大数据面试核心 101 问

作者：三石

公众号：　三石大数据

目录

1. HDFS 的架构

HDFS 主要包括三个部分，namenode，datanode 以及 secondary namenode。这里主要讲一下他们的作用：namenode 主要负责存储数据的元数据信息，不存储实际的数据块，而 datanode 就是存储实际的数据块，secondary namenode 主要是定期合并 FsImage 和 edits 文件（这里可以进行扩展，讲一下为什么有他们的存在？首先 namenode 存储的元数据信息是会放在内存中，因为会经常进行读写操作，放在磁盘的话效率就太低了，那么这时候就会有一个问题，如果断电了，元数据信息不就丢失了吗？所以也需要将元数据信息存在磁盘上，因此就有了用来备份元数据信息的 FsImage 文件，那么是不是每次更新元数据信息，都需要操作 FsImage 文件呢？当然不是，这样效率不就又低了吗，所以我们就引入了 edits 文件，用来存储对元数据的所有更新操作，并且是顺序写的方式，效率也不会太低，这样，一旦重启 namenode，那么首先就会进行 FsImage 文件和 edits 文件的合并，形成最新的元数据信息。这里还会有一个问题，但是如果一直向 edits 文件进行写入数据，这个文件就会变得很大，那么重启的时候恢复元数据就会很卡，所以这里就有了 secondary namenode 在 namenode 启动的时候定期来进行 fsimage 和 edits 文件的合并，这样在重启的时候就会很快完成元数据的合并）

2. HDFS 的读写流程

- 写流程：hadoop fs -put a.tat /user/sl/
  - 首先客户端会向 namenode 进行请求，然后 namenode 会检查该文件是否已经存在，如果不存在，就会允许客户端上传文件；
  - 客户端再次向 namenode 请求第一个 block 上传到哪几个 datanode 节点上，假设 namenode 返回了三个 datanode 节点；
  - 那么客户端就会向 datanode1 请求上传数据，然后 datanode1 会继续调用 datanode2，datanode2 会继续调用 datanode3，那么这个通信管道就建立起来了，紧接着 dn3，dn2，dn1 逐级应答客户端；
  - 然后客户端就会向 datanode1 上传第一个 block，以 packet 为单位（默认 64k），datanode1 收到后就会传给 datanode2，dn2 传给 dn3
  - 当第一个 block 传输完成之后，客户端再次请求 namenode 上传第二个 block。【写的时候，是串行的写入 数据块】

- 读流程：hadoop fs -get a.txt /opt/module/hadoop/data/
  - 首先客户端向 namenode 进行请求，然后 namenode 会检查文件是否存在，如果存在，就会返回该文件所在的 datanode 地址，这些返回的 datanode 地址会按照**集群拓扑结构**得出 datanode 与客户端的距离，然后进行排序；然后客户端会选择排序靠前的 datanode 来读取 block，客户端会以 packet 为单位进行接收，先在本地进行缓存，然后写入目标文件中。【读的时候，是并行的读取 数据块】

3. 小文件过多有什么危害，你知道的解决办法有哪些

- 危害：
  - 存储大量的小文件，会占用 namenode 大量的内存来存储元数据信息
  - 在计算的时候，每个小文件需要一个 maptask 进行处理，浪费资源
  - 读取的时候，寻址时间超过读取时间
- 解决方法：
  - 在上传到 hdfs 之前，对小文件进行合并之后再上传
  - 采用 har 归档的方式对小文件进行存储，这样能够将多个小文件打包为一个 har 文件
  - 在计算的时候，采用 combineinputformat 的切片方式，这样就可以将多个小文件放到一个切片中进行计算。
  - 开启 uber 模式，实现 JVM 的重用，也就是说让多个 task 共用一个 jvm，这样就不必为每一个 task 开启一个 jvm

4. Secondary NameNode 了解吗，它的工作机制是怎样的

- Secondary Namenode 主要是用于 edit logs 和 fsimage 的合并，edit logs 记录了对 namenode 元数据的增删改操作，fsimage 记录了最新的元数据检查点，在 namenode 重启的时候，会把 edit logs 和 fsimage 进行合并，形成新的 fsimage 文件
- 工作机制：

- Secondary NameNode 询问 NameNode 是否需要 checkpoint。直接带回 NameNode 是否检查结果
- Secondary NameNode 请求执行 checkpoint
- NameNode 滚动正在写的 edits 日志
- 将滚动前的编辑日志和镜像文件拷贝到 Secondary NameNode
- Secondary NameNode 加载编辑日志和镜像文件到内存，并合并
- 生成新的镜像文件 fsimage.chkpoint
- 拷贝 fsimage.chkpoint 到 NameNode
- NameNode 将 fsimage.chkpoint 重新命名成 fsimage
- 所以如果 NameNode 中的元数据丢失，是可以从 Secondary NameNode 恢复一部分元数据信息的，但不是全部，因为 NameNode 正在写的 edits 日志还没有拷贝到 Secondary NameNode，这部分恢复不了

5. 简述 MapReduce 整个流程

1. map 阶段：首先通过 **InputFormat** 把输入目录下的文件进行逻辑切片，默认大小等于 block 大小，并且每一个切片由一个 maptask 来处理，同时将切片中的数据解析成<key,value>的键值对，k 表示偏移量，v 表示一行内容；紧接着调用 Mapper 类中的 map 方法。将每一行内容进行处理，解析为<k,v>的键值对，在 wordCount 案例中，k 表示单词，v 表示数字 1 ；
2. shuffle 阶段：map 端 shuffle：将 map 后的<k,v>写入环形缓冲区【默认 100m】，一半写元数据信息（key 的起始位置，value 的起始位置，value 的长度，partition 号），一半写<k,v>数据，等到达 80%的时候，就要进行 spill 溢写操作，溢写之前需要对 key 按照**分区进行快速排序**【分区算法默认是 **HashPartitioner**，分区号是根据 key 的 hashcode 对 reduce task 个数取模得到的。这时候有一个优化方法可选，combiner 合并，就是预聚合的操作，将有相同 Key 的 Value 合并起来，减少溢写到磁盘的数据量，只能用来累加、最大值使用，不能在求平均值的时候使用】；然后**溢写**到文件中，并且进行 **merge 归并排序**（多个溢写文件）；reduce 端 shuffle：reduce 会**拉取 copy** 同一分区的各个 maptask 的结果到内存中，如果放不下，就会溢写到磁盘上；然后对内存和磁盘上的数据进行 **merge 归并排序**（这样就可以满足将 key 相同的数据聚在一起）；【Merge 有 3 种形式，分别是内存到内存，内存到磁盘，磁盘到磁盘。默认情况下第一种形式不启用，第二种 Merge 方式一直在运行（spill 阶段）直到结束，然后启用第三种磁盘到磁盘的 Merge 方式生成最终的文件。】

3. reduce 阶段：key 相同的数据会调用一次 reduce 方法，每次调用产生一个键值对，最后将这些键值对写入到 HDFS 文件中。

6. join 原理

1. Reduce 端 join：
   a) map 阶段的主要工作：对来自不同表的数据打标签，然后用连接字段作为 key，其余部分和标签作为 value，最后进行输出
   b) shuffle 阶段：根据 key 的值进行 **hash**，这样就可以将 key 相同的送入一个 reduce 中
   c) reduce 阶段的主要工作：同一个 key 的数据会调用一次 reduce 方法，就是**对来自不同表的数据进行 join**（笛卡尔积）
2. Map 端 join：
   - 原理：将小表复制多份，让每个 map task 内存中存在一份（比如存放到 HashMap 中），然后只扫描大表：对于大表中的每一条记录 key/value，在 HashMap 中查找是否有相同的 key 的记录，如果有，则 join 连接后输出即可。
   - 应用场景：一张表十分小、一张表很大
   - 优点：增加 Map 端业务，减少 Reduce 端数据的压力，尽可能的**减少数据倾斜**。
   - 具体办法：
     - 使用 **DistributedCache 缓存文件**到 Task 运行节点
     - 在 mapper 的 setup 方法中，将文件读取到缓存集合中

7. yarn 的任务提交流程是怎样的

- 首先客户端提交任务到 RM 上，同时客户端会向 RM 申请一个 application，然后 RM 会告诉客户端资源的提交路径（比如 jar 包，配置文件）；然后客户端就会提交任务运行需要的资源到对应路径上，提交完毕后，就会向 RM 申请 Appmaster。RM 会将用户的请求初始化成一个 task，放入调度队列中，接着就会有 NM 领取 task 任务并且创建 container 容器和启动 Appmaster。
- 然后 Appmaster 会向 RM 申请运行 MapTask 的资源，假设有两个切片，RM 就会将运行 maptask 任务分配给两个 nodemanager，这两个 nodemanager 分别

领取任务并创建容器；Appmaster 向这两个 NM 发送程序启动脚本，分别启动 maptask；Appmaster 等待所有 maptask 运行完毕后，再次向 rm 申请容器，运行 reducetask

- 程序运行完毕后，Appmaster 会向 RM 申请注销自己

8. 简述 Hadoop1.0 2.0 3.0 区别

- hadoop1.0 和 hadoop2.0 的区别：
  1. 新增了 YARN 框架，1.0 的时候，MapReduce 既负责资源调度又负责计算，到了 2.0，资源调度就交给了 yarn 框架；
  2. 新增了 HDFS 高可用机制，通过配置 Active 和 Standby 两个 NameNode 实现在集群中对 NameNode 的热备，解决了 1.0 存在的单点故障问题。
- hadoop2.0 和 hadoop3.0 的区别：
  1. hadoop3.0 要求的最低 java 版本为 jdk1.8；
  2. hadoop3.0 的时候支持 hdfs 的纠删码机制，作用就是节省存储空间【普通副本机制假设需要 3 倍存储空间而这种机制只需 1.4 倍即可】；
  3. hadoop3.0 的 MapReduce 进行了优化，性能提高了 30%；
  4. hadoop3.0 支持两个以上的 namenode，也就是可以设置一个 active 和多个 standby

9. 简述什么是 CAP 理论，zookeeper 满足 CAP 的哪两个

- C 表示 Consitency（一致性，也就是从每个节点读取的数据是一样的），A 表示 Avaliablity（可用性，也就是整个系统一直处于可用的状态），P 表示 Partition tolerance（分区容错性，分布式系统在 任何网络分区故障问题的时候，仍然能正常工作），对于一个分布式系统来说的话，最多只能满足其中的两项，并且满足 P 是必须的，所以往往选择就在 CP 或者 AP 中。而 zookeeper 就是满足了一致性和分区容错性。因为 leader 节点挂掉的时候，集群会重新选举出 leader，在这个期间集群是不满足可用性的
- 为什么只能满足其中的两项？
  - 我举个例子，比如说两个机器之间的数据同步出现了问题

- 如果想要保证可用性，不管数据的一致性，继续提供服务就可以了
- 如果想要保证一致性，那么就需要等待一段时间进行数据恢复，在这个期间，集群是不满足可用性的
- 所以一个分布式系统要么满足 AP，要么满足 CP

10. zookeeper 集群的节点数为什么建议奇数台

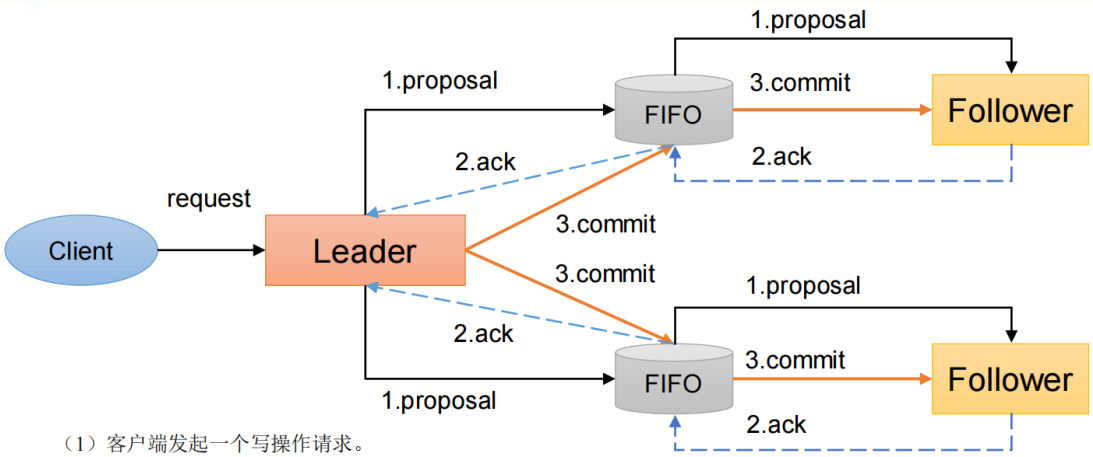
1. 因为 zookeeper 中有一个半数可用机制，就是说，集群中只要有半数以上的机器正常工作，那么整个集群对外就是可用的。比如说如果有 2 个 zookeeper，那么只要 1 个死了 zookeeper 就不能用了，因为 1 没有过半，那么 zookeeper 的死亡容忍度为 0，同理，如果有 3 个 zookeeper，如果死了 1 个，还剩 2 个正常，还是过半的，所以 zookeeper 的死亡容忍度为 1，我之前算过 4 个 5 个 6 个情况下的死亡容忍度，发现了一个规律，2n 和 2n-1 的容忍度是一样的，所以为了节约资源，就选择奇数台
2. 防止因为集群脑裂造成集群用不了。比如有 4 个节点，脑裂为 2 个小集群，都为 2 个节点，这时候，不能满足半数以上的机器正常工作，因此集群就不可用了，那么当有 5 个节点的时候，脑裂为 2 个小集群，分别为 2 和 3，这时候 3 这个小集群仍然可以选举出 leader，因此集群还是可用的

11. Paxos 算法

- 介绍：它是一种基于消息传递且具有高度容错特性的一致性算法，用来解决分布式系统的数据一致性的问题。在一个 paxos 系统中，首先将所有节点划分为 proposer（提议者），acceptor（接收者），learner（学习者）。主要分为三个阶段：
  - prepare 准备阶段：首先 proposer 向多个 acceptor 发出 propose 请求（proposer 生成全局唯一且递增的 proposal id，没有携带提案内容），然后 acceptor 针对收到的请求进行 promise 承诺
  - accept 接收阶段：首先 proposer 收到过半数 acceptor 的 promise 承诺后，向 acceptor 发出 propose 请求，然后 acceptor 针对收到的请求进行 accept 处理

- learn 学习阶段：proposer 将通过的决议发送给所有 learner（服从 proposer）

12. Zab 协议



（1）客户端发起一个写操作请求。

- zab 协议借鉴了 paxos 算法，是专门为 zookeeper 设计的支持崩溃恢复的原子广播协议。
- paxos 算法中采用多个提案者会存在竞争 acceptor 的问题，于是 zab 协议就只采用了一个提案者，而 zookeeper 的一致性就是基于 zab 协议实现的，也就是只有一个 leader 可以发起提案。它包括两种基本的工作模式：**正常和异常**的时候
  - 消息广播
    - 首先客户端向 leader 发送写操作请求，紧接着 leader 将客户端的请求转换为事务 proposal 提案，同时为每个 proposer 分配一个全局的 ID，即 zxid
    - 然后 leader 会为每个 follower 分配一个单独的队列，将需要广播的 proposal 依次放到队列中，并且根据先进先出的策略向 follower 发送提案，当 follower 接收到 proposal 提案之后，会首先写入本地磁盘中，写入成功后向 leader 响应 ack
    - 当 leader 接收到半数以上的服务器的 ack 响应之后，即认为提案发送成功，可以发送 commit 消息，然后 leader 就会向所有 follower 广播 commit 消息，同时自身也会进行事务提交，follower 接收到后，就会进行事务提交（两阶段提交）
  - 崩溃恢复模式：上面介绍的是集群正常的情况下，但是如果 leader 发起事务提案之后就宕机了，此时 follower 还没有收到提案，或者 在 leader

收到半数的 ack 以后，还没来得及发送 commit 消息就宕机了，那么这时候就涉及到 leader 选举和数据恢复两个过程

- 新的 leader 必须满足两个条件：第一 新的 leader 必须是已经提交了 proposal 的 follower 节点，第二 新的 leader 节点含有最大的 zxid。
- 在新的 leader 选举之后，在正式工作开始之前，leader 会确认所有的 proposal 是否已经被集群中过半的服务器提交，同时等到 follower 将所有尚未同步的提案都从 leader 上同步过，并且应用到内存数据中以后，leader 才会把该 follower 加入到真正可用的 follower 列表中。

