

大数据开发面试八股文



作者：三石

公众号：三石大数据

目录

目录.....	2
第一部分 大数据开发.....	17
HDFS	17
1. HDFS 的架构 **	17
2. HDFS 的读写流程 ***	17
3. HDFS 中，文件为什么以 block 块的方式存储	18
4. 小文件过多有什么危害，你知道的解决办法有哪些 **	18
5. 在 NameNode HA 中，会出现脑裂问题吗？怎么解决脑裂.....	18
6. 简述 hadoop 压缩和解压缩的框架.....	19
7. namenode 的安全模式有了解吗	19
8. Secondary NameNode 了解吗，它的工作机制是怎样的	20
9. 在上传文件的时候，其中一个 DataNode 突然挂掉了怎么办 *	20
10. 在读取文件的时候，其中一个块突然损坏了怎么办 *	21
11. 介绍 namenode 宕机的数据恢复过程	21
12. NameNode 在启动的时候会做哪些操作	21
MapReduce	22
1. 简述 MapReduce 整个流程 ***	22
2. 手写 wordcount *	22
3. join 原理 **	23
4. 文件切片相关问题.....	24
5. 环形缓冲区的底层实现.....	24
6. 全排序.....	25
7. MapReduce 实现 TopK 算法 *	25
Yarn	25
1. 简述 yarn 集群的架构 **	25
2. yarn 的任务提交流程是怎样的 ***	26
3. yarn 的资源调度的三种模型 **	26
4. 简述 Hadoop1.0 2.0 3.0 区别 ***	26
5. 任务的推测执行（spark UI 见过）	27
Zookeeper	27

1. 简述 leader 选举机制 ***	27
2. 简述什么是 CAP 理论，zookeeper 满足 CAP 的哪两个 *	28
3. zookeeper 集群的节点数为什么建议奇数台 ***	28
4. 简述 ZooKeeper 的监听原理.....	29
5. 请说一下 zookeeper 的典型应用场景有哪些 *	29
6. 客户端向服务端写数据流程.....	29
7. zookeeper 是如何实现分布式锁的 **	29
8. Paxos 算法 *	30
9. Zab 协议 *	30
Flume	31
1. 简述 flume 基础架构 **	31
2. 请说一下你提到的几种 source 的不同点 *	32
3. 简述 flume 的事务机制	32
4. flume 采集数据会丢失吗 *	32
5. 简述 flume 的 channel selector	32
Kafka.....	33
1. 为什么要使用 kafka **	33
2. 简述 kafka 的架构 ***	33
3. 命令行操作（了解）	34
4. 生产者发送流程 *	34
5. 简述 kafka 的分区策略 *	35
6. kafka 是如何保证数据不丢失和数据不重复 ***	36
7. kafka 中的数据是有序的吗，如何保证有序的呢 *	37
8. zookeeper 在 kafka 中的作用有哪些	38
9. broker 工作流程	38
10. 简述 kafka 消息的存储机制 ***	40
11. kafka 的数据是放在磁盘上还是内存上，为什么速度会快 **	40
12. kafka 消费方式.....	41
13 kafka 消息数据积压，消费者如何提高吞吐量.....	42
14. 你知道 Kafka 单条日志传输大小吗.....	43
15. Kafka 为什么同一个消费者组的消费者不能消费相同的分区 *	43
HBase.....	43
1. 简述 HBase 的数据模型 *	43
2. HBase 和 hive 的区别 **	44

3. HBase 的基本架构.....	44
4. 简述 HBase 的读写流程 ***	44
5. HBase 在写过程中的 region 的 split 时机 *.....	45
6. HBase 中 compact 用途是什么，什么时候触发，分为哪两种，有什么区别 *.....	45
7. 热点现象怎么产生的，以及解决方法有哪些 **	46
8. 说一下 HBase 的 rowkey 设计原则 ***	46
9. 列族的设计规则 **	47
Hive.....	47
1. 简述 hive **	47
2. 简述 hive 读写文件机制.....	48
3. hive 和传统数据库之间的区别 ***	49
4. hive 的内部表和外部表的区别 ***	49
5. hive 静态分区和动态分区的区别.....	49
6. 内连接、左外连接、右外连接的区别.....	50
7. hive 的 join 底层实现 ***	51
8. Order By 和 Sort By 的区别 **	51
9. 行转列和列转行函数 *	52
10. grouping_sets、cube 和 rollup	53
11. 自定义过 UDF、UDTF 函数吗 ***	53
12. hive3 的新特性有了解过吗.....	53
13. hive 小文件过多怎么办 *	54
14. Hive 优化 ***	55
15. 常用函数的补充.....	57
Spark	58
spark core	58
spark sql	66
spark streaming	68
Flink	70
1. 简单介绍一下 Flink **	70
2. Flink 和 SparkStreaming 区别 *	70
3. Flink 的重启策略你了解吗	70
4. Flink 的运行依赖于 hadoop 组件吗	71
5. Flink 集群有哪些角色？各自有什么作用 *	71

6. 简述 Flink 运行流程（基于 Yarn） **	71
7. max 算子和 maxBy 算子的区别 *	71
8. Connect 算子和 Union 算子的区别 *	72
9. Flink 的时间语义有哪几种 ***	72
10. 谈一谈你对 watermark 的理解 ***	72
11. Flink 对于迟到或者乱序数据是怎么处理的 ***	73
12. Flink 中，有哪几种类型的状态，你知道状态后端吗 **	73
13. Flink 是如何做容错的？	74
14. Flink 是如何保证 Exactly-once 语义的 ***	74
15. Flink 是如何处理反压的	76
16. Flink 是如何支持批流一体的 *	76
17. 你用过 Flink CEP 吗，简单介绍一下 *	76
第二部分 Java 基础	76
Java 基础	76
1. JDK、JRE、JVM 三者区别和联系 *	76
2. 基本数据类型和引用数据类型的区别 *	77
3. 8 种基本数据类型、字节大小 *	77
4. 访问修饰符权限 *	77
5. java 中方法的参数传递机制 **	78
6. final 关键字 **	78
7. static 关键字的作用是什么 *	79
8. Comparable 和 Comparator 区别 *	79
9. Object 类有哪些方法 *	79
10. java 的深拷贝和浅拷贝的区别 ***	80
11. java 中==和 equals 的区别 ***	80
12. String 和 StringBuffer、StringBuilder 的区别 ***	80
13. 简述面向对象三大特征 **	81
14. java 中方法重载和重写的区别 *	82
15. 抽象类和接口的区别 *	82
16. 集合之间的继承关系 **	83
17. ArrayList 和 LinkedList 区别 **	83
18. ArrayList 扩容过程 **	83
19. HashMap 底层实现 ***	83
20. HashMap 扩容过程 ***	84

21. HashMap 中为啥用红黑树不用二叉排序树或者平衡树 *	84
22. TreeMap 底层实现	85
23. HashMap 和 Hashtable 的区别 *	85
24. Hashtable 怎么保证线程安全的 *	85
25. ConcurrentHashMap 原理 ***	85
26. java 反射机制 ***	86
27. 异常体系 **	86
28. 常见的 IO 模型 *	86
29. 设计模式 *	87
30. 一致性 hash 算法 **	89
JVM.....	89
1. java 运行时一个类是什么时候加载的 *	89
2. JVM 一个类的加载过程 ***	90
3. 继承时父子类的初始化顺序是怎样的 ***	90
4. 什么是类加载器.....	90
5. JVM 有哪些类加载器 *	90
6. 什么是双亲委派模型 ***	91
7. JDK 为什么要设计双亲委派模型，有什么好处	91
8. 可以打破双亲委派模型吗？如何打破？	91
9. 如何自定义类加载器.....	92
10. ClassLoader 中的 loadClass()、findClass()、defineClass()区别	92
11. 加载一个类采用 Class.forName()和 ClassLoader.loadClass()有什么区别.....	92
12. Tomcat 的类加载机制.....	92
13. 为什么 Tomcat 要破坏双亲委派模型.....	92
14. 热加载和热部署，如何自己实现一个热加载.....	93
15. java 代码到底是如何运行起来的 *	93
16. JVM 内存结构 ***	94
17. Java 对象如何在堆内存分配	95
18. JVM 堆内存中的对象布局 *	96
19. JVM 什么情况下会发生堆内存溢出 **	96
20. JVM 如何判断对象可以被回收 ***	97
21. java 中不同的引用类型 *	97
22. JVM 堆中新生代的垃圾回收过程 **	97

23. JVM 对象动态年龄判断是怎么回事 *	98
24. 什么是老年代空间分配担保机制 *	98
25. 什么情况下对象会进入老年代 **	98
26. JVM 本机直接内存的特点及作用 *	98
27. 几个与 JVM 内存相关的核心参数 **	99
28. 堆为什么要分为新生代和老年代 *	99
29. 新生代为什么要有两个 survivor 区 **	99
30. eden 区与 survivor 区的空间大小比例为什么是 8:1:1 **	100
31. JVM 中的垃圾回收算法 ***	100
32. JVM 垃圾收集器 ***	102
33. full gc 是什么 *	106
并发编程.....	106
1. java 实现多线程有几种方式 ***	106
2. 线程池相关内容 ***.....	107
3. 线程有哪几种状态 **	109
4. sleep, wait, notify, yield 和 join 方法的区别 *	109
5. 什么是上下文切换 *	110
6. 设计一个简单的死锁程序 **	110
7. ThreadLocal 相关内容 **	111
8. 并发编程的三个问题 **	111
9. 介绍一下 java 内存模型 ***	112
10. synchronized 是如何保证三大特性 **	112
11. synchronized 的特性 **	114
12. synchronized 的原理 ***	114
13. ReentrantLock 底层原理 ***	115
14. synchronized 与 lock 的区别 ***	116
15. volatile 的作用 **	116
16. volatile 和 synchronized 的区别 ***	117
17. CAS 介绍一下 **	117
18. 乐观锁和悲观锁的区别 **	117
19. 锁升级的过程 *	118
20. synchronized 优化 *	119
Redis	119
1. redis 的数据类型 ***	119

2. zset 的底层实现 **	119
3. BitMaps.....	120
4. HyperLogLog.....	120
5. redis 的持久化方案 ***	121
6. 缓存穿透，缓存击穿，缓存雪崩 ***	121
7. redis 实现分布式锁 *	122
第三部分 计算机基础.....	123
计算机网络.....	123
1. OSI 七层模型 ***	123
2. TCP 连接管理 ***	124
3. TCP 连接建立为什么需要三次握手 **	124
4. TCP 连接释放为什么需要四次挥手 **	125
5. TCP 连接释放的第四次握手为什么要等待 2MSL **	125
6. TCP 是如何做到可靠传输的 ***	126
7. TCP 流量控制 **	126
8. TCP 拥塞控制 **	127
9. 流量控制和拥塞控制的区别 *	128
10. TCP 和 UDP 的区别 ***	128
11. 视频面试中用 TCP 还是 UDP *	128
12. UDP 如何实现可靠传输 *	128
13. TCP 粘包问题以及解决方案 *	129
14. 域名解析的过程 **	129
15. 浏览器输入 URL 到显示页面的过程 ***	130
16. 介绍 HTTP **	130
17. HTTP 1.0 和 HTTP 1.1 的主要区别是什么 *	130
18. HTTP 和 HTTPS 的区别 *	130
19. HTTP 是如何保存用户状态的 *	131
20. GET 和 POST 的区别 **	131
21. 常见的状态码 *	132
22. ARP 地址解析协议 *	133
操作系统.....	133
1. 什么是操作系统 *	133
2. 什么是系统调用 *	133
3. 进程和线程的区别 ***	134

4. 进程有哪几种状态 *	135
5. 进程间的通信方式 **	135
6. 进程同步的四种方式 **	136
7. 进程的调度算法 ***	136
8. 什么是死锁以及死锁的四个条件 ***	137
9. 常见的几种内存管理机制 *	138
10. 分页机制和分段机制的共同点和区别	138
11. 快表和多级页表 *	139
12. CPU 寻址了解吗?为什么需要虚拟地址空间?	139
13. 什么是虚拟内存(Virtual Memory) *	139
14. 什么是局部性原理	140
15. 虚拟存储器	140
16. 虚拟内存的几种实现 **	140
17. 页面置换算法 ***	141
18. 僵尸进程和孤儿进程是什么 *	141
数据库	142
1. 索引是什么 **	142
2. mysql 中索引的分类有哪些 **	142
3. B+树和 B 树的区别 ***	143
4. mysql 的索引结构 ***	143
5. 为什么不用二叉树, 红黑树, 哈希表, B 树 *	143
6. 聚集索引和非聚集索引的区别 **	144
7. 回表查询是什么 **	145
8. 覆盖索引是什么 **	145
9. 索引下推 *	146
10. 主键索引和辅助索引具体是什么 *	146
11. 为什么建议用自增 id 做索引而不用 UUID *	147
12. 主键索引使用 int 和 string 有啥区别 *	147
13. 缺少主键的话 mysql 怎么处理 *	147
14. 选什么字段当索引, 索引何时失效 ***	148
15. 索引合并和复合索引的区别 *	148
16. 简述事务 ***	149
17. 数据库事务并发会引发哪些问题 ***	149
18. 事务的四个隔离级别有哪些 ***	150

19. 什么是幻读，如何解决 *	150
20. MySQL 是如何保证 ACID 的 **	151
21. MySQL 支持的锁有哪些 *	151
22. MVCC 讲一下（怎么实现） **	152
23. 间隙锁讲一下 *	152
24. 怎么实现可重复读（读提交） *	153
25. select ... for update *	153
26. 乐观锁与悲观锁，mysql 如何实现乐观锁 ***	154
27. MySQL 中常见的几种日志 **	154
28. MySQL 主从复制的流程 *	155
29. 关系型数据库与非关系型数据库的区别 *	156
30. 说一说 drop、delete 和 truncate 的共同点和区别 **	156
31. 数据库 3 个范式 **	156
32. MySQL 中 char 和 varchar 的区别有哪些 *	157
33. MySQL 中 inner join、left join、right join 和 full join 的区别有哪些 **	157
34. MySQL 的执行顺序 *	157
35. having 和 where 的区别 *	158
36. MySQL 的存储引擎，以及区别 *	158
37. 大数据量里的分页查询怎么优化 *	158
38. SQL 的优化方法 *	159
39. MySQL 中视图和表的区别 *	159
40. 数据完整性约束 **	159
41. group by 的实现方式 *	159
数据结构.....	160
1. 链表和数组的区别.....	160
2. 栈和队列的区别.....	160
3. 红黑树了解吗.....	161
4. 常见的排序算法.....	162
5. 图的常见算法.....	162
第四部分 数仓基础.....	163
1. 数据仓库是什么.....	163
2. 数据仓库和数据库有什么区别.....	163
3. 为什么要对数据仓库分层.....	164

4. 为什么需要数据建模.....	164
5. 经典的数据仓库建模方法论有哪些.....	164
6. 数仓相关的名词术语解释，比如数据域、业务过程、衍生指标.....	164
7. 派生指标的种类.....	165
8. 经典数仓分层架构.....	165
9. 模型设计的基本原则.....	165
10. 模型实施的具体步骤.....	166
11. 维度建模有哪几种模型.....	167
12. 维度建模中表的类型.....	167
13. 维度表的设计过程.....	167
14. 维度表的设计中有哪些值得注意的地方.....	168
15. 维表整合的两种表现形式.....	168
16. 如何处理维度的变化.....	168
17. 事实表设计的八大原则.....	170
18. 事实表的设计过程.....	170
19. 事实表有哪几种类型.....	171
20. 多事务事实表如何对事实进行处理.....	171
21. 单事务事实表和多事务事实表哪种设计更好.....	172
22. 周期快照事实表的设计过程.....	172
23. 累计快照事实表的设计过程.....	173
24. 累计快照事实表的特点.....	173
第五部分 常考 SQL.....	174
1. 连续问题 ***.....	174
2. 分组问题 **.....	175
3. 间隔连续问题 *.....	176
4. 打折日期交叉问题 **.....	177
5. 同时在线问题 ***.....	178
6. 最大连续登陆的最大天数问题 ***.....	179
7. 留存问题 ***.....	180
第六部分 大数据开发场景题.....	181
1. 1亿个整数中找出最大的10000个数 **.....	181
2. 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	181
3. 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	182
4. 在 2.5 亿个整数中找出不重复的整数 **	182
5. 外部排序	182
6. 大数量中寻找中位数【网易】 **	183
7. 大整数相乘【网易】	183
8. 数据倾斜 ***	183
第七部分 大厂面经合集（持续更新~）	185
美团	185
30 篇面经合集	185
面经 1	186
面经 2	188
蚂蚁	189
面经 1	189
面经 2	189
阿里	190
面经 1	190
面经 2	191
字节跳动	191
20 篇面经合集	191
面经 1	192
面经 2	193
面经 3	194
百度	195
面经 1	195
面经 2	196
滴滴	197
面经 1	197
面经 2	198
网易	199
面经 1	199
面经 2	200
面经 3	200
快手	201
面经 1	201

面经 2.....	202
微众银行.....	203
面经 1.....	203
京东.....	204
面经 1.....	204
面经 2.....	205
携程.....	206
面经 1.....	206
面经 2.....	207
第八部分 大厂 SQL 真题（持续更新~）	207
短视频业务.....	207
1.各个视频的平均完播率.....	207
2.平均播放进度大于 60%的视频类别	208
3.每类视频近一个月的转发量/率	210
4.每个创作者每月的涨粉率及截止当前的总粉丝量.....	211
5.国庆期间每类视频点赞量和转发量.....	212
6.近一个月发布的视频中热度最高的 top3 视频	214
用户增长业务.....	216
1.2021 年 11 月每天的人均浏览文章时长.....	216
2.每篇文章同一时刻最大在看人数.....	217
3.2021 年 11 月每天新用户的次日留存率.....	218
4.统计活跃间隔对用户分级结果.....	219
5.每天的日活数及新用户占比.....	221
6.连续签到领金币.....	223
电商业务.....	224
1.计算商城中 2021 年每月的 GMV	224
2.统计 2021 年 10 月每个退货率不大于 0.5 的商品各项指标.....	225
3. 某店铺的各商品毛利率及店铺整体毛利率.....	226
4.零食类商品中复购率 top3 高的商品	228
5.10 月的新户客单价和获客成本.....	229
6.店铺 901 国庆期间的 7 日动销率和滞销率.....	230
打车业务.....	232
1.2021 年国庆在北京接单 3 次及以上的司机统计信息.....	232
2.有取消订单记录的司机平均评分.....	233

3. 每个城市中评分最高的司机信息.....	234
4. 国庆期间近 7 日日均取消订单量.....	235
5. 工作日各时段叫车量、等待接单时间和调度时间.....	237
6. 各城市最大同时等车人数.....	238
店铺分析业务.....	239
1. 某宝店铺的 SPU 数量	239
2. 某宝店铺的实际销售额与客单价.....	240
3. 某宝店铺折扣率.....	241
4. 某宝店铺动销率与售罄率.....	241
5. 某宝店铺连续 2 天及以上购物的用户及其对应的天数.....	242
教育业务.....	243
1. 牛客直播转换率.....	243
2. 牛客直播开始时各直播间在线人数.....	244
3. 牛客直播各科目平均观看时长.....	244
4. 牛客直播各科目出勤率.....	245
5. 牛客直播各科目同时在线人数.....	246
内容业务.....	248
1. 某乎问答 11 月份日人均回答量.....	248
2. 某乎问答高质量的回答中用户属于各级别的数量.....	248
3. 某乎问答单日回答问题数大于等于 3 个的所有用户.....	249
4. 某乎问答回答过教育类问题的用户里有多少用户回答过职场类问题.....	250
5. 某乎问答最大连续回答问题天数大于等于 3 天的用户及其对应等级.....	250
第九部分 企业级调优手法.....	252
Hadoop.....	252
1.job 执行三原则	252
2.shuffle 调优	253
3.job 调优	254
4.yarn 调优	255
5.namenode full gc.....	256
6.mapreduce 生产调优	257
Hive.....	258
1. 避免使用 count(distinct col)	258

2.小文件问题.....	258
3.避免使用 select *	259
4.不要在表关联之后加 where.....	259
5.尽量删除字段中带有空值的数据.....	259
6.设置并行执行任务数.....	260
7.设置合理的 Reducer 个数	260
8.开启 JVM 重用.....	261
9.为什么任务执行的时候只有一个 reduce.....	261
10.选择使用 Tez 引擎.....	261
11.选择使用本地模式.....	262
12.选择使用严格模式.....	262
Spark	262
1.常规性能调优.....	262
2.算子调优.....	264
3.shuffle 调优	265
4.JVM 调优.....	266
5.Spark 数据倾斜	267
第十部分 数据湖基础.....	267
1.你觉得数据仓库有哪些优点和缺点.....	267
2.数据湖是什么.....	268
3.聊聊数据湖和数仓之间的区别.....	268
4.你接触过的数据湖产品有哪些，分别有什么特性.....	269
5.数据湖可以替代数仓吗.....	270
6.湖仓一体了解过吗.....	270
第十一部分 项目经历.....	270
1.简历如何写项目经历.....	270
2.面试涉及到项目的环节有哪些.....	270
3.面试如何介绍项目经历.....	271
4.项目一之离线数仓.....	271
简单介绍一下你们的离线数仓架构.....	271
介绍一下这个项目的整体流程.....	271
你有没有遇到过什么问题.....	271
你认为你在这个项目中做的比较好的地方有哪些.....	272
项目中用到的框架的八股文要熟悉，问项目的同时也会问到.....	272

5.项目二之实时数仓.....	272
简单介绍一下你们的实时数仓架构.....	272
介绍一下这个项目的整体流程.....	272
你有没有遇到过什么问题.....	272
你认为你在这个项目中做的比较好的地方有哪些.....	272

三石大数据

第一部分 大数据开发

HDFS

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.txt /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. HDFS 中，文件为什么以 block 块的方式存储

- 主要是减少硬盘寻道时间
 - 不设置 block：因为数据是分散的存放磁盘上的，读取数据时需要不停的进行磁盘寻道，开销比较大。
 - 使用 block：一次可以读取一个 block 中的数据，减少磁盘寻道的次数和时间。

4. 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

5. 在 NameNode HA 中，会出现脑裂问题吗？怎么解决脑裂

- 假设 NameNode1 当前为 Active 状态，NameNode2 当前为 Standby 状态。如果某一时刻 NameNode1 对应的 ZKFailoverController 进程发生了"假死"

现象，那么 Zookeeper 服务端会认为 NameNode1 挂掉了，NameNode2 会替代 NameNode1 进入 Active 状态。但是此时 NameNode1 可能仍然处于 Active 状态正常运行，这样 NameNode1 和 NameNode2 都处于 Active 状态，都可以对外提供服务。这种情况称为脑裂

- Zookeeper 对这种问题的解决方法叫做 **fencing**（隔离），也就是想办法把旧的 Active NameNode 隔离起来，使它不能正常对外提供服务
 - 首先尝试调用这个旧 Active NameNode 的 HAServiceProtocol RPC 接口的 transitionToStandby 方法，看能不能把它转换为 Standby 状态。
 - 如果 transitionToStandby 方法调用失败，那么就执行 Hadoop 配置文件之中预定义的隔离措施，Hadoop 目前主要提供两种隔离措施，通常会选择 sshfence：
 - ◆ sshfence：通过 SSH 登录到目标机器上，执行命令 fuser 将对应的进程杀死
 - ◆ shellfence：执行一个用户自定义的 shell 脚本来将对应的进程隔离

6. 简述 hadoop 压缩和解压缩的框架

- gzip 压缩：压缩率比较高，而且压缩/解压缩速度也比较快，hadoop 本身支持，但是不支持切片
- bzip2 压缩：比 gzip 的压缩率更高，hadoop 本身支持，而且还支持切片，但是压缩/解压缩速度很慢
- lzo 压缩：合理的压缩率，压缩/解压缩速度也比较快，而且支持切片，但是 hadoop 本身不支持，需要安装，压缩率要比 gzip 低一些，为了支持切片，还需要手动为 lzo 压缩文件创建索引
- snappy 压缩：合理的压缩率，压缩/解压缩速度也比较快，但是不支持切片，hadoop 本身不支持，需要安装，压缩率要比 gzip 低一些

7. namenode 的安全模式有了解吗

- 安全模式是指 hdfs 对于客户端来说是只读的
- 什么时候会进入安全模式？
 - namenode 启动的时候，首先会进行 fsimage 和 edits 的合并，然后 namenode 监听 datanode，datanode 汇报最新的数据块信息，这个过程 namenode 就会处于安全模式
- 什么时候会退出安全模式？
 - 如果整个文件系统中 99.9% 的数据块满足最小的副本级别（默认为 1），

namenode 会在 30 秒之后退出安全模式。刚初始化 HDFS 的时候，因为系统中没有任何块，所以 namenode 不会进入安全模式

- 命令：`bin/hdfs dfsadmin -safemode get/enter/leave/wait`

8. 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

9. 在上传文件的时候，其中一个 DataNode 突然挂掉了怎么办 *

客户端上传文件时与 DataNode 建立 pipeline 管道，管道正向是客户端向 DataNode 发送的数据包，管道反向是 DataNode 向客户端发送 ack 确认，也就是正确接收到数据包之后发送一个已确认接收到的应答，当 DataNode 突然挂掉了，客户端接收不到这个 DataNode 发送的 ack 确认，客户端会通知 NameNode，NameNode 检查该块的副本与规定的不符，NameNode 会通知 DataNode 去复制副本，并将挂掉的 DataNode 作下线处理，不再让它参与文件上传与下载。

10. 在读取文件的时候，其中一个块突然损坏了怎么办 *

客户端读取完 DataNode 上的块之后会进行 checksum 验证，也就是把客户端读取到本地的块与 HDFS 上的原始块进行校验，如果发现校验结果不一致，客户端会通知 NameNode，然后再从下一个拥有该 block 副本的 DataNode 继续读

11. 介绍 namenode 宕机的数据恢复过程

当 namenode 发生故障宕机时，secondary namenode 会保存所有的元数据信息，在 namenode 重启的时候，secondary namenode 会将元数据信息发送给 namenode。

12. NameNode 在启动的时候会做哪些操作

- 读取 fsimage 和 edits 文件，然后通知 secondary namenode 进行合并生成新的 fsimage 文件，然后创建新的空的 edits 文件
 - 首次启动 NameNode
 - ◆ 格式化文件系统，为了生成 fsimage 镜像文件
 - ◆ 启动 NameNode
 - ◆ 读取 fsimage 文件，将文件内容加载进内存
 - ◆ 等待 DataNade 注册与发送 block report
 - ◆ 启动 DataNode
 - ◆ 向 NameNode 注册
 - ◆ 发送 block report（每 3 秒发一次心跳包：本节点的存储容量信息，块信息）
 - ◆ 检查 fsimage 中记录的块的数量和 block report 中的块的总数是否相同
 - ◆ 对文件系统进行操作（创建目录，上传文件，删除文件等）
 - ◆ 此时内存中已经有文件系统改变的信息，但是磁盘中没有文件系统改变的信息，此时会将这些改变信息写入 edits 文件中，edits 文件中存储的是文件系统元数据改变的信息。
 - 第二次启动 NameNode
 - ◆ 读取 fsimage 和 edits 文件
 - ◆ 将 fsimage 和 edits 文件合并成新的 fsimage 文件
 - ◆ 创建新的 edits 文件，内容为空
 - ◆ 启动 DataNode

MapReduce

1. 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。

2. 手写 wordcount *

Java

```
public class WordcountMapper extends Mapper<LongWritable, Text, Text,
IntWritable>{
    Text k = new Text();
    IntWritable v = new IntWritable(1);
    @Override
    protected void map(LongWritable key, Text value, Context context) throws
```

```

IOException, InterruptedException {
    // 1 获取一行
    String line = value.toString();
    // 2 切割
    String[] words = line.split(" ");
    // 3 输出
    for (String word : words) {
        k.set(word);
        context.write(k, v);
    }
}
}

public class WordcountReducer extends Reducer<Text, IntWritable, Text,
IntWritable>{
    int sum;
    IntWritable v = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
    // 1 累加求和
    sum = 0;
    for (IntWritable count : values) {
        sum += count.get();
    }
    // 2 输出
    v.set(sum);
    context.write(key,v);
}
}

```

3. 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

4. 文件切片相关问题

- 区分数据切片和数据块：
 - **数据切片**：只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。
 - **数据块**：默认为 128M，hdfs 物理上把数据分成一块一块
- MapTask 并行度决定机制：
 - 一个 job 的 map 阶段并行度由客户端在提交 job 时的切片数决定
 - 每一个切片分配一个 MapTask 进行处理
 - 默认情况下，切片大小=**blocksize**
 - 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

5. 环形缓冲区的底层实现

- 环形缓冲区底层就是一个数组，默认大小是 100M
- 数组中存放着 key 和 value 的数据，以及关于 key 和 value 的元数据信息。每个 key, value 对应一个元数据，元数据由 4 个 int 组成，第一个 int 存放 value 的起始位置，第二个 int 存放 key 的起始位置，第三个 int 存放 partition，第四个 int 存放 value 的长度。
- key/value 数据和元数据在环形缓冲区中的存储是由 equator（赤道）分隔的，key/value 按照索引递增的方向存储，元数据则按照索引递减的方向存储。将数组抽象为一个环形结构之后，以 equator 为界，key/value 顺时针存储，元

数据逆时针存储。

6. 全排序

补充知识：MapTask 和 ReduceTask 都会对数据按照 key 进行排序。该操作属于 hadoop 的默认行为。（字典顺序，快速排序）

区内排序：

1. 在全排序的基础上自定义分区（继承 Partitioner 类，重写 getPartition 方法）
 2. job 驱动中设置自定义分区类，并且设置 ReduceTask 个数
- 定义：最终输出结果只有一个文件，且文件内部有序，实现方式是只设置一个 **ReduceTask** 【慎用】
 - 关键步骤：
 1. 新建一个 javabean 类，用来存储输入的数据
 2. bean 对象作为 key 传输，实现 **WritableComparable** 接口，重写 compareTo 方法，序列化和反序列化方法。
 - 补充题：辅助排序（二次排序）
 - 二次排序：有两个字段参与排序
 - ◆ 跟一次排序实现的步骤一样，就在于逻辑上的不同，多来一个 ifelse 就可以了
 - 辅助排序：ReduceTask 对从 map 端拉取过来数据再次进行分组排序
 - ◆ 自定义类继承 WritableComparator，重写 compare 方法
 - ◆ job 驱动中设置 reduce 端的自定义分组类

7. MapReduce 实现 TopK 算法 *

- 自定义排序比较器（这里可以讲一下步骤），全局排序就行了
- 优化手段：在 map 阶段加上 TreeMap 集合，当集合大小超过 K 时，删除最小的元素，然后在 cleanup 方法中发送 topk 的元素；同时在 reduce 阶段也是一样的，因为会有不同的 maptask 发送 topk 过来，我们还需要再次去找 topk

Yarn

1. 简述 yarn 集群的架构 **

- Yarn 的基本思想就是将 Hadoop1.0 中的 jobTracker 拆分为了两个独立的服务： ResourceManager 和 ApplicationMaster。

- ResourceManager 负责整个系统的资源管理和分配，ApplicationMaster 负责单个应用程序的管理。除了这两个部分之外，还包括 NodeManager 和 Container，其中 NodeManager 是单个节点上的资源和任务管理器，Container 是 Yarn 的资源抽象，封装了各种资源，比如内存 CPU 磁盘。

2. 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

3. yarn 的资源调度的三种模型 **

- yarn 的资源调度器主要有三种：FIFO（先进先出调度器）、Capacity Scheduler（容量调度器）和 Fair Scheduler（公平调度器），Hadoop2.x 默认的是 Capacity Scheduler。
 - 先进先出调度器，支持单队列，先进先出【生产环境不会用】
 - ◆ 做法：先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配
 - 容量调度器，支持多个队列，每个队列采用先进先出策略
 - 公平调度器，也支持多个队列，保证每个任务公平享有队列资源
 - ◆ 做法：当第一个大 job 提交时，只有这一个 job 在运行，此时它获得了所有集群资源；当第二个小任务提交后，Fair 调度器会分配一半资源给这个小任务，让这两个任务公平的共享集群资源

4. 简述 Hadoop1.0 2.0 3.0 区别 ***

- hadoop1.0 和 hadoop2.0 的区别：
 - 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

5. 任务的推测执行 (spark UI 见过)

- 发现拖后腿的任务，比如某个任务运行速度远慢于任务平均速度。为拖后腿任务启动一个备份任务，同时运行。谁先运行完，则采用谁的结果。
【mapred-site.xml 配置】

Zookeeper

Zookeeper 是一个开源的分布式的，为分布式应用提供协调服务的 Apache 项目（文件系统+通知机制）

paxos 算法（今日头条面试题）

总结一下哪些地方用到了 zookeeper？

1. HDFS HA：配置高可用
2. kafka：选举 controller，注册 broker，注册 topic
3. HBase：保证 Master 的高可用、RegionServer 的监控

1. 简述 leader 选举机制 ***

- zookeeper 刚启动的时候：投票过半数时，服务器 id 大的胜出
 - 我举个例子吧，假设有 3 台服务器，服务器 1 先启动，此时只有它一台服务器启动了，没有任何服务器可以进行通信，因此处于 Looking 状态，紧接着服务器 2 启动，它就会和 1 进行通信，交换选举结果，此时 id 较大的 2 胜出，并且满足半数以上的服务器同意选举 2，所以 2 就成为了 leader，最后服务器 3 启动，虽然自己的 id 大一些，但是前面已经选出

了 leader，因此自己就成为了 follower

- leader 挂掉了之后： epoch 大的直接胜出；如果 epoch 相同，事务 id 大的胜出；如果事务 id 相同，服务器 id 大的胜出
- leader, follower, observer：
 - leader 是 zookeeper 集群的中心，负责协调集群中的其他节点，还有发起与提交写请求
 - follower 与老大 Leader 保持心跳连接，当 Leader 挂了的时候，经过投票后成为新的 leader
 - observer 的主要作用是提高 zookeeper 集群的读性能，不参与 leader 选举，没有投票权。

2. 简述什么是 CAP 理论，zookeeper 满足 CAP 的哪两个 *

- C 表示 Consistency（一致性，也就是从每个节点读取的数据是一样的），A 表示 Availability（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

3. 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

4. 简述 ZooKeeper 的监听原理

- 首先创建 zookeeper 客户端，这时就会创建两个线程，一个负责连接的 connect 线程，还有一个负责监听的 listener 线程，然后 connect 线程会将注册的监听事件发送给服务端，然后一旦触发监听事件，就会将消息发送给 listener 线程，listener 会调用 process 方法进行自定义的处理

5. 请说一下 zookeeper 的典型应用场景有哪些 *

- 统一命名服务：在分布式环境下，经常需要对服务进行统一命名，便于识别，例如 ip 地址
- 统一配置管理：在一个集群中，要求所有节点的配置信息是一致的
- 统一集群管理：在一个集群中，需要实时监控每个节点的状态变化
- 负载均衡：在 zookeeper 中记录每台服务器的访问数，再次请求的时候，让访问最少的服务器去处理当前请求

6. 客户端向服务端写数据流程

- 分为两种：
 - 将写入请求直接发送给 leader 节点
 - ◆ 半数以上的节点写入完毕，就会向客户端发送 ACK
 - 将写入请求发送给 follower 节点
 - ◆ 会将写入请求转发给 leader 节点，leader 节点写完后，再向 follower 进行同步，达到半数以后，向客户端发送 ACK

7. zookeeper 是如何实现分布式锁的 **

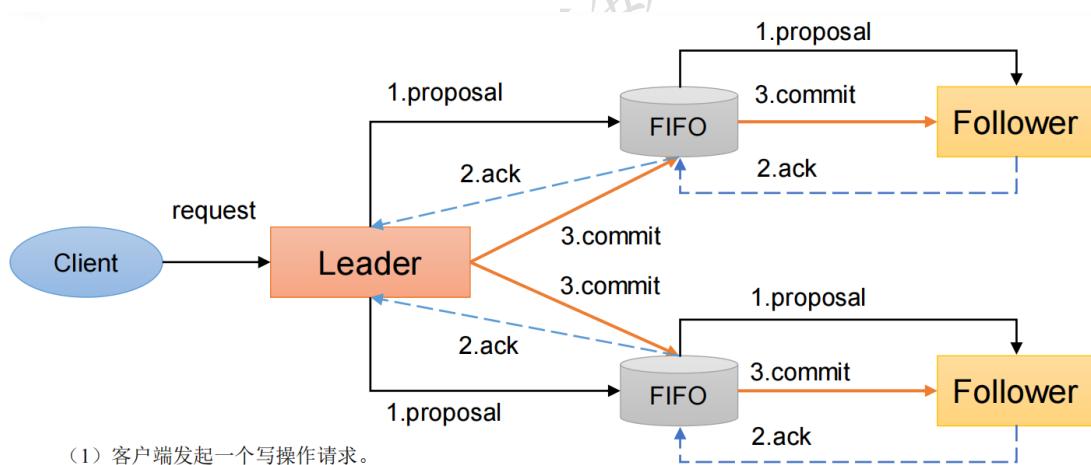
- 首先 zookeeper 集群接收到获取锁的请求时，就会在 locks 节点下创建一个临时顺序节点
- 然后先对当前节点下的所有子节点进行排序，判断自己是不是最小的节点，如果是，就可以获取到锁，如果不是，就说明锁已被其他客户端获取，然后可以对前一个节点进行监听，等待客户端把自己创建的节点删除
- 获取到锁后，进行业务的处理，释放锁的时候删除该临时节点，然后后面的

节点就会接收到通知。

8. 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)

9. Zab 协议 *



- 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。

Flume

Flume 是一个日志采集系统，主要作用是实时读取服务器本地磁盘的数据，将数据写入 HDFS 中

开启 Flume 命令：`bin/flume-ng agent -c conf/ -n a1 -f job/flume-netcat-logger.conf -Dflume.root.logger=INFO,console`

同类产品：DataX，Sqoop

1. 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

2. 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

3. 简述 flume 的事务机制

- flume 使用两个独立的事务 put 和 take 分别负责从 Source 到 Channel，以及从 Channel 到 Sink 的事件传递
- put 事务流程：
 - 将批数据先写入临时缓冲区 putList 中【doPut】，然后检查 channel 内存队列是否有空间【doCommit】，如果内存队列空间不足，回滚数据【doRollback】
- take 事务流程：
 - 将数据拉取到临时缓冲区 takeList 中，并将数据发送到目的地组件中【doTake】，如果全部发送成功，则清除临时缓冲区 takeList【doCommit】，如果发送的过程中出现异常，进行回滚【doRollback】

4. 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

5. 简述 flume 的 channel selector

- 作用是将事件按照不同的需求通过不同的 channel 发往不同的 sink 中
- 有两种类型：replicating channel selector【默认】和 multiplexing channel selector
 - replicating 会将 source 过来的 events 发往所有的 channel，而 multiplexing 可以选择发往哪些 channel

Kafka

Kafka 是一个分布式的基于发布/订阅模式的消息队列（Message Queue），主要应用于大数据实时处理领域

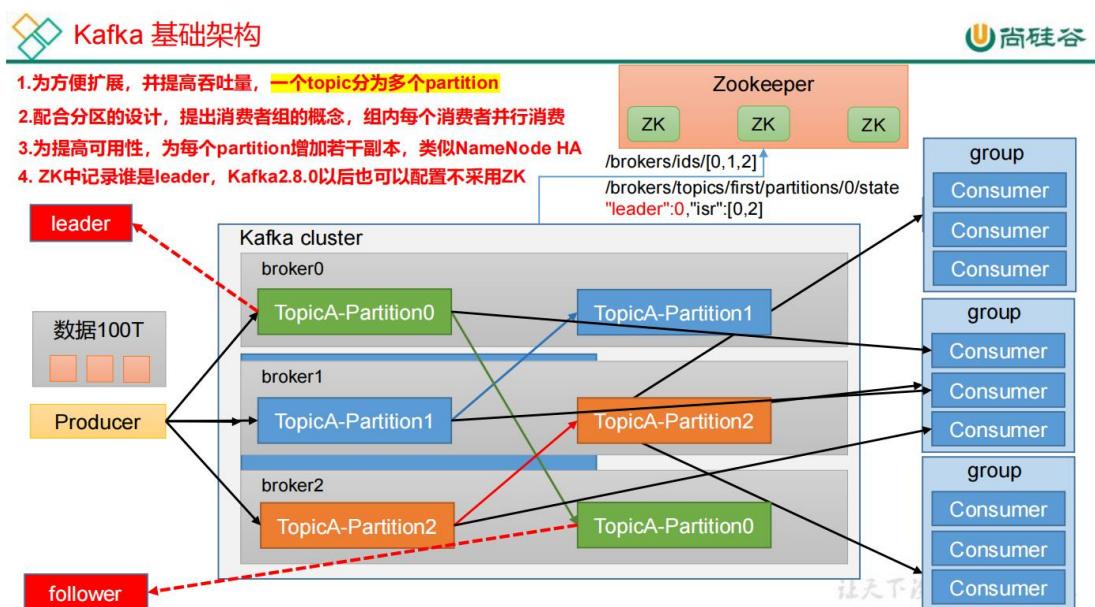
消息队列分为点对点模式（消费者主动拉取数据，消息收到后清除消息）和发布订阅模式（消费者消费数据之后，不删除数据）

kafka-kraft 模式抛弃了 zookeeper

1. 为什么要使用 kafka **

- 缓冲/消峰：**（解决生产消息和消费消息的处理速度不一致）当生产消息的速度过快时，kafka 在中间可以起到一个缓冲的作用，把消息暂存在 kafka 中，消费者就可以按照自己的节奏来处理。
- 解耦：**（接入不同的数据源，发往不同的目的地）消息队列可以作为一个接口层，允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束
- 异步通信：**（允许用户把一个消息放入队列，但并不立即处理它，然后在需要的时候再去处理它们）很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们

