大数据技术之高频面试题

目录

[第1章 项目涉及技术 6](#_Toc52287962)

[1.1 Linux&Shell相关总结 6](#_Toc52287963)

[1.1.1 Linux常用命令 6](#_Toc52287964)

[1.1.2 Shell常用工具 6](#_Toc52287965)

[1.2 Hadoop相关总结 7](#_Toc52287966)

[1.2.1 Hadoop常用端口号 7](#_Toc52287967)

[1.2.2 Hadoop配置文件以及简单的Hadoop集群搭建 7](#_Toc52287968)

[1.2.3 HDFS读流程和写流程 8](#_Toc52287969)

[1.2.4 MapReduce的Shuffle过程及Hadoop优化（包括：压缩、小文件、集群优化） 9](#_Toc52287970)

[1.2.5 Yarn的Job提交流程 12](#_Toc52287971)

[1.2.6 Yarn的默认调度器、调度器分类、以及他们之间的区别 12](#_Toc52287972)

[1.2.7 项目经验之LZO压缩 13](#_Toc52287973)

[1.2.8 Hadoop参数调优 14](#_Toc52287974)

[1.2.9 项目经验之基准测试 14](#_Toc52287975)

[1.2.10 Hadoop宕机 14](#_Toc52287976)

[1.2.11 Hadoop解决数据倾斜方法 14](#_Toc52287977)

[1.2.12 集群资源分配参数（项目中遇到的问题） 15](#_Toc52287978)

[1.3 Zookeeper相关总结 15](#_Toc52287979)

[1.3.1 选举机制 15](#_Toc52287980)

[1.3.2 常用命令 15](#_Toc52287981)

[1.4 Flume相关总结 16](#_Toc52287982)

[1.4.1 Flume组成，Put事务，Take事务 16](#_Toc52287983)

[1.4.2 Flume拦截器 16](#_Toc52287984)

[1.4.3 Flume Channel选择器 17](#_Toc52287985)

[1.4.4 Flume监控器 17](#_Toc52287986)

[1.4.5 Flume采集数据会丢失吗?（防止数据丢失的机制） 17](#_Toc52287987)

[1.4.6 Flume内存 17](#_Toc52287988)

[1.4.7 FileChannel优化 17](#_Toc52287989)

[1.4.8 Flume Channel容量 18](#_Toc52287990)

[1.4.9 HDFS Sink小文件处理 18](#_Toc52287991)

[1.5 Kafka相关总结 19](#_Toc52287992)

[1.5.1 Kafka架构 19](#_Toc52287993)

[1.5.2 Kafka压测 19](#_Toc52287994)

[1.5.3 Kafka的机器数量 19](#_Toc52287995)

[1.5.4 Kafka的日志保存时间 19](#_Toc52287996)

[1.5.5 Kafka的硬盘大小 19](#_Toc52287997)

[1.5.6 Kafka监控 19](#_Toc52287998)

[1.5.7 Kakfa分区数 20](#_Toc52287999)

[1.5.8 副本数设定 20](#_Toc52288000)

[1.5.9 多少个Topic 20](#_Toc52288001)

[1.5.10 Kafka丢不丢数据 20](#_Toc52288002)

[1.5.11 Kafka的ISR副本同步队列 20](#_Toc52288003)

[1.5.12 Kafka分区分配策略 20](#_Toc52288004)

[1.5.13 Kafka中数据量计算 21](#_Toc52288005)

[1.5.14 Kafka挂掉 21](#_Toc52288006)

[1.5.15 Kafka消息数据积压，Kafka消费能力不足怎么处理？ 21](#_Toc52288007)

[1.5.16 Kafka幂等性 21](#_Toc52288008)

[1.5.17 Kafka事务 22](#_Toc52288009)

[1.5.18 Kafka数据重复 22](#_Toc52288010)

[1.5.19 Kafka参数优化 22](#_Toc52288011)

[1.5.20 Kafka 高效读写数据 23](#_Toc52288012)

[1.5.21 Kafka支持传输 24](#_Toc52288013)

[1.5.22 Kafka过期数据清理 24](#_Toc52288014)

[1.5.23 Kafka可以按照时间消费数据 24](#_Toc52288015)

[1.5.24 Zookeeper中存储kafka哪些信息 25](#_Toc52288016)

[1.6 Hive总结 25](#_Toc52288017)

[1.6.1 Hive的架构 25](#_Toc52288018)

[1.6.2 Hive和数据库比较 25](#_Toc52288019)

[1.6.3 内部表和外部表 26](#_Toc52288020)

[1.6.4 4个By区别 26](#_Toc52288021)

[1.6.5 窗口函数 26](#_Toc52288022)

[1.6.6 自定义UDF、UDTF 27](#_Toc52288023)

[1.6.7 Hive优化 27](#_Toc52288024)

[1.6.8 Hive解决数据倾斜方法 29](#_Toc52288025)

[1.6.9 用的是动态分区吗？动态分区的底层原理是什么？ 32](#_Toc52288026)

[26. Hive里边字段的分隔符用的什么？为什么用\t？有遇到过字段里边有\t的情况吗，怎么处理的？ 32](#_Toc52288027)

[1.7 HBase总结 32](#_Toc52288028)

[1.7.1 HBase存储结构 32](#_Toc52288029)

[1.7.2 RowKey设计原则 32](#_Toc52288030)

[1.7.3 RowKey如何设计 32](#_Toc52288031)

[1.7.4 Phoenix二级索引（讲原理） 33](#_Toc52288032)

[1.8 Sqoop参数 33](#_Toc52288033)

[1.8.1 Sqoop导入导出Null存储一致性问题 33](#_Toc52288034)

[1.8.2 Sqoop数据导出一致性问题 33](#_Toc52288035)

[1.8.3 Sqoop底层运行的任务是什么 34](#_Toc52288036)

[1.8.4 Sqoop数据导出的时候一次执行多长时间 34](#_Toc52288037)

[1.8.5 Sqoop一天导多少数据 34](#_Toc52288038)

[1.8.6 Sqoop在导入数据的时候数据倾斜 34](#_Toc52288039)

[1.13 数据倾斜 34](#_Toc52288040)

[1.13.1 数据倾斜表现 34](#_Toc52288041)

[1.13.2 数据倾斜产生原因 35](#_Toc52288042)

[1.13.3 解决数据倾斜思路 36](#_Toc52288043)

[第2章 项目架构 37](#_Toc52288044)

[2.1 数仓概念 38](#_Toc52288045)

[2.2 系统数据流程设计 38](#_Toc52288046)

[2.3 框架版本选型 38](#_Toc52288047)

[2.4 服务器选型 39](#_Toc52288048)

[2.5 集群规模 40](#_Toc52288049)

[2.6 人员配置参考 40](#_Toc52288050)

[2.6.1 整体架构 40](#_Toc52288051)

[2.6.2 你们部门的职级等级，晋升规则 41](#_Toc52288052)

[2.6.3 人员配置参考 41](#_Toc52288053)

[第3章 用户行为数据分析 42](#_Toc52288054)

[3.1 数仓分层架构表 42](#_Toc52288055)

[3.2 埋点行为数据基本格式(基本字段) 42](#_Toc52288056)

[3.3 项目经验总结 44](#_Toc52288057)

[3.3.1 项目经验之元数据备份 44](#_Toc52288058)

[3.3.2 日期处理函数 44](#_Toc52288059)

[3.3.3 Union与Union all区别 44](#_Toc52288060)

[3.3.4 Shell中单引号和双引号区别 44](#_Toc52288061)

[3.3.5 Tez引擎优点？ 45](#_Toc52288062)

[3.4 需求逻辑（重点） 45](#_Toc52288063)

[3.4.1 如何分析用户活跃？ 45](#_Toc52288064)

[3.4.2 如何分析用户新增？vivo 45](#_Toc52288065)

[3.4.3 如何分析用户1天留存？ 45](#_Toc52288066)

[3.4.4 如何分析沉默用户？ 45](#_Toc52288067)

[3.4.5 如何分析本周回流用户？ 46](#_Toc52288068)

[3.4.6 如何分析流失用户？ 46](#_Toc52288069)

[3.4.7 如何分析最近连续3周活跃用户数？ 46](#_Toc52288070)

[3.4.8 如何分析最近七天内连续三天活跃用户数？ 46](#_Toc52288071)

[第8章 手写代码 46](#_Toc52288072)

[8.1 基本算法 46](#_Toc52288073)

[8.1.1 冒泡排序 46](#_Toc52288074)

[8.1.2 二分查找 47](#_Toc52288075)

[8.1.3 快排 49](#_Toc52288076)

[8.1.4 归并 50](#_Toc52288077)

[8.1.5 二叉树之Scala实现 51](#_Toc52288078)

[8.2 手写HQL 56](#_Toc52288079)

[8.2.1 手写HQL 第1题 56](#_Toc52288080)

[8.2.2 手写HQL 第2题 57](#_Toc52288081)

[8.2.3 手写HQL 第3题 59](#_Toc52288082)

[8.2.4 手写HQL 第4题 60](#_Toc52288083)

[8.2.5 手写HQL 第5题 61](#_Toc52288084)

[8.2.6 手写HQL 第6题 65](#_Toc52288085)

[8.2.7 手写HQL 第7题 66](#_Toc52288086)

[8.2.8 手写SQL 第8题 67](#_Toc52288087)

[8.2.9 手写HQL 第9题 68](#_Toc52288088)

[8.2.10 手写HQL 第10题 69](#_Toc52288089)

[第9章 JavaSE 73](#_Toc52288090)

[9.1 HashMap底层源码，数据结构 73](#_Toc52288091)

[9.2 Java自带哪几种线程池？ 76](#_Toc52288092)

[9.3 HashMap和HashTable区别 77](#_Toc52288093)

[9.4 TreeSet和HashSet区别 78](#_Toc52288094)

[9.5 String buffer和String build区别 78](#_Toc52288095)

[9.6 Final、Finally、Finalize 78](#_Toc52288096)

[9.7 ==和Equals区别 79](#_Toc52288097)

[第10章 Redis 79](#_Toc52288098)

[10.1 缓存穿透、缓存雪崩、缓存击穿 79](#_Toc52288099)

[10.2 哨兵模式 80](#_Toc52288100)

[10.3 数据类型 80](#_Toc52288101)

[10.4 持久化 80](#_Toc52288102)

[11.5 悲观锁 81](#_Toc52288103)

[11.6 乐观锁 81](#_Toc52288104)

[第11章 MySql 81](#_Toc52288105)

[11.1 MyISAM与InnoDB的区别 81](#_Toc52288106)

[11.2 索引优化 81](#_Toc52288107)

[11.3 b-tree和b+tree的区别 82](#_Toc52288108)

[11.4 redis是单线程的，为什么那么快 82](#_Toc52288109)

[11.5 MySQL的事务 82](#_Toc52288110)

[第12章 JVM 84](#_Toc52288111)

[12.1 JVM内存分哪几个区，每个区的作用是什么? 84](#_Toc52288112)

[12.2 Java类加载过程? 85](#_Toc52288113)

[12.3 java中垃圾收集的方法有哪些? 86](#_Toc52288114)

[12.4 如何判断一个对象是否存活?(或者GC对象的判定方法) 86](#_Toc52288115)

[12.5 什么是类加载器，类加载器有哪些? 87](#_Toc52288116)

[12.6 简述Java内存分配与回收策略以及Minor GC和Major GC（full GC） 87](#_Toc52288117)

[第13章 JUC 88](#_Toc52288118)

**[13.1 Synchronized与Lock的区别](#_Toc52288119)** [88](#_Toc52288119)

**[13.2 Runnable和Callable的区别](#_Toc52288120)** [88](#_Toc52288120)

**[13.3 什么是分布式锁](#_Toc52288121)** [88](#_Toc52288121)

**[13.4 什么是分布式事务](#_Toc52288122)** [88](#_Toc52288122)

[第14章 面试说明 88](#_Toc52288123)

[14.1 面试过程最关键的是什么？ 88](#_Toc52288124)

[14.2 面试时该怎么说？ 88](#_Toc52288125)

[14.3 面试技巧 89](#_Toc52288126)

[14.3.1 六个常见问题 89](#_Toc52288127)

[14.3.2 两个注意事项 90](#_Toc52288128)

[14.3.3 自我介绍（控制在4分半以内，不超过5分钟） 90](#_Toc52288129)

