的分布式快照机制是如何实现的？

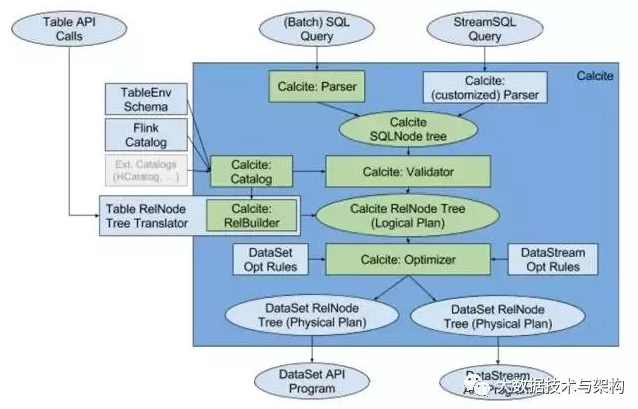
Flink的容错机制的核心部分是制作分布式数据流和操作算子状态的一致性快照。 这些快照充当一致性checkpoint，系统可以在发生故障时回滚。 Flink用于制作这些快照的机制在“分布式数据流的轻量级异步快照”中进行了描述。 它受到分布式快照的标准Chandy-Lamport算法的启发，专门针对Flink的执行模型而定制。



barriers在数据流源处被注入并行数据流中。快照n的barriers被插入的位置（我们称之为Sn）是快照所包含的数据在数据源中最大位置。例如，在Apache Kafka中，此位置将是分区中最后一条记录的偏移量。 将该位置Sn报告给checkpoint协调器（Flink的JobManager）。然后barriers向下游流动。当一个中间操作算子从其所有输入流中收到快照n的barriers时，它会为快照n发出barriers进入其所有输出流中。 一旦sink操作算子（流式DAG的末端）从其所有输入流接收到barriers n，它就向checkpoint协调器确认快照n完成。在所有sink确认快照后，意味快照着已完成。一旦完成快照n，job将永远不再向数据源请求Sn之前的记录，因为此时这些记录（及其后续记录）将已经通过整个数据流拓扑，也即是已经被处理结束。

### 1.16.10 简单说说FlinkSQL的是如何实现的？

Flink 将 SQL 校验、SQL 解析以及 SQL 优化交给了Apache Calcite。Calcite 在其他很多开源项目里也都应用到了，譬如 Apache Hive, Apache Drill, Apache Kylin, Cascading。Calcite 在新的架构中处于核心的地位，如下图所示。

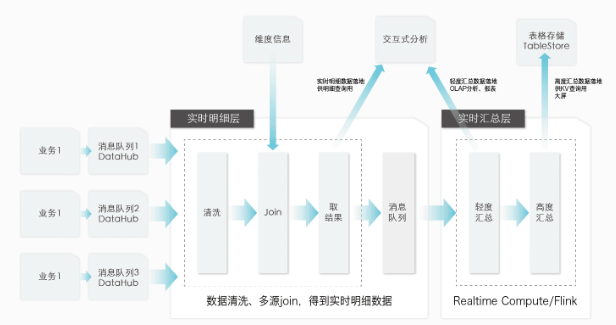


构建抽象语法树的事情交给了 Calcite 去做。SQL query 会经过 Calcite 解析器转变成 SQL 节点树，通过验证后构建成 Calcite 的抽象语法树（也就是图中的 Logical Plan）。另一边，Table API 上的调用会构建成 Table API 的抽象语法树，并通过 Calcite 提供的 RelBuilder 转变成 Calcite 的抽象语法树。然后依次被转换成逻辑执行计划和物理执行计划。在提交任务后会分发到各个 TaskManager 中运行，在运行时会使用 Janino 编译器编译代码后运行。

# 第2章 项目架构

云上数据仓库解决方案：<https://www.aliyun.com/solution/datavexpo/datawarehouse>





## 2.1 数仓概念

**数据仓库的输入数据源和输出系统分别是什么？**

输入系统：埋点产生的用户行为数据、JavaEE后台产生的业务数据、个别公司有爬虫数据。

输出系统：报表系统、用户画像系统、推荐系统

## 2.2 系统数据流程设计

参考实际项目作答

## 2.3 框架版本选型

1）Apache：运维麻烦，组件间兼容性需要自己调研。（一般大厂使用，技术实力雄厚，有专业的运维人员）

2）CDH：国内使用最多的版本，但 CM不开源，但其实对中、小公司使用来说没有影响（建议使用）10000美金一个节点 CDP

3）HDP：开源，可以进行二次开发，但是没有CDH稳定，国内使用较少

## 2.4 服务器选型

服务器使用物理机还是云主机？

1）机器成本考虑：

（1）物理机：以128G内存，20核物理CPU，40线程，8THDD和2TSSD硬盘，单台报价4W出头，惠普品牌。一般物理机寿命5年左右。

（2）云主机，以阿里云为例，差不多相同配置，每年5W

2）运维成本考虑：

（1）物理机：需要有专业的运维人员（1万\*13）、电费（商业用户）、安装空调

（2）云主机：很多运维工作都由阿里云已经完成，运维相对较轻松

3）企业选择

（1）金融有钱公司和阿里没有直接冲突的公司选择阿里云（上海）

（2）中小公司、为了融资上市，选择阿里云，拉倒融资后买物理机。

（3）有长期打算，资金比较足，选择物理机。

## 2.5 集群规模

根据实际情况作答

根据数据规模大家集群

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| nn | nn | dn | dn | dn | dn | dn | dn | dn | dn |
|  |  | rm | rm | nm | nm | nm | nm | nm | nm |
|  |  | nm | nm |  |  |  |  |  |  |
|  |  |  |  |  |  |  | zk | zk | zk |
|  |  |  |  |  |  |  | kafka | kafka | kafka |
|  |  |  |  |  |  |  | Flume | Flume | flume |
|  |  | Hbase | Hbase | Hbase |  |  |  |  |  |
| hive | hive |  |  |  |  |  |  |  |  |
| mysql | mysql |  |  |  |  |  |  |  |  |
| spark | spark |  |  |  |  |  |  |  |  |
|  |  |  |  |  | ES | ES |  |  |  |

## 2.6 人员配置参考

### 2.6.1 整体架构

属于**研发部**/技术部/数据部，我们属于**大数据组**，其他还有后端项目组，前端组、测试组、UI组等。其他的还有产品部、运营部、人事部、财务部、行政部等。

大数据开发工程师=>大数据组组长=》项目经理=>部门经理=》技术总监

### 2.6.2 你们部门的职级等级，晋升规则

职级就分初级，中级，高级。晋升规则不一定，看公司效益和职位空缺。

京东：T1、T2应届生；T3 14k左右 T4 18K左右 T5 24k-28k左右

阿里：p5、p6、p7、p8

### 2.6.3 人员配置参考

小型公司（3人左右）：组长1人，剩余组员无明确分工，并且可能兼顾javaEE和前端。

中小型公司（3~6人左右）：组长1人，离线2人左右，实时1人左右（离线一般多于实时），组长兼顾和javaEE、前端。

中型公司（5~10人左右）：组长1人，离线3~5人左右（离线处理、数仓），实时2人左右，组长和技术大牛兼顾和javaEE、前端。

中大型公司（10~20人左右）：组长1人，离线5~10人（离线处理、数仓），实时5人左右，JavaEE1人左右（负责对接JavaEE业务），前端1人（有或者没有人单独负责前端）。（发展比较良好的中大型公司可能大数据部门已经细化拆分，分成多个大数据组，分别负责不同业务）

上面只是参考配置，因为公司之间差异很大，例如ofo大数据部门只有5个人左右，因此根据所选公司规模确定一个合理范围，在面试前必须将这个人员配置考虑清楚，回答时要非常确定。

IOS多少人 安卓多少人 前端多少人 JavaEE多少人 测试多少人

（IOS、安卓） 1-2个人 前端1-3个人； JavaEE一般是大数据的1-1.5倍，测试：有的有，有的没有。1个左右。 产品经理1个、产品助理1-2个，运营1-3个

公司划分：

0-50 小公司

50-500 中等

500-1000 大公司

1000以上 大厂 领军的存在

# 第3章 用户行为数据分析

## 3.1 数仓分层架构表

数据分层架构：根据实际项目作答

分层优点：复杂问题简单化、清晰数据结构(方便管理)、增加数据的复用性、隔离原始数据(解耦)

ods 原始数据层 存放原始数据，保持原貌不做处理

dwd 明细数据层 对ods层数据清洗（去除空值，脏数据，超过极限范围的数据）

dws 服务数据层 轻度聚合

ads 应用数据层 具体需求

数仓中各层建的表都是外部表

## 3.2 埋点行为数据基本格式(基本字段)

公共字段：基本所有安卓手机都包含的字段

业务字段：埋点上报的字段，有具体的业务类型

下面就是一个示例，表示业务字段的上传。

行为数据启动日志/事件日志表关键字段:

{

"ap":"xxxxx",//项目数据来源 app pc

"cm": { //公共字段

**"mid": "", // (String) 设备唯一标识**

**"uid": "", // (String) 用户标识**

"vc": "1", // (String) versionCode，程序版本号

"vn": "1.0", // (String) versionName，程序版本名

"l": "zh", // (String) 系统语言

**"sr": "", // (String) 渠道号，应用从哪个渠道来的。**

"os": "7.1.1", // (String) Android系统版本

"ar": "CN", // (String) 区域

"md": "BBB100-1", // (String) 手机型号

**"ba": "blackberry", // (String) 手机品牌**

"sv": "V2.2.1", // (String) sdkVersion

"g": "", // (String) gmail

**"hw": "1620x1080", // (String) heightXwidth，屏幕宽高**

"t": "1506047606608", // (String) 客户端日志产生时的时间

**"nw": "WIFI", // (String) 网络模式**

"ln": 0, // (double) lng经度

"la": 0 // (double) lat 纬度

},

"et": [ //事件

{

"ett": "1506047605364", //客户端事件产生时间

"en": "display", //事件名称 启动和事件日志是根据事件名称的不同

"kv": { //事件结果，以key-value形式自行定义

"goodsid": "236",

"action": "1",

"extend1": "1",

"place": "2",

"category": "75"

}

}

]

}

根据事件标签的不同可以分成不同的日志表

## 3.3 项目经验总结

### 3.3.1 项目经验之元数据备份

元数据备份（重点，如数据损坏，可能整个集群无法运行，至少要保证每日零点之后备份到其它服务器两个复本） 或者mycat



### 3.3.2 日期处理函数

1）date\_add、date\_sub函数（加减日期）

2）next\_day函数（周指标相关）

3）date\_format函数（根据格式整理日期）

4）last\_day函数（求当月最后一天日期）

5）collect\_set函数

6）get\_json\_object解析json函数

### 3.3.3 Union与Union all区别

1）union会将联合的结果集去重，效率较union all差

2）union all不会对结果集去重，所以效率高

### 3.3.4 Shell中单引号和双引号区别

1）在/home/atguigu/bin创建一个test.sh文件

[atguigu@hadoop102 bin]$ vim test.sh

在文件中添加如下内容

#!/bin/bash

do\_date=$1

echo '$do\_date'

echo "$do\_date"

echo "'$do\_date'"

echo '"$do\_date"'

echo `date`

2）查看执行结果

[atguigu@hadoop102 bin]$ test.sh 2019-02-10

$do\_date

2019-02-10

'2019-02-10'

"$do\_date"

2019年 05月 02日 星期四 21:02:08 CST

3）总结：

（1）单引号不取变量值

（2）双引号取变量值

（3）反引号`，执行引号中命令

（4）双引号内部嵌套单引号，取出变量值

（5）单引号内部嵌套双引号，不取出变量值

### 3.3.5 Tez引擎优点？

Tez可以将多个有依赖的作业转换为一个作业，这样只需写一次HDFS，且中间节点较少，从而大大提升作业的计算性能。

Mr/tez/spark区别：

Mr引擎：多job串联，基于磁盘，落盘的地方比较多。虽然慢，但一定能跑出结果。一般处理，周、月、年指标。

Spark引擎：虽然在Shuffle过程中也落盘，但是并不是所有算子都需要Shuffle，尤其是多算子过程，中间过程不落盘 DAG有向无环图。 兼顾了可靠性和效率。一般处理天指标。

Tez引擎：完全基于内存。 注意：如果数据量特别大，慎重使用。容易OOM。一般用于快速出结果，数据量比较小的场景。

## 3.4 需求逻辑（重点）

### 3.4.1 如何分析用户活跃？

在启动日志中统计不同设备id出现次数。

### 3.4.2 如何分析用户新增？vivo

用活跃用户表 left join 用户新增表，用户新增表中mid为空的即为用户新增。

### 3.4.3 如何分析用户1天留存？

留存用户=前一天新增 join 今天活跃

用户留存率=留存用户/前一天新增

### 3.4.4 如何分析沉默用户？

(登录时间为7天前,且只出现过一次)

按照设备id对日活表分组，登录次数为1，且是在一周前登录。

### 3.4.5 如何分析本周回流用户？

本周活跃left join本周新增 left join上周活跃，且本周新增id和上周活跃id都为null

### 3.4.6 如何分析流失用户？

(登录时间为7天前)

按照设备id对日活表分组，且七天内没有登录过。

### 3.4.7 如何分析最近连续3周活跃用户数？

按照设备id对周活进行分组，统计次数大于3次。

### 3.4.8 如何分析最近七天内连续三天活跃用户数？

1）查询出最近7天的活跃用户，并对用户活跃日期进行排名

2）计算用户活跃日期及排名之间的差值

3）对同用户及差值分组，统计差值个数

4）将差值相同个数大于等于3的数据取出，然后去重(去的是什么重???)，即为连续3天及以上活跃的用户

7天连续收藏、点赞、购买、加购、付款、浏览、商品点击、退货

1个月连续7天

连续两周：

# 第4章 业务交互数据分析

## 4.1 电商常识

SKU：一台银色、128G内存的、支持联通网络的iPhoneX

SPU：iPhoneX

Tm\_id：品牌Id苹果，包括IPHONE，耳机，mac等

## 4.2 电商业务流程

根据实际项目作答

## 4.3 业务表关键字段

### 4.3.1 订单表（order\_info）

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | 订单编号 | |
| total\_amount | 订单金额 | |
| order\_status | 订单状态 | |
| user\_id | 用户id | |
| payment\_way | 支付方式 | |
| out\_trade\_no | 支付流水号 | |
| create\_time | 创建时间 | |
| operate\_time | 操作时间 | |

### 4.3.2 订单详情表（order\_detail）

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | 订单编号 | |
| order\_id | 订单号 | |
| user\_id | 用户id | |
| sku\_id | 商品id | |
| sku\_name | 商品名称 | |
| order\_price | 商品价格 | |
| sku\_num | 商品数量 | |
| create\_time | 创建时间 | |

### 4.3.3 商品表

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | skuId | |
| spu\_id | spuid | |
| price | 价格 | |
| sku\_name | 商品名称 | |
| sku\_desc | 商品描述 | |
| weight | 重量 | |
| tm\_id | 品牌id | |
| category3\_id | 品类id | |
| create\_time | 创建时间 | |

### 4.3.4 用户表

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | 用户id | |
| name | 姓名 | |
| birthday | 生日 | |
| gender | 性别 | |
| email | 邮箱 | |
| user\_level | 用户等级 | |
| create\_time | 创建时间 | |

### 4.3.5 商品一级分类表

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | id | |
| name | 名称 | |

### 4.3.6 商品二级分类表

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | id | |
| name | 名称 | |
| category1\_id | 一级品类id | |

### 4.3.7 商品三级分类表

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | id | |
| name | 名称 | |
| Category2\_id | 二级品类id | |