2. 简述 kafka 的架构 ***



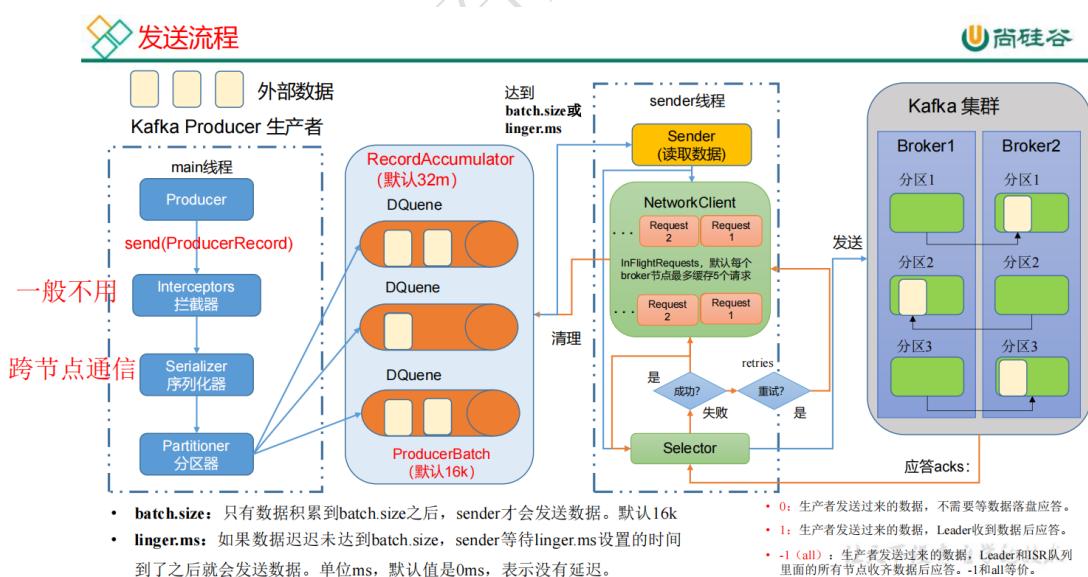
- 一个 kafka 集群由多个 broker 组成，一个 broker 可以有多个 topic，一个 topic

可以分为多个 partition，每个 partition 可以有若干个副本（一个 leader，若干 follower）

3. 命令行操作（了解）

- 启动集群: bin/kafka-server-start.sh -daemon config/server.properties
- 关闭集群: bin/kafka-server-stop.sh stop
- 查看所有 topic: bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --list
- 创建 topic: bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --create --replication-factor 3 --partitions 1 --topic first
- 删除 topic: bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --delete --topic first
- 发送消息: bin/kafka-console-producer.sh --bootstrap-server hadoop102:9092 --topic first
- 消费消息: bin/kafka-console-consumer.sh --bootstrap-server hadoop102:9092 --from-beginning --topic first

4. 生产者发送流程 *



- 在消息发送的过程中，涉及到了两个线程—main 线程和 Sender 线程。在 main 线程中创建了一个双端队列 RecordAccumulator。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka Broker。

- 调优---生产者如何提高吞吐量：

- batch.size: 批次大小， 默认 16k
- linger.ms: 等待时间， 修改为 5-100ms
- compression.type: 压缩 snappy
- RecordAccumulator: 缓冲区大小， 修改为 64m

5. 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 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

6. 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 $\geq 2 + \text{ISR 最小副本数} \geq 2$ ）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 $\langle \text{PID}, \text{partition}, \text{SeqNum} \rangle$ 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）

7. 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。

8. zookeeper 在 kafka 中的作用有哪些

- 记录有哪些服务器 /brokers/ids
- 记录谁是 leader，有哪些服务器可用 {"leader":0, "isr":[1,0,2]}
- 选举 controller，Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。{"brokerid": 0}
- 记录消费数据的 offset，在消费者消费数据的时候，需要定时的将分区消息的消费进度 offset 记录到 zookeeper 中（0.9 版本之前）【之所以后面将 offset 提交到系统主题中，是因为存放在 zookeeper 中需要进行频繁的通信】

9. broker 工作流程

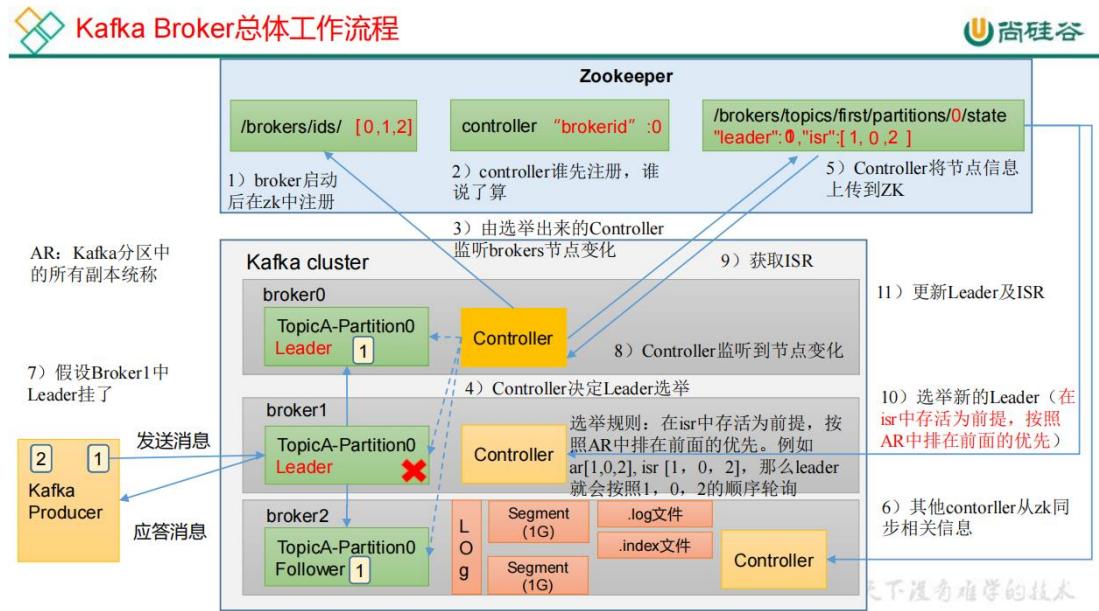
Kafka 分区中的所有副本统称为 AR (Assigned Replicas) $AR = ISR + OSR$

ISR：表示和 Leader 保持同步的 Follower 集合。如果 Follower 长时间未向 Leader 发

送通信请求或同步数据，则该 Follower 将被踢出 ISR，默认 30s

OSR：从 ISR 踢出的 follower

Kafka 集群中有一个 broker 的 Controller 会被选举为 Controller Leader，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 Leader 选举等工作；Controller 的信息同步工作是依赖于 Zookeeper 的



- 熟悉一下 leader 选举的流程
- leader 挂了或者 follower 挂了怎么办？

LEO (Log End Offset)：每个副本的最后一个 offset，LEO 其实就是最新的 offset + 1。

HW (High Watermark)：所有副本中最小的 LEO。



LEO (Log End Offset) : 每个副本的最后一个offset, LEO其实就是最新的offset + 1

HW (High Watermark) : 所有副本中最小的LEO



10. 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

11. kafka 的数据是放在磁盘上还是内存上, 为什么速度会快 **

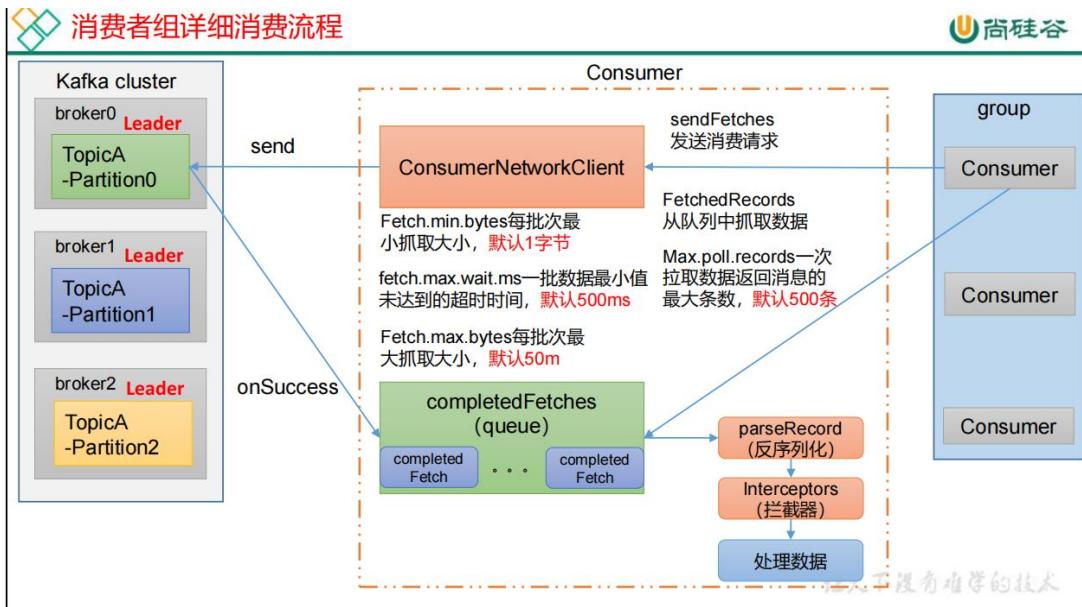
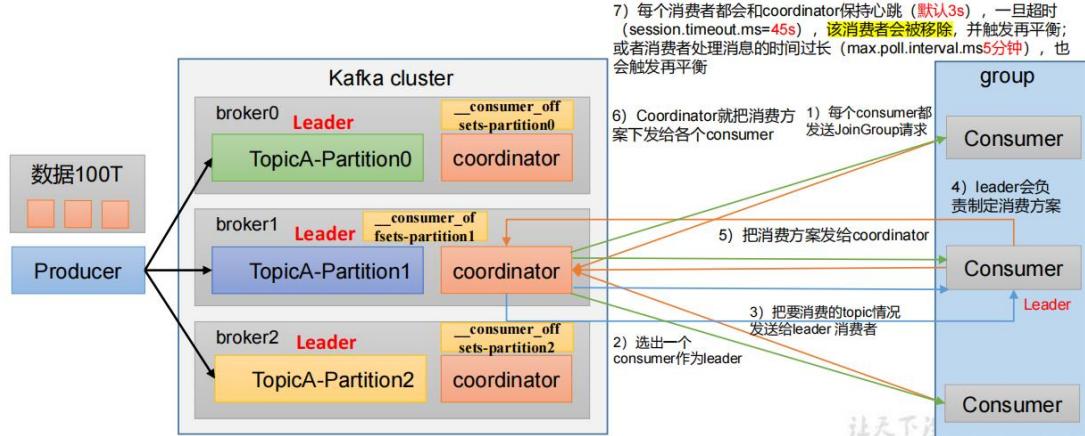
- 上面说到了放在本地 log 文件中, 所以是放在磁盘上
- 速度快有四个原因: (Kafka 每秒可以处理一百万条以上消息, 吞吐量达到每秒百万级。那么 Kafka 为什么那么高的吞吐量呢?)

- 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

12. 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）

- 1、coordinator: 辅助实现消费者组的初始化和分区的分配。
 coordinator节点选择 = `groupid的hashcode值 % 50` (`_consumer_offsets`的分区数量)
 例如: `groupid的hashcode值 = 1, 1 % 50 = 1`, 那么`_consumer_offsets`主题的1号分区, 在哪个broker上, 就选择这个节点的coordinator作为这个消费者组的老大。消费者组下的所有的消费者提交offset的时候就往这个分区去提交offset。



13 kafka 消息数据积压，消费者如何提高吞吐量

- 如果是 Kafka 消费能力不足，则可以考虑增加 Topic 的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）
- 如果是下游的数据处理不及时，提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。
 - `fetch.max.bytes=50m`
 - `max.poll.records=500 条`

14. 你知道 Kafka 单条日志传输大小吗

- 默认认为单条消息最大值是 1M，但是在我们应用场景中，常常会出现一条消息大于 1M 的情况，如果不对 kafka 进行配置，则会出现生产者无法将消息推送到 kafka 中或消费者消费不到 kafka 里面的数据的情况。我们可以配置两个参数：一个是副本的最大值，一个是单条消息的最大值，来解决消息最大限制的问题！
 - broker 端接收每个批次消息最大值： message.max.bytes=1m
 - 生产者发往 broker 每个请求消息最大值(针对 topic)：max.request.size=1m
 - 副本同步数据每个批次消息最大值： replica.fetch.max.bytes=1m
- 需要注意一点：副本的最大值一定要大于单个消息的最大值，否则就会导致数据同步失败

15. Kafka 为什么同一个消费者组的消费者不能消费相同的分区 *

- 因为这样可能会消费到重复的消息，因为 kafka 的 log 文件对应的数据都会存储自己的偏移量，而它是按照消费者组，主题，分区来进行区分的，那么同一个消费者组中的消费者使用的就是同一份偏移量，这样就很容易消费到重复的消息。

HBase

HBase 是一种分布式、可扩展、支持海量数据存储的非关系型数据库

HBase 的管理页面：host:16010

1. 简述 HBase 的数据模型 *

- HBase 的数据模型同关系型数据库很类似，数据存储在一张表中，有行有列
- 其中有一些专有名词：
 - namespace：命名空间，类似于关系型数据库中的 database，每个命名空间下有多个表。HBase 默认有两个命名空间，分别叫做 default 和 hbase
 - region：类似于关系型数据库中的 table。但是 HBase 在定义表的时候，不需要定义具体的列，只需要定义列族就可以了
 - row：表示一行数据，每行数据有一个 rowkey 和多个列组成，数据是按

照 rowkey 的字典顺序排列的

- column 列：每一列都有列族和列限定符组成
- timestamp：用于标识数据的不同版本，每条数据写入的时候，如果不指定时间戳，默认为写入 HBase 的时间
- cell 最小单元：由 {rowkey, column, timestamp} 唯一确定

2. 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

3. HBase 的基本架构

- HBase 主要包括 region server 和 master, region server 主要用于 region 的管理，而 master 主要用于管理 region server，另外还有 zookeeper 和 hdfs，zookeeper 主要是用来保证 master 的高可用，hdfs 提供存储服务。

4. 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

5. 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

6. 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 HFile 合并成一个大 HFile
 - ◆ 会清理掉过期和删除的数据

7. 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 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。

8. 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

9. 列族的设计规则 **

- 官网上建议：一张表应该有 1 到 3 个列族，所以列族的数量不应过多
- 原因：在向 hbase 写入数据的时候，我们会先写入到 memstore 中，每一个列族会对应一个 memstore 文件，如果列族数量很多，就会对应很多个 memstore 文件，当某个 memstore 中的文件满足了一定条件后就会进行 flush 操作，这个 flush 是 region 级别的，也就是只要有一个 memstore 要进行刷写，那么其他的 memstore 也要刷写，要是一个列族 100 万行，一个列族 10 行，这样就会产生大量的小文件并且大量的 IO 操作；另外一种场景：比如有些列族有 100W 行，而有些列族只有 10 行，这样在 Region Split 的时候会导致原本数据量很小的 Hfile 文件进一步被拆分，从而产生更多的小文件，也会影响到查询的效率
- 建议：在设置列族之前，我们最好想想，有没有必要将不同的列放到不同的列族里面。如果没有必要最好是放到同一个列族中。如果真要设置多个列族，但是其中的一些列族相对于其他列族数量差距非常悬殊，比如 1000W 相比 100 行，是不是考虑用另外一张表来存储相对小的列族。

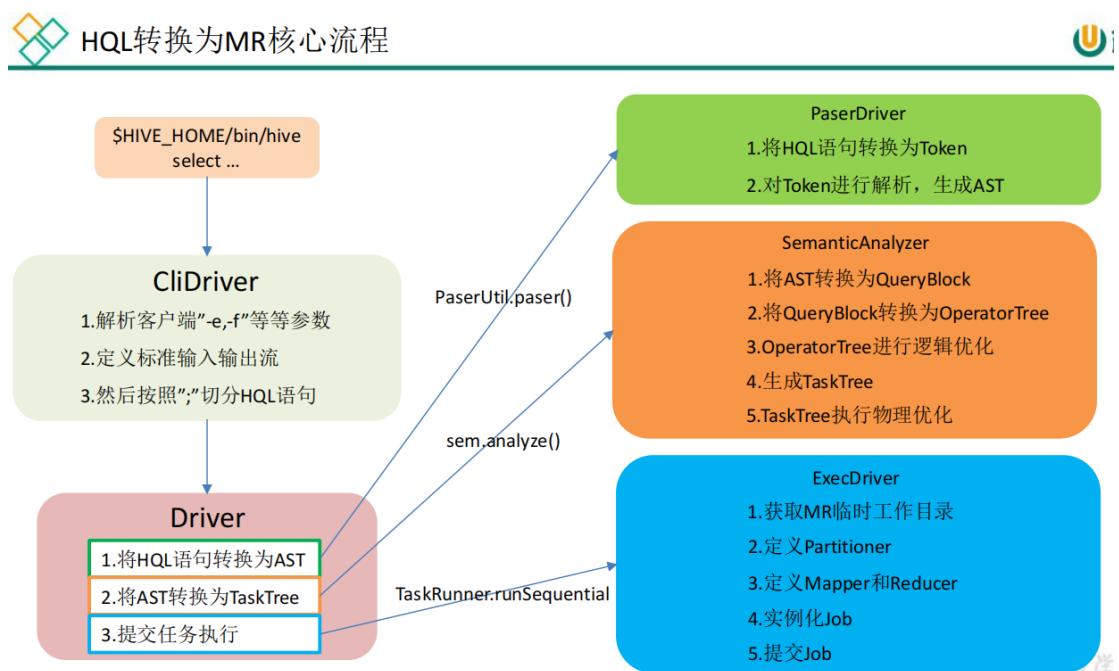
Hive

1. 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

2. 简述 hive 读写文件机制

- 读取文件：
 - 首先调用 InputFormat (默认 TextInputFormat) 对文件进行逻辑切片，返回一条一条的 kv 键值对，然后调用 SerDe (LazySimpleSerDe) 的反序列化方法，将一条记录中的 value 根据分隔符切分为各个对应的字段。

- 写文件：
 - 首先调用 SerDe（默认 LazySimpleSerDe）的序列化方法将对象序列化为字节序列，然后调用 OutputFormat 将数据写入 HDFS 文件中。

3. 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

4. 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人使用, 以免删除了, 还可以找到数据, 保证了数据安全

5. hive 静态分区和动态分区的区别

分区表，也叫分区裁剪，就是分目录，作用就是减少全表扫描

建表的时候: `partitioned by (day string)` 加载数据的时候: `partition (day="20210823")` 【静态分区】或者 `partition (day)` 【动态分区】

分区字段不能是表中已经存在的字段

- 静态分区：
 - 分区字段的值是在导入数据的时候手动指定的

- 导入数据的方式可以是 load data 方式，也可以是 insert into + select 方式
- 动态分区：
 - 分区字段的值是基于查询结果自动推断出来的，也就是最后查询结果的最后一个字段值就对应分区字段的值
 - 导入数据的方式必须是 insert into + select 方式
 - 想使用动态分区表的时候必须要对 hive 进行两个配置
 - ◆ 第一，开启动态分区功能 `hive.exec.dynamic.partition=true`
 - ◆ 第二，设置动态分区的模式为非严格模式，也就是说允许所有分区字段都可以使用动态分区 `hive.exec.dynamic.partition.mode=nonstrict`
- 补充题：你知道分桶表吗，谈谈这两个的区别？
 - 分桶表和分区表的作用都是用来减少全表扫描的，那么都有了分区表，为啥还要有分桶表呢？
 - 因为并非所有的数据都可以进行合理的分区，所以有了新的技术分桶表
 - ◆ 分桶表的分桶规则是，根据分桶字段的 hash 值，对桶的个数进行取余运算，然后得到该数据应该放到哪个桶里面去
 - 说了这么多，他们有什么区别呢？
 - ◆ 第一点，创建语句不同，分区表是 `partitioned by`，分桶表是 `clustered by`
 - ◆ 第二点，分区或分桶字段要求不同，分区字段不能是表中存在的字段，分桶字段一定是表中存在的字段
 - ◆ 第三点，表现形式不同，分区表是其实就是分了目录存放数据，分桶表是将一个文件拆分为很多文件存放

6. 内连接、左外连接、右外连接的区别

- 内连接：返回的是两个表的交集
- 左外连接：返回左表的所有行，如果左表的某行在右表没有匹配行，则将右表返回空值
- 右外连接：返回右表的所有行，如果右表的某行在左表没有匹配行，则将左表返回空值

7. 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

8. 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

9. 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

10. grouping_sets、cube 和 rollup

他们都是用于 group by 后面的一个函数，作用是将不同维度的 group by 进行简化

- grouping_sets(字段 1, 字段 2)会对字段 1 和字段 2 分别分组聚合, 然后 UNION ALL
- cube(字段 1, 字段 2)会对字段 1 和字段 2 的所有组合 2 的 n 次方种分别分组聚合, 然后 UNION ALL with cube
- rollup 是 cube 的一个子集, rollup 会以最左侧的维度为主 with rollup

11. 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"

12. hive3 的新特性有了解过吗

● 物化视图:

- 简述: 和普通视图的不同点在于普通视图不保存数据, 仅仅保存查询语句, 而物化视图是把查询的结果存入到了磁盘中, 它的作用是通过预计算保存好一些复杂的计算结果, 提高查询效率

- ◆ 语法：create materialized view ... as select ...
- 在向动态分区表中导入数据的时候，也可以使用 load 文件的方式，因为底层会自动转换为 insert+select 语句

13. 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 rcfile 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

14. 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 Parquet 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

15. 常用函数的补充

- NVL(value, default): 如果 value 为 null, 就返回 default, 否则返回 value
- IF(expr, value1, value2): 如果 expr 为 true, 返回 value1, 否则返回 value2
- concat_ws(seperator,str1,str2,...): 参数可以是字符串, 也可以是数组
- substring(value,start,len): 字符串索引的是从 1 开始, 我们要截取 value 中第 start 字符开始 len 长度的字符串
- 日期函数:

SQL

```
select to_date('2021-12-22 10:03:01') from test; -- 2021-12-22
select year('2021-12-22 10:03:01') from test; -- 2021
select month('2021-12-22 10:03:01') from test; -- 12
select day('2021-12-22 10:03:01') from test; -- 22
select hour('2021-12-22 10:03:01') from test; -- 10
select minute('2021-12-22 10:03:01') from test; -- 3
select second('2021-12-22 10:03:01') from test; -- 1
select datediff('2012-12-16','2012-12-12') from test; -- 4
select date_add('2012-12-08',10) from test; -- 2012-12-18
select date_sub('2012-12-08',6) from test; -- 2012-12-02
select date_format('2020-03-10','yyyy-MM'); -- 2020-03 [yyyy-MM-dd HH:mm:ss]
select next_day('2020-03-12','MO'); -- 2020-03-16 当前天的下一个周一
select date_add(next_day('2020-03-12','MO'),-7); -- 当前周的周一
select last_day('2020-03-10'); -- 当月最后一天日期
```

- 行列转换:

SQL

```
# max 一定是需要的
select user_name,
       MAX(CASE course WHEN '数学' THEN score else 0 END) 数学,
       MAX(CASE course WHEN '语文' THEN score else 0 END) 语文,
       MAX(CASE course WHEN '英语' THEN score else 0 END) 英语
FROM test
GROUP BY user_name
```

Spark

spark core

1. 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。（假设有两个转换操作，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。

2. 谈谈你对 RDD 的理解

- 它翻译过来就叫做弹性分布式数据集，是一种数据结构，可以理解成是一个集合。在代码中的话，RDD 是一个抽象类。还有一个非常重要的特点：RDD 是不保存数据的，仅仅封装了计算逻辑，也就是你直接打印 RDD 是看不见具体值的。

3. 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： $10 \text{ (Map Task)} * 2 \text{ (Reduce Task)}$

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

问 SortShuffle 产生的文件个数： $2 \text{ (Executer)} * (1 \text{ (合并的文件)}) + 1 \text{ (索引)}$

- 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)

4. 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。

5. spark driver 的作用，以及 client 模式和 cluster 模式的区别 *

- driver 主要负责管理整个集群的作业任务调度；executor 是一个 JVM 进程，专门用于计算的节点
- client 模式下，driver 运行在客户端；cluster 模式下，driver 运行在 yarn 集群

6. 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

7. 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】

8. 简述 map 和 mapPartitions 的区别 *

- 他们之间主要有三个区别
 - 第一个是, map 算子是串行操作, mapPartitions 算子是以分区为单位的批处理操作
 - 第二个是, map 算子主要是对数据进行转换和改变, 但是数量不能增加或者减少, 而 mapPartitions 算子可以增加或者减少数据
 - 第三个是, map 算子性能比 mapPartitions 算子要低, 但是 mapPartitions 算子会长时间占用内存, 可能会导致内存溢出的情况, 所以如果内存不是太多, 不推荐使用, 一般还是使用 map 算子

9. 你知道重分区的相关算子吗 *

- 我知道两个重分区的算子: coalesce 算子和 repartition 算子, 他们都是用来改变 RDD 的分区数量的, 而且 repartition 算子底层调用的就是 coalesce 方法
- 他们之间还有两个区别
 - coalesce 一般用来减少分区, 如果要增加分区必须设置 shuffle 参数=true, repartition 既可以增加分区也可以减少分区
 - coalesce 根据传入的参数来决定是否 shuffle, 设置为 false 容易发生数据倾斜, 但是 repartition 一定发生 shuffle, 也就是 shuffle 参数不可修改一定为 true
- 还有一个 partitionBy 算子也是进行重分区的, 参数传入一个分区器, 默认是 HashPartitioner

10. spark 目前支持哪几种分区策略

- spark 支持 Hash 分区, Range 分区和自定义分区
 - hash 分区就是 计算 key 的 hashCode, 然后和分区个数取余就可以得到对应的分区号
 - range 分区就是 将一定范围内的数据映射到一个分区中, 尽量保证每个分区数据均匀, 而且分区间有序

- 自定义分区就是 继承 Partitioner 类，重写 numPartitions 方法和 getPartition 方法

11. 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

12. 简述 reduceByKey、foldByKey、aggregateByKey、combineByKey 的区别 *

- reduceByKey：没有初始值，分区内和分区间计算规则一样
- foldByKey：有初始值，分区内和分区间计算规则一样
- aggregateByKey：有初始值，分区内和分区间计算规则可以不一样
- combineByKey：没有初始值，分区内和分区间计算规则可以不一样，同时返回值类型可以与输入类型不一致，因此通常在数据结构不满足要求的时候，才会使用该聚合函数

13. 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）

14. 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，那么就需要重新开始再执行一遍，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

15. 简述 spark 的容错机制

- 容错机制主要包括四个方面
 - 第一，stage 输出失败的时候，上层调度器 DAGScheduler 会进行重试
 - 第二，计算过程中，某个 task 失败，底层调度器会进行重试
 - 第三，计算过程中，如果部分计算结果丢失，可以根据窄依赖和宽依赖的血统重新恢复计算
 - 第四，如果血统非常长，可以对中间结果做检查点，写入磁盘中，如果后续计算结果丢失，那么就从检查点的 RDD 开始重新计算。

16. 除了 RDD，你还了解 spark 的其他数据结构吗 **

算子以外的代码都是在 Driver 端执行，算子里面的代码都是在 Executor 端执行

- 我还了解 累加器和广播变量 两种数据结构
 - 累加器就是分布式共享只写变量，简单说一下它的原理：累加器用来把 Executor 端变量信息聚合到 Driver 端。在 Driver 程序中定义的变量，在 Executor 端的每个 Task 都会得到这个变量的一份新的副本，每个

task 更新这些副本的值后，传回 Driver 端进行合并

- 广播变量就是分布式共享只读变量，简单说一下它的原理：用来高效分发较大的对象。向所有工作节点发送一个较大的只读值，以供一个或多个 Spark 操作使用
 - ◆ 补充：由 BlockManager 管理，广播变量会带来 Driver 的内存膨胀，因为广播变量会在内存中分割，分割完成之前，内存中有两份大小相同的广播变量

17. 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

spark sql

Spark 用于结构化数据(structured data)处理的 Spark 模块

18. 谈一谈 RDD, DataFrame, DataSet 的区别 *

共性: 他们都是分布式弹性数据集; 都有惰性机制

- 他们之间主要有四个区别

- 第一是，spark1.0 产生了 RDD，spark1.3 产生了 DataFrame，spark1.6 产生了 DataSet
- 第二是，RDD 不支持 sparksql 操作，但是 DataFrame 和 DataSet 支持 sparksql 操作
- 第三是，RDD 一般会和 spark mllib 同时使用，但是 DataFrame 和 DataSet 一般不和 spark mllib 同时使用
- 第四是，DataFrame 是 DataSet 的一个特例；DataFrame 的每一行的类型都是 Row，但是 Dataset 的每一行的数据类型不相同

19. Hive on Spark 与 SparkSql 的区别 *

- 只是 SQL 引擎不同，但是计算引擎都是 spark

20. 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 适合一张较小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

spark streaming

21. 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

22. 你知道 sparkstreaming 的背压机制吗 *

- 背压机制就是根据 JobScheduler 反馈作业的执行信息来动态调整 receiver 的数据接收率
- 首先说一下为什么会出现这个东西？
 - 在 spark1.5 之前，是没有背压机制这个东西的，那么就会引起一个问题，如果 producer 太快了怎么办？spark 还是提供了一个参数用来限制 receiver 的数据接收速率，如果配低一点可能会好一点，但是如果集群本身的处理能力比这个接收速率高很多，那么就会导致集群的资源利用率低。
- 可以通过配置一些参数开启背压机制和设置 receiver 的最大接收速率：
 - spark.streaming.receiver.maxRate: receiver 的最大接收速率
 - spark.streaming.backpressure.enabled 默认为 false，不启动背压机制

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

- ReceiverAPI: 需要一个专门的 Executor 去接收数据，然后发送给其他的 Executor 做计算。如果他们接收数据的 Executor 速度大于计算的 Executor 速度，就会内存溢出
- DirectAPI: 是由计算的 Executor 来主动消费 Kafka 的数据，速度由自身控

制。

24. 说一下你知道的 DStream 转换和输出原语 *

- 转换：
 - 无状态的转换
 - ◆ 将转换操作分别作用到每个 RDD 上。
 - ◆ 包含 RDD 的大多数转换算子，比如 map, flatmap, filter, reduceByKey 等
 - ◆ 这其中包括一个非常特殊的方法： transform
 - 可以调用 DStream 中没有的但是 RDD 转换算子中存在的 API
 - 有状态的转换
 - ◆ UpdateStateByKey
 - 可以记录历史状态，举一个例子，在求 wordcount 的时候，如果利用无状态的转换操作，只能求每个时间间隔内的 wordcount，如果想求历史间隔和当前间隔一起的 wordcount 怎么办呢？于是出现了有状态的转换操作，它可以记录历史的 wordcount 统计结果。
 - 使用它有一个前提，需要配置检查点目录，状态保存在检查点中
 - ◆ 窗口操作
 - 前提：设置窗口大小和窗口的间隔（这两者必须为采集周期大小的整数倍）
 - 可以实现任意大小窗口的动态变化的统计
 - API:
 - window
 - reduceByWindow 和 reduceByKeyAndWindow
 - countByWindow 和 countByValueAndWindow
- 输出：
 - print(): 打印 DStream 中每一批次数据的最开始 10 个元素
 - foreachRDD(func): 这是最通用的输出操作，即将函数 func 用于产生于 stream 的每一个 RDD。在 foreachRDD() 中，可以重用我们在 Spark 中

实现的所有行动操作。比如，常见的用例之一是把数据写到诸如 MySQL 的外部数据库中。

25. 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。

Flink

1. 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

2. 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。

3. Flink 的重启策略你了解吗

- 重启策略就是说 job 失败之后如何重启，Flink 支持不同的重启策略，包括

固定延迟重启策略、失败率重启策略、无重启策略

- 我是在使用检查点的时候，遇到过 Flink 重启的问题，我设置了检查点，但是实际上有一个地方会抛出异常，但是程序始终不抛出异常，并且一直输出异常之前的输出语句，这个时候就是因为，Flink 在开启 checkpoint 的情况下，重启策略会自动进行重启

4. Flink 的运行依赖于 hadoop 组件吗

- Flink 可以完全独立于 Hadoop，在不依赖 Hadoop 组件下运行。但是在实际的大数据应用场景下，Hadoop 体系是任何大数据框架都绕不过去的。Flink 可以和 Yarn 集成做资源调度，也可以读写 HDFS，或者利用 HDFS 做检查点
- 总结一句话：可以但是没必要

5. Flink 集群有哪些角色？各自有什么作用 *

slot：每一个 taskmanager 都包含一定数量的 slot，指 TaskManager 具有的 并行执行能力（静态概念）

并行度：taskmanager 运行程序的时候实际使用的并行能力（动态概念）

- jobmanager
 - 相当于一个集群的 Master，是整个集群的协调者，负责接收 job
- taskmanager
 - 实际负责执行计算的 Worker
- client
 - 它是 Flink 程序提交的客户端，当用户提交一个 Flink 程序时，会首先创建一个 Client

6. 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

7. max 算子和 maxBy 算子的区别 *

补充：分流算子 Split：根据某些特征把一个 DataStream 拆分成两个 DataStream；select

算子：从 SplitStream 中获取对应的 DataStream

- 这两个算子都是基于 KeyedStream 求最大值
- 不同点在于：
 - max：将原来的数据的该字段替换为最大值，然后返回该记录，返回的数据不是原来的数据
 - maxBy：会把该字段最大的整条记录全部返回，返回的数据还是原来的数据

8. 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

9. 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

10. 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

11. 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

12. 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

13. Flink 是如何做容错的？

- Flink 实现容错主要靠强大的 CheckPoint 机制和 State 机制。Checkpoint 负责定时制作分布式快照、对程序中的状态进行备份；State 用来存储计算过程中的中间状态
- state 和 checkpoint 之间的区别：
 - state 存储的是某一个操作的运行的状态或者历史值，维护在内存中
 - checkpoint 存储的是某一时刻所有操作的当前状态的快照，存在磁盘中

14. 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

15. Flink 是如何处理反压的

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

16. Flink 是如何支持批流一体的 *

- Flink 使用一个引擎就支持了 DataSet API 和 DataStream API。其中 DataSet API 用来处理有界流，DataStream API 既可以处理有界流又可以处理无界流，这样就实现了流批一体

17. 你用过 Flink CEP 吗，简单介绍一下 *

- 说一下我的理解，CEP 就是用来从无界流中得到满足一定规则的复杂事件
- 我之前在做用户行为分析的时候，做过一个需求：如果有用户 5 秒内连续登陆 3 次，就报警输出该用户
 - 使用 cep 编程特别的简单，首先定义一个匹配模式（begin where next where within），调用 Pattern 中的方法，然后将匹配模式应用到数据流上，调用 CEP.pattern 方法，最后检测出符合匹配条件的复杂事件，进行转换处理，输出报警信息

第二部分 Java 基础

Java 基础

1. JDK、JRE、JVM 三者区别和联系 *

- 分别解释（区别）：
 - jdk：全称是 java development kit，java 开发工具包，包含了 jre 和一堆开发工具，比如 javac/java 等。
 - jre：全称是 java runtime environment，java 运行环境，包含 jvm 和 java

核心类库。

- jvm: 全称是 java virtual machine, java 虚拟机, 把字节码文件解释成具体平台的机器指令执行。
- 联系:
 - JVM 不能单独解释字节码文件, 解释字节码文件的时候需要调用解释所需要的类库 lib。在 JDK 下面的 jre 目录里面有两个文件夹 bin 和 lib, 在这里可以认为 bin 里的就是 jvm, lib 中则是 jvm 工作所需要的类库, 而 jvm 和 lib 合起来就称为 jre。

2. 基本数据类型和引用数据类型的区别 *

1. 存储位置的不同

- 对于在方法中声明的变量, 如果是基本数据类型, 那么变量名和值都是存放在栈中, 如果是引用数据类型, 变量名是存放在栈中, 而指向的对象是存放在堆中

2. 传递方式的不同

- 调用方法时, 如果传递的参数是基本数据类型, 那么就是按数值传递, 如果传递的参数是引用数据类型, 那么就是按引用传递, 但是也不是说引用数据类型是引用传递, 在 java 中只有值传递

3. 8 种基本数据类型、字节大小 *

- byte: 占用 1 个字节, 取值范围-128 ~ 127
- short: 占用 2 个字节, 取值范围-2^15 ~ 2^15-1
- int: 占用 4 个字节, 取值范围-2^31 ~ 2^31-1
- long: 占用 8 个字节
- float: 占用 4 个字节
- double: 占用 8 个字节
- char: 占用 2 个字节
- boolean: 占用大小取决于 java 虚拟机

4. 访问修饰符权限 *

public > protected > default > private

- `private`：在同一类内可见。
- 默认：在同一包的类可见
- `protected`：对同一包的类和不同包的子类可见
- `public`：对所有类可见

5. java 中方法的参数传递机制 **

- 参数传递机制包括两种：值传递和引用传递
 - 值传递是指在调用函数时传递的是实际参数的值，那么在函数中对它进行修改，不会影响到实际参数
 - 引用传递是指在调用函数时传递的是实际参数的地址，那么在函数中对它进行修改，会影响到实际参数
- 但是在 java 中只有值传递参数
 - 为什么这样说呢？都知道数据类型分为两大类，基本数据类型和引用数据类型
 - ◆ 如果参数是基本数据类型，那么传过来的就是这个参数的一个副本，如果在函数中改变了副本的值明显不会改变原始的值
 - ◆ 如果参数是引用数据类型，那么传过来的就是这个引用参数的一个副本，这个副本存放的是参数的地址。在函数中仅仅是改变了地址中的值，那么原始的值会改变，但是地址仍然没有改变

6. final 关键字 **

- `final` 有三种用法：修饰类、变量和方法。
 - 修饰类的时候，表示它不可以被继承；
 - 修饰变量的时候，表示它不能被修改（对于基本数据类型，就是值不能被修改，对于引用数据类型，就是引用不能被修改）；
 - 修饰方法的时候，表示它不能在子类中被重写
- 补充题：`final`、`finally`、`finalize` 的区别是什么？
 - `finally` 作为异常处理的一部分，通常放在 `try...catch` 的后面，附带一个语句块用来表示这个语句最终一定被执行，可以将释放外部资源的代码写在 `finally` 块中
 - `finalize` 是 `Object` 类中的一个方法，在垃圾回收器准备释放对象占用的内

存时，首先会调用对象的 finalize() 方法，做一些清理的工作

7. static 关键字的作用是什么 *

- static 有四种用法：修饰成员变量、成员方法、代码块和内部类。
 - 修饰成员变量的时候，就是将其变为类的成员，可以用类.变量的方式使用，如果有数据需要被共享给所有对象使用时，那么就可以使用 static 修饰。
 - 修饰成员方法的时候，就是将其变为类的方法，可以用类.方法的方式调用，static 方法中不能使用 this 和 super 关键字，只能访问本类中的静态变量和静态方法。
 - 修饰代码块的时候，在类被加载的时候，就会被执行，并且只会执行一次。
 - 修饰内部类的时候，在静态内部类中，无法直接访问外部类的非静态变量和非静态方法，如果要访问的话，必须 new 一个外部类对象，使用 new 出来的对象来访问，但是可以直接访问外部类的静态变量和静态方法。

8. Comparable 和 Comparator 区别 *

- 思想：Comparable 是内部比较器，而 Comparator 是外部比较器；解释一下的话，如果一个类本身实现了 Comparable 接口（只有一个方法 compareTo），就说明它本身支持排序，使用的时候调用 Collections.sort 或者 Arrays.sort；如果本身没有实现 Comparable 接口，那么可以通过外部比较器 Comparator（比如实现 compare 方法）来进行排序，使用的时候也是一样，但是要传入这个外部比较器对象
- 使用场景：实现 Comparable 接口的方式比实现 Comparator 接口的耦合度要高一些。（如果要修改比较算法，那么 comparable 还需要在实现类中修改，而 comparator 不需要在实现类中修改）

9. Object 类有哪些方法 *

- equals 方法：比较两个对象是否相等
- hashCode 方法：返回对象的哈希码值
- toString 方法：返回对象的字符串表示
- clone 方法：实现对象的浅拷贝
- finalize 方法：用于释放资源