[第15章 LeetCode题目精选 90](#_Toc52288130)

[15.1 两数之和 90](#_Toc52288131)

[15.1.1 问题描述 90](#_Toc52288132)

[15.1.2 参考答案 90](#_Toc52288133)

[15.2 爬楼梯 91](#_Toc52288134)

[15.2.1 问题描述 91](#_Toc52288135)

[15.2.2 参考答案 92](#_Toc52288136)

[15.3 翻转二叉树 92](#_Toc52288137)

[15.3.1 问题描述 92](#_Toc52288138)

[15.3.2 参考答案 93](#_Toc52288139)

[15.4 反转链表 94](#_Toc52288140)

[15.4.1 问题描述 94](#_Toc52288141)

[15.4.2 参考答案 94](#_Toc52288142)

[15.5 LRU缓存机制 94](#_Toc52288143)

[15.5.1 问题描述 94](#_Toc52288144)

[15.5.2 参考答案 95](#_Toc52288145)

[15.6 最长回文子串 96](#_Toc52288146)

[15.6.1 问题描述 96](#_Toc52288147)

[15.6.2 参考答案 97](#_Toc52288148)

[15.7 有效的括号 98](#_Toc52288149)

[15.7.1 问题描述 98](#_Toc52288150)

[15.7.2 参考答案 99](#_Toc52288151)

[15.8 数组中的第K个最大元素 101](#_Toc52288152)

[15.8.1 问题描述 101](#_Toc52288153)

[15.8.2 参考答案 101](#_Toc52288154)

[15.9 实现 Trie (前缀树) 104](#_Toc52288155)

[15.9.1 问题描述 104](#_Toc52288156)

[15.9.2 参考答案 104](#_Toc52288157)

[15.10 编辑距离 106](#_Toc52288158)

[15.10.1 问题描述 106](#_Toc52288159)

[15.10.2 参考答案 107](#_Toc52288160)

# 第1章 项目涉及技术

## 1.1 Linux&Shell相关总结

### 1.1.1 Linux常用命令

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

### 1.1.2 Shell常用工具

awk、sed、cut、sort

## 1.2 Hadoop相关总结

### 1.2.1 Hadoop常用端口号

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

### 1.2.2 Hadoop配置文件以及简单的Hadoop集群搭建

（1）配置文件：

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

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

（2）简单的集群搭建过程：

JDK安装

配置SSH免密登录

配置hadoop核心文件:

格式化namenode

### 1.2.3 HDFS读流程和写流程





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





**一、Shuffle机制（分区，排序，溢写）**

1）Map方法之后Reduce方法之前这段处理过程叫Shuffle

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

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

**二、Hadoop优化**

0）HDFS小文件影响

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

（2）影响计算引擎的任务数量，比如每个小的文件都会生成一个Map任务

1）数据输入小文件处理：

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

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

（3）对于大量小文件Job，可以开启JVM重用。

2）Map阶段

（1）增大环形缓冲区大小。由100m扩大到200m

（2）增大环形缓冲区溢写的比例。由80%扩大到90%

（3）减少对溢写文件的merge次数。（10个文件，一次20个merge）

（4）不影响实际业务的前提下，采用Combiner提前合并，减少 I/O。

3）Reduce阶段

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

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

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

（4）增加每个Reduce去Map中拿数据的并行数

（5）集群性能可以的前提下，增大Reduce端存储数据内存的大小。

4）IO传输

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

（2）使用SequenceFile二进制文件

5）整体

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

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

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

（4）增加每个Container的CPU核数和内存大小

（5）调整每个Map Task和Reduce Task最大重试次数

**三、压缩**

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

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

**四、切片机制**

1）简单地按照文件的内容长度进行切片

2）切片大小，默认等于Block大小

3）切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

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

### 1.2.5 Yarn的Job提交流程



*评分标准：5分*

### 1.2.6 Yarn的默认调度器、调度器分类、以及他们之间的区别

1）Hadoop调度器重要分为三类：

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

Hadoop2.7.2默认的资源调度器是 容量调度器

2）区别：

FIFO调度器：先进先出，同一时间队列中只有一个任务在执行。



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



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

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



### 1.2.7 项目经验之LZO压缩

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

### 1.2.8 Hadoop参数调优

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

2）NameNode有一个工作线程池，用来处理不同DataNode的并发心跳以及客户端并发的元数据操作。

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

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

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

### 1.2.9 项目经验之基准测试

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

### 1.2.10 Hadoop宕机

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

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

### 1.2.11 Hadoop解决数据倾斜方法

**1）提前在map进行combine，减少传输的数据量**

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

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

**2）导致数据倾斜的key 大量分布在不同的mapper**

（1）局部聚合加全局聚合。

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

第二次mapreduce，去掉key的随机前缀，进行全局聚合。

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

这个方法进行两次mapreduce，性能稍差。

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

（3）实现自定义分区

根据数据分布情况，自定义散列函数，将key均匀分配到不同Reducer

### 1.2.12 集群资源分配参数（项目中遇到的问题）

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

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

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

## 1.3 Zookeeper相关总结

### 1.3.1 选举机制

半数机制：2n+1

10台服务器：3台

20台服务器：5台

100台服务器：11台

台数并不是越多越好。 太多选举时间过长影响性能。

### 1.3.2 常用命令

ls、get、create

## 1.4 Flume相关总结

### 1.4.1 Flume组成，Put事务，Take事务

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

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

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

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

Source到Channel是Put事务

Channel到Sink是Take事务®

### 1.4.2 Flume拦截器

（1）拦截器注意事项

项目中自定义了：ETL拦截器和区分类型拦截器。

采用两个拦截器的优缺点：优点，模块化开发和可移植性；缺点，性能会低一些

（2）自定义拦截器步骤

a）实现 Interceptor

b）重写四个方法

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

c）静态内部类，实现Interceptor.Builder

### 1.4.3 Flume Channel选择器



### 1.4.4 Flume监控器

Ganglia

### 1.4.5 Flume采集数据会丢失吗?（防止数据丢失的机制）

不会，Channel存储可以存储在File中，数据传输自身有事务。

### 1.4.6 Flume内存

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

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

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

### 1.4.7 FileChannel优化

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

官方说明如下：

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

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

### 1.4.8 Flume Channel容量

File channel 容量1000000条

Memory channel 容量 100条

### 1.4.9 HDFS Sink小文件处理

（1）HDFS存入大量小文件，有什么影响？

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

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

（2）HDFS小文件处理

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

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

（1）tmp文件在达到128M时会滚动生成正式文件

（2）tmp文件创建超3600秒时会滚动生成正式文件

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

/root/20180101/root.201801010520.tmp

即使文件内容没有达到128M，也会在06:23时滚动生成正式文件

## 1.5 Kafka相关总结

### 1.5.1 Kafka架构



### 1.5.2 Kafka压测

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

### 1.5.3 Kafka的机器数量

Kafka机器数量=2\*（峰值生产速度\*副本数/100）+1 （峰值生产速度\*副本数量\*110%\*70%）

### 1.5.4 Kafka的日志保存时间

3天

### 1.5.5 Kafka的硬盘大小

每天的数据量\*3天/70%

### 1.5.6 Kafka监控

公司自己开发的监控器；

开源的监控器：KafkaManager、KafkaMonitor、KafkaEagle

### 1.5.7 Kakfa分区数

1）创建一个只有1个分区的topic

2）测试这个topic的producer吞吐量和consumer吞吐量。

3）假设他们的值分别是Tp和Tc，单位可以是MB/s。

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

例如：producer吞吐量=20m/s；consumer吞吐量=50m/s，期望吞吐量100m/s；

分区数=100 / 20 =5分区

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

分区数一般设置为：3-10个

### 1.5.8 副本数设定

一般我们设置成2个或3个，很多企业设置为2个。

### 1.5.9 多少个Topic

通常情况：多少个日志类型就多少个Topic。也有对日志类型进行合并的。

### 1.5.10 Kafka丢不丢数据

Ack=0，相当于异步发送，消息发送完毕即offset增加，继续生产。

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

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

### 1.5.11 Kafka的ISR副本同步队列

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

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

### 1.5.12 Kafka分区分配策略

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

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

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

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6 分区

C2-1 将消费 7, 8, 9 分区

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

### 1.5.13 Kafka中数据量计算

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

平均每秒钟：1150条

低谷每秒钟：50条

高峰每秒钟：1150条\*（2-20倍）=2300条-23000条

每条日志大小：0.5k-2k（取1k）

每秒多少数据量：2.0M-20MB

### 1.5.14 Kafka挂掉

1）Flume记录

2）日志有记录

3）短期没事

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

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

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

### 1.5.16 Kafka幂等性

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

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

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

### 1.5.17 Kafka事务

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

1）Producer事务

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

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

2）Consumer事务

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

### 1.5.18 Kafka数据重复

幂等性+ack-1+事务

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

### 1.5.19 Kafka参数优化

**1）Broker参数配置（server.properties）**

1、网络和io操作线程配置优化

# broker处理消息的最大线程数（默认为3）

num.network.threads=cpu核数+1

# broker处理磁盘IO的线程数

num.io.threads=cpu核数\*2

2、log数据文件刷盘策略

# 每当producer写入10000条消息时，刷数据到磁盘

log.flush.interval.messages=10000

# 每间隔1秒钟时间，刷数据到磁盘

log.flush.interval.ms=1000

3、日志保留策略配置

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

log.retention.hours=72

4、Replica相关配置

offsets.topic.replication.factor:3

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

**2）Producer优化（producer.properties）**

buffer.memory:33554432 (32m)

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

compression.type:none

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

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

默认内存1个G，生产环境尽量不要超过6个G。

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

### 1.5.20 Kafka 高效读写数据

**1）Kafka本身是分布式集群，同时采用分区技术，并发度高。**

**2）顺序写磁盘**

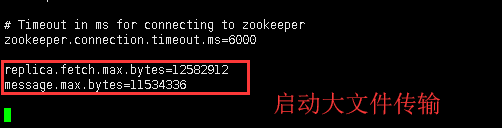
Kafka的producer生产数据，要写入到log文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到600M/s，而随机写只有100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