### 4.3.8 支付流水表

|  |  |  |
| --- | --- | --- |
| 标签 | | 含义 |
| id | 编号 | |
| out\_trade\_no | 对外业务编号 | |
| order\_id | 订单编号 | |
| user\_id | 用户编号 | |
| alipay\_trade\_no | 支付宝交易流水编号 | |
| total\_amount | 支付金额 | |
| subject | 交易内容 | |
| payment\_type | 支付类型 | |
| payment\_time | 支付时间 | |

订单表跟订单详情表有什么区别？

订单表的订单状态会变化，订单详情表不会，因为没有订单状态。

订单表记录user\_id，订单id订单编号，订单的总金额order\_status，支付方式，订单状态等。

订单详情表记录user\_id，商品sku\_id ,具体的商品信息（商品名称sku\_name，价格order\_price，数量sku\_num）

## 4.4 MySql中表的分类

实体表，维度表，事务型快照事实表，周期型快照事实表、累积型事实表

其实最终可以把事务型事实表，周期性事实表统称实体表，实体表，维度表统称维度表

订单表（order\_info）（周期型事实表）

订单详情表（order\_detail）(事务型事实表)

商品表(实体表)

用户表(实体表)

商品一级分类表(维度表)

商品二级分类表(维度表)

商品三级分类表(维度表)

支付流水表(事务型实体表)

## 4.5 同步策略



实体表，维度表统称维度表，每日全量或者每月（更长时间）全量

事务型事实表：每日增量

周期性事实表：拉链表

## 4.6 关系型数据库范式理论

1NF：属性不可再分割（例如不能存在5台电脑的属性，坏处：表都没法用）

2NF：不能存在部分函数依赖（例如主键（学号+课名）-->成绩，姓名，但学号--》姓名，所以姓名部分依赖于主键（学号+课名），所以要去除，坏处：数据冗余）

3NF：不能存在传递函数依赖（学号--》宿舍种类--》价钱，坏处：数据冗余和增删异常）

Mysql关系模型：关系模型主要应用与OLTP系统中，为了保证数据的一致性以及避免冗余，所以大部分业务系统的表都是遵循第三范式的。

Hive 维度模型：维度模型主要应用于OLAP系统中，因为关系模型虽然冗余少，

但是在大规模数据，跨表分析统计查询过程中，会造成多表关联，这会大大降低执行效率。

所以HIVE把相关各种表整理成两种：事实表和维度表两种。所有维度表围绕着事实表进行解释。

## 4.7 数据模型

雪花模型、星型模型和星座模型

（在维度建模的基础上又分为三种模型：星型模型、雪花模型、星座模型。）

星型模型（一级维度表），雪花（多级维度），星座模型（星型模型+多个事实表）

## 4.8 拉链表

拉链表处理的业务场景：主要处理缓慢变化维的业务场景。（用户表、订单表）

订单表拉链表 dwd\_order\_info\_his

`id` string COMMENT '订单编号',

`total\_amount` decimal(10,2) COMMENT '订单金额',

`order\_status` string COMMENT '订单状态',

`user\_id` string COMMENT '用户id' ,

`payment\_way` string COMMENT '支付方式',

`out\_trade\_no` string COMMENT '支付流水号',

`create\_time` string COMMENT '创建时间',

`operate\_time` string COMMENT '操作时间' ,

`start\_date` string COMMENT '有效开始日期',

`end\_date` string COMMENT '有效结束日期'

1）创建订单表拉链表，字段跟拉链表一样，只增加了有效开始日期和有效结束日期

初始日期，从订单变化表ods\_order\_info导入数据，且让有效开始时间=当前日期，有效结束日期=9999-99-99

（从mysql导入数仓的时候就只导了新增的和变化的数据ods\_order\_info，dwd\_order\_info跟ods\_order\_info基本一样，只多了一个id的判空处理）

2）建一张拉链临时表dwd\_order\_info\_his\_tmp，字段跟拉链表完全一致

3）新的拉链表中应该有这几部分数据，

（1）增加订单变化表dwd\_order\_info的全部数据

（2）更新旧的拉链表左关联订单变化表dwd\_order\_info，关联字段：订单id, where 过滤出end\_date只等于9999-99-99的数据，如果旧的拉链表中的end\_date不等于9999-99-99，说明已经是终态了，不需要再更新

如果dwd\_order\_info.id is null , 没关联上，说明数据状态没变，让end\_date还等于旧的end\_date

如果dwd\_order\_info.id is not null , 关联上了，说明数据状态变了，让end\_date等于当前日期-1

把查询结果插入到拉链临时表中

4）把拉链临时表覆盖到旧的拉链表中

# 第5章 即席查询数据仓库

1)Druid:是一个实时处理时序数据的OLAP数据库，因为它的索引首先按照时间分片，查询的时候也是按照时间线去路由索引。

2) Kylin:核心是Cube，Cube是一种预计算技术，基本思路是预先对数据作多维索引，查询时只扫描索引而不访问原始数据从而提速。

3)Presto:它没有使用MapReduce，大部分场景下比Hive快一个数量级，其中的关键是所有的处理都在内存中完成。

4) lmpala:基于内存运算，速度快，支持的数据源没有Presto多。

Kylin: T+1

Impala: CDH

Presto: Apache版本框架

5) Spark sQL:基于Spark平台上的一个OLAP框架，基本思路是增加机器来并行计算，从而提高查询速度。

6)ES:最大的特点是使用了倒排索引解决索引问题。根据研究，ES在数据获取和聚集用的资源比在Druid高。

7）框架选型:

(1）从超大数据的查询效率来看:

Druid > Kylin > Presto > Spark SQL

(2）从支持的数据源种类来讲:

Presto > Spark SQL > Kylin > Druid

# 第6章 项目开发经验

## 6.1 项目开发中遇到哪些问题

### 6.1.1 Hadoop

1）Hadoop集群基准测试（HDFS的读写性能、MapReduce的计算能力测试）

（1）一台服务器一般都有很多个硬盘插槽（插了几个插槽）2/3/4

如果datanode不配置datanode.data.dir多目录，每次插入一块新的硬盘都需要重启服务器。配置了即插即用

（2）增加磁盘后，保证每个目录数据均衡

开启数据均衡命令：

bin/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。

停止数据均衡命令：

bin/stop-balancer.sh

3）Hdfs参数调优**（项目中遇到的问题）**

Namenode有一个工作线程池，用来处理与datanode的心跳（报告自身的健康状况和文件恢复请求）和元数据请求 dfs.namenode.handler.count=20 \* log2(Cluster Size)

4）编辑日志存储路径dfs.namenode.edits.dir设置与镜像文件存储路径dfs.namenode.name.dir尽量分开，达到最低写入延迟（提高写入的吞吐量）

5）YARN参数调优yarn-site.xml**（项目中遇到的问题）**

（1）服务器节点上YARN可使用的物理内存总量，默认是8192（MB）

（2）单个任务可申请的最多物理内存量，默认是8192（MB）。

6）HDFS和硬盘使用控制在70%以下。

**7）Hadoop宕机（项目中遇到的问题）**

（1）如果MR造成系统宕机。此时要控制Yarn同时运行的任务数，和每个任务申请的最大内存。调整参数：yarn.scheduler.maximum-allocation-mb（单个任务可申请的最多物理内存量，默认是8192MB）

（2）如果写入文件过量造成NameNode宕机。那么调高Kafka的存储大小，控制从Kafka到HDFS的写入速度。高峰期的时候用Kafka进行缓存，高峰期过去数据同步会自动跟上。

**8）集群资源分配参数（项目中遇到的问题）**

集群有30台机器，跑mr任务的时候发现5个map任务全都分配到了同一台机器上，这个可能是由于什么原因导致的吗？

解决方案：yarn.scheduler.fair.assignmultiple 这个参数 默认是开的，需要关掉

<https://blog.csdn.net/leone911/article/details/51605172>

**9）HDFS 小文件**

**10）数据倾斜**

### 6.1.2 Flume

1）Flume内存配置为4G（flume-env.sh修改）

2）FileChannel优化

通过配置dataDirs指向多个路径，每个路径对应不同的硬盘，增大Flume吞吐量。

checkpointDir和backupCheckpointDir也尽量配置在不同硬盘对应的目录中，保证checkpoint坏掉后，可以快速使用backupCheckpointDir恢复数据

3）Sink：HDFS Sink小文件处理

这三个参数配置写入HDFS后会产生小文件，hdfs.rollInterval、hdfs.rollSize、hdfs.rollCount

**4）Ganglia监控（项目中遇到的问题）**

Ganglia监控Flume发现尝试提交的次数大于最终成功的次数

（1）增加Flume内存4-6G

（2）增加Flume台数（增加日志服务器）618/1111

### 6.1.3 Kafka

1. Kafka的吞吐量测试（测试生产速度（最快600m/s 实际20m/s）和消费速度（取决下级消费者））
2. Kafka内存默认1g，最大可以调到6G（不能超过6G）
3. Kafka数量确定：2 \* 峰值生产速度（m/s）\* 副本数 / 100 + 1 = ?
4. Kafka中的数据量计算

每天数据总量100g(1亿条) 10000万/24/60/60 = 1150条/s

平均每秒钟：1150条

低谷每秒：100条

高峰每秒钟：1150 \* 200 = 220000 条

每条日志大小： 1K左右

每秒多少数据量：1m/s 峰值20MB

1. **Kafka挂掉（项目中遇到的问题）**

磁盘满了，内存满了、断电了

（1）Flume Channel可以缓存一段时间，短期没事（memory 100 file 100万）

（2）日志服务器有30天记录，可以重写跑

1. **Kafka数据丢失（项目中遇到的问题）**

Ack:

0 发送过去就不等应答，很有可能丢数

1：leader应答，主要注重的效率，在企业中用的比较多

-1：leader和follower共同应答，可靠性高，效率低；在金融场景比较多。

1. **Kafka数据重复（项目中遇到的问题）**

**幂等性+(ack-1)+事务**

在下一级消费者中去重。（redis、SparkStreaming、hive的dwd层）

1. **Kafka消息数据积压，Kafka消费能力不足怎么处理？ （项目中遇到的问题）**

（1）如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

（2）如果是下游的数据处理不及时：提高每批次拉取的数量batchsize。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

1. **Kafka优化**

压缩

数据保存时间，默认7天，调整为3天

计算线程=cpu+1

IO线程=cpu\*2

零拷贝技术、顺序读写、分布式集群、分区

### 6.1.4 Hive

1）自定义UDF和UDTF解析和调试复杂字段

自定义UDF（extends UDF 实现evaluate方法） 解析公共字段

自定义UDTF(extends Genertic UDTF->实现三个方法init(指定返回值的名称和类型)、process(处理字段一进多出)、close方法) -> 更加灵活以及方便调试bug

2）Hive优化、数据倾斜

3）现场手写HQL 30个指标一定会

### 6.1.5 MySql

**1）MySQL之元数据备份（项目中遇到的问题）**

元数据备份（重点，如数据损坏，可能整个集群无法运行，至少要保证每日零点之后备份到其它服务器两个复本）

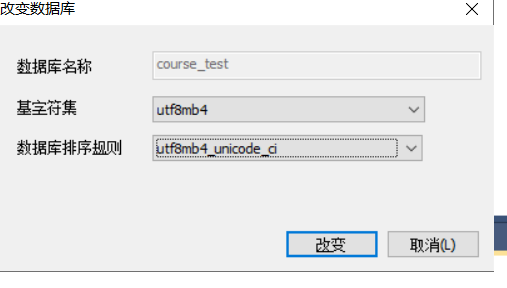
 Keepalived或者用mycat

2）Mysql utf8超过字节数问题

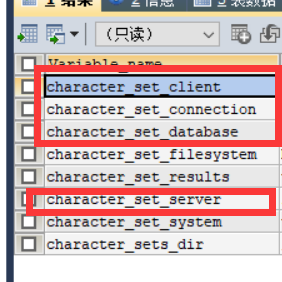
Mysql的utf8编码最多存储3个字节，当数据中存在表情号、特色符号时会占用超过3个字节数的字节，那么会出现错误 Incorrect string value: '\xF0\x9F\x91\x91\xE5\xB0...'

解决办法：将utf8修改为utf8mb4

首先修改库的基字符集和数据库排序规则



再使用 SHOW VARIABLES LIKE '%char%'; 命令查看参数



确保这几个参数的value值为utf8mb4 如果不是使用set命令修改

如：set character\_set\_server = utf8mb4;

### 6.1.6 Tez引擎优点？

Tez可以将多个有依赖的作业转换为一个作业，这样只需写一次HDFS，且中间节点较少，从而大大提升作业的计算性能。

Mr/tez/spark区别：

Mr引擎：多job串联，基于磁盘，落盘的地方比较多。虽然慢，但一定能跑出结果。一般处理，周、月、年指标。

Spark引擎：虽然在Shuffle过程中也落盘，但是并不是所有算子都需要Shuffle，尤其是多算子过程，中间过程不落盘 DAG有向无环图。 兼顾了可靠性和效率。一般处理天指标。

Tez引擎：完全基于内存。 注意：如果数据量特别大，慎重使用。容易OOM。一般用于快速出结果，数据量比较小的场景。

### 6.1.7 Sqoop

1）Sqoop数据导出Parquet**（项目中遇到的问题）**

Ads层数据用Sqoop往MySql中导入数据的时候，如果用了orc（Parquet）不能导入，需转化成text格式

（1）创建临时表，把Parquet中表数据导入到临时表，把临时表导出到目标表用于可视化

（2）Sqoop里面有参数，可以直接把Parquet转换为text

（3）ads层建表的时候就不要建Parquet表

2）Sqoop数据导出控制**（项目中遇到的问题）**

Sqoop中导入导出Null存储一致性问题：

Hive中的Null在底层是以“\N”来存储，而MySQL中的Null在底层就是Null，为了保证数据两端的一致性。在导出数据时采用--input-null-string和--input-null-non-string两个参数。导入数据时采用--null-string和--null-non-string。

3）Sqoop数据导出一致性问题**（项目中遇到的问题）**

当Sqoop导出数据到MySql时，使用4个map怎么保证数据的一致性

因为在导出数据的过程中map任务可能会失败，可以使用—staging-table –clear-staging

任务执行成功首先在tmp临时表中，然后将tmp表中的数据复制到目标表中（这个时候可以使用事务，保证事务的一致性）

4）Sqoop数据导出的时候一次执行多长时间

凌晨30分开始执行，Sqoop任务40-50分钟。取决于数据量。

### 6.1.8 Azkaban

（1）每天集群运行多少指标?