- `getClass` 方法：获取当前对象所属的字节码文件对象

10. 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；

11. 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。

12. 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

13. 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

14. 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

15. 抽象类和接口的区别 *

- 区别
 - 抽象类既可以定义抽象方法，也可以定义普通成员方法；而接口中只能定义抽象方法
 - 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 常量类型的
 - 一个类只能继承一个抽象类，但可以实现多个接口（单继承，多实现）
- 使用场景：当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口

16. 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

17. 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

18. 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);

19. 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。

20. 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 $(n-1) \& \text{hash}$ 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

21. HashMap 中为啥用红黑树不用二叉排序树或者平衡树 *

1. 如果使用二叉排序树, 极端情况下会形成一条链, 那么失去了将链表转换为树结构的意义
2. 在进行插入删除时, 红黑树需要更少的旋转次数就可以达到自平衡, 因此效率更高, 但是在进行查找的时候, 由于平衡二叉树的高度严格平衡, 因此它的查找效率更高, 基于插入删除查找效率的一个综合考虑, hashmap 就采用了红黑树数据结构。

22. TreeMap 底层实现

- TreeMap 底层实现是红黑树，由于红黑树是一种特殊的二叉排序树，所以默认情况下会按照 key 进行升序排列；由于红黑树的插入、删除、遍历时间复杂度都为 $O(\log n)$ ，所以性能上会低于 HashMap
- 使用场景：如果对 map 进行插入、删除或查找的操作更频繁，HashMap 是更好的选择；如果需要对 map 按照 key 值进行有序的输出，TreeMap 是更好的选择。

23. HashMap 和 Hashtable 的区别 *

- HashMap 方法没有 synchronized 修饰，线程不安全，而 HashTable 线程安全
- HashMap 允许 key 和 value 为 null，而 HashTable 不允许
- HashMap 的初始容量为 16，而 HashTable 的初始容量为 11；HashMap 扩容时是当前容量翻倍，HashTable 扩容时是容量翻倍-1
- HashMap 计算对 key 进行一次 hash 得到了 hashCode，再次对 hashCode 进行二次 hash，然后对数组长度取模 ($h = key.hashCode() \wedge (h >> 16)$ ：防止失去高位特征)；HashTable 是计算一次 hash 然后取模。 $(key.hashCode() \& 0x7FFFFFFF$ ：防止出现负数)

24. Hashtable 怎么保证线程安全的 *

- Hashtable 底层在只要和数据交互的方法【比如 put 和 get】上面都加上了 synchronized 关键字，来保证线程安全的。当一个线程调用该方法时，就会拿到锁对象，当其他线程也来调用此方法时，会进入阻塞状态，直到当前线程执行完该方法才会释放这把锁。

25. 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。

26. 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()

27. 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。

28. 常见的 IO 模型 *

- BIO 是同步阻塞的 IO 模型，NIO 是同步非阻塞的 IO 模型，IO 多路复用是异步阻塞的 IO 模型，AIO 是异步非阻塞的 IO 模型
- BIO：它是一个阻塞式的 IO 模型，阻塞主要发生在服务端，第一是监听客户

端请求的时候是阻塞的，第二是读取客户端发送的请求信息是阻塞的（等客户端发来信息才能继续执行）

- NIO: 它是一个同步非阻塞的 IO 模型，它使用 channel 和 buffer 来传输数据，数据总是从缓冲区写入通道，并从通道读取到缓冲区；利用 selector 来监视多个通道的对象，如数据到达，连接打开等，因此单线程可以监视多个通道中的数据；当我们将 channel 注册到 selector 中的时候，会返回一个 selection key 对象（表示一个特定的通道对象和一个特定的选择器对象之间的注册管理），通过 selection key 我们可以知道哪些 IO 事件已经就绪了，并且可以通过其获取 channel 并对其进行操作
 - 过程：
 - ◆ 将 channel 注册到 selector 中
 - ◆ 调用 selector 的 select 方法，这个方法会阻塞
 - ◆ 到注册在 selector 中的某个 channel 有新的 tcp 连接或者可读写事件的话，这个 channel 就会处于就绪状态，会被 selector 轮询出来
 - ◆ 然后通过 selectionkey 可以获取就绪 channel 的集合，进行后续的 IO 操作
- NIO 相比 BIO 的好处：
 - 使用比较少的线程便可以管理多个客户端的连接，提高了并发量并且减少了资源消耗
 - 在没有 IO 操作相关的事情的时候，线程可以被安排在其他任务上面，以让线程资源得到充分利用
- AIO：异步 IO 是基于事件和回调机制实现的，也就是发起读请求调用之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作

29. 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

30. 一致性 hash 算法 **

- 我先说一下为什么会有一致性 hash 算法, 也就是说 hash 算法有什么缺点吗?
- 假设我们有三台缓存服务器, 现在需要将很多数据缓存到这三台服务器上, 这个是很容易办到的, 就是通过 $\text{hash}(\text{key}) \% n$ 就可以了, 那么当数据越来越多, 三台服务器已经满足不了需求了, 需要增多服务器的时候, 但是当服务器增多的时候, 所有缓存在一定时间内是失效的, 那么就会大量的请求数据源, 增加数据源的访问压力, 当然有服务器崩了也是一样
- 那么我在说一下一致性 hash 算法是什么呢, 它也是使用取模的方法, 但是是对 2^{32} 次方取模, 具体的步骤:
 - 首先将整个哈希值空间按照顺时针方向组织成一个虚拟的圆环, 称为 Hash 环
 - 接着将各个服务器使用 Hash 函数进行哈希, 具体可以选择服务器的 IP 或主机名作为关键字进行哈希, 从而确定每台机器在哈希环上的位置
 - 最后使用算法定位数据访问到相应服务器: 将数据 key 使用相同的函数 Hash 计算出哈希值, 并确定此数据在环上的位置, 从此位置沿环顺时针寻找, 第一台遇到的服务器就是其应该定位到的服务器
 - 那么这个时候有服务器添加或者崩了, 我们只需要重新定位环空间中的一小部分数据, 也就是只有部分数据失效, 不至于大量的请求落到数据源上, 具有较好的容错性

JVM

1. java 运行时一个类是什么时候加载的 *

- 一个类在什么时候开始被加载, 《java 虚拟机规范》中并没有进行强制约束, 交给了虚拟机厂商自己去自由实现, HotSpot 虚拟机是按需加载的, 在需要用到该类的时候加载这个类

2. 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 static 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 static 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

3. 继承时父子类的初始化顺序是怎样的 ***

1. 父类 -- 静态变量和静态代码块
2. 子类 -- 静态变量和静态代码块
3. 父类 -- 变量和普通代码块
4. 父类 -- 构造器
5. 子类 -- 变量和普通代码块
6. 子类 -- 构造器

4. 什么是类加载器

- 在类“加载”阶段，通过一个类的全限定名来获取描述该类的二进制字节流的这个动作的“代码”被称为“类加载器”，这个动作是可以自定义实现的

5. JVM 有哪些类加载器 *

- 站在 java 虚拟机的角度来看，只存在两种不同的类加载器

- 启动类加载器（BootstrapClassLoader），使用 c++ 语言实现，是虚拟机的一部分
- 其他所有的类加载器，由 java 语言实现，独立存在于虚拟机外部，并且全部都继承自抽象类 ClassLoader
- 站在 java 开发者的角度来看，自 JDK1.2 开始，java 一直保持着三层加载器架构
 - 启动类加载器（BootstrapClassLoader）：C++ 语言实现
 - ◆ 加载<JAVA_HOME>/jre/lib/rt.jar, resources.jar, charsets.jar （并不是所有 jar 包，plugin.jar 就不是它加载的） 【通过类名.class.getClassLoader() 就可以知道这个类是哪个加载器加载的】
 - ◆ 被-Xbootclasspath 参数所制定的路径中所有的类库
 - 扩展类加载器（ExtClassLoader）：java 语言实现
 - ◆ 加载<JAVA_HOME>/jre/lib/ext 中的 jar 包
 - ◆ 被 java.ext.dirs 系统变量所指定的路径中所有的类库
 - 应用程序类加载器（AppClassLoader）：java 语言实现
 - ◆ 加载用户类路径（classpath）下所有的类库

6. 什么是双亲委派模型 ***

- 如果一个类加载器收到了类加载的请求，它首先不会尝试加载这个类，而是把这个请求委派给上一层类加载器去完成，每一层的类加载器都是如此，因此最终会传送到最顶层的启动类加载器中，然后它会尝试加载这个类，只有当它无法完成这个加载请求的时候，下一层的类加载器才会去尝试加载，如果所有的类加载器都无法加载，就会抛出 ClassNotFoundException

7. JDK 为什么要设计双亲委派模型，有什么好处

- 确保安全，避免 java 核心类库被修改
- 避免重复加载
- 保证类的唯一性

8. 可以打破双亲委派模型吗？如何打破？

- 可以
- 想要打破这种模型，那么就自定义一个类加载器，覆盖其中的 loadClass 方法

法，使其不进行双亲委派即可

9. 如何自定义类加载器

1. 继承 ClassLoader
2. 覆盖 findClass 方法或者 loadClass 方法(如果覆盖 loadClass 会打破双亲委派)

10. ClassLoader 中的 loadClass()、findClass()、defineClass()

区别

- loadClass(): 主要进行类的加载，双亲委派机制就在这个方法中实现
- findClass(): 根据类全限定名去加载.class 字节码
- defineClass(): 把字节码转换为 java.lang.Class 对象

11. 加载一个类采用 Class.forName() 和 ClassLoader.loadClass()有什么区别

- Class.forName(): 会进行初始化
- ClassLoader.loadClass(): 不会进行初始化

12. Tomcat 的类加载机制

在 tomcat 6 之后已经合并到根目录下的 lib 目录下

- common 类加载器：加载 common 目录下的 jar 包
- catalina 类加载器：加载 server 目录下的 jar 包
- shared 类加载器：加载 shared 目录下的 jar 包
- WebApp 类加载器：加载 webapps 目录下的文件，每一个 web 应用程序会对应一个 webapp 类加载器，打破了双亲委派机制，即如果收到了类加载的请求，首先会尝试自己加载，如果找不到再交给上一层加载器去加载，目的就是为了优先加载 web 应用自己定义的类
- Jsp 类加载器：加载 jsp 文件，每一个 jsp 文件会对应一个 jsp 类加载器

13. 为什么 Tomcat 要破坏双亲委派模型

- Tomcat 是 web 容器，那么一个 web 容器可能需要部署多个应用程序
 - 部署在同一个 Tomcat 的两个应用程序所使用的 java 类库要相互隔离

- 部署在同一个 Tomcat 的两个应用程序所使用的 java 类库要相互共享
- 保证 Tomcat 服务器自身的安全不受部署的 web 应用程序影响
- 需要支持 jsp 页面的热部署和热加载

14. 热加载和热部署，如何自己实现一个热加载

- 热加载：是指可以在不重启服务的情况下让更改的代码生效，可以显著的提升开发以及调试的效率，它是基于 java 的类加载器实现的，但是由于热加载的不安全性，一般不会用于正式的生产环境
- 热部署：是指可以在不重启服务的情况下重新部署整个项目，比如 Tomcat 热部署就是在程序运行时，如果我们修改了 war 包中的内容，那么 Tomcat 就会删除之前的 war 包解压的文件夹，重新解压新的 war 包生成新的文件夹
- 总结：
 - 热加载就是在运行时重新加载 class，后台会启动一个线程不断检测你的 class 是否发生变化
 - 热部署是在运行时重新部署整个项目，耗时相对较高
- 如何实现热加载呢？
 1. 实现自己的类加载器
 2. 从自己的类加载中加载要热加载的类
 3. 不断监测要热加载的类 class 文件是否有更新，如果有更新，重新加载

15. java 代码到底是如何运行起来的 *

- 三种情况：
 - Hello.java -> javac -> Hello.class -> java Hello (jvm 进程，也就是一个 jvm 虚拟机)
 - Hello.java -> javac -> Hello.class -> Hello.jar -> java -jar Hello.jar
 - Hello.java -> javac -> Hello.class -> Hello.war -> Tomcat -> startup.sh -> org.apache.catalina.startup.Bootstrap (jvm 进程，也就是一个 jvm 虚拟机)
- 总结：其实运行起来一个 java 程序，都是通过启动一个 jvm 虚拟机，在虚拟机里面运行.class 字节码文件

16. 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

17. Java 对象如何在堆内存分配

- 指针碰撞：对象在内存中排列非常整齐的情况，一个指针指向一块区域，按顺序放满之后，指针向后移动
- 空闲列表：对象在内存中排列不整齐的情况，一个列表存储空闲的内存区域的地址

- 本地线程分配缓冲（TLAB）：
 - 问题：对象创建在虚拟机中频繁发生，仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针分配内存
 - 解决：
 - ◆ 线程隔离，每个线程在 java 堆中预先分配一小块内存，称为 TLAB，哪个线程要分配内存，就在哪个线程的本地缓存区中分配，只有本地缓冲区用完了，分配新的缓冲区时才需要同步锁定（JVM 是采用 CAS 保证更新操作的原子性）

18. JVM 堆内存中的对象布局 *

- 对象头：由 mark word(8B)和类型指针(8B)组成
 - mark word 用于存储自身的运行时数据，如哈希码、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
 - 类型指针，就是对象指向它的类型元数据的指针，JVM 通过这个指针确定该对象是哪个类的实例
 - 如果对象是一个 java 数组，那么在对象头中还需要有一块用于记录数组长度的数据
- 实例数据：类中定义的各种成员变量的内容（包括父类继承的和自己定义的）
- 对齐填充：主要起着占位符的作用，Hotspot 虚拟机规定任何对象的大小都是 8 字节的整数倍

19. JVM 什么情况下会发生堆内存溢出 **

- java 堆中用于存储对象，只要不断的创建对象，并且保持 GC Roots 到对象之间有可达路径来避免垃圾回收机制清理这些对象，那么随着对象数量的增加，总容量达到最大堆的容量限制后就会产生内存溢出
- 可视化工具 VisualVM
- 快照分析工具 MAT
 - 捕获快照 headdump.hprof
 - MAT 对快照进行分析

20. JVM 如何判断对象可以被回收 ***

- 在 java 堆里面存放着所有的对象，垃圾回收器在对对象进行回收前，首先要确定这些对象哪些还活着，哪些已经死去
 - 引用计数器：有一个地方引用，计数器就加 1，当引用失效，计数器就减 1，只有当计数器等于 0 的对象，才可以被回收
 - java 通过可达性分析算法来判定对象是否存活的
 - ◆ 原理：通过一系列称为 GC Roots 的根对象作为起始节点，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为引用链，如果某个对象到 GC Roots 间没有任何引用链相连，则证明此对象是不可能再被使用的对象，就可以被垃圾回收器回收
 - ◆ 哪些对象可以作为 GC Roots 呢？
 - 虚拟栈中的局部变量所引用的对象
 - 元空间中的类静态变量引用的对象
 - 元空间中的常量引用的对象
 - 本地方法栈中 native 方法引用的对象（了解）
 - JVM 内部的引用，如基本数据类型对象的 Class 对象（了解）
 - 所有被同步锁（synchronized）持有的对象（了解）

21. java 中不同的引用类型 *

- 强引用：Object obj = new Object()，永远不会回收
- 软引用：SoftReference 内存充足时不回收，内存不足时则回收
- 弱引用：WeakReference 不管内存是否充足，只要 GC 一运行就会回收该引用对象
- 虚引用：PhantomReference 就等同于没有引用，作用就是被 GC 时触发一个系统通知

22. JVM 堆中新生代的垃圾回收过程 **

- 创建的对象会首先分配到 eden，当 eden 区内存空间不足时，会触发 minor gc，就会把 eden 区存活的对象复制到 s1 区，其他对象全部清理掉，同时把 s0 区存活的对象复制到 s1 区，并且将他们的寿命-1，然后交换 s0 区和 s1 区，当对象的寿命达到默认值 15 时，就会晋升到老年代中

23. JVM 对象动态年龄判断是怎么回事 *

- 虚拟机并不是永远的要求对象的年龄必须达到了 15 才能晋升到老年代
- 动态年龄判断：幸存区的对象年龄从小到大进行累加，当累加到 X 年龄时的总和大于 50% (survivor 占比, -XX:TargetSurvivorRatio=50 来设置保留多少空闲空间)，那么比 X 大的都会晋升到老年代

24. 什么是老年代空间分配担保机制 *

- 作用：为了避免频繁进行 full gc
- 参数：-XX:HandlePromotionFailure
- 过程：准备 minor gc -> 判断老年代可用空间是否大于新生代所有对象总大小
 - 大于 -> minor gc
 - 小于 -> 判断老年代可用空间是否大于每一次 minor gc 进入老年代的对象平均大小
 - ◆ 大于 -> minor gc -> minor gc 后, s 区放不下, 老年区也放不下 -> full gc
 - ◆ 小于 -> full gc

25. 什么情况下对象会进入老年代 **

- 当对象的寿命达到 15 时，会进入老年代
- 对象动态年龄判断
- 老年代空间担保机制
- 大对象直接进入老年代 (大对象是指需要大量连续内存空间的 java 对象，比如很长的字符串或者是很大的数组或者 list 集合，因为在幸存区来复制，消耗性能！！)

26. JVM 本机直接内存的特点及作用 *

- 直接内存不属于 JVM 运行时数据区，是本机直接物理内存
- 在 jdk1.4 中新加入了 NIO 类，一种基于通道与缓冲区的 I/O 方式，它可以使用 native 函数库直接分配堆外内存，然后通过一个存储在堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中提

高性能，因为避免了在 java 堆和 native 堆中来回复制数据

- 本机直接内存的分配不会受到 java 堆大小的限制，但是既然是内存，则肯定还是会受到本机总内存大小的限制，可能会出现 OOM 的异常
- -XX:MaxDirectMemorySize=size 默认情况下，大小设置为 0，这意味着 JVM 自动为 NIO 直接缓冲区分配大小

27. 几个与 JVM 内存相关的核心参数 **

- -Xms: java 堆内存的初始大小
- -Xmx: java 堆内存的最大大小
- -Xmn: java 堆内存中的新生代大小
- -XX:MetaspaceSize: 元空间初始大小
- -XX:MaxMetaspaceSize: 元空间最大大小
- -Xss: 每个线程的虚拟机栈内存大小
- -XX:SurvivorRatio=8: 设置 eden 区和 survivor 区的大小的比例，默认是 8:1:1
- -XX:MaxTenuringThreshold=15: 年龄阈值
- -XX:-UserConcMarkSweepGC: 指定 CMS 垃圾回收器
- -XX:-UseG1GC: 指定 G1 垃圾回收器

28. 堆为什么要分为新生代和老年代 *

- 因为有的对象寿命长，有的对象寿命短。应该将寿命长的对象放在一个区，寿命短的对象放在另外一个区。不同的区应该采用不同的垃圾收集算法，同时寿命短的区清理频次高一点，寿命长的区清理频次低一点，所以就有了新生代和老年代

29. 新生代为什么要有两个 survivor 区 **

- 如果没有 survivor 区，那么 eden 每次满了，就会触发 minor gc，存活的对象被放到老年区，老年区满了，就会触发 Full GC，Full GC 是非常耗时的，所以必须有 survivor 区
- 如果只有一个 survivor 区，一旦 Eden 满了，触发一次 Minor GC，eden 中的存活对象就会被移动到 survivor 区。下一次 eden 满了的时候，再次触发 Minor GC，eden 和 survivor 各有一些存活对象，如果此时把 eden 区存活的对象放到 survivor 区，由于对象所占用的内存不连续，就会产生内存碎片

- 所以我们要设置两个 survivor 区，它最大的好处就是不管什么时候，都有一块空的 survivor 区，这避免了内存碎片的产生

30. eden 区与 survivor 区的空间大小比例为什么是 8:1:1 **

- 一个 eden 区，新生代对象出生的地方
- 两个 survivor 区，一个用来保存上次新生代 GC 存活下来的对象，还有一个空着，在新生代 GC 时把 eden-survivor 中存活的对象复制到这个空的 survivor 中
- 经过统计和经验表明，90%的对象朝生夕死，存活时间极短，每次 GC 会有 90%对象被回收，剩下的 10%要预留一个 survivor 空间去保存

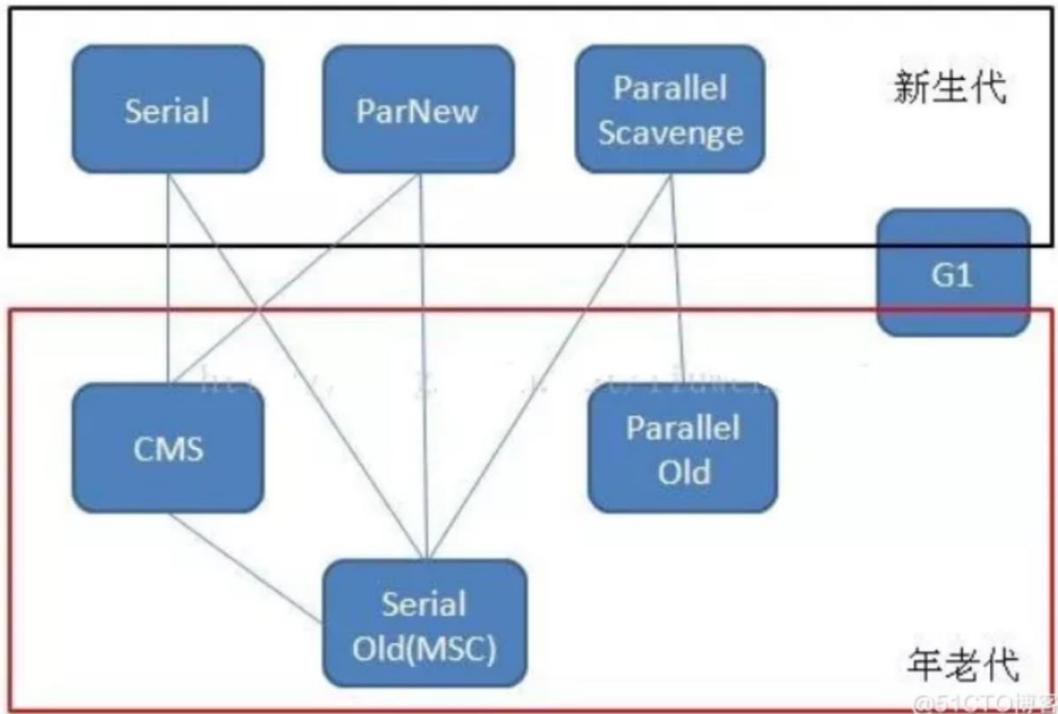
31. JVM 中的垃圾回收算法 ***

- 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带，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

32. 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 Scanvange 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 中的存活对象复制到 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

33. full gc 是什么 *

- minor gc：新生代（包括 eden 区和 survivor 区）的垃圾回收
- major gc：老年代的垃圾回收
- full gc：整个堆空间的垃圾回收

并发编程

并行：单位时间多个处理器同时处理多个任务

并发：一个处理器处理多个任务，按时间片轮流处理

1. 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

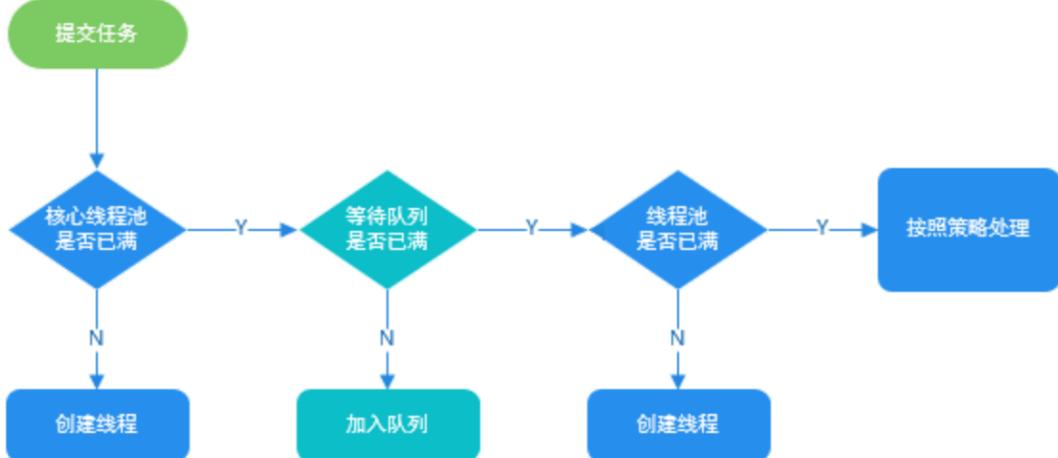
补充：多线程的优缺点？

- 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

2. 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，调度和监控
- 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：
 - 通过 ThreadPoolExecutor 构造函数实现（推荐）
 - 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）

3. 线程有哪几种状态 **

- 线程一共有 6 种状态，分别是 NEW 新建、Runnable 运行、Blocked 阻塞、Waiting 等待、Timed Waiting 限期等待、Terminated 终止
 - 新建状态表示 线程被创建但是还没有启动
 - 运行状态表示 线程有可能正在执行，也有可能在等待 CPU 分配资源
 - 阻塞状态表示 线程没有获取到 monitor 锁
 - 无限期等待状态表示 不会分配 CPU 资源，需要显式唤醒【线程在 run 方法的内部调用了 wait()或者 join()】
 - 限期等待状态表示 在一定时间后会有系统自动唤醒【线程在 run 方法的内部调用了 wait()、join()或者 sleep()，并且传入了等待时间参数】
 - 终止状态 表示 线程执行结束
- 线程中阻塞和等待的区别
 - 阻塞状态指线程在等待其他线程释放 monitor 锁，等待状态指线程在等待其他线程执行某些操作，比如 wait 方法执行完毕或者等待调用 notify 方法唤醒线程

4. sleep, wait, notify, yield 和 join 方法的区别 *

多线程下，需要调用这个对象的 `synchronized` 方法或 `synchronized` 块必须获得对象锁，此时没有获取到的线程就会进入 **锁池**；而获取到锁的线程如果调用了 `wait` 方法，线程就会进入 **等待池**，进入等待池的线程不会竞争该对象的锁

- `sleep()`方法：让当前正在执行的线程在指定的时间内暂停执行
 - 和 `wait()`方法的最大的区别：`sleep` 没有释放锁，而 `wait` 释放了锁；调用 `sleep` 后自动唤醒，而 `wait` 不会自动唤醒，需要其他线程调用 `notify` 方法
- `wait()`方法：释放对象锁，进入等待池（一定是写在 `synchronized` 代码块中）
- `notify()`方法：从等待池中移出任意一个线程放入锁池中（一定是写在 `synchronized` 代码块中）
- `notifyAll()`方法：会将等待池中的所有线程都移动到锁池中（一定是写在 `synchronized` 代码块中）
- `yield()`方法：使当前线程重新回到可执行状态（CPU 时间片用完了），所以执行 `yield()` 的线程有可能在进入到可执行状态后马上又被执行

- `join()`方法：会使当前线程等待调用 `join()`方法的线程结束后才能继续执行

5. 什么是上下文切换 *

- 单核处理器也支持多线程执行代码，CPU 通过给每个线程分配 CPU 时间片来实现这个机制，因为时间片非常短（几十毫秒），所以 CPU 通过不停的切换线程执行，让我们感觉多个线程是同时执行的
- CPU 通过时间片分配算法来循环执行任务，当前任务执行完一个时间片后会切换到下一个任务，但是在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态，所以，任务从保存到再加载的过程就是一次上下文切换

6. 设计一个简单的死锁程序 **

- 解释：两个线程同时开启，线程 1 先获得 `obj1` 对象的锁，然后想要获得对象 `obj2` 的锁，这个时候刚好线程 2 获得了对象 `obj2` 的锁，因此线程 1 会被阻塞，那么线程 2 也会想要获取对象 `obj1` 的锁，显然是无法得到的，也会进入阻塞状态，这就导致两个线程都无法释放自己的锁而结束。

Java

```
public class basicThread {
    public static void main(String[] args) {
        LockDemo A = new LockDemo(true);
        LockDemo B = new LockDemo(false);
        new Thread(A).start();
        new Thread(B).start();
    }
}

class LockDemo implements Runnable {
    public static Object obj1 = new Object();
    public static Object obj2 = new Object();
    boolean flag;
    public LockDemo(boolean flag) {
        this.flag = flag;
    }
    public void run() {
        if(flag) {
            synchronized (obj1) {
```

```

        synchronized (obj2) {
            System.out.println("xxxx");
        }
    }
} else {
    synchronized (obj2) {
        synchronized (obj1) {
            System.out.println("xxx");
        }
    }
}
}
}

```

7. ThreadLocal 相关内容 **

- 定义： ThreadLocal 叫做线程变量，也就是说该变量是当前线程独有的变量。 ThreadLocal 为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量，因此就不会有多线程安全问题。
- 方法：他们可以使用 get 和 set 方法来获取默认值或将其值更改为当前线程所存的副本的值
- 原理：我们从 Thread 类源码中可以看到有一个 threadLocals 变量，它是 ThreadLocalMap 类型的变量，而 ThreadLocalMap 是 ThreadLocal 的静态内部类，当我们调用 set 或者 get 方法的时候，实际上是将变量存放到了当前线程的 ThreadLocalMap
- 问题： 内存泄漏； ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，而 value 是强引用，所以在垃圾回收的时候， key 会被清理掉，而 value 不会被清理掉。这样一来， ThreadLocalMap 中就会出现 key 为 null 的 Entry。假如我们不做任何措施的话， value 永远无法被 GC 回收，这个时候就可能会产生内存泄露。解决手段： 手动调用 remove()方法

8. 并发编程的三个问题 **

- 可见性：是指一个线程对共享变量进行修改，另一个线程要立即得到修改后的最新值
- 原子性：是指在一次或多次操作中，要么所有的操作都执行并且不会受其他

因素干扰而中断，要么所有的操作都不执行

- 有序性：是指程序中代码的执行顺序，java 在编译时和运行时会对代码进行优化，会导致程序最终的执行顺序不一定就是编写代码时的顺序（指令重排在单线程下可以提高性能，但在多线程下会出现问题）

9. 介绍一下 java 内存模型 ***

- Java Memory Model，java 内存模型/JMM，千万不要和 java 内存结构混淆
- 定义：java 内存模型是一套规范，描述了 java 程序中各种变量（线程共享变量）的访问规则，**其规定所有的共享变量都存储在主内存中，每一个线程都有自己的工作内存，工作内存中只存储该线程对共享变量的副本**，线程对变量的所有操作都必须在工作内存中完成，而不能直接读写主内存的变量。
- 作用：在多线程读写共享数据时，对共享数据的可见性、有序性和原子性的保证（synchronized、volatile）
- 主内存和工作内存之间的数据交互过程： lock -> read -> load -> use -> assign -> store -> write -> unlock
- 注意：
 - 如果对一个变量执行 lock 操作，将会清除工作内存中此变量的值
 - 对一个变量执行 unlock 操作之前，必须先把此变量同步到主内存中
- 补充：缓存一致性协议（MSEI）：
 - 多个 CPU 从主内存读取同一个数据到各自的高速缓存，当其中某个 CPU 修改了缓存中的数据，该数据会马上同步回主内存，其它 CPU 通过**总线嗅探机制**可以感知数据的变化，从而使自己缓存中的数据失效

10. synchronized 是如何保证三大特性 **

`synchronized` 能够保证在同一时刻最多只有一个线程执行该段代码，以达到保证并发安全的效果

- `synchronized` 保证原子性的原理：`synchronized` 保证只有一个线程拿到锁，能够进入同步代码块
- `synchronized` 保证可见性的原理：执行 `synchronized` 时，会对应【lock 原子操作】，刷新工作内存中的共享变量的值，得到主内存中共享变量的最新值
- `synchronized` 保证有序性的原理：加上 `synchronized` 之后，【依然会发生重排序】，只不过，我们有同步代码块，可以保证只有一个线程执行同步代码

中的代码，从而保证有序性

- 什么是指令重排？
 - ◆ 在不影响单线程程序执行结果的前提下，计算机为了最大程度发挥机器的性能，会对机器指令进行重排序优化，指令重排不会对单线程程序的结果产生影响，但他可能导致多线程程序出现非预期结果，重排序会遵循 as-if-serial 和 happens-before 规则。
- 为什么要重排序？
 - ◆ 为了提高程序的执行效率，【编译器和 CPU】会对程序中代码进行重排序
- as-if-serial
 - 不管编译器和 CPU 怎么重排序，必须保证【单线程程序的执行结果】不能被改变。为了遵守 as-if-serial 语义，编译器和 CPU 处理器不会对存在数据依赖关系的操作进行重排序
- happens-before
 - 定义：如果一个操作发生在另一个操作之前，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前；如果两个操作之间存在 happens-before 关系，并不意味着必须要按照 happens-before 关系指定的顺序来执行。如果重排序之后的执行结果，与之前执行的结果一致，那么这种重排序也是合理的
 - 具体的规则：
 - 程序顺序规则：在一个线程内必须保证语义串行性，也就是说按照代码顺序执行
 - 锁规则：如果对一个锁解锁后，再加锁，那么加锁动作必须在解锁动作之后同一个锁
 - volatile 变量规则：volatile 变量的写，先发生于读，这保证了 volatile 变量的可见性
 - 线程启动规则：如果线程 A 执行启动线程 B，那么 A 线程的启动线程 B 的操作先于线程 B 的任意操作
 - 传递性：A 先于 B，B 先于 C，那么 A 必然先于 C
 - 线程终止规则：线程的所有操作先于线程的终结，`Thread.join()`方法的作用是等待当前执行的线程终止

- 线程中断规则：对线程 interrupt() 方法的调用先行发生于中断线程的代码检测到中断事件的发生，可通过 Thread.interrupted() 方法检测到线程是否发生中断
- 对象终结规则：在对象没有完成初始化之前，是不能调用 finalize() 方法的

11. synchronized 的特性 **

- 可重入特性
 - 什么是可重入？
 - ◆ 一个线程可以多次执行 synchronized，**重复获取同一把锁**
 - 可重入原理？
 - ◆ synchronized 的锁对象中有一个**计数器**（recursions 变量）会记录线程获得几次锁
 - 可重入的好处
 - ◆ **可以避免死锁**
 - ◆ 可以让我们更好的封装锁
 - 总结：synchronized 是可重入锁，内部锁对象中有一个计数器记录线程获取几次锁了，在执行完同步代码块时，计数器的数量会减 1，直到计数器的数量为 0，就释放这个锁。
- 不可中断性
 - 什么是不可中断？
 - ◆ 一个线程获得锁后，另一个线程想要获得锁，必须处于阻塞或等待状态，如果第一个线程不释放锁，第二个线程会一直阻塞或等待，**不可被中断**
 - 总结：synchronized 属于不可被中断；Lock 的 lock 方法是不可中断的，Lock 的 tryLock 方法是可中断的

12. 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

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

13. ReentrantLock 底层原理 ***

ReentrantLock 默认是非公平锁，也可以指定为公平锁

- ReentrantLock 先通过 CAS 尝试获取锁，如果获取了就将锁状态 state 设置为 1（state=0 表示无锁），但是如果锁已经被占用，先判断当前的锁是否是自己占用了，如果是的话就将锁计数器会 state++（可重入性），如果是被其他线程占用，那么将该线程加入 AQS 队列并 wait()
- 如果当前队列中的头结点的锁被释放时，AQS 队列的第一个等待线程就会被 notify() 唤醒，然后继续 CAS 尝试获取锁，此时
 - ◆ 非公平锁，如果有其他线程尝试 lock()，有可能被其他刚好申请锁的线程抢占。
 - ◆ 公平锁，只有在 FIFO 队列头的线程才可以获取锁，新来的线程只能插入到队尾。
- AQS 队列：
 - ◆ 原理：维护一个共享资源，然后使用队列来保证线程排队获取资源的一个过程
 - ◆ AQS 全称是 AbstractQueuedSynchronizer，抽象队列式同步器，是一个抽象类，定义了一套多线程访问共享资源的同步器框架，通俗理解，**AQS** 就像是一个队列管理员，当多线程操作时，对这些线程进行排队管理。主要通过维护两个变量来实现同步机制
 - state：它是 volatile 修饰的私有变量来表示同步状态。当 state=0 表示释放了锁，当 state>0 表示获得锁。
 - ◆ FIFO 队列：**AQS** 队列通过它来实现线程的排队工作。如果线程获取当

前锁失败，AQS 会将当前线程的信息封装成一个 Node 节点，加入同步队列中，并且阻塞该线程，当同步状态释放，则会将队列中的线程唤醒，重新尝试获取同步状态。

14. synchronized 与 lock 的区别 ***

除非需要使用 `ReentrantLock` 的高级功能，否则优先使用 `synchronized`。这是因为 `synchronized` 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 `ReentrantLock` 不是所有的 JDK 版本都支持。并且使用 `synchronized` 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放

- ◆ `synchronized` 是关键字，而 `lock` 是接口
- ◆ `synchronized` 会自动释放锁，而 `lock` 必须手动释放锁
- ◆ `synchronized` 是不可中断的，`lock` 可以中断也可以不中断
- ◆ 通过 `lock` 【阻塞加锁】（调用 `tryLock` 方法【非阻塞加锁】）可以知道线程有没有拿到锁，而 `synchronized` 不可以
- ◆ `synchronized` 能锁住方法和代码块，而 `lock` 只能锁住代码块
- ◆ `lock` 可以使用读锁提高多线程读效率
- ◆ `synchronized` 是非公平锁（很多等待的线程并不是按照等待的顺序依次执行），`ReentrantLock` 可以控制是否是公平锁（构造器传入参数）

15. volatile 的作用 **

- ◆ 原理：
 - 通过插入内存屏障指令禁止编译器和 CPU 对程序进行重排序
 - 当对声明了 `volatile` 的变量进行写操作时，JVM 就会向处理器发送一条 Lock 前缀的指令
- ◆ 两个作用：
 - 保证可见性：如果变量被 `volatile` 修饰，那么每次修改之后，接下来在读取这个变量的时候一定能读取到该变量最新的值
 - ◆ 如何保证可见性：在工作内存中，每次使用 `volatile` 变量前必须先从主内存刷新变量的最新值到工作内存；每次修改 `volatile` 变量后都必须立刻同步回主内存中，用于保证其他线程可以看到当前线程对 `volatile` 变量所做的修改。
 - 保证有序性：因为编译器或 CPU 会对代码的执行顺序进行优化，导致

代码的实际执行顺序可能与我们编写的顺序是不同的，在多线程的场景下就会出现问题，**利用 volatile 可以禁止这种重排序的发生**

- ◆ **volatile 不能保证原子性**，例如 `a--` 【java 中只有对基本数据类型变量的赋值和读取是原子操作，(`j=i`, `i--` 都不是原子操作)】

■ 适用场景：

- ◆ 如果某个共享变量自始至终只是被赋值或者读取，而没有其他类似于自加这样的复杂操作，我们就可以使用 volatile
- ◆ 布尔标记位，因为 boolean 类型的标记位是会被直接赋值的，此时不会存在复合操作，并且 volatile 可以保证可见性，那么标记位一旦改变，其他线程立马可以得到最新的结果值

16. volatile 和 synchronized 的区别 ***

1. volatile 只能修饰**变量**，synchronized 可以用来代码块或者方法
2. volatile **不能保证数据的原子性**，synchronized 可以保证原子性
3. volatile **不会造成线程的阻塞**，synchronized 会造成线程的阻塞
4. volatile 主要用于解决变量在多个线程之间的**可见性**，synchronized 主要用于解决多个线程之间访问资源的**同步性**

17. CAS 介绍一下 **

- CAS 的全称：compare and swap(比较相同再交换)。是现代 CPU 广泛支持的一种对【内存中的共享数据】进行操作的一种特殊指令
- CAS 的作用：CAS 可以将比较和交换转换为原子操作，这个原子操作直接由 CPU 保证。**CAS 可以保证【共享变量赋值】时的原子性。**
- CAS 的原理：CAS 操作依赖 3 个值：内存中的值 V，旧的预估值 X，要修改的新值 B，如果旧的预估值 X 等于内存中的值 V，就将新的值 B 保存到内存中

18. 乐观锁和悲观锁的区别 **

- 悲观锁从悲观的角度出发，总是假设最坏的情况，每次去修改数据的时候认为别人会修改，所以每次修改的时候都会上锁，这样别人想要修改数据就会被阻塞，因此 **synchronized 称为悲观锁**，同样 **ReentrantLock 也是一种悲观锁，性能较差**

- 乐观锁从乐观的角度出发，总是假设最好的情况，每次去修改数据的时候认为别人不会修改，所以不会上锁，但是在更新的时候会判断在此期间别人有没有去修改这个数据，如果没有人修改则更新，否则重试，**CAS 这种机制称为乐观锁**，性能较好（如果竞争激烈，可以想到重试频繁发生，这时候效率会很低）

