**Zookeeper**

Table of Contents

[Zookeeper 6](#_Toc16176737)

[1.（没问题）zookeeper是什么框架？ 6](#_Toc16176738)

[2.有哪些应用场景？ 8](#_Toc16176739)

[3.（没问题）zookeeper有哪几种节点类型？ 8](#_Toc16176740)

[4.（没问题）zookeeper对节点的watch监听通知是永久的吗？ 8](#_Toc16176741)

[5.有哪几种部署模式？ 8](#_Toc16176742)

[6.（没问题）集群中的机器角色都有哪些？ 8](#_Toc16176743)

[7.（没问题）集群最少要几台机器，集群规则是怎样的 9](#_Toc16176744)

[8.（没问题）集群如果有3台机器，挂掉一台集群还能工作吗？挂掉两台呢？ 9](#_Toc16176745)

[9.集群支持动态添加机器吗？ 9](#_Toc16176746)

[10.zookeeper的java客户端都有哪些？ 9](#_Toc16176747)

[11.（没问题）chubby是什么，和zookeeper比你怎么看？ 9](#_Toc16176748)

[12.说几个zookeeper常用的命令。 9](#_Toc16176749)

[13.（没问题）使用什么协议？说一说选举算法及流程 9](#_Toc16176750)

[14.主从架构下，leader 崩溃，leader如何快速选举，数据一致性怎么保证？ 14](#_Toc16176751)

[15.ZAB和Paxos算法的联系和区别 15](#_Toc16176752)

[16.Paxos协议是什么？ 15](#_Toc16176753)

[Spark 19](#_Toc16176754)

[2.讲下shuffle原理？ 19](#_Toc16176755)

[3.看了spark源码，有什么感想？ 19](#_Toc16176756)

[4.什么是列式存储？ 19](#_Toc16176757)

[1、（没问题）spark中的RDD是什么，有哪些特性 19](#_Toc16176758)

[2、（没问题）概述一下spark中的常用算子区别（map、mapPartitions、foreach、foreachPartition） 21](#_Toc16176759)

[3、（没问题）谈谈spark中的宽窄依赖 21](#_Toc16176760)

[4、（没问题）spark中如何划分stage 21](#_Toc16176761)

[5、(没问题)spark-submit的时候如何引入外部jar包 22](#_Toc16176762)

[6、spark 如何防止内存溢出 22](#_Toc16176763)

[7、（没问题）spark中cache和persist的区别 24](#_Toc16176764)

[8、简要描述Spark分布式集群搭建的步骤 24](#_Toc16176765)

[9、(没问题，不过需要把map join看看)spark中的数据倾斜的现象、原因、后果 24](#_Toc16176766)

[10、如何解决spark中的数据倾斜问题 25](#_Toc16176767)

[11.（没问题）写个broadcast看看 27](#_Toc16176768)

[1. spark的有几种部署模式，每种模式特点？ 29](#_Toc16176769)

[2. Spark技术栈有哪些组件，每个组件都有什么功能，适合什么应用场景？ 29](#_Toc16176770)

[3. （没问题）spark有哪些组件 30](#_Toc16176771)

[1. （没问题）spark工作机制 30](#_Toc16176772)

[2. （没问题）Spark应用程序的执行过程 30](#_Toc16176773)

[3. （没问题）driver的功能是什么？ 30](#_Toc16176774)

[4. （没问题）Spark中Work的主要工作是什么？ 31](#_Toc16176775)

[5. （没问题）task有几种类型？2种 31](#_Toc16176776)

[6. 什么是shuffle，以及为什么需要shuffle？ 31](#_Toc16176777)

[7. Spark master HA 主从切换过程不会影响集群已有的作业运行，为什么？ 31](#_Toc16176778)

[8. Spark并行度怎么设置比较合适 31](#_Toc16176779)

[9. Spaek程序执行，有时候默认为什么会产生很多task，怎么修改默认task执行个数？ 31](#_Toc16176780)

[10. Spark中数据的位置是被谁管理的？ 32](#_Toc16176781)

[11. 为什么要进行序列化 32](#_Toc16176782)

[12. Spark如何处理不能被序列化的对象？ 32](#_Toc16176783)

[13. （没问题）Spark提交你的jar包时所用的命令是什么？ 32](#_Toc16176784)

[1. Mapreduce和Spark的相同和区别 32](#_Toc16176785)

[2. 简答说一下hadoop的mapreduce编程模型 33](#_Toc16176786)

[3. 简单说一下hadoop和spark的shuffle相同和差异？ 33](#_Toc16176787)

[4. 简单说一下hadoop和spark的shuffle过程 33](#_Toc16176788)

[5. partition和block的关联 34](#_Toc16176789)

[6. （没问题）Spark为什么比mapreduce快？ 34](#_Toc16176790)

[7. Mapreduce操作的mapper和reducer阶段相当于spark中的哪几个算子？ 34](#_Toc16176791)

[1. RDD机制 34](#_Toc16176792)

[2. RDD的弹性表现在哪几点？ 34](#_Toc16176793)

[3. RDD有哪些缺陷？ 35](#_Toc16176794)

[4. （没问题）什么是RDD宽依赖和窄依赖？ 35](#_Toc16176795)

[1. （没问题）cache和pesist的区别 35](#_Toc16176796)

[2. （没问题）cache后面能不能接其他算子,它是不是action操作？ 35](#_Toc16176797)

[3. （没问题）什么场景下要进行persist操作？ 以下场景会使用persist 35](#_Toc16176798)

[4. （没问题）rdd有几种操作类型？三种！！ 35](#_Toc16176799)

[5. （没问题）reduceByKey是不是action？ 36](#_Toc16176800)

[6. （没问题）collect功能是什么，其底层是怎么实现的？ 36](#_Toc16176801)

[7. （没问题）map与flatMap的区别 36](#_Toc16176802)

[8. （没问题）列举你常用的action？ collect，reduce,take,count,saveAsTextFile等 36](#_Toc16176803)

[9. （没问题）union操作是产生宽依赖还是窄依赖？ 36](#_Toc16176804)

[10. Spark累加器有哪些特点？（需要搞清楚） 36](#_Toc16176805)

[11. （没问题）spark hashParitioner的弊端 36](#_Toc16176806)

[12. （没问题）RangePartitioner分区的原理 36](#_Toc16176807)

[13. （没问题？）Spark中的HashShufle的有哪些不足？这个问题不是在坞丝计划之后已经解决了吗？ 37](#_Toc16176808)

[1. （没问题）如何使用Spark解决TopN问题？（互联网公司常面） https://blog.csdn.net/oopsoom/article/details/25815443 37](#_Toc16176809)

[val data = sc.textFile("/dw/spark/mobile.txt") 37](#_Toc16176810)

[data.map(line=>"""\d+\.\d+\.\d+\.\d+""".r.findAllIn(line).mkString) 37](#_Toc16176811)

[.filter(\_!="") 37](#_Toc16176812)

[.map(word=>(word,1)) 37](#_Toc16176813)

[.reduceByKey(\_+\_) 37](#_Toc16176814)

[.map(word=>(word.\_2,word.\_1)) 37](#_Toc16176815)

[.sortByKey(false) 37](#_Toc16176816)

[.map(word=>(word.\_2,word.\_1)) 37](#_Toc16176817)

[take 50 37](#_Toc16176818)

[2. （没问题）如何使用Spark解决分组排序问题？（互联网公司常面） https://blog.csdn.net/huitoukest/article/details/51273143 37](#_Toc16176819)

[后续： 要了解spark中每个transformation和action函数的意思 39](#_Toc16176820)

[3. （没问题）给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url? 39](#_Toc16176821)

[要记住一亿等于10的八次方 一亿=10的八次方 39](#_Toc16176822)

[要记住一G等于10的九次方字节 1G=10的9次方Byte 39](#_Toc16176823)

[4. （没问题 ）有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M，要求返回频数最高的100个词。 40](#_Toc16176824)

[5. （没问题）现有海量日志数据保存在一个超级大的文件中，该文件无法直接读入内存，其中包含了所有访问百度的IP，要求从中提取某天出访问百度次数最多的那个IP。 40](#_Toc16176825)

[6. 在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。 40](#_Toc16176826)

[7. （没问题）腾讯面试题：给40亿个不重复的unsignedint的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中? 40](#_Toc16176827)

[*一定要记住2的32次方等于40亿多* 40](#_Toc16176828)

[*一定要记住512M=2的32次方个bit，即40亿个bit* 41](#_Toc16176829)

[*一定要记住1G=10亿Byte=80亿个bit.* 41](#_Toc16176830)

[8.（没问题）字节面试题：一亿个数，乱序，放不进内存，如何快速找到中间的5个数？ 41](#_Toc16176831)

[1. Hive中存放是什么？ 42](#_Toc16176832)

[2. Hive与关系型数据库的关系？ 43](#_Toc16176833)

[Spark钨丝计划？（需要了解） 43](#_Toc16176834)

[Hadoop 43](#_Toc16176835)

[Hadoop联邦机制？（需要搞清楚） 43](#_Toc16176836)

[Hadoop断电处理？（需要搞清楚） 43](#_Toc16176837)

[Kerberos 44](#_Toc16176838)

[数据库 44](#_Toc16176839)

[1.超键，候选键，主键，外键？ 44](#_Toc16176840)

[2.什么是事务？事务的四大特性及含义？ 44](#_Toc16176841)

[项目 49](#_Toc16176842)

[大公司面试题 49](#_Toc16176843)

[考拉面经1： 49](#_Toc16176844)

[4、（没问题）ThreadPool，Jdk原生线程池，四个参数详细解释原理，当线程池中poolSize达到corePoolSize且阻塞队列已满，再来一个任务，如何处理 49](#_Toc16176845)

[5、多线程实现同步的方式，互斥同步，非阻塞同步， 51](#_Toc16176846)

[6、reenterlocker和sychronize的区别？解释Synchronize关键字的锁优化技术，偏向锁，轻量级锁，重量级锁，这些锁是如何存储的，偏向锁撤销升级为轻量级锁的过程，结合源码解释 51](#_Toc16176847)

[1. （没问题）说一下 synchronized 底层实现原理？ 51](#_Toc16176848)

[2.（没问题）多线程中 synchronized 锁升级的原理是什么？ 52](#_Toc16176849)

[4.（没问题）reenterlocker和sychronize的区别？ 55](#_Toc16176850)

[Reenterlocker 55](#_Toc16176851)

[Synchronized 55](#_Toc16176852)

[5.为什么ConcurrentHashMap 1.8为什么要使用CAS+Synchronized取代Segment+ReentrantLock 60](#_Toc16176853)

[7、（没问题）volatile关键字语义，内存屏障如何实现，JMM对内存屏障做了哪些优化，volatile的语义增强 60](#_Toc16176854)

[8、指令重排序相关及其happen-before时，数据依赖等 62](#_Toc16176855)

[9、HashMap底层数据结构，如何处理hash冲突，为何HashMap的大小要设置为2的n次幂，为什么IndexFor方法里，需要hash&length-1，为什么HashMap允许null值，resize()过程，多线程下resize为什么会出现死循环，详细解释，结合源码 62](#_Toc16176856)

[10、HashMap jdk1.8之后，为何转链式结构为红黑树，为何长度为8的时候，才转成红黑树 62](#_Toc16176857)

[11、NIO了解吗？   这个。。没答好，属于知识空白 62](#_Toc16176858)

[12、阐述一下快排基本原理。easy 62](#_Toc16176859)

[13、手写单例模式。分别问了多线程情况如何处理，什么时候可以破坏单例模式，怎么防止单例模式被破坏 62](#_Toc16176860)

[14、JMM内存模型，如何保证缓存一致性。（重要）（需要搞清楚） 62](#_Toc16176861)

[15.reenterlocker和sychronize的区别？ 64](#_Toc16176862)

[1、自我介绍 64](#_Toc16176863)

[7、JVM了解吗，介绍一下，讲了垃圾回收机制 64](#_Toc16176864)

[8、Full GC和Minor GC区别，及各自的触发条件 64](#_Toc16176865)

[9、说说CMS垃圾回收器，及其适用场景 64](#_Toc16176866)

[考拉面经2： 64](#_Toc16176867)

[自我介绍 64](#_Toc16176868)

[2、spring java反射 64](#_Toc16176869)

[4、班级表和学生表，利用sql 语句查询每个班级的学生人数 64](#_Toc16176870)

[4、i++原子操作 65](#_Toc16176871)

[5、hashmap和hashtable区别，以及concurrenthashmap 65](#_Toc16176872)

[6、zookeeper 实现分布式锁 65](#_Toc16176873)

[7、团购系统的表设计 67](#_Toc16176874)

[8、（没问题）悲观锁 和乐观锁 以及sql如何实现 67](#_Toc16176875)

[什么是AbstractQueuedSynchronizer？ 68](#_Toc16176876)

[JVM优化经验（重要重要重要）（需要搞清楚） 69](#_Toc16176877)

[360面试总结 70](#_Toc16176878)

[hdfs中的读写操作？ 70](#_Toc16176879)

[为什么三次握手 四次挥手？ 70](#_Toc16176880)

[接口和抽象类的区别？接口中是否可以放属性？ 71](#_Toc16176881)

[hashmap中key有什么要求？ 71](#_Toc16176882)

[hashmap的get的流程？ 71](#_Toc16176883)

[equal 和等号的区别 72](#_Toc16176884)

[java8 9的新功能 72](#_Toc16176885)

[一个线程接着一个线程执行 有什么方法？ 72](#_Toc16176886)

[创建线程的方式 73](#_Toc16176887)

[如何停止一个线程？ 73](#_Toc16176888)

Zookeeper

1.（没问题）zookeeper是什么框架？

答：zookeeper是一个分布式协调服务，可用于维护配置信息，统一命名，分布式数据同步，分组服务，分布式锁，主从选举等等服务。

底层的数据结构是树状的目录结构，每个节点可包含数据和子节点（短暂结点除外）

其中节点类型包括

* PERSISTENT  （持久化节点）

客户端与zk服务器断开之后，该节点依然存在

* PERSISTENT\_SEQUENTIAL（持久顺序节点/test0000000019)