每天跑100多个指标，有活动时跑200个左右。

（2）任务挂了怎么办？运行成功或者失败都会发邮件、发钉钉、集成自动打电话**（项目中遇到的问题）**

最主要的解决方案就是重新跑。

### 6.1.9 Spark

**1）Spark OOM、数据倾斜解决（项目中遇到的问题）**



## 6.2 业务经验

### 6.2.1 ODS层采用什么压缩方式和存储格式？

1）保持数据原貌，不做任何修改

2）压缩采用LZO，压缩比是100g数据压缩完10g左右。

3）创建分区表

### 6.2.2 DWD层做了哪些事？

1）数据清洗

（1）空值去除

（2）过滤核心字段无意义的数据，比如订单表中订单id为null，支付表中支付id为空

（3）将用户行为宽表和业务表进行数据一致性处理

select case when a is null then b else a end as JZR,

...

from A

2）清洗的手段

Sql、mr、rdd、kettle、Python（项目中采用sql进行清除）

3）清洗掉多少数据算合理

1万条数据清洗掉1条。

4）脱敏

对手机号、身份证号等敏感数据脱敏

4）维度退化

对业务数据传过来的表进行维度退化和降维。（商品一级二级三级、省市县、年月日）

5）压缩LZO

6）列式存储parquet

### 6.2.3 DWS层做了哪些事？

1）DWS层有3-5张宽表（处理100-200个指标 70%以上的需求）

具体宽表名称：用户行为宽表，用户购买商品明细行为宽表，商品宽表，购物车宽表，物流宽表、登录注册、售后等。

2）哪个宽表最宽？大概有多少个字段？

最宽的是用户行为宽表。大概有60-100个字段

3）具体用户行为宽表字段名称

评论、打赏、收藏、关注--商品、关注--人、点赞、分享、好价爆料、文章发布、活跃、签到、补签卡、幸运屋、礼品、金币、电商点击、gmv

CREATE TABLE `app\_usr\_interact`(

`stat\_dt` date COMMENT '互动日期',

`user\_id` string COMMENT '用户id',

`nickname` string COMMENT '用户昵称',

`register\_date` string COMMENT '注册日期',

`register\_from` string COMMENT '注册来源',

`remark` string COMMENT '细分渠道',

`province` string COMMENT '注册省份',

`pl\_cnt` bigint COMMENT '评论次数',

`ds\_cnt` bigint COMMENT '打赏次数',

`sc\_add` bigint COMMENT '添加收藏',

`sc\_cancel` bigint COMMENT '取消收藏',

`gzg\_add` bigint COMMENT '关注商品',

`gzg\_cancel` bigint COMMENT '取消关注商品',

`gzp\_add` bigint COMMENT '关注人',

`gzp\_cancel` bigint COMMENT '取消关注人',

`buzhi\_cnt` bigint COMMENT '点不值次数',

`zhi\_cnt` bigint COMMENT '点值次数',

`zan\_cnt` bigint COMMENT '点赞次数',

`share\_cnts` bigint COMMENT '分享次数',

`bl\_cnt` bigint COMMENT '爆料数',

`fb\_cnt` bigint COMMENT '好价发布数',

`online\_cnt` bigint COMMENT '活跃次数',

`checkin\_cnt` bigint COMMENT '签到次数',

`fix\_checkin` bigint COMMENT '补签次数',

`house\_point` bigint COMMENT '幸运屋金币抽奖次数',

`house\_gold` bigint COMMENT '幸运屋积分抽奖次数',

`pack\_cnt` bigint COMMENT '礼品兑换次数',

`gold\_add` bigint COMMENT '获取金币',

`gold\_cancel` bigint COMMENT '支出金币',

`surplus\_gold` bigint COMMENT '剩余金币',

`event` bigint COMMENT '电商点击次数',

`gmv\_amount` bigint COMMENT 'gmv',

`gmv\_sales` bigint COMMENT '订单数')

PARTITIONED BY ( `dt` string)

5）商品详情 ----- 购物车 ----- 订单 ------ 付款的转换比率

5% 70% 90%

6）每天的GMV是多少，哪个商品卖的最好？每天下单量多少？

（1）100万的日活每天大概有10万人购买，平均每人消费100元，一天的GMV在1000万

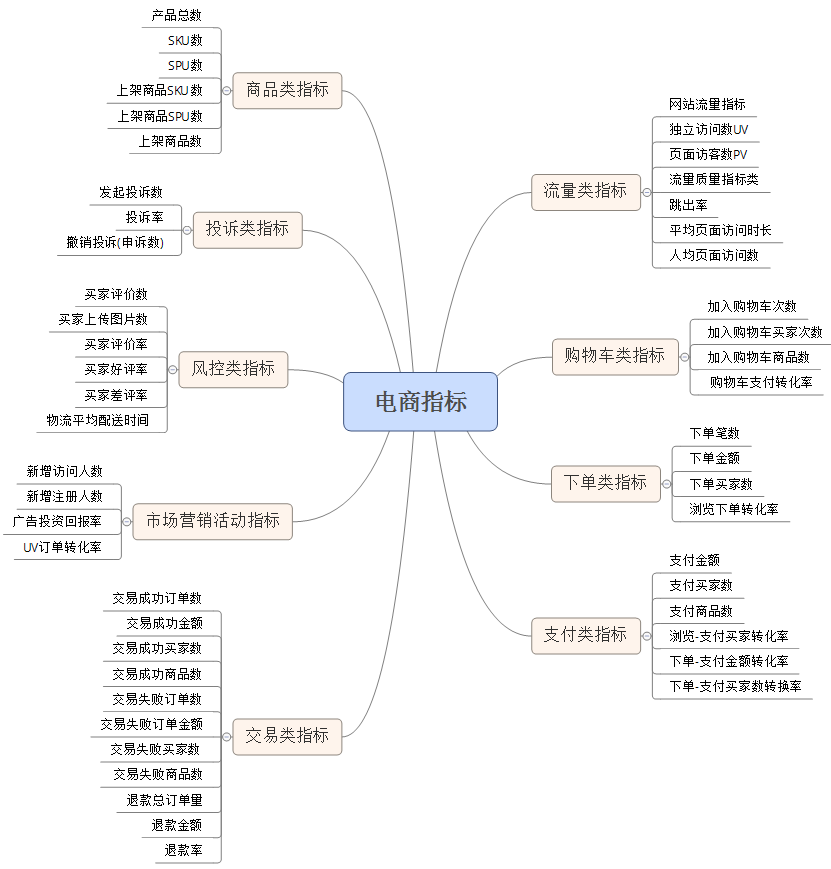
（2）面膜，每天销售5000个

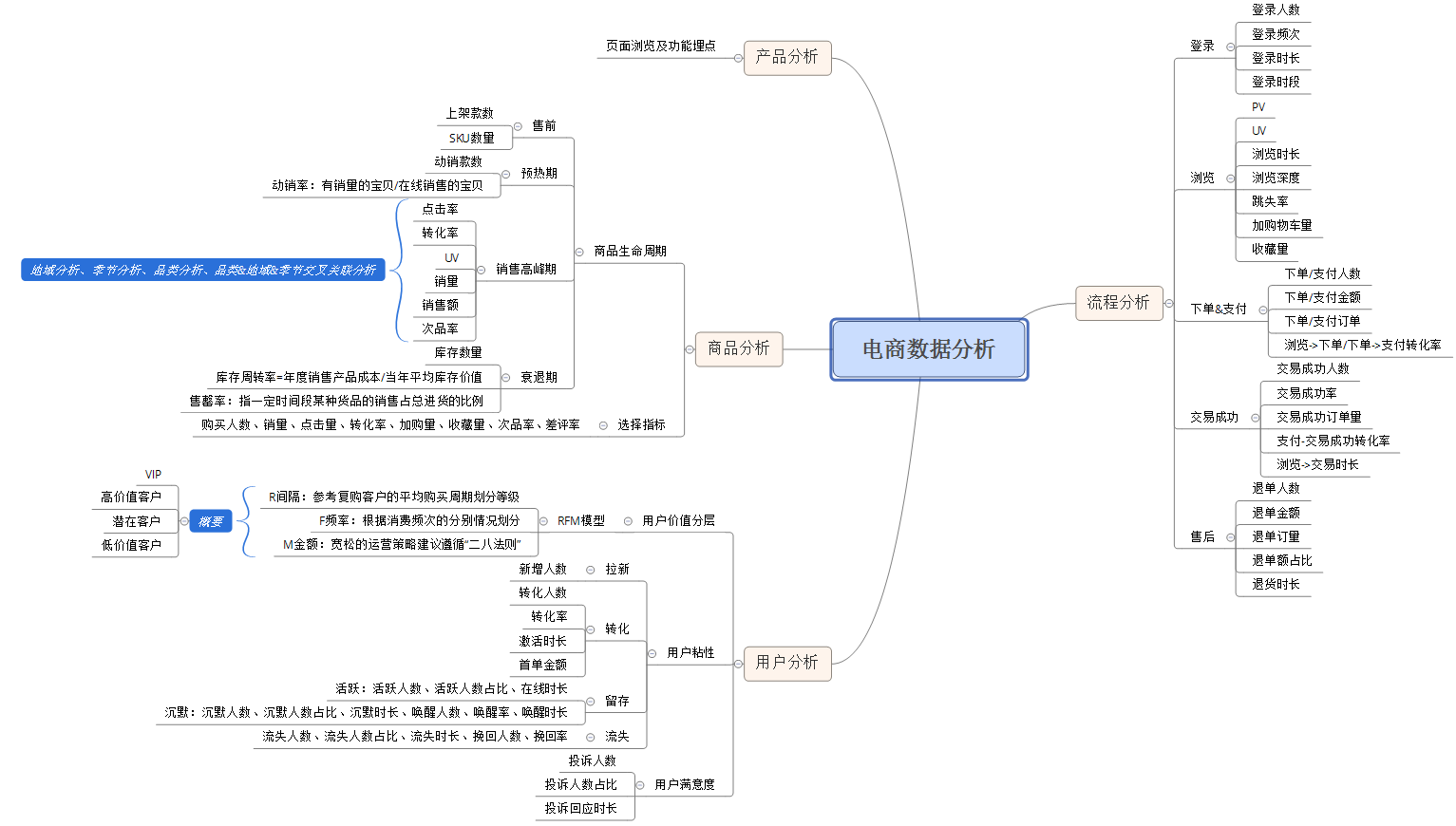
（3）每天下单量在10万左右

### 6.2.4 ADS层分析过哪些指标（一分钟至少说出30个指标）

日活、月活、周活、留存、留存率、新增（日、周、年）、转化率、流失、回流、七天内连续3天登录（点赞、收藏、评价、购买、加购、下单、活动）、连续3周（月）登录、GMV、复购率、复购率排行、点赞、评论、收藏、领优惠价人数、使用优惠价、沉默、值不值得买、退款人数、退款率 topn 热门商品

产品经理最关心的：留转G复活





日活跃用户，

月活跃用户，

各区域Top10商品统计，

季度商品品类点击率top10，

用户留存，

月APP的用户增长人数，

广告区域点击数top3，

活跃用户每天在线时长，

投诉人数占比，

沉默用户占比，

用户的新鲜度，

商品上架的sku数，

同种品类的交易额排名，

统计买家的评价率，

用户浏览时长，

统计下单的数量，

统计支付的数量，

统计退货的数量，

用户的（日活、月活、周活），

统计流失人数

日活，周活，月活，沉默用户占比，增长人数，活跃用户占比，在线时长统计，歌曲访问数，歌曲访问时长，各地区Top10歌曲统计 ,投诉人数占比，投诉回应时长，留存率，月留存率，转化率，GMV，复购vip率，vip人数，歌榜，挽回率，粉丝榜，打赏次数，打赏金额，发布歌曲榜单，歌曲热度榜单，歌手榜单，用户年龄组，vip年龄组占比，收藏数榜单，评论数

1.用户活跃数统计（日活，月活，周活）

2.某段时间的新增用户/活跃用户数

3.页面单跳转化率统计

4.活跃人数占比（占总用户比例）

5.在线时长统计（活跃用户每天在线时长）

12.统计本月的人均在线时长

6.订单产生效率（下单的次数与访问次数比）

7.页面访问时长（单个页面访问时长）

8.统计本季度付款订单

9.统计某广告的区城点击数top3

10.统计本月用户的流失人数

11.统计本月流失人数占用户人数的比例

13.统计本月APP的用户增长人数

14.统计本月的沉默用户

15.统计某时段的登录人数

16.统计本日用户登录的次数平均值

17.统计用户在某类型商品中的浏览深度（页面转跳率）

18.统计用户从下单开始到交易成功的平均时长

19.Top10热门商品的统计

20.统计下单的数量

21.统计支付的数量

22.统计退货的数量

23.统计动销率（有销量的商品/在线销售的宝贝）

24.统计支付转化率

25.统计用户的消费频率

26.统计商品上架的SKU数

27.统计同种品类的交易额排名

28.统计按下单退款排序的top10的商品

29.统计本APP的投诉人数占用户人数的比例

30.用户收藏商品

### 6.2.5 分析过最难的两个指标，现场手写

根据实际项目作答

### 6.2.6 数据仓库每天跑多少张表，大概什么时候运行，运行多久？

基本一个项目建一个库，表格个数为初始的原始数据表格加上统计结果表格的总数。（一般70-100张表格）

用户行为11张；业务数据27张表 =》ods 38 =》dwd=>32张=》dws 6张宽表=>ads=》30张 =》106张

每天0：30开始运行。=》sqoop 40-50分钟：1点20：=》 5-6个小时运行完指标

所有离线数据报表控制在8小时之内

大数据实时处理部分控制在5分钟之内。（分钟级别、秒级别）

如果是实时推荐系统，需要秒级响应

*评分标准：5分*

### 6.2.7 数仓中使用的哪种文件存储格式

常用的包括：textFile，rcFile，ORC，Parquet，一般企业里使用ORC或者Parquet，因为是列式存储，且压缩比非常高，所以相比于textFile，查询速度快，占用硬盘空间少

### 6.2.8 数仓中用到过哪些Shell脚本及具体功能

1）集群启动停止脚本（Hadoop、Flume、Kafka、Zookeeper）

2）用Sqoop实现MySQL和数仓之间的导入导出脚本

3）数仓层级之间的数据导入脚本（ods->dwd->dws->ads）。

### 6.2.9 Shell中提交了一个脚本，进程号已经不知道了，但是需要kill掉这个进程，怎么操作?

ssh $i "ps -ef | grep file-flume-kafka | grep -v grep |awk '{print \$2}' | xargs kill"

### 6.2.10 项目中用过的报表工具

Echarts（百度开源）、kibana（开源）、Tableau（功能强大的收费软件）、Superset（功能一般免费）、QuickBI（阿里云收费的离线）、DataV（阿里云收费的实时）

### 6.2.11 Hive表底层建模用的工具是什么？如何建模的？

PowerDesigner

1）选择业务过程

2）声明粒度

3）确定维度

4）确定事实

### 6.2.12 活动的话，数据量会增加多少？怎么解决？

日活增加50%，GMV增加多少。（留转G复活）情人节，促销手纸。

集群资源都留有预量。11.11，6.18，数据量过大，提前动态增加服务器。

### 6.2.13 哪张表最费时间，有没有优化

用户行为宽表，数据量过大。数据倾斜的相关优化手段。（hadoop、hive、spark）

### 6.2.14 用什么工具做权限管理

Ranger或Sentry （用户认证kerberos（张三、李四、王五）=>表级别权限（张三、李四）、字段级别权限（李四））

### 6.2.15 并发峰值多少？大概哪个时间点？

高峰期晚上7-8点。Kafka里面20m/s 2万/s 并发峰值在1-2万

### 6.2.16 测试相关

1）公司有多少台测试服务器？

测试服务器一般三台

2）测试环境什么样？

有钱的公司和生产环境电脑配置一样。

一般公司测试环境的配置是生产的一半

3）测试数据哪来的？

一部分自己写Java程序自己造（更灵活），一部分从生产环境上取一部分（更真实）。

4）如何保证写的sql正确性

需要造一些特定的测试数据，测试。

从生产环境抓取一部分数据，数据有多少你是知道的，运算完毕应该符合你的预期。

离线数据和实时数据分析的结果比较。（日活1万 实时10100），倾向取离线。

5）测试之后如何上线？

大公司：上线的时候，将脚本打包，提交git。先发邮件抄送经理和总监，运维。运维负责上线。

小公司：跟项目经理说一下，项目经理技术把关，项目经理通过了就可以上线了。风险意识。

### 6.2.17 项目实际工作流程

以下是**活跃用户**需求的整体开发流程。

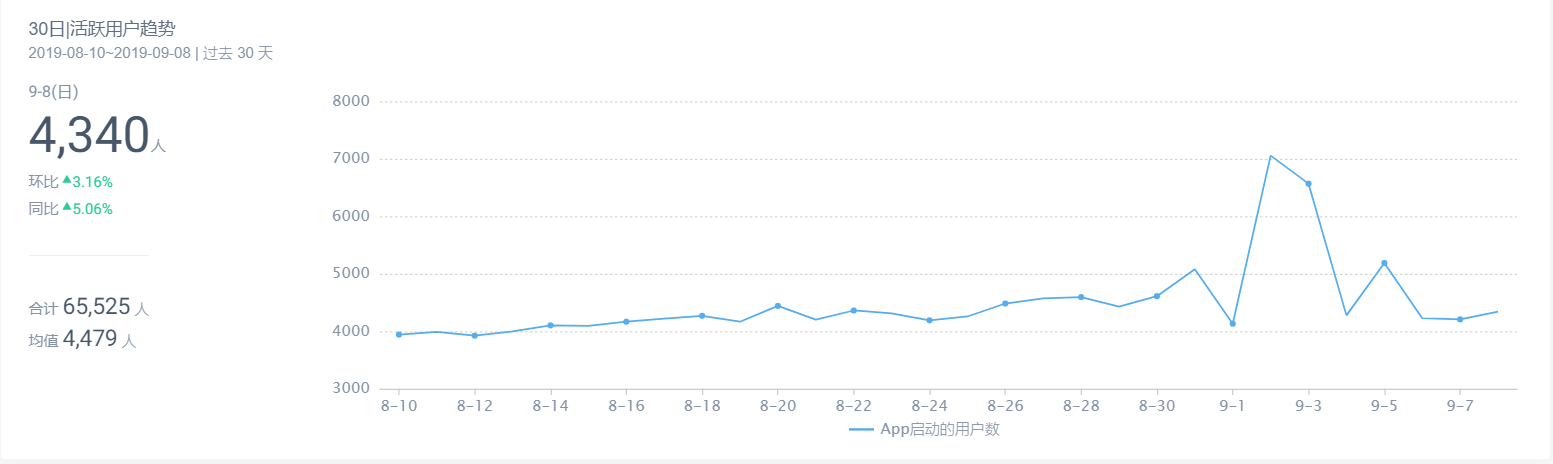
**第1步：确定指标的业务口径**

由产品经理主导，找到提出该指标的运营负责人沟通。首先要问清楚**指标是怎么定义的**，比如活跃用户是指启动过APP的用户。设备id 还是用户id。

邮件/需求文档-》不要口头

**第2步：需求评审**

由产品经理主导设计原型，对于活跃主题，我们最终要展示的是**最近n天的活跃用户数变化趋势** ，效果如下图所示。此处大数据开发工程师、后端开发工程师、前端开发工程师一同参与，一起说明整个功能的价值和详细的操作流程，确保大家理解的一致。