19. 锁升级的过程 *

- (jdk5 - jdk6) : 无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁
- 偏向锁
 - ◆ 原理：当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的锁标志位设置为 01，即偏向模式，同时使用 CAS 操作把获取到这个锁的线程的 ID 记录在 mark word 中，如果 CAS 操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步代码块时，虚拟机都可以不再进行任何同步操作，偏向锁的效率较高
 - ◆ 好处：用于在只有一个线程执行同步代码块时提高性能【如果有多个线程竞争，就会撤销偏向锁，提升为轻量级锁】
- 轻量级锁
 - ◆ 引入轻量级锁的目的：在多线程【交替执行】同步块的情况下，尽量避免重量级锁引起的性能消耗，但是如果多个线程再同一时刻进入临界区，会导致轻量级锁膨胀升级重量级锁，所以轻量级的出现并非是要替代重量级锁
 - ◆ 原理：将对象的 mark word 赋值到栈帧中的 Lock Record 中，mark word 更新为指向 lock record 的指针
 - ◆ 好处：在多线程【交替执行同步块】的情况下，可以避免重量级锁引起的性能消耗
- 轻量级到重量级之间的变化：
 - ◆ 自旋锁
 - 问题：线程的阻塞和唤醒需要 CPU 从用户态转为核心态，性能很低，jvm 的开发人员发现共享数据的锁定状态只会持续很短的一段时间，为了这段时间阻塞和唤醒线程并不值得。
 - 定义：让竞争锁的线程等待，只需要让线程执行一个忙循环（自旋次数默认为 10 次），这项技术就是自旋锁

◆ 锁清除

- 指虚拟机即时编译器在运行时，对一些代码上要求同步但是被检测到不可能存在共享数据竞争的锁进行消除
- 举例：StringBuffer 的 append()

◆ 锁粗化

- 指 JVM 会探测到一连串细小的操作都是使用的同一锁对象，那么就可以将同步代码块的范围放大，放到这串操作的外面，这样只需要加一次锁即可，它的核心思想就是扩大加锁范围，避免重复的加锁和解锁

20. synchronized 优化 *

1. 减少 synchronized 的范围：同步代码块中尽量短
2. 降低 synchronized 锁的粒度：将一个锁拆分为多个锁提高并行度
(HashTable -> ConcurrentHashMap)
3. 读写分离：读取时不加锁，写入和删除时加锁 (ConcurrentHashMap)

Redis

1. redis 的数据类型 ***

- redis 有五种数据类型，包括 string, list, set, hash, zset；string 就类似于 java 中的字符串，list 就类似于 java 中的列表，可以存放重复的元素，set 就类似于 java 中的 HashSet，不能存放重复的元素，hash 就类似于 java 中的 HashMap，存放键值对元素，zset 是一个有序集合，每个元素都会有一个 score，按照 score 来进行排序。

2. zset 的底层实现 **

- 它有两种实现形式：一个是压缩列表，还有一种是字典加上跳跃表
 - ◆ 压缩列表使用紧挨在一起的节点来保存元素和 score，第一个节点保存元素，第二个节点保存 score，压缩列表的元素按 score 从小到大排序。

- 它的使用需要满足两个条件，否则就使用字典加跳跃表：元素个数小于 128 个，元素长度小于 64 字节
- ◆ 跳跃表是一种可以进行二分查找的有序链表。原理就是在原有的有序链表上面增加了多级索引，通过索引来实现快速查找；

3. BitMaps

- BitMaps 不是一种数据结构，操作是基于 String 结构的，一个 String 最大可以存储 512M，那么一个 BitMaps 则可以设置 2^{32} 个位，它是针对位的操作的，相较于 String, Hash, Set 等存储方式更加节省空间
- BitMaps 单独提供了一套命令，所以在 Redis 中使用 BitMaps 和使用字符串的方法不太相同，可以把 **BitMaps** 想象成一个以位为单位的数组，数组的每个单元只能存储 0 和 1，数组的下标在 BitMaps 中叫做偏移量
- BitMaps 命令说明：将每个独立用户是否访问过网站放在 BitMaps 中，将访问的用户记作 1，没有访问过的用户记作 0，用偏移量作为用户的 id

4. HyperLogLog

- HyperLogLog 常用于大数据量的统计，比如页面访问量统计或者用户访问量统计
- 要统计一个页面的访问量（PV），可以直接用 redis 计数器或者直接存数据库都可以实现，如果要统计一个页面的用户访问量（UV），一个页用户一天内如果访问多次的话，也只能计算一次，这样，我们可以使用 set 集合来做，因为 set 集合是有去重功能的，key 存储页面对应的关键字，value 存储对应的 userid，这种方法是可行的，但如果访问量较多，假如有几千万的访问量，这就麻烦了。为了统计访问量，要频繁创建 set 集合对象。
- 限制：
 - ◆ 这个 HyperLogLog 结构存在一定的误差，误差很小，0.81%，所以它不适合对精确度要求特别高的统计，而 uv 这种操作，对精确度要求没那么高，是可以适用该结构的

- ◆ HyperLogLog 结构不会存储数据的明细，针对 uv 场景，它为了节省空间资源，只会存储数据经过算法计算后的基数值，基于基数值【去重后的长度】来进行统计的

5. redis 的持久化方案 ***

- RDB（默认的持久化方式）：定期保存数据快照至一个 rdb 文件中，并在启动时自动加载 rdb 文件，恢复之前保存的数据
- AOF：把每一个写请求都记录在一个 aof 文件中，并且在启动时，会把 aof 文件中记录的所有写操作顺序执行一遍，确保数据恢复到最新。
- 区别：RDB 效率要高，因为它是直接将 Redis 的最新数据完整的保存下来，但如果 Redis 中存储的数据量比较大时，也会有一定的性能消耗，因为它需要创建额外的进程来进行数据的持久化，所以要制定合理的策略；而 AOF 效率相对 RDB 要低一些，因为它会将历史的写操作都执行一遍来进行恢复，同时在执行写操作的时候也会将每一个指令存储下来，当我们对数据的安全性要求较高的时候，可以考虑 AOF
- 扩展 1：数据量较大时进行快照，用时相对会比较长。如果服务器在这个期间收到写请求，那么就不能保证快照的完整性。那么 Redis 是如何做的？
 - ◆ Redis 使用的是操作系统提供的写时复制技术(Copy-On-Write, COW)，在执行快照的同时，正常处理写操作。【如果主线程要修改一块数据，这块数据就会被复制一份，生成该数据的副本，此时主线程在该副本上进行修改，fork 的子线程继续把原来的数据写入 RDB 文件中】
- 扩展 2：在进行 RDB 快照的过程中，发生服务崩溃了怎么办？
 - ◆ 在快照操作过程中不能影响上一次备份的数据，将上一次完整的 RDB 快照文件作为恢复内存的参考

6. 缓存穿透，缓存击穿，缓存雪崩 ***

- 在应用程序和 mysql 数据库之间建立一个中间层：redis 缓存，通过 redis 缓存可以有效减少查询数据库的时间消耗，但是引入 redis 又有可能出现缓存穿透，缓存击穿，缓存雪崩等问题

- 缓存穿透：**key 在缓存和数据源都没有**，那么所有的请求都会去查询数据源，压垮数据源

◆ 解决：

- 当查询不存在时，也将空保存在缓存中 【key-null】；
- 采用布隆过滤器，将所有可能存在的数据 hash 到一个足够大的 bitmap 中，一个一定不存在的 key 会被这个 bitmap 拦截掉，从而避免了对数据源的查询压力。
- 缓存击穿：**key 在数据源中有，在缓存中没有**，通常是缓存过期了，这时候大量的并发就会全部去请求数据源，导致数据源瘫痪
 - ◆ 解决：使用互斥锁，利用 `setnx` 来实现该功能，因为当 key 不存在的时候，它才会设置一个 key
- 缓存雪崩：当缓存服务器重启或者大量 **key 缓存集中在某一个时间失效**，这时候给数据源的压力也非常大
 - ◆ 解决：为 key 设置不同的缓存失效时间

7. redis 实现分布式锁 *

锁是一种常用的并发控制机制，用于保证一项资源在任何时候只能被一个线程使用，如果其他线程也要使用同样的资源，必须排队等待上一个线程使用完。

上面说的锁指的是程序级别的锁，例如 Java 语言中的 `synchronized` 和 `ReentrantLock` 在单应用中使用不会有任何问题，但如果放到分布式环境下就不适用了，这个时候我们就要使用分布式锁。分布式锁比较好理解就是用于分布式环境下并发控制的一种机制，用于控制某个资源在同一时刻只能被一个应用所使用。

分布式锁比较常见的实现方式有三种：`Memcached` 实现的分布式锁：使用 `add` 命令，添加成功的情况下，表示创建分布式锁成功。`ZooKeeper` 实现的分布式锁：使用 `ZooKeeper` 顺序临时节点来实现分布式锁。`Redis` 实现的分布式锁。

- 使用 `setnx`（`set if not exists`），如果创建成功则表明此锁创建成功，否则代表这个锁已经被占用创建失败。
- `setnx lock true` 创建锁

- `del lock` 释放锁
- `setnx` 的问题：`setnx` 虽然可以成功地创建分布式锁，但存在一个问题，如果此程序在创建了锁之后，程序异常退出了，那么这个锁将永远不会被释放，就造成了死锁的问题。
 - ◆ 使用 `set` 命令来设置分布式锁，并设置超时时间了，而且 `set` 命令可以保证原子性：`set lock true ex 30 nx`

第三部分 计算机基础

计算机网络

1. 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**；

2. 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。

3. TCP 连接建立为什么需要三次握手 **

- 因为三次握手目的就是**让双方都确认自己与对方的发送与接收都是正常的**。通过第一次握手，服务端可以确认客户端发送正常和自己接收正常；通过第二次握手，客户端可以确认自己发送和接收正常，以及对方发送和接收正常；现在客户端就可以确认自己与对方的发送和接受都没问题，但是服务端就不

能确定自己发送和对方接收正常；所以还需要第三次握手，客户端向服务端发送 ACK 包，当服务端收到这个 ACK 包之后，就可以确定自己的发送能力和接收能力都是正常的。

- 不采用两次握手的原因：**防止已失效的连接请求报文段突然又传到了服务器而产生错误。**假如，第一次连接请求报文在网络中滞留了，然后就会重传一次连接请求，连接建立，数据传输然后断开。这个时候呢，滞留的连接请求又到达了服务器，此时服务器就会认为客户端又发来了请求，如果是两次握手，那么服务器就会返回确认报文，客户端认为是失效的请求就不会进行处理，那么服务器就会一直等待客户端传输数据，造成资源浪费。

拓展：

- 第 2 次握手传回了 ACK，为什么还要传回 SYN：服务端传回客户端所发送的 ACK 是为了告诉客户端，我接收到的信息确实就是你所发送的信号了，这表明从客户端到服务端的通信是正常的。而回传 SYN 则是为了建立并确认从服务端到客户端的通信。
- 为什么不能是四次：因为三次握手已经能说明握手时的通信是正常的，所以四次握手就会**浪费资源**

4. TCP 连接释放为什么需要四次挥手 **

- 我主要说一下为什么需要第二次挥手，因为当服务端收到客户端的 FIN 数据包后，服务端可能还有数据没发处理完，所以不会立即 close，只能先发送一个 ACK 包给客户端，表示服务端接收到了客户端的断开连接请求，然后再等一会，等服务端处理完数据，再发送 FIN 包给客户端

5. TCP 连接释放的第四次握手为什么要等待 2MSL **

- 先说一下为什么要等待，就是**为了保证客户端发送的最后一个确认报文段能够到达服务器**，如果 ACK 在 2MSL 内没有到达服务器，**可以进行重传**。
- 再说一下为什么是 2MSL 呢，因为如果客户端给服务端发送的 ACK 丢失，**服务端等待 1MSL 没收到，然后告诉客户端需要 1MSL**，所以在 2MSL 内可以进行重传确认报文，并且重启计时器

6. 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。

7. TCP 流量控制 **

- 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。
- 传输层的流量控制和数据链路层的流量控制的区别：
 - ◆ 传输层定义端到端用户之间的流量控制，数据链路层定义两个中间的相邻节点的流量控制
 - ◆ 数据链路层的窗口大小不能动态变化，传输层的则可以动态变化

8. TCP 拥塞控制 **

- 拥塞控制就是 防止过多的数据注入网络，保证网络中的 路由器或者链路 不致于过载。它和流量控制一样，都是通过控制发送方发送数据的速率来达到控制效果。发送方在确定发送报文段的速率时，既要根据接收方的接收能力，又要从全局考虑不要使网络发生拥塞。因此 TCP 协议要求发送方维护两个窗口，分别是接收窗口和拥塞窗口，接收窗口就是 接收方能够接收的数据量，拥塞窗口 是发送方根据自己估算的网络拥塞程度而设置的窗口值。发送窗口的上限值就是接收窗口和拥塞窗口中较小的一个。
- 主要有四种算法来进行拥塞控制，包括 慢开始、拥塞避免、快重传、快恢复
 - ◆ 慢开始：发送方刚开始发送数据的时候，将拥塞窗口设置为 1，当收到接收方的确认信息后，将拥塞窗口加倍，这样拥塞窗口就逐渐变大。
 - ◆ 拥塞避免：由于慢开始到后面传输速率会变得很大，那么网络拥塞的概率就会变大，所以当拥塞窗口到达一个阈值的时候，就会改用拥塞避免算法，就是让拥塞窗口 $cwnd$ 缓慢增加，不是加倍而是每次把发送方的 $cwnd$ 加 1。
- 网络出现拥塞的处理：无论是慢开始阶段还是拥塞避免阶段，只要发送方检测到超时事件的发生，就要把慢开始的阈值调整为当前拥塞窗口的一半，然后把拥塞窗口设置为 1，执行慢开始算法
- 快重传：如果接收方接收到一个不按顺序的报文段，它会给发送方回复一个 重复确认，那么当连续收到三个重复确认时，说明其下一个报文段丢失了，发送方就会立即重传丢失的报文段。
- 快恢复：当发送端连续收到三个重复确认，就将拥塞窗口设置为阈值的一半，而不是执行慢开始算法

当发送方检测到超时的时候，就采用慢开始和拥塞避免；当发送方接收到冗余 ACK，就采用快重传和快恢复

9. 流量控制和拥塞控制的区别 *

- 拥塞控制是让网络能够承受现有的网络负荷，是一个全局性的过程，涉及所有的主机、路由器以及降低网络传输性能有关的所有因素。相反，流量控制是点对点的通信量的控制，它所要做的就是抑制发送端发送数据的速率，使接收端来得及接收。

10. 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

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

11. 视频面试中用 TCP 还是 UDP *

- 用 UDP，因为 UDP 是一个无连接的协议并且不用保证可靠性，所以传输速度快。对于视频来说，即使偶尔丢失一两个数据包，也不会对接收端产生太大影响。难道它一点也不可靠吗？其实 UDP 传输的可靠性是由应用层来负责的。现在有一种 RUDP 协议，结合了 UDP 和 TCP 的优点，既保证了高效，又保证了可靠

12. UDP 如何实现可靠传输 *

- 因为传输层无法保证数据的可靠传输，所以只能通过应用层来实现了，实现的方式可以参照 TCP 协议可靠性传输的方式，只是实现不在传输层，转移到了应用层，比如添加 seq/ack 机制，添加滑动窗口机制，或者添加超时重

传机制

13. TCP 粘包问题以及解决方案 *

- TCP 粘包是指发送方发送的若干包数据到接收方接收时粘成一包，TCP 拆包是指发送方发送的一个包数据到接收方分成了多个包的数据。
- 解决：
 - ◆ 通过 特殊标识符 表示数据包的边界；
 - ◆ 在 TCP 报文的头部加上表示数据长度的字段，然后按照长度读取

14. 域名解析的过程 **

域名解析是 指把域名映射成为IP 地址或者把IP 地址映射成域名的过程，前者为正向解析，后者为反向解析

- 浏览器首先会在缓存中看是否有域名对应的 ip 地址，如果没有，就去本地磁盘找 hosts 文件。
- 如果本地的 hosts 文件没有，浏览器会发出一个 dns 请求到本地域名服务器（本地域名服务器一般都是你的网络接入服务器商提供，比如中国电信，中国移动等）
- 本地域名服务器收到请求后，查询本地缓存，如果没有，则以 DNS 客户的身份向根域名服务器发出解析请求。
- 根域名服务器收到请求后，会返回该域名对应的顶级域名服务器的 IP 地址给本地域名服务器
- 本地域名服务器会向顶级域名服务器发出解析请求
- 顶级域名服务器收到请求后，不会直接返回域名和 IP 地址的对应 关系，而是返回授权域名服务器的 IP 地址。
- 本地域名服务器会向授权域名服务器发出解析请求

- 授权域名服务器收到请求后，将查询结果返回给本地域名服务器
- 本地域名服务器将查询结果保存到本地缓存，同时返回给客户端

15. 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

16. 介绍 HTTP **

- HTTP 协议又叫做超文本传输协议，它是一个应用层协议，规定了在浏览器和服务器之间的请求和响应的格式与规则。它有几个重要的特点
 - ◆ HTTP 是无状态的
 - ◆ HTTP 采用 TCP 作为传输层协议，保证了数据的可靠传输
 - ◆ HTTP 是无连接的，也就是说通信的时候不需要先建立 HTTP 连接

17. HTTP 1.0 和 HTTP 1.1 的主要区别是什么 *

- 在 HTTP1.0 中默认是短连接，就是客户端向服务器每进行一次请求，就要建立一次 TCP 连接，任务结束就会关闭连接；从 HTTP1.1 起默认是长连接，就是当一个网页打开后，客户端和服务器之间的 TCP 连接不会立马关闭，当客户端再次访问这个服务器时，会继续使用这一条已经建立的连接

18. HTTP 和 HTTPS 的区别 *

密码学知识：

对称加密：加密和解密使用同一把密钥；非对称加密：加密和解密使用不同的密钥

Https 使用非对称加密来传输对称秘钥来保证安全性，使用对称加密来保证通信的效率

- HTTP 的默认端口号是 80，HTTPS 的默认端口号是 443

- HTTP 协议运行在 TCP 之上，所有传输的内容都是未加密的，而 HTTPS 协议是运行在 SSL 协议之上的 HTTP 协议，所有传输的内容都是加密的，所以 HTTP 安全性没有 HTTPS 高；
- HTTPS 协议需要到 CA 申请证书，所以需要耗费一定的费用
- 补充：HTTPS 加密的过程？
 - ◆ 客户端向服务端发起 SSL 连接请求
 - ◆ 服务端把公钥发送给客户端，并且服务端保存唯一的私钥
 - ◆ 客户端利用公钥进行加密，然后将数据发送给服务端
 - ◆ 服务端收到数据后利用私钥进行解密

19. HTTP 是如何保存用户状态的 *

- HTTP 是无状态的，也就是说，同一个客户第二次访问同一个服务器上的页面时，服务器的响应与第一次被访问时相同，这种设计使得服务器更容易支持大量并发的 HTTP 请求，在实际应用中，通常使用 cookie 和 session 来保存用户状态。

用来跟踪用户的整个会话，保存状态

- 存储的位置不同，cookie：存放在客户端，session：存放在服务端，因此 session 更加安全
- 存储的数据大小限制不同，cookie：大小受浏览器的限制，很多都是 4K 的大小， session：理论上受当前内存的限制
- 生命周期的不同，cookie 是从创建开始，20 分钟后，cookie 数据就会被清除，而 session 是每次访问后开始计时，如果 20 分钟内没有访问过，就会被清除，如果访问过，那么从访问时重新计时

20. GET 和 POST 的区别 **

- 用途不同：get 一般用于获取数据，而 post 一般用于提交数据

- 安全性不同：get 把请求的数据放在 url 上，而 post 把数据放在请求体中，**更加安全**
- 数据长度不同：get 对传输的数据长度有限制（与浏览器有关），而 post 理论上没有限制，因为数据都是放在请求体中的
- 是否自动缓存：get 请求会被浏览器自动缓存，而 post 的缓存需要手动设置
- 是否符合幂等性：GET 是幂等的，而 POST 不是幂等的（对同一 URL 的多个请求应该返回同样的结果）
 - ◆ get 只是查询数据，不会影响到资源的变化，而 post 如果调用多次，都将产生新的资源
- tcp 数据包不同：GET 请求会在发送过程中产生一个 TCP 数据报，而 POST 在提交过程中会产生两个 TCP 数据报
 - ◆ get：浏览器会将 http header 和数据 data 一起发送出去，服务器响应 200（返回请求的数据）
 - ◆ post：
 - 浏览器先发送 header，等待服务器响应 100 continue
 - 浏览器再发送 form，服务器响应 200 OK

21. 常见的状态码 *

- 200：请求正常处理；
- 304：如果客户端访问的资源本身就有**缓存**时；
- 403：服务器不允许客户端访问请求的资源；
- 404：服务器**没有客户端请求的资源**；
- 503：服务器**在维护状态**，不能处理请求

22. ARP 地址解析协议 *

- 数据链路层协议，作用：把 IP 地址解析为 MAC 地址
- 工作流程：
 - ◆ 发送 ARP 广播请求，内容是 我是 IP 多少，MAC 多少，谁是 IP 多少，MAC 多少
 - ◆ 接收 ARP 单播应答

操作系统

1. 什么是操作系统 *

- 操作系统本质上就是一个运行在计算机上的程序，用于管理计算机的硬件和软件资源；它里面最核心的部分就是操作系统内核，负责内存管理，硬件管理，文件系统管理以及应用程序管理，它可以说是连接应用程序和硬件资源的桥梁。

2. 什么是系统调用 *

内存分为用户空间和系统空间，系统空间是给操作系统使用的，用户空间是应用程序使用的，应用程序如果要访问系统空间，需要进行系统调用从用户态切换到内核态

- 系统调用就是说，当我们在运行程序的时候，如果需要调用操作系统提供的系统态级别的功能比如文件管理，进程通信，内存管理等，这时候就必须通过系统调用方式向操作系统请求，并且由操作系统代替完成
 - ◆ 用户态即上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源，包括 CPU 资源、存储资源、I/O 资源等。为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口：即系统调用。
- 补充：操作系统为什么会有用户态和内核态？
 - ◆ 用户态：用户态运行的进程可以直接读取用户程序的数据；系统态：系

统态运行的进程几乎可以访问计算机的所有资源

- ◆ 在计算机系统中，通常运行着两类程序，分别是系统程序和应用程序，为了保证系统程序不被应用程序有意或无意地破坏，因此为计算机设置了两种状态：内核态和用户态，操作系统程序在内核态运行，应用程序只能在用户态运行；在实际运行过程中，CPU会在内核态和用户态之间进行切换。
- 补充：用户态如何切换到内核态
 - ◆ 其中系统调用认为是用户进程主动发起的，异常和外围设备中断则是被动的
 - ◆ 系统调用，这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作；系统调用本身就是中断
 - ◆ 异常，当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常
 - ◆ 外围设备的中断，当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序

3. 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。

4. 进程有哪几种状态 *

■ 进程一共有五种状态，分别是新建状态（也就是进程刚创建的时候），就绪状态（做好了一些资源的准备，但是还在等待 CPU 资源），运行状态（进程正在执行的时候），阻塞状态（就是进程在运行的过程中由于等待某一个资源而暂停），结束状态（进程运行结束了）

5. 进程间的通信方式 **

■ 我知道的进程间的通信方式有管道，消息队列，共享内存，信号量，和套接字；

- ◆ 管道专门用于单向通信，比如 Linux 里面的那个|，就是会把前面的命令的输出作为后面的命令的输入
- ◆ 消息队列就有点像队列，比如进程 a 要给进程 b 发送数据，只需要把数据放到消息队列中，不用等 b 来取，进程 b 需要的时候去取就可以了
- ◆ 共享内存就是 让多个进程可以访问同一块内存空间，不同进程就可以及时看到对方进程对数据的更新
 - 但是会造成数据不安全，一般会将它和信号量一起使用，信号量就是一个计数器，用来解决进程安全问题，比如信号量初始的时候是 1，进程 a 来访问的时候发现是 1 就对内存进行访问，并且将信号量置为 0，当这个时候进程 b 也来了，发现是 0，就知道有人在访问，不能同时访问了。
- ◆ 以上方式都是多个进程在一台主机之间的通信，套接字主要用于客户端

和服务器之间的通信，例如我们平时通过浏览器发起一个 http 请求，然后服务器给你返回对应的数据

- 扩展：线程间的通信方式有哪些

- ◆ 同一进程中的线程之间有共享内存，因此它们之间的通信是通过共享内存实现的
- ◆ 不同进程的线程之间进行通信，等同于进程间的通信

6. 进程同步的四种方式 **

- 临界区：对临界资源进行访问的那段代码称为临界区；因为每次只准许一个进程进入临界区，进入后不允许其他进程进入。
- 同步与互斥：同步就是说多个进程访问同一个资源有一定的先后执行顺序，然后互斥就是说多个进程在同一时刻只有一个进程能进入临界区。
- 信号量：是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。down：如果信号量大于 0，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；up：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。基本原理就是两个或多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号；P 操作(递减操作)可以用于阻塞一个进程，V 操作(增加操作)可以用于解除阻塞一个进程。
- 管程：是由一个或多个过程、一个初始化序列和局部数据组成的软件模块，特点是局部数据变量只能被管程的过程访问，任何外部过程都不能访问；一个进程通过调用管程的一个过程进入管程；在任何时候，只能有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。

7. 进程的调度算法 ***

- 进程的调度算法包括 先到先服务，短作业优先，时间片轮转，优先级调度，多级反馈队列；
- ◆ 先到先服务，从就绪队列中选择一个最先进入该队列的进程分配资源；

- ◆ 短作业优先就是从就绪队列中选出一个估计运行时间最短的进程分配资源；
- ◆ 时间片轮转就是每次给队首分配一个时间片，然后时间片用完时就再加入到队尾中去；
- ◆ 优先级调度就是 首先为每个进程分配优先级，然后每次从就绪队列中选择优先级最高的进程分配资源，后面以此类推；
- ◆ 多级反馈队列就是 首先会设置多个就绪队列，并为每个队列赋予不同的优先级，优先级越高的队列中，那么分配的时间片越小，新的进程首先会放入优先级最高的队列的队尾，如果时间片够用，那么就可以执行完，如果不够用，就放入优先级低一点的队列中，以此类推

8. 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

9. 常见的几种内存管理机制 *

内存管理主要有两个功能：一个是负责内存的分配和回收，还有一个是负责地址转换，也就是将逻辑地址转换为物理地址

- 内存管理主要有块式管理，页式管理，段式管理和段页式管理；
- ◆ 块式管理就是将内存分为一些固定大小的块，如果程序运行需要内存的话，操作系统就会分配给它一块。它有一个缺点，就是如果程序只需要很小的空间，那么分配的这一块内存很多没有被利用。
- ◆ 页式管理就是 将内存分为一些固定大小的页，它比块要小，这样可以提高内存利用率。
- ◆ 段式管理就是 将内存分为一些段，它比页要小，这样内存利用率更高，但是由于段很小，那么一个程序执行可能需要很多个段，这样会浪费很多时间在计算每一段的物理地址上面。
- ◆ 段页式管理就是 结合了段式管理和页式管理，也就是 先将内存分为一些段，然后再把段分为一些页。

10. 分页机制和分段机制的共同点和区别

- 共同点：这两种内存管理机制都提高了内存的利用率，并且页和段都是离散存储的
- 区别：页的大小是固定的，由操作系统决定，而段的大小不固定，取决于当

前运行的程序；页是没有实际意义的，而段是有实际意义的，也就是说每个段定义了一组逻辑信息，比如代码段，数据段。

11. 快表和多级页表 *

- 两个问题：
 - ◆ 虚拟地址到物理地址的转换要快。--- 快表
 - ◆ 解决虚拟地址空间大，页表也会很大的问题。 --- 多级页表
- 快表的作用就是为了提高虚拟地址到物理地址的转换速度，它就相当于一个缓存器，大致的流程是 先根据虚拟地址中的页号查快表，如果快表中有，就直接返回对应的物理地址，如果快表中没有，就访问内存中的页表，再从页表中返回物理地址，同时将这个映射添加到快表中，如果快表满了，就淘汰最久未被使用的页的映射
- 多级页表的作用就是为了防止把全部页表放在内存中占用过多空间，就是通过增加多级的目录来降低存储空间。多级页表属于时间换空间的典型场景。

12. CPU 寻址了解吗？为什么需要虚拟地址空间？

- 虚拟寻址就是 CPU 需要将虚拟地址翻译成物理地址，这样才能访问到真实的物理内存。实际上完成虚拟地址转换为物理地址的硬件是 CPU 中含有一个被称为 内存管理单元（Memory Management Unit, MMU）的硬件。
- 如果没有虚拟地址空间，那么程序都是直接访问和操作的都是物理内存，这样可能会破坏操作系统，造成系统崩溃，同时如果同时运行多个程序会出现覆盖的问题。如果有虚拟地址空间，程序可以使用一系列相邻的虚拟地址来访问物理内存中不相邻的大内存缓冲区，同时程序可以使用一系列虚拟地址来访问大于可用物理内存的内存缓冲区。

13. 什么是虚拟内存(Virtual Memory) *

- 虚拟内存是操作系统内存管理的一种技术，它可以让程序使用一系列连续的虚拟地址来访问物理内存中不相邻的内存空间，并且有一部分映射到磁盘上，这样就可以让程序使用超过真实物理内存大小的空间。

14. 什么是局部性原理

- 局部性原理就是 指程序在执行的时候往往呈现局部性规律，主要包括时间局部性和空间局部性，时间局部性就是说如果程序中的某条指令一旦执行，不久以后该指令可能再次执行，空间局部性就是说一旦程序访问了某个存储单元，那么不久以后其附近的存储单元也将被访问

15. 虚拟存储器

- 基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其他部分留在外存，就可以启动程序执行。由于外存往往比内存大很多，所以我们运行的软件的内存大小实际上是可以比计算机系统实际的内存大小大的。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存，然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换到外存上，从而腾出空间存放将要调入内存的信息。这样，计算机好像为用户提供了一个比实际内存大的多的存储器——虚拟存储器。

16. 虚拟内存的几种实现 **

- 虚拟内存的实现需要建立在离散分配的内存管理方式的基础上，包括请求分页存储管理，请求分段存储管理，请求段页式存储管理。我详细说一下请求分页存储管理，是建立在分页管理之上，增加了页面置换的功能。它在作业运行之前，只需要装入要执行的部分页面即可，如果在作业运行的过程中发现要访问的页面不在内存中，则将相应的页面调入到内存中，如果发现内存中没有空闲的空间，那么可以通过页面置换算法选择一个页面移到外存中。
- 拓展：基本分页存储管理和请求分页存储管理的区别
 - ◆ 他们的主要区别是 基本分页存储管理不支持虚拟存储器功能，也不支持页面置换功能，同时 在作业运行的时候，要将作业一次性装入到内存中，而请求分页存储管理支持虚拟存储器功能，也支持页面置换功能，同时 在作业运行的时候，只需要将作业的一部分页面装入到内存中。
 - ◆ 总的来说，基本分页存储管理不提供虚拟内存，而请求分页存储管理提供虚拟内存

17. 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；

18. 僵尸进程和孤儿进程是什么 *

- 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程；孤儿进程将被 **init 进程**(进程号为 1)所收养，并由 init 进程对它们完成状态收集工作
- 僵尸进程：当一个子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的一些信息（比如进程号会一直占用）仍然会保存在系统中。这种进程称之为僵死进程。
- 扩展：守护进程是什么
 - ◆ 指在后台运行的，没有控制终端与之相连的进程。它独立于控制终端，周期性地执行某种任务，比如数据库服务器 mysqld
 - ◆ 和用户线程的区别？
 - 平时使用到的线程均为用户线程；用来服务用户线程的线程叫守护线程，比如垃圾回收线程
 - 主要区别：在于 jvm 是否存活（如果只有守护线程，JVM 结束）

数据库

1. 索引是什么 **

- ◆ 索引是一种帮助 mysql 提高查询效率的**数据结构**，就像书的目录一样
- ◆ 优点：大大加快数据查询速度
- ◆ 为什么使用索引
 - 索引可以加快数据**查询速度**
 - 索引帮助我们排序以避免使用**临时表**
 - 索引可以将随机的 IO 转换为顺序的 **IO(B+树的叶子结点是连接在一起的)**
- ◆ 索引这么多优点，为什么不对表中的每一个列创建一个索引
 - 创建索引和维护索引要耗费时间，并且时间随着数据量的增加而增加
 - 索引需要占用磁盘空间来进行存储

2. mysql 中索引的分类有哪些 **

- ◆ 普通索引，一个索引只包含单个列，一个表可以有多个普通索引
- ◆ 唯一索引，索引列的值必须唯一，但允许有空值，且只允许有一个空值
- ◆ 主键索引，设置为主键后数据库会自动建立索引，一个表只能有一个主键索引，并且不能为空
- ◆ 复合索引，一个索引包含多个列【**使用组合索引的时候，遵循最左前缀原则，并且 mysql 引擎为了更好利用索引，在查询的时候，会动态调整查询字段顺序以便利用索引**】

- ◆ 全文索引，主要用来查找文本中的关键字，而不是直接和索引中的值进行比较，只有 char,varchar,text 类型的列上可以创建全文索引（用来解决模糊匹配效率低的问题）

3. B+树和 B 树的区别 ***

- ◆ 他们都是一种平衡的多路查找树，但是有两点区别
 - B+树非叶子节点只存储键值信息，而 B 树非叶子节点还存储数据记录，这样 B+树会矮一些，那么查询的时候 IO 就少一些
 - B+的数据记录存储在叶子节点中，并且所有叶子结点之间都有一个链指针，而 B 树数据记录存储在任意节点中，这样在进行区间查询的时候，需要中序遍历，效率会更低

4. 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。

5. 为什么不用二叉树，红黑树，哈希表，B 树 *

- ◆ 为什么不用二叉树？
 - 二叉树的节点如果变多的时候，树的高度就会变很高，因此磁盘 IO 次数就会增多，查询效率就会变低，最坏的时候，退化成一个链表，以至于

出现了平衡二叉树，限制任何节点的左子树和右子树的高度不超过 1

◆ 为什么不用红黑树？

- 红黑树的出度都为 2，当节点很多的时候，红黑树的高度会比 B+树高很多，IO 次数就会多很多，导致查询变慢

◆ 为什么不用哈希表？

- 索引的值在哈希表中并不是有序的，所以不适合于区间查询；适用场景：等值查询
- 主要有两点原因：

- ◆ 从内存角度上看，hash 表需要将索引全部加载到内存中，如果数据量大的话，这就会很消耗内存，而采用 B+树的话，索引会分批加载到内存中，减小了内存压力
- ◆ 从业务场景上看，如果只查找一个值，那确实是 hash 更快，但是数据库中经常进行范围查询，由于 hash 结构索引是无序的，就会慢一点了，而 B+树索引是有序的，并且又有指针相连，它的查询效率就会高很多

- 哈希表存在哈希冲突，所以哈希索引的性能是不稳定的

◆ 为什么不用 B 树？

- 在第 4 题中已经回答了

6. 聚集索引和非聚集索引的区别 **

◆ 他们都是 B+树的数据结构，但是有三点区别

- 聚集索引的叶子结点存储数据记录，而非聚集索引的叶子结点不存储记录，而是存储数据行地址
- 聚集索引的索引和数据放在一块，索引存放顺序和数据的存放顺序也是

一致的，而非聚集索引的索引和数据是分开存储的，并且存放顺序是不一致的

- 聚集索引查询的时候可以直接获取数据，而非聚集索引（二级索引，辅助索引）需要第二次查询才能获取数据
- ◆ 补充：Innodb 和 MyISAM 的索引的区别？
 - Innodb 使用的是聚簇索引，**默认是主键**，如果表中没有定义主键，Innodb 会选择一个唯一且非空的索引代替，如果没有这样的索引，Innodb 会隐式定义一个主键来作为聚簇索引。主键索引 B+树的叶子节点存储的是行数据，辅助索引 B+树的叶子节点存储的是主键值，在利用辅助索引查询的时候，需要经过两个步骤，首先在辅助索引 B+树中找到对应的主键，然后在主键索引 B+树中找到对应的行数据。
 - MyISAM 使用的是非聚簇索引，主键索引 B+树和辅助索引 B+树的叶子结点都保存一个地址指向真正的表数据，表数据存储在其他地方，由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

7. 回表查询是什么 **

- ◆ 定义：如果是根据主键索引去查，那么只需要查询主键索引树，就可以拿到对应的结果；如果是根据非主键索引去查，那么需要首先查询非主键索引的 B+树找到对应的主键值，然后再去查询主键索引的 B+树找到对应的结果，这个过程叫做**回表查询**
- ◆ 比如，表的字段：id, name, age, gender id 主键， name 普通索引
- ◆ 执行语句：select * from table where name = 'zhangsan'
- ◆ 执行过程：先根据 name 到 name B+树找到对应叶子结点的 id 值，然后再根据 id 到 id B+树读取整行记录，这种查询方式叫做回表，不推荐使用，效率降低

8. 覆盖索引是什么 **

- ◆ 定义：只通过查询非主键索引的 B+树就可以覆盖查询的要求，也就是不需

要回表的过程

- ◆ 执行语句: `select id, name from table where name = 'zhangsan'`
- ◆ 执行过程: 根据 name 的值到 name B+树找到对应叶子结点的数据, 叶子结点中包含了全部要查询的字段, 此时叫做索引覆盖, 推荐使用, 效率高

补充. 最左匹配

SQL

```
-- 最左前缀原则: 可以是组合索引的最左 N 个字段, 也可以是字符串索引的最左 N 个字符
-- 表字段 id,name,age,gender  id 主键,  name, age 组合索引
select * from table where name = ? and age = ?      √
select * from table where name = ?      √
select * from table where age = ?      ✗
select * from table where age = ? and name = ?      √  (有优化器会对查询字段顺序进行优化)
```

9. 索引下推 *

- ◆ 在组合索引中, 当查询条件涉及到多个索引字段的时候, **并不是只判断一个索引字段的值, 就去回表**, 而是先对符合最左前缀原则的所有字段进行判断, 再去回表, 这样就可以减少回表查询的次数
- ◆ 扩展: 谓词下推

■ 谓词下推就是 SQL 中的 where 语句可以下推到数据源或者靠近数据源的部分, 通过尽早过滤数据, 减少处理时间

10. 主键索引和辅助索引具体是什么 *

- ◆ 聚簇索引默认就是主键索引, 一个字段设置为主键后数据库会自动建立索引, 一个表只能有一个主键索引, 并且不能为空; 在聚簇索引之上创建的索引就叫做辅助索引 (**非聚集索引都是辅助索引**), 辅助索引叶子节点存储的不再是行的物理地址, 而是主键值, 辅助索引访问数据总是需要二次查找

11. 为什么建议用自增 id 做索引而不用 UUID *

- ◆ 如果使用自增 id 作为主键索引，那么每次插入新的记录，记录就会按照顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页，索引结构相对紧凑，**磁盘碎片少，效率也高**
- ◆ 解释一下为什么自增呢？
 - 如果使用非自增主键，由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置，频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构（索引维护耗时）
 - 如果使用 UUID 作为主键索引，由于 UUID 的值太过离散，会造成非常多的数据插入和数据移动，**导致索引树调整复杂度变大**，消耗更多的时间和资源，还会产生大量的磁盘碎片，导致性能变低
- ◆ 缺点：使用 id 自增会泄漏业务量

12. 主键索引使用 int 和 string 有啥区别 *

- ◆ 用 int 可以自增，那么插入的时候按顺序插入到后面就可以了，而 string 是不可自增的
- ◆ string 类型作为主键比 int 类型占用的**存储空间大**，那么辅助索引中保存的主键值也会跟着变大，这样就会浪费存储空间，也会影响到查询的性能
- ◆ 在涉及到比较的查询中，string 类型的比较比 int 类型更加复杂一些

13. 缺少主键的话 mysql 怎么处理 *

- ◆ mysql 的默认存储引擎 Innodb 使用的是聚簇索引，**默认是主键**，如果表中没有定义主键，Innodb 会选择一个**唯一且非空**的索引代替，如果没有这样的索引，Innodb 会**隐式定义一个主键**来作为聚簇索引
- ◆ 存在的问题：
 - 使用不了主键索引，查询就会进行全表扫描

- 对于生成的 ROW_ID（自动生成的一个不可见的列名），其自增的实现来源于一个全局的序列，而所有有 ROW_ID 的表共享该序列，这也意味着插入的时候生成需要共享一个序列，那么高并发插入的时候为了保持唯一性就避免不了锁的竞争，进而影响性能

14. 选什么字段当索引，索引何时失效 ***

- ◆ 何时需要索引：【基数 $\text{distinct value / count} > 80\%$ 适合创建索引】

- 需要频繁被作为查询条件的字段
- 查询过程中排序的字段（因为索引已经排序....）
- 查询过程中分组或者聚合的字段
- 和其他表做连接查询的字段
- 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，所以其指定的范围是连续的）

- ◆ 索引何时失效：

- 查询语句中使用 like 关键字，如果匹配字符串的第一个字符为%，索引不会被使用
- 查询语句中使用复合索引，如果查询字段不满足最左前缀原则，索引不会被使用
- 查询语句中使用 or 关键字，如果 or 前后有一个列不是索引，索引就不会使用
- 在索引列上进行计算时，会导致索引失效