客户端与zk服务器断开之后，该节点依然存在，此外zk在创建该节点时它的名称后面附加了一个单调递增的顺序编号

这类持久化节点可用于统一命名服务，维护配置信息，分布式数据同步，分组等等服务。

* EPHEMERAL   （短暂节点）

客户端与zk服务器断开之后，该节点就被删除

* EPHEMERAL\_SEQUENTIAL  （短暂顺序节点）

客户端与zk服务器断开之后，该节点就被删除，此外zk在创建该节点时它的名称后面附加了一个单调递增的顺序编号。

这类临时化节点可用于分布式锁，主从选举，动态上下线感知等等功能。

Zookeeper还提供对节点的通知服务，并且监听是一次性的。客户端可以对某个节点建立一个watch事件，包括

get path [watch]: 监听数据通知

ls path [watch]：监听子节点的增删通知

stat path [watch]：

当节点发送变化时，客户端会受到zk的通知，客户端可以根据节点的变化做出业务上的改动。

分布式同步，分布式锁，等等利用到了监听功能。

------------------------------------------

Zookeeper是一个分布式协调服务；就是为用户的分布式应用程序提供协调服务

A、zookeeper是为别的分布式程序服务的

* Zookeeper本身就是一个分布式程序（只要有半数以上节点存活，zk就能正常服务）
* Zookeeper所提供的服务涵盖：**主从协调、服务器节点动态上下线、统一配置管理、分布式共享锁、统一名称服务**……
* 虽然说可以提供各种服务，但是zookeeper在底层其实只提供了两个功能：

管理(存储，读取)用户程序提交的状态数据；

    并为用户程序提供数据节点监听服务；

Zookeeper集群的角色：  Leader 和  follower  （Observer）

只要集群中有半数以上节点存活，集群就能提供服务。

2.有哪些应用场景？

* Zookeeper所提供的服务涵盖：**Leader选举，主从协调、服务器节点动态上下线、统一配置管理、分布式共享锁、统一名称服务，负载均衡**……
* 虽然说可以提供各种服务，但是zookeeper在底层其实只提供了两个功能：
  + 管理(存储，读取)用户程序提交的状态数据；
  + 并为用户程序提供数据节点监听服务；

3.（没问题）zookeeper有哪几种节点类型？

其中节点类型包括

* PERSISTENT  （持久化节点）

客户端与zk服务器断开之后，该节点依然存在

* PERSISTENT\_SEQUENTIAL（持久顺序节点/test0000000019)

客户端与zk服务器断开之后，该节点依然存在，此外zk在创建该节点时它的名称后面附加了一个单调递增的顺序编号

这类持久化节点可用于统一命名服务，维护配置信息，分布式数据同步，分组等等服务。

* EPHEMERAL   （短暂节点）

客户端与zk服务器断开之后，该节点就被删除

* EPHEMERAL\_SEQUENTIAL  （短暂顺序节点）

客户端与zk服务器断开之后，该节点就被删除，此外zk在创建该节点时它的名称后面附加了一个单调递增的顺序编号。

这类临时化节点可用于分布式锁，主从选举，动态上下线感知等等功能。

4.（没问题）zookeeper对节点的watch监听通知是永久的吗？

一次性的，需要在每次执行监听事件的时候，重新监听

5.有哪几种部署模式？

6.（没问题）集群中的机器角色都有哪些？

Leader follower，observer

7.（没问题）集群最少要几台机器，集群规则是怎样的

2N+1台，N>0,3台

8.（没问题）集群如果有3台机器，挂掉一台集群还能工作吗？挂掉两台呢？

集群需要一半以上的机器可用，所以3台机器挂掉一台，还能工作。2台不能。

9.集群支持动态添加机器吗？

3.5版本开始支持

10.zookeeper的java客户端都有哪些？

zk自带的zkClient，和apache开源的curator。

11.（没问题）chubby是什么，和zookeeper比你怎么看？

chubby是google的，完全实现paxos算法，不开源，zookeeper是chubby的开源实现，使用zab协议做一致性协议，paxos的变种。

12.说几个zookeeper常用的命令。

org.apache.zookeeper.Zookeeper是客户端入口主类，负责建立与server的会话

它提供了表 1 所示几类主要方法

|  |  |
| --- | --- |
| 功能 | 描述 |
| create | 在本地目录树中创建一个节点 |
| delete | 删除一个节点 |
| exists | 测试本地是否存在目标节点 |
| get/set data | 从目标节点上读取 / 写数据 |
| get/set ACL | 获取 / 设置目标节点访问控制列表信息 |
| get children | 检索一个子节点上的列表 |
| sync | 等待要被传送的数据 |

13.（没问题）使用什么协议？说一说选举算法及流程

Zookeeper的核心是zab协议，也叫原子广播协议，目的是实现zookeeper中各个节点的数据一致性问题。

Zab如何实现数据一致性问题的呢？

zk节点有两种角色leader和follower，还有一个核心概念zxid，

在 ZAB 协议的事务编号 Zxid 设计中，Zxid 是一个 64 位的数字，

* 低 32 位是一个简单的单调递增的计数器，针对客户端每一个事务请求，计数器加 1；
* 高 32 位则代表 Leader 周期 epoch 的编号，每个当选产生一个新的 Leader 服务器，就会从这个 Leader 服务器上取出其本地日志中最大事务的ZXID，并从中读取 epoch 值，然后加 1，以此作为新的 epoch，并将低 32 位从 0 开始计数。
* epoch：可以理解为当前集群所处的年代或者周期，每个 leader 就像皇帝，都有自己的年号，所以每次改朝换代，leader 变更之后，都会在前一个年代的基础上加 1。这样就算旧的 leader 崩溃恢复之后，也没有人听他的了，因为 follower 只听从当前年代的 leader 的命令。

Zab协议包括有两种模式，一种是恢复模式，一种是广播模式，

* 恢复模式指的是：当服务器启动或者当leader崩溃后，zab就进入了恢复模式，首先会进行leader快速选举，
  1. FLE 会选举拥有最新提议历史（lastZixd最大）的节点作为 leader，这样就省去了发现最新提议的步骤。这是基于拥有最新提议的节点也有最新提交记录的前提。选举流程如下：
     + 选epoch最大的
     + epoch相等，选 zxid 最大的
     + epoch和zxid都相等，选择serverID最大的（就是我们配置zoo.cfg中的myid）
  2. 当leader被选举出来之后，进入状态同步阶段，同步阶段会生成一个新的epoch，并followers同步leader的历史日志。之后恢复模式结束。
* 广播模式：一旦leader已经和多数的follower进行状态同步之后，他就可以开始广播消息了，广播模式类似两阶段提交，在广播模式中，所有写请求都会转发给leader，再由leader广播更新日志请求，如果收到半数以上的follower的ACK，再向followers发送提交事务请求。在广播事务之前，leader会为这个事务分配一个zxid，每个事务按照zxid的先后顺序来进行处理，以保证消息严格的因果关系。

**ZAB协议**

使用zab协议，是zookeeper原子广播协议的简称。

Zookeeper 客户端会随机连接到 Zookeeper 集群的一个节点，

* 如果是读请求，就直接从当前节点中读取数据；
* 如果是写请求，那么节点就会向 leader 提交事务，leader 会广播事务，只要有超过半数节点写入成功，该写请求就会被提交（类 2PC 协议）。

**ZAB协议的四个阶段**

**先说下关于zab的一些基本概念**

1.quorum：集群中超过半数的节点集合

2.在zookeeper集群中的节点有四种选举状态：

* **following**：当前节点是跟随者，服从 leader 节点的命令
* **leading**：当前节点是 leader，负责协调事务
* **election/looking**：节点处于选举状态

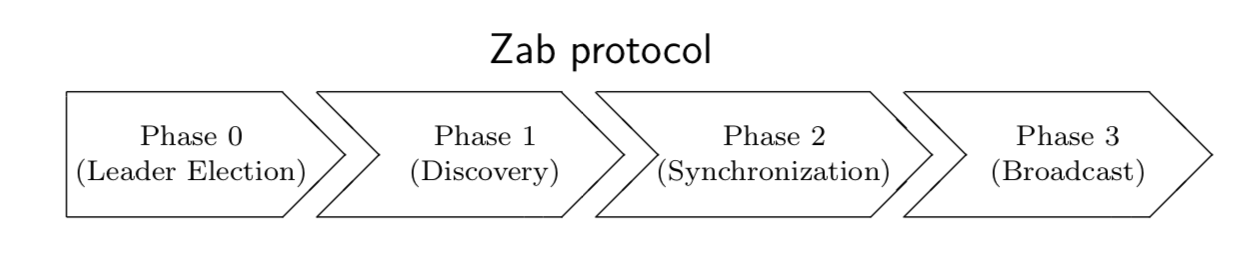
代码实现中多了一种：observing 状态，这是 Zookeeper 引入 Observer 之后加入的，Observer 不参与选举，是只读节点，跟 ZAB 协议没有关系。

3.在zookeeper集群中的节点数据的持久化信息：

* **history**：当前节点接收到事务提议的 log
* **acceptedEpoch**：follower 已经接受的 leader 更改年号的 NEWEPOCH 提议
* **currentEpoch**：当前所处的年代
* **lastZxid**：history 中最近接收到的提议的 zxid （最大的）
* **serverID**：我们配置zoo.cfg中的myid

|  |
| --- |
| 在 ZAB 协议的事务编号 Zxid 设计中，Zxid 是一个 64 位的数字，   * 低 32 位是一个简单的单调递增的计数器，针对客户端每一个事务请求，计数器加 1； * 高 32 位则代表 Leader 周期 epoch 的编号，每个当选产生一个新的 Leader 服务器，就会从这个 Leader 服务器上取出其本地日志中最大事务的ZXID，并从中读取 epoch 值，然后加 1，以此作为新的 epoch，并将低 32 位从 0 开始计数。 * epoch：可以理解为当前集群所处的年代或者周期，每个 leader 就像皇帝，都有自己的年号，所以每次改朝换代，leader 变更之后，都会在前一个年代的基础上加 1。这样就算旧的 leader 崩溃恢复之后，也没有人听他的了，因为 follower 只听从当前年代的 leader 的命令。\* |

#### ZAB协议的四个阶段

****

#### Phase 0: Leader election（选举阶段）

节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。只有到达 Phase 3 准 leader 才会成为真正的 leader。这一阶段的目的是就是为了选出一个准 leader，然后进入下一个阶段。

协议并没有规定详细的选举算法，后面我们会提到实现中使用的 Fast Leader Election。

#### Phase 1: Discovery（发现阶段）

在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。这个一阶段的主要目的是发现当前大多数节点接收的最新提议，并且准 leader 生成新的 epoch，让 followers 接受，更新它们的 acceptedEpoch。

一个 follower 只会连接一个 leader，如果有一个节点 f 认为另一个 follower p 是 leader，f 在尝试连接 p 时会被拒绝，f 被拒绝之后，就会进入 Phase 0。

#### Phase 2: Synchronization（同步阶段）

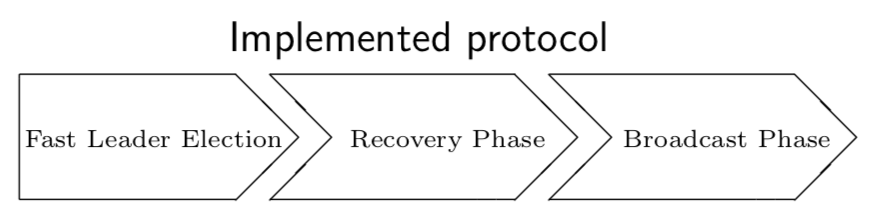
同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。只有当 quorum 都同步完成，准 leader 才会成为真正的 leader。follower 只会接收 zxid 比自己的 lastZxid 大的提议。

#### Phase 3: Broadcast（广播阶段）

到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

值得注意的是，ZAB 提交事务并不像 2PC 一样需要全部 follower 都 ACK，只需要得到 quorum （超过半数的节点）的 ACK 就可以了。

**zookeeper中zab协议的实现**

****

协议的 Java 版本实现跟上面的定义有些不同，选举阶段使用的是 ***Fast Leader Election（FLE）***，它包含了 Phase 1 的发现职责。因为 FLE 会选举拥有最新提议历史的节点作为 leader，这样就省去了发现最新提议的步骤。实际的实现将 Phase 1 和 Phase 2 合并为 Recovery Phase（恢复阶段）。所以，ZAB 的实现只有三个阶段：

* **Fast Leader Election（快速选举阶段）**
* **Recovery Phase（恢复阶段）**
* **Broadcast Phase（广播阶段）**

### Fast Leader Election(快速选举阶段)

前面提到 FLE 会选举拥有最新提议历史（lastZixd最大）的节点作为 leader，这样就省去了发现最新提议的步骤。这是基于拥有最新提议的节点也有最新提交记录的前提。

**选举流程**

1. 选*epoch*最大的
2. *epoch*相等，选 *zxid* 最大的
3. *epoch*和*zxid*都相等，选择*serverID*最大的（就是我们配置zoo.cfg中的myid）

节点在选举开始都默认投票给自己，当接收其他节点的选票时，会根据上面的条件更改自己的选票并重新发送选票给其他节点，当有一个节点的得票超过半数，该节点会设置自己的状态为 leading，其他节点会设置自己的状态为 following。

### Recovery Phase（恢复阶段）

这一阶段 follower 发送它们的 lastZixd 给 leader，leader 根据 lastZixd 决定如何同步数据。这里的实现跟前面 Phase 2 有所不同：Follower 收到 TRUNC 指令会中止 L.lastCommittedZxid 之后的提议，收到 DIFF 指令会接收新的提议。

|  |
| --- |
| history.lastCommittedZxid：最近被提交的提议的 zxid  history:oldThreshold：被认为已经太旧的已提交提议的 zxid |

### Broadcast Election(广播阶段)

和zab协议里的一样，到了这一阶段，说明Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

值得注意的是，ZAB 提交事务并不像 2PC 一样需要全部 follower 都 ACK，只需要得到 quorum （超过半数的节点）的 ACK 就可以了。

14.主从架构下，leader 崩溃，leader如何快速选举，数据一致性怎么保证？

leader 崩溃之后，集群会通过**Fast Leader Election**选出新的 leader，leader 的选举只需要超过半数的节点投票即可，这样不需要等待所有节点的选票，能够尽早选出 leader。选举流程如下：

**一个完全新的集群leader选举流程**

举例：

目前有5台服务器，每台服务器均没有数据，它们的编号分别是1,2,3,4,5,按编号依次启动，它们的选择举过程如下：

1. 服务器1启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器1的状态一直属于Looking。
2. 服务器2启动，给自己投票，同时与之前启动的服务器1交换结果，由于服务器2的编号大所以服务器2胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是LOOKING。
3. 服务器3启动，给自己投票，同时与之前启动的服务器1,2交换信息，由于服务器3的编号最大所以服务器3胜出，此时投票数正好大于半数，所以服务器3成为领导者，服务器1,2成为小弟。
4. 服务器4启动，给自己投票，同时与之前启动的服务器1,2,3交换信息，尽管服务器4的编号大，但之前服务器3已经胜出，所以服务器4只能成为小弟。
5. 服务器5启动，后面的逻辑同服务器4成为小弟。

**一个非完全新的集群的leader选举流程**

1. 选*epoch*最大的
2. *epoch*相等，选 *zxid* 最大的
3. *epoch*和*zxid*都相等，选择*serverID*最大的（就是我们配置zoo.cfg中的myid）