**3）零复制技术**



### 1.5.21 Kafka支持传输

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



### 1.5.22 Kafka过期数据清理

保证数据没有被引用（没人消费他）

日志清理保存的策略只有delete和compact两种

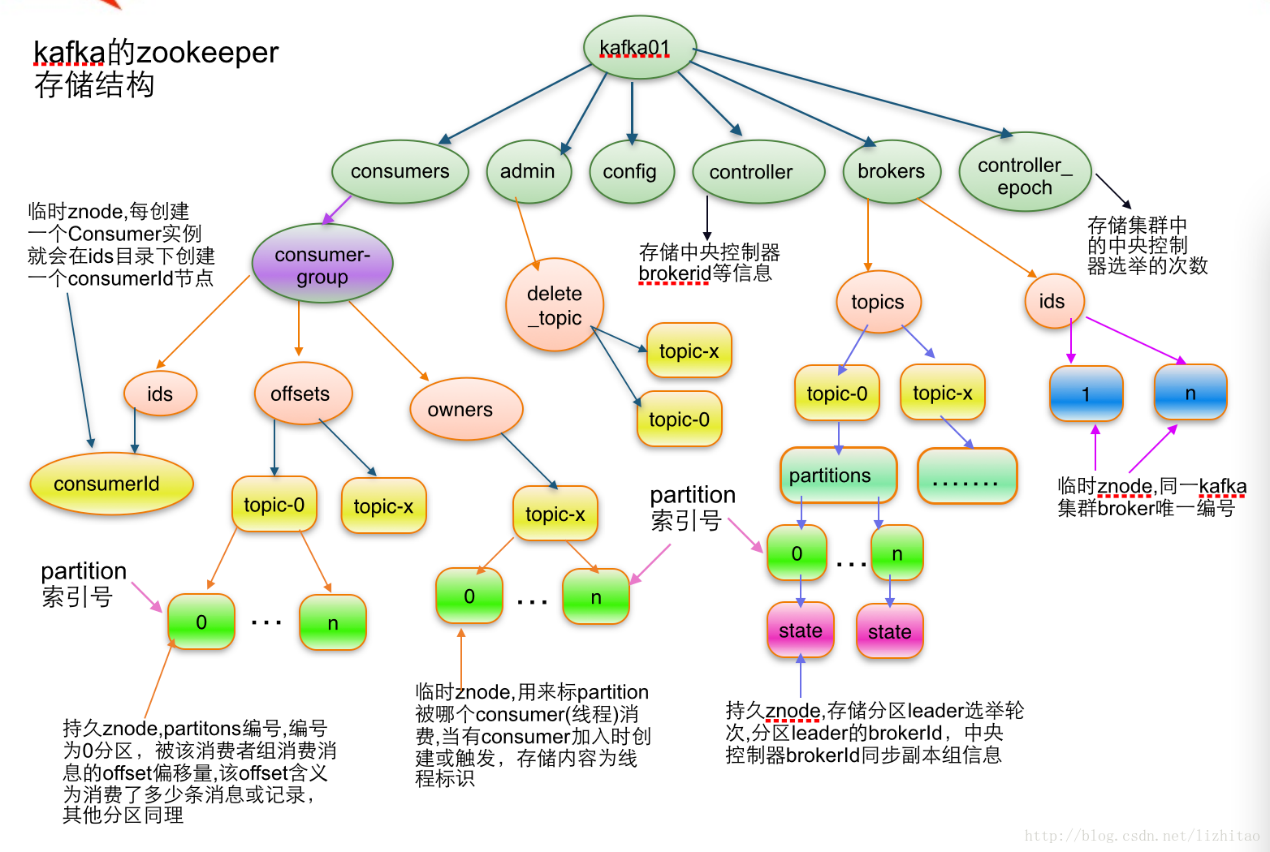
log.cleanup.policy=delete启用删除策略

log.cleanup.policy=compact启用压缩策略

### 1.5.23 Kafka可以按照时间消费数据

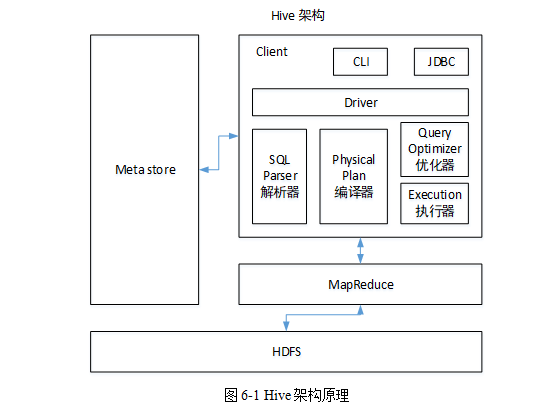
Map<TopicPartition, OffsetAndTimestamp> startOffsetMap = KafkaUtil.fetchOffsetsWithTimestamp(topic, sTime, kafkaProp);

### 1.5.24 Zookeeper中存储kafka哪些信息



## 1.6 Hive总结

### 1.6.1 Hive的架构



### 1.6.2 Hive和数据库比较

Hive 和数据库除了拥有类似的查询语言，再无类似之处。

1）数据存储位置

Hive 存储在 HDFS 。数据库将数据保存在块设备或者本地文件系统中。

2）数据更新

Hive中不建议对数据的改写。而数据库中的数据通常是需要经常进行修改的，

3）执行延迟

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

4）数据规模

Hive支持很大规模的数据计算；数据库可以支持的数据规模较小。

### 1.6.3 内部表和外部表

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

2）外部表：删除该表并不会删除掉原始数据，删除的是表的元数据

### 1.6.4 4个By区别

1）Sort By：分区内有序；

2）Order By：全局排序，只有一个Reducer；

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

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

### 1.6.5 窗口函数

RANK() 排序相同时会重复，总数不会变

DENSE\_RANK() 排序相同时会重复，总数会减少

ROW\_NUMBER() 会根据顺序计算

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

2）CURRENT ROW：当前行

3）n PRECEDING：往前n行数据

4） n FOLLOWING：往后n行数据

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

6） LAG(col,n)：往前第n行数据

7）LEAD(col,n)：往后第n行数据

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

### 1.6.6 自定义UDF、UDTF

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

1）自定义过。

2）用UDF函数解析公共字段；用UDTF函数解析事件字段。

自定义UDF：继承UDF，重写evaluate方法

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

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

### 1.6.7 Hive优化

**1）MapJoin**

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

**2）行列过滤**

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

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

**3）列式存储**

**4）采用分区技术**

**5）合理设置Map数**

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

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

**（2）是不是map数越多越好？**

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

**（3）是不是保证每个map处理接近128m的文件块，就高枕无忧了？**

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

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

**6）小文件进行合并**

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

**7）合理设置Reduce数**

Reduce个数并不是越多越好

（1）过多的启动和初始化Reduce也会消耗时间和资源；

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

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

**8）常用参数**

// 输出合并小文件

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

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

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

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

**9）开启map端combiner（不影响最终业务逻辑）**

set hive.map.aggr=true；

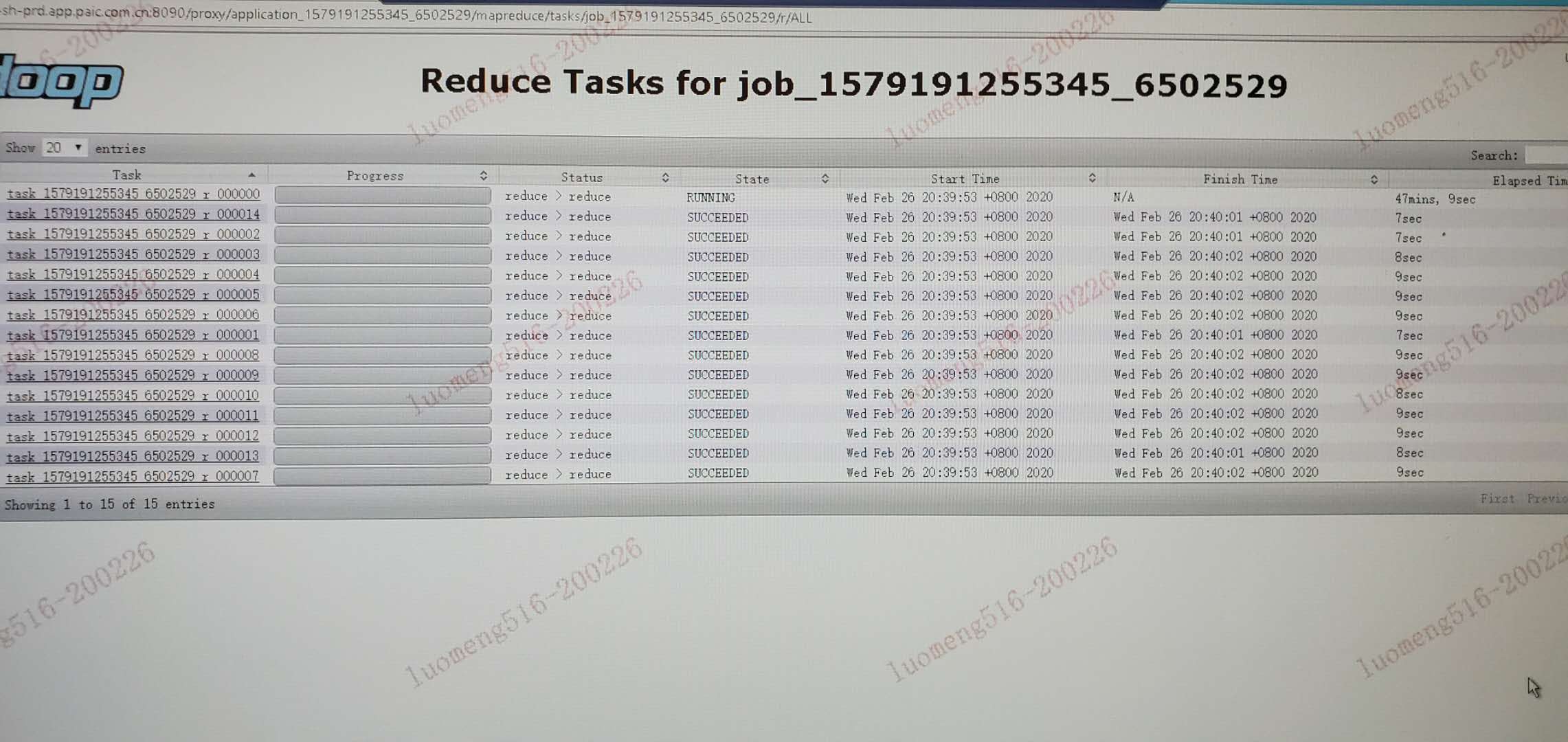
**10）压缩（选择快的）**

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

**11）开启JVM重用**

### 1.6.8 Hive解决数据倾斜方法

**1）数据倾斜长啥样？**



**2）怎么产生的数据倾斜？**

**不同数据类型关联产生数据倾斜**

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

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

解决方式：把数字类型转换成字符串类型

select \* from users a

left outer join logs b

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

**3）解决数据倾斜的方法？**

**（1）group by**

注：group by 优于distinct group

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

**（2）mapjoin**

**（3）开启数据倾斜时负载均衡**

set hive.groupby.skewindata=true;

思想：就是先随机分发并处理，再按照key group by来分发处理。

操作：当选项设定为true，生成的查询计划会有两个MRJob。

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

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

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

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

**（4）控制空值分布**

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

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

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

使用case when实例1：

select userid, name from user\_info a

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

使用case when实例2：

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

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

### 1.6.9 用的是动态分区吗？动态分区的底层原理是什么？

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

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

c. 动态分区是基于查询参数的位置去推断分区的名称，从而建立分区

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

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

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

## 1.7 HBase总结

### 1.7.1 HBase存储结构



### 1.7.2 RowKey设计原则

1）rowkey长度原则

2）rowkey散列原则

3）rowkey唯一原则

### 1.7.3 RowKey如何设计

1）生成随机数、hash、散列值

2）字符串反转

### 1.7.4 HBase二级索引（详见Phoenix）

## 1.8 Sqoop参数

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

--connect \

--username \

--password \

--target-dir \

--delete-target-dir \

--num-mappers \

--fields-terminated-by   \

--query   "$2" ' and $CONDITIONS;'

### 1.8.1 Sqoop导入导出Null存储一致性问题

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

### 1.8.2 Sqoop数据导出一致性问题

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

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

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

–staging-table方式

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

2）场景2：设置map数量为1个（不推荐，面试官想要的答案不只这个）

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

### 1.8.3 Sqoop底层运行的任务是什么

只有Map阶段，没有Reduce阶段的任务。

### 1.8.4 Sqoop数据导出的时候一次执行多长时间

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

### 1.8.5 Sqoop一天导多少数据

业务数据每天1G，就导1G。

### 1.8.6 Sqoop在导入数据的时候数据倾斜

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

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

通过ROWNUM() 生成一个严格均匀分布的字段，然后指定为分割字段

## 1.13 数据倾斜

公司一：总用户量1000万，5台64G内存的服务器。

公司二：总用户量10亿，1000台64G内存的服务器。

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

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

### 1.13.1 数据倾斜表现

1）hadoop中的数据倾斜表现：

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

2）hive中数据倾斜

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

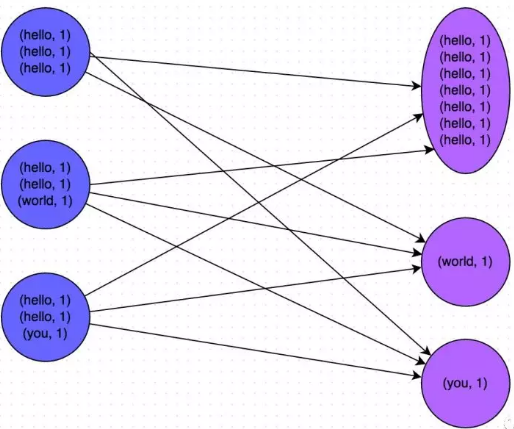
### 1.13.2 数据倾斜产生原因

以Hive的使用场景为例。

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

一般来说，数据倾斜原因有以下几方面：

**1）key分布不均匀；**



**2）建表时考虑不周**

我们举一个例子，就说数据默认值的设计吧，假设我们有两张表：

user（用户信息表）：userid，register\_ip

ip（IP表）：ip，register\_user\_cnt

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

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

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

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

**3）业务数据激增**

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

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

### 1.13.3 解决数据倾斜思路

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

**1）业务逻辑**

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

**2）程序层面**

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

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

（1）优化前 只有一个reduce，先去重再count负担比较大：

select name,count(distinct name)from user;