15. 索引合并和复合索引的区别 *

- ◆ 索引合并：对多个索引分别进行条件扫描，然后将它们各自的结果进行合并，

合并方式分为三种：union, intersection, 以及它们的组合(先内部 intersect 然后在外面 union)。

- ◆ 复合索引：一个索引包含多个列
- ◆ 复合索引会比索引合并快很多，索引合并需要在若干索引表之间跳转，而复合索引不需要跳转，所以速度会更快

16. 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。

17. 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，莫名其妙多出了一些之前不存在的数据，或莫名其妙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

18. 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。

19. 什么是幻读，如何解决 *

- ◆ 一个事务两次执行同一条查询语句，**读取的数据量不一样**，因为第二次读之前，其他事务删除了数据或者新增了数据
- ◆ 解决方法：
 - 第一种，将事务隔离级别设置为串行化

■ 第二种，MVCC-间隙锁

- ◆ MVCC：解决的是快照读的幻读（快照读就是普通的 select 语句【MVCC-undo log】）
- ◆ 间隙锁：解决的是当前读的幻读（当前读就是读取的是最新版本，并且需要先获取对应记录的锁【间隙锁-行锁】）

20. MySQL 是如何保证 ACID 的 **

- ◆ 如果问的是事务的实现原理
 - 第一句话先说，因为只要实现了 ACID 特性，就等于说实现了事务，所以事务的实现原理也就是说 ACID 是怎么实现的
 - ◆ A：利用 undo log 保证原子性。该 log 保存了数据修改之前的一个快照，如果数据修改出现错误，可以回滚，从而保证事务原子性。
 - ◆ D：利用 redo log 保证持久性。该 log 保存了数据修改的操作，数据库系统在重启时，根据 redo log 进行重做，从而使事务有持久性。
 - ◆ C：数据库必须要实现 ACID 三大特性，才有可能实现一致性。
 - ◆ I：利用 MVCC 保证隔离性。

21. MySQL 支持的锁有哪些 *

InnoDB 默认是行级别的锁，当有明确指定的主键时候，是行级锁。否则是表级别

- ◆ 按照数据操作的粒度来划分：
 - 行级锁：粒度最小，只针对当前操作的行进行加锁（MyISAM 不支持）
 - 表级锁：粒度最大，对当前操作的整张表进行加锁
 - 页级锁：粒度介于行级锁和表级锁之间，一次锁定相邻的一组记录（Innodb 不支持）

◆ 按照数据操作的类型来划分：

- 读锁（共享锁 S）：针对同一份数据，多个读操作可以同时进行
- 写锁（排他锁 X）：当前写操作没有完成前，它会阻断其他写锁和读锁
- 意向共享锁：当一个事务给一个数据行加共享锁时，必须先获得表的 IS 锁。
- 意向排他锁：当一个事务给一个数据行加排他锁时，必须先获得该表的 IX 锁。

22. 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。

23. 间隙锁讲一下 *

- ◆ 间隙锁（Gap Lock）是 Innodb 在可重复读隔离级别下为了解决幻读问题时引入的锁机制，在进行范围查询的时候，对于键值在条件范围内但并不存在的记录，叫做间隙，那么 Innodb 也会对这些间隙进行加锁，也就是间隙锁。
- ◆ 为什么需要间隙锁？

- 一个事务进行范围查询的时候，另外一个事务同时进行新增或者删除操作，这时候即使对读取的行加了行锁，也会出现数据不一致的问题，而需要对一定范围内的数据进行加锁，也就是间隙锁，因此就解决了幻读的问题。
- ◆ next-key lock:

- 行锁和间隙锁组合起来就叫 Next-Key Lock
- 默认情况下，InnoDB 工作在可重复读隔离级别下，并且会以 Next-Key Lock 的方式对数据行进行加锁，这样可以有效防止幻读的发生。Next-Key Lock 是行锁和间隙锁的组合，当 InnoDB 扫描索引记录的时候，会首先对索引记录加上行锁（Record Lock），再对索引记录两边的间隙加上间隙锁（Gap Lock）。加上间隙锁之后，其他事务就不能在这个间隙修改或者插入记录。

24. 怎么实现可重复读（读提交）*

- ◆ MVCC
- ◆ 读提交隔离级别是以 select 为单位生成 readview，可重复读是以事务为单位生成 readview。

25. select ... for update *

- ◆ for update 仅适用于 InnoDB，并且必须开启事务，在 begin 与 commit 之间才生效，通过 for update 语句，MySQL 会对查询结果集中每行数据都添加排他锁，其他线程对该记录的更新与删除操作都会阻塞。排他锁包含行锁、表锁。

- ◆ 加行锁还是表锁的一些条件：

- 明确指定主键，并且查询到数据，row lock
- 明确指定主键，但没有查询到数据，无 lock
- 非主键查询，并且查询到数据，table lock
- 非主键查询，但没有查询到数据，table lock

- 只根据主键查询，查询条件为不等于，并且查询到数据，table lock
- 只根据主键查询，查询条件为不等于，但没有查询到数据，table lock
- 只根据主键查询，查询条件为 like，并且查询到数据，table lock
- 只根据主键查询，查询条件为 like，但没有查询到数据，table lock

26. 乐观锁与悲观锁，mysql 如何实现乐观锁 ***

- ◆ 乐观锁：对于数据冲突保持一种乐观态度，操作数据时不会对操作的数据进行加锁，只有到数据提交的时候才通过一种机制来验证数据是否存在冲突。
- ◆ 悲观锁：对于数据冲突保持一种悲观态度，在修改数据之前把数据锁住，然后再对数据进行读写，在它释放锁之前任何人都不能对其数据进行操作，直到前面一个人把锁释放后下一个人才可对数据进行加锁，然后才可以对数据进行操作。 select ... for update
- ◆ 如何实现乐观锁？

- 第一种方式：为表增加一个数字类型的 version 字段来实现。当读取数据时，将 version 字段的值一同读出，数据每更新一次，对此 version 值加一。当我们提交更新的时候，判断表对应记录的当前版本信息与第一次取出来的 version 值进行比对，如果相等，则更新，否则就是数据冲突
- 第二种方式：和第一种方式基本一样，也是在表中增加时间戳类型(timestamp)的一个字段，名称无所谓，当我们提交更新的时候，检查当前表中数据的时间戳和自己更新前取到的时间戳进行对比，如果相等则更新，否则就是数据冲突。

27. MySQL 中常见的几种日志 **

- ◆ 常见的有 7 种日志，分别是 redo log、undo log、binlog、错误日志、慢查询日志、普通查询日志、中继日志
- redo log：它保存了事务发生之后的操作，在重启 mysql 服务的时候，根

据 redo log 进行重做，以此保证事务的持久性

- undo log：它保存了事务发生之前的数据的一个版本，可以用于回滚，以此保证事务的原子性
- bin log：二进制日志记录了所有的 DDL 语句和 DML 语句，但是不包括 DQL 语句，主要用于数据库的数据恢复。包括三种格式：
 - ◆ statement：语句级别，记录每一次写操作的语句（问题：比如执行函数 now()）
 - ◆ row：行级别，记录每次操作后每行记录的变化（问题：占用较大空间；优点是保证数据的绝对一致性）
 - ◆ mixed：statement 级别和 row 级别的混合版本，比如用到 UDF 的时候，采用 row 级别，但是实现比较复杂
- 慢查询日志：记录了在 MySQL 中响应时间超过阈值的语句，这个阈值默认是 10s。
- 中继日志：主从复制架构中，从服务器用于将主服务器的二进制日志中读取到的事件写入到中继日志中
- 错误日志：记录着 mysqld 启动和停止，以及服务器在运行过程中发生的错误的相关信息
- 普通查询日志：记录了服务器接收到的每一个查询或是命令，无论这些查询或是命令是否正确甚至是否包含语法错误

28. MySQL 主从复制的流程 *

- ◆ 主库的记录一旦变化，就会将操作写入到二进制日志（bin log）中
- ◆ 从库会开启一个线程不断读取主库的二进制日志文件，将它拷贝到自己的中继日志中
- ◆ 最后从库会重做中继日志中的事件，就可以将改变的数据同步到自己的数据

库

29. 关系型数据库与非关系型数据库的区别 *

- ◆ 关系型数据库按照表的结构来存储数据的，而非关系型数据库一般基于 k-v 键值对（例如 redis），基于文档（例如 mongodb）等形式来存储数据
- ◆ 非关系型数据库一般只能保证数据的最终一致性（更新后的数据不一定立马能访问，但是最后是能访问到的），而关系型数据库保证数据的强一致性，也就是更新后的数据立马能被访问到
- ◆ 关系型数据库横向扩展比较难，而一些非关系型数据库则原生就支持数据的水平扩展

30. 说一说 drop、delete 和 truncate 的共同点和区别 **

- ◆ 共同点：都表示删除
- ◆ 区别：（美团一面）
 - delete，DML 数据操纵语言，用来删除表的全部数据或者一部分数据，删除的数据可以回滚
 - truncate，DDL 数据定义语言，用来删除表的所有数据，删除的数据不可以回滚
 - drop，DDL 数据定义语言，用来删除表以及所有数据，删除的数据不可以回滚
- ◆ 执行速度：一般来说，drop>truncate>delete
- ◆ 使用场景：在不再需要一张表的时候，用 drop；在想删除部分数据行的时候，用 delete；在想保留表而要删除所有数据的时候用 truncate。

31. 数据库 3 个范式 **

- ◆ 第一范式：每一列属性都是不可再分的属性值 每一列的值都不能再进行

划分

- ◆ 第二范式：在 1NF 基础上消除非主键对主键的部分函数依赖，也就是需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关
消除了部分函数依赖
- ◆ 第三范式：在 2NF 基础上，所有的非主键只依赖于主键，不依赖于其他的非主键（消除了传递函数依赖），也就是需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关
消除了传递函数依赖

32. MySQL 中 char 和 varchar 的区别有哪些 *

补充：*Mysql varchar 字段怎么存储？*

varchar 字段开头包含一个变长字段的实际长度，后面存储的是真实字符。

- ◆ *char* 的长度是不可变的，而 *varchar* 的长度是可变的。因为 *char* 的长度是固定的，所以存储和查找的效率更高
- ◆ *char* 的存储方式是，对英文字符占用 1 个字节，对汉字占用 2 个字节；而 *varchar* 的存储方式是，对英文字符占用 2 个字节，汉字也占用 2 个字节

33. MySQL 中 inner join、left join、right join 和 full join 的区别有哪些 **

- ◆ 内连接：返回两个表的交集
- ◆ 左连接：只要左表存在的都会返回，右表中不存在的补 NULL
- ◆ 右连接：只要右表存在的都会返回，左表中不存在的补 NULL
- ◆ 满连接：返回两个表的并集，左表和右表中不存在的都补 NULL

34. MySQL 的执行顺序 *

- ◆ from > join > where > group by > having > order by > select

35. having 和 where 的区别 *

- ◆ where 是在结果返回之前进行过滤的，而 having 是在结果返回之后进行过滤的
- ◆ where 后面不能使用聚合函数，而 having 后面可以使用聚合函数
- ◆ where 用在 group by 的前面，而 having 用在 group by 的后面

36. MySQL 的存储引擎，以及区别 *

- ◆ InnoDB 和 MyISAM 区别：
 - 存储文件
 - ◆ MyISAM 每个表有两个文件【MYD 和 MYI 文件】，其中 MYD 是数据文件，MYI 是索引文件
 - ◆ InnoDB 每个表只有一个 idb 文件
 - InnoDB 支持事务，支持行级锁【MyISAM 支持表级锁】，支持外键
 - InnoDB 支持 XA 事务
 - InnoDB 支持 savePoints

37. 大数据量里的分页查询怎么优化 *

- ◆ SELECT * FROM 表名称 LIMIT M, N 这种方法只适合数据量较少的情况
 - M: offset; N: 长度
- ◆ 优化：
 - ◆ 利用表的覆盖索引，即查询的列中，只包含了索引列
 - ◆ 上面这种方法只能查询索引列，如果想查询所有的列怎么办呢？

- 可以通过主键索引，把 limit 放在子查询中，主键作为判断条件

38. SQL 的优化方法 *

- ◆ 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引
- ◆ 应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描
- ◆ 对慢查询进行优化：看是否查询了不需要的列，查询条件是否没有命中索引

39. MySQL 中视图和表的区别 *

- ◆ 视图只是一个或多个表依照某个条件组合而成的结果集，没有实际的物理记录。
- ◆ 视图的建立和删除只影响视图本身，不影响对应的基本表。
- ◆ 视图只能进行 select 操作。
- ◆ 视图是外模式，而表是内模式。

40. 数据完整性约束 **

- ◆ 实体完整性：主键不能为空
- ◆ 参照完整性：外键要么是关联的表的主键，要么为空值
- ◆ 用户自定义的完整性：比如唯一约束，非空约束

41. group by 的实现方式 *

1. 松散索引：当 MySQL 完全利用索引扫描来实现 groupby 的时候，并不需要扫描所有满足条件的索引键即可完成操作，也就是满足最左匹配原则就可以了
2. 紧凑索引：如果 group by 后面的不符合最左匹配，但是 where 语句后面有对缺失的

索引键进行了常量的限制（`user_id=1`），那么就会走紧湊索引。

3. 临时文件：如果上述 `where` 不是进行的常量限制，那么就无法利用索引，只能先将数据存入临时表，然后进行排序和分组操作，完成 `groupby`

数据结构

1. 链表和数组的区别

- 数组中的元素在内存中连续存放，而链表中的元素在内存中不连续
- 数组可以通过下标快速访问任何元素，而链表则需要遍历整个链表才能找到对应的元素
- 数组不适合插入或者删除元素，因为插入或者删除元素，就需要移动大量的元素，而链表插入或者删除元素，只用修改元素的指针就可以了
- 补充：静态链表和动态链表的区别？
 - ◆ 静态链表的元素在内存中连续存放；而动态链表中的元素在内存中不连续
 - ◆ 静态链表的初始长度一般是固定的；而动态链表是动态申请内存的方式，所以长度是不固定的
 - ◆ 静态链表的插入删除仅需要修改指针，而数组需要移动元素（数组和静态链表的区别）

2. 栈和队列的区别

- 区别：
 - ◆ 栈是先进后出，队列是先进先出
 - ◆ 栈只允许在表尾进行插入和删除，而队列只允许在表尾插入，在表头删除

■ 联系：

- ◆ 都是线性结构
- ◆ 插入删除时间复杂度都是 $O(1)$

■ 补充：

- ◆ 主要分为线性数据结构和非线性数据结构
- ◆ 线性数据结构包括 数组、链表、栈和队列；非线性数据结构包括 哈希表、树、堆和图。

3. 红黑树了解吗

- 红黑树是一棵二叉排序树，我们知道平衡二叉树也是一颗二叉排序树，他们之间的区别是 红黑树不要求所有节点的左右子树高度差不超过 1，只要求从一个节点到所有叶节点的路径中，最长路径不超过最短路径的两倍，所以红黑树只追求树的大致平衡
- 因为对树平衡程度的不同要求，平衡二叉树在插入和删除的过程中会花费比较大的代价来维护树的平衡，所以平衡二叉树不适合插入，删除太多的场景。而红黑树只要求弱平衡，它做到了当插入和删除时，只需最多旋转 3 次就能实现一定程度的平衡，所以将查询，插入和删除的时间复杂度维持在 $O(\log n)$
- 红黑树的性质：
 - ◆ 红黑树中节点分为黑色节点和红色节点
 - ◆ 根节点和所有叶子结点（NIL 节点）是黑色
 - ◆ 每个红色结点的两个子结点都是黑色（红色特性）
 - ◆ 从任一节点到所有叶子结点的路径上的黑色节点数量相同（黑色特性）
- 调整：先变色解决-左旋/右旋

补充：

- ◆ 平衡二叉树：满足 二叉排序树，并且所有节点的左右子树高度差不超过 1
(平衡因子就是 节点的左子树高度-节点的右子树高度)

4. 常见的排序算法

- ◆ 冒泡排序： 稳定 $O(N^2)$
- ◆ 快速排序： 不稳定 $O(N \log N)$
- ◆ 简单插入排序： 稳定 $O(N^2)$
- ◆ 希尔排序： 不稳定 $O(N \log N) \sim O(N^2)$
- ◆ 简单选择排序： 不稳定 $O(N^2)$
- ◆ 堆排序： 不稳定 初始建堆的时间复杂度是 $O(N)$ ；排序重建堆的时间复杂度为 $O(N \log N)$ ；最终的时间复杂度为 $O(N \log N)$
- ◆ 归并排序： 稳定 $O(N \log N)$
- ◆ 桶排序： 取决于桶内排序算法
- ◆ 基数排序： 稳定 $O(N * M)$ 其中 N 为数据个数， M 为数据位数

5. 图的常见算法

- ◆ 图的遍历方式：
 - 深度优先遍历（dfs）：找到一个起始点，然后找该点的第一个没有被访问到的儿子，访问该点，然后找该点第一个没有被访问到的儿子……，按着一条线一直往下走就好
 - 广度优先遍历（bfs）：找到一个起始点，然后找该点的所有儿子，依次访问所有的儿子，然后依次访问他儿子的所有儿子……，类似于层序遍历
- ◆ 图的最短路径算法：

- **迪杰斯特拉算法（单源最短路径 $O(N^2)$ ）：**从顶点 A 开始遍历，在图中找到一个满足 $A \rightarrow v_i$ 最短的顶点 v_i .更新由 A 经过 v_i 再到 v_j 的最短距离，即如果 A 到 v_i 再到 v_j ,则更新 $A \rightarrow v_j = A \rightarrow v_i + v_i \rightarrow v_j$ ，依次循环
- **弗洛伊德算法（多源最短路径 $O(N^3)$ ）：**对于每一对顶点 u 和 v ，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短，如果是就更新它的最短路径。所以会套上三重循环，来遍历 w, u, v
- **贝尔曼福特算法（单源最短路径【带负权边】 $O(\text{顶点数} * \text{边数})$ ）：**首先第一层枚举点（松弛操作：最多经过 k 条边），第二层枚举每一条边

第四部分 数仓基础

1. 数据仓库是什么

- 数据仓库是一个**面向主题的**（司机、乘客、订单）、**集成的**（来自不同数据源的统一数据规范比如男女的取值，命名规范的统一，字段类型的统一）、**非易失**（一般不会进行删除和修改操作）且**随时间变化**（并不是数据会变，而是数据随着时间会不断增多）的数据集合，主要用于存储历史数据，然后通过分析整理进而提供数据支持和辅助决策。

2. 数据仓库和数据库有什么区别

- 数据库中主要存放的是一些在线的数据，数据仓库中主要存放的是历史数据，并且存放的数据要比数据库多
- 数据库主要用于业务处理（比如交易系统），数据仓库主要用于数据分析
- 数据库的设计就是要避免冗余，而数据仓库通常会专门引入冗余，减少后面进行分析时大量的 join 操作

3. 为什么要对数据仓库分层

- 第一个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，比较容易理解和理解
- 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！

4. 为什么需要数据建模

爆发式的数据增长，如何对数据进行有序、有结构地分类组织和存储是一大挑战。数据模型是数据存储和组织方法，它强调从业务、数据存取和使用角度合理存储数据。有了适合业务的数据模型之后，那么大数据就可以获得以下好处：性能、成本、效率、质量

5. 经典的数据仓库建模方法论有哪些

- 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

6. 数仓相关的名词术语解释，比如数据域、业务过程、衍生指标

- 数据域：将业务过程或者维度进行抽象的集合，例如交易域、商品域等都是数据域
- 业务过程：一个不可拆分的行为事件，例如下单、支付、退款等都是业务过程

- 维度：用于分析事实所需要的多样环境/角度，例如时间、地理等都是维度
- 维度属性：维度属性隶属于一个维度，例如地理维度中的国家名称、国家 ID、省份名称等都是维度属性
- 原子指标：业务中定义的不可拆分的指标，用来度量某一行为事件，例如支付金额、下单数目等都是原子指标

7. 派生指标的种类

- 事务型指标：对业务活动进行衡量的指标。例如订单支付金额、新发商品数、新增注册会员数
- 存量型指标：对实体对象（比如商品、会员）某些状态的统计。例如商品总数、会员总数等，这类指标需维护原子指标及修饰词，在此基础上创建衍生指标，对应的时间周期一般为“截止当前某个时间”
- 复合型指标：在事务型指标和存量型指标的基础上复合而成的。通常为比率型或者比例型，例如 UV-下单买家数转化率，最近一天无线支付金额占比等
- 派生指标：就是对原子指标统计范围的限定，形式为一个原子指标+多个修饰词【可选】+时间周期，例如最近一天通过花呗支付的金额（原子指标：支付金额，修饰词：花呗支付，时间周期：最近一天）

8. 经典数仓分层架构

- 操作数据层（ODS）：把业务系统、日志等数据几乎无处理地同步到 ODS 层中
- 公共维度模型层（CDM）：存放明细事实数据、维表数据及公共指标汇总数据，其中明细事实数据和维表数据一般由 ODS 加工生成；公共指标汇总数据一般根据明细事实数据和维表数据加工生成
 - ◆ 明细数据层（DWD）
 - ◆ 汇总数据层（DWS）
- 应用数据层（ADS）：存放个性化的统计指标数据，一般根据 ODS 和 CDM 加工生成

9. 模型设计的基本原则

- 高内聚和低耦合：将业务相关、粒度相同的数据设计为同一个物理模型，将高频率同时访问的数据放一起，将低频率同时访问的数据分开存储

- 核心模型与扩展模型分离：核心模型包括的字段支持常用的核心业务，扩展模型包括的字段支持个性化或少量应用的需要
- 公共处理逻辑下沉：越是公用的处理逻辑，越应该在数据调度依赖的底层进行封装与实现，不要让公共逻辑多处同时存在
- 成本与性能平衡：适当的数据冗余可以换取查询和刷新性能，但不要过度冗余
- 数据可回滚：处理逻辑不变，在不同时间多次运行，数据结果确定不变
- 一致性：具有相同含义的字段在不同的表中命名必须相同
- 命名清晰可理解：表名需易于消费者理解和使用

10. 模型实施的具体步骤

1. 数据调研
 - i. 业务调研：以阿里为例，整个阿里集团涉及的领域包括电商、导航、本地生活等领域。各个领域又包括多条业务线，比如电商包括了淘宝、天猫、天猫国际等。那么是各个业务领域单独建设数仓还是一起建设数仓仍然是一个问题，在阿里，一般各个业务领域单独建设数仓，因为业务领域内的业务线由于业务相似、业务相关性较大，可进行统一集中建设
 - ii. 需求调研：通常包括两种途径，一是根据与分析师、业务运营人员沟通获知需求；二是对报表系统中的现有报表进行分析。比如说，分析师需要了解 大淘宝一级类目的成交金额。当获知这个需求后，我们需要分析根据什么汇总（维度），以及汇总什么（度量），这里类目是维度，成交金额是度量；明细数据和汇总数据应该怎么设计？这是一个公用的报表吗？是需要沉淀到汇总表里面，还是在报表工具中进行汇总？
2. 架构设计
 - i. 数据域划分：数据域需要抽象提炼，并且长期维护和更新，但不经常变动。在划分数据域的时候，既能涵盖当前所有的业务需求，又能在新业务进入的时候无影响地被包含进已有的数据域中或者扩展新的数据域。比如会员域、商品域、店铺域、日志域、交易域等
 - ii. 构建总线矩阵：需要做两件事情，一是明确每个数据域下有哪些业务过程，二是确定业务过程与哪些维度相关。
3. 规范定义：定义指标体系，包括原子指标和派生指标
4. 模型设计：包括明细事实表、维度表以及汇总事实表的模型设计
5. 代码开发
6. 部署运维

11. 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。

12. 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。

13. 维度表的设计过程

以淘宝的商品维度为例来对维度设计方法进行详细说明

第一步：选择维度。作为维度建模的核心，在企业级数仓中必须保证维度的唯一性。即为商品维度

第二步：确定主维表。一般是 ODS 表，直接与业务系统同步。以淘宝商品维度为例，`s_auction_auctions` 是与前台商品中心系统同步的商品表，即为主维表

第三步：确定相关维表。数据仓库是业务源系统的数据整合，不同业务系统或者同一业务系统中的表之间存在关联性。根据对业务的梳理，确定哪些表与主维表相关联。以淘宝商品维度为例，与类目、SPU、卖家、店铺等维度存在关联

第四步：确定维度属性。包括两个步骤，一是从主维表中选择维度属性或者生成新的维度属性，二是从相关维表中选择维度属性或者生成新的维度属性

14. 维度表的设计中有哪些值得注意的地方

- 尽可能生成丰富的维度属性
- 尽可能详细的对维度属性进行文字解释
- 区分数值型属性和事实
 - ◆ 数值型字段是作为事实还是维度属性，可以参考字段的一般用途。如果通常用于查询约束条件或者分组统计，则是作为维度属性；如果通常用于参与度量的计算，则是作为事实。
- 尽量沉淀通用的维度属性
 - ◆ 例如，淘宝商品中的 property 字段，使用 k:v 存储了多个商品属性。商品品牌就存放在该字段中，而商品品牌是重要的分组统计和查询约束的条件，所以需要将该字段解析出来
 - ◆ 例如，商品是否在线，是重要的查询约束条件，但是无法直接获取，需要进行加工，所以需要封装商品是否在线逻辑作为一个单独的属性字段

15.维表整合的两种表现形式

1. 垂直整合：不同的来源表包含相同的数据集，只是存储的信息不同。比如淘宝会员在源系统中有多张表，如会员基础信息表、会员扩展信息表、淘宝会员等级信息表等，这些表都属于会员信息表，依据维度设计方法，尽量整合至会员维度模型中，丰富其维度属性
2. 水平整合：不同的来源表包含不同的数据集。比如针对蚂蚁集团的数据仓库，其采集的会员数据有 淘宝会员、1688 会员、支付宝会员等，是否需要考虑将所有的会员整合到一个会员表中？如果进行整合，首先需要考虑各个会员体系是否有交叉，如果有交叉，则需要去重；如果不存在交叉，则需要考虑不同子集的自然键是否存在冲突，如果不冲突，则可以考虑将各子集的自然键作为整合后表的自然键；另一种方式是 设置超自然键，将来源表各子集的自然键加工成一个字段作为超自然键

16.如何处理维度的变化

背景：在现实世界中，维度的属性并不是静态的，它会随着时间的流逝发生缓慢的变化，与数据增长较为快速的事实表相比，维度变化相对缓慢，所以将这类维度叫做**缓慢变化维**

处理缓慢变化维的方式：

1. 重写维度值。无法保留历史数据
2. 插入新的维度行。虽然能够保留历史数据，但是 不能将变化前后记录的事 实归一化为变化前的维度或者变化后的维度。比如根据业务需求，需要将 11 月份的交易额全部统计到类目 2 上，该方式无法实现。
3. 添加维度列。保留历史数据
4. 快照维表。每天保存一份全量快照数据。
 - a) 优点：简单有效，开发和维护成本低；使用方便，理解性好，数据使用 方只需要限定日期，即可获取到当天的快照数据。任意一天的事实快照 和维度快照通过维度的自然键进行关联即可。
 - b) 缺点：存储的浪费。比如某维度，每天的变化量占总体数据量比例很低， 甚至无变化，那么全量保存就非常浪费存储空间
5. 拉链表。通过新增两个时间戳字段（start_dt 和 end_dt），将所有以天为 粒度的变更数据记录下来。
 - a) 背景：2016 年 1 月 1 日，卖家 A 在淘宝网发布了 B、C 两个商品，1 月 2 日，卖家 A 在淘宝网下架了商品 B，同时又发布了商品 D
 - b) 快照维表：**dt 是分区字段**
 - c) 拉链表：对于不变的数据，不再重复存储
 - i. 例如，用户访问 1 月 2 日的数据，只需要限制 start_dt<=20160102 和 end_dt>20160102 即可
 - ii. 缺点：理解性较差；存储方式用 start_dt 和 end_dt 做分区，随着 时间的推移，分区数量会极度膨胀，而现行的数据库系统对分区都 有所限制。
 - d) 为了解决上述两个问题，阿里巴巴提出采用极限存储的方式来进行处理
 - 透明化：底层的数据还是历史拉链存储，但是上层做一个视图操作 或者在 Hive 里做一个 hook。这样对于下游用户来说，极限存储表 和全量存储表的访问方式是一样的
 - 分月做历史拉链表
 - 假设用 start_dt 和 end_dt 做分区，并且不做限制，那么可以计 算出一年历史拉链表最多可能产生的分区数是：
 $365*364/2=66430$ ；如果每个月月初重新开始做历史拉链表，那 么可以计算出一年历史拉链表最多可能产生的分区数是：
 $12*(1+30*29/2)=5232$
 - 采用极限存储的存储方式，极大地压缩了全量存储的成本，又可 以达到对下游用户透明的效果，是一种比较理想的存储方式
 - ◆ 为什么企业中很少使用这种方式呢？
 - 产出效率很低，通常需要 t-2

- 对于变化频率高的数据并不能达到节约成本的效果

17. 事实表设计的八大原则

1. 尽可能包含所有与业务过程相关的事实
2. 只选择与业务过程相关的事实
3. 分解不可加事实为可加事实
4. 在选择维度和事实之前必须先声明粒度
5. 在同一个事实表中不能有不同粒度的事实
6. 事实的单位要保持一致
7. 对事实的 null 值要处理
8. 使用退化维度提高事实表的易用性

18. 事实表的设计过程

第一步：选择业务过程以及确定事实表类型

业务过程通常使用行为动词表示业务执行的活动，比如淘宝交易订单流转的业务过程有四个：创建订单、买家付款、卖家发货、卖家确认收货。在明确了流程所包含的业务过程之后，需要根据具体的业务需求来选择与维度建模有关的业务过程。

在选择了业务过程以后，相应的事实表类型也随之确定了。比如选择买家付款这个业务过程，那么事实表为 只包含买家付款这一业务过程的单事务事实表；如果选择的是所有四个业务过程，并且需要分析各个业务过程之间的时间间隔，那么事实表为 包含四个业务过程的累计快照事实表。

第二步：声明粒度

明确的粒度能确保对事实表中行的意思的理解不会产生混淆，保证所有的事实按照同样的细节层次记录。

应该尽量选择最细级别的原子粒度，以确保事实表的应用具有最大的灵活性。

第三步：确定维度

完成粒度声明以后，也就意味着确定了主键，对应的维度组合以及相关的维度字段就可以确定了，应该选择能够描述清楚业务过程所处的环境的维度信息。比如

在淘宝订单付款事务事实表中，粒度为子订单，相关的维度有 买家、卖家、商品、收货人信息、业务类型、订单时间等

第四步：确定事实

事实可以通过回答"过程的度量是什么"来确定。比如在淘宝订单付款事务事实表中，同粒度的事实有 子订单分摊的支付金额、邮费、优惠金额等

第五步：冗余维度

在传统的维度建模的星型模型中，对维度的处理是需要单独存放在专门的维表中的，通过事实表外键获取维度。这样做的目的是为了减少冗余，从而减少存储消耗。而在大数据的事实表模型设计中，考虑更多的是下游用户的使用效率，降低数据获取的复杂性，减少关联的表数量。所以通常会在事实表中冗余下游用户经常使用的维度。比如在淘宝订单付款事实表中，通常冗余的大量常用维度字段有商品类目、卖家店铺等

19.事实表有哪几种类型

- 事务事实表：用来描述业务过程，跟踪空间或时间上某点的度量事件，保存的是最原子的数据
- 周期快照事实表：以具有规律性的、可预见的时间间隔记录事实，时间间隔如每天、每月、每年等。当需要一些状态变量时，比如账户金额、买卖家星级、商品库存、卖家累计交易额等，则需要聚集与之相关的事务才能进行识别计算，往往和事务事实表成对出现。
- 累计快照事实表：用来表述过程开始和结束之间的关键步骤事件，覆盖过程的整个生命周期，通常有多个日期字段来记录关键时间点，当过程随着生命周期不断变化时，记录也会随着过程的变化而修改

20.多事务事实表如何对事实进行处理

主要有两种方法对事实进行处理

1. 不同业务过程的事实使用不同的事实字段进行存放。比如淘宝交易事务事实表，表中会设置 下单度量，支付度量，完结度量等字段。

2. 不同业务过程的事实使用同一个事实字段进行存放，但增加一个业务过程标签。比如收藏事务事实表，表中会设置 收藏删除类型，以及收藏删除度量等字段。

关于上述两种方法如何选择呢？

- 当不同业务过程的度量比较相似时，采用第二种方式；反之，当不同业务过程的度量差异比较大时，采用第一种方式

21.单事务事实表和多事务事实表哪种设计更好

主要从五个方面来进行分析

- 业务过程
 - 对于单事务事实表，一个业务过程建议一张事实表，只反映一个业务过程的事实；对于多事务事实表，在同一个事实表中反映多个业务过程的事实。多个业务过程是否放到同一张事实表中，首先需要分析不同业务过程之间的相似性。
- 粒度和维度
 - 在确定好业务过程后，需要基于不同的业务过程确定粒度和维度，当不同业务过程的粒度相同，同时拥有相似的维度时，此时就可以考虑采用多事务事实表。如果粒度不同，则必定是不同的事实表。比如交易中 支付和发货有不同的粒度，则无法将发货业务过程放到淘宝交易事务事实表中
- 事实
 - 如果单一业务过程的事实较多，同时不同业务过程的事实又不相同，则可以考虑单事务事实表，处理更加清晰；若使用多事务事实表，则会导致事实表零值或空值较多
- 下游业务使用
 - 单事务事实表对于下游用户更容易理解，关注哪个业务过程就使用哪张事实表；而多事务事实表包含多个业务过程，用户使用往往较为困惑。
- 计算存储成本
 - 当业务过程数据来源于同一个业务系统，具有相同的粒度和维度，且维度较多而事实不多时，此时可以考虑多事务事实表，不仅其加工计算成本较低，同时在存储上也相对节省

22.周期快照事实表的设计过程

第一步，确定粒度。采样周期为每天，针对卖家、买家、商品、类目、地区等

维度的快照事实表，比如淘宝卖家历史至今汇总事实表、淘宝商品自然月至今汇总事实，不同的采样粒度确定了不同的快照事实表。

第二步，确定状态度量。确定好粒度以后，就要针对这个粒度确定需要采样的状态度量。比如淘宝卖家历史至今汇总事实表，包含了历史截至当日的下单金额、历史截至当日的支付金额等度量

23. 累计快照事实表的设计过程

第一步，选择业务过程。淘宝交易订单的流转主要包括 下单，支付，发货，完成这四个业务过程，在事务统计中，只关注了下单、支付、完成这三个业务过程，而在统计事件时间间隔的需求中，卖家发货也是关键环节。所以在针对淘宝交易累计快照事实表，我们选择这四个业务过程

第二步，确定粒度。子订单在此表中只记录一行，事件发生时，对此实例进行更新

第三步，确定维度。与事务事实表相同，维度主要有买家、卖家、店铺、商品、类目、地区等。四个业务过程对应的时间字段，分别为下单时间、支付事件、发货时间、确认收货时间。

第四步，确定事实。对于累计快照事实表，需要将各个业务过程的事实均放入事实表中。比如淘宝交易累计快照事实表，包含了各业务过程对应的事，如下单金额、支付对应的折扣、邮费和支付金额、确认收货对应的金额等。累计快照事实表解决的最重要的问题是 统计不同业务过程之间的时间间隔，建议将每个过程的时间间隔作为事实放在事实表中。

第五步，退化维度。与事务事实表相同。

24. 累计快照事实表的特点

1. 数据不断更新
 1. 事务事实表记录事务发生时的状态，对于实体的某一实例不再更新；而累积快照事实表则对实体的某一实例定期更新。
2. 多业务过程日期
 1. 累计快照事实表适用于具有明确起止时间的短生命周期的实体，比如交易订单、物流订单等，对于实体的每一个实例，都会经历从诞生到消亡等一系列步骤。对于商品、用户等具有长生命周期的实体，一般采用周期快照事实表更合适
 2. 累计快照事实表的典型特征就是多业务过程日期，用于计算业务过程之间的时间间隔。还有一个重要作用是保存全量数据。

第五部分 常考 SQL

1. 连续问题 ***

问题：如下数据为蚂蚁森林中用户领取的减少碳排放量，找出连续 3 天及以上减少碳排放量在 100 以上的用户

```
id dt lowcarbon
1001 2021-12-12 123
1002 2021-12-12 45
1001 2021-12-13 43
1001 2021-12-13 45
1001 2021-12-13 23
1002 2021-12-14 45
1001 2021-12-14 230
1002 2021-12-15 45
1001 2021-12-15 23
```

答案：

```
select id
from (
    select
        id,
        dt,
        date_sub(dt, row_number() over(partition by id order by dt) rk) as diff
    from (
        select id, dt, sum(lowcarbon) lowcarbon
        from test1
        group by id, dt
        having lowcarbon > 100
    ) t
) t
group by id, diff
```

```
having count(*) >= 3
```

2. 分组问题 **

问题：如下为电商公司用户访问时间数据，某个用户连续的访问记录如果时间间隔小于 60 秒，则分为同一个组

```
id ts(秒)
1001 17523641234
1001 17523641256
1002 17523641278
1001 17523641334
1002 17523641434
1001 17523641534
1001 17523641544
1002 17523641634
1001 17523641638
1001 17523641654
-- 输出
id ts(秒) group
1001 17523641234 1
1001 17523641256 1
1001 17523641334 2
1001 17523641534 3
1001 17523641544 3
1001 17523641638 4
1001 17523641654 4
1002 17523641278 1
1002 17523641434 2
1002 17523641634 3
```

答案：

```
select
-- 这种就可以求出两两之间登陆的时间差小于 60 就可以分为同一组的问题
```

```

-- 从首行累加到当前行，如果某行大于 60，那么就将分组 id 加 1，否则不变
id, ts,
sum(if(diff >= 60, 1, 0)) over(partition by id order by ts) groupid
from (
    select
        id,
        ts,
        -- 将 ts 上移一行，求出相邻记录之间的时间差值
        ts - lag(ts, 1, 0) over(partition by id order by ts) diff
    from test2
) t

```

3. 间隔连续问题 *

问题：某游戏公司记录的用户每日登录数据，计算每个用户最大的连续登录天数，可以间隔一天。解释：如果一个用户在 1,3,5,6 登录游戏，则视为连续 6 天登录。

id	dt
1001	2021-12-12
1002	2021-12-12
1001	2021-12-13
1001	2021-12-14
1001	2021-12-16
1002	2021-12-16
1001	2021-12-19
1002	2021-12-17
1001	2021-12-20

思路：将间隔一天的数据分到一个组里面，然后取每个用户同一个组的最大日期
-最小日期+1

答案：

```

select
    id, max(dt) - min(dt) + 1

```

```

from (
    select
        id, dt,
        sum(if(diff > 2, 1, 0)) over(partition by id order by dt) groupid
    from (
        select
            id, dt,
            date_sub(dt, lag(dt, 1, 0) over(partition by id order by dt)) diff
        from test3
    ) t
) t
group id, groupid

```

4. 打折日期交叉问题 **

问题：如下为平台商品促销数据：字段为品牌，打折开始日期，打折结束日期，计算每个品牌的打折销售天数，注意其中的交叉日期

brand	stt	edt
oppo	2021-06-05	2021-06-09
oppo	2021-06-11	2021-06-21
vivo	2021-06-05	2021-06-15
vivo	2021-06-09	2021-06-21
redmi	2021-06-05	2021-06-21
redmi	2021-06-09	2021-06-15
redmi	2021-06-17	2021-06-26
huawei	2021-06-05	2021-06-26
huawei	2021-06-09	2021-06-15
huawei	2021-06-17	2021-06-21

答案：

```

select
    brand, sum(if(days >= 0, days + 1, 0)) days
from (

```

```

select
    brand, datediff(edt, stt) days
from (
    select
        brand,
        -- 将当前行前面的最大终止时间比自己开始时间大的，开始时间
        替换为最大终止时间 + 1
        if (maxEdt is null, stt, if(stt > maxEdt, stt, date_add(maxEdt, 1))) stt,
        edt
    from (
        select
            brand, stt, edt,
            -- 求出当前行的前面所有行的终止时间的最大值
            max(edt) over(partition by brand order by edt rows between
unbounded preceding and 1 preceding) as maxEdt
        from test4
    ) t
) t
) t
group by brand

```

5. 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

...