节点在选举开始都默认投票给自己，当接收其他节点的选票时，会根据上面的条件更改自己的选票并重新发送选票给其他节点，当有一个节点的得票超过半数，该节点会设置自己的状态为 leading，其他节点会设置自己的状态为 following。

选举完成之后

然后就会进入恢复阶段，新的 leader 具有所有已经提交的提议，因此它会保证让 followers 同步已提交的提议，丢弃未提交的提议（以 leader 的记录为准），这就保证了整个集群的数据一致性。

15.ZAB和Paxos算法的联系和区别

#### 联系

* 两者都存在一个类似于Leader进程的角色，由其负责协调多个Follower进程的运行。
* Leader进程都会等待超过半数的Follower做出正确的反馈后，才会将一个提案进行提交。
* 在ZAB协议中，每个Proposal都包含一个epoch值，用来代表当前的Leader周期，在Paxos算法中，同样存在这样的一个标识，只是名字变成了Ballot。

#### 本质区别

**两者设计目标不太一样。ZAB协议主要用于构建一个高可用的分布式数据主备系统，而Paxos算法则是用于构建一个分布式的一致性状态机系统**

16.Paxos协议是什么？

## Basic Paxos 协议

Paxos 协议是一个解决分布式系统中 多个节点之间就某个值（提案value）达成一致（决议）的通信协议，也就是说，每个节点 提出的提案 会 对 同一个 提案内容 达成一致(知悉 且 要commit成功)。下面简称提案ID为 ProposalID，提案内容为value。

### 算法(提案的提出与批准)主要分为两个阶段:

**第一阶段 Prepare**

**P1a：Proposer 发送 Prepare请求**

Proposer 生成全局唯一且递增的ProposalID，向 Paxos 集群的所有机器发送 Prepare请求，这里不携带value，只携带 ProposalID 。

**P1b：Acceptor 应答 Prepare**

Acceptor 收到 Prepare请求 后，判断：收到的ProposalID 是否 比 之前已响应的所有提案的ProposalID 大：

* 如果是，则：

(1) 在本地持久化 ProposalID，可记为Max\_ProposalID。

(2) 回复请求，并带上 已Accept的提案中 ProposalID 最大的 value（若此时还没有已Accept的提案，则返回value为空）。

(3) 做出承诺：不会Accept 任何 小于 Max\_ProposalID的提案。

* 如果否：不回复。

**第二阶段 Accept**

**P2a：Proposer 发送 Accept**

经过一段时间后，Proposer 收集到一些 Prepare 回复，有下列几种情况：

(1) 回复数量 > 一半的Acceptor数量，且 所有的回复的 value 都为空，则 Porposer发出accept请求，并带上 自己指定的value。

(2) 回复数量 > 一半的Acceptor数量，且 有的回复 value 不为空，则 Porposer发出accept请求，并带上 回复中 ProposalID最大 的value(作为自己的提案内容)。

(3) 回复数量 <= 一半的Acceptor数量，则 尝试 更新生成更大的 ProposalID，再转 P1a 执行。

**P2b：Acceptor 应答 Accept**

Accpetor 收到 Accpet请求 后，判断：

(1) 收到的ProposalID >= Max\_ProposalID (一般情况下是 等于)，则 回复 提交成功，并 持久化 ProposalID 和value。

(2) 收到的ProposalID < Max\_ProposalID，则 不回复 或者 回复 提交失败。

**P2c: Proposer 统计投票**

经过一段时间后，Proposer 收集到一些 Accept 回复提交成功，有几种情况：

(1) 回复数量 > 一半的Acceptor数量，则表示 提交value成功。此时，可以发一个广播给所有Proposer、Learner，通知它们 已commit的value。

(2) 回复数量 <= 一半的Acceptor数量，则 尝试 更新生成更大的 ProposalID，再转 P1a 执行。

(3) 收到 一条提交失败的回复，则 尝试 更新生成更大的 ProposalID，再转 P1a 执行。

**最后，经过多轮投票后，达到的结果是：**

(1) 所有Proposer都 提交提案成功了，且提交的value是同一个value。

(2) 过半数的 Acceptor都提交成功了，且提交的是 同一个value。

### Paxos 协议 的几个约束：

* P1: 一个Acceptor必须接受(accept)第一次收到的提案;
* P2a: 一旦一个具有value v的提案被批准(chosen)，那么之后任何Acceptor 再次接受(accept)的提案必须具有value v;
* P2b: 一旦一个具有value v的提案被批准(chosen)，那么以后任何 Proposer 提出的提案必须具有value v;
* P2c: 如果一个编号为n的提案具有value v，那么存在一个多数派，要么他们中所有人都没有接受(accept)编号小于n的任何提案，要么他们已经接受(accpet)的所有编号小于n的提案中编号最大的那个提案具有value v;

### 常见的疑问、及异常处理：

1. Paxos算法的核心思想是什么？

(1) 引入了 多个Acceptor，避免 单个Acceptor成为单点。

Proposer用 更大ProposalID 来抢占 临时的访问权，避免 其中一个 Proposer崩溃宕机 导致 死锁。

(2) 保证一个ProposalID，只有一个Proposer能进行到第二阶段运行，Proposer按照ProposalID递增的顺序依次运行。

(3) 新ProposalID 的 proposer 采用 后者认同前者的思路运行。

* 在肯定 旧ProposalID 还没有生成确定的value (Acceptor 提交成功一个value)时，新ProposalID 会提交自己的value，不会冲突。
* 一旦 旧ProposalID 生成了确定的value，新ProposalID 肯定可以获取到此值，并且认同此值。

容错要求：

(1) 半数以内的Acceptor失效、任意数量的Proposer 失效，都能运行。

(2) 一旦value值被确定，即使 半数以内的Acceptor失效，此值也可以被获取，并不再修改。

1. 工程实践中 ProposalID 怎么定？

在《Paxos made simple》中提到，推荐Proposer从不相交的数据集合中进行选择，例如系统有5个Proposer，则可为每一个Proposer分配一个标识j(0~4)，则每一个proposer每次提出决议的编号可以为5\*i + j(i可以用来表示提出议案的次数)。

在实践过程中，可以用 时间戳 + 提出提案的次数 + 机器 IP/机器ID 来保证唯一性和递增性。

1. 如何保证 更大的ProposalID的Proposer 不会破坏 已经达成的确定性取值value？

在P2a阶段中，Proposer会以 所有回复中ProposalID最大 的value 作为自己的提案内容。

其中，prepare阶段的目的有两个: 1) 检查是否有被批准的值，如果有，就改用批准的值。2) 如果之前的提案还没有被批准，则阻塞掉他们以便不让他们和我们发生竞争，当然最终由ProposalID 的大小决定。

1. Paxos协议的活锁问题

新轮次的抢占会使旧轮次停止运行，如果每一轮在第二阶段执行成功之前 都 被 新一轮抢占，则导致活锁。怎么解决？

这个问题在实际应用会发生地比较少，一般可通过 随机改变 ProposalID的增长幅度 或者 增加Proposer发送新一轮提案的间隔 来解决。

1. Paxos 运行过程中，半数以内的Acceptor失效，都能运行。为什么？

(1) 如果 半数以内的Acceptor失效时 还没确定最终的value，此时，所有Proposer会竞争 提案的权限，最终会有一个提案会 成功提交。之后，会有半过数的Acceptor以这个value提交成功。

(2) 如果 半数以内的Acceptor失效时 已确定最终的value，此时，所有Proposer提交前 必须以 最终的value 提交，因为，一个Proposer要拿到过半数的accept响应，必须同一个已提交的Acceptor存在交集，故会在P2a阶段中会继续沿用该value。

1. 若两个Proposer以不同的ProposalID，在进行到P2a阶段，收到的prepare回复的value值都为空，则两个proposer都以自己的值作为value(提案内容)，向Acceptor提交请求，最后，两个proposer都会认为自己提交成功了吗？

不会，因为Acceptor会根据ProposalID，批准执行最大的ProposalID的value，另一个会回复 执行失败。当proposer收到执行失败的回复时，就知道：当前 具有更大的ProposalID的提案提交成功了。

1. 由于大ProposalID 可以 抢占 小ProposalID 的提交权限，如果 此时 Acceptor还没有一个确定性取值，有一个具有最大ProposalID的proposer进行到P2a阶段了，但这时 这个proposer挂了，会造成一种死锁状态（小ProposalID的会提交失败，但是 具有最大ProposalID的proposer却不能提交accept请求），如何解决这种死锁状态？

不会产生这种死锁状态，acceptor回复提交失败后，proposer再生成 更大的ProposalID，下一轮可以用自己value提交成功。

### ZAB和Paxos算法的联系和区别

#### 联系

* 两者都存在一个类似于Leader进程的角色，由其负责协调多个Follower进程的运行。
* Leader进程都会等待超过半数的Follower做出正确的反馈后，才会将一个提案进行提交。
* 在ZAB协议中，每个Proposal都包含一个epoch值，用来代表当前的Leader周期，在Paxos算法中，同样存在这样的一个标识，只是名字变成了Ballot。

#### 本质区别

两者设计目标不太一样。ZAB协议主要用于构建一个高可用的分布式数据主备系统，而Paxos算法则是用于构建一个分布式的一致性状态机系统

Spark

2.讲下shuffle原理？

3.看了spark源码，有什么感想？

4.什么是列式存储？

<https://zhuanlan.zhihu.com/p/35622907>

1、（没问题）spark中的RDD是什么，有哪些特性

答：RDD全称叫弹性分布式数据集，首先它是个数据集，可以存放不同类型的数据，第二他是分布式的，分片的，表现在可以在集群中并行计算的，第三，它是弹性的，表现在

* 第一，RDD中的数据可以存储在内存或者磁盘的
* 第二，数据是可以分片的，分片可以由碎片合成大的，也可以由大分成各个小分片。
* 第三，具有基于血统的高效容错机制。
* 第四，task如果失败，会自动进行特定次数的重试，stage如果失败，也会自动进行特定次数的重试，而且只会计算失败的分片。
* 第五，数据调度弹性，DAT task 调度与资源无关。

此外，RDD还具有五大重要的属性：(PCDPL)

* + P: A list of partitions   
    一个分区列表，RDD中的数据都存在一个分区列表里面，这个特性决定了RDD是可以进行并行计算的
  + C: A function for computing each split   
    作用在每一个分区中的函数，这个上层的实现，就有transformation函数和action函数，transformation函数会将RDD转换成另一个RDD，例如，Map，而action函数会触发一个job，返回一个值，例如collect。
  + D:A list of dependencies on other RDDs   
    一个RDD依赖于其他多个RDD，这个点很重要，RDD的依赖关系形成一个DAG图， RDD的容错机制就是依据这个特性而来的。
  + P: Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)   
    可选的，针对于kv类型的RDD才具有这个特性，作用是决定了数据分片的分片函数，默认的哈希分片，这种情况容易造成数据倾斜，还有一个范围分片
  + spark hashParitioner的弊端
    - 分区原理：对于给定的key，计算其hashCode
    - 弊端是数据不均匀，容易导致数据倾斜
  + RangePartitioner分区的原理
* 尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，也就是说一个分区中的元素肯定都是比另一个分区内的元素小或者大
* 分区内的元素是不能保证顺序的
* 简单的说就是将一定范围内的数映射到某一个分区内
  + L: Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)   
    可选项，计算每个分片的优先位置列表，思想是移动数据不如移动计算。例如：尽可能的在hdfs文件的block位置附近计算。

2、（没问题）概述一下spark中的常用算子区别（map、mapPartitions、foreach、foreachPartition）

* map：用于遍历RDD,将函数f应用于每一个元素，返回新的RDD(transformation算子)。
* foreach:用于遍历RDD,将函数f应用于每一个元素，无返回值(action算子)。
* mapPartitions:用于遍历操作RDD中的每一个分区，返回生成一个新的RDD（transformation算子）。
* foreachPartition: 用于遍历操作RDD中的每一个分区。无返回值(action算子)。
* 总结：一般使用mapPartitions或者foreachPartition算子比map和foreach更加高效，推荐使用。

3、（没问题）谈谈spark中的宽窄依赖

* RDD和它依赖的父RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。
  + 宽依赖：指的是多个子RDD的Partition会依赖同一个父RDD的Partition（一个父RDD对于多个子RDD）
  + 窄依赖：指的是每一个父RDD的Partition最多被子RDD的一个Partition使用。(一个父RDD对应一个子RDD)

4、（没问题）spark中如何划分stage

* 1.Spark Application中可以因为不同的Action触发众多的job，一个Application中可以有很多的job，每个job是由一个或者多个Stage构成的，后面的Stage依赖于前面的Stage，也就是说只有前面依赖的Stage计算完毕后，后面的Stage才会运行。
* 2.Stage划分的依据就是宽依赖，何时产生宽依赖，例如reduceByKey,groupByKey的算子，会导致宽依赖的产生。
* 3.由Action（例如collect）导致了SparkContext.runJob的执行，最终导致了DAGScheduler中的submitJob的执行，其核心是通过发送一个case class JobSubmitted对象给eventProcessLoop。   
  eventProcessLoop是DAGSchedulerEventProcessLoop的具体实例，而DAGSchedulerEventProcessLoop是eventLoop的子类，具体实现EventLoop的onReceive方法，onReceive方法转过来回调doOnReceive
* 4.在doOnReceive中通过模式匹配的方法把执行路由到
* 5.在handleJobSubmitted中首先创建finalStage，创建finalStage时候会建立父Stage的依赖链条
* 总结：以来是从代码的逻辑层面上来展开说的，可以简单点说：先介绍什么是RDD中的宽窄依赖，然后在根据DAG有向无环图进行划分，从当前job的最后一个算子往前推，遇到宽依赖，那么当前在这个批次中的所有算子操作都划分成一个stage,然后继续按照这种方式在继续往前推，如在遇到宽依赖，又划分成一个stage,一直到最前面的一个算子。最后整个job会被划分成多个stage,而stage之间又存在依赖关系，后面的stage依赖于前面的stage。

5、(没问题)spark-submit的时候如何引入外部jar包

* 在通过spark-submit提交任务时，可以通过添加配置参数来指定   
  + –driver-class-path 外部jar包
  + –jars 外部jar包