13. 简述 flume 基础架构

- flume 传输数据依靠的是 Agent 进程，它主要由 3 个组件组成：source，channel，sink
  - source 负责接收数据，可以处理各种类型的日志数据，包括 netcat，exec，spooldir，taildir 等
  - channel 是位于 source 和 sink 之间的缓冲区，包括 memory channel，file channel，kafka channel 等
  - sink 负责将 channel 中的数据写入存储系统或者下一个 Flume 中，包括 HDFS，HBase 等

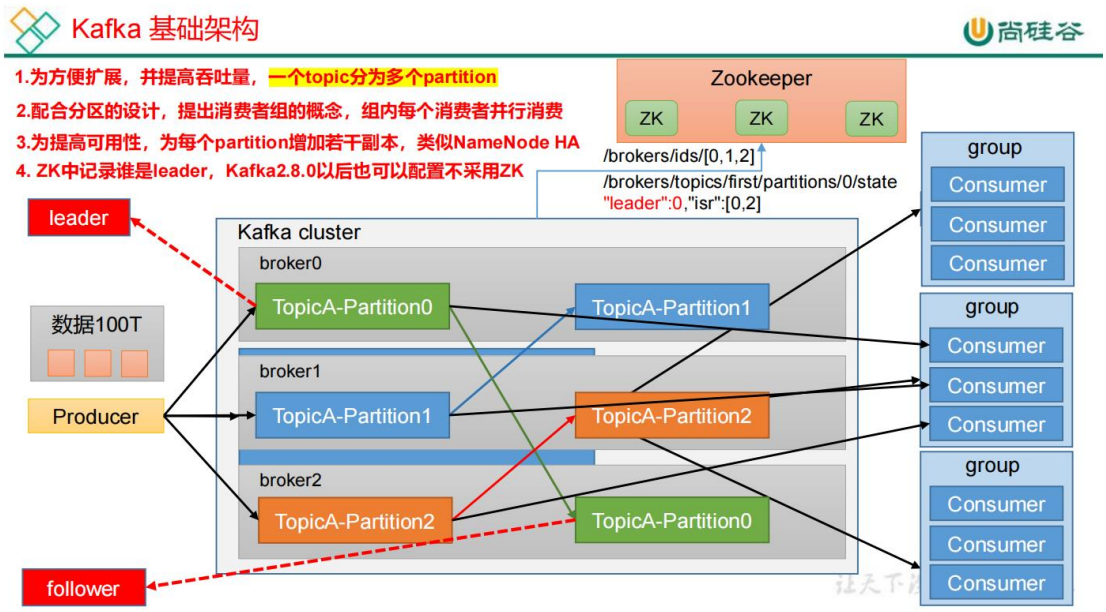
14. 请说一下你提到的几种 source 的不同点

- exec source 能够实时监控，但是不能保证数据不丢失
- spooldir source 能够保证数据不丢失，并且能够实现断点续传，但是不能实时监控
- taildir source 能保证数据不丢失，并且能够实现断点续传，还能够实时监控

15. flume 采集数据会丢失吗

- 如果是 File Channel 不会，因为它将数据存储在文件中，而且传输过程还有事务保证。
- 如果是 Memory Channel 有可能丢，因为它将数据保存在内存中，机器断电重启都会造成数据丢失

16. 简述 kafka 的架构



- 一个 kafka 集群由多个 broker 组成，一个 broker 可以有多个 topic，一个 topic 可以分为多个 partition，每个 partition 可以有若干个副本（一个 leader，若干 follower）

17. 简述 kafka 的分区策略

分区好处：便于合理使用存储资源；提高并行度

- 生产者分区策略
  - 直接指明 partition 的值
  - 没有指明 partition 的值但有 key，那么分区值=key 的 hash 值与 topic 的分区数取余
  - 没有指明 partition 也没有 key，kafka 采用 sticky partition(黏性分区器)，会随机选择一个分区，并尽可能一直使用该分区，待该分区达到了 batchsize 大小或者达到了默认发送时间，kafka 就会再次选择一个分区使用，只要与上一次的分区不同就可以了
  - 自定义分区器：实现 Partitioner 接口，重写 partition 方法
- 消费者分区策略【一个消费者组有多个消费者，一个 topic 有多个分区，所以就会出现到底由哪个 consumer 来消费哪个 partition 的数据】

partition.assignment.strategy

- Range（默认）：【针对一个 topic】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，然后用分区数除以消费者数，得到每个消费者消费几个 partition，然后按分区顺序连续分配若干 partition，除不尽的话 前面几个消费者会多分配一个分区的数据。
  - 问题：如果只是针对 1 个 topic，消费者 0 多消费一个分区影响不大；但是如果有 N 个 topic，那么消费者 0 就会多消费 N 个分区，那么就容易发生数据倾斜
  - 再平衡：挂掉了一个消费者之后，45 秒以内重新发送消息，此时剩余的消费者暂时不能消费到挂掉的消费者应该消费的分区，等到了 45 秒以后，消费者就真正的挂掉了，此时会把它应该消费的分区数都分配给消费者 1 或者消费者 2
- RoundRobin：【针对所有 topic】首先将所有 partition 和 consumer 按照一定顺序排列，然后按照 consumer 依次分配排好序的 partition，若该 consumer 没有订阅即将要 分配的主题，那么直接跳过，继续向下分配
  - 再平衡：也会进行轮询
- sticky：尽量均匀的分配分区给消费者（随机），黏性体现在 在执行新的分配之前，考虑上一次的分配结果，尽量少的变动，这样就可以节省大量的开销
  - 再平衡：均匀分配

18. kafka 是如何保证数据不丢失和数据不重复

- 保证数据不丢失：
  - 生产者端：
    - producer 发送数据到 kafka 的时候，当 kafka 接收到数据之后，需要向 producer 发送 ack 确认收到，如果 producer 接收到 ack，才会进行下一轮的发送，否则重新发送数据
    - 这里面有一个问题：什么时候发送 ack 呢？
      - 于是 kafka 提供了 3 种 ack 应答级别：ack=0，生产者发送过来的数据，不需要等数据落盘就会应答，一般不会使用；ack=1，生

产者发送过来的数据，Leader 收到数据后就会应答，因为这个级别也会丢失数据，所以一般用于传输普通日志；ack=-1（默认级别），生产者发送过来的数据，Leader 和 ISR 队列里面的所有节点收到数据后才会应答，不会丢失数据，一般用于传输和钱相关的数据，那么为什么提出了这个 ISR 呢？因为如果我们等待所有的 follower 都同步完成，才发送 ack，假设有一个 follower 迟迟不能同步，那怎么办呢？难道要一直等吗？因此就出现了 ISR 队列，这里面会存放和 leader 保持同步的 follower 集合，如果长时间（30s）未和 leader 通信或者同步数据，就会被踢出去。

- 消费者端：
  - 消费者消费数据的时候会不断提交 offset，就是消费数据的偏移量，以免挂了，下次可以从上次消费结束的位置继续消费，这个 offset 在 0.9 版本之前，保存在 Zookeeper 中，从 0.9 版本开始，consumer 将 offset 保存在 Kafka 一个内置的 topic 中（__consumer_offsets）。
- broker 端：每个 partition 都会有多个副本
  - 每个 broker 中的 partition 我们一般都会设置有 replication（副本）的个数，生产者写入的时候首先根据分区分配策略（有 partition 按 partition，有 key 按 key，都没有轮询）写入到 leader 中，follower（副本）再跟 leader 同步数据，这样有了备份，也可以保证消息数据的不丢失。

- 保证数据不重复： 如何保证 exactly once 语义？
  - 问题：当我们把 ack 级别设置为-1 之后，假设 leader 收到数据并且同步 ISR 队列之后，在返回 ack 之前 leader 挂掉了，那么 producer 端就会认为数据发送失败，再次重新发送，那么此时集群就会收到重复的数据，这样在生产环境中显然是有问题的
  - 0.11 版本之后，kafka 提出了一个非常重要的特性，幂等性（默认是开启的），也就是说无论 producer 发送多少次重复的数据，kafka 只会持久化一条数据，把这个特性和至少一次语义（ack 级别设置为-1+副本数>=2+ISR 最小副本数>=2）结合在一起，就可以实现精确一次性（既不丢失又不重复）。我大致介绍一下它的底层原理：在 producer 刚启动的时候会分配一个 PID，然后发送到同一个分区的消息都会携带一个 SequenceNum（单调自增的），broker 会对<PID,partition,SeqNum>做缓存，也就是把它当做主键，如果有相同主键的消息提交时，broker 只会持久化一条数据。但是这个机制只能保证单会话的精准一次性，如果想要保证跨会话的精准一次性，那么就需要事务的机制来进行保证

（producer 在使用事务功能之前，必须先自定义一个唯一的事务 id，这样，即使客户端重启，也能继续处理未完成的事务；并且这个事务的信息会持久化到一个特殊的主题当中）

- 如何保证精确一次性的消费？
  - 问题：
    - 重复消费：自动提交 offset；consumer 每 5s 自动提交 offset，如果提交后的 2s，consumer 挂掉了，再次重启 consumer，则从上一次提交 offset 处继续消费，导致重复消费
    - 漏消费：手动提交 offset；消费者消费的数据还在内存中，消费者挂掉了，导致漏消费
  - 解决：手动提交 offset + 采用消费者事务，比如 mysql，也就是说下游的消费者必须支持事务（能够回滚）

19. kafka 中的数据是有序的吗，如何保证有序的呢

- kafka 只能保证 partition 内是有序的，但是 partition 间的有序是没办法保证的
- 解决办法：
  - 设置 topic 有且只有一个 partition
  - 从业务上把需要有序的打到同一个 partition【指定相同的分区号，或者使用相同的 key】
- partition 内有序性的保证：
  - 1.x 版本之前：将 允许最多没有返回 ack 的次数 参数设置为 1
  - 1.x 版本之后：如果开启了幂等性，那么只要设置 这个参数 小于等于 5 就可以了
  - 原因：启用幂等后，kafka 服务端会缓存 producer 发来的最近 5 个 request 的元数据，因此无论如何，都可以保证最近 5 个 request 的数据都是有序的
- kafka 如何实现消息的有序的？

- 生产者：通过分区的 leader 副本负责数据以先进先出的顺序写入，来保证消息顺序性。
- 消费者：同一个分区内的消息只能被一个 group 里的一个消费者消费，保证分区内消费有序。

20. 简述 kafka 消息的存储机制

- kafka 中的消息就是 topic，topic 只是逻辑上的概念，而 partition 才是物理上的概念，每个 partition 会对应一个 log 文件，它存储的就是 producer 生产的数据。生产者生产的数据会不断追加到 log 文件中，如果 log 文件很大了，就会导致定位数据变慢，因此 kafka 会将大的 log 文件分为多个 segment，每个 segment 会对应.log 文件和.index 文件和.timeindex 文件，.log 存储数据，.index 存储偏移量索引信息，.timeindex 存储时间戳索引信息。它的存储结构大概就是这样的【log.segment.bytes = 1g：指 log 日志划分成块的大小】
- 注意：
  - .index 为稀疏索引，大约每往 log 文件写入 4kb 数据，会往 index 文件写入一条索引。
  - log.index.interval.bytes=4kb
  - Index 文件中保存的 offset 为相对 offset，这样能确保 offset 的值所占空间不会过大
  - Kafka 中默认的日志保存时间为 7 天 log.retention.hours

21. kafka 的数据是放在磁盘上还是内存上，为什么速度会快

- 上面说到了放在本地 log 文件中，所以是放在磁盘上
- 速度快有四个原因：（Kafka 每秒可以处理一百万条以上消息，吞吐量达到每秒百万级。那么 Kafka 为什么那么高的吞吐量呢？）
  - kafka 本身是分布式集群，并且采用分区技术，并行度高
  - 读数据采用稀疏索引，可以快速定位要消费的数据
  - 写 log 文件的时候，一直是追加到文件末端，是顺序写的方式，官网中

说了，同样的磁盘，顺序写能达到 600M/s，而随机写只有 100K/s

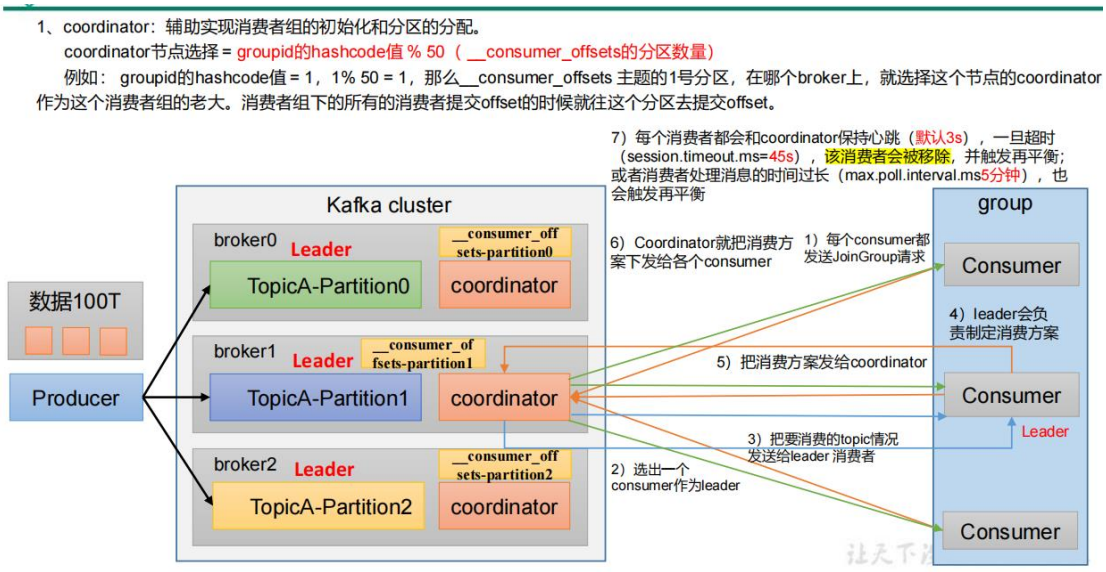
- 实现了零拷贝技术，只用将磁盘文件的数据复制到页面缓冲区一次，然后将数据从页面缓冲区直接发送到网络中，这样就避免了在内核空间和用户空间之间的拷贝

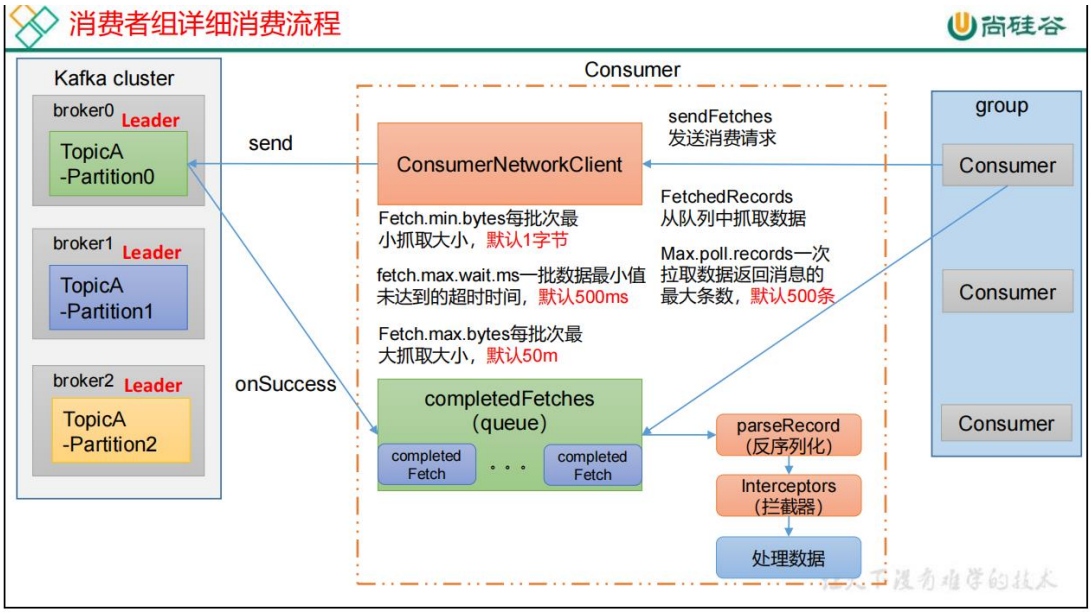
- 补充：传统的读取数据发送到网络中的步骤？
  - 操作系统将数据从磁盘文件读取到内核空间的页面进行缓存
  - 应用程序将数据从内存空间读入用户空间缓冲区【×】
  - 应用程序将读到的数据写回到内核空间并放入 socket 缓冲区【×】
  - 操作系统将数据从 socket 缓冲区复制到网卡接口，此时数据才能通过网络进行发送