答案：

```
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
```

6. 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;
```

7. 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<='20211124'
    group by uid, dt
) t2
on t1.uid=t2.uid
group by t1.dt
```

第六部分 大数据开发场景题

1. 1亿个整数中找出最大的 10000 个数 **

- 首先想到的就是**全局排序**, 那么需要判断内存是否能够装的下? $1*10^9*4B = 4GB$, 需要 4G 的内存, 如果机器的内存小于 4G, 显然是不行的
- 第二种方法就是**分治法**, 将这 1亿个数通过 hash 算法, 分为 1000 份, 每份 100 万个数据, 找到每份数据中最大的 10000 个数, 最后在 $100*10000$ 中找出最大的 10000 个数, 最大占用内存为 $1000000*4B = 4MB$ 。**从 100 万个数中找到最大的 10000 个数的方法是快速排序的方法**, 但是我们没有必要将这 100 万个数排序, **只用找到前 10000 个数**, 大致的思路是: 将第一个数字设置为基准元素, 然后将这 100 万个数分为两堆, 如果大于基准的堆的个数大于 10000 个, 那么继续对该堆进行一次快速排序, 如果此时大的堆的个数 n 小于 10000 个, 那么在小的堆中找到前 $10000-n$ 的数字。
- 第三种方法就是**小顶堆**, 首先**读入前 10000 个数来创建大小为 10000 的最小堆**, 建堆的时间复杂度是 $O(m)$, 然后遍历后续的数字, 并与堆顶元素(最小)进行比较, **如果比堆顶元素小, 则继续遍历后面的数字即可; 如果比堆顶元素大, 则替换堆顶元素并重新调整堆为最小堆**。直至遍历完所有的数字, 最后输出当前堆的所有数字就可以了。整体的时间复杂度是 $O(mn)$

2. 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

- 考虑全部加载到内存中, 需要 $2 * 50 * 10^9 * 64B = 6400 \text{ GB}$, 显然不可能
- 那么可以考虑**分治法**, 步骤如下
 - 遍历文件 a, 对每个 url 求取 $\text{hash(url)} \% 1000$, 然后根据所取得的值**将 url 分别存储到 1000 个小文件 ai**
 - 遍历文件 b, 采取和 a 相同的方式**将 url 分别存储到 1000 个小文件 bi**, 这样处理后, 所有可能相同的 url 都被保存在对应的小文件中, 不对应的小文件大概率不可能有相同的 url

- 然后把 ai 的 url 存储到 hash_set 中，并且遍历 bi 的每个 url，看其是否在刚才构建的 hash_set 中出现过，如果出现过，那么就是共同的 url，返回即可

3. 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

- 遍历文件，对每个词 hash(x) %5000，然后根据算的值将单词分别存储到 5000 个小文件中，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。
- 对每个小文件，利用 hashmap 先统计每个词出现的频数，然后建小顶堆的方法，取出频数最高的 100 个单词，然后把单词以及频数写入文件中
- 最后把新得到的 5000 个小文件，进行归并排序即可

4. 在 2.5 亿个整数中找出不重复的整数 **

- 采用 2bit 的 bitmap (00 表示不存在，01 表示出现 1 次，10 表示出现多次，11 表示无意义)，共需内存 $2^{32} \times 2\text{bit} = 1\text{GB}$ 内存，可以接受，然后扫描这 2.5 亿个整数，查看 bitmap 中相对应的位，如果是 00 变 01，01 变 10，10 保持不变，扫描完后，查看 bitmap，把对应为是 01 的整数输出。

5. 外部排序

- 定义：指大文件的排序，即待排序的记录存储在外存储器上，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的
- 步骤：一般来说外部排序分为两个步骤，预处理和合并排序。首先，根据可用内存的大小，将外存上含有 n 个记录的文件分成若干长度为 t 的子文件，其次，利用内部排序的方法（快排），对每个子文件的 t 个记录进行内部排序，这些经过排序的子文件叫做顺串，顺串生成后即将其写入外存，这样在外存上就得到了 m 个顺串，最后，将这些顺串进行归并，使顺串的长度逐渐增大，直到所有的待排序的记录成为一个顺串为止

- 对于外排来说，影响整体排序效率的因素主要取决于读写外存的次数。

6. 大数量中寻找中位数【网易】 **

- 我们可以将整个数据分成两部分，分别用两个数据容器存储。如果能保证数据容器左边的数据都小于右边的数据，那么即使左、右两边内部的数据没有排序，也可以根据左边最大的数及右边最小的数得到中位数。**我们可以用最大堆来实现左边的数据容器，因为位于堆顶的就是最大的数据。同样，也可以用最小堆来实现右边的数据容器。**

7. 大整数相乘【网易】

- 分治法【递归】

- 当我们输入两个大整数 num1 , num2 , 长度分别为 n , m , 计算机无法直接计算其结果，采用分而治之的思想，我们可以分别将两个数均分为四个部分，记作 A, B, C, D, 其中：A 为 num1 的前 $n/2$, B 为 num1 的后 $n/2$, C 为 num2 的前 $m/2$ D 为 num2 的后 $m/2$
- $\text{num1} * \text{num2} = (A * 10^{(n/2)} + B) * (C * 10^{(m/2)} + D) = AC * 10^{(n+2+m)/2} + AD * 10^{(n/2)} + BC * 10^{(m/2)} + BD$
- 此时将一个大整数的乘法分为了四个相对较小的整数相乘，此时，我们就可以采用递推的思想，再将每个相乘的数进行拆分相乘再相加，当递推到相乘的值较小时，我们可以求出相应的值然后对其进行相加，将结果依次返回，最终得到大整数相乘的结果。

- 乘法变加法【遍历】

- 同字符串相乘

8. 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： $\text{hashCode}(\text{key}) \% \text{reduce 个数}$
 - 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】

第七部分 大厂面经合集 (持续更新~)

美团

30 篇面经合集

面试部门	面经发布日期	面经链接
到店	2020.08.19	https://www.nowcoder.com/discuss/482359
到店	2021.04.23	https://www.nowcoder.com/discuss/647394
美团优选	2020.08.25	https://www.nowcoder.com/discuss/487574
美团优选	2021.03.31	https://www.nowcoder.com/discuss/628767
美团优选	2021.04.06	https://www.nowcoder.com/discuss/633189
美团优选	2021.04.28	https://www.nowcoder.com/discuss/650562
广告部	2020.11.30	https://www.nowcoder.com/discuss/573239
基础架构部	2021.03.29	https://www.nowcoder.com/discuss/627093
基础架构部	2021.04.01	https://www.nowcoder.com/discuss/630021
基础架构部	2021.03.22	https://www.nowcoder.com/discuss/620701
	2023.03.28	https://www.nowcoder.com/discuss/470533474290618368
	2023.04.03	https://www.nowcoder.com/feed/main/detail/f4ba6aa8303c4bb3af292bd78d78b5c6
	2023.04.12	https://www.nowcoder.com/discuss/475594456176123904
	2023.03.25	https://www.nowcoder.com/feed/main/detail/4bcea290d27b4

		a8ab18a85a6a383f0cc
到家	2023.04.02	https://www.nowcoder.com/discuss/471998941211480064
	2021.03.29	https://www.nowcoder.com/discuss/627275
	2021.03.30	https://www.nowcoder.com/discuss/627844
	2021.04.09	https://www.nowcoder.com/discuss/635208
	2021.04.13	https://www.nowcoder.com/discuss/638417
	2021.04.15	https://www.nowcoder.com/discuss/639369
	2021.04.20	https://www.nowcoder.com/discuss/644590
	2021.04.23	https://www.nowcoder.com/discuss/646936
	2021.04.23	https://www.nowcoder.com/discuss/647288
	2020.08.21	https://www.nowcoder.com/discuss/484810
	2021.04.26	https://www.nowcoder.com/discuss/649034
	2021.03.19	https://www.nowcoder.com/discuss/618597
	2020.08.26	https://www.nowcoder.com/discuss/489606
	2020.08.31	https://www.nowcoder.com/discuss/493639
	2020.10.10	https://www.nowcoder.com/discuss/534762
	2021.03.25	https://www.nowcoder.com/discuss/623104

面经 1

一面

- 1.自我介绍
- 2.http 协议是哪一层的协议，讲一下对 http 的了解
- 3.tcp 协议和 udp 协议是哪一层的协议，讲一下他们之间的区别，以及他们各自的应用场景
- 4.你知道 get 和 post 请求吗，讲一下他们之间的区别
- 5.当我们输入美团网址的时候，这个从输入到显示页面的过程
- 6.你了解操作系统吗，linux 了解吧，你说一下你用过的命令
- 7.当我们要查看文件的具体属性用什么命令
- 8.讲一下静态链表和动态链表的区别
- 9.数据的 3 范式了解吗
- 10.索引是什么
- 11.讲一下索引的数据结构
- 12.说一下 drop、delete、truncate 的区别， truncate 删除数据后可以回滚吗
- 13.MR 的过程
- 14.HBASE 和 hive 的区别，以及他们的应用场景
- 15.hive 中内部表和外部表的区别，他们各自的应用场景，你们公司建表是内部

还是外部

16.hive 小文件过多有什么问题，以及对应的解决方案

17.你了解的 hive 中文件的存储格式有哪些

18.orc 和 rc 的区别是什么

19.你知道 OLAP 和 OLTP 的区别吗，你用过或者了解哪些 OLAP 工具，OLAP 的全称是什么

20.你知道雪花模型和星型模型的区别吗，你们公司一般用哪一种

21.缓慢变化维是怎么处理的

22.数仓分层的架构

23.给了 1 张 sno,cno,grade 表，出了 3 题 sql（比较简单）

24.LC300. 最长递增子序列

二面

1.自我介绍

2.公司用到的平台介绍一下

3.实习经历详细说一下 这里延伸问了很多

4.还问了一下实时数仓项目的收获是什么

5.flink 中你学到了哪些东西，深入讲了一下 watermark

6.flink 和 spark streaming 的区别

7.flink 里面保证容错性的技术是什么，回答 checkpoint，那么 checkpoint 和

8.safepoint 的区别是什么

9.spark 参数调优了解多少

10.spark 数据倾斜遇到过吗

11.MapReduce 的运行过程

12.聚簇索引和非聚簇索引的区别

13.覆盖索引是什么

14.flink 流和表之间的联系

15.刷了两道 SQL 题，第一道比较简单，第二道是 表(poิ_id, biz_time, order_id)，

计算门店经营效率，每一分钟内的结账订单数>2 的分钟个数，超过 60 个的算 1 小时，然后统计区间段

面经 2

一面

- 1.自我介绍
- 2.围绕项目：数据接入表的介绍
- 3.MySQL 索引分类
- 4.索引如何添加的，为什么要添加索引，哪些不需要添加索引，有些情况为什么索引会很慢
- 5.实习公司自研调度平台介绍
- 6.数据流介绍，数据写入的过程
- 7.异构数据源融合是怎么做的，碰到了什么问题，如何解决的
- 8.数仓分层
- 9.数仓建模方式：范式建模，维度建模
- 10.Hive 表的存储格式，为什么要这么存，不用别的存储
- 11.数据倾斜
- 12.Mapjoin 原理，介绍一下
- 13.主题表是如何制作的，处于数仓的哪一层，你这个项目的数仓是如何分层的
- 14.写 sql，自联结的，还算简单。没有手撕

二面

- 1.自我介绍
- 2.范式建模，说一下数据库范式，分别举个例子
- 3.范式建模用的多还是维度建模用的多
- 4.MR 应该了解吧，详细说说；说完之后，spark 和他有什么区别
- 5.说说 shuffle 的过程，哪些算子，我又说了说对不同算子的优化
- 6.索引机制，用到了哪些引擎，区别是什么
- 7.事务和锁
- 8.平时用的最多的开发语言是什么，我说是 Java，然后面试官不说话了，我就自己说了说 HashMap 1.7 和 1.8 的区别。。。说的一半被打断了，问我用 Java 写过什么？我说 hive 里的 udf 写过一些，然后开始了 udf 的介绍
- 9.Udf 分为哪几种，每种类型是干什么的，如何使用
- 10.那你用 Java 写个快排把。
- 11.写完 partition 被打断了，又写了一个 sql，很简单，但是当时脑子有点短路，引导了一下，说了思想，面试官要去吃饭了，就没让写了

蚂蚁

面经 1

一面

- 1.自我介绍
- 2.你会的语言 java,scala,python 说说他们的区别(问了很长时间)
- 3.说说你对大数据生态的理解，需要那些组件(存储，计算，采集等等啥的)
- 4.Mapreduce 的详细流程说一下
- 5.hive， spark， MapReduce 都是怎么进行数据计算的，有没有什么联系
- 6.数仓分层，传统数仓和大数据数仓啥区别
- 7.电商业务常见的数据域说一下
- 8.数据湖了解过吗？ 数仓发展趋势 了解过吗？

二面

- 1.一道算法题和一道 sql
- 2.实习的时候遇到的难点，怎么解决的，后续怎么优化的
- 3.你在做报表开发的时候有没有想过怎么下沉到 dwd 层(???? 没听懂啥意思)
- 4.索引 B+树索引的一个优势
- 5.基本上没了，因为做题浪费了很多时间。

面经 2

一面

- 1.介绍项目
- 2.遇到了什么问题 怎么解决的
- 3.介绍一下大数据的发展历程
- 4.项目选型，为什么用这些组件
- 5.介绍另一个项目
- 6.除了准确率还用了什么指标来评判模型？
- 7.医疗大数据有没有前景
- 8.想从事什么行业

二面

- 1.介绍项目
- 2.数仓的分层模型怎么做的，每一层做了什么事情？
- 3.Hive 中大表和小表怎么聚合的？复制小表到 maptask 的过程是什么样的？
- 4.维度模型里各个维度之间怎么聚合的？
- 5.聚合过程的数据倾斜怎么解决？
- 6.项目中遇到了什么问题，怎么解决的？
- 7.随机森林和决策树的区别是什么？怎么体现随机的？
- 8.场景题：
 - 8.1 一个城市里经纬度不同的一百万条数据 在 hive 中，怎么聚类？
 - 8.2 m 个班， n 个同学，两两之间聊天就会产生一个边，怎么评判一个班的活跃程度和是否有明星同学(一个点的边很多)，产生明星同学的变化过程怎么衡量？
- 9.概率：
17 个水果，10 个苹果，4 个橘子，3 个香蕉，从里面拿 9 个，问恰好拿出 4 个苹果，3 个橘子，两个香蕉的概率。

三面

- 1.介绍项目
- 2.每一个部分具体的组件是做什么的？
- 3.sparkStreaming 里的数据怎么处理的？
- 4.维度建模怎么建的？分了哪几层
- 5.建模的过程怎么进行数据质量管理？上万张维度表的话怎么很快的找见？
- 6.每天数据量的规模（千万级）？
- 7.十亿级的数据量这个系统要怎么优化，kafka 里怎么优化？

阿里

面经 1

一面

- 1.哈希 优点
- 2.二叉树 平衡二叉树 b 树与 b+树的区别
- 3.sql 中 on 和 where 的区别

4.sql 中的几个 join

5.hadoop 生态

6.hadoop 与 sql 区别

二面

1.实习经历

2.简单介绍 wordcount

3.HDFS 读写机制

4.spark 和 MapReduce 的区别

5.spark streaming 和 flink 的区别

6.kafka 好处

7.数仓架构

8.分层原理

面经 2

三面汇总

1.SQL 题

2.实习经历

3.hadoop 和 spark 的区别

4.spark rdd 和 dataset 的关系

5.spark 宽窄依赖

6.spark 如何解决数据倾斜

7.kafka 如何保证消息顺序性

8.flink 的双流 join

9.flink 反压机制

10.分布式一致性协议

字节跳动

20 篇面经合集

面试部门	面经发布日期	面经链接
抖音	2023.04.12	https://www.nowcoder.com/discuss/475592930086952 960

抖音	2021.03.11	https://www.nowcoder.com/discuss/611776
广告	2021.01.31	https://www.nowcoder.com/discuss/592114
AI Lab	2021.04.06	https://www.nowcoder.com/discuss/632801
商业化	2023.04.19	https://www.nowcoder.com/discuss/478214676891856896
	2021.07.19	https://www.nowcoder.com/discuss/353158112901275648
	2021.05.13	https://www.nowcoder.com/discuss/353157927605313536
	2021.03.09	https://www.nowcoder.com/discuss/609584
	2021.03.14	https://www.nowcoder.com/discuss/613574
	2021.03.18	https://www.nowcoder.com/discuss/617913
	2021.03.23	https://www.nowcoder.com/discuss/621799
	2021.03.24	https://www.nowcoder.com/discuss/622742
	2021.03.25	https://www.nowcoder.com/discuss/623104
	2021.04.06	https://www.nowcoder.com/discuss/633122
	2021.04.08	https://www.nowcoder.com/discuss/635448
	2021.04.21	https://www.nowcoder.com/discuss/645291
	2021.04.21	https://www.nowcoder.com/discuss/645554
	2021.01.28	https://www.nowcoder.com/discuss/592112
	2021.03.01	https://www.nowcoder.com/discuss/602031
	2021.03.03	https://www.nowcoder.com/discuss/604268

面经 1

一面

1. 主要考察的是 SQL 的撰写，特别是 HAVING，GROUP BY 等的考察（四题）
2. 第一个在数组中有重复数字的位置 $O(n)$

二面

一、Spark 部分

- 1.Shuffle 详细 Shuffle Read Shuffle Write 溢写
- 2.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 $10 \text{ (Map Task)} * 2 \text{ (Reduce Task)}$
问优化过的 Hash Shuffle 产生的文件个数 $(2 \text{ (Executer)} * 2 \text{ (Core)}) * 2 \text{ (Reduce Task)}$
3. SortShuffle 产生的文件个数 $2 \text{ (Executer)} * (1 \text{ (合并的文件)} + 1 \text{ (索引)})$

二、项目介绍

项目目标及难点

三、计算机基础及通信网络

1.数据通信 7 层/5 层

2.进程之间的通信

3.进程相关问题，进程开销问题

4.进程和线程问题

5.内存方面问题

四、算法

1.SQL 算曝光率

2.剑指 Offer 30. 包含 min 函数的栈 $O(1)$

3.搜索旋转排序数组

面经 2

一面

一、java 部分

1.类加载（加载流程，加载器，强弱软虚引用会被问到，得了解）

2.垃圾回收（怎么解决计数法的弊端）

3.多线程（线程间的通信，锁，volatile，CAS）

4.内存模型，内存管理（溢出和泄漏的区别）

5.NIO

二、redis 部分

1.数据结构（zset 怎么实现的）

2.备份（RDB 和 AOF）

3.RDB 过程中修改数据，怎么办

4.缓存击穿/穿透/雪崩（常考点，但是面试官没问我）

三、spark 部分

1.client 和 cluster 模式的区别

2.stage 划分

3.宽窄依赖

4.spark shuffle（从 stage 划分递进到 shuffle 过程，条理性很强）

5.数据倾斜问题

6.join 的种类

7.两张表 join (小表直接广播出去，不能广播的表怎么解决)

8.spark shuffle

9.spark 内存管理

四、算法部分

链表排序，面试官让我用归并排序实现。

二面

1.主要聊了聊业务，基础知识问的比较少

2.说说 spark 和 flink 的区别

3.介绍介绍 kudu

4.如何设计一个类似 kudu 的数据库

5.数据仓库建模，雪花模型，星型模型，ODS,DWD,DWS 多层结构

6.数据结构相关的问了平衡二叉树

7.常见的排序算法

8.hashmap 的实现原理

9.什么场景下用归并排序，什么场景下用快速排序（从 mapreduce shuffle 的角度出发，就能发现）

10.实现常数级的时间复杂度取出栈中的最小值

三面

1.继续聊项目聊业务，整个项目的细节，项目的难点，项目中遇到的问题

2.spark on yarn 的流程，分部署模式答

3.spark 程序故障重启，checkpoint 检查点

4.抖音的视频转发场景：视频的原作者发布了一个视频吗，TA 的关注者看到后会转发，后续关注者的关注者会继续转发，以此类推。我们现在有这样一个数据集，要找到视频的源头，以及转发的层数次。

面经 3

一面

1.synchronized 和 volited 的区别

2.hadoop 是什么

3.TCP 和 UDP 的区别

4.用过那些 Linux 命令

- 5.ArrayList 原理，为什么初始是 10，为什么扩容 1.5 倍
- 6.算法 二叉树之字遍历
- 7.写一个 wordconunt
- 8.进程和线程有什么异同点
- 9.实现单例模式
- 10.java 限定词（private 那些）

二面

- 1.能实习多久
- 2.说一下 mapreduce
- 3.事务隔离级别
- 4.上述级别会带来什么问题
- 5.mysql 索引有啥
- 6.B+树与 B 树区别
- 7.java 锁都有什么，JUC 包
- 8.lock 是公平的还是非公平的（答案是可以根据逻辑去自己实现是否公平）
- 9.场景题，给一个字段（userid, logintime, logouttime），统计用户在线最大峰值和持续时间段
- 10.股票交易 1 2
- 11.问项目，为什么不继续干数据挖掘
- 12.hdfs 写数据流程
- 13.synchronized 与 lock 的区别
- 14.公平锁与非公锁的区别
- 15.reduce 怎么知道从哪里下载 map 输出的文件
- 16.如果 map 输出太多小文件怎么办

百度

面经 1

一面

- 1.自我介绍
- 2.八股文

你写的这个实时数仓，维表是怎么更新的
flink 了解吧， flink 里面断流怎么处理
flink 的 exactly-once 是怎么实现的
checkpoint 的时候 barrier 什么时候发送
checkpoint 产生了很多快照，怎么进行处理呢
sparkstreaming 和 structedstreaming 的区别
zookeeper 的分布式一致性协议是什么， raft 讲一下， paxos 讲一下
hive 调优怎么调的
数据倾斜怎么处理的
你在做需求的过程中遇到过的最大的难点是什么

3.SQL 题

题目：tbl_v2x starttime, endtime > 10s 是一个任务，tbl_pnc starttime, enditme, status = 'auto-driver'的统计，统计每一天，任务的闭环率

面经 2

一面

1. 自我介绍
2. 项目介绍
3. Java NIO 原理，与 BIO 有什么区别
4. 线程不安全的类有哪些
4. G1 回收器原理
4. 类加载过程
4. GCROOTS 由什么组成
5. 数据库索引
6. 数据库的事务和一致性
7. 计网分层结构
8. TCP 和 UDP 的区别
9. TCP 怎么实现可靠传输
10. HTTP3.0 展开讲讲
11. 进程和线程的区别
12. 进程通信方式
13. mapReduce 原理以及运行流程
14. hadoop 中小文件如何处理

15. Hive SQL 转换为 MR 的过程
16. Hive UDF 怎么实现和使用
17. left semi join 是什么
18. Hive 窗口函数有哪些
19. lateral view explode 有什么用
20. spark 任务执行流程
21. RDD 是什么
22. RDD 和 DataFrame 有什么区别

三道算法题，两道实现一道口述：

1. 简单 dp
2. 简单链表
3. 判断无向图是否存在环

二面

- 1 自我介绍
- 2 数据倾斜——结合业务说了分组聚合和 join, 大表大表 join 怎么解决数据倾斜，展开说
- 3 SQL 没做出来，但是面试官全程提示，沟通交流
- 4 Hive 和 Spark 哪个比较熟悉，我们聊一聊——都比较熟悉，问了 Hive 组件和底层执行逻辑，逻辑计划优化有哪些方法，什么是谓词下推（自己提到了）
- 5 算法：数组 找所有元素之和为 0 的三个不重复的数字，
- 6 HashMap 安全吗？不安全，为什么，延伸聊一聊，说了保证安全的 synchronizedXXX 以及 ConcurrentHashMap, ConcurrentHashMap 获取全局属性的时候存在问题（自己提出），有什么优化方法

滴滴

面经 1

一面

1. 自我介绍
 2. 八股文
- hive 的数据类型有哪些
- hive 中计算排名前 N 的函数有哪些

hive 的优化手段你知道哪些

遇到过数据倾斜吗

mapjoin 如何开启，参数是什么

3.SQL 题

rt_data 表存放了当天每半小时的店铺销售数据，表名: rt_data，字段名: shop id (店铺 id), stat date (时间), ord_amt(销售额)。找出昨天 10 点各店铺的销售金额及前半小时的销售金额和后 1 个小时的销售金额。

思路：（开窗函数，lead 和 lag）

4.数仓理论

谈谈你对数仓的理解

你们的数仓是怎么分层的，为什么要对数仓进行分层

介绍一下你做的离线数仓的数据全链路

面经 2

一面

1.自我介绍

2.问实习经历问了很久

3..五道 sql 题

4.数仓理论

二面

1.自我介绍

2.问实习经历

3.sql 题

4.mapreduce 的流程及其 shuffle

5.mapreduce 与 spark 优劣好处

6.hive 实习的优化，自己的理解，自己遇到的难题，数据倾斜怎么解决，数据量啥的，根据实习经历来回答

7.快排，归并理解与手撕

网易

面经 1

一面

- 1.自我介绍
- 2.说说项目用到了哪些技术
- 3.你刚刚说到了即席查询，项目里是怎么做的
- 4.四道 sql 十分钟后对答案。
- 5.迪卡尔积了解吗
- 6.会产生什么问题
- 7.你刚刚说了数据倾斜，介绍一下
- 8.迪卡尔积就会产生数据倾斜吗
- 9.mr 流程介绍一下
- 10.你多久能来实习
- 11.你刚刚说到了数仓分层，介绍一下
- 12.大学里什么事是你一直在做的。
- 13.能实习一年吗
- 14.大学里最有成就感的事

二面

- 1.自我介绍
- 2.聊项目 10 分钟
- 3.聊下大数据组件，后改成结合你的项目来说下用了哪些，为什么用。
- 4.平时怎么学习的
- 5.为什么想找实习，实习希望收获什么
- 6.看你 24 届。学校没课，不需要考勤吗。
- 7.有什么想去的公司。
- 8.为什么不去阿里。
- 9.面过哪些公司，什么原因没通过

面经 2

一面

- 1.项目介绍（自己做的一个数据预测+挖掘的项目）
- 2.项目中遇到的困难
- 3.Hadoop 有的解（介绍一下 Hadoop，简单说了一下 Hadoop 的组成和定义）
- 4.介绍一下 MR 的原理（工作流程，Map-Shuffle-Reduce）
- 5.Hive 介绍一下（这里有点懵，因为感觉 hive 关键点在 SQL，只说了 Hive 的定义和架构组成和一些底层 MR、Tez、Spark 的引擎的东西）
- 6.面试官说大数据分：数据产品（算法）、数据平台（服务端开发）、数仓（业务）三个方向，偏向于哪一个

二面

- 1.介绍项目
- 2.项目中的困难（一面之后提前准备了这个）
- 3.重新开始，能够怎么去优化项目（从硬件设备、算法架构、人员配置、团队分工几个方面说了一下）
- 4.Hadoop 的使用经验（介绍了一下 Hadoop 的生态和组成以及应用场景）
- 5.Hadoop 使用中遇到的困难（从自身设备、内存分配、组件核心配置说了几点，重点说了数据倾斜）
- 6.Hadoop 数据倾斜的解决方案（combiner、局部聚合加全局聚合、自定义分区、增加 jvm 内存）
- 7.HiveSQL 数据倾斜和 Spark 数据倾斜（因为那时候还没学到 spark，hive 也不是很熟悉，就按照操作经验随便说了几点 hive 的）
- 8.个人对于算法研究的认知（因为项目里是偏算法的，但是我个人不喜欢做算法就分析了个人性格和实际的能力，以及对于学术研究的态度）
- 9.对于数仓的看法（之前做过尚硅谷的阿里云的离线数仓，按照那个数仓的架构体系说了一下）

面经 3

三面汇总

- 1.hadoop 存储，MR 和 shuffle，reduce 分三个阶段，你怎么分，资源调度

- 2.Hive 执行流程（SQL 转 MR 过程）
- 3.ZK 的原理 paxos 一致性算法
- 4.进程和线程区别
- 5.内存模型
- 6.锁的机制，什么时候用到锁，
- 7.翻转字符串
- 8.括号字符串的有效性
- 9.问项目，问一下数据建设情况，可视化用的什么技术
- 10.对 Kylin 的了解，Cube 的优化，减少膨胀率
- 11.写 SQL，我记得一个是用 rank 求排名的，一个是用 explode 函数，把数组炸裂开，让一行变多行的，然后连表
- 12.hive 优化相关，SQL 书写优化，连表优化等，hive 的调参数优化等
- 13.HBase，写数据流程

快手

面经 1

三面汇总

- 1.说一下 shuffle 过程
- 2.分区是根据啥分区的，自定义分区
- 3.数据倾斜产生，做 join 的倾斜问题，map join 的原理，调整 map 数和 reduce 数，怎么调
- 4.数仓整个过程，项目相关
- 5.HashMap 的实现
- 6.快速排序 实现，时间复杂度，稳定性。
- 7.区间合并的题，LeetCode
- 8.java 基础封装继承多态
- 9.集合相关，list 和 map 那些
- 10.线程相关，数据库相关
- 11.HDFS，MR，Hive
- 12.项目介绍
- 13.现在有大学考试成绩如下表 A: id, subject, score, 找出每一科都是这一科前 30% 的学生的 id

面经 2

一面

1. 双流 Join，讲一下过程
2. 你的数据过期时间是多久？为什么这么设置？
3. 双流 join 之后你的数据会从 Redis 里面删除吗？
4. 你说到了用 Redis 来缓存延迟的数据，如果缓存存储的数据过多会出现什么问题，怎么解决？
猜：会出现 OOM 问题，我觉得可以用 Redis 集群来解决单个 Redis 节点上数据过多的问题
5. 追问场景：如果两个流，一个出现了异常，重启后两个流之间出现了很大的延迟。是不是会导致数据全部缓存到 Redis 里面？
6. 那么你打算怎么解决这个问题？
7. 你在做 Hive 数仓项目的时候有出现过数据倾斜的现象吗？是如何解决的？
8. 还有哪些倾斜的场景？
9. 继续答了 join 倾斜，大表和小表 join。小表复制 N 份大表打 1-N 的前缀，join，之后再聚合。忘记说采样了，面试官没有继续追问。下次再改进回答。
10. 说一下怎么判断链表有环？
11. 说一下怎么找到环入口？给出详细证明过程？
12. 说一下 HashMap 的底层实现
13. 说一下重载和重写的区别
14. 说一下抽象类和接口的区别
15. 反射是什么？反射是怎么创建对象的
16. 用过线程池吗，说一说线程池的各个参数？以及他们的含义？
17. maximumPoolSize 和 corePoolSize 有关联吗？
18. 说一下有哪几种拒绝策略？
19. SQL 题
给定 temp 表，每个 id（用户）会被打上不同的标签。labels. show 是曝光次数，click 是点击次数。
要求：求出每个 label 的曝光次数和点击次数。
20. 算法 力扣 103

二面

1. 介绍下 Spark 项目全流程

- 2.介绍下数仓分层架构，如何分层的
- 3.Flume + Kafka + Scoop 的采集架构，如何保证数据的可靠性
- 4.Redis 为什么快？
- 5.ElasticSearch 你在项目中是怎么用的，如何建的索引？
- 6.Spring 了解吗，说一说？
- 7.场景：打印错误日志，同一个链路的方法要求打出相同的数字？好像是这个题目。面试官提醒用 ThreadLocal 做。这里不太了解，有懂的兄弟吗？
- 8.场景：现在有直播带货业务，你来搭建数仓，你会怎么去搭建？
- 9.SQL 题目：一面的题目如何反向得到 labels 列。
- 10.Spark 和 Hadoop 的区别，说一下
- 11.Spark shuffle 和 hadoop shuffle 的区别说一下。
- 12.算法：K 个一组反转链表

三面

- 1.有实习吗？
- 2.做自动驾驶的啊，讲讲自己的项目？
- 3.一道题目：二叉树最大宽度。

微众银行

面经 1

一面

- 1.一下之前的项目经历
- 2.来源
- 3.已的职业规划是怎样的
- 4.的知识体系或者技能体系是什么样的
- 5.设计里面的要点或者难点是什么
- 6.维度建模吗？简单讲一下维度建模
- 7.建模解决的是什么问题
- 8.自己有哪些优点，有什么示例能够支撑
- 9.引以为豪的事情有哪些
- 10.么多大数据框架难点是什么？整体的学习方法、规划？想分享的

京东

面经 1

一面

- 1.介绍
- 2.实时数仓的项目
- 3.里用到了 clickhouse, clickhouse 的写入和读取为什么快
- 4.flink 有哪些算子
- 5.flink 的窗口函数了解吗
- 6.flink 的精准一次性如何保证的
- 7.kafka 是如何保证数据不丢失和不重复的，从生产者和消费者考虑
- 8.hbase 用过吗， rowkey 的设计原则是什么
- 9.如何解决热点现象
- 10.redis 的数据结构了解吗
- 11.java 的 spring 会吗
- 12.java 的集合类有哪些
- 13.java 实现多线程的几种方式
- 14.你知道有哪些实现线程池的方式吗，讲一下有哪些类
- 15.udf 函数的分类
- 16.你实现的 udf 函数的功能
- 17.项目中最大的收获是什么

二面

- 1.自我介绍
- 2.介绍了一下你的数仓项目
- 3.遇到过的最难的需求，怎么解决的
- 4.MapReduce 的执行过程
- 5.zookeeper 的 leader 选举机制，常见的一些应用场景，举例说明
- 6.kafka 介绍一下
- 7.spark 中 jvm 调优怎么调
- 8.hive 优化你用过哪些，数据倾斜遇到过吗
- 9.你采用的数据存储格式是什么，相比于其他有什么优势

- 10.flink 和 spark 的区别是什么
- 11.hashmap 的底层原理是什么
- 12.你用过的一些 linux 命令
- 13.刷题： Leetcode10 正则表达式匹配

面经 2

一面

- 1.谈谈 GC
- 2.乐观锁与悲观锁区别
- 3.如何理解线程安全
- 4.一个大数(超过了 Long 的最大值)，你怎么存储
- 5.kafka 的有点
- 6.项目中最难的点及解决方案
- 7.什么是 mysql 联合索引，建立 abc 索引，查 aba,ac,bc 分别走不走索引
- 8.kafka 的消费者组是怎么回事，为什么有消费者组，作用是啥
- 9.hashmap 与 concurrenthashmap
- 10.spark 的 repartiton 底层原理，它与 coalesce 的区别

二面

- 1.手撕二分查找
- 2.一个数组中数字有重复的，找到重复次数前 k 多的数，只说了思路，分析了时间复杂度
- 3.有 16T 的有重复的数据求找到重复次数前 k 多的数
- 4.spark 如何实现问题 3，需要用到 spark 的什么算子
- 5.性能调优，说说常见的数据倾斜及调优方案
- 6.手撕代码：二叉树从头出发到叶子节点的所有路径中路径和等于 target 的，剑指 offer 原题

携程

面经 1

一面

- 1.int 和 Integer 有什么区别？
- 2.Integer(200) new 两次，他们是一样的吗？
- 3.valueOf 方法介绍一下
- 4.拆箱和装箱指什么
- 5.为什么需要拆箱和装箱
- 6.jvm 内存模型
- 7.hashmap 底层结构，为什么要链表转红黑树？
- 8.Concurrent hashmap 底层为什么线程安全？
- 9.try catch finally，try 里面有 return 语句，finally 的语句还会执行吗？finally 也有 return，这个 return 会执行吗？
- 10.堆栈区别，为什么要堆栈
- 11.堆，栈，方法区一般有什么
- 12.synchronized 关键字干什么用的，作用在哪儿
- 13.线程同步你一般怎么做，你有什么套路总结出来了，那怎么优雅的停止线程，除了 interrupt 呢？
- 14.线程池用过吗？有哪些线程池，线程池构造方法参数最多的那个，每个参数说遍作用
- 15.线程池你最常用哪个，为什么，适用场景在哪里？（ps：线程池相关的我一窍不通）
- 16.线程池的 execute 和 submit 方法有什么区别？
- 17.spring 的 ioc 和 aop
- 18.用过哪些注解，Autowire 干嘛的
- 19.循环依赖问题怎么解决
- 20.你用过 mybatis 吗（用过，但我说不懂原理，面试官笑了笑，翻了下一页，开始大数据八股时间）
- 21.Spark 的 shuffle 过程（背过，忘了，我解释了下没怎用 spark）
- 22.mapreduce 的过程
- 23.flink 了解过吗

面经 2

一面

- 1.数仓分层，每一层的含义。
- 2.数仓建模
- 3.元数据
- 4.SQL:面试官首先问了一个语句，我没见过。然后问了另一道题，问我的想法，
- 5.MySQL 分库分表
- 6.表的删除方式以及它们之间的区别

二面

1. JAVA GC 有哪几种
2. Linux 命令查看内存和 CPU (top) 查看网络 (netstat)
- 3.计算机底层通过什么码来计算（补码 为什么？）
- 4.然后就说大数据组件你了解哪些 手写一个 wordcount (分分钟写出来)
- 5.spark 和 mapreduce 的对比
- 6.HDFS 如何处理小文件
- 6.KAFKA 消费者如何消费多分区（一个消费者消费一个分区 能不能两个消费者消费一个分区）
- 7.TCP 三次握手（为什么要进行第三次）
- 8.四次挥手（为什么要多进行两次）
- 9.mysql 支持哪些锁

第八部分 大厂 SQL 真题 (持续更新~)

短视频业务

1.各个视频的平均完播率

- 题目描述:
 - 用户-视频互动表 tb_user_video_log
 - ◆ uid-用户 ID, video_id-视频 ID

- ◆ start_time-开始观看时间, end_time-结束观看时间
- ◆ if_follow-是否关注, if_like-是否点赞, if_retweet-是否转发
- ◆ comment_id-评论 ID
- 短视频信息表 tb_video_info
 - ◆ video_id-视频 ID, author-创作者 ID
 - ◆ tag-类别标签, duration-视频时长（秒）, release_time-发布时间
- 问题描述:
- 计算 2021 年里有播放记录的每个视频的完播率(结果保留三位小数), 并按完播率降序排序
 - ◆ 完播率的口径: 指完成播放次数占总播放次数的比例。其中, 结束观看时间与开始播放时间的差 \geq 视频时长时, 视为完成播放。
- 分析: 从需求出发, 我们去进行拆解, 每个视频的完播率 = 每个视频完成播放的次数 / 每个视频总播放的次数。求解每个视频完成播放的次数, 需要用到的字段有 start_time, end_time, 以及 duration, 只要判断当前视频播放的 $end_time - start_time \geq duration$, 就视为一次完成播放; 求解每个视频总播放的次数, 就是计算每个视频出现了多少次就行了, 也就是对 video_id 进行 count
- 答案:

```

select
    t1.video_id,
    round(
        sum(if (end_time - start_time >= duration, 1, 0)) /
        count(t1.video_id), 3) as avg_comp_play_rate
from
    tb_user_video_log t1
left join tb_video_info t2
on t1.video_id = t2.video_id
where
    year (start_time) = '2021'
group by
    t1.video_id
order by
    avg_comp_play_rate desc
;

```

2.平均播放进度大于 60%的视频类别

- 问题描述:

- 计算各类视频的平均播放进度，将进度大于 60% 的类别输出
 - ◆ 播放进度的口径：播放时长 ÷ 视频时长，当播放时长大于视频时长时，播放进度均记为 100%。结果保留两位小数，并按播放进度倒序排序。
- 分析：
 - 从需求出发，我们进行拆解，要求某类视频的平均播放进度，需要先求出播放进度，根据其口径，要得到每个视频的播放时长，来源是 start_time 和 end_time 两个字段，其中需要注意的细节就是当 end_time-start_time>duration 的时候，播放进度取值为 100%
 - 补充函数：
 - ◆ timestampdiff(day/hour/second, 开始时间, 结束时间)：获取时间差，单位可以是天/小时/秒
 - ◆ substring_index(col, str, index)：获取字段中第 index 个 str 左边的所有字符
 - 例如 substring_index(abcabc, 'b', 1)=> a ||
substring_index(abcabc, 'b', 2)=> abca
- 答案：

```

select
    tag,
    concat(round(avg(play_progress) * 100, 2), '%') as avg_play_progress
from
(
    select
        tag,
        if(
            TIMESTAMPDIFF(second, start_time, end_time) >
duration,
            1,
            TIMESTAMPDIFF(second, start_time, end_time) /
duration
        ) as play_progress
    from
        tb_video_info t1
        join tb_user_video_log t2 on t1.video_id = t2.video_id
)
```

```

) tmp
group by
tag
having
substring_index (avg_play_progress, '%', 1) > 60
order by
avg_play_progress desc

```

3.每类视频近一个月的转发量/率

- 问题描述：
 - 统计在有用户互动的最近一个月（按包含当天在内的近 30 天算，比如 10 月 31 日的近 30 天为 10.2~10.31 之间的数据）中，每类视频的转发量和转发率（保留 3 位小数）。
 - ◆ 转发率=转发量÷播放量。结果按转发率降序排序。
- 分析：
 - 从需求可以知道，我们需求求转发量和播放量，转发量对应字段 if_retweet，每类视频的播放量就是对 每类视频的个数进行累加。其中需要注意的口径就是 有用户互动 且 最近一个月，因为只有在 tb_user_video_log 这张表的数据从需求可以知道，我们需求求转发量和播放量，转发量对应字段 if_retweet，每类视频的播放量就是对 每类视频的个数进行累加。
 - ◆ 其中需要注意的口径就是 有用户互动 且 最近一个月，因为只有在 tb_user_video_log 这张表的数据才是用户视频互动数据，那么在 join 的时候，将该表作为主表并使用 left join 即可；最近一个月怎么去判断呢，只需要找出当前互动表中最大的日期，向前推算一个月即可