（2）优化后

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

set mapred.reduce.tasks=3;

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

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

**3）调参方面**

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

**4）从业务和数据上解决数据倾斜**

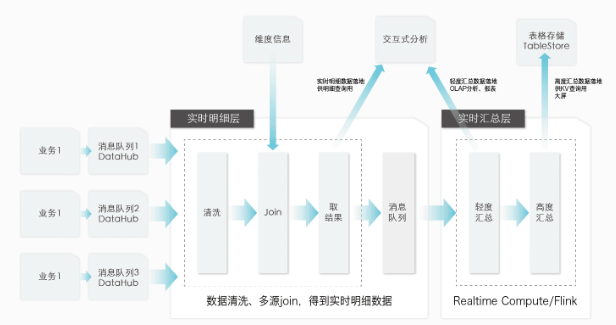
很多数据倾斜都是在数据的使用上造成的。我们举几个场景，并分别给出它们的解决方案。

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

# 第2章 项目架构

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





## 2.1 数仓概念

**数据仓库的输入数据源和输出系统分别是什么？**

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

输出系统：报表系统、用户画像系统、推荐系统

## 2.2 系统数据流程设计



## 2.3 框架版本选型

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

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

3）HDP：开源，可以进行二次开发，但是没有CDH稳定，国内使用较少



## 2.4 服务器选型

服务器使用物理机还是云主机？

1）机器成本考虑：

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

（2）云主机，以阿里云为例，差不多相同配置，每年5W

2）运维成本考虑：

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

（2）云主机：很多运维工作都由阿里云已经完成，运维相对较轻松

3）企业选择

（1）金融有钱公司和阿里没有直接冲突的公司选择阿里云（上海）

（2）中小公司、为了融资上市，选择阿里云，拉倒融资后买物理机。

（3）有长期打算，资金比较足，选择物理机。

## 2.5 集群规模



根据数据规模大家集群

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

## 2.6 人员配置参考

### 2.6.1 整体架构

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

大数据开发工程师=>大数据组组长=》项目经理=>部门经理=》技术总监

### 2.6.2 你们部门的职级等级，晋升规则

职级就分初级，中级，高级。晋升规则不一定，看公司效益和职位空缺。

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

阿里：p5、p6、p7、p8

### 2.6.3 人员配置参考

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

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

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

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

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

IOS多少人 安卓多少人 前端多少人 JavaEE多少人 测试多少人

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

公司划分：

0-50 小公司

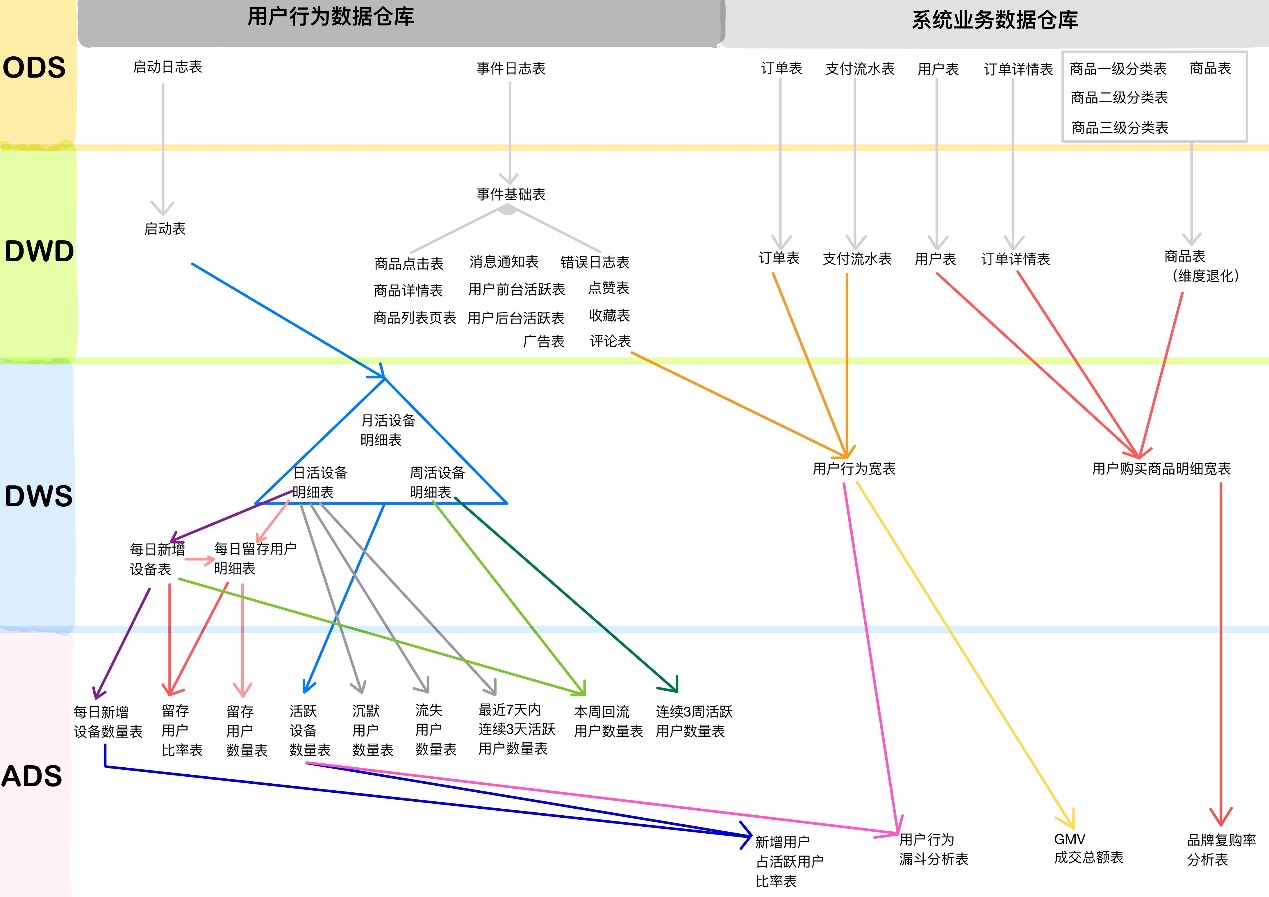
50-500 中等

500-1000 大公司

1000以上 大厂 领军的存在

# 第3章 用户行为数据分析

## 3.1 数仓分层架构表



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

ods 原始数据层 存放原始数据，保持原貌不做处理

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

dws 服务数据层 轻度聚合

ads 应用数据层 具体需求

数仓中各层建的表都是外部表

## 3.2 埋点行为数据基本格式(基本字段)

公共字段：基本所有安卓手机都包含的字段

业务字段：埋点上报的字段，有具体的业务类型

下面就是一个示例，表示业务字段的上传。

行为数据启动日志/事件日志表关键字段:

{

"ap":"xxxxx",//项目数据来源 app pc

"cm": { //公共字段

**"mid": "", // (String) 设备唯一标识**

**"uid": "", // (String) 用户标识**

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

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

"l": "zh", // (String) 系统语言

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

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

"ar": "CN", // (String) 区域

"md": "BBB100-1", // (String) 手机型号

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

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

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

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

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

**"nw": "WIFI", // (String) 网络模式**

"ln": 0, // (double) lng经度

"la": 0 // (double) lat 纬度

},

"et": [ //事件

{

"ett": "1506047605364", //客户端事件产生时间

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

"kv": { //事件结果，以key-value形式自行定义

"goodsid": "236",

"action": "1",

"extend1": "1",

"place": "2",

"category": "75"

}

}

]

}

根据事件标签的不同可以分成不同的日志表

## 3.3 项目经验总结

### 3.3.1 项目经验之元数据备份

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

### 3.3.2 日期处理函数

1）date\_add、date\_sub函数（加减日期）

2）next\_day函数

3）date\_format函数（根据格式整理日期）

4）last\_day函数（求当月最后一天日期）

5）collect\_set函数

6）get\_json\_object解析json函数

### 3.3.3 Union与Union all区别

1）union会将联合的结果集去重，效率较union all差

2）union all不会对结果集去重，所以效率高

### 3.3.4 Shell中单引号和双引号区别

1）在/home/root/bin创建一个test.sh文件

[root@hadoop102 bin]$ vim test.sh

在文件中添加如下内容

#!/bin/bash

do\_date=$1

echo '$do\_date'

echo "$do\_date"

echo "'$do\_date'"

echo '"$do\_date"'

echo `date`

2）查看执行结果

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

$do\_date

2019-02-10

'2019-02-10'

"$do\_date"

2019年 05月 02日 星期四 21:02:08 CST

3）总结：

（1）单引号不取变量值

（2）双引号取变量值

（3）反引号`，执行引号中命令

（4）双引号内部嵌套单引号，取出变量值

（5）单引号内部嵌套双引号，不取出变量值

### 3.3.5 Tez引擎优点？

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

Mr/tez/spark区别：

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

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

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

## 3.4 需求逻辑（重点）

### 3.4.1 如何分析用户活跃？

在启动日志中统计不同设备id出现次数。

### 3.4.2 如何分析用户新增？vivo

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

### 3.4.3 如何分析用户1天留存？

留存用户=前一天新增 join 今天活跃

用户留存率=留存用户/前一天新增

### 3.4.4 如何分析沉默用户？

(登录时间为7天前,且只出现过一次)

按照设备id对日活表分组，登录次数为1，且是在一周前登录。

### 3.4.5 如何分析本周回流用户？

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

### 3.4.6 如何分析流失用户？

(登录时间为7天前)

按照设备id对日活表分组，且七天内没有登录过。

### 3.4.7 如何分析最近连续3周活跃用户数？

按照设备id对周活进行分组，统计次数大于3次。

### 3.4.8 如何分析最近七天内连续三天活跃用户数？

1）查询出最近7天的活跃用户，并对用户活跃日期进行排名

2）计算用户活跃日期及排名之间的差值

3）对同用户及差值分组，统计差值个数

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

7天连续收藏、点赞、购买、加购、付款、浏览、商品点击、退货

1个月连续7天

连续两周：

# 第8章 手写代码

## 8.1 基本算法

### 8.1.1 冒泡排序

/\*\*

\* 冒泡排序 时间复杂度 O(n^2) 空间复杂度O(1)

\*/

public class BubbleSort {

public static void bubbleSort(int[] data) {

System.out.println("开始排序");

int arrayLength = data.length;

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

boolean flag = false;

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

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

int temp = data[j + 1];

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

data[j] = temp;

flag = true;

}

}

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

if (!flag)

break;

}

}

public static void main(String[] args) {

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

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

bubbleSort(data);

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

}

}

### 8.1.2 二分查找



图4-二分查找核心思路

实现代码：

/\*\*

 \* 二分查找 时间复杂度O(log2n);空间复杂度O(1)

 \*/

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

  if(left>right){//递归退出条件，找不到，返回-1

    -1

  }

  val midIndex = (left+right)/2

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

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

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

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

  }else{//查找到，返回下标

    midIndex

  }

}

拓展需求：当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到。

代码实现如下：

/\*

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

//分析

1. 返回的结果是一个可变数组 ArrayBuffer

2. 在找到结果时，向左边扫描，向右边扫描 [条件]

3. 找到结果后，就加入到ArrayBuffer

\*/

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

findVal: Int): ArrayBuffer[Int] = {

//找不到条件?

if (l > r) {

return ArrayBuffer()

}

val midIndex = (l + r) / 2

val midVal = arr(midIndex)

if (midVal > findVal) {

//向左进行递归查找

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

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

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

} else {

println("midIndex=" + midIndex)

//定义一个可变数组

val resArr = ArrayBuffer[Int]()

//向左边扫描

var temp = midIndex - 1

breakable {

while (true) {

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

break()

}

if (arr(temp) == findVal) {

resArr.append(temp)

}

temp -= 1

}

}

//将中间这个索引加入

resArr.append(midIndex)

//向右边扫描

temp = midIndex + 1

breakable {

while (true) {

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

break()

}

if (arr(temp) == findVal) {

resArr.append(temp)

}

temp += 1

}

}

return resArr

}

### 8.1.3 快排

图1-快速排序核心思想

代码实现：

/\*\*

\* 快排

\* 时间复杂度:平均时间复杂度为O(nlogn)