22. kafka 消费方式

- 分为两种消费方式
  - pull 模式：主动从 broker 中拉取数据，可以根据消费者的消费能力以适当的速率消费消息
  - push 模式：kafka 没有采用这种方式，因为由 broker 决定消息发送速率，很难适应所有消费者的消费速率

- kafka 采用 pull 模式，但是仍然有一个不足：如果 kafka 中没有数据，消费者可能会陷入循环中，一直返回空数据。（于是又提出了 timeout 机制）





23. HBase 和 hive 的区别

- hbase 是一个数据库，而 hive 一般用于构建数据仓库
- hbase 可以看做是一个存储框架，而 hive 是一款分析框架
- hbase 的查询延迟比较低，常用于在线实时的业务，而 hive 常用于离线的业务

24. 简述 HBase 的读写流程

- 写流程：put 'student', '1001', 'info:sex', 'male'
  - 客户端先访问 zookeeper，获取元数据表 hbase:meta 在哪个 region server 中
  - 访问对应的 region server，获取到元数据表，根据写的请求，确定数据应该写到哪个 region server 的哪个 region 中，然后将 region 信息和元数据表的信息缓存在客户端的 meta cache 中，以便下次访问
  - 与对应的 region server 进行通讯
  - 将数据先写入到 WAL 文件中，然后再写入 memstore 中，数据会在 memstore 中进行排序
  - 写入完成后，region server 会向客户端发送 ack

- 等到达 memstore 的刷写时机（达到一个默认值大小或者达到刷写的时间），将数据刷写到 HFile 中

- 读流程：get 'student', '1001'
  - 客户端先访问 zookeeper，获取元数据表 hbase:meta 在哪个 region server 中
  - 访问对应的 region server，获取到元数据表，根据读的请求，确定数据位于哪个 region server 的哪个 region 中，然后将 region 信息和元数据表的信息缓存在客户端的 meta cache 中，以便下次访问
  - 与对应的 region server 进行通讯
  - 先在 block cache（读缓存）中读，如果没有，然后去 memstore 和 hfile 中读取，将读到的数据返回给客户端并且写入 block cache 中，方便下一次读取

- 扩展：block cache 底层实现
  - https://blog.csdn.net/weixin_40954192/article/details/106963979

25. HBase 在写过程中的 region 的 split 时机

每一个 region 有一个或多个 store 组成，至少是一个 store，一个 store 由一个 memstore 和 0 或多个 StoreFile 组成。

- 默认情况下，每个 table 只有一个 region，随着数据的不断写入，region 会自动进行拆分

- region 切分时机
  - hbase0.94 版本之前，当一个 region 中的某个 store 下的所有 storefile 总大小超过 10G 的时候，就会自动拆分，这个 10G 是默认值，也可以改配置参数
  - hbase0.94 版本之后，当一个 region 中的某个 store 下的所有 storefile 总大小超过 min(表的个数的平方*128M,10G)的时候

26. HBase 中 compact 用途是什么，什么时候触发，分为哪两种，有什么区别

- 用途
  - 合并 HFile 文件，提高读写数据的效率
  - 清除过期和删除的数据
- 触发时间
  - 由于 memstore 每次刷写都会生成一个新的 HFile，当 HFile 的数量达到一定程度后，就需要进行 StoreFile Compaction
- 分类以及区别
  - minor compaction：
    - 会将临近的若干个较小的 HFile 合并成一个较大的 HFile
    - 不会清理过期和删除的数据
  - major compaction：
    - 会将一个 Store 下的所有的 HFile 合并成一个大 HFile
    - 会清理掉过期和删除的数据

27. 热点现象怎么产生的，以及解决方法有哪些

- 热点现象：
  - 某时间段内，对 HBase 的读写请求集中到极少数的 Region 上，导致这些 region 所在的 RegionServer 处理请求量骤增，负载量明显偏大，而其他的 RegionServer 明显空闲
- 原因：
  - hbase 的中的数据是按照字典序排序的，大量连续的 rowkey 集中写在个别的 region，各个 region 之间数据分布不均衡
  - 创建表时没有提前预分区，创建的表默认只有一个 region，大量的数据

写入当前 region

- 创建表已经提前预分区，但是设计的 rowkey 不合理

- 解决办法：
  - 总的来说就是 预分区+rowkey 设计
  - 预分区就是 在创建表的时候，就提前划分出多个 region 而不是默认的一个；rowkey 设计就是 通过设计出合理的 rowkey，让数据均匀的分布到所有的 region 中。

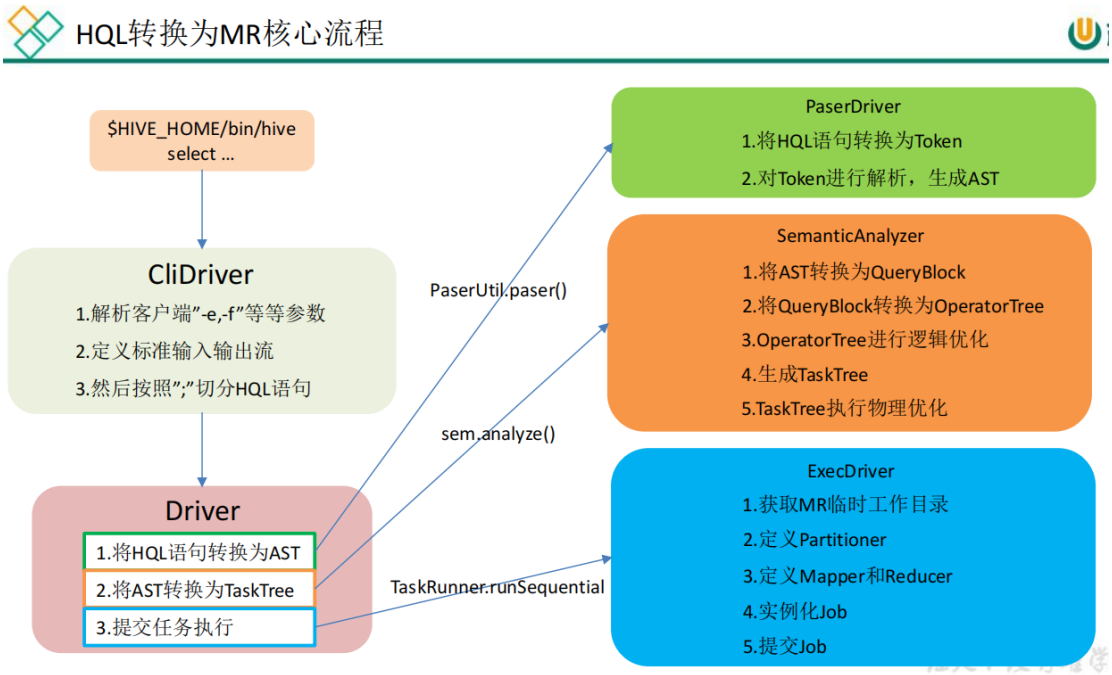
28. 说一下 HBase 的 rowkey 设计原则

- 长度原则：一般是 100 位以内
- 散列原则：rowkey 要具有散列性
  - 计算 hash 值
  - 字符串反转
  - 字符串拼接
- 唯一原则：一个 rowkey 只能出现一次

29. 简述 hive

- 我理解的，hive 就是一款构建数据仓库的工具，它可以将结构化的数据映射为一张表，并且可以通过 SQL 语句进行查询分析。本质上是将 SQL 转换为 MapReduce 或者 spark 来进行计算，数据是存储在 hdfs 上，简单理解来说 hive 就是 MapReduce 的一个客户端工具。
- 补充 1：你可以说一下 HQL 转换为 MR 的任务流程吗？
  - 首先客户端提交 HQL 以后，hive 通过解析器将 SQL 转换成抽象语法树，然后通过编译器生成逻辑执行计划，再通过优化器进行优化，最后通过执行器转换为可以运行的物理计划，比如 MapReduce/spark，然后提交到 yarn 上执行。
  - 详细来说：
    - 首先客户端提交 SQL 以后，Hive 利用 Antlr 框架对 HQL 完成词法语法解析，将 HQL 转换成抽象语法树

- 然后遍历 AST，将其转换成 queryblock 查询块，可以理解为最小的查询执行单元，比如 where
- 然后遍历查询块，将其转换为 操作树，也就是逻辑执行计划
- 然后使用优化器对操作树进行逻辑优化，源码中会遍历所有的优化方式，比如 mapjoin，谓词下推等，来达到减少 MapReduce Job，减少 shuffle 数据量的目的
- 最后通过执行器将逻辑执行计划转换为物理执行计划（MR 到这就结束了）（Tez 和 Spark 还需要 使用物理优化器对任务树进行物理优化），提交到 hadoop 集群运行



- 补充 2：你可以说一下 hive 的元数据保存在哪里吗？
  - 默认是保存在 java 自带的 derby 数据库，但是这有一个缺点：derby 数据库不支持并发，也就是说不能同时两个客户端去操作 derby 数据库，因此通常情况下，都会配置一个 mysql 去存放元数据

30. hive 和传统数据库之间的区别

- 我认为主要有三点的区别：
  - 数据量，hive 支持大规模的数据计算，mysql 支持的小一些
  - 数据更新快不快，hive 官方是不建议对数据进行修改的，因为非常的慢，

这一点我也测试过，而 mysql 经常会进行数据修改，速度也挺快的

- 查询快不快，hive 大多数延迟都比较高的，mysql 会低一些，当然这也与数据规模有关，数据规模很大的时候，hive 不一定比 mysql 慢

- 为什么处理小表延迟比较高：因为 hive 计算是通过 MapReduce，而 MapReduce 是批处理，高延迟的。Hive 的优势在于处理大数据，对于处理小数据是没有优势的

31. hive 的内部表和外部表的区别

从建表语句来看，加上了 **external** 关键字修饰的就是外部表，没加的就是内部表

- 我认为主要有两点的区别：
  - 内部表的数据由 hive 自身管理，外部表的数据由 hdfs 管理
  - 删除内部表的时候，元数据和原始数据都会被删除，而删除外部表的时候仅仅会删除元数据，原始数据不会被删除
- 使用场景：通常都会建外部表，因为一个表通常要多个人使用，以免删除了，还可以找到数据，保证了数据安全

32. hive 的 join 底层实现

- 首先 hive 的 join 分为 common join 和 map join，common join 就是 join 发生在 reduce 端，map join 就是 join 发生在 map 端
- common join：
  - 分为三个阶段：map 阶段，shuffle 阶段，reduce 阶段
    - map 阶段：对来自不同表的数据打标签，然后用连接字段作为 key，其余部分和标签作为 value，最后进行输出
    - shuffle 阶段：根据 key 的值进行 hash，这样就可以将 key 相同的送入一个 reduce 中
    - reduce 阶段：对来自不同表的数据进行 join 操作就可以了
- map join：
  - 首先它是有一个适用前提的，适用于小表和大表的 join 操作
  - 小表多小为小呢？所以就有了一个参数进行配置：

hive.mapjoin.smalltable.filesize=25M

- 它的原理是 将小表复制多份，让每个 map task 内存中存在一份，比如我可以存放到 HashMap 中，然后 join 的时候，扫描大表，对于大表中的每一条记录 key/value，在 HashMap 中查找是否有相同的 key 的记录，如果有，则 join 连接后输出即可，因为这里不涉及 reduce 操作。
- 0.7 版本之后，都会自动转换为 map join，如果之前的版本，我们配置一个参数就可以了：hive.auto.convert.join=true

33. Order By 和 Sort By 的区别

distribute by：将数据根据 by 的字段散列到不同的 reduce 中

cluster by：当 distribute by 和 sort by 字段相同的时候，就等价于 cluster by，但是排序只能是升序

- order by： 全局排序，只有一个 reducer，缺点：当数据规模大的时候，就会需要很长的计算时间
- sort by：分区排序，保证每个 reducer 内有序，一般结合 distribute by 来使用
- 使用场景：在生产环境中，order by 用的比较少，容易导致 OOM；一般使用 distribute by+sort by

34. 行转列和列转行函数

JSON 解析函数：

1. get_json_object：每次只能返回 json 对象中的一列值 `select get_json_object(data,'$.movie') as movie from json;`

2. json_tuple：每次可以返回多列的值 `select b.b_movie, b.b_rate, b.b_timeStamp, b.b_uid from json lateral view json_tuple(json.data,'movie','rate','timeStamp','uid') b as b_movie, b_rate, b_timeStamp, b_uid;`

如果是 json 数组的话，那么就不能直接使用上述的操作，我们可以先使用 regexp_replace 方法进行字符串的替换，将它处理成多个 json，然后再使用上述的方法就可以了

URL 解析函数： HOST QUERY

1. parse_url：一对一

2. parse_url_tuple：一对多

- 常见的行转列包括：一般的聚合函数，比如 max，min，sum；还有汇总函数，比如 collect_list，collect_set
- 常见的列转行就是：explode 函数（json_tuple 函数），只能传入 array 或者 map 的数据，将它拆分成多行，一般会和 lateral view 一起使用
  - SELECT movie, category_name FROM movie_info lateral VIEW explode(split(category,",")) movie_info_tmp AS category_name
- 窗口函数：
  - Rank：
    - rank()：排序相同的时候，排名会重复，总数不变
    - dense_rank()：排序相同的时候，排名会重复，总数减少
    - row_number()：排序相同的时候，排名不会重复，总数不变
  - lag(col,n,default)：返回往上移 n 行的数据，不存在则返回 default
  - lead(col,n,default)：返回往下移 n 行的数据，不存在则返回 default
  - first_value(col)：取分组内排序后，第一个值
  - last_value(col)：取分组内排序后，最后一个值
- over 用法：首先通过 over 来指定窗口的特性，比如可以传入 partition by（分组），order by（排序），rows between .. and .. 指定窗口的范围
  - CURRENT ROW：当前行
  - n PRECEDING/FOLLOWING：往前/后 n 行数据
  - UNBOUNDED PRECEDING/FOLLOWING 表示从前面的起点/到后面的终点
  - 默认是 rows between UNBOUNDED PRECEDING and current row

35. 自定义过 UDF、UDTF 函数吗

1．自定义函数

(1) 自定义 UDF：

① 继承 UDF

② 重写 evaluate 方法

(2) 自定义 UDTF：

① 继承 GenericUDTF

② 重写 3 个方法：initialize，process，close

2．打成 jar 包，上传到服务器中

3．执行命令：add jar "路径" ，目的是将 jar 添加到 hive 中

4．注册临时函数：create temporary function 函数名 as "自定义函数全类名"

36. hive 小文件过多怎么办

- 首先我说一下为什么会产生小文件呢
  - hive 中产生小文件 就是在向表中导入数据的时候，通常来说，我们在生产环境下，一般会使用 insert+select 的方式导入数据，这样会启动 MR 任务，那么 reduce 有多少个就会输出多少个文件，也就是说 insert 每执行一次啊，就至少会生成一个文件，有些场景下，数据同步可能每 10 分钟就会执行一次，这样就会产生大量的小文件。
- 然后我再说一下为什么要解决小文件呢，不解决不行吗？
  - 首先对于 hdfs 来说，不适合存储大量的小文件，文件多了，namenode 需要记录元数据就非常大，就会占用大量的内存，影响 hdfs 性能 存储
  - 对于 hive 来说，每个文件会启动一个 maptask 来处理，这样也会浪费资源 计算
- 最后我说一下怎么解决
  - 使用 hive 自带的 concatenate 命令合并小文件，但是它只支持 recfile 和 orc 存储格式
  - MR 过程中合并小文件
  - map 前
    - 设置 inputformat 为 combinehiveinputformat：在 map 的时候会把多个文件作为一个切片输入
  - map 后，reduce 前
    - map 输出的时候合并小文件 hive.merge.mapfiles
  - reduce 后

- reduce 输出的时候合并小文件 hive.merge.mapredfiles
- 直接设置少一点的 reduce 数量 mapreduce.job.reduces
- 使用 hadoop 的 archive 归档方式