6、spark 如何防止内存溢出

* driver端的内存溢出
  + 可以增大driver的内存参数：spark.driver.memory (default 1g)
  + 这个参数用来设置Driver的内存。在Spark程序中，SparkContext，DAGScheduler都是运行在Driver端的。对应rdd的Stage切分也是在Driver端运行，如果用户自己写的程序有过多的步骤，切分出过多的Stage，这部分信息消耗的是Driver的内存，这个时候就需要调大Driver的内存。
* map过程产生大量对象导致内存溢出 ？？？？？？？？？？
  + 这种溢出的原因是在单个map中产生了大量的对象导致的，例如：rdd.map(x=>for(i <- 1 to 10000) yield i.toString)，这个操作在rdd中，每个对象都产生了10000个对象，这肯定很容易产生内存溢出的问题。针对这种问题，在不增加内存的情况下，可以通过减少每个Task的大小，以便达到每个Task即使产生大量的对象Executor的内存也能够装得下。具体做法可以在会产生大量对象的map操作之前调用repartition方法，分区成更小的块传入map。例如：rdd.repartition(10000).map(x=>for(i <- 1 to 10000) yield i.toString)。   
    面对这种问题注意，不能使用rdd.coalesce方法，这个方法只能减少分区，不能增加分区，不会有shuffle的过程。
* 数据不平衡导致内存溢出
  + 数据不平衡除了有可能导致内存溢出外，也有可能导致性能的问题，解决方法和上面说的类似，就是调用repartition重新分区。这里就不再累赘了。
* shuffle后内存溢出
  + shuffle内存溢出的情况可以说都是shuffle后，单个文件过大导致的。在Spark中，join，reduceByKey这一类型的过程，都会有shuffle的过程，在shuffle的使用，需要传入一个partitioner，大部分Spark中的shuffle操作，默认的partitioner都是HashPatitioner，默认值是父RDD中最大的分区数,这个参数通过spark.default.parallelism控制(在spark-sql中用spark.sql.shuffle.partitions) ， spark.default.parallelism参数只对HashPartitioner有效，所以如果是别的Partitioner或者自己实现的Partitioner就不能使用spark.default.parallelism这个参数来控制shuffle的并发量了。如果是别的partitioner导致的shuffle内存溢出，就需要从partitioner的代码增加partitions的数量。
* standalone模式下资源分配不均匀导致内存溢出
  + 在standalone的模式下如果配置了–total-executor-cores 和 –executor-memory 这两个参数，但是没有配置–executor-cores这个参数的话，就有可能导致，每个Executor的memory是一样的，但是cores的数量不同，那么在cores数量多的Executor中，由于能够同时执行多个Task，就容易导致内存溢出的情况。这种情况的解决方法就是同时配置–executor-cores或者spark.executor.cores参数，确保Executor资源分配均匀。
* 使用rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)代替rdd.cache()
  + rdd.cache()和rdd.persist(Storage.MEMORY\_ONLY)是等价的，在内存不足的时候rdd.cache()的数据会丢失，再次使用的时候会重算，而rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)在内存不足的时候会存储在磁盘，避免重算，只是消耗点IO时间。

7、（没问题）spark中cache和persist的区别

* cache：缓存数据，默认是缓存在内存中，其本质还是调用persist
* persist:缓存数据，有丰富的数据缓存策略。数据可以保存在内存也可以保存在磁盘中，使用的时候指定对应的缓存级别就可以了。
  + Memory-Only

8、简要描述Spark分布式集群搭建的步骤

* 地球人都知道
* 这里可以概述下如何搭建高可用的spark集群（HA）   
  + 主要是引入了zookeeper

9、(没问题，不过需要把map join看看)spark中的数据倾斜的现象、原因、后果

* (1)、数据倾斜的现象
  + 多数task执行速度较快,少数task执行时间非常长，或者等待很长时间后提示你内存不足，执行失败。
* (2)、数据倾斜的原因
  + 数据问题
    - 1、key本身分布不均衡（包括大量的key为空）
    - 2、key的设置不合理
  + spark使用问题
    - 1、shuffle时的并发度不够
    - 2、计算方式有误
* (3)、数据倾斜的后果
  + 1、spark中的stage的执行时间受限于最后那个执行完成的task,因此运行缓慢的任务会拖垮整个程序的运行速度（分布式程序运行的速度是由最慢的那个task决定的）。
  + 2、过多的数据在同一个task中运行，将会把executor撑爆。

10、如何解决spark中的数据倾斜问题

* 发现数据倾斜的时候，不要急于提高executor的资源，修改参数或是修改程序，首先要检查数据本身，是否存在异常数据。
  + 1、数据问题造成的数据倾斜
    - 找出异常的key
      * 如果任务长时间卡在最后最后1个(几个)任务，首先要对key进行抽样分析，判断是哪些key造成的。   
        选取key，对数据进行抽样，统计出现的次数，根据出现次数大小排序取出前几个。
      * 比如: df.select(“key”).sample(false,0.1).(k=>(k,1)).reduceBykey(+).map(k=>(k.\_2,k.\_1)).sortByKey(false).take(10)
      * 如果发现多数数据分布都较为平均，而个别数据比其他数据大上若干个数量级，则说明发生了数据倾斜。
    - 经过分析，倾斜的数据主要有以下三种情况:   
      * 1、null（空值）或是一些无意义的信息()之类的,大多是这个原因引起。
      * 2、无效数据，大量重复的测试数据或是对结果影响不大的有效数据。
      * 3、有效数据，业务导致的正常数据分布。
    - 解决办法   
      * 第1，2种情况，直接对数据进行过滤即可（因为该数据对当前业务不会产生影响）。
      * 第3种情况则需要进行一些特殊操作，常见的有以下几种做法   
        + (1) 隔离执行，将异常的key过滤出来单独处理，最后与正常数据的处理结果进行union操作。
        + (2) 对key先添加随机值，进行操作后，去掉随机值，再进行一次操作。
        + (3) 使用reduceByKey 代替 groupByKey(reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义.)
        + (4) 使用map join。
    - 案例   
      * 如果使用reduceByKey因为数据倾斜造成运行失败的问题。具体操作流程如下:   
        + (1) 将原始的 key 转化为 key + 随机值(例如Random.nextInt)
        + (2) 对数据进行 reduceByKey(func)
        + (3) 将 key + 随机值 转成 key
        + (4) 再对数据进行 reduceByKey(func)
    - 案例操作流程分析：   
      * 假设说有倾斜的Key，我们给所有的Key加上一个随机数，然后进行reduceByKey操作；此时同一个Key会有不同的随机数前缀，在进行reduceByKey操作的时候原来的一个非常大的倾斜的Key就分而治之变成若干个更小的Key，不过此时结果和原来不一样，怎么破？进行map操作，目的是把随机数前缀去掉，然后再次进行reduceByKey操作。（当然，如果你很无聊，可以再次做随机数前缀），这样我们就可以把原本倾斜的Key通过分而治之方案分散开来，最后又进行了全局聚合
      * 注意1: 如果此时依旧存在问题，建议筛选出倾斜的数据单独处理。最后将这份数据与正常的数据进行union即可。
      * 注意2: 单独处理异常数据时，可以配合使用Map Join解决。
  + 2、spark使用不当造成的数据倾斜
    - 提高shuffle并行度
      * dataFrame和sparkSql可以设置spark.sql.shuffle.partitions参数控制shuffle的并发度，默认为200。
      * rdd操作可以设置spark.default.parallelism控制并发度，默认参数由不同的Cluster Manager控制。
      * 局限性: 只是让每个task执行更少的不同的key。无法解决个别key特别大的情况造成的倾斜，如果某些key的大小非常大，即使一个task单独执行它，也会受到数据倾斜的困扰。
    - 使用map join 代替reduce join
      * 在小表不是特别大(取决于你的executor大小)的情况下使用，可以使程序避免shuffle的过程，自然也就没有数据倾斜的困扰了.（详细见<http://blog.csdn.net/lsshlsw/article/details/50834858>、<http://blog.csdn.net/lsshlsw/article/details/48694893>）
      * 局限性: 因为是先将小数据发送到每个executor上，所以数据量不能太大.

11.（没问题）写个broadcast看看

import org.apache.spark.{SparkContext, SparkConf}

import scala.collection.mutable.ArrayBuffer

object joinTest extends App{

val conf = new SparkConf().setMaster("local[2]").setAppName("test")

val sc = new SparkContext(conf)

/\*\*

\* map-side-join

\* 取出小表中出现的用户与大表关联后取出所需要的信息

\* \*/

//部分人信息(身份证,姓名)

val people\_info = sc.parallelize(Array(("110","lsw"),("222","yyy"))).collectAsMap()

//全国的学生详细信息(身份证,学校名称,学号...)

val student\_all = sc.parallelize(Array(("110","s1","211"),

("111","s2","222"),

("112","s3","233"),

("113","s2","244")))

//将需要关联的小表进行关联

val people\_bc = sc.broadcast(people\_info)

/\*\*

\* 使用mapPartition而不是用map，减少创建broadCastMap.value的空间消耗

\* 同时匹配不到的数据也不需要返回（）

\* \*/

val res = student\_all.mapPartitions(iter =>{

val stuMap = people\_bc.value

val arrayBuffer = ArrayBuffer[(String,String,String)]()

iter.foreach{case (idCard,school,sno) =>{

if(stuMap.contains(idCard)){

arrayBuffer.+= ((idCard, stuMap.getOrElse(idCard,""),school))

}

}}

arrayBuffer.iterator

})

/\*\*

\* 使用另一种方式实现

\* 使用for的守卫

\* \*/

val res1 = student\_all.mapPartitions(iter => {

val stuMap = people\_bc.value

for{

(idCard, school, sno) <- iter

if(stuMap.contains(idCard))

} yield (idCard, stuMap.getOrElse(idCard,""),school)

})

res.foreach(println)

}

# Spark section-0 基础(3)

****1. spark的有几种部署模式，每种模式特点？****

* 本地模式
  + Spark不一定非要跑在hadoop集群，可以在本地，起多个线程的方式来指定。方便调试，本地模式分三类
    - local：只启动一个executor
    - local[k]: 启动k个executor
    - local：启动跟cpu数目相同的 executor
* standalone模式
  + 分布式部署集群，自带完整的服务，资源管理和任务监控是Spark自己监控，这个模式也是其他模式的基础
* Spark on yarn模式
  + 分布式部署集群，资源和任务监控交给yarn管理
  + 粗粒度资源分配方式，包含cluster和client运行模式
    - cluster 适合生产，driver运行在集群子节点，具有容错功能
    - client 适合调试，dirver运行在客户端
* Spark On Mesos模式

****2. Spark技术栈有哪些组件，每个组件都有什么功能，适合什么应用场景？****

* Spark core
  + 是其它组件的基础，spark的内核
  + 主要包含：有向循环图、RDD、Lingage、Cache、broadcast等
* SparkStreaming
  + 是一个对实时数据流进行高通量、容错处理的流式处理系统
  + 将流式计算分解成一系列短小的批处理作业
* Spark sql：
  + 能够统一处理关系表和RDD，使得开发人员可以轻松地使用SQL命令进行外部查询
* MLBase
  + 是Spark生态圈的一部分专注于机器学习，让机器学习的门槛更低
  + MLBase分为四部分：MLlib、MLI、ML Optimizer和MLRuntime。
* GraphX
  + 是Spark中用于图和图并行计算

****3. （没问题）spark有哪些组件****

* master：管理集群和节点，不参与计算。
* worker：计算节点，进程本身不参与计算，和master汇报。
* Driver：运行程序的main方法，创建spark context对象。
* spark context：控制整个application的生命周期，包括dagsheduler和task scheduler等组件。
* client：用户提交程序的入口。

# Spark section-1 Spark运行细节(13)

****1. （没问题）spark工作机制****

* 用户在client端提交作业后，会由Driver运行main方法并创建spark context上下文。
* 执行add算子，形成dag图输入dagscheduler
* 按照add之间的依赖关系划分stage输入task scheduler
* task scheduler会将stage划分为taskset分发到各个节点的executor中执行

****2. （没问题）Spark应用程序的执行过程****

* 构建Spark Application的运行环境（启动SparkContext）
* SparkContext向资源管理器（可以是Standalone、Mesos或YARN）注册并申请运行Executor资源；
* 资源管理器分配Executor资源，Executor运行情况将随着心跳发送到资源管理器上；
* SparkContext构建成DAG图，将DAG图分解成Stage，并把Taskset发送给Task Scheduler
* Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行，SparkContext将应用程序代码发放给Executor。
* Task在Executor上运行，运行完毕释放所有资源。

****3. （没问题）driver的功能是什么？****

* 一个Spark作业运行时包括一个Driver进程，也是作业的主进程，具有main函数，并且有SparkContext的实例，是程序的人口点；
* 功能：
  + 向集群申请资源
  + 负责了作业的调度和解析
  + 生成Stage并调度Task到Executor上（包括DAGScheduler，TaskScheduler）

****4. （没问题）Spark中Work的主要工作是什么？****

* 管理当前节点内存，CPU的使用状况，接收master分配过来的资源指令，通过ExecutorRunner启动程序分配任务
* worker就类似于包工头，管理分配新进程，做计算的服务，相当于process服务
* worker不会运行代码，具体运行的是Executor是可以运行具体appliaction写的业务逻辑代码

****5. （没问题）task有几种类型？2种****

* resultTask类型，最后一个task
* shuffleMapTask类型，除了最后一个task都是

****6. 什么是shuffle，以及为什么需要shuffle？****

* shuffle中文翻译为洗牌，需要shuffle的原因是：某种具有共同特征的数据汇聚到一个计算节点上进行计算

****7. Spark master HA 主从切换过程不会影响集群已有的作业运行，为什么？****

* 因为程序在运行之前，已经申请过资源了，driver和Executors通讯，不需要和master进行通讯的。

****8. Spark并行度怎么设置比较合适****

* spark并行度，每个core承载2~4个partition（并行度）
* 并行度和数据规模无关，只和内存和cpu有关

****9. Spaek程序执行，有时候默认为什么会产生很多task，怎么修改默认task执行个数？****

* 有很多小文件的时候，有多少个输入block就会有多少个task启动
* spark中有partition的概念，每个partition都会对应一个task，task越多，在处理大规模数据的时候，就会越有效率

****10. Spark中数据的位置是被谁管理的？****

* 每个数据分片都对应具体物理位置，数据的位置是被blockManager管理

****11. 为什么要进行序列化****

* 减少存储空间，高效存储和传输数据，缺点：使用时需要反序列化，非常消耗CPU

****12. Spark如何处理不能被序列化的对象？****

* 封装成object

****13. （没问题）Spark提交你的jar包时所用的命令是什么？****

* spark-submit

# Spark section-2 Spark 与 Hadoop/MapReduce 比较(7)

****1. Mapreduce和Spark的相同和区别****

* 两者都是用mr模型来进行并行计算
* hadoop的一个作业：job
  + job分为map task和reduce task，每个task都是在自己的进程中运行的
  + 当task结束时，进程也会结束
* spark用户提交的任务：application
  + 一个application对应一个sparkcontext，app中存在多个job
  + 每触发一次action操作就会产生一个job
  + 这些job可以并行或串行执行
  + 每个job中有多个stage，stage是shuffle过程中DAGSchaduler通过RDD之间的依赖关系划分job而来的
  + 每个stage里面有多个task，组成taskset有TaskSchaduler分发到各个executor中执行
  + executor的生命周期是和app一样的，即使没有job运行也是存在的，所以task可以快速启动读取内存进行计算。
* hadoop的job只有map和reduce操作，表达能力比较欠缺
  + 在mr过程中会重复的读写hdfs，造成大量的io操作，多个job需要自己管理关系。
* spark的迭代计算都是在内存中进行的
  + API中提供了大量的RDD操作如join，groupby等
  + 通过DAG图可以实现良好的容错