**第3步：大数据开发**

大数据开发工程师，通过数据同步的工具如Flume、Sqoop等将数据同步到ODS层，然后就是一层一层的通过SQL计算到DWD、DWS层，最后形成可为应用直接服务的数据填充到ADS层。

**第4步：后端开发**

后端工程师负责，为大数据工程师提供业务数据接口；

同时还负责读取ADS层分析后，写入MySQL中的数据。

**第5步：前端开发**

前端工程师负责，前端埋点。

对分析后的结果数据进行可视化展示。

**第6步：联调**

此时数据开发工程师、前端开发工程师、后端开发工程师都要参与进来。此时会要求大数据开发工程师基于历史的数据执行计算任务，大数据开发工程师承担数据准确性的校验。前后端解决用户操作的相关BUG保证不出现低级的问题完成自测。

**第7步：测试**

测试工程师对整个大数据系统进行测试。测试的手段包括，边界值、等价类等。

提交测试异常的软件有：禅道、bugzila（测试人员记录测试问题1.0，输入是什么，结果是什么，跟预期不一样->需要开发人员解释，是一个bug，下一个版本解决1.1->测试人员再测试。测试1.1ok->测试经理关闭bug）

**第8步：上线**

运维工程师会配合我们的前后端开发工程师更新最新的版本到服务器。此时产品经理要找到该指标的负责人长期跟进指标的准确性。重要的指标还要每过一个周期内部再次验证，从而保证数据的准确性。

### 6.2.18 项目中实现一个需求大概多长时间

刚入职第一个需求大概需要7天左右。

对业务熟悉后，平均一天一个需求。

影响时间的因素：测试服务器购买获取环境准备、对业务熟悉、开会讨论需求、表的权限申请、测试等。

### 6.2.19 项目在3年内迭代次数，每一个项目具体是如何迭代的。公司版本迭代多久一次，迭代到哪个版本

瀑布式开发、敏捷开发

差不多一个月会迭代一次。每月都有节日（元旦、春节、情人节、3.8妇女节、端午节、618、国庆、中秋、1111/6.1/5.1、生日）新产品、新区域

就产品或我们提出优化需求，然后评估时间。每周我们都会开会做下周计划和本周总结。（日报、周报、月报、季度报、年报）需求1周的时间，周三一定完成。周四周五（帮同事写代码、自己学习工作额外的技术）

有时候也会去预研一些新技术。Flink

5.1.2

5是大版本号：必须是重大升级

1：一般是核心模块变动

2：一般版本变化

### 6.2.20 项目开发中每天做什么事

1）新需求（活动、优化、新产品、新市场）。

2）故障分析：数仓的任何步骤出现问题，需要查看问题，比如日活，月活下降或快速上升等。

3）新技术的预言（比如flink、数仓建模、数据质量、元数据管理）

4）开会

晨会-》10做操-》讨论中午吃什么-》12点出去吃1点-》睡到2点-》3点茶歇水果

-》晚上吃啥-》吃加班餐-》开会-》晚上6点吃饭-》7点开始干活-10点-》11点

### 6.2.21 100G的数据离线数仓一套跑下来大概多少时间?

4-6小时 sqoop 0:30执行=》导完数据1:20=》4-6小时 计算任务=》8点前有报表

### 6.2.22 跑实时任务，怎么分配内存和CPU资源

128m数据对应1g内存

1个Kafka分区对应1个CPU

### 6.2.23 跑实时任务，每天数据量多少？

用户行为：实时任务用到了用户行为多少张表（10g）

业务数据：实时任务用到了业务数据多少张表(34m)

活动、风控、销售、流量

## 6.3 元数据管理（Atlas血缘系统）

<https://www.cnblogs.com/mantoudev/p/9986408.html>

## 6.4 数据质量监控（Griffin）

### 6.4.1 为什么要做数据质量监控（2019年下半年）

**1)数据不一致**

企业早期没有进行统一规划设计，大部分信息系统是逐步迭代建设的，系统建设时间长短各异，各系统数据标准也不同。企业业务系统更关注业务层面，各个业务系统均有不同的侧重点，各类数据的属性信息设置和要求不统一。另外，由于各系统的相互独立使用，无法及时同步更新相关信息等各种原因造成各系统间的数据不一致，严重影响了各系统间的数据交互和统一识别，基础数据难以共享利用，数据的深层价值也难以体现。

**2)数据不完整**

由于企业信息系统的孤立使用，各个业务系统或模块按照各自的需要录入数据，没有统一的录入工具和数据出口，业务系统不需要的信息就不录，造成同样的数据在不同的系统有不同的属性信息，数据完整性无法得到保障。

**3)数据不合规**

没有统一的数据管理平台和数据源头，数据全生命周期管理不完整，同时企业各信息系统的数据录入环节过于简单且手工参与较多，就数据本身而言，缺少是否重复、合法、对错等校验环节，导致各个信息系统的数据不够准确，格式混乱，各类数据难以集成和统一，没有质量控制导致海量数据因质量过低而难以被利用，且没有相应的数据管理流程。

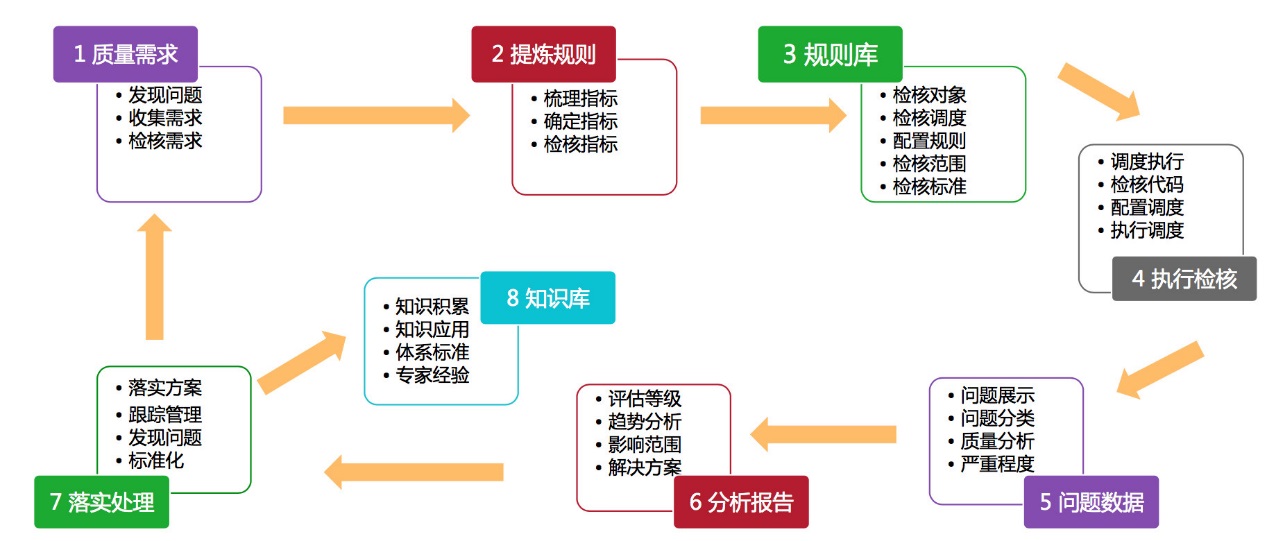
**4)数据不可控**

海量数据多头管理，缺少专门对数据管理进行监督和控制的组织。企业各单位和部门关注数据的角度不一样，缺少一个组织从全局的视角对数据进行管理，导致无法建立统一的数据管理标准、流程等，相应的数据管理制度、办法等无法得到落实。同时，企业基础数据质量考核体系也尚未建立，无法保障一系列数据标准、规范、制度、流程得到长效执行。

**5)数据冗余**

各个信息系统针对数据的标准规范不一、编码规则不一、校验标准不一，且部分业务系统针对数据的验证标准严重缺失，造成了企业顶层视角的数据出现“一物多码”、“一码多物”等现象。

### 6.4.2 建设方法



质量监管平台建设，主要包含如下8大流程步骤：

质量需求：发现数据问题；信息提报、收集需求；检核规则的需求等；

提炼规则：梳理规则指标、确定有效指标、检核指标准确度和衡量标准；

规则库构建：检核对象配置、调度配置、规则配置、检核范围确认、检核标准确定等；

执行检核：调度配置、调度执行、检核代码；

问题检核：检核问题展示、分类、质量分析、质量严重等级分类等；

分析报告：数据质量报告、质量问题趋势分析，影响度分析，解决方案达成共识；

落实处理：方案落实执行、跟踪管理、解决方案Review及标准化提炼；

知识库体系形成：知识经验总结、标准方案沉淀、知识库体系建设。

### 6.4.3 监控指标

**1）单表数据量监控**

一张表的记录数在一个已知的范围内，或者上下浮动不会超过某个阈值

* + SQL结果：var 数据量 = select count（\*）from 表 where 时间等过滤条件
  + 报警触发条件设置：如果数据量不在[数值下限, 数值上限]， 则触发报警
  + 同比增加：如果((本周的数据量 - 上周的数据量)/上周的数据量\*100)不在 [比例下线，比例上限]，则触发报警
  + 环比增加：如果((今天的数据量 - 昨天的数据量)/昨天的数据量\*100)不在 [比例下线，比例上限]，则触发报警
  + 报警触发条件设置一定要有。如果没有配置的阈值，不能做监控

日活、周活、月活、留存（日周月）、转化率（日、周、月）GMV（日、周、月）

复购率（日周月）

**2）单表空值检测**

某个字段为空的记录数在一个范围内，或者占总量的百分比在某个阈值范围内

* + 目标字段：选择要监控的字段，不能选“无”
  + SQL结果：var 异常数据量 = select count(\*) from 表 where 目标字段 is null
  + 单次检测：如果(异常数据量)不在[数值下限, 数值上限]，则触发报警

**3）单表重复值检测**

一个或多个字段是否满足某些规则

* + 目标字段：选择要监控的字段，group by 这里的字段列表后，没有重复
  + 单次检测：如果(异常数据量)不在[数值下限, 数值上限]， 则触发报警

**4）单表值域检测**

一个或多个字段没有重复记录

* + 目标字段：选择要监控的字段，支持多选
  + 检测规则：填写“目标字段”要满足的条件。其中$1表示第一个目标字段，$2表示第二个目标字段，以此类推。上图中的“检测规则”经过渲染后变为“delivery\_fee = delivery\_fee\_base+delivery\_fee\_extra”
  + 阈值配置与“空值检测”相同

**5）跨表数据量对比**

主要针对同步流程，监控两张表的数据量是否一致

* + SQL结果：count(本表) - count(关联表)
  + 阈值配置与“空值检测”相同

### 6.4.4 Griffin数据质量监控实现

<https://blog.csdn.net/An342647823/article/details/86543432>

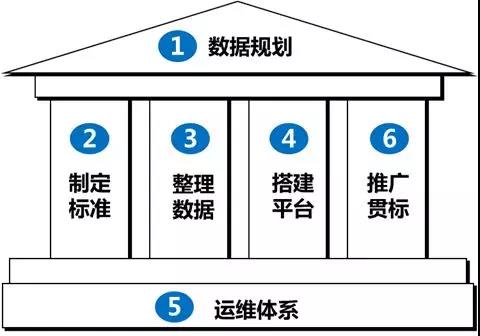
## 6.5 数据治理

包括：数据质量管理、元数据管理、权限管理（ranger sentry）。

CDH cloudmanager-》sentry； HDP ambari=>ranger

数据治理是一个复杂的系统工程，涉及到企业和单位多个领域，既要做好顶层设计，又要解决好统一标准、统一流程、统一管理体系等问题，同时也要解决好数据采集、数据清洗、数据对接和应用集成等相关问题。

数据治理实施要点主要包含**数据规划、制定数据标准、整理数据、搭建数据管理工具、构建运维体系及推广贯标**六大部分，其中数据规划是纲领、制定数据标准是基础、整理数据是过程、搭建数据管理工具是技术手段、构建运维体系是前提，推广贯标是持续保障。



首先运用方法论并结合企业实际情况，制定数据整体实施路线图。然后确定数据范围，与业务部门共同制定数据标准，标准内容包括确定分类规范、编码结构、数据模型、属性描述等。标准制定后，按照数据标准进行数据检查、数据排重、数据编码、数据加载等，建立符合数据标准和规范的数据代码库。同时应建设数据管理工具，为数据的管理提供技术支持，实现数据查询、申请、修改、审核、发布、冻结、归档等全生命周期管理。同步建立数据管理和标准管理的运维组织、管理流程、考核机制等，保证数据标准规范得到有效执行。最后统一执行数据标准规范，扩大数据标准的应用范围，实现信息系统间的互联互通及共享利用。

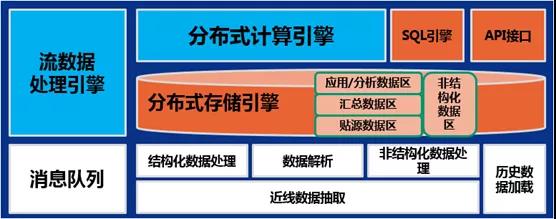
## 6.6 数据中台

<https://mp.weixin.qq.com/s/nXI0nSSOneteIClA7dming>

## 6.7 数据湖

**数据湖（**Data Lake）**是一个存储企业的各种各样原始数据的大型仓库，其中的数据可供存取、处理、分析及传输。**数据湖是以其自然格式存储的数据的系统或存储库，通常是对象blob或文件。数据湖通常是企业所有数据的单一存储，包括源系统数据的原始副本，以及用于报告、可视化、分析和机器学习等任务的转换数据。数据湖可以包括来自关系数据库（行和列）的结构化数据，半结构化数据（CSV，日志，XML，JSON），非结构化数据（电子邮件，文档，PDF）和二进制数据（图像，音频，视频）。来源：维基百科。