37. Hive 优化

- 建表优化：
  - 分区表：减少全表扫描，通常查询的时候先基于分区过滤，再查询
  - 分桶表：按照 join 字段进行分桶，join 的时候就不会全局 join，而是桶与桶之间进行 join
  - 合适的文件格式：公司中默认采用的是 ORC 的存储格式，这样可以降低存储空间，内部有两个索引（行组索引和布隆过滤器索引）的东西，可以加快查询速度
    - 我知道的 hive 的文件存储格式有 textFile，sequenceFile，ORC，Parquet；其中 textFile 为 hive 的默认存储格式，它和 sequenceFile 一样都是基于行存储的，ORC 和 Parquet 都是基于列存储的。sequenceFile、ORC 和 Parque 文件都是以二进制的方式存储的。
  - 合适的压缩格式：减少了 IO 读写和网络传输的数据量，比如常用的 LZO（可切片）和 snappy
- 语法优化：
  - 单表查询优化：
    - 列裁剪和分区裁剪：如果 select * 或者不指定分区，全列扫描和全表扫描效率都很低（公司规定了必须指定分区，select *没有明确规定）
    - group by 优化：
      - 开启 map 端聚合
      - 开启负载均衡：这样生成的查询计划会有两个 MR Job，一个是局部聚合（加随机数），另外一个是全局聚合（删随机数）
    - SQL 写成多重模式：有多条 SQL 重复扫描一张表，那么我们可以写成 from 表 select... select...
  - 多表查询优化：

- CBO 优化：选择代价最小的执行计划；自动优化 HQL 中多个 Join 的顺序，并选择合适的 Join 算法
  - set hive.cbo.enable = true（默认开启）
- 谓词下推：将 SQL 语句中的 where 谓词逻辑都尽可能提前执行，减少下游处理的数据量。
  - hive.optimize.ppd = true（默认开启）
- MapJoin：将 join 双方比较小的表直接分发到各个 Map 进程的内存中，在 Map 进程中进行 join 操作，这样就不用进行 Reduce，从而提高了速度
  - set hive.auto.convert.join=true（默认开启）
  - set hive.mapjoin.smalltable.filesize=25000000（默认 25M 以下是小表）
- SMB Join：分桶 join，大表转换为很多小表，然后分别进行 join，最后 union 到一起

- job 优化：
  - map 优化
    - 复杂文件增加 map 数
    - 小文件合并
    - map 端聚合
    - 推测执行
  - reduce 优化
    - 合理设置 reduce：
      - 为什么 reduce 的数量不是越多越好？
        - 过多的启动和初始化 reduce 也会消耗时间和资源；
        - 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；
    - 推测执行
  - 任务整体优化：
    - fetch 抓取：Hive 中对某些情况的查询可以不必使用 MapReduce 计

算【全局查找、字段查找、limit 查找】hive.fetch.task.conversion=more

- 小数据集启用本地模式 hive.exec.mode.local.auto=true
- 多个阶段并行执行 set hive.exec.parallel=true
- JVM 重用：针对小文件过多的时候使用

38. 简述 hadoop 和 spark 的不同点（为什么 spark 更快）

shuffle 都是需要落盘的，因为在宽依赖中需要将上一个阶段的所有分区数据都准备好，才能进入下一个阶段，那么如果一直将数据放在内存中，是耗费资源的

- MapReduce 需要将计算的中间结果写入磁盘，然后还要读取磁盘，从而导致了频繁的磁盘 IO；而 Spark 不需要将计算的中间结果写入磁盘，这得益于 Spark 的 RDD 弹性分布式数据集和 DAG 有向无环图，中间结果能够以 RDD 的形式存放在内存中，这样大大减少了磁盘 IO。（假设有多个转换操作，那么 spark 是不需要将第一个 job 的结果写入磁盘，然后再读入磁盘进行第二个 job 的，它是直接将结果缓存在内存中）
- MapReduce 在 shuffle 时需要花费大量时间排序，而 spark 在 shuffle 时如果选择基于 hash 的计算引擎，是不需要排序的，这样就会节省大量时间。
- MapReduce 是多进程模型，每个 task 会运行在一个独立的 JVM 进程中，每次启动都需要重新申请资源，消耗了大量的时间；而 Spark 是多线程模型，每个 executor 会单独运行在一个 JVM 进程中，每个 task 则是运行在 executor 中的一个线程。

39. 简述 spark 的 shuffle 过程

如果问 Spark 与 MapReduce 的 Shuffle 的区别，先说 MapReduce 的 shuffle，再说 spark 的 shuffle，再总结

有 10 个 Map Task，2 个 Reduce Task，2 个 Executer，每个 Executer 有两个 2 Core：

问 Hash Shuffle 产生的文件个数：10 (Map Task) * 2 (Reduce Task)

问优化过的 Hash Shuffle 产生的文件个数： (2 (Executer) * 2 (Core)) * 2 (Reduce Task)

问 SortShuffle 产生的文件个数 ：2 (Executer) * (1 (合并的文件) + 1 (索引))

- spark 的 shuffle 分为两种实现，分别为 HashShuffle（spark1.2 以前）和 SortShuffle（spark1.2 以后）
  - HashShuffle 分为普通机制和合并机制，分为 write 阶段和 read 阶段，write 阶段就是根据 key 进行分区，开始先将数据写入对应的 buffer 中，当 buffer 满了之后就会溢写到磁盘上，这个时候会产生 mapper 的数量*reducer 的数量的小文件，这样就会产生大量的磁盘 IO，read 阶段就是 reduce 去拉取各个 maptask 产生的同一个分区的数据；HashShuffle 的合并机制就是让多个 mapper 共享 buffer，这时候落盘的数量等于 reducer 的数量*core 的个数，从而可以减少落盘的小文件数量，但是当 Reducer 有很多的时候，依然会产生大量的磁盘小文件。
  - SortShuffle 分为普通机制和 bypass 机制
    - 普通机制：map task 计算的结果数据会先写入一个内存数据结构(默认 5M)中，每写一条数据之后，就会判断一下，是否达到了阈值，如果达到阈值的话，会先尝试增加内存到当前内存的 2 倍，如果申请不到才会溢写，溢写的时候先按照 key 进行分区和排序，然后将数据溢写到磁盘，最后会将所有的临时磁盘文件合并为一个大的磁盘文件，同时生成一个索引文件，然后 reduce task 去 map 端拉取数据的时候，首先解析索引文件，根据索引文件再去拉取对应的数据。
    - bypass 机制：将普通机制的排序过程去掉了，它的触发条件是而当 shuffle map task 数量小于 200（配置参数）并且算子不是聚合类的 shuffle 算子(比如 reduceByKey)的时候，该机制不会进行排序，极大的提高了其性能。
- spark shuffle 源码：
  - 比如我们调优的时候说可以增大 shuffle write 的缓存区，减少溢写次数，这是一种错误的说法！！！我看了源码之后才发现这个 5M 的内存结构，是不能手动指定的（internal 修饰），如果资源足够会自动扩容不需要我们进行设置。我们应该调整的其实是 map 溢写时输出流 buffer 的大小，默认是 32k，因为溢写的时候是先往输出流缓冲区写，然后等到 1 万条，溢写到磁盘上。
  - 比如我们 sort merge join 由普通机制变为 bypass 机制，是满足 map task 数量小于 200 并且不是 shuffle 类算子，这其实是错的！！！应该是不使用预聚合才是对的，对应的代码 if(dep.mapSideCombine)

40. spark 的作业运行流程是怎么样的

--num-executors 配置 Executor 的数量

--executor-memory 配置每个 Executor 的内存大小（堆内内存）

--executor-cores 配置每个 Executor 的虚拟 CPU core 数量

- 首先 spark 的客户端将作业提交给 yarn 的 RM，然后 RM 会分配 container，并且选择合适的 NM 启动 ApplicationMaster，然后 AM 启动 Driver，紧接着向 RM 申请资源启动 executor，Executor 进程启动后会向 Driver 反向注册，全部注册完成后 Driver 开始执行 main 函数，当执行到行动算子，触发一个 Job，并根据宽依赖开始划分 stage（阶段的划分），每个 stage 生成对应的 TaskSet（任务的切分），之后将 task 分发到各个 Executor 上执行。
- 扩展 1：spark 怎么提高并行度
  - spark 作业中，各个 stage 的 task 的数量，也就代表了 spark 作业在各个 stage 的并行度
  - 官方推荐，并行度设置为总的 cpu 核心数的 2 到 3 倍
  - 设置参数：spark.defalut.parallelism
- 扩展 2：spark 内存管理
  - spark 分为堆内内存和堆外内存，堆内内存由 JVM 统一管理，而堆外内存直接向操作系统进行内存的申请，不受 JVM 控制
    spark.executor.memory 和 spark.memory.offHeap.size
  - 堆内内存又分为存储内存和执行内存和其他内存和预留内存，存储内存主要存放缓存数据和广播变量，执行内存主要存放 shuffle 过程的数据，其他内存主要存放 rdd 的元数据信息，预留内存和其他内存作用相同。
    - 可用内存：
      - 统一内存（0.6） 存储内存和执行内存的占比是动态变化的
        - 存储内存（0.5）
        - 执行内存（0.5）
      - 其他内存（0.4）
    - 预留内存：300M
  - 堆外内存：

- 优点：减少了垃圾回收的工作，因为垃圾回收会暂停其他的工作；
- 缺点：堆外内存难以控制，如果内存泄漏，那么不容易排查。

41. 你知道 Application、Job、Stage、Task 他们之间的关系吗

- 初始化一个 SparkContext 就会生成一个 Application
- 一个行动算子就会生成一个 Job
- Stage(阶段)等于宽依赖的个数+1
- 一个阶段中，最后一个 RDD 的分区个数就是 Task 的个数
- 总结：Application>Job>Stage>Task

42. Spark 常见的算子介绍一下（10 个以上）

- 算子分为转换算子和行动算子，转换算子主要是将旧的 RDD 包装成新的 RDD，行动算子就是触发任务的调度和作业的执行。
- 转换算子：
  - map：将数据逐条进行转换，可以是数据类型的转换，也可以是值的转换
  - flatMap：先进行 map 操作，再进行扁平化操作
  - filter：根据指定的规则进行筛选
  - coalesce：减少分区的个数，默认不进行 shuffle
  - repartition：可以增加或者减少分区的个数，一定发生 shuffle
  - union：两个 RDD 求并集
  - zip：将两个 RDD 中的元素，以键值对的形式进行合并
  - reduceByKey：按照 key 对 value 进行聚合
  - groupByKey：按照 key 对 value 进行分组
  - cogroup：按照 key 对 value 进行合并
- 行动算子用的比较多的有：

- collect：将数据采集到 Driver 端，形成数组
- take：返回 RDD 的前 n 个元素组成的数组
- foreach：遍历 RDD 中的每一个元素【executor 端】

43. 简述 groupByKey 和 reduceByKey 的区别

- 他们之间主要有两个区别
  - 第一个是，groupByKey 只能分组，不能聚合，而 reduceByKey 包含分组和聚合两个功能
  - 第二个是，reduceByKey 会在 shuffle 前对分区内相同 key 的数据进行预聚合（类似于 MapReduce 的 combiner），减少落盘的数据量，而 groupByKey 只是 shuffle，不会进行预聚合的操作，因此 reduceByKey 的性能会高一些

44. 宽依赖和窄依赖之间的区别

多个连续的 RDD 的依赖关系，称之为血缘关系；每个 RDD 会保存血缘关系

- 宽依赖：父的 RDD 的一个分区的数据会被子 RDD 的多个分区依赖，涉及到 Shuffle
- 窄依赖：父的 RDD 的一个分区的数据只会被子 RDD 的一个分区依赖
- 为什么要涉及宽窄依赖
  - 窄依赖的多个分区可以并行计算；而宽依赖必须等到上一阶段计算完成才能计算下一阶段
- 如何进行 stage 划分：
  - 对于窄依赖，不会进行划分，也就是将这些转换操作尽量放在在同一个 stage 中，可以实现流水线并行计算。 对于宽依赖，由于有 shuffle 的存在，只能在父 RDD 处理完成后，才能开始接下来的计算，也就是说需要要划分 stage（从后往前，遇到宽依赖就切割 stage）

45. spark 为什么需要 RDD 持久化，持久化的方式有哪几种，他们之间的区别是什么

- 因为 RDD 实际上是不存储数据的，那么如果 RDD 要想重用，那么就需要重头开始再执行一遍，所以为了提高 RDD 的重用性，就有了 RDD 持久化
- 分类：缓存和检查点
- 区别：
  - 他们主要有两个区别：
    - 第一个是，缓存有两种方法，一种是 cache，将数据临时存储在内存中（默认就会调用 persist(memory_only)），还有一种是 persist，将数据临时存储在磁盘中，程序结束就会自动删除临时文件；而检查点，就是 checkpoint，将数据长久保存在磁盘中
    - 第二个是，缓存不会切断 RDD 之间的血缘关系，检查点会切断 RDD 之间的血缘关系

46. spark 调优

- 执行计划的过程：
  - 未决断的逻辑计划 （检查 SQL 语法上是否有问题） -- 逻辑计划（检查表名、列名等）-- 优化的逻辑计划 -- 物理计划 -- 选择的物理计划（经过了 CBO 优化）
    - 基于 RBO 的优化：从逻辑计划到优化的逻辑计划
    - 谓词下推：将过滤条件的谓词（where 或者 on）逻辑都尽可能提前执行，减少下游处理的数据量。
      - 下推规则：
        - 内关联：on 和 where，下推结果一致，两表都会下推
        - 左外关联：过滤条件为左表，on 会下推右表，where 会两表都下推；过滤条件为右表，on 会下推右表，where 会两表都下推
      - 列裁剪：扫描数据源的时候，只读取那些与查询相关的字段

- 常量替换：比如过滤条件是 age > 2 + 3，会自动优化为 age > 5
- 基于 CBO 的优化：从物理计划到选择的物理计划（计算所有可能的物理计划的代价，挑选出代价最小的物理执行计划）
- 在进行 CBO 优化之前，应该进行 statistics 收集
  - 表级别的统计信息：整个行数多少和整个表的大小 analyze table 表名 compute statistics
  - 列级别的统计信息：最大值和最小值，有多少空值等 analyze table 表名 compute statistics for columns 列 1,列 2,..
- 使用 CBO 优化：spark.sql.cbo.enabled=true
  - 比如 sort merge join 优化为 broadcast join（大表 join 小表）或者 sort merge bucket join（大表 join 大表）
  - SMB JOIN：首先会排序，然后根据 key 的 hash 值散列到不同的 bucket 中，这样大表就变成了小表。并且相同 key 的数据都在同一个桶之中，再进行 join 操作，那么在联合的时候就会大幅度减少无关项的扫描（条件：两表都是分桶表，个数相等；join 列=排序列=分桶列）

- CPU 的优化：
  - 低效原因：
    - 并行度较低，数据分片较大容易导致 CPU 线程挂起
    - 并行度较高，数据过于分散会让调度开销更多
  - 解决：将并行度设置为并发度（cpu 的总核数）的 2~3 倍
- AQE（自适应查询执行）：spark.sql.adaptive.enabled=true 在运行时，每当 shuffle map 阶段执行完毕，AQE 会结合这个阶段的统计信息，基于既定的规则动态的调整，修改尚未执行的逻辑计划和物理计划，来完成对原始查询语句的运行时优化
  - 动态合并分区：可以在任务开始时设置较多的 shuffle 分区个数，然后在运行时通过查看 shuffle 文件统计信息将相邻的小分区合并成更大的分区
  - 动态切换 join 策略：上述 CBO 优化中提到了
  - 动态优化 join 倾斜：spark.sql.adaptive.skewjoin.enabled=true 开启倾斜 join 检测，如果开启了，那么会将倾斜的分区数据拆分成多个分区
- 数据本地化：

- 进程本地化：task 计算的数据在同一个 executor 中（性能最好）
- 节点本地化：task 计算的数据在同一个 worker 的不同 executor 中