\* 空间复杂度:O(logn)，因为递归栈空间的使用问题

\*/

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

case Nil => Nil

case List() => List()

case head :: tail =>

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

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

}

### 8.1.4 归并



图2-归并排序核心思想

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



图3-归并排序“治”流程

代码实现：

/\*\*

\* 快排

\* 时间复杂度:O(nlogn)

\* 空间复杂度:O(n)

\*/

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

case (Nil, \_) => right

case (\_, Nil) => left

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

if (x <= y) x :: merge(xTail, right)

else y :: merge(left, yTail)

}

### 8.1.5 二叉树之Scala实现

1）二叉树概念



2）二叉树的特点

（1）树执行查找、删除、插入的时间复杂度都是O(logN)

（2）遍历二叉树的方法包括前序、中序、后序

（3）非平衡树指的是根的左右两边的子节点的数量不一致

（4）在非空二叉树中，第i层的结点总数不超过 , i>=1；

（5）深度为h的二叉树最多有个结点(h>=1)，最少有h个结点；

（6）对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1；

3） 二叉树的Scala代码实现

定义节点以及前序、中序、后序遍历

class TreeNode(treeNo:Int){

val no = treeNo

var left:TreeNode = null

var right:TreeNode = null

//后序遍历

def postOrder():Unit={

//向左递归输出左子树

if(this.left != null){

this.left.postOrder

}

//向右递归输出右子树

if (this.right != null) {

this.right.postOrder

}

//输出当前节点值

printf("节点信息 no=%d \n",no)

}

//中序遍历

def infixOrder():Unit={

//向左递归输出左子树

if(this.left != null){

this.left.infixOrder()

}

//输出当前节点值

printf("节点信息 no=%d \n",no)

//向右递归输出右子树

if (this.right != null) {

this.right.infixOrder()

}

}

//前序遍历

def preOrder():Unit={

//输出当前节点值

printf("节点信息 no=%d \n",no)

//向左递归输出左子树

if(this.left != null){

this.left.postOrder()

}

//向右递归输出右子树

if (this.right != null) {

this.right.preOrder()

}

}

//后序遍历查找

def postOrderSearch(no:Int): TreeNode = {

//向左递归输出左子树

var resNode:TreeNode = null

if (this.left != null) {

resNode = this.left.postOrderSearch(no)

}

if (resNode != null) {

return resNode

}

if (this.right != null) {

resNode = this.right.postOrderSearch(no)

}

if (resNode != null) {

return resNode

}

println("ttt~~")

if (this.no == no) {

return this

}

resNode

}

//中序遍历查找

def infixOrderSearch(no:Int): TreeNode = {

var resNode : TreeNode = null

//先向左递归查找

if (this.left != null) {

resNode = this.left.infixOrderSearch(no)

}

if (resNode != null) {

return resNode

}

println("yyy~~")

if (no == this.no) {

return this

}

//向右递归查找

if (this.right != null) {

resNode = this.right.infixOrderSearch(no)

}

return resNode

}

//前序查找

def preOrderSearch(no:Int): TreeNode = {

if (no == this.no) {

return this

}

//向左递归查找

var resNode : TreeNode = null

if (this.left != null) {

resNode = this.left.preOrderSearch(no)

}

if (resNode != null){

return resNode

}

//向右边递归查找

if (this.right != null) {

resNode = this.right.preOrderSearch(no)

}

return resNode

}

//删除节点

//删除节点规则

//1如果删除的节点是叶子节点，则删除该节点

//2如果删除的节点是非叶子节点，则删除该子树

def delNode(no:Int): Unit = {

//首先比较当前节点的左子节点是否为要删除的节点

if (this.left != null && this.left.no == no) {

this.left = null

return

}

//比较当前节点的右子节点是否为要删除的节点

if (this.right != null && this.right.no == no) {

this.right = null

return

}

//向左递归删除

if (this.left != null) {

this.left.delNode(no)

}

//向右递归删除

if (this.right != null) {

this.right.delNode(no)

}

}

}

定义二叉树，前序、中序、后序遍历，前序、中序、后序查找，删除节点

class BinaryTree{

var root:TreeNode = null

//后序遍历

def postOrder(): Unit = {

if (root != null){

root.postOrder()

}else {

println("当前二叉树为空，不能遍历")

}

}

//中序遍历

def infixOrder(): Unit = {

if (root != null){

root.infixOrder()

}else {

println("当前二叉树为空，不能遍历")

}

}

//前序遍历

def preOrder(): Unit = {

if (root != null){

root.preOrder()

}else {

println("当前二叉树为空，不能遍历")

}

}

//后序遍历查找

def postOrderSearch(no:Int): TreeNode = {

if (root != null) {

root.postOrderSearch(no)

}else{

null

}

}

//中序遍历查找

def infixOrderSeacher(no:Int): TreeNode = {

if (root != null) {

return root.infixOrderSearch(no)

}else {

return null

}

}

//前序查找

def preOrderSearch(no:Int): TreeNode = {

if (root != null) {

return root.preOrderSearch(no)

}else{

//println("当前二叉树为空，不能查找")

return null

}

}

//删除节点

def delNode(no:Int): Unit = {

if (root != null) {

//先处理一下root是不是要删除的

if (root.no == no){

root = null

}else {

root.delNode(no)

}

}

}

## 8.2 手写HQL

### 8.2.1 手写HQL 第1题

表结构：uid,subject\_id,score

求：找出所有科目成绩都大于某一学科平均成绩的学生

数据集如下

1001 01 90

1001 02 90

1001 03 90

1002 01 85

1002 02 85

1002 03 70

1003 01 70

1003 02 70

1003 03 85

1）建表语句

create table score(

uid string,

subject\_id string,

score int)

row format delimited fields terminated by '\t';

2）求出每个学科平均成绩

select

uid,

score,

avg(score) over(partition by subject\_id) avg\_score

from

score;t1

3）根据是否大于平均成绩记录flag，大于则记为0否则记为1

select

uid,

if(score>avg\_score,0,1) flag

from

t1;t2

4）根据学生id进行分组统计flag的和，和为0则是所有学科都大于平均成绩

select

uid

from

t2

group by

uid

having

sum(flag)=0;

5）最终SQL

select

uid

from

(select

uid,

if(score>avg\_score,0,1) flag

from

(select

uid,

score,

avg(score) over(partition by subject\_id) avg\_score

from

score)t1)t2

group by

uid

having

sum(flag)=0;

### 8.2.2 手写HQL 第2题

我们有如下的用户访问数据

|  |  |  |
| --- | --- | --- |
| userId | visitDate | visitCount |
| u01 | 2017/1/21 | 5 |
| u02 | 2017/1/23 | 6 |
| u03 | 2017/1/22 | 8 |
| u04 | 2017/1/20 | 3 |
| u01 | 2017/1/23 | 6 |
| u01 | 2017/2/21 | 8 |
| U02 | 2017/1/23 | 6 |
| U01 | 2017/2/22 | 4 |

要求使用SQL统计出每个用户的累积访问次数，如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 用户id | 月份 | 小计 | 累积 |
| u01 | 2017-01 | 11 | 11 |
| u01 | 2017-02 | 12 | 23 |
| u02 | 2017-01 | 12 | 12 |
| u03 | 2017-01 | 8 | 8 |
| u04 | 2017-01 | 3 | 3 |

数据集

u01 2017/1/21 5

u02 2017/1/23 6

u03 2017/1/22 8

u04 2017/1/20 3

u01 2017/1/23 6

u01 2017/2/21 8

u02 2017/1/23 6

u01 2017/2/22 4

1）创建表

create table action

(userId string,

visitDate string,

visitCount int)

row format delimited fields terminated by "\t";

2）修改数据格式

select

userId,

date\_format(regexp\_replace(visitDate,'/','-'),'yyyy-MM') mn,

visitCount

from

action;t1

3）计算每人单月访问量

select

userId,

mn,

sum(visitCount) mn\_count

from

t1

group by

userId,mn;t2

4）按月累计访问量

select

userId,

mn,

mn\_count,

sum(mn\_count) over(partition by userId order by mn)

from t2;

5）最终SQL

select

userId,

mn,

mn\_count,

sum(mn\_count) over(partition by userId order by mn)

from

( select

userId,

mn,

sum(visitCount) mn\_count

from

(select

userId,

date\_format(regexp\_replace(visitDate,'/','-'),'yyyy-MM') mn,

visitCount

from

action)t1

group by userId,mn)t2;

### 8.2.3 手写HQL 第3题

有50W个京东店铺，每个顾客访客访问任何一个店铺的任何一个商品时都会产生一条访问日志，访问日志存储的表名为Visit，访客的用户id为user\_id，被访问的店铺名称为shop，请统计：

1）每个店铺的UV（访客数）

2）每个店铺访问次数top3的访客信息。输出店铺名称、访客id、访问次数

数据集

u1 a

u2 b

u1 b

u1 a

u3 c

u4 b

u1 a

u2 c

u5 b

u4 b

u6 c

u2 c

u1 b

u2 a

u2 a

u3 a

u5 a

u5 a

u5 a

1）建表

create table visit(user\_id string,shop string) row format delimited fields terminated by '\t';

2）每个店铺的UV（访客数）

select shop,count(distinct user\_id) from visit group by shop;

3）每个店铺访问次数top3的访客信息。输出店铺名称、访客id、访问次数

（1）查询每个店铺被每个用户访问次数

select shop,user\_id,count(\*) ct

from visit

group by shop,user\_id;t1

（2）计算每个店铺被用户访问次数排名

select shop,user\_id,ct,rank() over(partition by shop order by ct) rk

from t1;t2

（3）取每个店铺排名前3的

select shop,user\_id,ct

from t2

where rk<=3;

（4）最终SQL

select

shop,

user\_id,

ct

from

(select

shop,

user\_id,

ct,

rank() over(partition by shop order by ct) rk

from

(select

shop,

user\_id,

count(\*) ct

from visit

group by

shop,

user\_id)t1

)t2

where rk<=3;

### 8.2.4 手写HQL 第4题

已知一个表STG.ORDER，有如下字段:Date，Order\_id，User\_id，amount。请给出sql进行统计:数据样例:2017-01-01,10029028,1000003251,33.57。

1）给出 2017年每个月的订单数、用户数、总成交金额。

2）给出2017年11月的新客数(指在11月才有第一笔订单)

建表

create table order\_tab(dt string,order\_id string,user\_id string,amount decimal(10,2)) row format delimited fields terminated by '\t';

1）给出 2017年每个月的订单数、用户数、总成交金额。

select

date\_format(dt,'yyyy-MM'),

count(order\_id),

count(distinct user\_id),

sum(amount)

from

order\_tab

where

date\_format(dt,'yyyy')='2017'

group by

date\_format(dt,'yyyy-MM');

2）给出2017年11月的新客数(指在11月才有第一笔订单)

select

count(user\_id)

from

order\_tab

group by

user\_id

having

date\_format(min(dt),'yyyy-MM')='2017-11';

### 8.2.5 手写HQL 第5题

有日志如下，请写出代码求得所有用户和活跃用户的总数及平均年龄。（活跃用户指连续两天都有访问记录的用户）日期 用户 年龄

数据集

2019-02-11,test\_1,23

2019-02-11,test\_2,19

2019-02-11,test\_3,39

2019-02-11,test\_1,23

2019-02-11,test\_3,39

2019-02-11,test\_1,23

2019-02-12,test\_2,19

2019-02-13,test\_1,23

2019-02-15,test\_2,19

2019-02-16,test\_2,19

1）建表

create table user\_age(dt string,user\_id string,age int)row format delimited fields terminated by ',';

2）按照日期以及用户分组，按照日期排序并给出排名

select

dt,

user\_id,

min(age) age,

rank() over(partition by user\_id order by dt) rk

from

user\_age

group by

dt,user\_id;t1

3）计算日期及排名的差值

select

user\_id,

age,

date\_sub(dt,rk) flag

from

t1;t2

4）过滤出差值大于等于2的，即为连续两天活跃的用户

select

user\_id,

min(age) age

from

t2

group by

user\_id,flag

having

count(\*)>=2;t3

5）对数据进行去重处理（一个用户可以在两个不同的时间点连续登录），例如：a用户在1月10号1月11号以及1月20号和1月21号4天登录。

select

user\_id,

min(age) age

from

t3

group by

user\_id;t4

6）计算活跃用户（两天连续有访问）的人数以及平均年龄

select

count(\*) ct,

cast(sum(age)/count(\*) as decimal(10,2))

from t4;