****2. 简答说一下hadoop的mapreduce编程模型****

* 首先map task会从本地文件系统读取数据，转换成key-value形式的键值对集合，使用的是hadoop内置的数据类型（longwritable、text）
* 将键值对集合输入mapper进行业务处理过程，将其转换成需要的key-value在输出
* 之后会进行一个partition分区操作，默认使用的是hashpartitioner，自定义分区：重写getpartition方法
* 之后会对key进行进行sort排序，grouping分组操作将相同key的value合并分组输出
* 之后进行一个combiner归约操作，其实就是一个本地段的reduce预处理，以减小后面shufle和reducer的工作量
* reduce task会通过网络将各个数据收集进行reduce处理
* 最后将数据保存或者显示，结束整个job

****3. 简单说一下hadoop和spark的shuffle相同和差异？****

* high-level 角度：
  + 两者并没有大的差别 都是将 mapper（Spark: ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）  
    Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce()。
* low-level 角度：
  + Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。
  + 好处：combine/reduce() 可以处理大规模的数据
    - 因为其输入数据可以通过外排得到
    - mapper 对每段数据先做排序
    - reducer 的 shuffle 对排好序的每段数据做归并
  + Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不提前排序
  + 如果用户需要经过排序的数据：sortByKey()
* 实现角度：
  + Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spilt, merge, shuffle, sort, reduce()
  + Spark 没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，spill, merge, aggregate 等操作需要蕴含在 transformation() 中

****4. 简单说一下hadoop和spark的shuffle过程****

* hadoop：map端保存分片数据，通过网络收集到reduce端
* spark：spark的shuffle是在DAGSchedular划分Stage的时候产生的，TaskSchedule要分发Stage到各个worker的executor，减少shuffle可以提高性能

****5. partition和block的关联****

* hdfs中的block是分布式存储的最小单元，等分，可设置冗余，这样设计有一部分磁盘空间的浪费，但是整齐的block大小，便于快速找到、读取对应的内容
* Spark中的partition是RDD的最小单元，RDD是由分布在各个节点上的partition组成的。
* partition是指的spark在计算过程中，生成的数据在计算空间内最小单元
* 同一份数据（RDD）的partion大小不一，数量不定，是根据application里的算子和最初读入的数据分块数量决定
* block位于存储空间；partion位于计算空间，block的大小是固定的、partion大小是不固定的，是从2个不同的角度去看数据。

****6. （没问题）Spark为什么比mapreduce快？****

* 基于内存计算，减少低效的磁盘交互
* 高效的调度算法，基于DAG
* 容错机制Linage

****7. Mapreduce操作的mapper和reducer阶段相当于spark中的哪几个算子？****

* 相当于spark中的map算子和reduceByKey算子，区别：MR会自动进行排序的，spark要看具体partitioner

# Spark section-3 RDD(4)

****1. RDD机制****

* 分布式弹性数据集，简单的理解成一种数据结构，是spark框架上的通用货币
* 所有算子都是基于rdd来执行的
* rdd执行过程中会形成dag图，然后形成lineage保证容错性等
* 从物理的角度来看rdd存储的是block和node之间的映射

****2. RDD的弹性表现在哪几点？****

* 自动的进行内存和磁盘的存储切换；
* 基于Lingage的高效容错；
* task如果失败会自动进行特定次数的重试；
* stage如果失败会自动进行特定次数的重试，而且只会计算失败的分片；
* checkpoint和persist，数据计算之后持久化缓存
* 数据调度弹性，DAG TASK调度和资源无关
* 数据分片的高度弹性，分片很多碎片可以合成大的

****3. RDD有哪些缺陷？****

* 不支持细粒度的写和更新操作（如网络爬虫）
  + spark写数据是粗粒度的，所谓粗粒度，就是批量写入数据 （批量写）
  + 但是读数据是细粒度的也就是说可以一条条的读 （一条条读）
* 不支持增量迭代计算，Flink支持

****4. （没问题）什么是RDD宽依赖和窄依赖？****

* RDD和它依赖的parent RDD(s)的关系有两种不同的类型
  + 窄依赖：每一个parent RDD的Partition最多被子RDD的一个Partition使用 （一父一子）
  + 宽依赖：多个子RDD的Partition会依赖同一个parent RDD的Partition （一父多子）

# Spark section-4 RDD操作(13)

1. （没问题）cache和pesist的区别

* cache和persist都是用于缓存RDD，避免重复计算
* .cache() == .persist(MEMORY\_ONLY)

****2. （没问题）cache后面能不能接其他算子,它是不是action操作？****

* 可以接其他算子，但是接了算子之后，起不到缓存应有的效果，因为会重新触发cache
* cache不是action操作

3. （没问题）什么场景下要进行persist操作？  
以下场景会使用persist

* 某个步骤计算非常耗时或计算链条非常长，需要进行persist持久化
* shuffle之后为什么要persist，shuffle要进性网络传输，风险很大，数据丢失重来，恢复代价很大
* shuffle之前进行persist，框架默认将数据持久化到磁盘，这个是框架自动做的。

****4. （没问题）rdd有几种操作类型？三种！！****

* transformation，rdd由一种转为另一种rdd
* action
* cronroller，控制算子(cache/persist) 对性能和效率的有很好的支持

****5. （没问题）reduceByKey是不是action？****

* 不是，很多人都会以为是action，reduce rdd是action

****6. （没问题）collect功能是什么，其底层是怎么实现的？****

* driver通过collect把集群中各个节点的内容收集过来汇总成结果
* collect返回结果是Array类型的，合并后Array中只有一个元素，是tuple类型（KV类型的）的。

****7. （没问题）map与flatMap的区别****

* map：对RDD每个元素转换，文件中的每一行数据返回一个数组对象
* flatMap：对RDD每个元素转换，然后再扁平化，将所有的对象合并为一个对象，会抛弃值为null的值

8. （没问题）列举你常用的action？  
collect，reduce,take,count,saveAsTextFile等

****9. （没问题）union操作是产生宽依赖还是窄依赖？****

* 窄依赖

****10. Spark累加器有哪些特点？（需要搞清楚）****

* 全局的，只增不减，记录全局集群的唯一状态
* 在exe中修改它，在driver读取
* executor级别共享的，广播变量是task级别的共享
* 两个application不可以共享累加器，但是同一个app不同的job可以共享

****11. （没问题）spark hashParitioner的弊端****

* 分区原理：对于给定的key，计算其hashCode
* 弊端是数据不均匀，容易导致数据倾斜

****12. （没问题）RangePartitioner分区的原理****

* 尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，也就是说一个分区中的元素肯定都是比另一个分区内的元素小或者大
* 分区内的元素是不能保证顺序的
* 简单的说就是将一定范围内的数映射到某一个分区内

****13. （没问题？）Spark中的HashShufle的有哪些不足？这个问题不是在坞丝计划之后已经解决了吗？****

* shuffle产生海量的小文件在磁盘上，此时会产生大量耗时的、低效的IO操作；
* 容易导致内存不够用，由于内存需要保存海量的文件操作句柄和临时缓存信息
* 容易出现数据倾斜，导致OOM

# Spark section-5 大数据问题(7)

1. （没问题）如何使用Spark解决TopN问题？（互联网公司常面）  
<https://blog.csdn.net/oopsoom/article/details/25815443>

val data = sc.textFile("/dw/spark/mobile.txt")

data.map(line=>"""\d+\.\d+\.\d+\.\d+""".r.findAllIn(line).mkString)

.filter(\_!="")

.map(word=>(word,1))

.reduceByKey(\_+\_)

.map(word=>(word.\_2,word.\_1))

.sortByKey(false)

.map(word=>(word.\_2,word.\_1))

take 50

2. （没问题）如何使用Spark解决分组排序问题？（互联网公司常面）  
<https://blog.csdn.net/huitoukest/article/details/51273143>

注意：

groupBy和groupByKey是不同的，比如（A，1），（A，2）；使用groupBy之后结果是（A，（（A，1），（A，2）））；

使用groupByKey之后结果是：（A，（1,2））；关键区别就是合并之后是否会自动去掉key信息；

示例：

输入：

Spark 95

Hadoop 68

Flink 55

Spark 95

Hadoop 98

Flink 85

Kafka 67

Spark 85

Hadoop 98

Flink 82

Kafka 76

Spark 98

Kafka 70

Spark 87

Hadoop 91

代码：

object TopNSecond {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setAppName("TopNSecond by Scala").setMaster("local");

val sc=new SparkContext(conf);

val data=sc.textFile("D:/tmp/TopNSecond.txt",1);

val lines=data.map{ line => (line.split(" ")(0),line.split(" ")(1).toInt) };

val groups=lines.groupByKey();

val groupsSort=groups.map(tu=>{

val key=tu.\_1;

val values=tu.\_2;

val sortValues=values.toList.sortWith(\_>\_).take(4);

(key,sortValues);

});

groupsSort.sortBy(tu=>tu.\_1, false, 1).collect().foreach(value=>{

print(value.\_1);

value.\_2.foreach(v=>print("\t"+v));

println();

});

sc.stop();

}

}

默认是降序排列：

核心思想是相对分组内部排序，然后取前Ｎ个，然后对分组之间进行排序；

输出结果：

Spark 98 95 95 87

Kafka 76 70 67

Hadoop 98 98 91 68

Flink 85 82 55

后续： 要了解spark中每个transformation和action函数的意思

****3. （没问题）给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url?****

要记住一亿等于10的八次方 一亿=10的八次方

要记住一G等于10的九次方字节 1G=10的9次方Byte

* 方案1：可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。  
  遍历文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件(记为a0,a1,…,a999)中。这样每个小文件的大约为300M。  
  遍历文件b，采取和a相同的方式将url分别存储到1000小文件(记为b0,b1,…,b999)。这样处理后，所有可能相同的url都在对应的小文件(a0vsb0,a1vsb1,…,a999vsb999)中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。  
  求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。
* 方案2：如果允许有一定的错误率，可以使用Bloomfilter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloomfilter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloomfilter，如果是，那么该url应该是共同的url(注意会有一定的错误率)。

****4. （没问题 ）有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M，要求返回频数最高的100个词。****

* Step1：顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件(记为f0,f1,...,f4999)中，这样每个文件大概是200k左右，如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M;
* Step2：对每个小文件，统计每个文件中出现的词以及相应的频率(可以采用trie树/hash\_map等)，并取出出现频率最大的100个词(可以用含100个结点的最小堆)，并把100词及相应的频率存入文件，这样又得到了5000个文件;
* Step3：把这5000个文件进行归并(类似与归并排序) 或者 在内存中用第一个文件的元素创建最小堆，然后取出每个文件的顶元素与堆顶比较，如果比堆顶大，则把堆顶删除，替换成新插入的元素，然后调整堆，如果比堆顶小，则取下一个元素进行比较;

5. （没问题）现有海量日志数据保存在一个超级大的文件中，该文件无法直接读入内存，其中包含了所有访问百度的IP，要求从中提取某天出访问百度次数最多的那个IP。

分而治之+Hash  
1)**IP地址最多有2^32=4G种取值情况**，所以不能完全加载到内存中处理;  
2)可以考虑采用“分而治之”的思想，按照IP地址的Hash(IP)%1024值，把海量IP日志分别存储到1024个小文件中。这样，每个小文件最多包含4MB个IP地址;  
3)对于每一个小文件，可以构建一个IP为key，出现次数为value的Hashmap，同时记录当前出现次数最多的那个IP地址;  
4)可以得到1024个小文件中的出现次数最多的IP，再依据常规的排序算法得到总体上出现次数最多的IP;

****6. 在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。****

* 方案1：采用2-Bitmap(每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义)进行，共需内存2^32\*2bit=1GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。
* 方案2：也可采用与第1题类似的方法，进行划分小文件的方法。然后在小文件中通过hashmap找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

****7. （没问题）腾讯面试题：给40亿个不重复的unsignedint的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中?****

*一定要记住2的32次方等于40亿多*

*一定要记住512M=2的32次方个bit，即40亿个bit*

*一定要记住1G=10亿Byte=80亿个bit.*

* 申请512M的内存，一个bit位代表一个unsignedint值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

8.（没问题）字节面试题：一亿个数，乱序，放不进内存，如何快速找到中间的5个数？

 关于中位数：数据排序后，位置在最中间的数值。即将数据分成两部分，一部分大于该数值，一部分小于该数值。中位数的位置：当样本数为奇数时，中位数=(N+1)/2 ; 当样本数为偶数时，中位数为N/2与1+N/2的均值（那么10G个数的中位数，就第5G大的数与第5G+1大的数的均值了）。

分析：明显是一道工程性很强的题目，和一般的查找中位数的题目有几点不同。  
       1. 原数据不能读进内存，不然可以用快速选择，如果数的范围合适的话还可以考虑桶排序或者计数排序，但这里假设是32位整数，仍有4G种取值，需要一个16G大小的数组来计数。  
       2. 若看成从N个数中找出第K大的数，如果K个数可以读进内存，可以利用最小或最大堆，但这里K=N/2,有5G个数，仍然不能读进内存。  
       3. 接上，对于N个数和K个数都不能一次读进内存的情况，《编程之美》里给出一个方案：设k<K,且k个数可以完全读进内存，那么先构建k个数的堆，先找出第0到k大的数，再扫描一遍数组找出第k+1到2k的数，再扫描直到找出第K个数。虽然每次时间大约是nlog(k)，但需要扫描ceil(K/k) 次，这里要扫描5次。

解法：首先假设是32位无符号整数。  
       1. 读一遍10G个整数，把整数映射到256M个区段中，用一个64位无符号整数给每个相应区段记数。  
说明：整数范围是0 - 2^32 - 1，一共有4G种取值，映射到256M个区段，则每个区段有16（4G/256M = 16）种值，每16个值算一段， 0～15是第1段，16～31是第2段，……2^32-16 ～2^32-1是第256M段。一个64位无符号整数最大值是0～8G-1，这里先不考虑溢出的情况。总共占用内存256M×8B=2GB。  
       2. 从前到后对每一段的计数累加，当累加的和超过5G时停止，找出这个区段（即累加停止时达到的区段，也是中位数所在的区段）的数值范围，设为[a，a+15]，同时记录累加到前一个区段的总数，设为m。然后，释放除这个区段占用的内存。  
       3. 再读一遍10G个整数，把在[a，a+15]内的每个值计数，即有16个计数。  
       4. 对新的计数依次累加，每次的和设为n，当m+n的值超过5G时停止，此时的这个计数所对应的数就是中位数。  
  