47. sparksql 的三种 join 实现

- 包括 broadcast hash join，shuffle hash join，sort merge join，前两种都是基于 hash join；broadcast 适合一张很小的表和一张大表进行 join，shuffle 适合一张较大的小表和一张大表进行 join，sort 适合两张较大的表进行 join。
- 先说一下 hash join 吧，这个算法主要分为三步，首先确定哪张表是 build table 和哪张表是 probe table，这个是由 spark 决定的，通常情况下，小表会作为 build table，大表会作为 probe table；然后构建 hash table，遍历 build table 中的数据，对于每一条数据，根据 join 的字段进行 hash，存放到 hashtable 中；最后遍历 probe table 中的数据，使用同样的 hash 函数，在 hashtable 中寻找 join 字段相同的数据，如果匹配成功就 join 到一起。这就是 hash join 的过程
- broadcast hash join 分为 broadcast 阶段和 hash join 阶段，broadcast 阶段就是 将小表广播到所有的 executor 上，hash join 阶段就是在每个 executor 上执行 hash join，小表构建为 hash table，大表作为 probe table
- shuffle hash join 分为 shuffle 阶段和 hash join 阶段，shuffle 阶段就是 对两张表分别按照 join 字段进行重分区，让相同 key 的数据进入同一个分区中；hash join 阶段就是 对每个分区中的数据执行 hash join
- sort merge join 分为 shuffle 阶段，sort 阶段和 merge 阶段，shuffle 阶段就是 将两张表按照 join 字段进行重分区，让相同 key 的数据进入同一个分区中；sort 阶段就是 对每个分区内的数据进行排序；merge 阶段就是 对排好序的分区表进行 join，分别遍历两张表，key 相同就 join 输出，如果不同，左边小，就继续遍历左边的表，反之，遍历右边的表

48. 简单介绍下 sparkstreaming

- sparkstreaming 是一种准实时，微批次的数据处理框架
- 和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流 (discretized stream)作为抽象表示，叫做 DStream，代表一个持续不断的数据流。在内部实现上，DStream 是一系列连续的 RDD 来表示。每个 RDD 含有一段时间间隔内的数据

- 然后我再说一下 sparkstreaming 的基本工作原理：首先接受实时输入数据流，然后将数据封装成 batch，比如设置一秒的延迟，那么就会每到一秒就会将这一秒内的数据封装成一个 batch，最后将每个 batch 交给 spark 的计算引擎进行处理，输出一个结果数据流

49. 简述 SparkStreaming 窗口函数的原理

- 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。

50. 简单介绍一下 Flink

- Flink 是一个分布式的计算框架，主要用于对有界和无界数据流进行有状态计算，其中有界数据流就是指离线数据，有明确的开始和结束时间，无界数据流就是指实时数据，源源不断没有界限，有状态计算指的是 在进行当前数据计算的时候，我们可以使用之前数据计算的结果。Flink 还有一个优点就是提供了很多高级的 API，比如 DataSet API、DataStream API、Table API 和 FlinkSQL。Flink 的主要特点大概就是这些！
- 扩展：Flink 的 shuffle 了解吗
  - 其实就是 redistribute，一对多

51. Flink 和 SparkStreaming 区别

- 我觉得他们区别挺大的，其中最大的三点是
  - 第一，计算速度的不同，Flink 是真正的实时计算框架，而 sparkstreaming 是一个准实时微批次的计算框架，也就是说，sparkstreaming 的实时性比起 Flink，差了一大截
  - 第二，架构模型的不同，Spark Streaming 在运行时的主要角色包括：Driver、Executor，而 Flink 在运行时主要包含：Jobmanager、Taskmanager。
  - 第三，时间机制的不用，Spark Streaming 只支持处理时间，而 Flink 支持的时间语义包括处理时间、事件时间、注入时间，并且还提供了 watermark 机制来处理迟到数据。

52. 简述 Flink 运行流程（基于 Yarn）

- 首先 Flink 的客户端将作业提交给 yarn 的 RM，然后 RM 会分配 container，并且选择合适的 NM 启动 ApplicationMaster，然后 AM 启动 jobmanager，向 RM 申请资源启动 taskmanager，然后 jobmanager 就可以分配任务给 taskmanager

53. Connect 算子和 Union 算子的区别

- 他们之间主要有两点区别
  - 第一点，union 算子的两个流类型必须是一样的，而 connect 算子的两个流类型可以不一样，因为 connect 之后，内部其实两个流还是独立，一般还需要使用 comap 来进行转换；
  - 第二点，union 算子可以连接多个流，而 connect 算子只能连接两个流

Flink 的时间语义有哪几种

- event time：表示事件创建的时间，通常由事件中的时间戳描述
- ingestion time：表示数据进入 Flink 的时间
- processing time：表示执行算子的本地系统时间
- 总结一句话：在 Flink 的流式处理中，绝大部分的业务都会使用 eventTime

55. 谈一谈你对 watermark 的理解

只有考虑事件时间语义，才会发生乱序（到达窗口的事件先后顺序和事件时间先后顺序不一致）

- 我先说一下 watermark 是什么，它就是一种特殊的时间戳，作用就是为了让事件时间慢一点，等迟到的数据都到了，才触发窗口计算。我举个例子说一下为什么会出现 watermark？
  - 比如现在开了一个 5 秒的窗口，但是 2 秒的数据在 5 秒数据之后到来，那么 5 秒的数据来了，是否要关闭窗口呢？可想而知，关了的话，2 秒

的数据就丢失了，如果不关的话，我们应该等多久呢？所以需要有一个机制来保证一个特定的时间后，关闭窗口，这个机制就是 watermark

- 什么是 watermark 呢？
  - 我的理解就是，watermark 是一种特殊的时间戳，等于直到当前事件发生的最大事件时间减去设定的延迟时间 assignTimestampsWithWatermarks
  - 它的作用说简单点，就是让事件时间慢一点，等到迟到的数据都到了，才去触发窗口计算
- 什么时候触发窗口计算呢？
  - 当 watermark 等于窗口时间的时候，就会触发计算

56. Flink 对于迟到或者乱序数据是怎么处理的

- watermark 设置延迟时间
- window 的 allowedLateness 方法，可以设置窗口允许处理迟到数据的时间
- window 的 sideOutputLateData 方法，可以将迟到的数据写入侧输出流

57. Flink 中，有哪几种类型的状态，你知道状态后端吗

主要有两种类型的状态，包括 operator state 和 keyed state，operator state 和 key 无关，而 keyed state 和 key 相关。状态后端就是用来保存状态的东西，它主要分为三种，….

- 状态可以理解为一个本地变量
- 有两种类型的状态：
  - operator state【算子状态】：该类型的状态，对于同一任务而言，是共享的
  - keyed state【键控状态】：每一个 key 都会保存一个状态
- 状态后端：

- Flink 的状态在底层是如何保存的呢？因此需要一个东西来进行状态的存储、访问和维护，这个东西就是状态后端
- 分为三种：（持久化存储的三种方式）
  - MemoryStateBackend：内存级的状态后端，会将状态作为内存中的对象进行管理，将它们存储在 TaskManager 的 JVM 堆上。而将 checkpoint 存储在 JobManager 的内存中
  - FsStateBackend：将 checkpoint 存到远程的持久化文件系统（FileSystem）上。而对于本地状态，跟 MemoryStateBackend 一样，也会存在 TaskManager 的 JVM 堆上
  - RocksDBStateBackend：将所有状态序列化后，存入本地的 RocksDB 中存储

58. Flink 是如何保证 Exactly-once 语义的

at-most-once：什么都不干，既不恢复丢失的状态，也不重播丢失的数据。

at-least-once：一些事件可能被处理多次

exactly-once：没有事件丢失，并且对于每个事件，有且仅有处理一次

- 整个端到端的一致性级别取决于所有组件中一致性最弱的组件
- 端到端的一致性包括：
  - 内部保证 —— 依赖 checkpoint
  - source 端 —— 需要外部源可重置偏移量
  - sink 端 —— 需要保证从故障恢复时，数据不会重复写入外部系统
    - 幂等写入：同一份数据无论写入多少次，只保存一份结果
    - 事务性写入：
      - 两种实现方式：WAL 和 2PC
      - WAL（预写日志）：把结果数据先写入 log 文件中，然后在收到 checkpoint 完成的通知时，一次性写入 sink 系统
      - 2PC（两阶段提交）：对于每个 checkpoint，sink 任务会启动一个事务，并将接下来所有接收的数据添加到事务里；然后 将这些数据写入外部 sink 系统，但不提交它们---这时只是预提交；当收到 checkpoint 完成的通知时，它 才正式提交事务，实现结

果的真正写入。

- 如何保证精准一次性呢？
  - 使用 checkpoint 检查点，其实就是 所有任务的状态，在某个时间点的一份快照；这个时间点，应该是所有任务都恰好处理完一个相同 的输入数据的时候。
  - checkpoint 的步骤：
    - Flink 应用在启动的时候，Flink 的 JobManager 创建 CheckpointCoordinator
    - CheckpointCoordinator(检查点协调器) 周期性的向该流应用的所有 source 算子发送 barrier(屏障)。
    - 当某个 source 算子收到一个 barrier 时，便暂停数据处理过程，然后将自己的当前状态制作成快照，并保存到指定的持久化存储（hdfs）中，最后向 CheckpointCoordinator 报告自己快照制作情况，同时向自身所有下游算子广播该 barrier，恢复数据处理
    - 下游算子收到 barrier 之后，会暂停自己的数据处理过程，然后将自身的相关状态制作成快照，并保存到指定的持久化存储中，最后向 CheckpointCoordinator 报告自身快照情况，同时向自身所有下游算子广播该 barrier，恢复数据处理。
    - 每个算子按照 上面这个操作 不断制作快照并向下游广播，直到最后 barrier 传递到 sink 算子，快照制作完成。
    - 当 CheckpointCoordinator 收到所有算子的报告之后，认为该周期的快照制作成功; 否则，如果在规定的时间内没有收到所有算子的报告，则认为本周期快照制作失败。
  - 检查点的保存：
    - 什么时候保存？
      - 在 Flink 中，检查点的保存是周期性触发的，间隔时间可以进行设置
      - 保存的时间点？
        - 当所有任务都恰好处理完一个相同的输入数据的时候，将它们的状态保存下来
  - checkpoint 和 savepoint 的区别
    - 目的：checkpoint 重点是在于自动容错，savepoint 重点在于手动备份、

恢复暂停作业

- 触发者：checkpoint 是 Flink 自动触发，而 savepoint 是用户主动触发
- 状态文件保存：checkpoint 一般都会自动删除；savepoint 一般都会保留下来，除非用户去做相应的删除操作

59. java 的深拷贝和浅拷贝的区别

- 对于基本类型，深拷贝和浅拷贝都是一样的，都是对原始数据的复制，修改原始数据，不会对复制数据产生影响。两者的区别，在于对引用类型的复制
  - 对于浅拷贝，只会复制引用，没有复制指向的对象，所以对原始对象的修改，会对复制对象产生影响；
  - 对于深拷贝，它是复制该引用指向的对象，所以修改原始对象，不会对复制对象产生影响；
- 实现方式：继承 Cloneable 接口，重写 clone 方法，因为 Object 中的 clone 方法就是浅拷贝，所以对于浅拷贝实现就是在 clone 方法中直接调用 super.clone()就可以了；对于深拷贝，一般需要调用强制类型转换操作；

60. java 中==和 equals 的区别

- 对于基本数据类型，==比较的是对应的值，对于引用数据类型，==比较的是地址值
- equals 方法如果未被重写，其作用和==一致，但是通常会重写该方法，比如 String 类型，equals 方法可以用来比较变量值
- 补充题 1：为什么重写 equals 方法要重写 hashcode 方法
  - 因为 hashcode 中有一个规定：如果两个对象相等，那么他们的 hashcode 值一定相等，如果只重写了 equals 方法，那么当两个对象的属性值相等的时候会返回 true，但是显然如果没有重写 hashcode 方法，hashcode 值明显不一样，这样就会和规定产生矛盾。
- 补充题 2：为什么有 equals 方法还需要 hashcode 方法
  - 他们通常是在集合插入元素时配合使用的，在插入对象的时候，先调用该对象的 hashcode 方法，得到哈希码值，如果 table 中不存在该哈希码值，那么直接插入，但是如果 table 中存在该哈希码值，就会继续调用

equals 方法，判断两个对象是否真的相同，相同的就不存，不相同就存进去。

61. String 和 StringBuffer、StringBuilder 的区别

- String 类采用 final 修饰的字符数组来保存字符串，属于不可变类，一旦修改了 String 的值，就会产生新的 String 对象
- StringBuilder 类采用无 final 修饰的字符数组来保存字符串，属于可变类，修改值的时候，直接在原对象上进行操作
- StringBuffer 类和 StringBuilder 类基本一样，唯一的不同就是 StringBuffer 中的方法都是用 synchronized 修饰的，因此是线程安全的
- 应用场景：如果需要经常修改字符串，就使用 StringBuffer 和 StringBuilder，优先选择 StringBuilder，效率较高，但是多线程使用共享变量的时候，优先选择 StringBuffer，保证线程安全

62. 简述面向对象三大特征

- 给出定义：
  - 它和面向过程，是两种不同的处理问题的角度。面向过程更关注事情的每一个步骤和顺序，而面向对象更关注事情有哪些参与者，也就是对象，以及各自需要做什么。
- 举例：
  - 比如对于用洗衣机洗衣服这件事，面向过程会将任务拆解成一系列的步骤，打开洗衣机->放衣服->放洗衣粉->启动洗衣机->清洗->烘干；那么面向对象会拆除人和洗衣机两个对象，人要做的事情：打开洗衣机，放衣服，放洗衣粉，启动洗衣机，而洗衣机需要做的事情：清洗。
- 总结例子：
  - 从这个例子可以看出，面向过程比较直接，而面向对象更具有复用性，扩展性，但是性能开销更大
- 三大特性：

- 封装：把自己的属性和方法让可信的类或对象操作，对不可信的隐藏。比如用 private 修饰成员变量就是一种封装，这样让外面的对象不能直接访问对象的成员变量
  - 作用：隐藏了类的内部实现机制，对外界而言，它的内部细节是隐藏的，暴露给外界的只是它的访问方法。
- 继承：就是从已有的类中派生出新的类，新的类可以使用已有的类的属性和方法，并且还能扩展新的属性和方法
- 多态：就是一个父类能够引用不同的子类
  - 三个条件：继承，方法重写，父类引用指向子类对象
  - 缺点：父类引用无法调用子类特有的功能

63. java 中方法重载和重写的区别

- 定义：
  - 重载：就是让类以统一的方式处理不同类型数据的一种手段，调用方法时通过传递给它们的不同个数和类型的参数来决定具体使用哪个方法，重载是一个类中多态性的一种表现。
  - 重写：当子类需要修改父类的一些方法进行扩展时，这样的操作就称为重写
- 区别：
  - 重载发生在同一个类中，而重写发生在有继承关系的子类中
  - 重载的方法参数列表不同（包括参数顺序、个数、类型），而重写的方法参数列表（包括参数顺序、个数、类型）相同
  - 重载的方法返回值类型和访问修饰符都没有限制，而重写子类方法的返回值类型必须和父类相同，对于访问修饰符，子类的访问范围要大于等于父类的访问范围
- 拓展：
  - 对于重载而言，在方法调用之前，就已经确定所要调用的方法，也叫作静态绑定（早绑定） 编译
  - 对于多态而言，在方法调用时，才会确定所要调用的具体方法，也叫作动态绑定（晚绑定） 运行

64. 集合之间的继承关系

- 集合分为单列集合 Collection 接口和双列集合 Map 接口
  - Collection 接口又包括 List 和 Set 子接口，List 可以存放重复的元素，Set 不能存放重复的元素
    - List 接口有两个实现类，分别是 ArrayList 和 LinkedList，ArrayList 底层是数组，LinkedList 底层是链表
    - Set 接口有两个实现类，分别是 HashSet 和 TreeSet，HashSet 是无序的，TreeSet 是有序的
  - Map 接口有两个实现类，分别是 HashMap 和 TreeMap，HashMap 是无序的，TreeMap 是有序的

65. ArrayList 和 LinkedList 区别

- ArrayList 是基于动态数组（动态数组实际上是新建一个新的符合要求大小的数组）的数据结构，而 LinkedList 是基于链表的数据结构
- ArrayList 存储元素在内存空间是连续的，而 LinkedList 存储元素在内存空间是不连续的
- ArrayList 适合随机访问（下标的方式），不适合插入删除操作，因为这样会移动大量的元素，而 LinkedList 不适合随机访问，适合插入和删除操作

66. ArrayList 扩容过程

- 当我们创建 ArrayList 对象，调用空参构造方法的时候，首先初始化的数组大小为 0，第一次添加元素的时候，会将数组的长度扩充为 10，当添加的元素超过 10 个的时候，首先会扩容到 15，依次类推，每次扩充为原长度的 1.5 倍，最大能扩容的长度为 Integer.MAX_VALUE=2 的 31 次方-1。
- 源码：int newCapacity = oldCapacity - (oldCapacity >> 1);