7）对全量数据集进行按照用户去重

select

user\_id,

min(age) age

from

user\_age

group by

user\_id;t5

8）计算所有用户的数量以及平均年龄

select

count(\*) user\_count,

cast((sum(age)/count(\*)) as decimal(10,1))

from

t5;

9）将第5步以及第7步两个数据集进行union all操作

select

0 user\_total\_count,

0 user\_total\_avg\_age,

count(\*) twice\_count,

cast(sum(age)/count(\*) as decimal(10,2)) twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

(select

user\_id,

min(age) age

from

(

select

user\_id,

age,

date\_sub(dt,rk) flag

from

(

select

dt,

user\_id,

min(age) age,

rank() over(partition by user\_id order by dt) rk

from

user\_age

group by

dt,user\_id

)t1

)t2

group by

user\_id,flag

having

count(\*)>=2)t3

group by

user\_id

)t4

union all

select

count(\*) user\_total\_count,

cast((sum(age)/count(\*)) as decimal(10,1)),

0 twice\_count,

0 twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

user\_age

group by

user\_id

)t5;t6

10）求和并拼接为最终SQL

select

sum(user\_total\_count),

sum(user\_total\_avg\_age),

sum(twice\_count),

sum(twice\_count\_avg\_age)

from

(select

0 user\_total\_count,

0 user\_total\_avg\_age,

count(\*) twice\_count,

cast(sum(age)/count(\*) as decimal(10,2)) twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

(select

user\_id,

min(age) age

from

(

select

user\_id,

age,

date\_sub(dt,rk) flag

from

(

select

dt,

user\_id,

min(age) age,

rank() over(partition by user\_id order by dt) rk

from

user\_age

group by

dt,user\_id

)t1

)t2

group by

user\_id,flag

having

count(\*)>=2)t3

group by

user\_id

)t4

union all

select

count(\*) user\_total\_count,

cast((sum(age)/count(\*)) as decimal(10,1)),

0 twice\_count,

0 twice\_count\_avg\_age

from

(

select

user\_id,

min(age) age

from

user\_age

group by

user\_id

)t5)t6;

### 8.2.6 手写HQL 第6题

请用sql写出所有用户中在今年10月份第一次购买商品的金额，表ordertable字段（购买用户：userid，金额：money，购买时间：paymenttime(格式：2017-10-01)，订单id：orderid）

1）建表

create table ordertable(

userid string,

money int,

paymenttime string,

orderid string)

row format delimited fields terminated by '\t';

2）查询出

select

userid,

min(paymenttime) paymenttime

from

ordertable

where

date\_format(paymenttime,'yyyy-MM')='2017-10'

group by

userid;t1

select

t1.userid,

t1.paymenttime,

od.money

from

t1

join

ordertable od

on

t1.userid=od.userid

and

t1.paymenttime=od.paymenttime;

select

t1.userid,

t1.paymenttime,

od.money

from

(select

userid,

min(paymenttime) paymenttime

from

ordertable

where

date\_format(paymenttime,'yyyy-MM')='2017-10'

group by

userid)t1

join

ordertable od

on

t1.userid=od.userid

and

t1.paymenttime=od.paymenttime;

### 8.2.7 手写HQL 第7题

有一个线上服务器访问日志格式如下（用sql答题）

时间 接口 ip地址

2016-11-09 11：22：05 /api/user/login 110.23.5.33

2016-11-09 11：23：10 /api/user/detail 57.3.2.16

.....

2016-11-09 23：59：40 /api/user/login 200.6.5.166

求11月9号下午14点（14-15点），访问api/user/login接口的top10的ip地址

数据集

2016-11-09 14:22:05 /api/user/login 110.23.5.33

2016-11-09 11:23:10 /api/user/detail 57.3.2.16

2016-11-09 14:59:40 /api/user/login 200.6.5.166

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 11:23:10 /api/user/detail 57.3.2.16

2016-11-09 23:59:40 /api/user/login 200.6.5.166

2016-11-09 14:22:05 /api/user/login 110.23.5.34

2016-11-09 11:23:10 /api/user/detail 57.3.2.16

2016-11-09 23:59:40 /api/user/login 200.6.5.166

2016-11-09 14:22:05 /api/user/login 110.23.5.35

2016-11-09 14:23:10 /api/user/detail 57.3.2.16

2016-11-09 23:59:40 /api/user/login 200.6.5.166

2016-11-09 14:59:40 /api/user/login 200.6.5.166

2016-11-09 14:59:40 /api/user/login 200.6.5.166

1）建表

create table ip(

time string,

interface string,

ip string)

row format delimited fields terminated by '\t';

2）最终SQL

select

ip,

interface,

count(\*) ct

from

ip

where

date\_format(time,'yyyy-MM-dd HH')>='2016-11-09 14'

and

date\_format(time,'yyyy-MM-dd HH')<='2016-11-09 15'

and

interface='/api/user/login'

group by

ip,interface

order by

ct desc

limit 2;t1

### 8.2.8 手写SQL 第8题

有一个账号表如下，请写出SQL语句，查询各自区组的money排名前十的账号（分组取前10）

1）建表（MySQL）

CREATE TABLE `account`

( `dist\_id` int（11）DEFAULT NULL COMMENT '区组id',

`account` varchar（100）DEFAULT NULL COMMENT '账号',

`gold` int（11）DEFAULT 0 COMMENT '金币'）;

2）最终SQL

select

\*

from

account as a

where

(select

count(distinct(a1.gold))

from

account as a1

where

a1.dist\_id=a.dist\_id

and

a1.gold>a.gold)<3;

### 8.2.9 手写HQL 第9题

1）有三张表分别为会员表（member）销售表（sale）退货表（regoods）

（1）会员表有字段memberid（会员id，主键）credits（积分）；

（2）销售表有字段memberid（会员id，外键）购买金额（MNAccount）；

（3）退货表中有字段memberid（会员id，外键）退货金额（RMNAccount）。

2）业务说明

（1）销售表中的销售记录可以是会员购买，也可以是非会员购买。（即销售表中的memberid可以为空）；

（2）销售表中的一个会员可以有多条购买记录；

（3）退货表中的退货记录可以是会员，也可是非会员；

（4）一个会员可以有一条或多条退货记录。

查询需求：分组查出销售表中所有会员购买金额，同时分组查出退货表中所有会员的退货金额，把会员id相同的购买金额-退款金额得到的结果更新到表会员表中对应会员的积分字段（credits）

数据集

sale

1001 50.3

1002 56.5

1003 235

1001 23.6

1005 56.2

25.6

33.5

regoods

1001 20.1

1002 23.6

1001 10.1

23.5

10.2

1005 0.8

1）建表

create table member(memberid string,credits double) row format delimited fields terminated by '\t';

create table sale(memberid string,MNAccount double) row format delimited fields terminated by '\t';

create table regoods(memberid string,RMNAccount double) row format delimited fields terminated by '\t';

2）最终SQL

insert into table member

select

t1.memberid,

MNAccount-RMNAccount

from

(select

memberid,

sum(MNAccount) MNAccount

from

sale

where

memberid!=''

group by

memberid

)t1

join

(select

memberid,

sum(RMNAccount) RMNAccount

from

regoods

where

memberid!=''

group by

memberid

)t2

on

t1.memberid=t2.memberid;

### 8.2.10 手写HQL 第10题

1.用一条SQL语句查询出每门课都大于80分的学生姓名

name   kecheng   fenshu

张三    语文    81

张三    数学    75

李四    语文    76

李四    数学    90

王五    语文    81

王五    数学    100

王五    英语    90

A: select distinct name from table where name not in (select distinct name from table where fenshu<=80)

B：select name from table group by name having min(fenshu)>80

2. 学生表 如下:  
自动编号   学号  姓名 课程编号 课程名称 分数  
1     2005001 张三  0001   数学   69  
2     2005002 李四  0001   数学   89  
3     2005001 张三  0001   数学   69  
删除除了自动编号不同, 其他都相同的学生冗余信息  
  
A: delete tablename where 自动编号 not in(select min(自动编号) from tablename group by学号, 姓名, 课程编号, 课程名称, 分数)

3.一个叫team的表，里面只有一个字段name,一共有4条纪录，分别是a,b,c,d,对应四个球队，现在四个球队进行比赛，用一条sql语句显示所有可能的比赛组合.

答：select a.name, b.name  
from team a, team b  
where a.name < b.name

4.面试题：怎么把这样一个  
year   month amount  
1991   1     1.1  
1991   2     1.2  
1991   3     1.3  
1991   4     1.4  
1992   1     2.1  
1992   2     2.2  
1992   3     2.3  
1992   4     2.4  
查成这样一个结果  
year m1  m2  m3 m4  
1991 1.1 1.2 1.3 1.4  
1992 2.1 2.2 2.3 2.4   
  
答案  
select year,   
(select amount from aaa m where month=1 and m.year=aaa.year) as m1,  
(select amount from aaa m where month=2 and m.year=aaa.year) as m2,  
(select amount from aaa m where month=3 and m.year=aaa.year) as m3,  
(select amount from aaa m where month=4 and m.year=aaa.year) as m4  
from aaa group by year

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*5.说明：复制表(只复制结构,源表名：a新表名：b)   
  
SQL: select \* into b from a where 1<>1 (where1=1，拷贝表结构和数据内容)  
ORACLE:create table b

As

Select \* from a where 1=2

[<>（不等于）(SQL Server Compact)

比较两个表达式。 当使用此运算符比较非空表达式时，如果左操作数不等于右操作数，则结果为 TRUE。 否则，结果为 FALSE。]

6.

原表:  
courseid coursename score  
-------------------------------------  
1 java 70  
2 oracle 90  
3 xml 40  
4 jsp 30  
5 servlet 80  
-------------------------------------  
为了便于阅读,查询此表后的结果显式如下(及格分数为60):  
courseid coursename score mark  
---------------------------------------------------  
1 java 70 pass  
2 oracle 90 pass  
3 xml 40 fail  
4 jsp 30 fail  
5 servlet 80 pass  
---------------------------------------------------  
写出此查询语句  
select courseid, coursename ,score ,if(score>=60, "pass","fail") as mark from course

7.表名：购物信息

购物人 商品名称 数量

A 甲 2

B 乙 4

C 丙 1

A 丁 2

B 丙 5

……

给出所有购入商品为两种或两种以上的购物人记录

答：select \* from 购物信息 where 购物人 in (select 购物人 from 购物信息 group by 购物人 having count(\*) >= 2);

8.

info 表

date result

2005-05-09 win

2005-05-09 lose

2005-05-09 lose

2005-05-09 lose

2005-05-10 win

2005-05-10 lose

2005-05-10 lose

如果要生成下列结果, 该如何写sql语句?

　　 win lose

2005-05-09 2 2

2005-05-10 1 2

答案：

(1) select date, sum(case when result = "win" then 1 else 0 end) as "win", sum(case when result = "lose" then 1 else 0 end) as "lose" from info group by date;

(2) select a.date, a.result as win, b.result as lose

　　from

　　(select date, count(result) as result from info where result = "win" group by date) as a

　　join

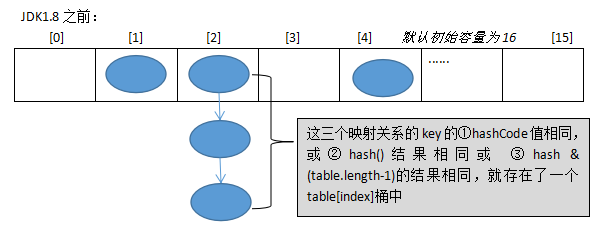
　　(select date, count(result) as result from info where result = "lose" group by date) as b

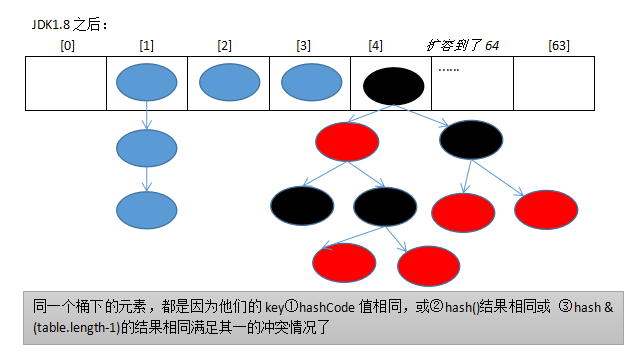
on a.date = b.date;

