目录

一、代码手写

1.1 快排

```
/**
 * 快速排序 时间平均 O(NlogN) 最坏 O( N^2 ) 空间平均 O(logN)和 最坏
O(N)
 * 不稳定
 */
public class QuickRem {

   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));

      quickSort (data);

      System.out.println    ("   排   序   之   后   :    \n"    +
java.util.Arrays.toString (data));
   }

   private static void quickSort(int[] data) {

      if (data.length != 0) {
         subSort (data, 0, data.length - 1);
      }
   }

   private  static  void  subSort(int[]  data,  int  start/*0*/,
int end/*最后一个索引*/) {

      if (start < end) {

         int base = data[start];
         int i = start;
         int j = end + 1;//+1 +1 +1

         while (true) {

            while (i < end && data[++i] <= base)
               ;
            while (j > start && data[--j] >= base)
               ;
            if (i < j)
               swap (data, i, j);
            else
               break;
         }
         swap (data, start, j);
         subSort (data, start, j - 1);
         subSort (data, j + 1, end);
      }
   }
```

```
   private static void swap(int[] data, int i, int j) {
      int temp = data[i];
      data[i] = data[j];
      data[j] = temp;
   }
}
```

1.2 归并

```
/**
 * 归并排序 O(NlogN)  空间 O(N) 稳定
 */
public class MergeSort {
   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));

      mergeSort(data);

      System.out.println("   排   序   之   后   :   \n"   +
java.util.Arrays.toString(data));
   }

   public static void mergeSort(int[] data) {
      // 归并排序
      sort(data, 0, data.length - 1);
   }

   // 将索引从 left 到 right 范围的数组元素进行归并排序
   private static void sort(int[] data, int left, int right)
{
      if(left < right){

         //找出中间索引
         int center = (left + right)/2;
         sort(data,left,center);
         sort(data,center+1,right);

         //合并
         merge(data,left,center,right);
      }
   }

   // 将两个数组进行归并，归并前两个数组已经有序，归并后依然有序
   private static void merge(int[] data, int left, int center,
int right) {

      int[] tempArr = new int[data.length];
      int mid = center + 1;
      int third = left;
      int temp = left;

      while (left <= center && mid <= right) {
```

```
         if (data[left] - data[mid] <= 0) {
            tempArr[third++] = data[left++];
         } else {
            tempArr[third++] = data[mid++];
         }
      }

      while (mid <= right) {
         tempArr[third++] = data[mid++];
      }

      while (left <= center) {
         tempArr[third++] = data[left++];
      }

      while (temp <= right) {
         data[temp] = tempArr[temp++];
      }
   }
}
```

1.3 手写 Spark-WordCount

```
val conf: SparkConf =
new SparkConf().setMaster("local[*]").setAppName("WordCount")

val sc = new SparkContext(conf)

sc.textFile("/input")
  .flatMap(_.split(" "))
  .map((_, 1))
  .reduceByKey(_ + _)
  .saveAsTextFile("/output")

sc.stop()
```

1.4 冒泡排序

```
/**
 * 冒泡排序 时间O(n^2) 空间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));
      }
}
```

1.5 二分查找算法之 JAVA 实现

1.5.1 算法概念

二分查找算法也称为折半搜索、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。请注意这种算法是建立在有序数组基础上的。

1.5.2 算法思想

①搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；

②如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。

③如果在某一步骤数组为空，则代表找不到。

这种搜索算法每一次比较都使搜索范围缩小一半。

1.5.3 实现思路

①找出位于数组中间的值，并存放在一个变量中（为了下面的说明，变量暂时命名为 temp）；

②需要找到的 key 和 temp 进行比较；

③如果 key 值大于 temp，则把数组中间位置作为下一次计算的起点；重复① ②。

④如果 key 值小于 temp，则把数组中间位置作为下一次计算的终点；重复① ② ③。

⑤如果 key 值等于 temp，则返回数组下标，完成查找。

1.5.4 实现代码

```
/**
 * @param <E>
 * @param array 需要查找的有序数组
 * @param from 起始下标
 * @param to 终止下标
 * @param key 需要查找的关键字
 * @return
 * @throws Exception
 */
public static <E extends Comparable<E>> int binarySearch(E[]
array, int from, int to, E key) throws Exception {

   if (from < 0 || to < 0) {
      throw new IllegalArgumentException("params from &
length must larger than 0 .");
   }

   if (from <= to) {

      int middle = (from >>> 1) + (to >>> 1); // 右移即除 2
      E temp = array[middle];

      if (temp.compareTo(key) > 0) {
         to = middle - 1;
      } else if (temp.compareTo(key) < 0) {
         from = middle + 1;
      } else {
         return middle;
      }
   }

   return binarySearch(array, from, to, key);
}
```

1.6 二叉树之 Java 实现

1.6.1 二叉树概念

二叉树是一种非常重要的数据结构，它同时具有数组和链表各自的特点：它可以像数组一样快速查找，也可以像链表一样快速添加。但是他也有自己的缺点：删除操作复杂。

二叉树：是每个结点最多有两个子树的有序树，在使用二叉树的时候，数据并不是随便插入到节点中的，一个节点的左子节点的关键值必须小于此节点，右子节点的关键值必须大于或者是等于此节点，所以又称二叉查找树、二叉排序树、二叉搜索树。