67. HashMap 底层实现

- 在 jdk1.8 之前，hashmap 由 数组-链表数据结构组成，在 jdk1.8 之后 hashmap 由 数组-链表-红黑树数据结构组成；当我们创建 hashmap 对象的时候，jdk1.8 以前会创建一个长度为 16 的 Entry 数组，jdk1.8 以后就不是初始化对象的时候创建数组了，而是在第一次 put 元素的时候，创建一个长度为 16 的 Node 数组；当我们向对象中插入数据的时候，首先调用 hashcode 方法计算出 key 的 hash 值，然后对数组长度取余（(n-1)&hash(key)等价于 hash 值对数组取余）计算出向 Node 数组中存储数据的索引值；如果计算出的索引位置处没有数据，则直接将数据存储到数组中；如果计算出的索引位置处已经有数据了，此时会比较两个 key 的 hash 值是否相同，如果不相同，那么在此位置划出一个节点来存储该数据（拉链法）；如果相同，此时发生 hash 碰撞，那么底层就会调用 equals 方法比较两个 key 的内容是否相同，如果相同，就将后添加的数据的 value 覆盖之前的 value；如果不相同，就继续向下和其他数据的 key 进行比较，如果都不相等，就划出一个节点存储数据；如果链表长度大于阈值 8（链表长度符合泊松分布，而长度为 8 个命中概率很小），并且数组长度大于 64，则将链表变为红黑树，并且当长度小于等于 6（不选择 7 是防止频繁的发生转换）的时候将红黑树退化为链表。

68. HashMap 扩容过程

- 什么时候需要扩容
  - 当 hashmap 中的元素个数超过 数组长度*加载因子 时，就会进行数组扩容，默认情况下当 hashmap 中的元素个数超过 16*0.75=12 的时候
  - 当 hashmap 中的其中一个链表的对象个数如果超过了 8 个，此时如果数组长度没有达到 64，那么 hashmap 会先扩容解决，如果已经达到了 64，那么这个链表会变为红黑树，节点类型由 Node 变成 TreeNode 类型。
- 原理分析
  - 把原有的数组扩大为原来的两倍，然后重新计算每个元素在新数组的位置，这本来是一个很耗时的过程，但是因为每次扩容都是翻倍，与原来计算(n-1)&hash 的结果相比，只是多了一个 bit 位，所以只需要看该 bit 对应的 hash 值是 1 还是 0 就可以了，是 0 的话，索引不变，如果是 1 的话，就变为 原位置+旧容量

69. ConcurrentHashMap 原理

海康威视研究院一面

- hashmap 线程不安全体现在哪些方面：
  - 数组扩容进行 rehash 的过程中会死循环（链表：头会变成尾）
  - 数据覆盖，比如有多个线程向 hashmap 中插入数据，恰好他们对应的下标相同，这时候就有可能出现数据覆盖的问题
- concurrenthashmap 主要是解决了 hashmap 线程不安全的问题，我主要说一下 concurrenthashmap 是如何保证线程安全的，在 jdk1.7 的时候【锁粒度是 segment】，底层是 segment 数组（本身就是一个 hashmap 对象）-hashentry 数组-reentrantlock 锁，向 concurrenthashmap 添加元素的时候，首先根据 key 计算出 segment 数组对应的下标，然后调用 lock 方法将该位置锁住，然后调用 segment 对象的 put 方法进行插入，最后调用 unlock 方法释放该位置的锁；然后在 jdk1.8 的时候【锁粒度就是每个元素】，底层是数组-链表-红黑树，首先根据 key 计算数组对应的下标，如果下标没有元素，就利用 CAS 来保证线程安全，如果有元素，那么采用的是 synchronize 同步锁的方式来保证线程安全。

70. java 反射机制

- 反射让我们在代码运行的时候，可以知道任意一个类有哪些属性和方法，并且能够调用任意一个类的属性和方法，这种动态获取信息以及动态调用方法或属性的功能就称为反射机制
- 获取 Class 对象有三种方式:【在运行期间，一个类只有一个 Class 对象产生】
  - Object 类中的 getClass 方法
  - 类名.class
  - Class.forName("带包名的类路径")
- 获取类的所有构造方法：Class 对象.getDeclaredConstructors()
- 获取类的所有成员变量：Class 对象.getDeclaredFields()
- 获取类的所有成员方法：Class 对象.getDeclaredMethods()

71. 异常体系

- Java 异常分为 Error 和 Exception 两种，其中 Error 表示程序无法处理的错误，Exception 表示程序本身可以处理的异常。这两个类均继承自 Throwable。Error 常见的有 StackOverFlowError，OutOfMemoryError 等。Exception 可分为运行时异常和非运行时异常。对于运行时异常比如 RuntimeException，可以利用 try catch 的方式进行处理，也可以不处理。对于非运行时异常比如 IOException，必须处理，不处理的话程序无法通过编译。

72. 设计模式

- 设计模式的 7 大原则：单一职责原则；接口隔离原则；依赖倒转（倒置）原则；里氏替换原则；开闭原则；迪米特法则；合成复用原则
- 设计模式分类：
  - 创建型模式
    - 单例模式、抽象工厂模式、原型模式、建造者模式、工厂模式
  - 结构型模式
    - 适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式
  - 行为型模式
    - 模板方法模式、命令模式、访问者模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式、状态模式、策略模式、职责链模式
- 单例模式
  - 定义：采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法（静态方法）
  - 实现方式：
    - 饿汉式（静态常量）：在类中定义一个静态常量，然后在静态方法中返回这个常量对象
    - 饿汉式（静态代码块）：将 new 对象放到了一个静态代码块里面，其他都是一样的

- 懒汉式（线程不安全）：直接在静态方法中进行对象的创建的判断 if(instance==null) new
- 懒汉式（线程安全，同步方法）：直接在静态方法之上加上了 synchronized，来保证线程安全
- 懒汉式（线程安全，同步代码块）：直接在创建对象上加了同步代码块，并不能保证线程安全
- 双重检查：通过两次判断 instance 对象是否为空，并且在第二次判断之前加了同步代码块
- 静态内部类：写一个静态内部类，该类中有一个静态属性用来 new 对象
- 枚举：写一个枚举，instance 作为一个枚举值

- JDK 源码：java.lang.Runtime 就是经典的单例模式（懒汉式）

- 观察者模式
  - 定义：又叫发布-订阅模式，它定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己
    - subject：提供三个接口，可以增加和删除观察者对象以及通知所有的观察者
    - observer：定义一个更新接口
  - 好处：观察者模式设计后，会以集合的方式来管理用户，包括注册，移除和通知
  - JDK 源码：java.util.Observable 就是 被观察者，其中 java.util.Observer 是 观察者

73. JVM 一个类的加载过程

1. 加载：通过一个类的全限定名来获取此类的二进制字节流，然后在内存中生成一个代表这个类的 Class 对象

2. 验证：确保 Class 文件的字节流中包含的信息符合《java 虚拟机规范》的全部约束要求，保证虚拟机的安全

3. 准备：为类变量（即静态变量，被 staic 修饰的变量）赋默认初始值，int 为 0，long 为 0L，boolean 为 false，引用类型为 null；常量（被 staic final 修饰的变量）

赋真实值

4. 解析：把符号引用翻译为直接引用
5. 初始化：执行类构造器<clinit>()方法，真正初始化类变量和其他资源
6. 使用：使用这个类
7. 卸载：一般情况下 JVM 很少会卸载类，如果卸载类需要满足一下三个条件

1) 该类所有的实例都已经被垃圾回收，也就是 JVM 中不存在该类的任何实例
2) 加载该类的类加载器已经被垃圾回收
3) 该类的 Class 对象没有在任何地方被引用

74. JVM 内存结构

线程共享区：堆和元空间

- 程序计数器
  - 存储：字节码行号指示器
  - 作用：记录当前线程执行的字节码指令的地址
  - 特点：
    - 每条线程都有一个独立的程序计数器，是线程私有的内存区域
    - 不存在 OOM，没有 GC
- 虚拟机栈
  - 存储：方法、局部变量、运行数
  - 作用：在执行方法的时候，jvm 会同步创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息，如果方法执行完毕，就会将栈帧从虚拟机栈中出栈。
  - 特点：
    - 每条线程都有一个独立的虚拟机栈，是线程私有的内存区域
    - 当栈深度大于虚拟机所允许的最大深度，就会报 StackOverflowError
    - 当栈需要扩展但是无法申请空间 OOM（比较少见）；Hotspot 虚拟机是没有的
    - 栈所占的空间很小，默认为 1M（栈内存小了，就可以运行更多的线

程）

- 没有 GC（执行的时候进栈，执行完了出栈）
- -Xss 设置栈大小

- 本地方法栈
  - 存储：本地 native 方法
  - 作用：和虚拟机栈基本一样，区别是它是为 native 方法服务的
  - 特点：
    - 每条线程都有一个独立的本地方法栈，是线程私有的内存区域
    - Hotspot 虚拟机将虚拟机栈和本地方法栈合二为一
    - 没有 GC
- 堆
  - 存储：所有创建的对象以及对象变量，数组
  - 作用：用来存放创建的对象
  - 特点：
    - 线程共享的内存区域
    - 虚拟机启动时就会创建
    - 堆内存的分代模型：堆可分为新生代和老年代，默认占比为 1:2，新生代又可以划分为 eden、from survivor、to survivor，默认占比为 8:1:1
    - -Xms 设置初始 Java 堆大小，而-Xmx 设置最大 Java 堆大小
    - 堆无法再扩展时，会报 OOM
    - 在堆为对象分配内存的时候，所有线程共享的堆中可以划分出多个线程私有的分配缓冲区（TLAB），为了解决对象分配时的效率问题（竞争资源、锁...）
- 元空间
  - 存储：虚拟机加载的字节码数据，静态变量，常量，运行时常量池
  - 作用：存储被加载的类信息，常量，静态变量，常量池等数据
  - 特点：
    - jdk1.8 之前叫做方法区/永久代
    - 线程共享的内存区域

- 元空间采用的是本地内存，本地内存有多少剩余空间，它就能扩展到多大空间
- 元空间很少有 GC，因为回收条件比较苛刻（类要等到卸载才可以收集），能回收的信息很少
- 元空间内存不足时，将抛出 OOM

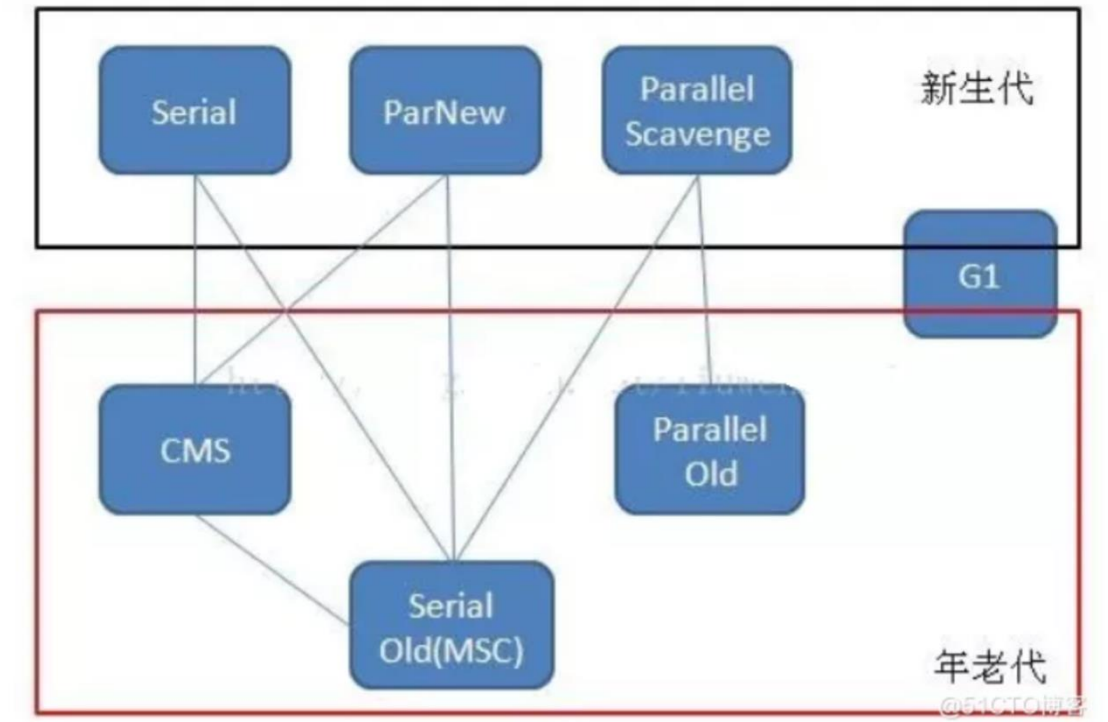
75. JVM 中的垃圾回收算法

- 标记-清除算法
  - 原理：分为标记和清除两个阶段，标记就是标记出所有需要回收的对象，也可以反过来，标记出所有存活的对象，标记的时候是基于可达性分析算法来判断对象是否可以回收；清除，就是标记之后，对所有未被标记的对象进行回收
  - 优点：实现简单
  - 缺点：
    - 执行效率不稳定，如果堆中包含大量对象，并且大部分是需要回收的，这时必须进行大量标记和清除的动作，效率就会降低
    - 内存空间的碎片化问题，标记清除之后会产生大量不连续的内存碎片，碎片太多可能会导致在需要分配较大对象时无法找到足够的连续内存，而不得不触发一次 GC
- 标记-复制算法
  - 原理：将可用内存按容量分为大小相等的两块，每次只使用其中一块，当这一块的内存用完了，就将还存活的对象复制到另外一块内存上，然后再把已使用过的内存空间一次清理掉。
  - 优点：实现简单，效率高，解决了标记清除算法导致的内存碎片问题
  - 缺点：
    - 将可分配内存缩小了一半，空间浪费太多了
    - 当对象存活率比较高时，就要进行较多的复制操作，效率将会降低
- 标记-整理算法
  - 原理：分为标记和整理两个阶段，标记就是基于可达性分析算法对需要回收的对象进行标记；整理就是，根据存活对象进行整理，让存活对象

都向一端移动，然后直接清理掉边界以外的内存。

- STW：在对存活对象进行整理，移动存活对象的时候，必须暂停用户应用程序
- 优点：
  - 提高了空间利用率（和标记复制算法相比）
  - 不会产生不连续的内存碎片
- 缺点：
  - 效率变低了，因为移动存活对象是一个很耗时的操作，并且会停止其他用户程序的运行
  - 分代收集算法
- 现在一般虚拟机的垃圾收集都是采用分代收集算法
  - 根据对象存活周期的不同将内存划分为几块，一般把 java 堆分为新生代和老年代，JVM 根据各个年代的特点采用不同的收集算法
    - 新生代中，每次进行垃圾回收都会发现大量对象死去，只有少量存活，因此采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集
  - 老年代中，因为对象存活率较高，采用标记清理、标记整理算法来进行回收

76. JVM 垃圾收集器



- 新生代收集器：
  - Serial（Client 模式下默认的垃圾回收器 ）
    - 特点：单线程的，收集时需要暂停所有用户线程的工作，所以有卡顿现象，效率不高
    - 优点：简单，不会有线程切换的开销
    - 参数：-XX:-UseSerialGC
  - ParNew
    - 特点：Serial 收集器的多线程版本，大部分基本一样，单 CPU 下，ParNew 还需要切换线程，可能还不如 Serial
    - Serial 或者 ParNew 收集器可以配合 CMS 收集器，前者收集新生代，后者收集老年代
    - 参数：
      - -XX:-UseConcMarkSweepGC 指定使用 CMS 后，会默认使用 ParNew 作为新生代垃圾回收器

- -XX:-UseParNewGC 强制指定使用 ParNew

- Parallel Scavenge（简称 Parallel，JVM 默认的新生代垃圾回收器）
  - 特点：基于复制算法，并行的多线程收集器，侧重于达到一个可控的吞吐量（虚拟机运行 100 分钟，垃圾收集花 1 分钟，则吞吐量为 99%），有时候也叫做吞吐量垃圾回收器或者吞吐量优先的垃圾回收器。
  - 参数：
    - -XX:MaxGCPauseMills：最大停顿时间（不是设置的越小越好，GC 暂停时间越短，那么 GC 的次数就会变得更多）
    - -XX:-UseAdaptiveSizePolicy 自适应新生代大小策略（默认开启），当整个参数开启之后，就不需要人工指定新生代的大小，eden 与 survivor 的比例、晋升老年代对象大小等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间获得最大的吞吐量，这种调节方式称为垃圾收集的自适应的调节策略
    - -XX:-UseParallelGC 指定使用 Parallel Scanvenge 垃圾回收器