# 第9章 JavaSE

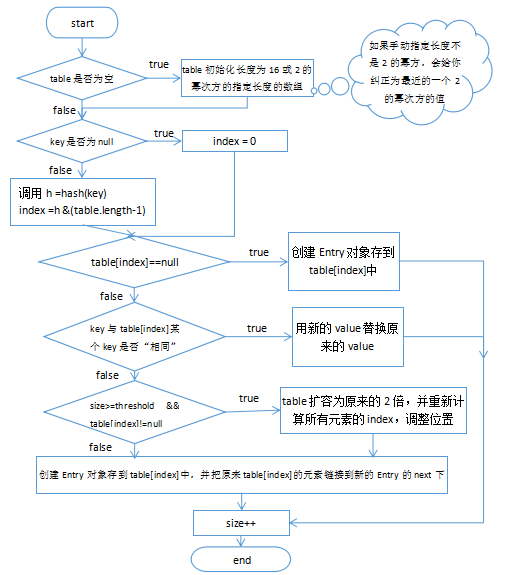
## 9.1 HhashMap底层源码，数据结构

hashMap的底层结构在jdk1.7中由数组+链表实现，在jdk1.8中由数组+链表+红黑树实现，以数组+链表的结构为例。

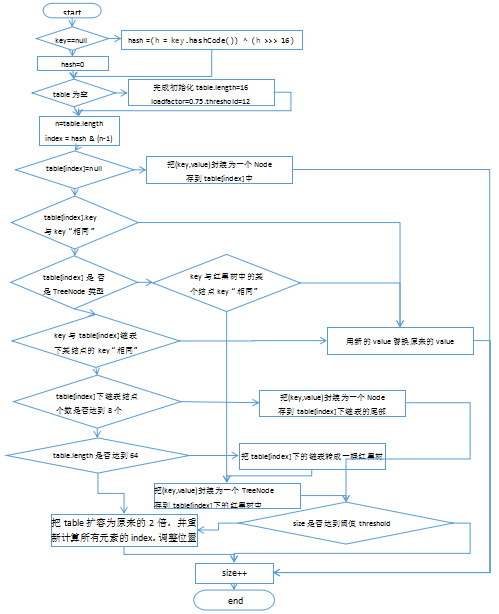




**JDK1.8之前Put方法：**



**JDK1.8之后Put方法：**



## 9.2 Java自带哪几种线程池？

**1）newCachedThreadPool**

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。这种类型的线程池特点是：

工作线程的创建数量几乎没有限制（其实也有限制的，数目为Interger. MAX\_VALUE）, 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间（默认为1分钟），则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

**2）newFixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

**3）newSingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序（FIFO, LIFO, 优先级）执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

**4）newScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。延迟3秒执行。

## 9.3 HashMap和HashTable区别

1. 线程安全性不同

HashMap是线程不安全的，HashTable是线程安全的，其中的方法是Synchronize的，在多线程并发的情况下，可以直接使用HashTabl，但是使用HashMap时必须自己增加同步处理。

1. 是否提供contains方法

HashMap只有containsValue和containsKey方法；HashTable有contains、containsKey和containsValue三个方法，其中contains和containsValue方法功能相同。

1. key和value是否允许null值

Hashtable中，key和value都不允许出现null值。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。

1. 数组初始化和扩容机制

 HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

 Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

## 9.4 TreeSet和HashSet区别

HashSet是采用hash表来实现的。其中的元素没有按顺序排列，add()、remove()以及contains()等方法都是复杂度为O(1)的方法。

TreeSet是采用树结构实现（红黑树算法）。元素是按顺序进行排列，但是add()、remove()以及contains()等方法都是复杂度为O(log (n))的方法。它还提供了一些方法来处理排序的set，如first()，last()，headSet()，tailSet()等等。

## 9.5 String buffer和String build区别

1、StringBuffer与StringBuilder中的方法和功能完全是等价的。

2、只是StringBuffer中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而StringBuilder没有这个修饰，可以被认为是线程不安全的。

3、在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全而StringBuffer则每次都需要判断锁，效率相对更低

## 9.6 Final、Finally、Finalize

final：修饰符（关键字）有三种用法：修饰类、变量和方法。修饰类时，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。修饰变量时，该变量使用中不被改变，必须在声明时给定初值，在引用中只能读取不可修改，即为常量。修饰方法时，也同样只能使用，不能在子类中被重写。

finally：通常放在try…catch的后面构造最终执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。

finalize：Object类中定义的方法，Java中允许使用finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize() 方法可以整理系统资源或者执行其他清理工作。

## 9.7 ==和Equals区别

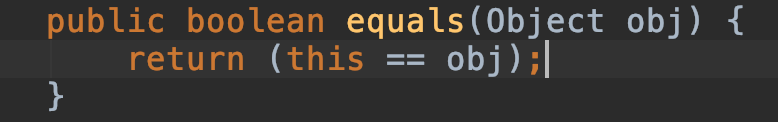
 == : 如果比较的是基本数据类型，那么比较的是变量的值

如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

 equals:如果没重写equals方法比较的是两个对象的地址值。

 如果重写了equals方法后我们往往比较的是对象中的属性的内容

equals方法是从Object类中继承的，默认的实现就是使用==



# 第10章 Redis

## 10.1 缓存穿透、缓存雪崩、缓存击穿

1）缓存穿透是指查询一个一定不存在的数据。由于缓存命不中时会去查询数据库，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决方案：

1. 是将空对象也缓存起来，并给它设置一个很短的过期时间，最长不超过5分钟
2. 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力

2）如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，就会造成缓存雪崩。

解决方案：

尽量让失效的时间点不分布在同一个时间点

3）缓存击穿，是指一个key非常热点，在不停的扛着大并发，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

解决方案：

可以设置key永不过期

## 10.2 哨兵模式

主从复制中反客为主的自动版，如果主机Down掉，哨兵会从从机中选择一台作为主机，并将它设置为其他从机的主机，而且如果原来的主机再次启动的话也会成为从机。

## 10.3 数据类型

|  |  |
| --- | --- |
| string | 字符串 |
| list | 可以重复的集合 |
| set | 不可以重复的集合 |
| hash | 类似于Map<String,String> |
| zset(sorted set） | 带分数的set |

## 10.4 持久化

**1）RDB持久化：**

1. **在指定的时间间隔内持久化**
2. **服务shutdown会自动持久化**

**③ 输入bgsave也会持久化**

**2）AOF : 以日志形式记录每个更新操作**

Redis重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。

**保存策略：**

推荐（并且也是默认）的措施为每秒持久化一次，这种策略可以兼顾速度和安全性。

**缺点：**

1 比起RDB占用更多的磁盘空间

2 恢复备份速度要慢

3 每次读写都同步的话，有一定的性能压力

4 存在个别Bug，造成恢复不能

**选择策略：**

官方推荐：

如果对数据不敏感，可以选单独用RDB；不建议单独用AOF，因为可能出现Bug;如果只是做纯内存缓存，可以都不用

## 11.5 悲观锁

执行操作前假设当前的操作肯定（或有很大几率）会被打断（悲观）。基于这个假设，我们在做操作前就会把相关资源锁定，不允许自己执行期间有其他操作干扰。

## 11.6 乐观锁

执行操作前假设当前操作不会被打断（乐观）。基于这个假设，我们在做操作前不会锁定资源，万一发生了其他操作的干扰，那么本次操作将被放弃。Redis使用的就是乐观锁。

# 第11章 MySql

## 11.1 MyISAM与InnoDB的区别

|  |  |  |
| --- | --- | --- |
| **对比项** | **MyISAM** | **InnoDB** |
| **外键** | **不支持** | **支持** |
| **事务** | **不支持** | **支持** |
| **行表锁** | **表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作** | **行锁,操作时只锁某一行，不对其它行有影响，**  **适合高并发的操作** |
| **缓存** | **只缓存索引，不缓存真实数据** | **不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响** |

## 11.2 索引优化

**数据结构：B+Tree**

一般来说能够达到range就可以算是优化了 idx name\_deptId

**口诀（两个法则加6种索引失效的情况）**

全值匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

LIKE百分写最右，覆盖索引不写\*；

不等空值还有OR，索引影响要注意；

VAR引号不可丢，SQL优化有诀窍。

## 11.3 b-tree和b+tree的区别

1) B-树的关键字、索引和记录是放在一起的， B+树的非叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。

2) 在B-树中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而B+树中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。

## 11.4 redis是单线程的，为什么那么快

1)完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。

2)数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的

3)采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

4)使用多路I/O复用模型，非阻塞IO

5)使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

## 11.5 MySQL的事务

**一、事务的基本要素（ACID）**

1、原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位

2、一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如A向B转账，不可能A扣了钱，B却没收到。

3、隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

4、持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

**二、事务的并发问题**

1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致

3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

**三、MySQL事务隔离级别**

事务隔离级别 脏读 不可重复读 幻读

读未提交（read-uncommitted） 是 是 是

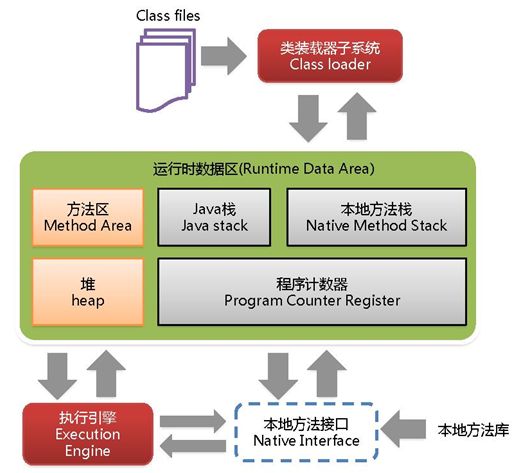
不可重复读（read-committed） 否 是 是

可重复读（repeatable-read） 否 否 是

串行化（serializable） 否 否 否

# 第12章 JVM

## 12.1 JVM内存分哪几个区，每个区的作用是什么?



java虚拟机主要分为以下几个区:

1. **方法区**：
2. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
3. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
4. 该区域是被线程共享的。
5. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。
6. **虚拟机栈**:
7. 虚拟机栈也就是我们平常所称的栈内存,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
8. 虚拟机栈是线程私有的，它的生命周期与线程相同。
9. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
10. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
11. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。
12. **本地方法栈**：  
    本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。
13. **堆**：

java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

1. **程序计数器：**

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

## 12.2 Java类加载过程?

Java类加载需要经历一下几个过程：

1. 加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。
2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
3. 在内存中生成该类的Class对象，作为该类的数据访问入口。
4. 验证

验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

1. 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。
5. 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

1. 解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

1. 初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

## 12.3 java中垃圾收集的方法有哪些?

**1）引用计数法**   应用于：微软的COM/ActionScrip3/Python等

a) 如果对象没有被引用，就会被回收，缺点：需要维护一个引用计算器

**2）复制算法** 年轻代中使用的是Minor GC，这种GC算法采用的是复制算法(Copying)

a) 效率高，缺点：需要内存容量大，比较耗内存

b) 使用在占空间比较小、刷新次数多的新生区

**3）标记清除** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率比较低，会差生碎片。

**4）标记压缩** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率低速度慢，需要移动对象，但不会产生碎片。

**5）标记清除压缩**标记清除-标记压缩的集合，多次GC后才Compact

a) 使用于占空间大刷新次数少的养老区，是3 4的集合体

## 12.4 如何判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:

1. 引用计数法
2. 可达性算法(引用链法)

## 12.5 什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
3. 系统类加载器（system class loader）也叫应用类加载器：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

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

**内存分配：**

1. **栈区**：栈分为java虚拟机栈和本地方法栈
2. **堆区**：堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区，主要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1。
3. **方法区**：被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）
4. **程序计数器**：当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。线程私有的。

**回收策略以及Minor GC和Major GC：**

1. 对象优先在堆的Eden区分配。
2. 大对象直接进入老年代。
3. 长期存活的对象将直接进入老年代。

当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor GC通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生GC的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