目前，Hadoop是最常用的部署数据湖的技术，所以很多人会觉得数据湖就是Hadoop集群。数据湖是一个概念，而Hadoop是用于实现这个概念的技术。





|  |  |
| --- | --- |
| **数据仓库** | **数据湖** |
| 主要**处理历史的、结构化的数据**，而且这些数据必须与数据仓库事先定义的模型吻合。 | 能处理所有类型的数据，如结构化数据，非结构化数据，半结构化数据等，数据的类型依赖于数据源系统的原始数据格式。非结构化数据（语音、图片、视频等） |
| 处理结构化数据，将它们或者转化为多维数据，或者转换为报表，以满足后续的高级报表及数据分析需求。 | 拥有**足够强的计算能力**用于处理和分析所有类型的数据，分析后的数据会被存储起来供用户使用。 |
| 数据仓库通常用于存储和维护长期数据，因此数据可以按需访问。 | 数据湖通常包含更多的相关的信息，这些信息有很高概率会被访问，并且能够为企业挖掘新的运营需求。 |

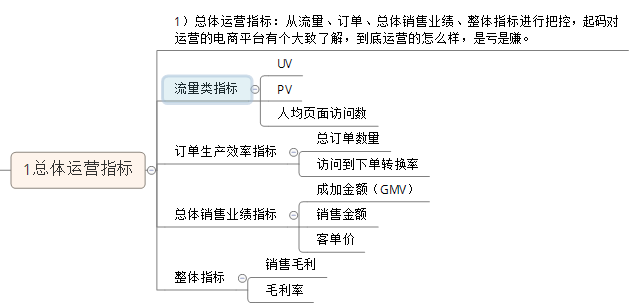
## 6.8 埋点

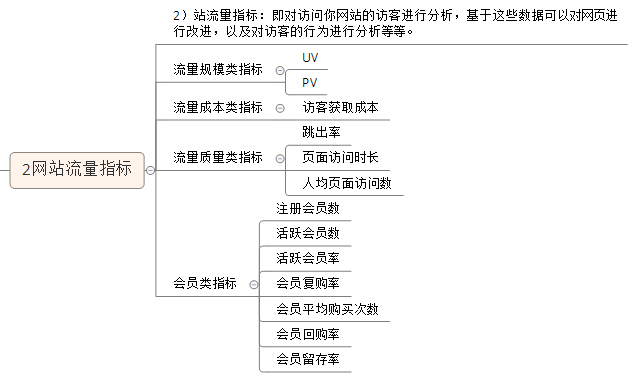
神策

<https://mp.weixin.qq.com/s/Xp3-alWF4XHvKDP9rNWCoQ>

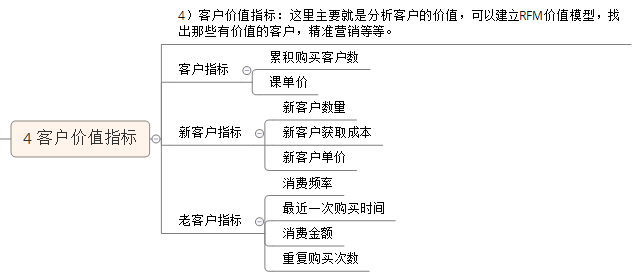
# 第7章 电商运营经验

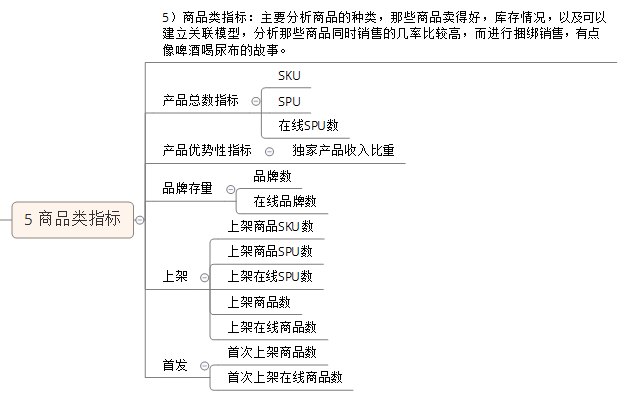
## 7.1 电商8类基本指标

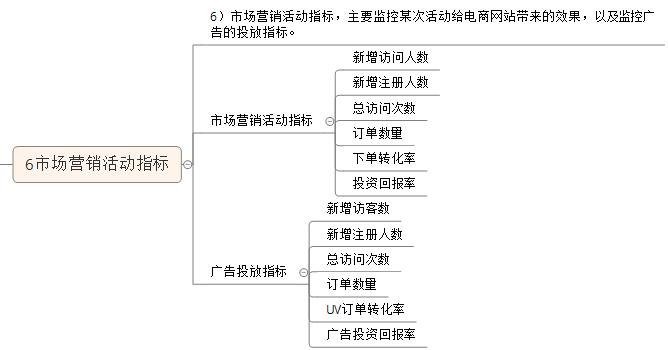


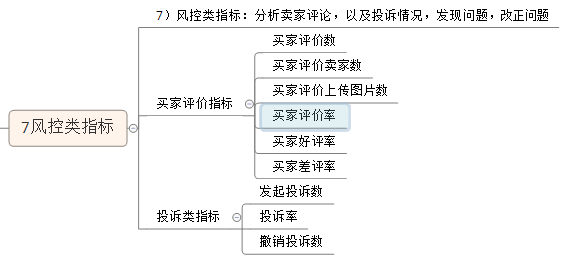




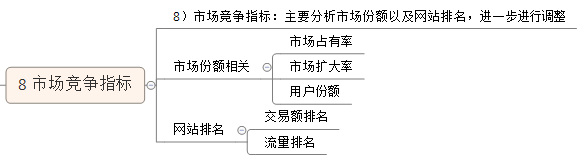




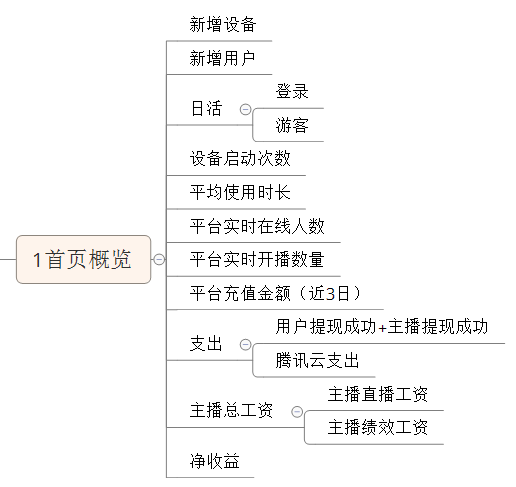


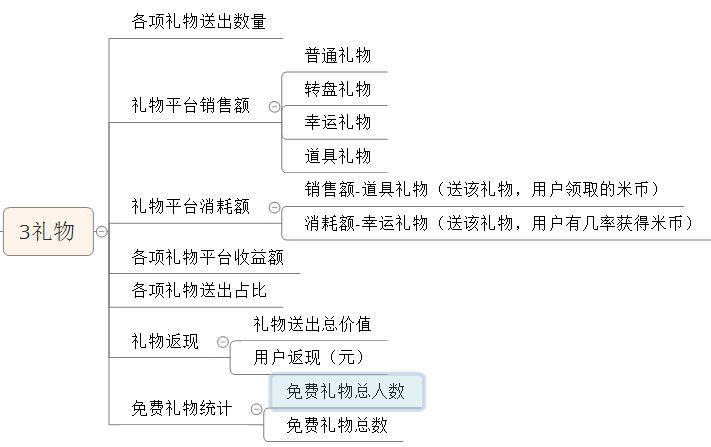


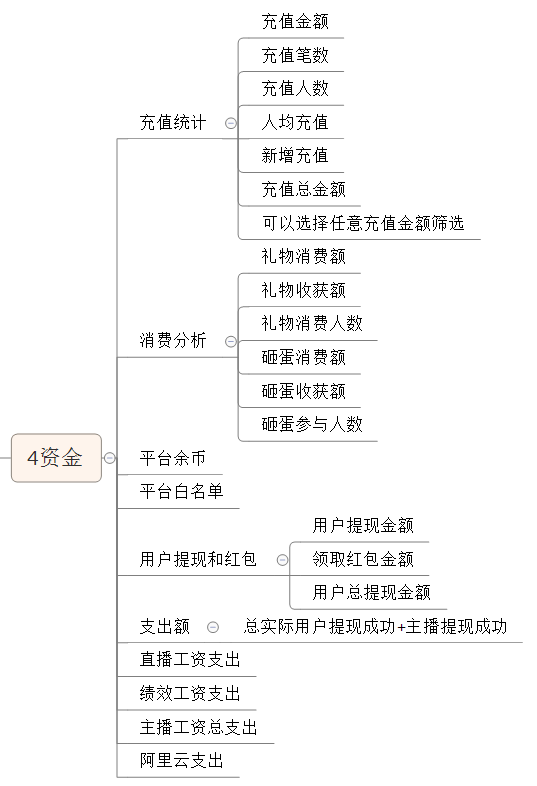
8）市场竞争指标：主要分析市场份额以及网站排名，进一步进行调整

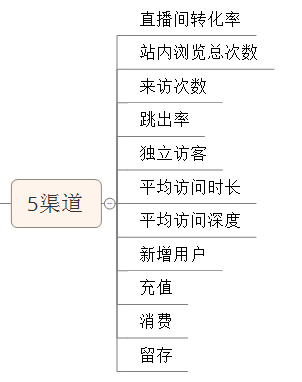


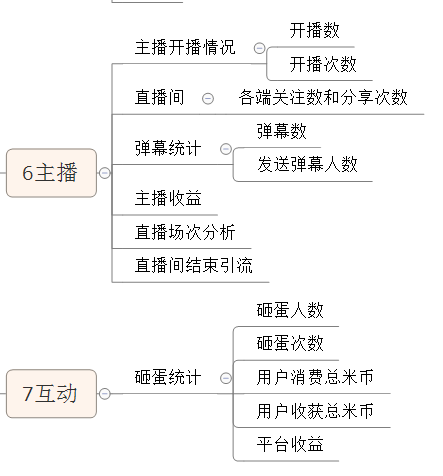
## 7.2 直播指标











# 第8章 手写代码

## 8.1 基本算法

### 8.1.1 冒泡排序

/\*\*

\* 冒泡排序 时间复杂度 O(n^2) 空间复杂度O(1)

\*/

public class BubbleSort {

public static void bubbleSort(int[] data) {

System.out.println("开始排序");

int arrayLength = data.length;

for (int i = 0; i < arrayLength - 1; i++) {

boolean flag = false;

for (int j = 0; j < arrayLength - 1 - i; j++) {

if(data[j] > data[j + 1]){

int temp = data[j + 1];

data[j + 1] = data[j];

data[j] = temp;

flag = true;

}

}

System.out.println(java.util.Arrays.toString(data));

if (!flag)

break;

}

}

public static void main(String[] args) {

int[] data = { 9, -16, 21, 23, -30, -49, 21, 30, 30 };

System.out.println("排序之前：\n" + java.util.Arrays.toString(data));

bubbleSort(data);

System.out.println("排序之后：\n" + java.util.Arrays.toString(data));

}

}

### 8.1.2 二分查找

实现代码：

/\*\*

 \* 二分查找 时间复杂度O(log2n);空间复杂度O(1)

 \*/

def binarySearch(arr:Array[Int],left:Int,right:Int,findVal:Int): Int={

  if(left>right){//递归退出条件，找不到，返回-1

    -1

  }

  val midIndex = (left+right)/2

  if (findVal < arr(midIndex)){//向左递归查找

    binarySearch(arr,left,midIndex-1,findVal)

  }else if(findVal > arr(midIndex)){//向右递归查找

    binarySearch(arr,midIndex+1,right,findVal)

  }else{//查找到，返回下标

    midIndex

  }

}

拓展需求：当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到。

代码实现如下：

/\*

{1,8, 10, 89, 1000, 1000，1234} 当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到，比如这里的 1000.

//分析

1. 返回的结果是一个可变数组 ArrayBuffer

2. 在找到结果时，向左边扫描，向右边扫描 [条件]

3. 找到结果后，就加入到ArrayBuffer

\*/

def binarySearch2(arr: Array[Int], l: Int, r: Int,

findVal: Int): ArrayBuffer[Int] = {

//找不到条件?

if (l > r) {

return ArrayBuffer()

}

val midIndex = (l + r) / 2

val midVal = arr(midIndex)

if (midVal > findVal) {

//向左进行递归查找

binarySearch2(arr, l, midIndex - 1, findVal)

} else if (midVal < findVal) { //向右进行递归查找

binarySearch2(arr, midIndex + 1, r, findVal)

} else {

println("midIndex=" + midIndex)

//定义一个可变数组

val resArr = ArrayBuffer[Int]()

//向左边扫描

var temp = midIndex - 1

breakable {

while (true) {

if (temp < 0 || arr(temp) != findVal) {

break()

}

if (arr(temp) == findVal) {

resArr.append(temp)

}

temp -= 1

}

}

//将中间这个索引加入

resArr.append(midIndex)

//向右边扫描

temp = midIndex + 1

breakable {

while (true) {

if (temp > arr.length - 1 || arr(temp) != findVal) {

break()

}

if (arr(temp) == findVal) {

resArr.append(temp)

}

temp += 1

}

}

return resArr

}

### 8.1.3 快排

代码实现：

/\*\*

\* 快排

\* 时间复杂度:平均时间复杂度为O(nlogn)

\* 空间复杂度:O(logn)，因为递归栈空间的使用问题

\*/

def quickSort(list: List[Int]): List[Int] = list match {

case Nil => Nil

case List() => List()

case head :: tail =>

val (left, right) = tail.partition(\_ < head)

quickSort(left) ::: head :: quickSort(right)

}

### 8.1.4 归并

核心思想：不断的将大的数组分成两个小数组，直到不能拆分为止，即形成了单个值。此时使用合并的排序思想对已经有序的数组进行合并，合并为一个大的数据，不断重复此过程，直到最终所有数据合并到一个数组为止。

代码实现：

/\*\*

\* 快排

\* 时间复杂度:O(nlogn)

\* 空间复杂度:O(n)

\*/

def merge(left: List[Int], right: List[Int]): List[Int] = (left, right) match {

case (Nil, \_) => right

case (\_, Nil) => left

case (x :: xTail, y :: yTail) =>

if (x <= y) x :: merge(xTail, right)

else y :: merge(left, yTail)

}

### 8.1.5 二叉树之Scala实现

1）二叉树概念

1)树有很多种，每个节点最多只能有两个子节点的一种形式称为二叉树。2)二叉树的子节点分为左节点和右节点。

3)如果该二叉树的所有叶子节点都在最后一层，并且结点总数=2^n -1 , n为层数，则我们称为满二叉树。

4)如果该二叉树的所有叶子节点都在最后一层或者倒数第二层，而且最后一层的叶子节点在左边连

续，倒数第二层的叶子节点在右边连续，我们称为完全二叉树。

2）二叉树的特点

（1）树执行查找、删除、插入的时间复杂度都是O(logN)

（2）遍历二叉树的方法包括前序、中序、后序

（3）非平衡树指的是根的左右两边的子节点的数量不一致

（4）在非空二叉树中，第i层的结点总数不超过 , i>=1；

（5）深度为h的二叉树最多有个结点(h>=1)，最少有h个结点；

（6）对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1；

3） 二叉树的Scala代码实现

定义节点以及前序、中序、后序遍历

class TreeNode(treeNo:Int){

val no = treeNo

var left:TreeNode = null

var right:TreeNode = null

//后序遍历

def postOrder():Unit={

//向左递归输出左子树

if(this.left != null){

this.left.postOrder

}

//向右递归输出右子树

if (this.right != null) {

this.right.postOrder

}

//输出当前节点值

printf("节点信息 no=%d \n",no)

}

//中序遍历

def infixOrder():Unit={

//向左递归输出左子树

if(this.left != null){

this.left.infixOrder()