- 老年代收集器
  - Serial Old
    - 它是 Serial 收集器的老年代版本，同 Serial 一样，单线程，可在 Client 模式下使用，也可以在 Server 模式下使用，采用标记-整理算法，Serial Old 收集器也可以作为 CMS 收集器发生失败时的后备方案，在并发收集发生 Concurrent Mode Failure 时使用
  - Parallel Old
    - 它是 Parallel 的老年代版本，多线程，标记-整理算法，它是 jdk1.6 才开始提供的，在注重吞吐量和 CPU 资源的情况下，Parallel 新生代 -Parallel Old 老年代是一个很好的搭配（默认）
  - Concurrent Mark Sweep(CMS)
    - 特点：追求最短回收停顿时间；我们知道垃圾回收会带来 STW 的问题，会导致系统卡死时间过长，很多响应无法处理，所以 CMS 收集器采取的是垃圾回收线程和系统工作尽量同时执行的模式来处理的，基于标记-清除算法
    - 参数：-XX:-UseConcMarkSweepGC 指定使用 CMS 垃圾回收器（那

么新生代垃圾收集器就是 ParNew）

- 运作过程（4 个阶段）
  - 初始标记（stw）：标记一下 GC Roots 能直接关联到的对象，那么这些对象也就是需要存活的对象，速度很快
  - 并发标记（不会 stw）：追踪 GC Roots 的整个链路，从 GC Roots 的直接关联对象开始遍历整个对象引用链路，这个过程耗时较长，但是不需要停顿用户线程，可以与垃圾回收收集线程一起并发运行
  - 重新标记（stw）：修正并发标记期间，因用户程序继续运行而导致标记产生变化的那一部分对象的标记记录，这个阶段的停顿时间通常会比初始标记稍微长一些，但也远比并发标记阶段的时间短，它其实就是对在第二阶段中被系统程序运行变动的少数对象进行标记，所以运行速度很快
  - 并发清除（不会 stw）：清理删除掉标记阶段的已经死亡的对象，这个阶段其实是很耗时的，但由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发运行的
- 缺点：
  - 并发阶段，它虽然不会导致用户线程停顿，但会因为占用了处理器的计算能力而导致应用程序变慢，降低总吞吐量
  - 无法处理浮动垃圾（并发清除过程，其他用户线程有可能会产生新的垃圾对象）
  - 清理后会产生内存碎片，将会导致大对象无法分配，往往会出现老年代还有很多剩余空间，但没有足够大的连续空间来分配当前对象，而不得不提前触发一次 Full GC
- 解决：在 full gc 之前，整理内存碎片

- 整堆收集器
  - G1：全称 Garbage First 【jdk9 之后的默认垃圾回收器】
  - G1 可以让我们设置一个预期最大停顿时间，默认值是 200ms，这样就可以尽量把垃圾回收对系统造成的影响控制在你指定的范围内，同时在有限的时间内尽量回收尽可能多的垃圾对象
  - 它把整个堆内存拆分为多个大小相等的 region，让 G1 收集器去追踪各个 region 里面的垃圾回收价值，然后根据用户设置的停顿时间，在后台维

护一个优先级列表，优先回收价值大的那些 region，这也是 Garbage First 名字的由来，这种使用 region 划分堆内存空间，基于回收价值的回收方式，保证了 G1 收集器在有限时间内尽可能收集更多的垃圾

- G1 认为只要大小超过了一个 region 容量一半的对象就可以判定为大对象，放在 H 区中，而对于那些超过了整个 region 容量的超级大对象，将会被存放在 N 个连续的 humongous region 中，G1 通常把 H 区当作老年代来看待
- G1 每个 region=1m~32m，最多有 2048 个 region
- 刚开始新生代对堆内存的占比是 5%，在系统运行中，JVM 会不停的给新生代增加更多的 region，但是最多不能超过 60%，也就是新生代:老年代= 6:4
- 新生代垃圾回收：
  - 也包括 eden 区和 survivor 区，随着不停的在新生代 eden 区对应的 region 中存放对象，JVM 就会不停的给新生代加入更多的 region，直到新生代占据堆大小的最大比例 60%
  - G1 采用复制算法来进行垃圾回收，进入 stw 状态，然后把 eden 对应的 region 中的存货对象复制到 S0 对应的 region 中，接着回收掉 eden 对应的 region 中的垃圾对象。但是它会保证在预期的停顿时间内基于回收价值尽可能多的回收对象
- 老年代垃圾回收：
  - 初始标记（stw），仅仅标记一下 GC Roots 直接能引用的对象，这个过程速度很快，而且是借助 minor gc 的时候同步完成的，所以 G1 在这个阶段并没有额外的停顿
  - 并发标记（不会 stw），这个阶段会从 GC Roots 开始追踪所有的存活对象，这个阶段是很耗时的，但可以和系统程序并发执行，所以对系统程序的影响不大
  - 重新标记（stw），用户程序停止运行，最终标记一下有哪些存活对象，有哪些是垃圾对象
  - 筛选回收（stw），对各个 region 的回收价值和成本进行排序，然后把决定回收的那一部分 region 的存活对象复制到空的 region 中，再清理掉整个旧 region 的全部空间，这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的。
- 混合垃圾收集 mixed gc：

- 它不是一个 old gc，除了回收整个 young region，还会回收一部分的 old region，是回收一部分老年代，而不是全部老年代，可以选择部分 old region 进行收集，从而可以对垃圾回收的耗时时间进行控制。
- 默认当老年代占据了堆内存的 45%时，会尝试触发 mixed gc
- 无论是 年轻代 还是 老年代 都是基于复制算法进行垃圾回收，把各个 region 中存活的对象复制到其他空闲的 region 中。如果万一出现复制时没有空闲 region 可以存活对象了，就会停止系统程序，然后采用单线程进行标记清除和压缩整理，空闲出来一批 region，这个过程很慢。如果内存回收的速度赶不上内存分配的速度，G1 收集器也要被迫暂停用户线程，导致 full gc 而产生长时间 stw

77. java 实现多线程有几种方式

实现接口会更好一些，因为java 不支持多重继承，因此继承了 Thread 类就无法继承其他类，但是可以实现多个接口

1. 继承 Thread 类，只需要创建一个类继承 Thread 类然后重写 run 方法，在 main 方法中调用该类实例对象的 start 方法

2. 实现 Runnable 接口，只需要创建一个类实现 Runnable 接口然后重写 run 方法，在 main 方法中将该类的实例对象传给 Thread 类的构造方法，然后调用 start 方法

3. 实现 Callable 接口，只需要创建一个类实现 Callable 接口然后重写 call 方法（有返回值），在 main 方法中将该类的实例对象传给 Future 接口的实现类 FutureTask 的构造方法，然后再将返回的对象传给 Thread 类的构造方法，最后调用 start 方法

4. 线程池，首先介绍它的好处，然后再说它可以通过 ThreadPoolExecutor 类的构造方法来进行创建。

补充：为什么不能直接调用 run 方法

- 调用 start 方法方可启动线程并使线程进入就绪状态，直接执行 run 方法的话不会以多线程的方式执行

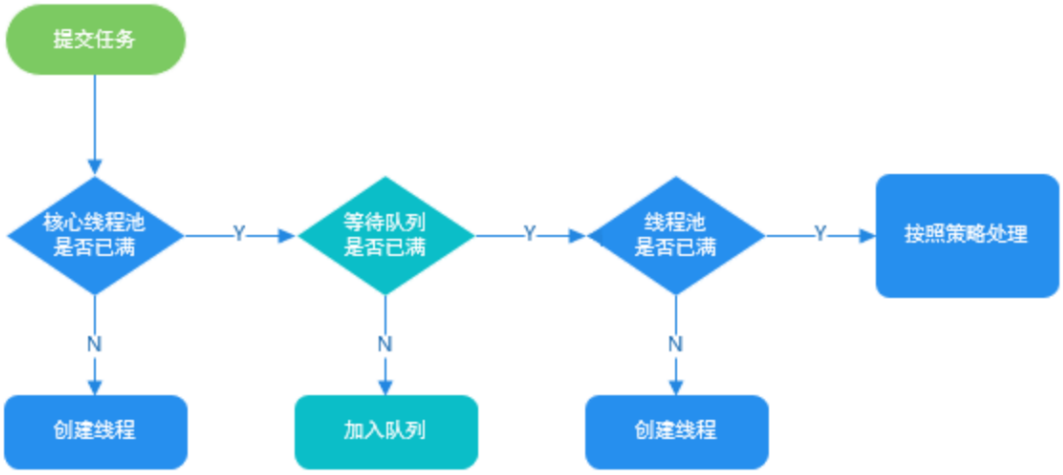
补充：多线程的优缺点？

- 优点：当一个线程进入阻塞或等待状态，cpu 可以先去执行其他线程，提高 cpu 的利用率
- 缺点：
  - 频繁的上下文切换会影响多线程的执行速度

- 容易死锁

78. 线程池相关内容

- 好处：
  - 降低资源消耗，通过重复利用已创建的线程降低线程创建和销毁造成的消耗
  - 提高响应速度，当任务到达时，任务可以不需要等待线程就能立即执行
  - 提高线程的可管理性，线程是稀缺资源，如果无限制的创建，不仅消耗资源而且降低系统的稳定性，使用线程池可以进行线程的统一分配，调优和监控
- Executor 框架的主要成员：ThreadPoolExecutor、ScheduledThreadPoolExecutor、future 接口、Runnable 接口、Callable 接口 和 Executors
- 创建线程池的两种方式：
  a. 通过 ThreadPoolExecutor 构造函数实现（推荐）
  b. 通过 Executor 框架的工具类 Executors 来实现我们可以创建三种类型的 ThreadPoolExecutor：（不推荐，容量为 Integer.MAX_VALUE，所以容易 OOM）
     - CachedThreadPool：返回一个可根据实际情况调整线程数量的线程池，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
     - FixedThreadPool：返回一个固定线程数量的线程池，可控制线程最大并发数，超过的线程会在队列中等待
     - SingleThreadExecutor：返回一个只有一个线程的线程池，它只会用唯一的线程来执行任务，保证所有任务按照执行顺序执行。



- ThreadPoolExecutor 构造方法的 7 个参数：
  - corePoolSize：线程池的核心线程数（最小可以同时运行的线程数）
  - maximumPoolSize：线程池的最大线程数（当任务队列存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数）
  - keepAliveTime：当线程数大于核心线程数时，多余的空闲线程存活的最长时间（如果此时没有新的任务提交，核心线程外的线程不会立即销毁，而是等到这么长时间，会被回收销毁）
  - unit：时间单位
  - workQueue：任务队列，用来存储等待执行任务的队列（当新的任务来的时候，会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，就会被放到队列中）
  - ThreadFactory：线程工厂，用来创建线程
  - RejectedExecutionHandler：拒绝策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务
    - abortPolicy：抛出 RejectedExecutionException 来拒绝新任务的处理（默认）
    - CallerRunsPolicy：用调用者的线程运行任务
    - DiscardPolicy：不处理新任务，直接丢弃掉
    - DiscardOldestPolicy：丢弃最早的未处理的任务请求

- ScheduledThreadPoolExecutor：用来在给定的延迟后运行任务，或者定期执行任务（一般不会使用）

79. synchronized 的原理

- synchronized 的锁对象会关联一个 monitor（监视器，它才是真正的锁对象），这个 monitor 不是我们主动创建的，是 JVM 的线程执行到这个同步代码块，发现锁对象没有 monitor 就会创建 monitor，monitor 内部有两个重要的成员变量 owner：拥有这把锁的线程，recursions：记录线程拥有锁的次数，当一个线程拥有 monitor 后，其他线程只能等待。当执行到 monitorexit 时，recursions 会减 1，当计数器为 0 时，这个线程会释放锁
- 同步方法在反编译之后，会增加 ACC_SYNCHRONIZED 修饰。它会隐式调用 monitorenter 和 monitorexit。在执行同步方法前调用 monitorenter，在执行完同步方法后调用 monitorexit
- monitor 是重量级锁
    - ObjectMonitor 的函数调用中会涉及内核函数，执行同步代码块，没有竞争到锁对象会 park()被挂起，竞争到锁对象的线程会 unpark()唤醒。这个时候就会【存在操作系统用户态和内核态的转换】，这种切换会消耗大量的系统资源，所以说 synchronized 是 java 语言中一个重量级操作

80. OSI 七层模型

*如果问 TCP/IP 模型，这里就不说**表示层和会话层**，数据链路层和物理层一起称为**网络接口层***

- 自上而下分别是应用层，表示层，会话层，传输层，网络层，数据链路层，物理层；
    - 应用层为各种各样的应用提供网络服务，因为网络应用非常的多，所以就要求应用层采用不同的应用协议来解决不同的应用请求，因此应用层是最复杂的一层。典型的协议有 http 80，ftp 21，SMTP 25 等；
    - 表示层主要处理在两个进程之间交换信息的方式；（一个系统的信息另一个系统看得懂）
    - 会话层主要负责两个进程之间的会话管理，包括建立，管理及终止进程间的会

话；

- 传输层主要负责两个进程之间数据的传输服务；
- 网络层主要负责将传输层产生的 TCP 报文段或者 UDP 数据报封装成 IP 数据报；（选择合适的网间路由分发数据）
- 数据链路层主要负责将网络层产生的 IP 数据报封装成帧；
- 物理层主要就是实现比特流的透明传输；

81. TCP 连接管理

- TCP 是面向连接的协议，因此每个 TCP 连接都有三个阶段：连接建立，数据传送和连接释放
  - TCP 连接的建立——三次握手
    - 第一次握手，就是客户端向服务端发送一个连接请求报文，在报文段中将 SYN 置为 1；第二次握手，就是服务端向客户端发送一个确认连接报文，在报文段中将 SYN 和 ACK 都置为 1；第三次握手，就是客户端向服务端发送一个确认的报文，表明自己收到了服务端的确认信息，在报文段中将 ACK 置为 1；这样经过三次握手，TCP 连接就建立了。
  - TCP 连接的释放——四次挥手
    - 第一次挥手，就是客户端向服务端发送一个连接释放请求的 FIN 包；第二次挥手，就是服务端向客户端发送一个确认的 ACK 包，表示我接收到了断开连接的请求，不过服务端可能还有一些数据正在处理；第三次挥手，就是服务端处理完了所有的数据，向客户端发送 FIN 包，表示服务端现在可以断开连接了；第四次挥手，就是客户端向服务端发送一个确认的 ACK 包，此时 TCP 连接还未释放，必须经过计时器设置的时间 2MSL（**MSL：TCP 报文在 Internet 上最长生存时间**）后，客户端才断开连接了；这样经过四次挥手，TCP 连接就断开了。

82. TCP 是如何做到可靠传输的

- TCP 协议主要通过校验和，序号，确认号，超时重传，流量控制等机制来保证数据的可靠传输

  - 校验和 就是说 如果接收方计算出来的校验和与发送方的不一致，那么数据就会被丢弃；（在计算校验和时，需要在 TCP 报文段的前面加上 12 字节的伪首部）

  - 序号 就是本报文段发送的数据的第一个字节的序号，用来保证数据能有序的提交给应用层。

  - 确认号 就是期望收到对方的下一个报文段的数据的第一个字节的序号。

  - 超时重传 就是 TCP 每发送一个报文段，就对这个报文段设置一次计时器，如果设置的重传时间到了但没有收到确认，就要重传这一报文段。

  - 流量控制 就是 基于滑动窗口实现的，主要就是接收方在向发送方发送确认报文的时候，通过设置窗口字段来将接收窗口的大小通知给发送方，这样就可以控制发送方的发送速率，以免发送方发送速度过快，导致数据丢失。

83. TCP 和 UDP 的区别

TCP 准确性相对高，适合文件传输，远程登录

UDP 效率要求相对高，适合 QQ 聊天，在线视频

- 第一，TCP 面向连接，而 UDP 面向无连接；
- 第二，TCP 可靠，而 UDP 不可靠；
- 第三，TCP 面向字节流，而 UDP 面向报文；
- 第四，TCP 首部是 20B，而 UDP 首部是 8B；
- 第五，TCP 只支持一对一通信，而 UDP 支持广播通信；
- 第六，UDP 传输比 TCP 更快，所以更适合即时通信

84. 浏览器输入 URL 到显示页面的过程

- 首先浏览器会向 DNS 请求解析域名的 ip 地址，然后和该服务器建立 TCP 连接，浏览器会向服务器发出 HTTP 请求，服务器处理完请求后就会将 HTTP 响应结果发送给浏览器，然后关闭 TCP 连接，浏览器对响应结果进行解析并且渲染页面