# 第13章 JUC

**13.1 Synchronized与Lock的区别**

1）Synchronized能实现的功能Lock都可以实现，而且Lock比Synchronized更好用，更灵活。

2）Synchronized可以自动上锁和解锁；Lock需要手动上锁和解锁

**13.2 Runnable和Callable的区别**

1）Runnable接口中的方法没有返回值；Callable接口中的方法有返回值

2）Runnable接口中的方法没有抛出异常；Callable接口中的方法抛出了异常

3）Runnable接口中的落地方法是call方法；Callable接口中的落地方法是run方法

**13.3 什么是分布式锁**

当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。分布式锁可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存，如 Redis，通过set (key,value,nx,px,timeout)方法添加分布式锁。

**13.4 什么是分布式事务**

分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。

# 第14章 面试说明

## 14.1 面试过程最关键的是什么？

1）大大方方的聊，放松

2）体现优势，避免劣势

## 14.2 面试时该怎么说？

1）语言表达清楚

（1）思维逻辑清晰，表达流畅

（2）一二三层次表达

2）所述内容不犯错

（1）不说前东家或者自己的坏话

（2）往自己擅长的方面说

（3）实质，对考官来说，内容听过，就是自我肯定；没听过，那就是个学习的过程。

## 14.3 面试技巧

### 14.3.1 六个常见问题

1）你的优点是什么？

大胆的说出自己各个方面的优势和特长

2）你的缺点是什么？

不要谈自己真实问题；用“缺点”衬托自己的优点

3）你的离职原因是什么？

* + 不说前东家坏话，哪怕被伤过
  + 合情合理合法
  + 不要说超过1个以上的原因

4）您对薪资的期望是多少？

* + 非终面不深谈薪资
  + 只说区间，不说具体数字
  + 底线是不低于当前薪资
  + 非要具体数字，区间取中间值，或者当前薪资的+20%

5）您还有什么想问的问题？

* + 这是体现个人眼界和层次的问题
  + 问题本身不在于面试官想得到什么样的答案，而在于你跟别的应聘者的对比
  + 标准答案：

公司希望我入职后的3-6个月内，给公司解决什么样的问题

公司（或者对这个部门）未来的战略规划是什么样子的？

以你现在对我的了解，您觉得我需要多长时间融入公司？

6）您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前。

### 14.3.2 两个注意事项

1）职业化的语言

2）职业化的形象

### 14.3.3 自我介绍（控制在4分半以内，不超过5分钟）

1）个人基本信息

2）工作履历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因

3）深度沟通（也叫压力面试）

刨根问底下沉式追问（注意是下沉式，而不是发散式的）

基本技巧：往自己熟悉的方向说

# 第15章 LeetCode题目精选

## 15.1 两数之和

问题链接：https://leetcode-cn.com/problems/two-sum/

### 15.1.1 问题描述

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

```

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9

所以返回 [0, 1]

```

### 15.1.2 参考答案

```java

class Solution {

public int[] twoSum(int[] nums, int target) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

int complement = target - nums[i];

if (map.containsKey(complement)) {

return new int[] { map.get(complement), i };

}

map.put(nums[i], i);

}

throw new IllegalArgumentException("No two sum solution");

}

}

```

## 15.2 爬楼梯

问题链接：https://leetcode-cn.com/problems/climbing-stairs/

### 15.2.1 问题描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

```

输入： 2

输出： 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

```

示例 2：

```

输入： 3

输出： 3

解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶

2. 1 阶 + 2 阶

3. 2 阶 + 1 阶

```

### 15.2.2 参考答案

```java

public class Solution {

public int climbStairs(int n) {

if (n == 1) {

return 1;

}

int[] dp = new int[n + 1];

dp[1] = 1;

dp[2] = 2;

for (int i = 3; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

}

```

## 15.3 翻转二叉树

链接：https://leetcode-cn.com/problems/invert-binary-tree/

### 15.3.1 问题描述

翻转一棵二叉树。

示例：

输入：

```

4

/ \

2 7

/ \ / \

1 3 6 9

```

输出：

```

4

/ \

7 2

/ \ / \

9 6 3 1

```

### 15.3.2 参考答案

```java

public TreeNode invertTree(TreeNode root) {

if (root == null) {

return null;

}

TreeNode right = invertTree(root.right);

TreeNode left = invertTree(root.left);

root.left = right;

root.right = left;

return root;

}

```

## 15.4 反转链表

链接：https://leetcode-cn.com/problems/reverse-linked-list/

### 15.4.1 问题描述

反转一个单链表。

示例:

```

输入: 1->2->3->4->5->NULL

输出: 5->4->3->2->1->NULL

```

### 15.4.2 参考答案

```java

public ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextTemp = curr.next;

curr.next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

```

## 15.5 LRU缓存机制

链接：https://leetcode-cn.com/problems/lru-cache/

### 15.5.1 问题描述

运用你所掌握的数据结构，设计和实现一个  LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例:

```

LRUCache cache = new LRUCache( 2 /\* 缓存容量 \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // 返回 1

cache.put(3, 3); // 该操作会使得密钥 2 作废

cache.get(2); // 返回 -1 (未找到)

cache.put(4, 4); // 该操作会使得密钥 1 作废

cache.get(1); // 返回 -1 (未找到)

cache.get(3); // 返回 3

cache.get(4); // 返回 4

```

### 15.5.2 参考答案

```java

class LRUCache extends LinkedHashMap<Integer, Integer>{

private int capacity;

public LRUCache(int capacity) {

super(capacity, 0.75F, true);

this.capacity = capacity;

}

public int get(int key) {

return super.getOrDefault(key, -1);

}

public void put(int key, int value) {

super.put(key, value);

}

@Override

protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {

return size() > capacity;

}

}

/\*\*

\* LRUCache 对象会以如下语句构造和调用:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

```

## 15.6 最长回文子串

链接：https://leetcode-cn.com/problems/longest-palindromic-substring/

### 15.6.1 问题描述

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

```

输入: "babad"

输出: "bab"

注意: "aba" 也是一个有效答案。

```

示例 2：

```

输入: "cbbd"

输出: "bb"

```

### 15.6.2 参考答案

```java

public String longestPalindrome(String s) {

if (s == null || s.length() < 1) return "";

int start = 0, end = 0;

for (int i = 0; i < s.length(); i++) {

int len1 = expandAroundCenter(s, i, i);

int len2 = expandAroundCenter(s, i, i + 1);

int len = Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

int L = left, R = right;

while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {

L--;

R++;

}

return R - L - 1;

}

```

## 15.7 有效的括号

链接：https://leetcode-cn.com/problems/valid-parentheses/

### 15.7.1 问题描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。

2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```

输入: "()"

输出: true

```

示例 2:

```

输入: "()[]{}"

输出: true

```

示例 3:

```

输入: "(]"

输出: false

```

示例 4:

```

输入: "([)]"

输出: false

```

示例 5:

```

输入: "{[]}"

输出: true

```

### 15.7.2 参考答案

```java

class Solution {

// Hash table that takes care of the mappings.

private HashMap<Character, Character> mappings;

// Initialize hash map with mappings. This simply makes the code easier to read.

public Solution() {

this.mappings = new HashMap<Character, Character>();

this.mappings.put(')', '(');

this.mappings.put('}', '{');

this.mappings.put(']', '[');

}

public boolean isValid(String s) {

// Initialize a stack to be used in the algorithm.

Stack<Character> stack = new Stack<Character>();

for (int i = 0; i < s.length(); i++) {

char c = s.charAt(i);

// If the current character is a closing bracket.

if (this.mappings.containsKey(c)) {

// Get the top element of the stack. If the stack is empty, set a dummy value of '#'

char topElement = stack.empty() ? '#' : stack.pop();

// If the mapping for this bracket doesn't match the stack's top element, return false.

if (topElement != this.mappings.get(c)) {

return false;

}

} else {

// If it was an opening bracket, push to the stack.

stack.push(c);

}

}

// If the stack still contains elements, then it is an invalid expression.

return stack.isEmpty();

}

}

```

## 15.8 数组中的第K个最大元素

链接：https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

### 15.8.1 问题描述

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```

输入: [3,2,1,5,6,4] 和 k = 2

输出: 5

```

示例 2:

```

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4

输出: 4

```

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

### 15.8.2 参考答案

```java

import java.util.Random;

class Solution {

int [] nums;

public void swap(int a, int b) {

int tmp = this.nums[a];

this.nums[a] = this.nums[b];

this.nums[b] = tmp;

}

public int partition(int left, int right, int pivot\_index) {

int pivot = this.nums[pivot\_index];

// 1. move pivot to end

swap(pivot\_index, right);

int store\_index = left;

// 2. move all smaller elements to the left

for (int i = left; i <= right; i++) {

if (this.nums[i] < pivot) {

swap(store\_index, i);

store\_index++;

}

}

// 3. move pivot to its final place

swap(store\_index, right);

return store\_index;

}

public int quickselect(int left, int right, int k\_smallest) {

/\*

Returns the k-th smallest element of list within left..right.

\*/

if (left == right) // If the list contains only one element,

return this.nums[left]; // return that element

// select a random pivot\_index

Random random\_num = new Random();

int pivot\_index = left + random\_num.nextInt(right - left);

pivot\_index = partition(left, right, pivot\_index);

// the pivot is on (N - k)th smallest position

if (k\_smallest == pivot\_index)

return this.nums[k\_smallest];

// go left side

else if (k\_smallest < pivot\_index)

return quickselect(left, pivot\_index - 1, k\_smallest);

// go right side

return quickselect(pivot\_index + 1, right, k\_smallest);

}

public int findKthLargest(int[] nums, int k) {

this.nums = nums;

int size = nums.length;

// kth largest is (N - k)th smallest

return quickselect(0, size - 1, size - k);

}

}

```

## 15.9 实现 Trie (前缀树)

### 15.9.1 问题描述

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

示例:

```

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // 返回 true

trie.search("app"); // 返回 false

trie.startsWith("app"); // 返回 true

trie.insert("app");

trie.search("app"); // 返回 true

```

说明:

- 你可以假设所有的输入都是由小写字母 a-z 构成的。

- 保证所有输入均为非空字符串。

### 15.9.2 参考答案

```java

class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

// Inserts a word into the trie.

public void insert(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

char currentChar = word.charAt(i);

if (!node.containsKey(currentChar)) {

node.put(currentChar, new TrieNode());

}

node = node.get(currentChar);

}

node.setEnd();

}

// search a prefix or whole key in trie and

// returns the node where search ends

private TrieNode searchPrefix(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

char curLetter = word.charAt(i);

if (node.containsKey(curLetter)) {

node = node.get(curLetter);

} else {

return null;

}

}

return node;

}

// Returns if the word is in the trie.

public boolean search(String word) {

TrieNode node = searchPrefix(word);

return node != null && node.isEnd();

}

}

```

## 15.10 编辑距离

链接：https://leetcode-cn.com/problems/edit-distance/

### 15.10.1 问题描述

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

1. 插入一个字符

2. 删除一个字符

3. 替换一个字符

示例 1:

```

输入: word1 = "horse", word2 = "ros"

输出: 3

解释:

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

```

示例 2:

```

输入: word1 = "intention", word2 = "execution"

输出: 5

解释:

intention -> inention (删除 't')

inention -> enention (将 'i' 替换为 'e')

enention -> exention (将 'n' 替换为 'x')

exention -> exection (将 'n' 替换为 'c')

exection -> execution (插入 'u')

```

### 15.10.2 参考答案

```java

class Solution {

public int minDistance(String word1, String word2) {

int n = word1.length();

int m = word2.length();

// if one of the strings is empty

if (n \* m == 0)

return n + m;

// array to store the convertion history

int [][] d = new int[n + 1][m + 1];

// init boundaries

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

d[i][0] = i;

}

for (int j = 0; j < m + 1; j++) {

d[0][j] = j;

}

// DP compute

for (int i = 1; i < n + 1; i++) {

for (int j = 1; j < m + 1; j++) {

int left = d[i - 1][j] + 1;

int down = d[i][j - 1] + 1;

int left\_down = d[i - 1][j - 1];

if (word1.charAt(i - 1) != word2.charAt(j - 1))

left\_down += 1;

d[i][j] = Math.min(left, Math.min(down, left\_down));

}

}

return d[n][m];

}

}

```