}

//输出当前节点值

printf("节点信息 no=%d \n",no)

//向右递归输出右子树

if (this.right != null) {

this.right.infixOrder()

}

}

//前序遍历

def preOrder():Unit={

//输出当前节点值

printf("节点信息 no=%d \n",no)

//向左递归输出左子树

if(this.left != null){

this.left.postOrder()

}

//向右递归输出右子树

if (this.right != null) {

this.right.preOrder()

}

}

//后序遍历查找

def postOrderSearch(no:Int): TreeNode = {

//向左递归输出左子树

var resNode:TreeNode = null

if (this.left != null) {

resNode = this.left.postOrderSearch(no)

}

if (resNode != null) {

return resNode

}

if (this.right != null) {

resNode = this.right.postOrderSearch(no)

}

if (resNode != null) {

return resNode

}

println("ttt~~")

if (this.no == no) {

return this

}

resNode

}

//中序遍历查找

def infixOrderSearch(no:Int): TreeNode = {

var resNode : TreeNode = null

//先向左递归查找

if (this.left != null) {

resNode = this.left.infixOrderSearch(no)

}

if (resNode != null) {

return resNode

}

println("yyy~~")

if (no == this.no) {

return this

}

//向右递归查找

if (this.right != null) {

resNode = this.right.infixOrderSearch(no)

}

return resNode

}

//前序查找

def preOrderSearch(no:Int): TreeNode = {

if (no == this.no) {

return this

}

//向左递归查找

var resNode : TreeNode = null

if (this.left != null) {

resNode = this.left.preOrderSearch(no)

}

if (resNode != null){

return resNode

}

//向右边递归查找

if (this.right != null) {

resNode = this.right.preOrderSearch(no)

}

return resNode

}

//删除节点

//删除节点规则

//1如果删除的节点是叶子节点，则删除该节点

//2如果删除的节点是非叶子节点，则删除该子树

def delNode(no:Int): Unit = {

//首先比较当前节点的左子节点是否为要删除的节点

if (this.left != null && this.left.no == no) {

this.left = null

return

}

//比较当前节点的右子节点是否为要删除的节点

if (this.right != null && this.right.no == no) {

this.right = null

return

}

//向左递归删除

if (this.left != null) {

this.left.delNode(no)

}

//向右递归删除

if (this.right != null) {

this.right.delNode(no)

}

}

}

定义二叉树，前序、中序、后序遍历，前序、中序、后序查找，删除节点

class BinaryTree{

var root:TreeNode = null

//后序遍历

def postOrder(): Unit = {

if (root != null){

root.postOrder()

}else {

println("当前二叉树为空，不能遍历")

}

}

//中序遍历

def infixOrder(): Unit = {

if (root != null){

root.infixOrder()

}else {

println("当前二叉树为空，不能遍历")

}

}

//前序遍历

def preOrder(): Unit = {

if (root != null){

root.preOrder()

}else {

println("当前二叉树为空，不能遍历")

}

}

//后序遍历查找

def postOrderSearch(no:Int): TreeNode = {

if (root != null) {

root.postOrderSearch(no)

}else{

null

}

}

//中序遍历查找

def infixOrderSeacher(no:Int): TreeNode = {

if (root != null) {

return root.infixOrderSearch(no)

}else {

return null

}

}

//前序查找

def preOrderSearch(no:Int): TreeNode = {

if (root != null) {

return root.preOrderSearch(no)

}else{

//println("当前二叉树为空，不能查找")

return null

}

}

//删除节点

def delNode(no:Int): Unit = {

if (root != null) {

//先处理一下root是不是要删除的

if (root.no == no){

root = null

}else {

root.delNode(no)

}

}

}

## 8.2 开发代码

### 8.2.1 手写Spark-WordCount

val conf: SparkConf =

new SparkConf().setMaster("local[\*]").setAppName("WordCount")

val sc = new SparkContext(conf)

sc.textFile("/input")

.flatMap(\_.split(" "))

.map((\_, 1))

.reduceByKey(\_ + \_)

.saveAsTextFile("/output")

sc.stop()

## 8.3 手写HQL

### 8.3.1 手写HQL 第1题

表结构：uid,subject\_id,score

求：找出所有科目成绩都大于某一学科平均成绩的学生

数据集如下

1001 01 90

1001 02 90

1001 03 90

1002 01 85

1002 02 85

1002 03 70

1003 01 70

1003 02 70

1003 03 85

1）建表语句

create table score(

uid string,

subject\_id string,

score int)

row format delimited fields terminated by '\t';

2）求出每个学科平均成绩

select

uid,

score,

avg(score) over(partition by subject\_id) avg\_score

from

score;t1

3）根据是否大于平均成绩记录flag，大于则记为0否则记为1

select

uid,

if(score>avg\_score,0,1) flag

from

t1;t2

4）根据学生id进行分组统计flag的和，和为0则是所有学科都大于平均成绩

select

uid

from

t2

group by

uid

having

sum(flag)=0;

5）最终SQL

select

uid

from

(select

uid,

if(score>avg\_score,0,1) flag

from

(select

uid,

score,

avg(score) over(partition by subject\_id) avg\_score

from

score)t1)t2

group by

uid

having

sum(flag)=0;

### 8.3.2 手写HQL 第2题

我们有如下的用户访问数据

|  |  |  |
| --- | --- | --- |
| userId | visitDate | visitCount |
| u01 | 2017/1/21 | 5 |
| u02 | 2017/1/23 | 6 |
| u03 | 2017/1/22 | 8 |
| u04 | 2017/1/20 | 3 |
| u01 | 2017/1/23 | 6 |
| u01 | 2017/2/21 | 8 |
| U02 | 2017/1/23 | 6 |
| U01 | 2017/2/22 | 4 |

要求使用SQL统计出每个用户的累积访问次数，如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 用户id | 月份 | 小计 | 累积 |
| u01 | 2017-01 | 11 | 11 |
| u01 | 2017-02 | 12 | 23 |
| u02 | 2017-01 | 12 | 12 |
| u03 | 2017-01 | 8 | 8 |
| u04 | 2017-01 | 3 | 3 |

数据集

u01 2017/1/21 5

u02 2017/1/23 6

u03 2017/1/22 8

u04 2017/1/20 3

u01 2017/1/23 6

u01 2017/2/21 8

u02 2017/1/23 6

u01 2017/2/22 4

1）创建表

create table action

(userId string,

visitDate string,

visitCount int)

row format delimited fields terminated by "\t";

2）修改数据格式

select

userId,

date\_format(regexp\_replace(visitDate,'/','-'),'yyyy-MM') mn,

visitCount

from

action;t1

3）计算每人单月访问量

select

userId,

mn,

sum(visitCount) mn\_count

from

t1

group by

userId,mn;t2

4）按月累计访问量

select

userId,

mn,

mn\_count,

sum(mn\_count) over(partition by userId order by mn)

from t2;

5）最终SQL

select

userId,

mn,

mn\_count,

sum(mn\_count) over(partition by userId order by mn)

from

( select

userId,

mn,

sum(visitCount) mn\_count

from

(select

userId,

date\_format(regexp\_replace(visitDate,'/','-'),'yyyy-MM') mn,

visitCount

from

action)t1

group by userId,mn)t2;

### 8.3.3 手写HQL 第3题

有50W个京东店铺，每个顾客访客访问任何一个店铺的任何一个商品时都会产生一条访问日志，访问日志存储的表名为Visit，访客的用户id为user\_id，被访问的店铺名称为shop，请统计：

1）每个店铺的UV（访客数）

2）每个店铺访问次数top3的访客信息。输出店铺名称、访客id、访问次数

数据集

u1 a

u2 b

u1 b

u1 a

u3 c

u4 b

u1 a

u2 c

u5 b

u4 b

u6 c

u2 c

u1 b

u2 a

u2 a

u3 a

u5 a

u5 a

u5 a

1）建表

create table visit(user\_id string,shop string) row format delimited fields terminated by '\t';

2）每个店铺的UV（访客数）

select shop,count(distinct user\_id) from visit group by shop;

3）每个店铺访问次数top3的访客信息。输出店铺名称、访客id、访问次数

（1）查询每个店铺被每个用户访问次数

select shop,user\_id,count(\*) ct

from visit

group by shop,user\_id;t1

（2）计算每个店铺被用户访问次数排名

select shop,user\_id,ct,rank() over(partition by shop order by ct) rk

from t1;t2

（3）取每个店铺排名前3的

select shop,user\_id,ct

from t2

where rk<=3;

（4）最终SQL

select

shop,

user\_id,

ct

from

(select

shop,

user\_id,

ct,

rank() over(partition by shop order by ct) rk

from

(select

shop,

user\_id,

count(\*) ct

from visit

group by

shop,

user\_id)t1

)t2

where rk<=3;

### 8.3.4 手写HQL 第4题

已知一个表STG.ORDER，有如下字段:Date，Order\_id，User\_id，amount。请给出sql进行统计:数据样例:2017-01-01,10029028,1000003251,33.57。

1）给出 2017年每个月的订单数、用户数、总成交金额。

2）给出2017年11月的新客数(指在11月才有第一笔订单)

建表

create table order\_tab(dt string,order\_id string,user\_id string,amount decimal(10,2)) row format delimited fields terminated by '\t';

1）给出 2017年每个月的订单数、用户数、总成交金额。

select

date\_format(dt,'yyyy-MM'),

count(order\_id),

count(distinct user\_id),

sum(amount)

from

order\_tab

where

date\_format(dt,'yyyy')='2017'

group by

date\_format(dt,'yyyy-MM');

2）给出2017年11月的新客数(指在11月才有第一笔订单)

select

count(user\_id)

from

order\_tab

group by

user\_id

having

date\_format(min(dt),'yyyy-MM')='2017-11';

### 8.3.5 手写HQL 第5题

有日志如下，请写出代码求得所有用户和活跃用户的总数及平均年龄。（活跃用户指连续两天都有访问记录的用户）日期 用户 年龄

数据集

2019-02-11,test\_1,23

2019-02-11,test\_2,19

2019-02-11,test\_3,39

2019-02-11,test\_1,23

2019-02-11,test\_3,39

2019-02-11,test\_1,23

2019-02-12,test\_2,19

2019-02-13,test\_1,23

2019-02-15,test\_2,19

2019-02-16,test\_2,19

1）建表

create table user\_age(dt string,user\_id string,age int)row format delimited fields terminated by ',';

2）按照日期以及用户分组，按照日期排序并给出排名

select

dt,

user\_id,

min(age) age,

rank() over(partition by user\_id order by dt) rk

from

user\_age

group by

dt,user\_id;t1

3）计算日期及排名的差值

select

user\_id,

age,

date\_sub(dt,rk) flag

from

t1;t2

4）过滤出差值大于等于2的，即为连续两天活跃的用户

select

user\_id,

min(age) age

from

t2

group by

user\_id,flag

having

count(\*)>=2;t3

5）对数据进行去重处理（一个用户可以在两个不同的时间点连续登录），例如：a用户在1月10号1月11号以及1月20号和1月21号4天登录。

select

user\_id,

min(age) age

from

t3

group by

user\_id;t4

6）计算活跃用户（两天连续有访问）的人数以及平均年龄

select

count(\*) ct,

cast(sum(age)/count(\*) as decimal(10,2))

from t4;

7）对全量数据集进行按照用户去重

select

user\_id,

min(age) age

from

user\_age

group by

user\_id;t5

8）计算所有用户的数量以及平均年龄

select

count(\*) user\_count,

cast((sum(age)/count(\*)) as decimal(10,1))

from

t5;

9）将第5步以及第7步两个数据集进行union all操作

select

0 user\_total\_count,

0 user\_total\_avg\_age,

count(\*) twice\_count,

cast(sum(age)/count(\*) as decimal(10,2)) twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

(select

user\_id,

min(age) age

from

(

select

user\_id,

age,

date\_sub(dt,rk) flag

from

(

select

dt,

user\_id,

min(age) age,

rank() over(partition by user\_id order by dt) rk

from

user\_age

group by

dt,user\_id

)t1

)t2

group by

user\_id,flag

having

count(\*)>=2)t3

group by

user\_id

)t4

union all

select

count(\*) user\_total\_count,

cast((sum(age)/count(\*)) as decimal(10,1)),

0 twice\_count,

0 twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

user\_age

group by

user\_id

)t5;t6

10）求和并拼接为最终SQL

select

sum(user\_total\_count),

sum(user\_total\_avg\_age),

sum(twice\_count),

sum(twice\_count\_avg\_age)

from

(select

0 user\_total\_count,

0 user\_total\_avg\_age,

count(\*) twice\_count,

cast(sum(age)/count(\*) as decimal(10,2)) twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

(select

user\_id,

min(age) age

from

(

select

user\_id,

age,

date\_sub(dt,rk) flag

from

(

select

dt,

user\_id,

min(age) age,

rank() over(partition by user\_id order by dt) rk

from

user\_age

group by

dt,user\_id

)t1

)t2

group by

user\_id,flag

having

count(\*)>=2)t3

group by

user\_id

)t4

union all

select

count(\*) user\_total\_count,

cast((sum(age)/count(\*)) as decimal(10,1)),

0 twice\_count,

0 twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

user\_age

group by

user\_id

)t5)t6;

### 8.3.6 手写HQL 第6题

请用sql写出所有用户中在今年10月份第一次购买商品的金额，表ordertable字段（购买用户：userid，金额：money，购买时间：paymenttime(格式：2017-10-01)，订单id：orderid）

1）建表

create table ordertable(

userid string,

money int,

paymenttime string,

orderid string)

row format delimited fields terminated by '\t';

2）查询出

select

userid,

min(paymenttime) paymenttime

from

ordertable

where

date\_format(paymenttime,'yyyy-MM')='2017-10'

group by

userid;t1

select

t1.userid,

t1.paymenttime,

od.money

from

t1

join

ordertable od

on

t1.userid=od.userid

and

t1.paymenttime=od.paymenttime;

select

t1.userid,

t1.paymenttime,

od.money

from

(select

userid,

min(paymenttime) paymenttime

from

ordertable

where

date\_format(paymenttime,'yyyy-MM')='2017-10'

group by

userid)t1

join

ordertable od

on

t1.userid=od.userid

and

t1.paymenttime=od.paymenttime;

### 8.3.7 手写HQL 第7题

有一个线上服务器访问日志格式如下（用sql答题）

时间 接口 ip地址

2016-11-09 11：22：05 /api/user/login 110.23.5.33

2016-11-09 11：23：10 /api/user/detail 57.3.2.16

.....