- 答案：

```

select
author,
date_format (start_time, '%Y-%m') as month,
round(
(

```

```

        sum(if(if_follow = 1, 1, 0)) - sum(if(if_follow = 2, 1, 0))
    ) / count(*),
    3
) as fans_growth_rate,
sum(
    sum(if(if_follow = 1, 1, 0)) - sum(if(if_follow = 2, 1, 0))
) over (
    partition by
        author
    order by
        date_format(start_time, '%Y-%m')
) as total_fans
from
tb_user_video_log t1
left join tb_video_info t2 on t1.video_id = t2.video_id
where
    year(start_time) = '2021'
group by
    author,
    date_format(start_time, '%Y-%m')
order by
    author,
    total_fans

```

4. 每个创作者每月的涨粉率及截止当前的总粉丝量

- 问题描述：
 - 计算 2021 年里每个创作者每月的涨粉率及截止当月的总粉丝量
 - ◆ 涨粉率=(加粉量 - 掉粉量) / 播放量。结果按创作者 ID、总粉丝量升序排序。
 - ◆ if_follow-是否关注为 1 表示用户观看视频中关注了视频创作者，为 0 表示此次互动前后关注状态未发生变化，为 2 表示本次观看过程中取消了关注。
- 分析：
 - 分析需求，要求两个指标，分别是 涨粉率 以及 累计粉丝量；涨粉率 ->

if_follow; 累计粉丝量，开窗函数就可以了。

- 答案：

```

select
    author,
    date_format(start_time, '%Y-%m') as month,
    round(
        (
            sum(if(if_follow = 1, 1, 0)) - sum(if(if_follow = 2, 1, 0))
        ) / count(*),
        3
    ) as fans_growth_rate,
    sum(
        sum(if(if_follow = 1, 1, 0)) - sum(if(if_follow = 2, 1, 0))
    ) over (
        partition by
            author
        order by
            date_format(start_time, '%Y-%m')
    ) as total_fans
from
    tb_user_video_log t1
    left join tb_video_info t2 on t1.video_id = t2.video_id
where
    year(start_time) = '2021'
group by
    author,
    date_format(start_time, '%Y-%m')
order by
    author,
    total_fans

```

5. 国庆期间每类视频点赞量和转发量

- 问题描述：

- 统计 2021 年国庆头 3 天每类视频每天的近一周总点赞量和一周内最大单天转发量，结果按视频类别降序、日期升序排序。假设数据库中数据足够多，至少每个类别下国庆头 3 天及之前一周的每天都有播放记录。
 - 分析：
 - 首先分析本题的需求，有两个指标，每类视频每天 的近一周总点赞量和每类视频每天 的一周内最大单天转发量；因此应该先求出每类视频每天的点赞量和转发量，然后再去求题目的指标；每类视频每天的点赞量和转发量很容易求出，就是分组+聚合；那么如何求这三天近一周的总点赞量呢？我们发现每天去求最近一周的时候，这一周是动态变化的，但是窗口大小是不变的，都是最近 7 天，所以可以考虑开窗函数，**窗口大小就是 7 天前到今天，这个用代码如何表示呢—— rows between 6 preceding and current row**（注意！！！是前 6 行，而不是前 7 行！！！）

- 答案：

```
select
    tag,
    dt,
    like_cnt,
    retweet_cnt
from
(
    select
        tag,
        dt,
        sum(like_cnt) over (
            partition by
                tag
            order by
                dt rows between 6 preceding
                and current row
        ) as like_cnt,
        max(retweet_cnt) over (
            partition by
                tag
            order by
```

```

dt rows between 6 preceding
and current row
) as retweet_cnt
from
(
    select
        tag,
        date_format(start_time, "%Y-%m-%d") as dt,
        sum(if_like) as like_cnt,
        sum(if_retweet) as retweet_cnt
    from
        tb_user_video_log t1
    join tb_video_info t2 on t1.video_id = t2.video_id
    where
        date_format(start_time, "%Y-%m-%d") between
        '2021-09-25' and '2021-10-03'
        group by
            tag,
            dt
    ) t
)
where
dt between '2021-10-01' and '2021-10-03'
order by
tag desc,
dt;

```

6.近一个月发布的视频中热度最高的 top3 视频

- **问题描述:**
 - 找出近一个月发布的视频中热度最高的 top3 视频。
- **分析:**
 - 对需求进行拆解，要求的指标就是 每个视频的热度 -> 每个视频的完播率 + 点赞数 + 评论数 + 转发数 + 新鲜度；这些小指标中，完播率是

指每个视频完成播放的次数除以播放的次数，点赞数/评论数/转发数都很容易获取，新鲜度又需要去求 **视频最近无播放天数**；以所有视频结束时间的最大值为基准，判断每个视频的最大结束时间和基准的差即为最近无播放天数。

- 本题还需要注意的一个口径，近一个月发布的视频如何判断？？？
- ◆ 我们这里的题解把该口径规定为：发布时间和基准的差在 30 天以内

● 答案：

```

select
    video_id,
    round(
        (
            100 * avg_comp_play_rate + 5 * like_cnt + 3 * comment_cnt + 2 *
            retweet_cnt
        ) / (no_play_cnt + 1)
    ) as hot_index
from
    (
        select
            t1.video_id,
            sum(
                if(
                    timestampdiff(second, start_time, end_time) >= duration,
                    1,
                    0
                )
            ) / count(t1.video_id) as avg_comp_play_rate,
            sum(if_like) as like_cnt,
            sum(if_retweet) as retweet_cnt,
            count(comment_id) as comment_cnt,
            datediff(
                date(
                    (
                        select
                            max(end_time)
                )
            )
        )
    )

```

```
from
    tb_user_video_log
)
),
max(date(end_time))
) as no_play_cnt
from
    tb_user_video_log t1
    join tb_video_info t2 on t1.video_id = t2.video_id
where
datediff(
    date(
        (
            select
                max(end_time)
            from
                tb_user_video_log
        )
    ),
    date(release_time)
) < 30
group by
    t1.video_id
)t
order by
    hot_index desc
limit
    3;
```

用户增长业务

1.2021 年 11 月每天的人均浏览文章时长

- 题目描述：

- 用户行为日志表 tb_user_log
 - ◆ uid-用户 ID, artical_id-文章 ID
 - ◆ in_time-进入时间, out_time-离开时间, sign_in-是否签到
- 说明：artical_id 为 0 表示用户在非文章内容页（比如 App 内的列表页、活动页等）。
- 问题描述：
- 统计 2021 年 11 月每天的人均浏览文章时长（秒数），结果保留 1 位小数，并按时长由短到长排序。
- 分析：1、过滤 2021 年 11 月数据 2、计算文章阅读时长 3、按日期分组统计，每天的人均浏览文章时长=当天阅读总时长/总人数
- 答案：

```

SELECT
    DATE_FORMAT(in_time, '%Y-%m-%d') date_str,
    ROUND(SUM(read_duration) / COUNT(DISTINCT uid),
        1) avg_duration
FROM
    (SELECT
        *, TIMESTAMPDIFF(SECOND, in_time, out_time) read_duration
    FROM
        tb_user_log
    WHERE
        DATE_FORMAT(in_time, '%Y-%m') = '2021-11'
        AND artical_id != 0) a
GROUP BY DATE_FORMAT(in_time, '%Y-%m-%d') order by avg_duration
ASC;
  
```

2. 每篇文章同一时刻最大在看人数

- 题目描述：
- 用户行为日志表 tb_user_log
 - ◆ uid-用户 ID, artical_id-文章 ID
 - ◆ in_time-进入时间, out_time-离开时间, sign_in-是否签到
- 说明：artical_id 为 0 表示用户在非文章内容页（比如 App 内的列表页、活动页等）。

- 问题描述：

- 统计每篇文章同一时刻最大在看人数，如果同一时刻有进入也有离开时，先记录用户数增加再记录减少，结果按最大人数降序。

- 分析：参考《第五部分》同时在线问题

- 答案：

```

select artical_id, max(cnt) as ax
from
  (select artical_id,
    sum(Mark) over (partition by artical_id order by Time, Mark desc) as cnt
  from
    (select artical_id, in_time as Time, 1 Mark from tb_user_log where
      artical_id != 0
    union all
    select artical_id, out_time as Time, -1 Mark from tb_user_log where
      artical_id != 0) as a
  )as b
group by artical_id
order by ax desc;

```

3.2021 年 11 月每天新用户的次日留存率

- 题目描述：

- 用户行为日志表 tb_user_log
 - ◆ uid-用户 ID, artical_id-文章 ID
 - ◆ in_time-进入时间, out_time-离开时间, sign_in-是否签到
- 说明：artical_id 为 0 表示用户在非文章内容页（比如 App 内的列表页、活动页等）。

- 问题描述：

- 统计 2021 年 11 月每天新用户的次日留存率（保留 2 位小数）

- 分析：1、先查询出每个用户第一次登陆时间（最小登陆时间）--每天新用户表；2、因为涉及到跨天活跃，所以要进行并集操作，将登录时间和登出时间取并集，这里 union 会去重--用户活跃表；3、将每天新用户表和用户活跃表左连接，只有是同一用户并且该用户第 2 天依旧登陆才会保留整个记录，否则右表记录为空；4、得到每天新用户第二天是否登陆表后，开始计算每天

的次日留存率：根据日期分组计算，次日活跃用户个数/当天新用户个数

- 答案：

```
select t1.dt,round(count(t2.uid)/count(t1.uid),2) uv_rate
from (select uid
       ,min(date(in_time)) dt
     from tb_user_log
    group by uid) as t1 -- 每天新用户表
left join (select uid , date(in_time) dt
            from tb_user_log
           union
           select uid , date(out_time)
            from tb_user_log) as t2 -- 用户活跃表
  on t1.uid=t2.uid
  and t1.dt=date_sub(t2.dt,INTERVAL 1 day)
  where date_format(t1.dt,'%Y-%m') = '2021-11'
  group by t1.dt
  order by t1.dt;
```

4. 统计活跃间隔对用户分级结果

- 题目描述：

- 用户行为日志表 tb_user_log
 - ◆ uid-用户 ID, artical_id-文章 ID
 - ◆ in_time-进入时间, out_time-离开时间, sign_in-是否签到
- 说明：artical_id 为 0 表示用户在非文章内容页（比如 App 内的列表页、活动页等）。

- 问题描述：

- 统计活跃间隔对用户分级后，各活跃等级用户占比，结果保留两位小数，且按占比降序排序
- 用户等级标准简化为：忠实用户(近 7 天活跃过且非新晋用户)、新晋用户(近 7 天新增)、沉睡用户(近 7 天未活跃但更早前活跃过)、流失用户(近 30 天未活跃但更早前活跃过)。

- 分析：通过“今日与最晚活跃日期时间差”和“今日与最早活跃日期时间差”共同来确定用户的等级，所以首先生成一个表 a, 主要包含三个字段：uid, 今

日与最晚活跃日期时间差，今日与最早活跃日期时间差；然后，判断的逻辑如下：如果今日与最后活跃日时间差大于 29 天（包含今日在内），那么必然是流失用户；如果今日与最后活跃日时间差小于等于 29 天但大于等于 7 天，那么必然是沉睡用户；如果今日与最后活跃日时间小于等于 6 天，则如果“今日与最晚活跃日期时间差”和“今日与最早活跃日期时间差”相等说明是新晋用户，不相等说明是忠实用户，因此从之前建立的表 a 中可以提取设置新变量用户等级：

- 答案：

```

select
    用户分级,
    round(
        count(uid) / (
            select
                count(distinct uid)
            from
                tb_user_log
        ),
        2
    ) as 比例
from
    (
        select
            uid,
            case
                when 今日与最后活跃日时间差 > 29 then '流失用户'
                when 今日与最后活跃日时间差 <= 29
                    and 今日与最后活跃日时间差 >= 7 then '沉睡用户'
                when 今日与最后活跃日时间差 <= 6
                    and 今日与最后活跃日时间差 = 今日与最早活跃日时间
                差 then '新晋用户'
                else '忠实用户'
            end as 用户分级
        from
            (

```

```
select
    uid,
    datediff(
        (
            select
                max(out_time)
            from
                tb_user_log
        ),
        max(out_time)
    ) as 今日与最后活跃日时间差,
    datediff(
        (
            select
                max(out_time)
            from
                tb_user_log
        ),
        min(in_time)
    ) as 今日与最早活跃日时间差
from
    tb_user_log
group by
    uid
) as a
) as b
group by
    用户分级
order by
    比例 desc;
```

5.每天的日活数及新用户占比

- 题目描述：

- 用户行为日志表 tb_user_log
 - ◆ uid-用户 ID, artical_id-文章 ID
 - ◆ in_time-进入时间, out_time-离开时间, sign_in-是否签到
- 说明：artical_id 为 0 表示用户在非文章内容页（比如 App 内的列表页、活动页等）。
- 问题描述：
 - 统计每天的日活数及新用户占比
 - 新用户占比=当天的新用户数÷当天活跃用户数（日活数）。
- 分析：日活就是每天访问的不同用户数，所以我们首先要得到一张登录表，登录表记录了每天登录的用户，并按天对用户进行了去重，也就是下面的 t1。而要统计新用户的占比，我们就需要识别每天登录用户中哪些用户是新用户（即第一次登录）。一个可行的思路是，使用窗口函数对每个用户的登录日期进行排序得到下面的 t2。统计的时候进行判断，如果统计当天该用户的序号为 1，则表示用户今天是第一次登录，即为新用户
- 答案：

```

with t1 as(      # 用户登录表，记录了用户 id 和登录时间，对每天的登录
  用户进行了去重
    select uid,date(in_time) dt
    from tb_user_log
    union      # union 实现去重，union all 不去重
    select uid,date(out_time) dt
    from tb_user_log
),
t2 as (      # 对每个用户的登录日期进行排序，注册日期的序号是 1
  select
    uid,dt,
    row_number() over(partition by uid order by dt) rn
  from t1
)
select
  dt,
  count(uid) dau,
  round(sum(if(rn=1,1,0))/count(uid),2) uv_new_ration
from t2
  
```

```
group by dt
order by dt;
```

6.连续签到领金币

- 题目描述：
 - 用户行为日志表 tb_user_log
 - ◆ uid-用户 ID, artical_id-文章 ID
 - ◆ in_time-进入时间, out_time-离开时间, sign_in-是否签到
 - 说明：artical_id 为 0 表示用户在非文章内容页（比如 App 内的列表页、活动页等）
 - ◆ 从 2021 年 7 月 7 日 0 点开始，用户每天签到可以领 1 金币，并可以开始累积签到天数，连续签到的第 3、7 天分别可额外领 2、6 金币。每连续签到 7 天后重新累积签到天数（即重置签到天数：连续第 8 天签到时记为新一轮签到的第一天，领 1 金币）
- 问题描述：
 - 计算每个用户 2021 年 7 月以来每月获得的金币数（该活动到 10 月底结束，11 月 1 日开始的签到不再获得金币）。结果按月份、ID 升序排序。
- 分析：1) 首先给 tb_user_log 表做初步的筛选和编号，方便后续做标记；2) 首先登录几天就有几个金币，第 3、7 天有额外金币；3) 利用中间表做编号，由于每隔 7 天就清零，还有换了账号就清零，所以必须 case when 来判断逻辑算金币；需要达到的效果如下；4) 这个表是两个表错一个 ID 相连接，以达到判断连续的逻辑；
- 答案：

WITH t1 AS(-- t1 表筛选出活动期间内的数据，并且为了防止一天有多次签到活动，distinct 去重

```

SELECT
    DISTINCT uid,
    DATE(in_time) dt,
    DENSE_RANK() over(PARTITION BY uid ORDER BY
    DATE(in_time)) rn -- 编号
FROM
    tb_user_log
WHERE
```

```

        DATE(in_time) BETWEEN '2021-07-07' AND '2021-10-31' AND
artical_id = 0 AND sign_in = 1
),
t2 AS (
    SELECT
    *,
    DATE_SUB(dt,INTERVAL rn day) dt_tmp,
    case DENSE_RANK() over(PARTITION BY DATE_SUB(dt,INTERVAL rn
day),uid ORDER BY dt )%7 -- 再次编号
        WHEN 3 THEN 3
        WHEN 0 THEN 7
        ELSE 1
    END as day_coin -- 用户当天签到时应该获得的金币数
    FROM
    t1
)
SELECT
    uid,DATE_FORMAT(dt,'%Y%m') `month`, sum(day_coin) coin -- 总
金币数
    FROM
    t2
    GROUP BY
        uid,DATE_FORMAT(dt,'%Y%m')
    ORDER BY
        DATE_FORMAT(dt,'%Y%m'),uid;

```

电商业务

1. 计算商城中 2021 年每月的 GMV

- 题目描述：
 - 订单总表 tb_order_overall
 - ◆ order_id-订单号, uid-用户 ID, event_time-下单时间, total_amount-订
 单总金额, total_cnt-订单商品总件数, status-订单状态

● 问题描述：

- order_id-订单号, uid-用户 ID, event_time-下单时间, total_amount-订单总金额, total_cnt-订单商品总件数, status-订单状态

● 答案：

```

SELECT
    date_format(event_time, '%Y-%m') `month`,
    sum(total_amount) GMV
from
    tb_order_overall
where
    (status != 2)
    and year(event_time) = 2021
group by
    `month`
having
    GMV > 100000
order by
    GMV;

```

2. 统计 2021 年 10 月每个退货率不大于 0.5 的商品各项指标

● 问题描述：

- 请统计 2021 年 10 月每个有展示记录的退货率不大于 0.5 的商品各项指标。
 - ◆ 商品点展比=点击数÷展示数;
 - ◆ 加购率=加购数÷点击数;
 - ◆ 成单率=付款数÷加购数; 退货率=退款数÷付款数

● 答案：

```

select
    product_id,
    round(click_cnt / show_cnt, 3) as ctr,
    round(IF (click_cnt > 0, cart_cnt / click_cnt, 0), 3) as cart_rate,
    round(IF (cart_cnt > 0, payment_cnt / cart_cnt, 0), 3) as payment_rate,
    round(

```

```

    IF (payment_cnt > 0, refund_cnt / payment_cnt, 0),
    3
) as refund_rate
from
(
select
    product_id,
    COUNT(1) as show_cnt,
    sum(if_click) as click_cnt,
    sum(if_cart) as cart_cnt,
    sum(if_payment) as payment_cnt,
    sum(if_refund) as refund_cnt
from
    tb_user_event
where
    DATE_FORMAT(event_time, '%Y%m') = '202110'
group by
    product_id
) as t_product_index_cnt
where
    payment_cnt = 0
    or refund_cnt / payment_cnt <= 0.5
order by
    product_id;

```

3. 某店铺的各商品毛利率及店铺整体毛利率

- 题目描述：
 - 商品信息表 tb_product_info
 - ◆ product_id-商品 ID, shop_id-店铺 ID, tag-商品类别标签, in_price-进货价格, quantity-进货数量, release_time-上架时间
 - 订单明细表 tb_order_detail
 - ◆ order_id-订单号, product_id-商品 ID, price-商品单价, cnt-下单数量
- 问题描述：

- 请计算 2021 年 10 月以来店铺 901 中商品毛利率大于 24.9%的商品信息及店铺整体毛利率。
 - ◆ 商品毛利率=(1-进价/平均单件售价)*100%;
 - ◆ 店铺毛利率=(1-总进价成本/总销售收入)*100%。
 - ◆ 结果先输出店铺毛利率，再按商品 ID 升序输出各商品毛利率，均保留 1 位小数。

- 答案：

```

SELECT
    product_id,
    CONCAT (profit_rate, "%") as profit_rate
FROM
(
    SELECT
        IFNULL (product_id, '店铺汇总') as product_id,
        ROUND(
            100 * (1 - SUM(in_price * cnt) / SUM(price * cnt)),
            1
        ) as profit_rate
    FROM
    (
        SELECT
            product_id,
            price,
            cnt,
            in_price
        FROM
            tb_order_detail
        JOIN tb_product_info USING (product_id)
        JOIN tb_order_overall USING (order_id)
        WHERE
            shop_id = 901
            and DATE (event_time) >= "2021-10-01"
    ) as t_product_in_each_order
    GROUP BY

```

```

product_id
WITH
    ROLLUP
HAVING
    profit_rate > 24.9
    OR product_id IS NULL
ORDER BY
    product_id
) as t1;

```

4. 零食类商品中复购率 top3 高的商品

- 问题描述：

- 请统计零食类商品中复购率 top3 高的商品。
 - ◆ 复购率指用户在一段时间内对某商品的重复购买比例，复购率越大，则反映出消费者对品牌的忠诚度就越高，也叫回头率
 - ◆ 此处我们定义：某商品复购率 = 近 90 天内购买它至少两次的人数 ÷ 购买它的总人数
 - ◆ 近 90 天指包含最大日期（记为当天）在内的近 90 天。结果中复购率保留 3 位小数，并按复购率倒序、商品 ID 升序排序

- 答案：

```

SELECT
    product_id,
    ROUND(SUM(repurchase) / COUNT(repurchase), 3) as repurchase_rate
FROM
(
    SELECT
        uid,
        product_id,
        IF(COUNT(event_time) > 1, 1, 0) as repurchase
    FROM
        tb_order_detail
        JOIN tb_order_overall USING (order_id)
        JOIN tb_product_info USING (product_id)
)
```

```

WHERE
    tag = "零食"
    AND event_time >= (
        SELECT
            DATE_SUB(MAX(event_time), INTERVAL 89 DAY)
        FROM
            tb_order_overall
    )
GROUP BY
    uid,
    product_id
) as t_uid_product_info
GROUP BY
    product_id
ORDER BY
    repurchase_rate DESC,
    product_id
LIMIT
    3;

```

5.10月的新户客单价和获客成本

- 问题描述：

- 请计算 2021 年 10 月商城里所有新用户的首单平均交易金额（客单价）
和平均获客成本（保留一位小数）。

- 答案：

```

select
    round(sum(total_amount) / count(order_id), 1) avg_amount,
    round(avg(cost), 1) avg_cost
from
    (
        select
            a.order_id,
            total_amount,

```

```

        (sum(price * cnt) - total_amount) as cost
from
    tb_order_detail a
left join tb_order_overall b on a.order_id = b.order_id
where
    date_format(event_time, '%Y-%m') = '2021-10'
    and (uid, event_time) in (
        select
            uid,
            min(event_time) -- 用户和其第一次购买的时间
        from
            tb_order_overall
        GROUP BY
            uid
    )
    GROUP BY
        a.order_id
) a;

```

6. 店铺 901 国庆期间的 7 日动销率和滞销率

- 问题描述：

- 请计算店铺 901 在 2021 年国庆头 3 天的 7 日动销率和滞销率，结果保留 3 位小数，按日期升序排序。
 - ◆ 动销率定义为店铺中一段时间内有销量的商品占当前已上架总商品数的比例（有销量的商品/已上架总商品数）。
 - ◆ 滞销率定义为店铺中一段时间内没有销量的商品占当前已上架总商品数的比例。（没有销量的商品/已上架总商品数）。
 - ◆ 只要当天任一店铺有任何商品的销量就输出该天的结果，即使店铺 901 当天的动销率为 0。

- 答案：

```

SELECT
    dt,
    ROUND(cnt / total_cnt, 3) AS sale_rate,
    ROUND(1 - cnt / total_cnt, 3) AS unsale_rate

```

```

FROM
(
    SELECT DISTINCT
        DATE(event_time) AS dt,
        (
            SELECT
                COUNT(DISTINCT (IF (shop_id != 901, null,
product_id)))
            FROM
                tb_order_overall
                JOIN tb_order_detail USING (order_id)
                JOIN tb_product_info USING (product_id)
            WHERE
                TIMESTAMPDIFF (DAY, event_time, to1.event_time)
            BETWEEN 0 AND 6
        ) AS cnt,
        (
            SELECT
                COUNT(DISTINCT product_id)
            FROM
                tb_product_info
            WHERE
                shop_id = 901
        ) AS total_cnt
    FROM
        tb_order_overall to1
    WHERE
        DATE(event_time) BETWEEN '2021-10-01' AND '2021-10-03'
) AS t0
ORDER BY
dt;

```

打车业务

1.2021 年国庆在北京接单 3 次及以上的司机统计信息

- 题目描述：

- 用户打车记录表 tb_get_car_record
 - ◆ uid-用户 ID, city-城市, event_time-打车时间, end_time-打车结束时间, order_id-订单号
- 打车订单表 tb_get_car_order
 - ◆ order_id-订单号, uid-用户 ID, driver_id-司机 ID,
 - ◆ order_time-接单时间, start_time-开始计费的上车时间, finish_time-订单完成时间, mileage-行驶里程数, fare-费用, grade-评分

- 问题描述：

- 请统计 2021 年国庆 7 天期间在北京市接单至少 3 次的司机的平均接单数和平均兼职收入(暂不考虑平台佣金, 直接计算完成的订单费用总额), 结果保留 3 位小数。

- 答案：

```

SELECT
    "北京" as city,
    ROUND(AVG(order_num), 3) as avg_order_num,
    ROUND(AVG(income), 3) as avg_income
FROM
(
    SELECT
        driver_id,
        COUNT(order_id) as order_num,
        SUM(fare) as income
    FROM
        tb_get_car_order
    JOIN tb_get_car_record USING (order_id)
    WHERE
        city = "北京"
        and DATE_FORMAT (order_time, "%Y%m%d") BETWEEN
        '20211001' AND '20211007'
)
```

```

    GROUP BY
        driver_id
    HAVING
        COUNT(order_id) >= 3
    ) as t_driver_info;

```

2.有取消订单记录的司机平均评分

- 问题描述：

- 请找到 2021 年 10 月有过取消订单记录的司机，计算他们每人全部已完成的有评分订单的平均评分及总体平均评分，保留 1 位小数。先按 driver_id 升序输出，再输出总体情况。

- 答案：

```

SELECT
    COALESCE(driver_id, '总体'),
    round(sum(grade) / count(grade), 1)
from
    tb_get_car_order
where
    driver_id in (
        select
            driver_id
        from
            tb_get_car_order
        where
            start_time is null
            and DATE_FORMAT(finish_time, '%Y-%m') = '2021-10'
    )
group by
    driver_id
WITH
    ROLLUP;

```

3. 每个城市中评分最高的司机信息

- 问题描述：

- 请统计每个城市中评分最高的司机平均评分、日均接单量和日均行驶里程数。

- 答案：

```

with
city_driver as (
    SELECT
        city,
        driver_id,
        round(avg(grade), 1) as avg_grade,
        round(
            count(order_time) / count(DISTINCT DATE_FORMAT
(order_time, '%Y%m%d'))),
            1
        ) as avg_order_num,
        round(
            sum(mileage) / count(DISTINCT DATE_FORMAT
(order_time, '%Y%m%d'))),
            3
        ) as avg_mileage,
        rank() over (
            partition by
                city
            order by
                round(avg(grade), 1) desc
        ) as rk
from
    tb_get_car_order
    join tb_get_car_record using (order_id)
group by
    driver_id,
    city
)

```

```

)
select
    city,
    driver_id,
    avg_grade,
    avg_order_num,
    avg_mileage
from
    city_driver
where
    rk = 1
order by
    avg_order_num

```

4. 国庆期间近 7 日日均取消订单量

- 问题描述：

- 请统计国庆头 3 天里，每天的近 7 日日均订单完成量和日均订单取消量，按日期升序排序。结果保留 2 位小数。

- 答案：

```

SELECT
    dt,
    finish_num_7d,
    cancel_num_7d
FROM
    (
        SELECT
            dt,
            ROUND(
                AVG(finish_num) over (
                    ORDER BY
                        dt ROWS 6 preceding
                ),
                2
            )
    ) AS subquery
WHERE
    dt BETWEEN '2023-10-01' AND '2023-10-03'
ORDER BY
    dt

```

```

) as finish_num_7d,
ROUND(
    AVG(cancel_num) over (
        ORDER BY
            dt ROWS 6 preceding
    ),
    2
) as cancel_num_7d
FROM
(
    SELECT
        dt,
        SUM(is_finish) as finish_num,
        COUNT(1) - SUM(is_finish) as cancel_num
    FROM
    (
        SELECT
            DATE(order_time) as dt,
            IF(start_time IS NULL, 0, 1) as is_finish
        FROM
            tb_get_car_order
        WHERE
            DATE(order_time) BETWEEN '2021-09-25'
and '2021-10-03'
        ) as t_order_status
    GROUP BY
        dt
    ) as t_finish_cancel_daily
) as t_finish_cancel_7d
WHERE
dt >= '2021-10-01';

```

5. 工作日各时段叫车量、等待接单时间和调度时间

- 问题描述：

- 统计周一到周五各时段的叫车量、平均等待接单时间和平均调度时间。
全部以 event_time-开始打车时间为时段划分依据，平均等待接单时间和平均调度时间均保留 1 位小数，平均调度时间仅计算完成了的订单，结果按叫车量升序排序。
- 不同时段定义：早高峰 [07:00:00 , 09:00:00)、工作时间 [09:00:00 , 17:00:00)、晚高峰 [17:00:00 , 20:00:00)、休息时间 [20:00:00 , 07:00:00)

- 答案：

```

select
    period,
    count(get_car) as get_car_num,
    round(avg(wait_time / 60), 1) as avg_wait_time,
    round(avg(dispatch_time / 60), 1) as avg_dispatch_time
from
(
    select
        o.order_id as get_car,
        case
            when hour(event_time) >= 7
                and hour(event_time) < 9 then '早高峰'
            when hour(event_time) >= 9
                and hour(event_time) < 17 then '工作时间'
            when hour(event_time) >= 17
                and hour(event_time) < 20 then '晚高峰'
            else '休息时间'
        end as period,
        timestampdiff(second, event_time, order_time) as wait_time,
        timestampdiff(second, order_time, start_time) as dispatch_time
    from
        tb_get_car_record r
        left join tb_get_car_order o using (order_id)
    where

```

```

    weekday(event_time) between 0 and 4
) t1
group by
    period
order by
    count(get_car);

```

6.各城市最大同时等车人数

- 问题描述：

- 请统计各个城市在 2021 年 10 月期间，单日中最大的同时等车人数。

- 答案：

```

WITH
    t1 AS (
        SELECT
            city,
            SUM(uv) OVER (
                PARTITION BY
                    city
                ORDER BY
                    uv_time,
                    uv DESC
            ) AS uv_cnt
    )
FROM
    (
        SELECT
            city,
            event_time uv_time,
            1 AS uv
        FROM
            tb_get_car_record
        UNION ALL
        SELECT
            city,

```

```

        end_time uv_time,
        -1 AS uv
    FROM
        tb_get_car_record
    WHERE
        order_id IS NULL
    UNION ALL
    SELECT
        city,
        IFNULL (start_time, finish_time) uv_time,
        -1 AS uv
    FROM
        tb_get_car_order
    LEFT JOIN tb_get_car_record USING (order_id)
    ) AS t
    WHERE
        DATE_FORMAT (uv_time, '%Y%m') = '202110'
    )
SELECT
    city,
    MAX(uv_cnt) max_wait_uv
FROM
    t1
GROUP BY
    city
ORDER BY
    max_wait_uv,
    city;

```

店铺分析业务

1. 某宝店铺的 SPU 数量

- 题目描述：

- 产品情况表 product_tb
 - ◆ item_id 指某款号的具体货号, style_id 指款号, tag_price 表示标签价格, inventory 指库存量
- 问题描述:
 - 请你统计每款的 SPU (货号) 数量, 并按 SPU 数量降序排序
- 答案:

```
select
    style_id,
    count(style_id) as SPU_num
from
    product_tb
group by
    style_id
order by
    SPU_num desc
```

2. 某宝店铺的实际销售额与客单价

- 题目描述:
 - 11 月份销售数据表 sales_tb
 - ◆ sales_date 表示销售日期, user_id 指用户编号, item_id 指货号, sales_num 表示销售数量, sales_price 表示结算金额
- 问题描述:
 - 请你统计实际总销售额与客单价 (人均付费, 总收入/总用户数, 结果保留两位小数)
- 答案:

```
select
    sum(sales_price) as sales_total,
    round(sum(sales_price) / count(distinct user_id), 2) as per_trans
from
    sales_tb
```

3.某宝店铺折扣率

- 问题描述：

■ 请你统计折扣率（GMV/吊牌金额，GMV 指的是成交金额）

- 答案：

```
SELECT
    ROUND(
        SUM(sales_price) * 100 / SUM(sales_num * tag_price),
        2
    ) 'discount_rate(%)'
FROM
    sales_tb
LEFT JOIN product_tb USING (item_id);
```

4.某宝店铺动销率与售罄率

- 问题描述：

■ 请你统计每款的动销率（pin_rate，有销售的SKU数量/在售SKU数量）
与售罄率（sell-through_rate，GMV/备货值，备货值=吊牌价*库存数），
按 style_id 升序排序

- 答案：

```
select
    style_id,
    round(100 * sum(num) / (sum(inventory) - sum(num)), 2) as 'pin_rate(%)',
    round(
        100 * sum(item_GMV) / sum(inventory * tag_price),
        2
    ) as sell-through_rate(%)
from
    product_tb a
    join (
        select
            item_id,
            sum(sales_num) as num,
```

```

    sum(sales_price) as item_GMV
from
    sales_tb
group by
    item_id
) as b on a.item_id = b.item_id
group by
    style_id
order by
    style_id

```

5.某宝店铺连续 2 天及以上购物的用户及其对应的天数

- 问题描述：

- 请你统计连续 2 天及以上在该店铺购物的用户及其对应的次数（若有很多用户，按 user_id 升序排序）

- 答案：

```

SELECT
    user_id,
    COUNT(*) days_count
FROM
(
    SELECT DISTINCT
        user_id,
        sales_date,
        DENSE_RANK() over (
            PARTITION by
                user_id
            ORDER BY
                sales_date
        ) rn
    FROM
        sales_tb

```

```

) a
GROUP BY
    user_id,
    DATE_ADD(sales_date, INTERVAL - rn day)
HAVING
    days_count >= 2
ORDER BY
    user_id

```

教育业务

1.牛客直播转换率

- 题目描述：
 - 课程表 course_tb
 - ◆ course_id 代表课程编号, course_name 表示课程名称, course_datetime 代表上课时间
 - 用户行为表 behavior_tb
 - ◆ user_id 表示用户编号、if_vw 表示是否浏览、if_fav 表示是否收藏、if_sign 表示是否报名、course_id 代表课程编号
- 问题描述：
 - 请你统计每个科目的转换率(sign_rate(%)), 转化率=报名人数/浏览人数, 结果保留两位小数)
- 答案：

```

SELECT
    course_id,
    course_name,
    ROUND(SUM(if_sign) * 100 / SUM(if_vw), 2) 'sign_rate(%)'
FROM
    behavior_tb
    JOIN course_tb USING (course_id)
GROUP BY
    course_id,

```

```
course_name
ORDER BY
course_id;
```

2.牛客直播开始时各直播间在线人数

- 题目描述：
 - 上课情况表 attend_tb
 - ◆ user_id 表示用户编号、course_id 代表课程编号、in_datetime 表示进入直播间的时间、out_datetime 表示离开直播间的时间
- 问题描述：
 - 请你统计直播开始时（19: 00），各科目的在线人数
- 答案：

```
select a.course_id,
       b.course_name,
       count(distinct user_id) online_num
  from attend_tb a
  left join course_tb b
    on a.course_id = b.course_id
   where '19:00' between DATE_FORMAT(in_datetime,'%H:%i') and
DATE_FORMAT(out_datetime,'%H:%i')
 group by a.course_id,b.course_name
order by a.course_id
```

3.牛客直播各科目平均观看时长

- 问题描述：
 - 请你统计每个科目的平均观看时长（观看时长定义为离开直播间的时间与进入直播间的时间之差，单位是分钟），输出结果按平均观看时长降序排序，结果保留两位小数。
- 答案：

```
select
       course_name,
       round(
           avg(timestampdiff(minute, in_datetime, out_datetime)),
```

```

2
) as avg_Len
from
course_tb
left join attend_tb using (course_id)
group by
course_name
order by
avg_Len desc

```

4.牛客直播各科目出勤率

- 题目描述：

- 用户行为表 behavior_tb
 - ◆ user_id 表示用户编号、if_vw 表示是否浏览、if_fav 表示是否收藏、if_sign 表示是否报名、course_id 代表课程编号

- 问题描述：

- 请你统计每个科目的出勤率（attend_rate(%), 结果保留两位小数），出勤率=出勤（在线时长 10 分钟及以上）人数 / 报名人数，输出结果按 course_id 升序排序

- 答案：

```

with
a as (
    select
        course_id,
        course_name
    from
        course_tb
),
b as (
    select
        course_id,
        sum(if_sign) total_num
    from

```

```

behavior_tb
group by
course_id
),
c as (
select
course_id,
count(
distinct case
when timestampdiff (minute, in_datetime,
out_datetime) >= 10 then user_id
else null
end
) cc
from
attend_tb
group by
course_id
)
select
course_id,
course_name,
round(round(c.cc / b.total_num, 4) * 100, 2) attend_rate
from
a
join b using (course_id)
join c using (course_id)

```

5.牛客直播各科目同时在线人数

- 问题描述：

- 请你统计每个科目最大同时在线人数（按 course_id 排序）

- 答案：

```
SELECT
```

```
course_id,  
course_name,  
MAX(uv_cnt) max_num  
FROM  
(  
    SELECT  
        course_id,  
        course_name,  
        SUM(uv) OVER (  
            PARTITION BY  
                course_id  
            ORDER BY  
                dt,  
                uv DESC  
        ) uv_cnt  
    FROM  
(  
        SELECT  
            course_id,  
            user_id,  
            in_datetime dt,  
            1 AS uv  
        FROM  
            attend_tb  
        UNION ALL  
        SELECT  
            course_id,  
            user_id,  
            out_datetime dt,  
            -1 AS uv  
        FROM  
            attend_tb  
) uv_tb  
JOIN course_tb USING (course_id)
```

```

) t1
GROUP BY
    course_id,
    course_name
ORDER BY
    course_id;

```

内容业务

1.某乎问答 11 月份日人均回答量

- 题目描述：
 - 某乎问答创作者回答情况表 answer_tb
 - ◆ answer_date 表示创作日期、author_id 指创作者编号
 - ◆ issue_id 表示问题 id、char_len 表示回答字数
- 问题描述：
 - 请你统计 11 月份日人均回答量（回答问题数量/答题人数），按回答日期排序，结果保留两位小数
- 答案：

```

select
    answer_date,
    round(count(issue_id) / count(DISTINCT author_id), 2) per_num
from
    answer_tb
where
    month(answer_date) = 11
group by
    answer_date

```

2.某乎问答高质量的回答中用户属于各级别的数量

- 题目描述：
 - 某乎问答创作者信息表 author_tb

◆ author_id 表示创作者编号、author_level 表示创作者级别，共 1-6 六个级别、sex 表示创作者性别说明：

- 问题描述：

- 回答字数大于等于 100 字的认为是高质量回答，请你统计某乎问答高质量的回答中用户属于 1-2 级、3-4 级、5-6 级的数量分别是多少，按数量降序排列

- 答案：

```
SELECT
    case
        when b.author_level in (1,2) then '1-2 级'
        when b.author_level in (3,4) then '3-4 级'
        when b.author_level in (5,6) then '5-6 级'
        else " end as level_cut,
    count(issue_id) num
from answer_tb a
left join author_tb b
on a.author_id = b.author_id
where char_len >= 100
group by level_cut
order by num desc;
```

3. 某乎问答单日回答问题数大于等于 3 个的所有用户

- 问题描述：

- 请你统计 11 月份单日回答问题数大于等于 3 个的所有用户信息（author_date 表示回答日期、author_id 表示创作者 id，answer_cnt 表示回答问题个数）

- 答案：

```
select
    answer_date,
    author_id,
    count(issue_id) as answer_cnt
from answer_tb
where month(answer_date) = 11
```

```
group by answer_date,author_id
having count(issue_id) >= 3
order by answer_date;
```

4.某乎问答回答过教育类问题的用户里有多少用户回答过职场类问题

- 题目描述:
 - 某乎问答题目信息表 issue_tb
 - ◆ issue_id 代表问题编号, issue_type 表示问题类型
- 问题描述:
 - 请你统计回答过教育类问题的用户里有多少用户回答过职场类问题
- 答案:

```
select count(distinct author_id) num
from issue_tb t1
join answer_tb t2
on t1.issue_id=t2.issue_id
where issue_type = 'Education'
and author_id in (
    select author_id
    from issue_tb a
    join answer_tb b
    on a.issue_id=b.issue_id
    where issue_type = 'Career'
);
```

5.某乎问答最大连续回答问题天数大于等于 3 天的用户及其对应等级

- 问题描述:
 - 请你统计最大连续回答问题的天数大于等于 3 天的用户及其等级（若有
多条符合条件的数据，按 author_id 升序排序）
- 答案:

```

select
    a.author_id,
    b.author_level,
    a.days_cnt
FROM
(
    select
        t1.author_id,
        count(distinct t1.answer_date) days_cnt
    from
    (
        select
            t.author_id,
            t.answer_date,
            t.answer_date - t.ranking dt
    FROM
    (
        select
            author_id,
            answer_date,
            dense_rank() over (
                PARTITION BY
                    author_id
                order by
                    answer_date
            ) ranking
    from
        answer_tb
    ) t
    ) t1
GROUP BY
    t1.author_id,
    t1.dt
having

```

```

        count(distinct t1.answer_date) >= 3
    ) a
    LEFT JOIN author_tb b on a.author_id = b.author_id
order by
    a.author_id;

```

第九部分 企业级调优手法

Hadoop

1.job 执行三原则

原则一、充分利用集群资源

Job 运行时，尽量让所有的节点都有任务处理，这样能尽量保证集群资源被充分利用，任务的并发度达到最大。可以通过调整处理的数据量大小，以及调整 map 和 reduce 个数来实现。

- ◆ Reduce 个数的控制使用 “mapreduce.job.reduces”
- ◆ Map 个数取决于使用了哪种 InputFormat，默认的 TextFileInputFormat 将根据 block 的个数来分配 map 数(一个 block 一个 map)。

原则二、ReduceTask 并发调整

努力避免出现以下场景

- ◆ 观察 Job 如果大多数 ReduceTask 在第一轮运行完后，剩下很少甚至一个 ReduceTask 刚开始运行。这种情况下，这个 ReduceTask 的执行时间将决定了该 job 的运行时间。可以考虑将 reduce 个数减少。
- ◆ 观察 Job 的执行情况如果是 MapTask 运行完成后，只有个别节点有 ReduceTask 在运行。这时候集群资源没有得到充分利用，需要增加 Reduce 的并行度以便每个节点都有任务处理。

原则三、Task 执行时间要合理

一个 job 中，每个 MapTask 或 ReduceTask 的执行时间只有几秒钟，这就意味着这个 job 的大部分时间都消耗在 task 的调度和进程启停上了，因此可以考虑增加每个 task 处理的数据大小。建议一个 task 处理时间为 1 分钟。

2.shuffle 调优

Shuffle 阶段是 MapReduce 性能的关键部分，包括了从 MapTask 将中间数据写到磁盘一直到 ReduceTask 拷贝数据并最终放到 Reduce 函数的全部过程。这一块 Hadoop 提供了大量的调优参数。

1) map 阶段

第一、判断 map 内存使用

- ◆ 判断 Map 分配的内存是否够用，可以查看运行完成的 job 的 Counters 中(历史服务器)，对应的 task 是否发生过多次 GC，以及 GC 时间占总 task 运行时间之比。通常，GC 时间不应超过 task 运行时间的 10%，即 $GC\ time\ elapsed\ (ms)/CPU\ time\ spent\ (ms) < 10\%$ 。
- ◆ Map 需要的内存还需要随着环形缓冲区的调大而对应调整。可以通过如下参数进行调整。`mapreduce.map.memory.mb`
- ◆ Map 需要的 CPU 核数可以通过如下参数调整。`mapreduce.map.cpu.vcores`
- ◆ 如果集群资源充足建议调整：
 - `mapreduce.map.memory.mb=3G`(默认 1G)`mapreduce.map.cpu.vcores=1`(默认也是 1)

环形缓冲区：

Map 方法执行后首先把数据写入环形缓冲区，为什么 MR 框架选择先写内存而不是直接写磁盘？这样的目的主要是为了减少磁盘 i/o

- ◆ 环形缓冲默认 100M (`mapreduce.task.io.sort.mb`)，当到达 80% (`mapreduce.map.sort.spill.percent`) 时就会溢写磁盘。
- ◆ 每达到 80% 都会重写溢写到一个新的文件。

当集群内存资源充足，考虑增大 `mapreduce.task.io.sort.mb` 提高溢写的效率，而且会减少中间结果的文件数量。

建议：

- ◆ 调整 `mapreduce.task.io.sort.mb=512M`。
- ◆ 当文件溢写完后，会对这些文件进行合并，默认每次合并 10 (`mapreduce.task.io.sort.factor`) 个溢写的文件，建议调整 `mapreduce.task.io.sort.factor=64`。这样可以提高合并的并行度，减少合并的次数，降低对磁盘操作的次数。

第二、combiner

在 Map 阶段，有一个可选过程，将同一个 key 值的中间结果合并，叫做 Combiner。(一般将 reduce 类设置为 combiner 即可)通过 Combine，一般情况下可以显著减少 Map 输出的中间结果，从而减少 shuffle 过程的网络带宽占用。

建议：不影响最终结果的情况下，加上 Combiner!!