总结：  
1.以上方法只要读两遍整数，对每个整数也只是常数时间的操作，总体来说是线性时间。  
2. 考虑其他情况。  
若是有符号的整数，只需改变映射即可。若是64为整数，则增加每个区段的范围，那么在第二次读数时，要考虑更多的计数。若过某个计数溢出，那么可认定所在的区段或代表整数为所求，这里只需做好相应的处理。噢，忘了还要找第5G+1大的数了，相信有了以上的成果，找到这个数也不难了吧。  
3. 时空权衡。  
花费256个区段也许只是恰好配合2GB的内存（其实也不是，呵呵）。可以增大区段范围，减少区段数目，节省一些内存，虽然增加第二部分的对单个数值的计数，但第一部分对每个区段的计数加快了（总体改变？？待测）。  
4. 映射时尽量用位操作，由于每个区段的起点都是2的整数幂，映射起来也很方便。

# Spark section-6 机器学习算法(4)

\*\*1. mllib支持的算法？ \*\*

* 分类、聚类、回归、协同过滤

**2. kmeans算法原理**

* 随机初始化中心点范围，计算各个类别的平均值得到新的中心点。
* 重新计算各个点到中心值的距离划分，再次计算平均值得到新的中心点，直至各个类别数据平均值无变化。

**3. 朴素贝叶斯分类算法原理**  
对于待分类的数据和分类项，根据待分类数据的各个特征属性，出现在各个分类项中的概率判断该数据是属于哪个类别的。

**4. 关联规则挖掘算法apriori原理**

* 一个频繁项集的子集也是频繁项集，针对数据得出每个产品的支持数列表，过滤支持数小于预设值的项，对剩下的项进行全排列，重新计算支持数，再次过滤，重复至全排列结束，可得到频繁项和对应的支持数。

# Spark section-7 Hive(2)

1. Hive中存放是什么？

* 表（数据+元数据） 存的是和hdfs的映射关系，hive是逻辑上的数据仓库，实际操作的都是hdfs上的文件，HQL就是用sql语法来写的mr程序。

2. Hive与关系型数据库的关系？

* 没有关系，hive是数据仓库，不能和数据库一样进行实时的CURD操作。
* 是一次写入多次读取的操作，可以看成是ETL工具。

Spark服务端口

* 8080 spark集群web ui端口
* 4040 sparkjob监控端口
* 18080 jobhistory端口

Spark Job 默认的调度模式 - FIFO  
RDD 特点 - 可分区/可序列化/可持久化  
Broadcast - 任何函数调用/是只读的/存储在各个节点  
Accumulator - 支持加法/支持数值类型/可并行  
Task 数量由 **Partition** 决定  
Task 运行在 **Workder node** 中 Executor 上的工作单元  
master 和 worker 通过 **Akka** 方式进行通信的  
默认的存储级别 - MEMORY\_ONLY  
hive 的元数据存储在 derby 和 MySQL 中有什么区别 - 多会话  
DataFrame 和 RDD 最大的区别 - 多了 schema

Spark钨丝计划？（需要了解）

Hadoop

Hadoop联邦机制？（需要搞清楚）

Hadoop断电处理？（需要搞清楚）

Kerberos

数据库

1.超键，候选键，主键，外键？

**超键：**在关系中能唯一标识元组中的属性集称为关系模式的超键。一个属性可以作为一个超键，多个属性组也可以作为一个超键，超键包含候选键和主键

**候选键：**是最小超键，即没有冗余元素的超键

**主键：**数据库中对存储数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，既不能为空值NULL。关系型数据库中的一条记录中有若干个属性，若其中某一个属性组(注意是组)能唯一标识一条记录，该属性组就可以成为一个主键

**外键：**在一个表中更存在另外一个表的主键成为此表的外键。

2.什么是事务？事务的四大特性及含义？

**事务**：事务是由一系列数据库操作组成的一个完整的逻辑过程。就是有多个sql语句组成的一个逻辑单元。不可分割。为了保证在写入/更新数据的过程中，保证事务执行过程中的安全可靠，作为一个事务需要经过ACID的四大特性的测试。

**事务的四大特性ACID**：原子性，一致性，隔离性，持久性。

* **原子性：**一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
* **一致性：**在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
* **隔离性：**数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。由于在并发事务中，会导致事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
* **持久性：**事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

由于并发事务的三大问题：脏读，不可重复读，幻读

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，不可重复读的侧重点在更新修改数据。

3、幻读：和不可重复读一个意思，只不过幻读的侧重点在新增和删除，而不可重复读的侧重点是更改数据，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，对数据作了新增或删除行，导致事务A多次读取同一数据时，返回的数据是不是多了行，就是少了行，幻读的侧重点在新增和删除。

因此产生了四种隔离级别：未提交读，提交读，可重读，串行化



**3.什么是锁？**

**在所有的DBMS中，锁是实现事务的关键，锁可以保证事务的完整性和并发性（隔离性），**

**乐观锁和悲观锁**是并发控制主要采用的技术手段。

* 悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。
* 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据的完整性

**3.什么是视图？**

视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增删改查操作，视图通常是由一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得相比多表查询我们获取数据更容易。

如下两种场景一般会使用到视图：

（1）不希望访问者获取整个表的信息，只暴露部分字段给访问者，所以就建一个虚表，就是视图。

（2）查询的数据来源于不同的表，而查询者希望以统一的方式查询，这样也可以建立一个视图，把多个表查询结果联合起来，查询者只需要直接从视图中获取数据，不必考虑数据来源于不同表所带来的差异。

注：这个视图是在数据库中创建的 而不是用代码创建的。

**4.触发器的作用？**

触发器是一中特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

**5.维护数据库的完整性和一致性，你喜欢用触发器还是自写业务逻辑？为什么？**

1. 尽可能使用约束，如 check, 主键，外键，非空字段等来约束，这样做效率最高，也最方便。
2. 其次是使用触发器，这种方法可以保证，无论什么业务系统访问数据库都可以保证数据的完整新和一致性。
3. 最后考虑的是自写业务逻辑，但这样做麻烦，编程复杂，效率低下。

**6.索引的作用？和它的优缺点是什么？**

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

创建索引可以大大提高系统的性能（优点）：

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面：

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。

一般来说，应该在这些列上创建索引：

（1）在经常需要搜索的列上，可以加快搜索的速度；

（2）在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

（3）在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；

（4）在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

（5）在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

（6）在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

**7.drop，delete，truncate的区别**

drop直接删掉表 。

truncate删除表中数据，再插入时自增长id又从1开始 。

delete删除表中数据，可以加where字句。

（1） DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对TABLE；DELETE可以是table和view

（5） TRUNCATE 和DELETE只删除数据，而DROP则删除整个表（结构和数据）。

（6） truncate与不带where的delete ：只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete语句为DML（data maintain Language),这个操作会被放到 rollback segment中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop是DLL（data define language),操作立即生效，原数据不放到 rollback segment中，不能回滚。

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用delete且注意结合where来约束影响范围。回滚段要足够大。要删除表用drop;若想保留表而将表中数据删除，如果于事务无关，用truncate即可实现。如果和事务有关，或老师想触发trigger,还是用delete。

（10） Truncate table 表名 速度快,而且效率高,因为:

truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

**9、SQL常用命令：**

CREATE TABLE Student(

ID NUMBER PRIMARY KEY,

NAME VARCHAR2(50) NOT NULL);//建表

CREATE VIEW view\_name AS

Select \* FROM Table\_name;//建视图

Create UNIQUE INDEX index\_name ON TableName(col\_name);//建索引

INSERT INTO tablename {column1,column2,…} values(exp1,exp2,…);//插入

INSERT INTO Viewname {column1,column2,…} values(exp1,exp2,…);//插入视图实际影响表

UPDATE tablename SET name='zang 3' condition;//更新数据

DELETE FROM Tablename WHERE condition;//删除

GRANT (Select,delete,…) ON (对象) TO USER\_NAME [WITH GRANT OPTION];//授权

REVOKE (权限表) ON(对象) FROM USER\_NAME [WITH REVOKE OPTION] //撤权

列出工作人员及其领导的名字：

Select [E.NAME](http://e.name/), [S.NAME](http://s.name/) FROM EMPLOYEE E S

WHERE E.SUPERName=S.Name

**6.讲一讲三范式？<https://www.zhihu.com/question/24696366>**

**1NF的定义为：符合1NF的关系中的每个属性都不可再分。**

**1NF是所有关系型数据库的最基本要求**

**2NF:2NF在1NF的基础之上，消除了非主属性对于码的部分函数依赖。**

**第三范式（3NF） 3NF在2NF的基础之上，消除了非主属性对于码的传递函数依赖。**

**7.存储过程是什么？与函数的区别**

存储过程是为了以后的使用而保存的一条或者多条sql语句集，主要作用：封装复杂的sql语句，不需要反复简历一些列处理步骤，保证了数据的完整性，简化对变动的管理，简而言之就是简单，安全，高性能。但是写存储过程比写

**8.内连接和外连接的区别？还有左连接，右连接？**

项目

大公司面试题

考拉面经1：

考拉海购Java岗  两轮技术面+HR面试

一面：

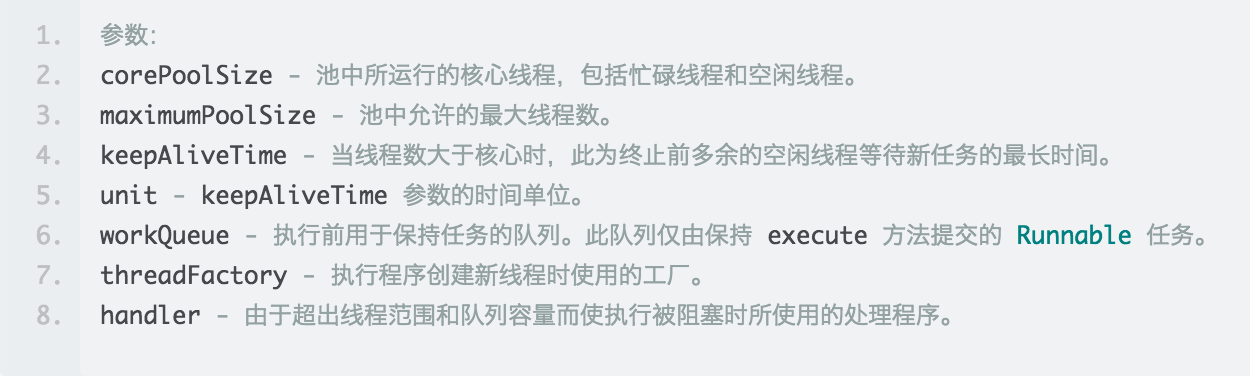
1、自我介绍

2、实习期间项目相关，然后转到Spring

3、Spring 在项目中用了哪些特性，IOC，及Kafka监听接口相关实现。

4、（没问题）ThreadPool，Jdk原生线程池，四个参数详细解释原理，当线程池中poolSize达到corePoolSize且阻塞队列已满，再来一个任务，如何处理

**答：**



### 各参数之间的关系-corePoolSize，maximumPoolSize，workQueue

**当新任务在方法 execute(java.lang.Runnable) 中提交时，**

1. 当运行的线程数少于corePoolSize核心线程数，则创建线程来处理任务，即使核心线程中有空闲的线程，也照样创建新的线程来处理任务。
2. 当运行的线程数大于corePoolSize核心线程数而少于maximumPoolSize最大线程数，且任务队列未满时，将任务放入任务队列。
3. 当运行的线程大于corePoolSize核心线程数，且任务队列已满

1.若运行的线程数小于最大线程数，创建线程。

2.若运行的线程数等于最大线程数，则抛出异常。

**拒绝任务-(RejectedExecutionHandler)handler**

1.当ThreadPoolExecutor对象调用了shutdown()或者shutdownNow()，即线程池关闭的时，

2.当线程池的任务队列已满，并且线程池中的线程数目达到maxmumpoolSize时

以下两种情况，将会导致线程池拒绝执行任务，在ThreadPoolExecutor类中的execute方法调用RejectedExecutionHandler接口中的rejectedExecution方法。

**下面提供了四种预定义的处理策略**

1.默认的使用ThreadPoolExecutor.AbortPolicy，这种处理是一旦拒绝接接受，丢弃任务，将会抛出一个RejectedExecutionException的异常。

2.ThreadPoolExecutor.DiscardPolicy，也是丢弃任务，但是不抛出异常。

3.ThreadPoolExecutor.CallerRunsPolicy，由调用executor方法自己的线程运行任务。这提供了一个简单的反馈控制机制，可以减慢新任务提交的速度。

4.ThreadPoolExecutor.DiscardOldestPolicy：如果executor还没有shutdown，则处在任务队列头的任务将丢弃，则重新尝试执行任务（如果再次失败，继续重复此过程）

5、多线程实现同步的方式，互斥同步，非阻塞同步，

6、reenterlocker和sychronize的区别？解释Synchronize关键字的锁优化技术，偏向锁，轻量级锁，重量级锁，这些锁是如何存储的，偏向锁撤销升级为轻量级锁的过程，结合源码解释

**答：**

1. （没问题）说一下 synchronized 底层实现原理？

<http://bigdatadecode.club/JavaSynchronizedTheory.html>

synchronized 语句块是由一对 monitorenter/monitorexit 指令来显示获取monitor对象和释放monitor对象来实现的，

sychronnized方法是通过判断是否设置常量池中ACC\_SYNCHRONIZED标示符来隐式地获得monitor对象和是否monitor对象来实现的，

不管是显示还是隐式的获取monitor，本质没有区别。

说到底synchronized是通过一个获取和释放monitor对象来实现的。每个对象都有一个监视器锁(monitor)与之对应。当monitor被占用时就会处于锁定状态，例如monitorenter/monitorexit：

* 线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：
  1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
  2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
  3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。
* 执行monitorexit的线程必须是objectref所对应的monitor的所有者。
  1. 指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个monitor的所有权。

通过这两个指令我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

什么是Monitor对象？它是一种同步机制，monitor是由ObjectMonitor实现的，其主要数据结构有四个属性，如下：

* \_count用来记录该线程获取锁的次数
* \_WaitSet存放处于wait状态的线程队列
* \_EntryList存放处于等待获取锁block状态的线程队列，即被阻塞的线程
* \_owner指向持有ObjectMonitor对象的线程

当多个线程同时访问一段同步代码时，首先会进入\_EntryList队列中，当某个线程获取到对象的monitor后进入\_Owner区域并把monitor中的\_owner变量设置为当前线程，同时monitor中的计数器\_count加1，若线程调用wait()方法，将释放当前持有的monitor，\_owner变量恢复为null，\_count自减1，同时该线程进入\_WaitSet集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。如下图所示



在 Java 6 之前，synchronized是重量级锁，因为monitor对象的实现依赖于底层的操作系统的互斥锁来实现的，而操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了偏向锁、轻量级锁和自旋锁等概念，大大改进了性能。