2016-11-09 23：59：40 /api/user/login 200.6.5.166

求11月9号下午14点（14-15点），访问api/user/login接口的top10的ip地址

数据集

2016-11-09 14:22:05 /api/user/login 110.23.5.33

2016-11-09 11:23:10 /api/user/detail 57.3.2.16

2016-11-09 14:59:40 /api/user/login 200.6.5.166

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 11:23:10 /api/user/detail 57.3.2.16

2016-11-09 23:59:40 /api/user/login 200.6.5.166

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 11:23:10 /api/user/detail 57.3.2.16

2016-11-09 23:59:40 /api/user/login 200.6.5.166

2016-11-09 14:22:05 /api/user/login 110.23.5.35

2016-11-09 14:23:10 /api/user/detail 57.3.2.16

2016-11-09 23:59:40 /api/user/login 200.6.5.166

2016-11-09 14:59:40 /api/user/login 200.6.5.166

2016-11-09 14:59:40 /api/user/login 200.6.5.166

1）建表

create table ip(

time string,

interface string,

ip string)

row format delimited fields terminated by '\t';

2）最终SQL

select

ip,

interface,

count(\*) ct

from

ip

where

date\_format(time,'yyyy-MM-dd HH')>='2016-11-09 14'

and

date\_format(time,'yyyy-MM-dd HH')<='2016-11-09 15'

and

interface='/api/user/login'

group by

ip,interface

order by

ct desc

limit 2;t1

### 8.3.8 手写SQL 第8题

有一个账号表如下，请写出SQL语句，查询各自区组的money排名前十的账号（分组取前10）

1）建表（MySQL）

CREATE TABLE `account`

( `dist\_id` int（11）DEFAULT NULL COMMENT '区组id',

`account` varchar（100）DEFAULT NULL COMMENT '账号',

`gold` int（11）DEFAULT 0 COMMENT '金币'）;

2）最终SQL

select

\*

from

account as a

where

(select

count(distinct(a1.gold))

from

account as a1

where

a1.dist\_id=a.dist\_id

and

a1.gold>a.gold)<3;

### 8.3.9 手写HQL 第9题

1）有三张表分别为会员表（member）销售表（sale）退货表（regoods）

（1）会员表有字段memberid（会员id，主键）credits（积分）；

（2）销售表有字段memberid（会员id，外键）购买金额（MNAccount）；

（3）退货表中有字段memberid（会员id，外键）退货金额（RMNAccount）。

2）业务说明

（1）销售表中的销售记录可以是会员购买，也可以是非会员购买。（即销售表中的memberid可以为空）；

（2）销售表中的一个会员可以有多条购买记录；

（3）退货表中的退货记录可以是会员，也可是非会员；

（4）一个会员可以有一条或多条退货记录。

查询需求：分组查出销售表中所有会员购买金额，同时分组查出退货表中所有会员的退货金额，把会员id相同的购买金额-退款金额得到的结果更新到表会员表中对应会员的积分字段（credits）

数据集

sale

1001 50.3

1002 56.5

1003 235

1001 23.6

1005 56.2

25.6

33.5

regoods

1001 20.1

1002 23.6

1001 10.1

23.5

10.2

1005 0.8

1）建表

create table member(memberid string,credits double) row format delimited fields terminated by '\t';

create table sale(memberid string,MNAccount double) row format delimited fields terminated by '\t';

create table regoods(memberid string,RMNAccount double) row format delimited fields terminated by '\t';

2）最终SQL

insert into table member

select

t1.memberid,

MNAccount-RMNAccount

from

(select

memberid,

sum(MNAccount) MNAccount

from

sale

where

memberid!=''

group by

memberid

)t1

join

(select

memberid,

sum(RMNAccount) RMNAccount

from

regoods

where

memberid!=''

group by

memberid

)t2

on

t1.memberid=t2.memberid;

### 8.3.10 手写HQL 第10题

1.用一条SQL语句查询出每门课都大于80分的学生姓名

name   kecheng   fenshu

张三    语文    81

张三    数学    75

李四    语文    76

李四    数学    90

王五    语文    81

王五    数学    100

王五    英语    90

A: select distinct name from table where name not in (select distinct name from table where fenshu<=80)

B：select name from table group by name having min(fenshu)>80

2. 学生表 如下:  
自动编号   学号  姓名 课程编号 课程名称 分数  
1     2005001 张三  0001   数学   69  
2     2005002 李四  0001   数学   89  
3     2005001 张三  0001   数学   69  
删除除了自动编号不同, 其他都相同的学生冗余信息  
  
A: delete tablename where 自动编号 not in(select min(自动编号) from tablename group by学号, 姓名, 课程编号, 课程名称, 分数)

3.一个叫team的表，里面只有一个字段name,一共有4条纪录，分别是a,b,c,d,对应四个球队，现在四个球队进行比赛，用一条sql语句显示所有可能的比赛组合.

答：select a.name, b.name  
from team a, team b  
where a.name < b.name

4.面试题：怎么把这样一个  
year   month amount  
1991   1     1.1  
1991   2     1.2  
1991   3     1.3  
1991   4     1.4  
1992   1     2.1  
1992   2     2.2  
1992   3     2.3  
1992   4     2.4  
查成这样一个结果  
year m1  m2  m3 m4  
1991 1.1 1.2 1.3 1.4  
1992 2.1 2.2 2.3 2.4   
  
答案  
select year,   
(select amount from aaa m where month=1 and m.year=aaa.year) as m1,  
(select amount from aaa m where month=2 and m.year=aaa.year) as m2,  
(select amount from aaa m where month=3 and m.year=aaa.year) as m3,  
(select amount from aaa m where month=4 and m.year=aaa.year) as m4  
from aaa group by year

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*5.说明：复制表(只复制结构,源表名：a新表名：b)   
  
SQL: select \* into b from a where 1<>1 (where1=1，拷贝表结构和数据内容)  
ORACLE:create table b

As

Select \* from a where 1=2

[<>（不等于）(SQL Server Compact)

比较两个表达式。 当使用此运算符比较非空表达式时，如果左操作数不等于右操作数，则结果为 TRUE。 否则，结果为 FALSE。]

6.

原表:  
courseid coursename score  
-------------------------------------  
1 java 70  
2 oracle 90  
3 xml 40  
4 jsp 30  
5 servlet 80  
-------------------------------------  
为了便于阅读,查询此表后的结果显式如下(及格分数为60):  
courseid coursename score mark  
---------------------------------------------------  
1 java 70 pass  
2 oracle 90 pass  
3 xml 40 fail  
4 jsp 30 fail  
5 servlet 80 pass  
---------------------------------------------------  
写出此查询语句  
select courseid, coursename ,score ,if(score>=60, "pass","fail") as mark from course

7.表名：购物信息

购物人 商品名称 数量

A 甲 2

B 乙 4

C 丙 1

A 丁 2

B 丙 5

……

给出所有购入商品为两种或两种以上的购物人记录

答：select \* from 购物信息 where 购物人 in (select 购物人 from 购物信息 group by 购物人 having count(\*) >= 2);

8.

info 表

date result

2005-05-09 win

2005-05-09 lose

2005-05-09 lose

2005-05-09 lose

2005-05-10 win

2005-05-10 lose

2005-05-10 lose

如果要生成下列结果, 该如何写sql语句?

　　 win lose

2005-05-09 2 2

2005-05-10 1 2

答案：

(1) select date, sum(case when result = "win" then 1 else 0 end) as "win", sum(case when result = "lose" then 1 else 0 end) as "lose" from info group by date;

(2) select a.date, a.result as win, b.result as lose

　　from

　　(select date, count(result) as result from info where result = "win" group by date) as a

　　join

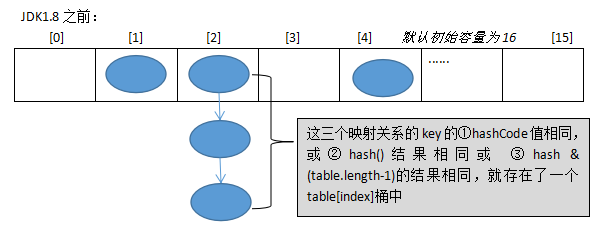
　　(select date, count(result) as result from info where result = "lose" group by date) as b

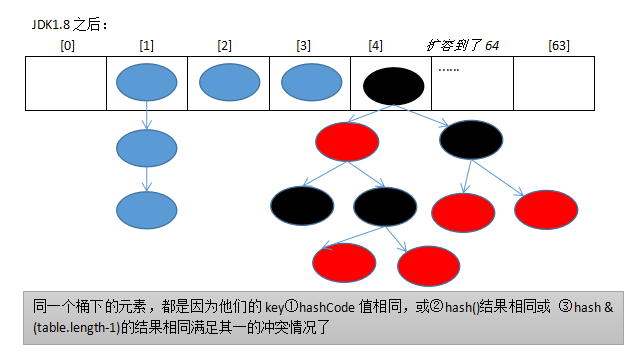
on a.date = b.date;

# 第9章 JavaSE

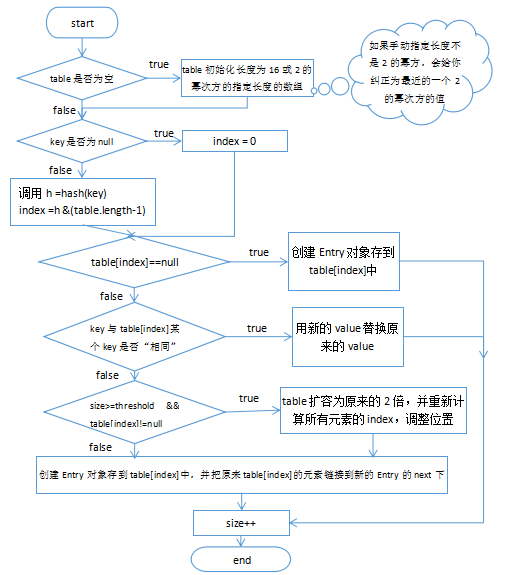
## 9.1 HhashMap底层源码，数据结构

hashMap的底层结构在jdk1.7中由数组+链表实现，在jdk1.8中由数组+链表+红黑树实现，以数组+链表的结构为例。

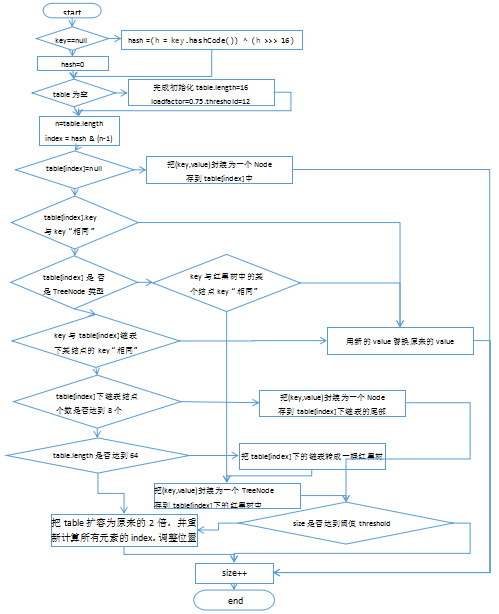




**JDK1.8之前Put方法：**



**JDK1.8之后Put方法：**



## 9.2 Java自带哪几种线程池？

**1）newCachedThreadPool**

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。这种类型的线程池特点是：

工作线程的创建数量几乎没有限制（其实也有限制的，数目为Interger. MAX\_VALUE）, 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间（默认为1分钟），则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

**2）newFixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

**3）newSingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序（FIFO, LIFO, 优先级）执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

**4）newScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。延迟3秒执行。

## 9.3 HashMap和HashTable区别

1. 线程安全性不同

HashMap是线程不安全的，HashTable是线程安全的，其中的方法是Synchronize的，在多线程并发的情况下，可以直接使用HashTabl，但是使用HashMap时必须自己增加同步处理。

1. 是否提供contains方法

HashMap只有containsValue和containsKey方法；HashTable有contains、containsKey和containsValue三个方法，其中contains和containsValue方法功能相同。

1. key和value是否允许null值

Hashtable中，key和value都不允许出现null值。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。

1. 数组初始化和扩容机制

 HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

 Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

## 9.4 TreeSet和HashSet区别

HashSet是采用hash表来实现的。其中的元素没有按顺序排列，add()、remove()以及contains()等方法都是复杂度为O(1)的方法。

TreeSet是采用树结构实现（红黑树算法）。元素是按顺序进行排列，但是add()、remove()以及contains()等方法都是复杂度为O(log (n))的方法。它还提供了一些方法来处理排序的set，如first()，last()，headSet()，tailSet()等等。

## 9.5 String buffer和String build区别

1、StringBuffer与StringBuilder中的方法和功能完全是等价的。

2、只是StringBuffer中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而StringBuilder没有这个修饰，可以被认为是线程不安全的。

3、在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全而StringBuffer则每次都需要判断锁，效率相对更低

## 9.6 Final、Finally、Finalize

final：修饰符（关键字）有三种用法：修饰类、变量和方法。修饰类时，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。修饰变量时，该变量使用中不被改变，必须在声明时给定初值，在引用中只能读取不可修改，即为常量。修饰方法时，也同样只能使用，不能在子类中被重写。

finally：通常放在try…catch的后面构造最终执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。

finalize：Object类中定义的方法，Java中允许使用finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize() 方法可以整理系统资源或者执行其他清理工作。

## 9.7 ==和Equals区别

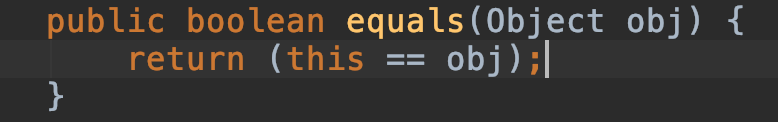
 == : 如果比较的是基本数据类型，那么比较的是变量的值

如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

 equals:如果没重写equals方法比较的是两个对象的地址值。

 如果重写了equals方法后我们往往比较的是对象中的属性的内容

equals方法是从Object类中继承的，默认的实现就是使用==



# 第10章 Redis

## 10.1 缓存穿透、缓存雪崩、缓存击穿

1）缓存穿透是指查询一个一定不存在的数据。由于缓存命不中时会去查询数据库，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决方案：

1. 是将空对象也缓存起来，并给它设置一个很短的过期时间，最长不超过5分钟

② 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力

2）如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，就会造成缓存雪崩。

解决方案：

尽量让失效的时间点不分布在同一个时间点

3）缓存击穿，是指一个key非常热点，在不停的扛着大并发，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

解决方案：

可以设置key永不过期

## 10.2 哨兵模式

主从复制中反客为主的自动版，如果主机Down掉，哨兵会从从机中选择一台作为主机，并将它设置为其他从机的主机，而且如果原来的主机再次启动的话也会成为从机。

## 10.3 数据类型

|  |  |
| --- | --- |
| string | 字符串 |
| list | 可以重复的集合 |
| set | 不可以重复的集合 |
| hash | 类似于Map<String,String> |
| zset(sorted set） | 带分数的set |

## 10.4 持久化

**1）RDB持久化：**

1. **在指定的时间间隔内持久化**
2. **服务shutdown会自动持久化**

**③ 输入bgsave也会持久化**

**2）AOF : 以日志形式记录每个更新操作**

Redis重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。

**保存策略：**

推荐（并且也是默认）的措施为每秒持久化一次，这种策略可以兼顾速度和安全性。

**缺点：**

1 比起RDB占用更多的磁盘空间

2 恢复备份速度要慢

3 每次读写都同步的话，有一定的性能压力

4 存在个别Bug，造成恢复不能

**选择策略：**

官方推荐：

如果对数据不敏感，可以选单独用RDB；不建议单独用AOF，因为可能出现Bug;如果只是做纯内存缓存，可以都不用

## 11.5 悲观锁

执行操作前假设当前的操作肯定（或有很大几率）会被打断（悲观）。基于这个假设，我们在做操作前就会把相关资源锁定，不允许自己执行期间有其他操作干扰。

## 11.6 乐观锁

执行操作前假设当前操作不会被打断（乐观）。基于这个假设，我们在做操作前不会锁定资源，万一发生了其他操作的干扰，那么本次操作将被放弃。Redis使用的就是乐观锁。

# 第11章 MySql

## 11.1 MyISAM与InnoDB的区别

|  |  |  |
| --- | --- | --- |
| **对比项** | **MyISAM** | **InnoDB** |
| **外键** | **不支持** | **支持** |
| **事务** | **不支持** | **支持** |
| **行表锁** | **表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作** | **行锁,操作时只锁某一行，不对其它行有影响，**  **适合高并发的操作** |
| **缓存** | **只缓存索引，不缓存真实数据** | **不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响** |

## 11.2 索引优化

**数据结构：B+Tree**

一般来说能够达到range就可以算是优化了 idx name\_deptId

**口诀（两个法则加6种索引失效的情况）**

全值匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

LIKE百分写最右，覆盖索引不写\*；

不等空值还有OR，索引影响要注意；

VAR引号不可丢，SQL优化有诀窍。

## 11.3 b-tree和b+tree的区别

1) B-树的关键字、索引和记录是放在一起的， B+树的非叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。

2) 在B-树中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而B+树中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。