2) copy 阶段

- ◆ 对 Map 的中间结果进行压缩，当数据量大时，会显著减少网络传输的数据量，
- ◆ 但是也因为多了压缩和解压，带来了更多的 CPU 消耗。因此需要做好权衡。
当任务属于网络瓶颈类型时，压缩 Map 中间结果效果明显。
- ◆ 在实际经验中 Hadoop 的运行的瓶颈一般都是 IO 而不是 CPU，压缩一般可以 10 倍的减少 IO 操作

3) reduce 阶段

第一、reduce 资源

mapreduce.reduce.memory.mb=5G (默认 1G)

mapreduce.reduce.cpu.vcores=1 (默认为 1)

第二、copy

ReduceTask 在 copy 的过程中默认使用 5 (mapreduce.reduce.shuffle.parallelcopies 参数控制) 个并行度进行复制数据。该值在实际服务器上比较小，建议调整为 50-100.

第三、溢写合并

- ◆ Copy 过来的数据会先放入内存缓冲区中，然后当使用内存达到一定量的时候 spill 磁盘。这里的缓冲区大小要比 map 端的更为灵活，它基于 JVM 的 heap size 设置。这个内存大小的控制是通过 mapreduce.reduce.shuffle.input.buffer.percent (default 0.7) 控制的。
- ◆ shuffle 在 reduce 内存中的数据最多使用内存量为: $0.7 \times \text{maxHeap of reduce task}$ ，内存到磁盘 merge 的启动可以通過 mapreduce.reduce.shuffle.merge.percent (default 0.66) 配置。
- ◆ copy 完成后，reduce 进入归并排序阶段，合并因子默认为 10 (mapreduce.task.io.sort.factor 参数控制)，如果 map 输出很多，则需要合并很多趟，所以可以提高此参数来减少合并次数。

3.job 调优

1) 推测执行

集群规模很大时（几百上千台节点的集群），个别机器出现软硬件故障的概率就变大了，并且会因此延长整个任务的执行时间。推测执行通过将一个 task 分给多台机器跑，取先运行完的那个，会很好的解决这个问题。对于小集群，可以将这个功能关闭。

建议：

- ◆ 大型集群建议开启，小集群建议关闭！
 - ◆ 集群的推测执行都是关闭的。在需要推测执行的作业执行的时候开启
- 2) slow start

MapReduce 的 AM 在申请资源的时候，会一次性申请所有的 Map 资源，延后申请 reduce 的资源，这样就能达到先执行完大部分 Map 再执行 Reduce 的目的。

`mapreduce.job.reduce.slowstart.completedmaps`: 当多少占比的 Map 执行完后开始执行 Reduce。默认 5% 的 Map 跑完后开始起 Reduce。如果想要 Map 完全结束后执行 Reduce 调整该值为 1

3) 小文件优化

- ◆ HDFS: hadoop 的存储每个文件都会在 NameNode 上记录元数据，如果同样大小的文件，文件很小的话，就会产生很多文件，造成 NameNode 的压力。
- ◆ MR: Mapreduce 中一个 map 默认处理一个分片或者一个小文件，如果 map 的启动时间都比数据处理的时间还要长，那么就会造成性能低，而且在 map 端溢写磁盘的时候每一个 map 最终会产生 reduce 数量个数的中间结果，如果 map 数量特别多，就会造成临时文件很多，而且在 reduce 拉取数据的时候增加磁盘的 IO。

如何处理小文件？

- ◆ 从源头解决，尽量在 HDFS 上不存储小文件，也就是数据上传 HDFS 的时候就合并小文件
- ◆ 通过运行 MR 程序合并 HDFS 上已经存在的小文件
- ◆ MR 计算的时候可以使用 `CombineTextInputFormat` 来降低 MapTask 并行度

4) 数据倾斜

前文有很详细的解决方案

4.yarn 调优

1) NM 配置

可用内存: 刨除分配给操作系统、其他服务的内存外，剩余的资源应尽量分配给 YARN。默认情况下，Map 或 Reduce container 会使用 1 个虚拟 CPU 内核和 1024MB 内存，ApplicationMaster 使用 1536MB 内存。

`yarn.nodemanager.resource.memory-mb` 默认是 8192

CPU 虚拟核心数: 建议将此配置设定在逻辑核数的 1.5~2 倍之间。如果 CPU 的计算能力要求不高，可以配置为 2 倍的逻辑 CPU。

`yarn.nodemanager.resource.cpu-vcores`

2) container 启动模式

默认，YARN 为每一个 Container 启动一个 JVM，JVM 进程间不能实现资源共享，导致资源本地化的时间开销较大。针对启动时间较长的问题，新增了基于线程资源本地化启动模式，能够有效提升 container 启动效率。

- ◆ 设置为“org.apache.hadoop.yarn.server.nodemanager.DefaultContainerExecutor”，则每次启动 container 将会启动一个线程来实现资源本地化。该模式下，启动时间较短，但无法做到资源（CPU、内存）隔离。
- ◆ 设置为“org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor”，则每次启动 container 都会启动一个 JVM 进程来实现资源本地化。该模式下，启动时间较长，但可以提供较好的资源（CPU、内存）隔离能力。

3) AM 调优

- ◆ 运行的一个大任务，map 总数达到了上万的规模，任务失败，发现是 ApplicationMaster（以下简称 AM）反应缓慢，最终超时失败。
- ◆ 失败原因是 Task 数量变多时，AM 管理的对象也线性增长，因此就需要更多的内存来管理。AM 默认分配的内存大小是 1.5GB。

建议：任务数量多时增大 AM 内存 【yarn.app.mapreduce.am.resource.mb】

5.namenode full gc

1) 对象分代

- ◆ 新生成的对象首先放到年轻代 Eden 区，
- ◆ 当 Eden 空间满了，触发 Minor GC，存活下来的对象移动到 Survivor0 区，
- ◆ Survivor0 区满后触发执行 Minor GC，Survivor0 区存活对象移动到 Survivor1 区，这样保证了一段时间内总有一个 survivor 区为空。
- ◆ 经过多次 Minor GC 仍然存活的对象移动到老年代。
- ◆ 老年代存储长期存活的对象，占满时会触发 Major GC（Full GC），GC 期间会停止所有线程等待 GC 完成，所以对响应要求高的应用尽量减少发生 Major GC，避免响应超时。

2) JStat

查看当前 jvm 内存使用以及垃圾回收情况

3) GC 日志解析

总结在 HDFS Namenode 内存中的对象大都是文件，目录和 blocks，这些数据只要不被程序或者数据的拥有者人为的删除，就会在 Namenode 的运行生命期内一直存在，所以这些对象通常是存在在 old 区中，所以，如果整个 hdfs 文件和目录数多，blocks 数也多，内存数据也会很大，如何降低 Full GC 的影响？

- ◆ 计算 NN 所需的内存大小，合理配置 JVM

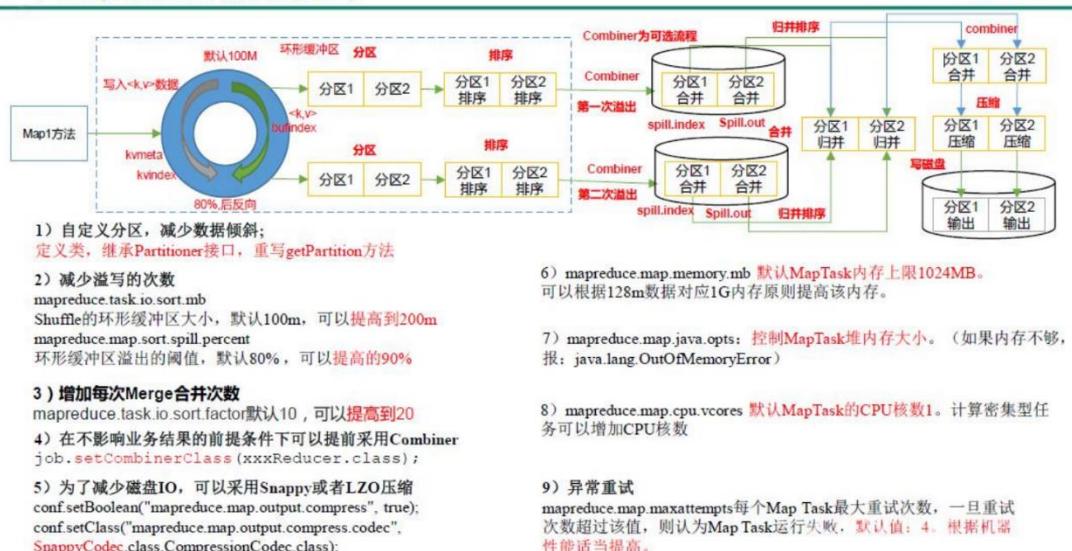
◆ 使用低卡顿 G1 收集器

6.mapreduce 生产调优

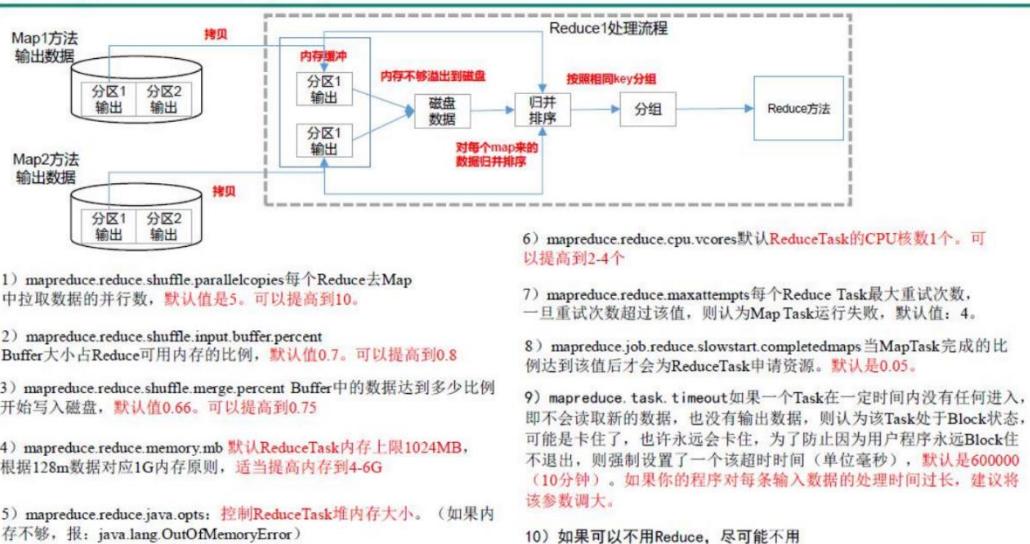
MapReduce 跑的慢的原因

- (1) 计算机性能：CPU、内存、磁盘、网络
- (2) I/O 操作优化：数据倾斜；Map 运行时间太长，导致 Reduce 等待过久；小文件过多

MapReduce优化 (上)



MapReduce优化 (下)



Hive

1. 避免使用 count(distinct col)

原因：

distinct 会将 col 列所有的数据保存到内存中，形成一个类似 hash 的结构，速度是十分的快；但是在大数据背景下，因为 col 列所有的值都会形成以 key 值，极有可能发生 OOM

解决方案：

所以，可以考虑使用 Group By 或者 ROW_NUMBER() OVER(PARTITION BY col) 方式代替 COUNT(DISTINCT col)

2. 小文件问题

原因：

- 众所周知，小文件在 HDFS 中存储本身就会占用过多的内存空间，那么对于 MR 查询过程中过多的小文件又会造成启动过多的 Mapper Task，每个 Mapper 都是一个后台线程，会占用 JVM 的空间
- 在 Hive 中，动态分区会造成在插入数据过程中，生成过多零碎的小文件（请回忆昨天讲的动态分区的逻辑）
- 不合理的 Reducer Task 数量的设置也会造成小文件的生成，因为最终 Reducer 是将数据落地到 HDFS 中的

解决方案：

在数据源头 HDFS 中控制小文件产生的个数，比如

- 采用 Sequencefile 作为表存储格式，不要用 textfile，在一定程度上可以减少小文件（常见于在流计算的时候采用 Sequencefile 格式进行存储）
- 减少 reduce 的数量（可以使用参数进行控制）
- 慎重使用动态分区，最好在分区中指定分区字段的 val 值
- 最好数据的校验工作，比如通过脚本方式检测 hive 表的文件数量，并进行文件合并
- 合并多个文件数据到一个文件中，重新构建表

3. 避免使用 select *

原因：

在大数据量多字段的数据表中，如果使用 SELECT * 方式去查询数据，会造成很多无效数据的处理，会占用程序资源，造成资源的浪费

解决方案：

在查询数据表时，指定所需的待查字段名，而非使用 * 号

4. 不要在表关联之后加 where

原因：

比如以下语句：

```
SELECT * FROM stu as t  
LEFT JOIN course as t1  
ON t.id=t2.stu_id  
WHERE t.age=18;
```

请思考上面语句是否具有优化的空间？如何优化？

解决方案：

- ◆ 采用谓词下推的技术，提早进行过滤有可能减少必须在数据库分区之间传递的数据量
- ◆ 谓词下推的解释：
 - 所谓谓词下推就是通过嵌套的方式，将底层查询语句尽量推到数据底层去过滤，这样在上层应用中就可以使用更少的数据量来查询，这种 SQL 技巧被称为谓词下推(Predicate pushdown)

那么上面语句就可以采用这种方式来处理：

```
SELECT * FROM (SELECT * FROM stu WHERE age=18) as t  
LEFT JOIN course AS t1  
on t.id=t1.stu_id
```

5. 尽量删除字段中带有空值的数据

原因：

一个表内有许多空值时会导致 MapReduce 过程中，空成为 key 值，对应的会有

大量的 value 值，而一个 key 的 value 会一起到达 reduce 造成内存不足

解决方式：

1、在查询的时候，过滤掉所有为 NULL 的数据，比如：

```
create table res_tbl as
select n.* from
(select * from res where id is not null ) n
left join org_tbl o on n.id = o.id;
```

2、查询出空值并给其赋上随机数，避免了 key 值为空（数据倾斜中常用的一种技巧）

```
create table res_tbl as
select n.* from res n
full join org_tbl o on
case when n.id is null then concat('hive', rand()) else n.id end = o.id;
```

6. 设置并行执行任务数

- ◆ 通过设置参数 `hive.exec.parallel` 值为 `true`，就可以开启并发执行。不过，在共享集群中，需要注意下，如果 job 中并行阶段增多，那么集群利用率就会增加。

```
set hive.exec.parallel=true; //打开任务并行执行
set hive.exec.parallel.thread.number=16; //同一个 sql 允许最大并行度，默认为 8
```

7. 设置合理的 Reducer 个数

原因：

- 过多的启动和初始化 reduce 也会消耗时间和资源
- 有多少个 Reducer 就会有多少个文件产生，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

解决方案：

- Reducer 设置的原则：
 - 每个 Reduce 处理的数据默认是 256MB
 - ◆ `hive.exec.reducer.bytes.per.reducer=256000000`
 - 每个任务最大的 reduce 数，默认为 1009

- ◆ hive.exec.reducers.max=1009
- 计算 reduce 数的公式
 - $N = \min(\text{参数 2}, \frac{\text{总输入数据量}}{\text{参数 1}})$
- 设置 Reducer 的数量
 - set mapreduce.job.reduces=n

8.开启 JVM 重用

- JVM 重用是 Hadoop 中调优参数的内容，该方式对 Hive 的性能也有很大的帮助，特别对于很难避免小文件的场景或者 Task 特别多的场景，这类场景大数据书执行时间都很短
- Hadoop 的默认配置通常是使用派生 JVM 来执行 map 和 reduce 任务的，会造成 JVM 的启动过程比较大的开销，尤其是在执行 Job 包含有成百上千个 task 任务的情况下。
- JVM 重用可以使得 JVM 实例在同一个 job 中重新使用 N 次，N 的值可以在 hadoop 的 mapred-site.xml 文件中进行设置

```
<property>
<name>mapred.job.reuse.jvm.num.tasks</name>
<value>10</value>
</property>
```

9.为什么任务执行的时候只有一个 reduce

原因：

- ◆ 使用了 Order by （Order By 是会进行全局排序）
- ◆ 直接 COUNT(1),没有加 GROUP BY，比如：
 - SELECT COUNT(1) FROM table WHERE pt='201909'
- ◆ 有笛卡尔积操作

解决方案：

- ◆ 避免使用全局排序，可以使用 sort by 进行局部排序
- ◆ 使用 GROUP BY 进行统计，不会进行全局排序，比如：
 - SELECT pt, COUNT(1) FROM table WHERE pt='201909';

10.选择使用 Tez 引擎

- ◆ Tez: 是基于 Hadoop Yarn 之上的 DAG(有向无环图, Directed Acyclic Graph)

计算框架。它把Map/Reduce 过程拆分成若干个子过程，同时可以把多个Map/Reduce 任务组合成一个较大的 DAG 任务，减少了Map/Reduce 之间的文件存储。同时合理组合其子过程，也可以减少任务的运行时间

- ◆ 设置 `hive.execution.engine = tez;`
- ◆ 通过上述设置，执行的每个 HIVE 查询都将利用 Tez
- ◆ 当然，也可以选择使用 spark 作为计算引擎

11.选择使用本地模式

有时候 Hive 处理的数据量非常小，那么在这种情况下，为查询出发执行任务的时间消耗可能会比实际 job 的执行时间要长，对于大多数这种情况，hive 可以通过本地模式在单节点上处理所有任务，对于小数据量任务可以大大的缩短时间，可以通过 `hive.exec.mode.local.auto=true`

12.选择使用严格模式

- ◆ Hive 提供了一种严格模式，可以防止用户执行那些可能产生意想不到的不好的影响查询，比如：
 - 对于分区表，除非 WHERE 语句中含有分区字段过滤条件来限制数据范围，否则不允许执行，也就是说不允许扫描所有分区
 - 使用 ORDER BY 语句进行查询是，必须使用 LIMIT 语句，因为 ORDER BY 为了执行排序过程会将所有结果数据分发到同一个 reduce 中进行处理，强制要求用户添加 LIMIT 可以防止 reducer 额外的执行很长时间
- ◆ 严格模式的配置：`hive.mapred.mode=strict`

Spark

1.常规性能调优

常规性能调优一：最优资源配置

Spark 性能调优的第一步，就是为任务分配更多的资源，在一定范围内，增加资源的分配与性能的提升是成正比的，实现了最优的资源配置后，在此基础上再考虑进行后面论述的性能调优策略。

名称	说明	
--num-executors	配置Executor的数量	50~100
--driver-memory	配置Driver内存 (影响不大)	1~5G
--executor-memory	配置每个Executor的内存大小	6~10G
--executor-cores	配置每个Executor的CPU core数量	3

常规性能调优二： RDD 调优

①RDD 复用

在对 RDD 进行算子时，要避免相同计算逻辑之下对 RDD 进行重复的计算；

②RDD 持久化

通过持久化将公共 RDD 的数据缓存到内存/磁盘中,之后对于公共 RDD 的计算都会从内存/磁盘中直接获取 RDD 数据即可；

③RDD 尽可能早的 filter 操作

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

常规性能调优三： 并行度调节

- ◆ Spark 作业中的并行度指各个 stage 的 task 的数量。
- ◆ 理想的并行度设置，应该是让并行度与资源相匹配，简单来说就是在资源允许的前提下，并行度要设置的尽可能大，达到可以充分利用集群资源。合理的设置并行度，可以提升整个 Spark 作业的性能和运行速度。
- ◆ Spark 官方推荐，task 数量应该设置为 Spark 作业总 CPU core 数量的 2~3 倍。

常规性能调优四： 广播变量

当算子函数中使用到外部变量时,需要用到广播变量；广播变量会保证每个 Executor 的内存中，只驻留 1 份变量副本，而 Executor 中的 task 执行时共享该 Executor 中的那份变量副本。这样的话，可以大大减少变量副本的数量，减少网络传输性能开销，减少对 Executor 内存的占用的开销，提高 spark 的效率

常规性能调优五： Kryo 序列化

- ◆ 默认情况下，Spark 使用 Java 的序列化机制，Java 序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。
- ◆ Kryo 序列化机制比 Java 序列化机制性能提高 10 倍左右，Spark 之所以没有默认使用 Kryo 作为序列化类库，是因为它不支持所有对象的序列化，同时

Kryo 需要用户在使用前注册需要序列化的类型，不够方便，但从 Spark 2.0.0 版本开始，简单类型、简单类型数组、字符串类型的 Shuffling RDDs 已经默认使用 Kryo 序列化方式了。

2. 算子调优

算子调优一：调节 mapPartitions

- ◆ 普通的 map 算子对 RDD 中的每一个元素进行操作，而 mapPartitions 算子对 RDD 中每一个分区进行操作。但是如果使用 mapPartitions 算子，但数据量非常大时，function 一次处理一个分区的数据，如果一旦内存不足，此时无法回收内存，就可能会 OOM，即内存溢出。
- ◆ 因此，mapPartitions 算子适用于数据量不是特别大的时候，此时使用 mapPartitions 算子对性能的提升效果还是不错的。（当数据量很大的时候，一旦使用 mapPartitions 算子，就会直接 OOM）；

算子调优二：foreachPartition 优化数据库操作

如果使用 foreach 算子完成数据库的操作，由于 foreach 算子是遍历 RDD 的每条数据，因此，每条数据都会建立一个数据库连接，这是对资源的极大浪费；**在生产环境中，通常使用 foreachPartition 算子来完成对 1 个数据库的写入，通过 foreachPartition 算子，可以优化写数据库的性能。**

算子调优三：filter 与 coalesce 的配合使用

在 Spark 任务中我们经常会使用 filter 算子完成 RDD 中数据的过滤，但是一旦进过 filter 过滤后，每个分区的数据量有可能会存在较大差异；

为了解决 filter 过滤后每个分区数据量不均匀的情况，我们可以使用 coalesce 进行重分区操作。

算子调优四：repartition 解决 sparksql 低并行度问题

为了解决 Spark SQL 无法手动设置并行度和 task 数量的问题，我们可以使用 repartition 算子，以提高 Spark SQL 并行度和 task 的数量。

算子调优五：reduceByKey 本地聚合

使用 reduceByKey 对性能的提升如下：

(1) 本地聚合后，在 map 端的数据量变少，减少了磁盘 IO，也减少了对磁盘空

间的占用；

- (2) 本地聚合后，下一个 stage 拉取的数据量变少，减少了网络传输的数据量；
- (3) 本地聚合后，在 reduce 端进行数据缓存的内存占用减少；
- (4) 本地聚合后，在 reduce 端进行聚合的数据量减少。

3.shuffle 调优

Shuffle 调优一：调节 map 端缓冲区大小

- ◆ 在 Spark 任务运行过程中，如果 shuffle 的 map 端处理的数据量比较大，但是 map 端缓冲的大小是固定的，可能会出现 map 端缓冲数据频繁溢写到磁盘文件中的情况，使得性能非常低下，**通过调节 map 端缓冲的大小，可以避免频繁的磁盘 IO 操作，进而提升 Spark 任务的整体性能。**
- ◆ map 端缓冲的默认配置是 32KB，如果每个 task 处理 640KB 的数据，那么会 $640/32 = 20$ 次溢写，如果每个 task 处理 64000KB 的数据，机会发生 $64000/32=2000$ 此溢写，这对于性能的影响是非常严重的。 $32kb[按 2 倍左右调]==>64KB;$

Shuffle 调优二：调节 reduce 端拉取数据缓冲区大小

- ◆ Spark Shuffle 过程中，shuffle reduce task 的 buffer 缓冲区大小决定了 reduce task 每次能够缓冲的数据量，也就是每次能够拉取的数据量，如果内存资源较为充足，适当增加拉取数据缓冲区的大小，可以减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。
- ◆ reduce 端数据拉取缓冲区的大小可以通过 spark.reducer.maxSizeInFlight 参数进行设置，**默认为 48MB，可优化调为 96M；**

Shuffle 调优三：调节 reduce 端拉取数据重试次数

- ◆ Spark Shuffle 过程中，reduce task 拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试。对于那些包含了特别耗时的 shuffle 操作的作业，建议增加重试最大次数（比如 6 次），以避免由于网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的 shuffle 过程，调节该参数可以大幅度提升稳定性。
- ◆ reduce 端拉取数据重试次数可以通过 spark.shuffle.io.maxRetries 参数进行设置，该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败，**默认为 3 次.**

Shuffle 调优四：调节 reduce 端拉取数据等待间隔

- ◆ Spark Shuffle 过程中，reduce task 拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试，在一次失败后，会等待一定的时间间隔再进行重试，可以通过加大间隔时长（比如 60s），以增加 shuffle 操作的稳定性。
- ◆ reduce 端拉取数据等待间隔可以通过 spark.shuffle.io.retryWait 参数进行设置，默认值为 5s.

Shuffle 调优五：调节 SortShuffle 排序操作阈值

- ◆ 对于 SortShuffleManager，如果 shuffle reduce task 的数量小于某一阈值则 shuffle write 过程中不会进行排序操作，而是直接按照未经优化的 HashShuffleManager 的方式去写数据，但是最后会将每个 task 产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。
- ◆ 当你使用 SortShuffleManager 时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于 shuffle read task 的数量，那么此时 map-side 就不会进行排序了，减少了排序的性能开销，但是这种方式下，依然会产生大量的磁盘文件，因此 shuffle write 性能有待提高。

4.JVM 调优

JVM 调优一：降低 cache 操作的内存占比

①静态内存管理机制 [老版本--->现在已经不需要手动调节]

根据 Spark 静态内存管理机制，堆内存被划分为了两块，Storage 和 Execution。Storage 主要用于缓存 RDD 数据和 broadcast 数据，Execution 主要用于缓存在 shuffle 过程中产生的中间数据，Storage 占系统内存的 60%，Execution 占系统内存的 20%，并且两者完全独立。

在 Spark UI 中可以查看每个 stage 的运行情况，包括每个 task 的运行时间、gc 时间等等，如果发现 gc 太频繁，时间太长，就可以考虑调节 Storage 的内存占比，让 task 执行算子函数式，有更多的内存可以使用。

②统一内存管理机制 [动态内存管理 1.6 版本开始引进--->2016.1 月推出]

根据 Spark 统一内存管理机制，堆内存被划分为了两块，Storage 和 Execution。Storage 主要用于缓存数据 RDD 数据和 broadcast 数据，Execution 主要用于缓存在 shuffle 过程中产生的中间数据，两者所组成的内存部分称为统一内存，Storage

和 Execution 各占统一内存的 50%，由于动态占用机制的实现，shuffle 过程需要的内存过大时，会自动占用 Storage 的内存区域，因此无需手动进行调节。

JVM 调优二：调节 Executor 堆外内存

- ◆ Executor 的堆外内存主要用于程序的共享库、Perm Space、线程 Stack 和一些 Memory mapping 等，或者类 C 方式 allocate object。
- ◆ 在运行 Spark 任务的时候，也打开堆外内存，为堆内内存分担些压力；堆外但是堆外内存打开时默认只取内存的 10% 大小，对于任务来说不够用，要把堆外内存提高至少 1G 以上。

以上参数配置完成后，会避免掉某些 JVM OOM 的异常问题，同时，可以提升整体 Spark 作业的性能。

JVM 调优三：调节连接等待时长

垃圾回收会导致工作现场全部停止，也就是说，垃圾回收一旦执行，Spark 的 Executor 进程就会停止工作。此时，由于没有响应，无法建立网络连接，会导致网络连接超时。

解决：连接等待时长可以在 spark-submit 脚本中进行设置。

5.Spark 数据倾斜

前文给出了详细的解决方案

第十部分 数据湖基础

1. 你觉得数据仓库有哪些优点和缺点

优点：

1. 结构化信息存储

以结构化方式存储的信息。这意味着这里存储的数据是根据其来源划分的，并存储在各自的数据集市中。如果有数据必须添加到销售中，则该信息将进入已在数据仓库中创建的销售数据集市。数据仓库中没有杂乱的数据输入。

2. 集成数据

数据集成是数据仓库的主要功能之一。数据的集成存储意味着来自多个来源的数据一起存储在数据仓库中。

3. 非易失性

存储在数据仓库中的数据是非易失性的，因为存储在数据仓库中的信息无法编辑。一旦将信息插入到数据仓库中，就只能对其进行更新或完全删除。

4. 时变性

存储在数据仓库中的信息无法编辑，因此可以在数据仓库中长期存储。如果一个组织想要评估为什么销售额在过去一年中下降，而在两年前的销售额却是不可计数的，这可以通过参考存储在数据库中的信息来参考其当时使用的策略。

缺点：

1. 我们只能增加字段，很难去下线一个字段，真的要下线某一个字段，风险非常高
2. 数据更新的底层是把整个表的数据重新生成一遍，再覆盖掉历史数据，成本非常高
3. 增删改没有事务性保障，而在传统的数据库中有成熟的一致性解决方案
4. 索引功能是有限的，一般很少建立索引，3.0 版本已经没有索引了
5. 无法满足算法人员对数据的所有诉求，比如在机器学习场景中，需要一些文本、语音、视频等非结构化数据
6. 流批一体开发周期长，当下最流行的流批一体开发模式是基于 lambda 架构的，这种模式需要同时进行离线和实时的单独开发。

2. 数据湖是什么

数据湖是一个集中式存储库，允许存储所有结构化和非结构化数据（即便没问上一个问题，在此之前先聊一下数仓有哪些问题）

3. 聊聊数据湖和数仓之间的区别

特性	数据仓库	数据湖
数据	结构化数据，抽取自事务系统和业务应用系统	所有类型的数据，结构化、半结构化和非结构化
schema	通常在数仓实施之前设计但也可在分析时编写	在分析时编写
性价比	起步成本高，使用本地存储以获得最快查询结果	起步成本低，计算存储分离
数据质量	可作为重要事实依据的数据	包含原始数据在内的任何数据
用户	业务分析师为主	数据科学家、数据开发人员为主
分析	批处理报告、BI、可视化分析	机器学习、探索性分析、流处理、特征分析

4. 你接触过的数据湖产品有哪些，分别有什么特性

目前数据湖的相关产品主要有 **Delta Lake**、**Apache Iceberg** 和 **Apache Hudi**

下面我分别介绍一下这三个数据湖的关键特性：

Delta Lake

- 1、ACID 事务：通过不同等级的隔离策略，Delta Lake 支持多个 pipeline 的并发读写；
- 2、数据版本管理：Delta Lake 通过 Snapshot 等来管理、审计数据及元数据的版本，并进而支持 time-travel 的方式查询历史版本数据或回溯到历史版本；
- 3、开源文件格式：Delta Lake 通过 parquet 格式来存储数据，以此来实现高性能的压缩等特性；
- 4、批流一体：Delta Lake 支持数据的批量和流式读写；
- 5、元数据演化：Delta Lake 允许用户合并 schema 或重写 schema，以适应不同时期数据结构的变更；
- 6、丰富的 DML：Delta Lake 支持 Upsert，Delete 及 Merge 来适应不同场景下用户的使用需求，比如 CDC 场景；

Apache Iceberg

- 1、格式演变：支持添加，删除，更新或重命名，并且没有副作用
- 2、隐藏分区：可以防止导致错误提示或非常慢查询的用户错误
- 3、分区布局演变：可以随着数据量或查询模式的变化而更新表的布局
- 4、快照控制：可实现使用完全相同的表快照的可重复查询，或者使用户轻松检查更改
- 5、版本回滚：使用户可以通过将表重置为良好状态来快速纠正问题
- 6、快速扫描数据：无需使用分布式 SQL 引擎即可读取表或查找文件
- 7、数据修剪优化：使用表元数据使用分区和列级统计信息修剪数据文件
- 8、支持事务：序列化隔离，表更改是原子性的，读取操作永远不会看到部分更改或未提交的更改
- 9、高并发：高并发写入使用乐观并发，即使写入冲突，也会重试以确保兼容更新成功

Apache Hudi

- 1、使用快速、可插入的索引进行更新、删除
- 2、增量查询，记录级别更改流
- 3、事务、回滚、并发控制
- 4、自动调整文件大小、数据集群、压缩、清理

- 5、用于可扩展存储访问的内置元数据跟踪
- 6、式摄取、内置 CDC 源和工具
- 7、支持 Spark、Presto、Trino、Hive 等的 SQL 读/写
- 8、向后兼容的模式演变和实施

5.数据湖可以替代数仓吗

数据湖和数仓的关系，其实是一种相互补充的关系，生产落地的场景数据湖会作为上游存在，然后接入数仓，这也就是常说的湖仓一体。

6.湖仓一体了解过吗

湖仓一体是一种新的数据管理模式，将数据仓库和数据湖的价值进行叠加，让湖中的数据可以流到数据仓库中；而数据仓库中的数据也可以保存于数据湖中，供数据挖掘时可以使用。通过将数据仓库和数据湖两者之间的差异进行融合，并将数据仓库构建在数据湖上，从而有效简化了企业数据的基础架构，提升数据存储弹性和质量的同时还能降低成本，减小数据冗余。

第十一部分 项目经历

1.简历如何写项目经历

我帮助 100+同学修改过简历，发现大家有一些共性问题，比如项目描述不够清楚，项目职责写的很多（实际上都是抄的网上的），项目组织不清晰等。

今天来跟大家聊聊怎么写好这部分呢？

写项目经历主要分为两部分，包括项目描述和项目亮点。项目描述部分需要写清楚用到了哪些技术栈以及整个项目流程是什么，并且简单介绍下业务背景。项目亮点主要写项目的亮点部分以及自己解决了什么问题，因为对于应届生来说，做的很多项目面试官看过上百次了，所以一定要突出自己的亮点，自己的思考！

2.面试涉及到项目的环节有哪些

自我介绍/一面/二面/三面，这些环节都会涉及到项目，其中，自我介绍的时候简单介绍下这个项目的整体流程即可，不用介绍的太清楚，留一些空间给面试官在

面试过程中去提问。在面试过程中被问到“介绍一下自己的项目”，这个时候需要详细介绍整个流程以及自己对项目的思考或者解决了哪些问题，都可以聊一聊，也可以和面试官进行互动。

3.面试如何介绍项目经历

下面是一套模板：

面试官问：你可以介绍一下你的项目经历吗？

我回答：可以的，我主要做过两个项目，其中一个是 xxx，还有一个是 xxx。我先介绍一下 xxx 项目吧，这个项目是基于 xxx 业务背景下做的，整体的流程是这样的，首先，xxx，其次，xxx，最后，xxx。这就是整个项目的流程，在这个项目，我认为比较有亮点的地方有，一是 xxx，二是 xxx。

4.项目一之离线数仓

简单介绍一下你们的离线数仓架构

我们整体上分了五层，包括 ODS 原始数据层，DWD 数据明细层，DWS 数据服务层，DWT 数据主题层，ADS 数据应用层。然后说一下每一层有什么作用...(这些大家应该烂熟于心)

介绍一下这个项目的整体流程

该项目通过 Java 模拟生成电商业务数据和用户行为数据，业务数据存储在 MySQL 中，用户行为数据以日志文件的格式存在；然后通过 Sqoop 将 MySQL 中的数据导入到 HDFS 中，同时通过 Flume 进行日志文件的采集，利用 Kafka 作为缓冲队列，再经过 Flume 采集到 HDFS 中；然后利用 Hive 进行数据仓库的搭建；最后按照主题将 Hive 中的数据导入到 MySQL 中，并且利用 Superset 进行可视化分析。

你有没有遇到过什么问题

数据倾斜问题？

小文件问题？

集群问题？

这些都可以聊，反正编一个完整的故事就行

你认为你在这个项目中做的比较好的地方有哪些

比如在进行用户行为数据采集的时候，利用 Kafka 作为缓冲队列，防止因 Flume 采集的速度过快，导致 HDFS 来不及接收。

项目中用到的框架的八股文要熟悉，问项目的同时也会问到

5.项目二之实时数仓

简单介绍一下你们的实时数仓架构

我们总共分为了 5 层：ODS 原始数据层，DWD 数据明细层，DIM 维度数据层，DWM 数据中间层，DWS 数据服务层。然后说一下每一层有什么作用...(这些大家应该烂熟于心)

介绍一下这个项目的整体流程

该项目通过 Java 模拟实时生成电商业务数据和用户行为数据，用户行为数据写入到 Kafka 的主题中，电商业务数据写入 MySQL 中，然后通过 Maxwell 实时读取 MySQL 二进制日志 binlog 文件并且以 JSON 格式发送给 Kafka；然后通过编写 Flink 程序来对用户行为数据和业务数据进行分流，以及按照主题对数据进行 join 和聚合等操作形成主题宽表，最终写入 ClickHouse 数据库中，并且进行可视化。

你有没有遇到过什么问题

数据倾斜？

乱序问题？

你认为你在这个项目中做的比较好的地方有哪些

比如动态分流，在 DWD 层，需要对业务数据进行动态分流处理，将维度数据写入 HBase 中，同时将事实数据写回到 Kafka 主题中。

比如 DWM 层搭建，在 DWM 层，需要对事实数据和维度数据进行关联，对于事实数据之间的关联，采用了 interval join；对于事实和维度的关联，涉及到了外部数据源 HBase 的查询，所以做了两个优化处理，分别是旁路缓存和异步查询。旁路缓存是 xxx，异步查询是 xxx（做过这个项目的应该都知道）。