2.（没问题）多线程中 synchronized 锁升级的原理是什么？

**为什么优化？**

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒会耗性能。

**为什么要引入偏向锁？**

因为经过HotSpot的作者大量的研究发现，大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

**为什么要引入轻量级锁？**

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。*因为阻塞线程需要CPU从用户态转到内核态，代价较大*，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

**偏向锁，轻量级锁，重量级锁，这些锁是如何存储的？**

这些锁的存储，就要说到Java对象头

下面重点说下java对象头。

众所周知Java中万物皆对象，那对象在内存中是怎么存储的呢？

每个对象分为三块区域:对象头、实例数据和对齐填充。

* 对象头包含两部分，第一部分是Mark Word，用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等，这一部分占一个字节。第二部分是Klass Pointer（类型指针），是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，这部分也占一个字节。(如果对象是数组类型的，则需要3个字节来存储对象头，因为还需要一个字节存储数组的长度)
* 实例数据存放的是类属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。
* 填充数据是因为虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐。

对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码(HashCode)，4Bits用于存储对象分代年龄，2Bits用于存储锁标志 位，1Bit固定为0，在其他状态(轻量级锁定、重量级锁定、GC标记、可偏向)下对象的存储内容如下表所示对象头存储结构。  


Synchronized通常被称为重量级锁，但是1.6之后对其进行优化，新增了轻量级锁和偏向锁，这里重点说下重量级锁，随后对Synchronized的优化简单介绍下。

从对象头的存储内容可以看出锁的状态都保存在对象头中，Synchronized也不例外，当其从轻量级锁膨胀为重量级锁时，锁标识位为10，其中指针指向的是monitor对象(也称为管程或监视器锁)的起始地址。

**synchronized 锁升级原理：**在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，或者第二个线程还在自旋中，这时又有一个线程3过来竞争这个锁对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

这几种锁的优缺点（偏向锁、轻量级锁、重量级锁）



4.（没问题）reenterlocker和sychronize的区别？

syncthronized是基于这个关键字加锁的,ReentrantLock是基于其api方法加锁的,syncthronized是自动释放锁的,Reentrant需要手动释放,在业务编写中ReentrantLock一般在try{}finally{}中搭配使用的,避免锁的释放。

syncthronized属于悲观锁方式,是阻塞式的,采用独占的方式,每次获取和释放锁频繁,上下文资源切换较大,存在短暂调度延迟。

ReentrantLock属于乐观锁思想,是非阻塞式的,是轻量级的,其内部的CAS就是基于乐观锁的思想实现的,CAS是建立在硬件指令集中的,是通过指令切换来达到原子性,比锁的切换是要快很多的,CAS中有三个参数,内存变量地址A,预期值B,要更新的值C,当多个线程想要更新CAS的内存变量值时会进行判断,若当前CAS内存地址指向B,则将C更新为A,若当前A指向不是预期的B,则线程更新失败,失败的线程不会被挂起,会重新准备再次尝试更新。

|  |  |  |
| --- | --- | --- |
|  | Reenterlocker | Synchronized |
| 定义 | 是基于其api方法加锁的 | 基于这个关键字加锁的 |
| 可重入性 | 可重入的，可用在递归执行的方法上，重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样 | 可重入的，可用在递归执行的方法上，不用担心释放锁 |
| 便利性 | 自动释放锁 | Reentrant需要手动释放,在业务编写中ReentrantLock一般在try{}finally{}中搭配使用的,避免锁的释放。 |
| 性能区别 | ReentrantLock属于乐观锁思想,是非阻塞式的,是轻量级的,其内部的CAS就是基于乐观锁的思想实现的,CAS是建立在硬件指令集中的,是通过指令切换来达到原子性,比锁的切换是要快很多的,CAS中有三个参数,内存变量地址A,预期值B,要更新的值C,当多个线程想要更新CAS的内存变量值时会进行判断,若当前CAS内存地址指向B,则将C更新为A,若当前A指向不是预期的B,则线程更新失败,失败的线程不会被挂起,会重新准备再次尝试更新。 | syncthronized属于悲观锁方式,是阻塞式的,采用独占的方式,每次获取和释放锁频繁,上下文资源切换较大,存在短暂调度延迟。  在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）,重量级锁后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。 |
| 高级功能，reenterlocker的优势 | 主要有以下3项：          1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。          2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，可以防止饥饿，非公平锁则当锁释放的时候，其他线程随机抢占锁。Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。          3.一个ReentrantLock对象可以同时绑定多个condition。Condition充当ReentrantLock的辅助监控责任,将Object的监控(wait,notify)分解成继而不同的对象,从而更方便的对lock的实现  其 await和sginal()和Object的wait和notify相识,不同的是 一个lock可实现多个Condition,  await(); 造成当前线程在被中断和释放前一直处于等待状态。  signal();随机唤醒一个等待的线程。  signAll();唤醒所有等待的线程。 |  |
|  |  |  |

*相似点：*

1.这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的（操作系统需要在用户态与内核态之间来回切换，代价很高，不过可以通过对锁优化进行改善）。

2.ReentrantLock和synchronized都是可重入的。synchronized因为可重入因此可以放在被递归执行的方法上,且不用担心线程最后能否正确释放锁；而ReentrantLock在重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样，否则可能导致其他线程无法获得该锁

*功能区别：*

这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成

*便利性：*很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

*性能的区别*：

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）,重量级锁后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

*1.Synchronized实现原理*

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器就减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

2.ReentrantLock

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：

        1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。

        2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。公平锁和非公平锁的区别？

公平锁是指当锁可用时,在锁上等待时间最长的线程将获得锁的使用权。而非公平锁则随机分配这种使用权。和synchronized一样，默认的ReentrantLock实现是非公平锁,因为相比公平锁，非公平锁性能更好。当然公平锁能防止饥饿,某些情况下也很有用。在创建ReentrantLock的时候通过传进参数true创建公平锁,如果传入的是false或没传参数则创建的是非公平锁

        3.锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象。ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

*ReenTrantLock实现的原理：（？？？）*

之后还会总结一篇ReenTrantLock相关的原理底层原理分析，简单来说，ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

每一个ReentrantLock实例都有且只有一个AQS实例，一个AQS实例维护一个Sync Queue和多个Condition Queue（看你新建了多少condition）。所以说，当我们的业务代码中的多个线程对同一个ReentrantLock实例进行锁竞争操作时，其实际就是对同一个Sync Queue的队列进行入队、出队操作。Condition 仿照实现Object类的wait signal signallAll等函数功能的，

每个AQS里有x（视你newCondition几次）个Condition Queue，它的结点类也是AQS内部类Node。Node里有一个nextWaiter，指向下一个在同一Condition Queue里的Node。

 首先明确下是，condition.wait一定是在成功lock的线程里调用才有效，不然不符合逻辑，同时也会抛出IlleagleMornitorException。

 获取锁的线程处于Sync Queue的队首，当调用condition.wait时，该线程会释放锁（即将AQS的state置为0），同时唤醒后继结点，后继结点在acquire的循环里会成功获取锁，然后将自己所在结点置为队首，然后开始自己线程自己的业务代码。

* 当waiter\_1收到相应condition的signal后，在Condition Queue中的Node会从Condition Queue中出队，进入Sync Queue队列，开始它的锁竞争的过程。
* 这里可以看出来，即使是被signal了，被signal的线程也不是直接就开始跑，而是再次进入Sync Queue开始竞争锁而已。这里的这个逻辑，跟Object.wait Object.signal也是完全一样的。

*什么情况下使用ReenTrantLock：*

答案是，如果你需要实现ReenTrantLock的三个独有功能时。

5.为什么[ConcurrentHashMap 1.8为什么要使用CAS+Synchronized取代Segment+ReentrantLock](https://www.cnblogs.com/yangfeiORfeiyang/p/9694383.html)

<https://juejin.im/post/5bc87409f265da0ad701da35>

7、（没问题）volatile关键字语义，内存屏障如何实现，JMM对内存屏障做了哪些优化，volatile的语义增强

内存屏障，又称内存栅栏，是一个CPU指令，它的作用有两个，*一是保证特定操作的执行顺序，二是保证某些变量的内存可见性*（利用该特性实现volatile的内存可见性）。由于编译器和处理器都能执行指令重排优化。如果在指令间插入一条Memory Barrier则会告诉编译器和CPU，不管什么指令都不能和这条Memory Barrier指令重排序，也就是说通过插入内存屏障禁止在内存屏障前后的指令执行重排序优化。Memory Barrier的另外一个作用是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读取到这些数据的最新版本。总之，volatile变量正是通过内存屏障实现其在内存中的语义，即可见性和禁止重排优化。

**内存屏障是什么**

* 硬件层的内存屏障分为两种：Load Barrier 和 Store Barrier即读屏障和写屏障。
* 内存屏障有两个作用：

1. 阻止屏障两侧的指令重排序；
2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。

* 对于Load Barrier（读屏障）来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据；
* 对于Store Barrier（写屏障）来说，在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

 java的内存屏障通常所谓的四种即LoadLoad,StoreStore,LoadStore,StoreLoad实际上也是上述两种的组合，完成一系列的屏障和数据同步功能。

 LoadLoad屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。 **在后续读操作执行之前，****之前的读操作要读取的数据已被读取。**

 StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。**在后续写操作执行之前，****之前****的写操作已对其他处理器可见。**

 LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。 **在后续写操作执行之前，之前的读操作要读的数据已被读取**

 StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。**在后续读操作执行之前，之前的写操作已对其他处理器可见。*它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能***

***Volitle内存屏障规则：由于内存屏障的作用，避免了volatile变量和其它指令重排序、线程之间实现了通信，使得volatile表现出了锁的特性。***

内存屏障规则： SS,写操作,SL, 读操作,LL, 读操作，LS

（1）在每一个volatile写操作前面插入一个StoreStore屏障。这确保了在进行volatile写之前前面的所有普通的写操作都已经刷新到了内存。

（2）在每一个volatile写操作后面插入一个StoreLoad屏障。这样可以避免volatile写操作与后面可能存在的volatile读写操作发生重排序。

（3）在每一个volatile读操作后面插入一个LoadLoad屏障。这样可以避免volatile读操作和后面普通的读操作进行重排序。

（4）在每一个volatile读操作后面插入一个LoadStore屏障。这样可以避免volatile读操作和后面普通的写操作进行重排序。

*volatile的内存屏障插入策略非常保守，其实在实际中，只要不改变volatile写-读得内存语义，编译器可以根据具体情况优化，省略不必要的屏障。*

比如在volatile读操作后面，还是volatile读的话，那么前一个volatile读操作后面的LL，由于禁止下面所有的普通写与上面的volatile读重排序，但是由于存在第二个volatile读，那个普通的读根本无法越过第二个volatile读。所以可以省略。

再比如在volatile写操作后面，还是volatile写的话，那么前一个volatile写操作后面的SL，可以省略，因为后面跟的是volatile写，没有读操作。

再比如：在volatile读操作后面，是普通写的话，volatile读操作后面的LL就可以省了。因为下面不存在普通读了。

8、指令重排序相关及其happen-before时，数据依赖等

9、HashMap底层数据结构，如何处理hash冲突，为何HashMap的大小要设置为2的n次幂，为什么IndexFor方法里，需要hash&length-1，为什么HashMap允许null值，resize()过程，多线程下resize为什么会出现死循环，详细解释，结合源码

10、HashMap jdk1.8之后，为何转链式结构为红黑树，为何长度为8的时候，才转成红黑树

11、NIO了解吗？   这个。。没答好，属于知识空白

12、阐述一下快排基本原理。easy

13、手写单例模式。分别问了多线程情况如何处理，什么时候可以破坏单例模式，怎么防止单例模式被破坏

14、JMM内存模型，如何保证缓存一致性。（重要）（需要搞清楚）

缓存一致性问题是由于在多核系统中，由于cpu缓存更新没有及时刷新到主内存时，会导致不同处理器缓存的数据不一致的问题。

假如说现在有core1和core2两个核心，在主存中存在int a = 0，core1进行写操作a = 1，但是cpu为了提升性能，不会将a=1写到主存，而是直接写到自己的cache（cpu操作主存很慢），core2这时候读取a = 0（主存中a=0），2个core互相观察对方，都是乱序执行的，在core1看来，指令顺序是store load（store是core1自己的写操作 a = 1，load是core2读取主存的操作a = 0），但是在core2看来，指令顺序是load（load是core2自己的读操作，那这里为什么没有core1的写操作？因为core2看不到，core之间只能看到对方对主存的操作），这里的乱序明显对core2造成了影响（没有读到最新的数据）。其实在并发编程里的各种问题，本质上都是指令乱序造成的。缓存一致性就是用来解决这种问题的（其实lock指令也可以解决这种问题，但是lock指令代价太大，其原理是锁住总线，其他cpu全部停止执行，只有当前cpu可以访问内存）

[image:E885013B-23C1-4A36-97AC-4003438DBD3A-17488-0001E67BFCBFE9F9/20190412002906429.png]

解决方案有：

1.通过在总线加LOCK#锁的方式

Lock指令代价太大，其原理是锁住总线，其他cpu全部停止执行，只有当前cpu可以访问内存，那么只有等待当前CPU完全执行完毕之后，其他CPU才能从其内存读取变量，然后进行相应的操作。由于在锁住总线期间，其他CPU无法访问内存，会导致效率低下。

2.通过缓存一致性协议

最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从主内存重新读取。

MESI协议是以缓存行(缓存的基本数据单位，在Intel的CPU上一般是64字节)的几个状态来命名的(全名是Modified、Exclusive、Share or Invalid)。该协议要求在每个缓存行上维护两个状态位，使得每个数据单位可能处于M、E、S和I这四种状态之一，各种状态含义如下：

M：被修改的。处于这一状态的数据，只在本处理器中有缓存数据，而其他处理器中没有。同时其状态相对于内存中的值来说，是已经被修改的，且没有更新到内存中。

E：独占的。处于这一状态的数据，只有在本处理器中有缓存，且其数据没有修改，即与内存中一致。

S：共享的。处于这一状态的数据在多个处理器中都有缓存，且与内存一致。

I：无效的。当前处理器中的这份缓存已经无效，一旦进入这个状态，处理器读数据就必须发出总线事务，从内存读

例如：一个处于S状态的缓存行，必须时刻监听使该缓存行无效的请求，如果监听到，则必须把其缓存行状态设置为I（其他处理器将缓存置为M状态，则当前缓存置为I）。

思考：

当我把缓存一致性原则理顺了之后，突然开始怀疑我之前的理论知识，昨天晚上没睡好，好像做梦也在想这个。

我的疑惑：既然处理器自己能够做到通过缓存一致性机制，保证各个处理器中的缓存和内存一致，那就不存在并发问题了，那Java就没有必要做内存模型去控制并发问题了？

后来我突然想明白了：

其实缓存一致性并不能保证随时的内存可见性，CPU为了提高效率，不会时刻去做这些缓存一致性的处理的，只有在JVM声明了锁操作的时候（也就是Lock#信号或者内存屏障），会\*立刻触发\*缓存行中数据刷新到主存中，此时处理器缓存一致性保证内存的可见性，普通对象共享缓存行时，处理器在合适的时机（未知的时间）才会触发缓存行刷新到主内存，在这个等待过程中，缓存行可能会被其他线程复写。