## 11.4 redis是单线程的，为什么那么快

1)完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。

2)数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的

3)采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

4)使用多路I/O复用模型，非阻塞IO

5)使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

## 11.5 MySQL的事务

**一、事务的基本要素（ACID）**

1、原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位

2、一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如A向B转账，不可能A扣了钱，B却没收到。

3、隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

4、持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

**二、事务的并发问题**

1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致

3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

**三、MySQL事务隔离级别**

事务隔离级别 脏读 不可重复读 幻读

读未提交（read-uncommitted） 是 是 是

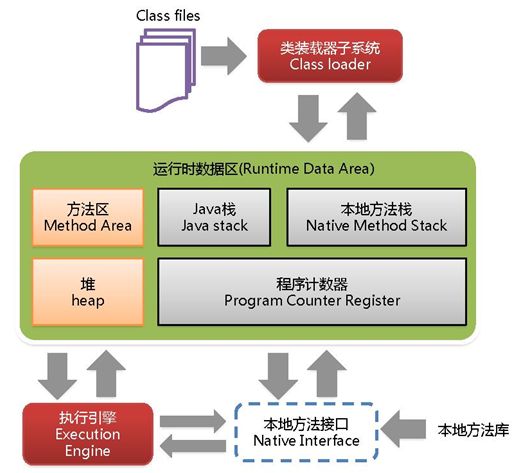
不可重复读（read-committed） 否 是 是

可重复读（repeatable-read） 否 否 是

串行化（serializable） 否 否 否

# 第12章 JVM

## 12.1 JVM内存分哪几个区，每个区的作用是什么?



java虚拟机主要分为以下几个区:

1. **方法区**：
2. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
3. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
4. 该区域是被线程共享的。
5. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。
6. **虚拟机栈**:
7. 虚拟机栈也就是我们平常所称的栈内存,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
8. 虚拟机栈是线程私有的，它的生命周期与线程相同。
9. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
10. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
11. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。
12. **本地方法栈**：  
    本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。
13. **堆**：

java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

1. **程序计数器：**

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

## 12.2 Java类加载过程?

Java类加载需要经历一下几个过程：

1. 加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。
2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
3. 在内存中生成该类的Class对象，作为该类的数据访问入口。
4. 验证

验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

1. 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。
5. 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

1. 解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

1. 初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

## 12.3 java中垃圾收集的方法有哪些?

**1）引用计数法**   应用于：微软的COM/ActionScrip3/Python等

a) 如果对象没有被引用，就会被回收，缺点：需要维护一个引用计算器

**2）复制算法** 年轻代中使用的是Minor GC，这种GC算法采用的是复制算法(Copying)

a) 效率高，缺点：需要内存容量大，比较耗内存

b) 使用在占空间比较小、刷新次数多的新生区

**3）标记清除** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率比较低，会差生碎片。

**4）标记压缩** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率低速度慢，需要移动对象，但不会产生碎片。

**5）标记清除压缩**标记清除-标记压缩的集合，多次GC后才Compact

a) 使用于占空间大刷新次数少的养老区，是3 4的集合体

## 12.4 如何判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:

1. 引用计数法
2. 可达性算法(引用链法)

## 12.5 什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
3. 系统类加载器（system class loader）也叫应用类加载器：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

## 12.6 简述Java内存分配与回收策略以及Minor GC和Major GC（full GC）

**内存分配：**

1. **栈区**：栈分为java虚拟机栈和本地方法栈
2. **堆区**：堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区，主要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1。
3. **方法区**：被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）
4. **程序计数器**：当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。线程私有的。

**回收策略以及Minor GC和Major GC：**

1. 对象优先在堆的Eden区分配。
2. 大对象直接进入老年代。
3. 长期存活的对象将直接进入老年代。

当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor GC通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生GC的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

# 第13章 JUC

**13.1 Synchronized与Lock的区别**

1）Synchronized能实现的功能Lock都可以实现，而且Lock比Synchronized更好用，更灵活。

2）Synchronized可以自动上锁和解锁；Lock需要手动上锁和解锁

**13.2 Runnable和Callable的区别**

1）Runnable接口中的方法没有返回值；Callable接口中的方法有返回值

2）Runnable接口中的方法没有抛出异常；Callable接口中的方法抛出了异常

3）Runnable接口中的落地方法是call方法；Callable接口中的落地方法是run方法

**13.3 什么是分布式锁**

当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。分布式锁可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存，如 Redis，通过set (key,value,nx,px,timeout)方法添加分布式锁。

**13.4 什么是分布式事务**

分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。

# 第14章 面试说明

## 14.1 面试过程最关键的是什么？

1）大大方方的聊，放松

2）体现优势，避免劣势

## 14.2 面试时该怎么说？

1）语言表达清楚

（1）思维逻辑清晰，表达流畅

（2）一二三层次表达

2）所述内容不犯错

（1）不说前东家或者自己的坏话

（2）往自己擅长的方面说

（3）实质，对考官来说，内容听过，就是自我肯定；没听过，那就是个学习的过程。

## 14.3 面试技巧

### 14.3.1 六个常见问题

1）你的优点是什么？

大胆的说出自己各个方面的优势和特长

2）你的缺点是什么？

不要谈自己真实问题；用“缺点”衬托自己的优点

3）你的离职原因是什么？

* + 不说前东家坏话，哪怕被伤过
  + 合情合理合法
  + 不要说超过1个以上的原因

4）您对薪资的期望是多少？

* + 非终面不深谈薪资
  + 只说区间，不说具体数字
  + 底线是不低于当前薪资
  + 非要具体数字，区间取中间值，或者当前薪资的+20%

5）您还有什么想问的问题？

* + 这是体现个人眼界和层次的问题
  + 问题本身不在于面试官想得到什么样的答案，而在于你跟别的应聘者的对比
  + 标准答案：

公司希望我入职后的3-6个月内，给公司解决什么样的问题

公司（或者对这个部门）未来的战略规划是什么样子的？

以你现在对我的了解，您觉得我需要多长时间融入公司？

6）您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前。

### 14.3.2 两个注意事项

1）职业化的语言

2）职业化的形象

### 14.3.3 自我介绍（控制在4分半以内，不超过5分钟）

1）个人基本信息

2）工作履历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因

3）深度沟通（也叫压力面试）

刨根问底下沉式追问（注意是下沉式，而不是发散式的）

基本技巧：往自己熟悉的方向说

# 第15章 LeetCode题目精选

## 15.1 两数之和

问题链接：https://leetcode-cn.com/problems/two-sum/

### 15.1.1 问题描述

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

```

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9

所以返回 [0, 1]

```

### 15.1.2 参考答案

```java

class Solution {

public int[] twoSum(int[] nums, int target) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

int complement = target - nums[i];

if (map.containsKey(complement)) {

return new int[] { map.get(complement), i };

}

map.put(nums[i], i);

}

throw new IllegalArgumentException("No two sum solution");

}

}

```

## 15.2 爬楼梯

问题链接：https://leetcode-cn.com/problems/climbing-stairs/

### 15.2.1 问题描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

```

输入： 2

输出： 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

```

示例 2：

```

输入： 3

输出： 3

解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶

2. 1 阶 + 2 阶

3. 2 阶 + 1 阶

```

### 15.2.2 参考答案

```java

public class Solution {

public int climbStairs(int n) {

if (n == 1) {

return 1;

}

int[] dp = new int[n + 1];

dp[1] = 1;

dp[2] = 2;

for (int i = 3; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

}

```

## 15.3 翻转二叉树

链接：https://leetcode-cn.com/problems/invert-binary-tree/

### 15.3.1 问题描述

翻转一棵二叉树。

示例：

输入：

```

4

/ \

2 7

/ \ / \

1 3 6 9

```

输出：

```

4

/ \

7 2

/ \ / \

9 6 3 1

```

### 15.3.2 参考答案

```java

public TreeNode invertTree(TreeNode root) {

if (root == null) {

return null;

}

TreeNode right = invertTree(root.right);

TreeNode left = invertTree(root.left);

root.left = right;

root.right = left;

return root;

}

```

## 15.4 反转链表

链接：https://leetcode-cn.com/problems/reverse-linked-list/

### 15.4.1 问题描述

反转一个单链表。

示例:

```

输入: 1->2->3->4->5->NULL

输出: 5->4->3->2->1->NULL

```

### 15.4.2 参考答案

```java

public ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextTemp = curr.next;

curr.next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

```

## 15.5 LRU缓存机制

链接：https://leetcode-cn.com/problems/lru-cache/

### 15.5.1 问题描述

运用你所掌握的数据结构，设计和实现一个  LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例:

```

LRUCache cache = new LRUCache( 2 /\* 缓存容量 \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // 返回 1

cache.put(3, 3); // 该操作会使得密钥 2 作废

cache.get(2); // 返回 -1 (未找到)

cache.put(4, 4); // 该操作会使得密钥 1 作废

cache.get(1); // 返回 -1 (未找到)

cache.get(3); // 返回 3

cache.get(4); // 返回 4

```

### 15.5.2 参考答案

```java

class LRUCache extends LinkedHashMap<Integer, Integer>{

private int capacity;

public LRUCache(int capacity) {

super(capacity, 0.75F, true);

this.capacity = capacity;

}

public int get(int key) {

return super.getOrDefault(key, -1);

}

public void put(int key, int value) {

super.put(key, value);

}

@Override

protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {

return size() > capacity;

}

}

/\*\*

\* LRUCache 对象会以如下语句构造和调用:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

```

## 15.6 最长回文子串

链接：https://leetcode-cn.com/problems/longest-palindromic-substring/

### 15.6.1 问题描述

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

```

输入: "babad"

输出: "bab"

注意: "aba" 也是一个有效答案。

```

示例 2：

```

输入: "cbbd"

输出: "bb"

```

### 15.6.2 参考答案

```java

public String longestPalindrome(String s) {

if (s == null || s.length() < 1) return "";

int start = 0, end = 0;

for (int i = 0; i < s.length(); i++) {

int len1 = expandAroundCenter(s, i, i);

int len2 = expandAroundCenter(s, i, i + 1);

int len = Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

int L = left, R = right;

while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {

L--;

R++;

}

return R - L - 1;

}

```

## 15.7 有效的括号

链接：https://leetcode-cn.com/problems/valid-parentheses/

### 15.7.1 问题描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。

2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```

输入: "()"

输出: true

```

示例 2:

```

输入: "()[]{}"

输出: true

```

示例 3:

```

输入: "(]"

输出: false

```

示例 4:

```

输入: "([)]"

输出: false

```

示例 5:

```

输入: "{[]}"

输出: true

```

### 15.7.2 参考答案

```java

class Solution {

// Hash table that takes care of the mappings.

private HashMap<Character, Character> mappings;

// Initialize hash map with mappings. This simply makes the code easier to read.

public Solution() {

this.mappings = new HashMap<Character, Character>();

this.mappings.put(')', '(');

this.mappings.put('}', '{');

this.mappings.put(']', '[');

}

public boolean isValid(String s) {

// Initialize a stack to be used in the algorithm.

Stack<Character> stack = new Stack<Character>();

for (int i = 0; i < s.length(); i++) {

char c = s.charAt(i);

// If the current character is a closing bracket.

if (this.mappings.containsKey(c)) {

// Get the top element of the stack. If the stack is empty, set a dummy value of '#'

char topElement = stack.empty() ? '#' : stack.pop();

// If the mapping for this bracket doesn't match the stack's top element, return false.

if (topElement != this.mappings.get(c)) {

return false;

}

} else {

// If it was an opening bracket, push to the stack.

stack.push(c);

}

}

// If the stack still contains elements, then it is an invalid expression.

return stack.isEmpty();

}

}

```

## 15.8 数组中的第K个最大元素

链接：https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

### 15.8.1 问题描述

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```

输入: [3,2,1,5,6,4] 和 k = 2

输出: 5

```

示例 2:

```

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4

输出: 4

```

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

### 15.8.2 参考答案

```java

import java.util.Random;

class Solution {

int [] nums;

public void swap(int a, int b) {

int tmp = this.nums[a];

this.nums[a] = this.nums[b];

this.nums[b] = tmp;

}

public int partition(int left, int right, int pivot\_index) {

int pivot = this.nums[pivot\_index];

// 1. move pivot to end

swap(pivot\_index, right);

int store\_index = left;

// 2. move all smaller elements to the left

for (int i = left; i <= right; i++) {

if (this.nums[i] < pivot) {

swap(store\_index, i);

store\_index++;

}

}

// 3. move pivot to its final place

swap(store\_index, right);

return store\_index;

}

public int quickselect(int left, int right, int k\_smallest) {

/\*

Returns the k-th smallest element of list within left..right.

\*/

if (left == right) // If the list contains only one element,

return this.nums[left]; // return that element

// select a random pivot\_index

Random random\_num = new Random();

int pivot\_index = left + random\_num.nextInt(right - left);

pivot\_index = partition(left, right, pivot\_index);

// the pivot is on (N - k)th smallest position

if (k\_smallest == pivot\_index)

return this.nums[k\_smallest];

// go left side

else if (k\_smallest < pivot\_index)

return quickselect(left, pivot\_index - 1, k\_smallest);

// go right side

return quickselect(pivot\_index + 1, right, k\_smallest);

}

public int findKthLargest(int[] nums, int k) {

this.nums = nums;

int size = nums.length;

// kth largest is (N - k)th smallest

return quickselect(0, size - 1, size - k);

}

}

```

## 15.9 实现 Trie (前缀树)

### 15.9.1 问题描述

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

示例:

```

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // 返回 true

trie.search("app"); // 返回 false

trie.startsWith("app"); // 返回 true

trie.insert("app");

trie.search("app"); // 返回 true

```

说明:

- 你可以假设所有的输入都是由小写字母 a-z 构成的。

- 保证所有输入均为非空字符串。

### 15.9.2 参考答案

```java

class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

// Inserts a word into the trie.

public void insert(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

char currentChar = word.charAt(i);

if (!node.containsKey(currentChar)) {

node.put(currentChar, new TrieNode());

}

node = node.get(currentChar);

}

node.setEnd();

}

// search a prefix or whole key in trie and

// returns the node where search ends

private TrieNode searchPrefix(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

char curLetter = word.charAt(i);

if (node.containsKey(curLetter)) {

node = node.get(curLetter);

} else {

return null;

}

}

return node;

}

// Returns if the word is in the trie.

public boolean search(String word) {

TrieNode node = searchPrefix(word);

return node != null && node.isEnd();

}

}

```

## 15.10 编辑距离

链接：https://leetcode-cn.com/problems/edit-distance/

### 15.10.1 问题描述

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

1. 插入一个字符

2. 删除一个字符

3. 替换一个字符

示例 1:

```

输入: word1 = "horse", word2 = "ros"

输出: 3

解释:

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

```

示例 2:

```

输入: word1 = "intention", word2 = "execution"

输出: 5

解释:

intention -> inention (删除 't')

inention -> enention (将 'i' 替换为 'e')

enention -> exention (将 'n' 替换为 'x')

exention -> exection (将 'n' 替换为 'c')

exection -> execution (插入 'u')

```

### 15.10.2 参考答案

```java

class Solution {

public int minDistance(String word1, String word2) {

int n = word1.length();

int m = word2.length();

// if one of the strings is empty

if (n \* m == 0)

return n + m;

// array to store the convertion history

int [][] d = new int[n + 1][m + 1];

// init boundaries

for (int i = 0; i < n + 1; i++) {

d[i][0] = i;

}

for (int j = 0; j < m + 1; j++) {

d[0][j] = j;

}

// DP compute

for (int i = 1; i < n + 1; i++) {

for (int j = 1; j < m + 1; j++) {

int left = d[i - 1][j] + 1;

int down = d[i][j - 1] + 1;

int left\_down = d[i - 1][j - 1];

if (word1.charAt(i - 1) != word2.charAt(j - 1))

left\_down += 1;

d[i][j] = Math.min(left, Math.min(down, left\_down));

}

}

return d[n][m];

}

}

```