85. 进程和线程的区别

- 他们之间有 3 点区别：第一点，进程是资源分配的最小单位，而线程是程序执行的最小单位，一个线程只能属于一个进程，而一个进程可以多个线程，第二点，进程有自己独立的地址空间，而线程共享进程的地址空间，第三点，进程间不会相互影响，而一个进程内某个线程挂掉，将会导致整个多线程程序挂掉
- 扩展：协程是一种用户态的轻量级线程、协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切换回来的时候，恢复先前保存的寄存器上下文和栈，没有内核切换开销所以非常快
- 补充：协程和线程之间的区别？
    - 一个线程可以有多个协程，一个进程也可以单独拥有多个协程
    - 线程进程都是同步机制，而协程则是异步
    - 协程能保留上一次调用时的状态，每次过程重入时，就相当于进入上一次调用的状态·

86. 什么是死锁以及死锁的四个条件

- 定义：多个进程在执行的过程中，因为争夺资源造成了一种相互等待的状态，就叫死锁
- 四个必要条件：互斥，占有并等待，非抢占和循环等待；互斥，就是一个资源每次只能被一个进程使用；占有并等待，就是一个进程在等待其他资源的时候，对已经获得的资源不会释放；非抢占，就是对于其他进程已经获得的资源，不能强行占有他们的资源；循环等待，就是多个进程之间形成了一种首尾相连的等待资源关系；如果系统中以上四个条件同时成立，那么就能引起死锁

- 解决死锁主要有四种方法：预防死锁，避免死锁，死锁检测与恢复，鸵鸟策略
- **破坏死锁条件**（预防死锁）：
  - 破坏占有并等待条件：所有的进程在开始运行之前，必须一次性地申请其在整个运行过程中所需要的全部资源。
  - 破坏非抢占条件：占用部分资源的线程申请其他资源时，如果申请不到，可以主动释放自己的资源。
  - 破坏循环等待条件：申请资源的时候按顺序申请，我们可以将每个资源编号，当一个进程占有编号为 i 的资源时，那么它下一次申请资源只 能申请编号大于 i 的资源。
- **避免死锁**：银行家算法：预先给进程分配资源，然后判断当前是否处于安全状态（查看资源池所剩资源是否能满足所有进程中所需资源最小的进程的需求），如果是，那么对这个进程的资源分配有效，否则无效
- **死锁检测与恢复**：不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复

87. 页面置换算法

- 缺页中断：在作业运行的过程中，如果发现要访问的页面不在内存中，就发生了缺页中断，此时操作系统就会将对应的页面调入到内存中。
- 页面置换算法就是 当发生缺页中断的时候，如果当前内存中没有空闲的空间，那么操作系统就必须在内存中选择一个页面将其移出内存，用来选择淘汰哪一个页面的规则就叫做页面置换算法
- 主要有四种算法：先进先出，最少使用，最近最少使用，最佳页面置换算法；
  - 先进先出就是 淘汰最先进入内存的页面；最少使用就是 淘汰最少使用过的页面；最近最少使用就是 淘汰最近的一段时间内未被使用的页面；最佳页面置换就是 淘汰以后永不使用或者很长时间内不会被访问的页面，这怎么可以知道撒，所以是一种无法实现的算法；

88. mysql 的索引结构

*在数据库中，B+树的高度一般为 2~4 层，同时 Innodb 存储引擎在设计的时候会将**根节***

点常驻在内存中，也就是说在查找的时候最多只需要 1~3 次 I/O 操作

- mysql 常见的索引存储结构有二叉树、红黑树、哈希表、B 树和 B+树
- mysql 的默认存储引擎 Innodb 就是用 **B+树**实现索引结构的
- B+树就是在 B 树基础上的一种优化。B 树中每个节点不仅包含数据的 key 值，还有 data 值，而每一页的存储空间是有限的（16KB），如果 data 数据较大时，将会导致一个页能存储的节点较少，这样会导致 B 树的深度较大，因此会增大查询的磁盘 I/O 次数，影响查询效率。在 B+树中，所有数据记录都是按照键值大小**顺序存放**在叶子节点上，而非叶子节点只存储键值信息，这样可以加大每页存储的节点数量，降低 B+树的高度，减少磁盘 IO。

89. 简述事务

- 我从两个方面来介绍一下事务
  - 第一，事务是什么？
    - 一个事务是由一条或者多条 sql 语句组成的不可分割的单元，要么全部执行成功，要么全部执行失败。
  - 第二，事务的基本特性？
    - 事务有**四个基本特性**，分别是原子性，一致性，隔离性，持久性 ACID
      - 原子性是说一个事务中的所有操作要么全部完成，要么全部不完成。
      - 一致性是说一个事务执行之前和执行之后都必须处于一致性状态。
      - 隔离性：同一时间，只允许一个事务请求同一数据，事务间互不干扰。
      - 持久性：一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的。

90. 数据库事务并发会引发哪些问题

- 主要会有四个问题，分别是脏读、不可重复读、幻读、丢失更改
  - 脏读就是一个事务读取到了另外一个事务未提交的数据（A 事务读取 B 事务尚未提交的数据并在此基础上操作，而 B 事务执行回滚，那么 A 读取到的数据就是

脏数据）

- 不可重复读就是一个事务两次执行同一条查询语句，读取到的结果不一致，可能因为另外一个事务更新了数据

- 幻读就是一个事务两次执行同一条查询语句，莫名多出了一些之前不存在的数据，或莫名少了一些原先存在的数据，可能因为另外一个事务新增或者删除了数据

- 丢失修改就是两个事务都对同一记录进行修改操作，结果后修改的记录将会覆盖前面修改的记录，因此前面的修改就丢失掉了

91. 事务的四个隔离级别有哪些

InnoDB 的默认事务隔离级别是可重复读

- read uncommitted（读未提交）：一个事务可以读取另外一个事务未提交的数据，隔离级别最低。
- read committed（读提交）：一个事务只能读取另外一个事务提交的数据。
- repeatable read（可重复读）：一个事务多次读取同一条记录，返回的结果是一致的（即使有其他事务对该条记录进行了修改）。
- serializable（序列化）：要求事务序列化执行，也就是只能一个接着一个地执行，隔离级别最高。

92. MVCC 讲一下（怎么实现）

- MVCC 多版本并发控制，就是同一条记录在系统中存在多个版本。其存在目的是在保证数据一致性的前提下提供一种高并发的访问性能。对数据读写在不加读写锁的情况下实现互不干扰，从而实现数据库的隔离性，在事务隔离级别为读提交和可重复读中使用到
- MVCC 通过维持一个数据的多个版本，在不加锁的情况下，使读写操作没有冲突。主要由三个隐式字段（事务 ID，回滚指针，聚集索引 ID）、undo log（存放了数据修改之前的快照）和 read view（存放开始了还未提交的事务）去实现。在 InnoDB 中，事务在开始前会向事务系统申请一个事务 ID，同时每行数据具有多个版本，我们每次更新数据都会生成新的数据版本，而不会直接覆盖旧的数据版本。每行数据中包含多个隐式字段，其中实现 MVCC 的主要涉及最近更改该行数据的事务 ID 和可以找到历史数据版

本的指针。InnoDB 在每个事务开启瞬间会为其构造一个记录当前已经开启但未提交的事务 ID 的 readview 视图数组。通过比较链表中的事务 ID 与该行数据的值对应的 DB_TRX_ID，并通过回滚指针找到历史数据的值以及对应的 DB_TRX_ID 来决定当前版本的数据是否应该被当前事务所见。最终实现在不加锁的情况下保证数据的一致性。

93. 为什么要对数据仓库分层

- 第一个是**将复杂的需求简单化**：我们通过将复杂的问题分解为多个步骤来完成，每一层只处理单一的步骤，比较容易和理解
- 第二个是**提高数据的复用性**：比如在已经得到最终结果之后，又需要中间层的一些数据，我可以直接查询中间层的数据，不必重新进行计算
- 补充说一下：我觉得数据仓库就是一种**以空间换取时间**的架构！

94. 数据仓库建模的方法有哪些

这里如果面试的是阿里，可以说大数据之路这本书中一共介绍了四种数仓建模的方法，ER 模型，维度模型，Data Vault 模型和 Anchor 模型，后面两种模型都是基于 ER 模型的一种优化，我主要介绍一下 ER 模型和维度模型

- ER 模型是 Inmon 提出的，这个模型是符合 3 范式的，他的出发点就是整合数据，将各个系统中的数据以整个企业角度按主题进行分类，但是不能直接用于分析决策
- 维度模型是 Kimball 提出的，这个人和 Inmon 算是数仓的两个流派，他的出发点就是分析决策，为分析需求服务，而现在多数的数仓的搭建都是基于维度模型进行搭建的。
- 区别：ER 模型冗余更少，但是在大规模数据跨表分析中，会造成多表关联，这会大大降低执行效率

95. 维度建模有哪几种模型

- **星型模型**：这是最常用的维度建模的方式，核心就是 以事实表为中心，所有的维度表直接连接在事实表上，就和星星一样，所以叫做星型模型
- **雪花模型**：这是星型模型衍生而来的，相对于它不同的是 维度表可以再连接其他维度表，有点类似于 3NF 模型
- 星座模型：这也是星型模型的衍生而来的，相对于它不同的是 基于多张事实表，

并且共享维度信息，也就是维度表之间可能会连在一起。一般来说，企业中不会只有一张事实表，所以大多数情况下都是星座模型进行维度建模的。

96. 维度建模中表的类型

- 维度表：一张维度表就表示对一个对象的一些描述信息。每个维度表都包含单一的主键列，和一些对该主键的描述信息，通常维度表会很宽。比如 乘客信息表，司机信息表，城市首都表
- 事实表：一张事实表就表示对业务过程的描述，比如播单，下单，支付。每个事实表都包含若干维度外键，若干退化维度（维度属性存储到事实表中，减少关联），和数值型的度量值，通常事实表会比较大。

97. 事实表的设计过程

- 一共有五步，分别是选择业务过程，声明粒度，确定维度，确定事实，冗余维度
- 选择业务过程 就是对业务的整个生命周期进行分析，然后选择与需求有关的业务过程，比如滴滴呼单的整个过程，乘客呼单，平台播单，司机抢单，司机接驾，完成订单，（买家下单，买家付款，卖家发货，买家确认收货）然后就是根据我们的需求去选择对应的过程
- 声明粒度 ，粒度就是用于确定事实表中一行所表示的业务的细节层次，通常在设计事实表的时候，粒度定义的越细越好，比如订单明细表的粒度就是 订单级别
- 确定维度，选择描述清楚业务过程所处环境的维度信息中，比如订单明细表中 出发城市，到达城市，产品线，司机，订单状态等（支付事实表，买家，买家，商品，收货人信息，业务类型，订单时间）
- 确定事实，事实就是分析业务过程中的度量值，比如订单金额，订单次数等
- 冗余维度，在事实表中冗余一些下游用户需要使用的常用维度，减少多表之间的关联。

真实案例：交易域构建维度模型（事实表）

- 主要选择了和订单有关的业务过程：冒泡 发单 播单 完单 支付；其中发单和完单和支付放在一张事实表中，构成多事务事实表 （流量域：冒泡，打开，登陆，浏览，点击，退出）
- 确认每张事实表的粒度，我们的订单相关的表的粒度都是采用最小的粒度，冒泡表

采用 乘客粒度，订单表采用 订单粒度，播单表采用 订单粒度

- 确认分析的维度，也就是看待事实的角度，主要包括日期 国家 城市 产品线 乘客 司机 订单状态 播单类型 等
- 确认事实，也就是度量值，主要包括 订单量，订单金额，接驾时长，接驾距离
- 经过上述四个步骤，交易相关的 dwd 的雏形就建立起来了，但是为了后续分析的方便，我们还进行了退化维度，就是把后续分析可能用到的维度退化到事实表中，比如城市的经纬度信息，后续可能分析某一位置的订单量更多，就需要这一维度

98. 同时在线问题

问题：如下为某直播平台主播开播及关播时间，根据该数据计算出平台最高峰同时在线的主播人数。

id stt edt

1001 2021-06-14 12:12:12 2021-06-14 18:12:12

1003 2021-06-14 13:12:12 2021-06-14 16:12:12

1004 2021-06-14 13:15:12 2021-06-14 20:12:12

1002 2021-06-14 15:12:12 2021-06-14 16:12:12

1005 2021-06-14 15:18:12 2021-06-14 20:12:12

1001 2021-06-14 20:12:12 2021-06-14 23:12:12

1006 2021-06-14 21:12:12 2021-06-14 23:15:12

1007 2021-06-14 22:12:12 2021-06-14 23:10:12

…

```
select max(cnt)
from (
    select
        id, dt, sum(flag) over(order by dt) cnt
    from (
        select id, stt dt, 1 as flag
        from test5
        union
        select id, ett dt, -1 as flag
        from test5
```

```
    ) t
) t
```

99. 最大连续登陆的最大天数问题

```
SELECT
B.uid uid, max(B.num) cnt_days
FROM
(
    SELECT
        A.uid, A.dt1, count(A.dt1) num
    FROM
    (
        -- 连续登陆的话  dt1 都会是相同的
        SELECT
            uid,
            date_sub( dt, row_number() over ( PARTITION BY uid ORDER
BY dt )) AS dt1
        FROM user_login
    ) A
    group by A.uid, A.dt1
) B
group by B.uid;
```

100. 留存问题

```
-- 从前往后算
select
    t1.dt,
    count(t1.uid) as active_users,   -- 当天活跃用户数
    count(case when datediff(t2.dt,t1.dt)=1 then t2.uid end) as day2_active_users,
    count(case when datediff(t2.dt,t1.dt)=6 then t2.uid end) as day7_active_users
from
```

```
(
    select
        uid, dt
    from ods_app_open
    where dt='20211118'
    group by uid, dt
) t1
left join
(
    select
    uid, dt
    from ods_app_open
    where dt>'20211118' and dt<='2021124'
    group by uid, dt
) t2
on t1.uid=t2.uid
group by t1
```

101. 数据倾斜

- 定义：绝大部分任务都很快完成，只有一个或者少数几个任务执行的很慢甚至最终执行失败
  - 为什么：
    - **map task 数据倾斜：**
      - 输入数据文件导致的数据倾斜
      - 不可切分的压缩算法
      - 数据文件大小不一致
      - 数据中有很多空值
    - **reduce task 数据倾斜**
      - shuffle + key 分布不均（主要原因）（**这两个是一个整体，分开不一定会导致数据倾斜**）
      - 那么**为什么 key 分布不均匀？**
        - 填充默认值：某些信息获取不到时被填充了默认值

- 业务本身存在热点：特价商品或者热销商品的购买量显著大于其他商品
- 存在恶意数据：爬虫使用同一 IP 刷广告

- 怎么解决：
  - Map
    - 选择可切分的压缩算法【注意：lzo 压缩文件是可切片的，但是它的可切片特性依赖于其索引，所以需要手动为 lzo 压缩文件创建索引】
    - 让每个数据文件大小基本一致
    - 过滤异常数据：空值，无效数据
  - Reduce
    - 消除 shuffle
      - map 端 join：
        - 适用场景：大表 join 小表【10M】【不适用于大表 join 大表，如果广播的数据很大，可能内存溢出】
        - 对较小的 RDD 创建一个广播变量【数据压缩、高效的通信框架 Netty、BT 协议】，广播给所有的 executor 节点，然后利用 map 算子实现来进行 join 即可
    - 增大 reduce 并行度
      - 计算公式：hashCode(key)%reduce 个数
      - 优点：实现十分简单；缺点：可能缓解数据倾斜，不一定有效果
    - 加盐。给 key 添加随机数强行打散数据
      - 方法 1：
        - 没有固定适应场景
        - 在 map 阶段先将 key 加上前缀或者后缀，shuffle 之后，先对打上随机数后的 key 进行局部聚合，再将各个 key 的随机数去掉后进行全局聚合，就得到了最终的结果
      - 方法 2：
        - 适应场景：join 的时候发生数据倾斜，经检测是由少数 key 的数据量大造成的
        - 为数据量特别大的 key 增加随机前缀或后缀，使得这些 key 分散到不同的 task 中；那么此时数据倾斜的 key 变了，如何 join 呢？于是我想到了将另外一份对应相同

key 的数据与随机前缀或者后缀作笛卡尔积，保证两个表可以 join

- 方法 3：
  - 场景：如果出现数据倾斜的 key 比较多，无法将这些倾斜拆分出来
  - 大表加盐，小表扩容【扩容就是将该表和前缀作笛卡尔积】