总结：

多核处理器下，CPU执行锁操作或者内存屏障时，使用MESI（我称之为梅西）协议维护了缓存一致性。当共享的缓存行数据修改后，由于存在监听（嗅探）机制会失效其他处理器的共享的该缓存行，并把修改后的数据写入内存，处理器针对失效的缓存行进行读操作时，从内存中读取最新的数据，这样就实现了缓存一致性。

15.reenterlocker和sychronize的区别？

二面：

1、自我介绍

2、介绍实习项目

3、Kafka监听接口如何实现，为何采用Kafka而不是用其他消息队列

4、实现Kafka监听接口的时候，遇到哪些坑，采取了哪些措施或者解决方案

5、Kafka的基本概念，及特点和优势。offset+zk+持久化分区日志顺序存储之类的

6、MySQL了解吗，介绍一下InnoDB

7、JVM了解吗，介绍一下，讲了垃圾回收机制

8、Full GC和Minor GC区别，及各自的触发条件

9、说说CMS垃圾回收器，及其适用场景

HR面：

HR小姐姐，真的很漂亮=。=

HR面比较常规，相比于那些让讲笑话之类的，中规中矩，有没有实习，在哪儿实习，负责什么系统，为何让你独立负责，是否拿到offer，offer薪资多少，哪里人，以后打算在哪儿发展，是否接受入职前实习，是否接受996。

考拉面经2：

自我介绍

面试岗位：java研发

时间 ：2017-08-23

部门 ： 考拉海购

面试次数：2轮技术面+1面HR面

一面技术

1、介绍项目

2、spring java反射

3、数据结构题

4、班级表和学生表，利用sql 语句查询每个班级的学生人数

create table student(stu\_id int,name varchar(10),class\_id int);

insert into student values(1,'w',1);

insert into student values(2,'y',1);

insert into student values(3,'z',2);

insert into student values(4,'zx',3);

create table class(class\_id int, class\_name varchar(10));

insert into class value(1,'一班');

insert into class value(2,'二班');

insert into class value(3,'三班');

insert into class value(4,'四班');

select ifnull(s.class\_id,c.class\_id) class\_id,ifnull(count(s.class\_id),0) num from class c left join student s on c.class\_id=s.class\_id group by s.class\_id,c.class\_id order by num desc;

Select classId,count() from students group by classId

5、int和Integer区别

二面

1、自我介绍

2、情感摘要介绍

3、spring java反射

4、i++原子操作

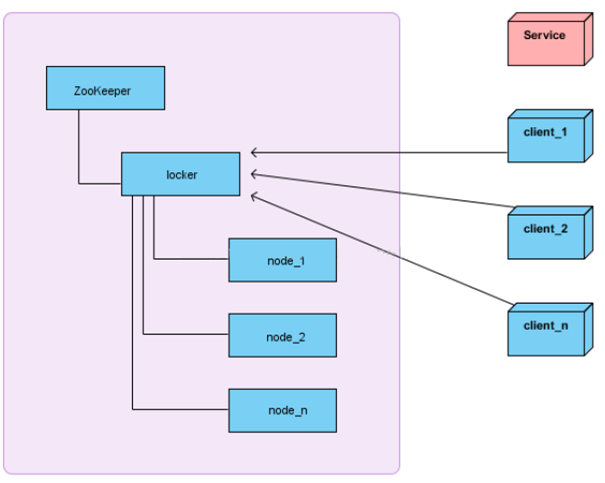
5、hashmap和hashtable区别，以及concurrenthashmap

6、zookeeper 实现分布式锁

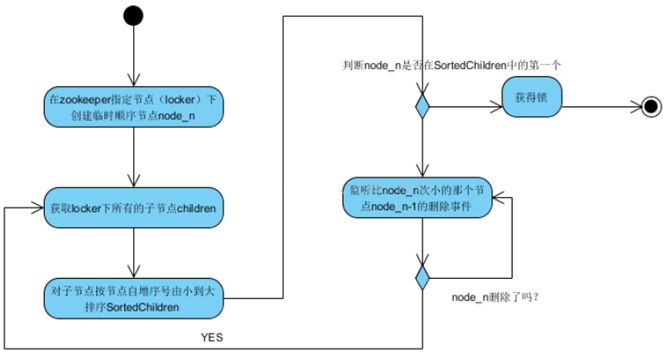
#### ****参考答案****：

有了zookeeper的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。 对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的distribute\_lock 节点就释放出锁。 对于第二类， /distribute\_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，编号最小的获得锁，用完删除，依次方便。

**获取分布式锁的流程**

[](https://github.com/0voice/interview_internal_reference/blob/master/12.Zookeeper%E7%AF%87/zk_distributed_locker1.png)

在获取分布式锁的时候在locker节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用createNode方法在locker下创建临时顺序节点， 然后调用getChildren(“locker”)来获取locker下面的所有子节点，注意此时不用设置任何Watcher。客户端获取到所有的子节点path之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非locker所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，然后对其调用exist()方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的Watcher会收到相应通知，此时再次判断自己创建的节点是否是locker子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。

[](https://github.com/0voice/interview_internal_reference/blob/master/12.Zookeeper%E7%AF%87/zk_distributed_locker2.png)

代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于BaseDistributedLock，实现了基于Zookeeper实现分布式锁的细节。

7、团购系统的表设计

8、（没问题）悲观锁 和乐观锁 以及sql如何实现

悲观锁、乐观锁的区别及使用场景

定义：

悲观锁（Pessimistic Lock）：

每次获取数据的时候，都会担心数据被修改，所以每次获取数据的时候都会进行加锁，确保在自己使用的过程中数据不会被别人修改，使用完成后进行数据解锁。由于数据进行加锁，期间对该数据进行读写的其他线程都会进行等待。

代表：sychronize，lock

实现：

乐观锁（Optimistic Lock）：

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

实现：CAS实现和version方式

version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

核心SQL代码：

|  |  |
| --- | --- |
| 1 | update table set x=x+1, version=version+1 where id=#{id} and version=#{version}; |

CAS操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

代表：

适用场景：

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

什么是AbstractQueuedSynchronizer？

AbstractQueuedSynchronizer是JUC并发包中锁的底层支持，AbstractQueuedSynchronizer是抽象同步队列，简称AQS，是实现同步器的基础组件，并发包中锁的实现底层就是使用AQS实现，当然大多数人不会直接用到AQS，但是学习这个类对并发包的底层理解还是有莫大的帮助的。

AQS中维持了一个单一的状态信息state，可以通过getState,setState,compareAndSetState 函数修改其值，AQS内部维持一个FIFO队列来完成资源获取线程的排队工作。对于ReentrantLock 的实现来说，state 可以用来表示当前线程获取锁的可重入次数；

对应读写锁ReentrantReadWriteLock 来说state 的高16位表示读状态，也就是获取该读锁的次数，低 16位 表示获取到写锁的线程的可重入次数；对于FuterTask 来说，state用来表示任务状态(例如，还没开始，运行，完成，取消）；

对应CountDownlatch 和CyclicBarrie 来说，state用来表示计数器当前的值。

AQS有个内部类ConditionObject 是用来结合锁实现线程同步，ConditionObject可以直接访问AQS对象内部的变量，比如 state 状态值 和AQS队列；

ConditionObject是条件变量，每个条件变量对应一个条件队列（单向链表队列)，用来存放调用条件变量的await（）方法后被阻塞的线程。

对于AQS 来说，线程同步的关键是对状态值state进行操作，根据state是否属于一个线程，操作state的方式分为独占模式和共享模式。

独占模式下获取和释放资源使用方法的源码如下：

void acquire(int arg)

void acquireInterruptibly(int arg)

boolean release(int arg)

共享模式下获取和释放资源方法的源码如下：

void acquireShared(int arg)

void acquireSharedInterruptibly(int arg)

boolean releaseShared(int arg)

另外还有个查询同步队列等待线程情况的方法如下：

Collection<Thread> getQueuedThreads()

对于独占锁方式获取的资源是与具体线程绑定的，也就是说如果一个线程获取到了资源，就会标记是那个线程获取到的，其他线程尝试操作state获取资源时候发现当前该资源不是自己持有，就会获取失败后被阻塞；

比如独占锁ReentrantLock的实现，当一个线程获取了ReentrantLock的锁后，AQS内部会首先使用CAS操作把state状态从0 变成 1，然后设置当前锁的持有者为当前线程，当该线程再次获取锁的时候，发现当前线程就是锁的持有者，则会把state状态值从1变成2，

也就是设置可重入次数，当另外一个线程获取锁的时候发现自己并不是该锁的持有者就会被放入AQS阻塞队列后挂起。

对于共享操作方式资源是与具体线程不相关的，多个线程去请求资源时候是通过CAS方式竞争获取资源，当一个线程获取到了资源后，另外一个线程再次获取时候，如果 当前资源还能满足它的需要，则当前线程只需要使用CAS方式进行获取即可，

共享模式下并不需要记录哪个线程获取了资源；比如 Semaphore 信号量，当一个线程通过acquire()方法获取一个信号量时候，会首先看当前信号两个数是否满足需要，不满足则把当前线程放入阻塞队列，如果满足则通过自旋CAS获取信号量。

JVM优化经验（重要重要重要）（需要搞清楚）

HR面

1、自我介绍

2、研究生的学习情况

3、意向工作地点

4、碰到的问题，解决方法，以及实现了什么效果

5、担任职务

6、实习情况

360面试总结

hdfs中的读写操作？

为什么三次握手 四次挥手？

**TCP连接建立过程：**

1、客户端向服务器发送SYN，其中seq=x。

2、服务器收到SYN报文段后，发送SYN+ACK，其中seq=y，确认号=x+1。

3、客户端收到SYN+ACK报文段后，发送ACK，确认号=y+1。服务器收到ACK报文段后，连接建立。

**为什么连接需要三次握手？**

三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误”

假设连接时时2次握手，正常情况下，客户端发送请求，服务器返回确认，这样是没问题的，但是出现另外一种异常情况，比如客户端发送请求，因为网络延时，客户端发出的请求延误到连接释放以后的某个时间才到达服务器，本来是一个早已失效的报文段，但是服务器接受到之后，误认为客户端有发出了一次新的连接，服务器端返回确认，新连接就创建了，由于客户端没有发出请求，并不会理睬服务器的确认，所以不会向服务器发送数据，但是服务器确认为连接已经建立，并一直等待客户端客户端发来数据，服务器的许多资源就这样白白浪费。

由于三次握手之后，双方发送和接受正常。所以四次没必要

**TCP连接断开过程：**

1、客户端TCP模块在收到应用程序的通知后，发送FIN，seq=x。

2、服务器收到FIN报文段，发送ACK，确认号=x+1，并且通知应用程序客户端关闭了连接。客户端收到ACK报文段。

3、服务器端的应用程序通知TCP关闭连接，服务器端TCP发送FIN+ACK，seq=y，确认号=x+1（这里ACK只是一般性的捎带ACK，TCP总是这样，以增强健壮性，反正也不费力气，从原理上说，对连接断开不是必须的）。

4、客户端收到FIN+ACK报文段后，发送ACK，确认号y+1。服务器收到ACK报文段后，连接断开。

**为什么断开需要4次握手？**

Ps: 全双工传输：是指交换机在发送数据的同时也能够接收数据，两者同步进行，所以连接关闭的时候，通信双方都需要独立关闭自己的通信通道，也就是半关闭。

TCP的连接释放的四次握手，可以看成2个2次握手

client先发送FIN告知对方我已经完成数据发送了，server回复ack来确定我知道了。这样一个流程，就关闭了client到服务端的连接。但是还可以接收来自server方的数据。

server此时已经知道接收不到client的数据了，但是还可以给它发送数据。如果server也没有啥数据要发送给对方了，server也会以FIN标志位发送一个信息给client，client接到后，也会传递一个ack表示知道了。这样子，双方都完成了关闭。

**为什么客户端A在time-wait状态必须等待2MSL的时间？**

**1.为了保证A发送的最后一个ACK报文段能够到达B。这个ACK报文段有可能丢失，因而使得处于LAST-ACK状态的B收不到已发送的FIN+ACK的确认。如果超时，B会重传FIN+ACK，A就能在2MSL之内收到B的重传，接着A重传最后一个ACK，重新启动2MSL计时器。最后A和B都正常进入到CLOSED。如果A发送完最后一个ACK之后，不等待2MSL时间立即释放的话，那么就无法收到B的重传的FIN+ACK，因为也不会再发送一次ACK了，这样B就无法按照正常步骤进入CLOSED状态.**

**2.防止连接三次握手时提到的 “已失效的连接请求报文段出现在“ 出现在本连接中。A在发送最后一个ACK报文之后，在经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络消失，这样就可以使下一个新连接中，不会出现这种旧的连接请求。**

接口和抽象类的区别？接口中是否可以放属性？

hashmap中key有什么要求？

<https://blog.csdn.net/weixin_42054155/article/details/80715750>

1、HashMap 的 key 和 value 都可以是 null,

2、Map 的 key 和 value 都 不允许 是 基本数据类型

3、HashMap 的 key 可以是 任意对象，意思都要继承Object类的hashcode()方法，但如果 对象的 hashCode 改变了，那么将找不到原来该 key 所对应的 value, 所以最好Key用final修饰。

hashmap的get的流程？

equal 和等号的区别

java8 9的新功能

一个线程接着一个线程执行 有什么方法？

<https://blog.csdn.net/qq_35571554/article/details/82834486>

public static void main(String[] args) throws InterruptedException {  
  
 final Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 add();  
 }  
 });  
 final Thread t2 = new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 //引用t1线程，等待t1线程执行完  
 t1.start();  
 t1.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 add();  
 }  
 });  
 Thread t3 = new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 //引用t2线程，等待t2线程执行完,一定要在此处t2.start()  
 //在主线程中调用t2.start()执行顺序不确定，因为若t2.join()先于t2.start()运行  
 //导致结果不确定  
 t2.start();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 add();  
 }  
 });  
 t3.start();  
// t2.start();  
// t1.start();  
 }

使用thread.join方法， 首先join() 是一个synchronized方法， 里面调用了wait()，

当t1调用join方法时，相当于在外部线程中写了一个同步同步代码块，

t1.start()

sychronized(t1) {

while(true)

t1.wait() // 当前线程等待

}

}

add()

当t1线程执行完之后，也就是执行了t1.notofyall(), 也就相当于唤醒当前线程，然后当前线程继续执行add()。

创建线程的方式

如何停止一个线程？