完全二叉树：若设二叉树的高度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉

树。

满二叉树——除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。

深度——二叉树的层数，就是深度。

1.6.2 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；

1.6.3 二叉树的 Java 代码实现

首先是树的节点的定义，下面的代码中使用的是最简单的 int 基本数据类型作为节点的数据，如果要使用节点带有更加复杂的数据类型，换成对应的对象即可。

```
public class TreeNode {

 // 左节点
 private TreeNode lefTreeNode;
 // 右节点
 private TreeNode rightNode;
 // 数据
 private int value;

 private boolean isDelete;

 public TreeNode getLefTreeNode() {
  return lefTreeNode;
 }

 public void setLefTreeNode(TreeNode lefTreeNode) {
  this.lefTreeNode = lefTreeNode;
 }

 public TreeNode getRightNode() {
  return rightNode;
 }

 public void setRightNode(TreeNode rightNode) {
  this.rightNode = rightNode;
 }
```

```
 public int getValue() {
  return value;
 }

 public void setValue(int value) {
  this.value = value;
 }

 public boolean isDelete() {
  return isDelete;
 }

 public void setDelete(boolean isDelete) {
  this.isDelete = isDelete;
 }

 public TreeNode() {
  super();
 }

 public TreeNode(int value) {
  this(null, null, value, false);
 }

 public  TreeNode(TreeNode  lefTreeNode,  TreeNode  rightNode,
int value,
   boolean isDelete) {
  super();
  this.lefTreeNode = lefTreeNode;
  this.rightNode = rightNode;
  this.value = value;
  this.isDelete = isDelete;
 }

 @Override
 public String toString() {
  return   "TreeNode   [lefTreeNode="   +   lefTreeNode   +   ",
rightNode="
    +  rightNode  +  ",  value="  +  value  +  ",  isDelete="  +
isDelete
    + "]";
 }

}
```

下面给出二叉树的代码实现。由于在二叉树中进行节点的删除非常繁琐，因此，下面的代码使用的是利用节点的 isDelete 字段对节点的状态进行标识

```
public class BinaryTree {

 // 根节点
 private TreeNode root;

 public TreeNode getRoot() {
  return root;
 }

 /**
```

```
 * 插入操作
 *
 * @param value
 */
public void insert(int value) {

 TreeNode newNode = new TreeNode(value);

 if (root == null) {
  root = newNode;
  root.setLefTreeNode(null);
  root.setRightNode(null);
 } else {

  TreeNode currentNode = root;
  TreeNode parentNode;

  while (true) {

   parentNode = currentNode;
   // 往右放
   if (newNode.getValue() > currentNode.getValue()) {

    currentNode = currentNode.getRightNode();

    if (currentNode == null) {
     parentNode.setRightNode(newNode);
     return;
    }
   } else {
    // 往左放
    currentNode = currentNode.getLefTreeNode();

    if (currentNode == null) {
     parentNode.setLefTreeNode(newNode);
     return;
    }
   }
  }
 }
}

/**
 * 查找
 *
 * @param key
 * @return
 */
public TreeNode find(int key) {

 TreeNode currentNode = root;

 if (currentNode != null) {

  while (currentNode.getValue() != key) {

   if (currentNode.getValue() > key) {
    currentNode = currentNode.getLefTreeNode();
```

```
    } else {
     currentNode = currentNode.getRightNode();
    }

    if (currentNode == null) {
     return null;
    }

   }

   if (currentNode.isDelete()) {
    return null;
   } else {
    return currentNode;
   }

  } else {
   return null;
  }
 }

 /**
  * 中序遍历
  *
  * @param treeNode
  */
 public void inOrder(TreeNode treeNode) {

  if (treeNode != null && treeNode.isDelete() == false) {

   inOrder(treeNode.getLefTreeNode());

   System.out.println("--" + treeNode.getValue());

   inOrder(treeNode.getRightNode());
  }
 }
}
```

在上面对二叉树的遍历操作中，使用的是中序遍历，这样遍历出来的数据是增序的。

下面是测试代码:

```
public class Main {

 public static void main(String[] args) {

  BinaryTree tree = new BinaryTree();

  // 添加数据测试
  tree.insert(10);
  tree.insert(40);
  tree.insert(20);
  tree.insert(3);
  tree.insert(49);
  tree.insert(13);
  tree.insert(123);

  System.out.println("root=" + tree.getRoot().getValue());
```

```
  // 排序测试
  tree.inOrder(tree.getRoot());

  // 查找测试
  if (tree.find(10) != null) {
   System.out.println("找到了");
  } else {
   System.out.println("没找到");
  }

  // 删除测试
  tree.find(40).setDelete(true);

  if (tree.find(40) != null) {
   System.out.println("找到了");
  } else {
   System.out.println("没找到");
  }
 }
}
```

1.7 高效读取大数据文本文件（上亿行数据）

```
/**
     * 通过 BufferedRandomAccessFile 读取文件,推荐
     *
     * @param file     源文件
     * @param encoding 文件编码
     * @param pos      偏移量
     * @param num      读取量
     * @return pins 文件内容，pos 当前偏移量
     */
    public          static          Map<String,          Object>
BufferedRandomAccessFileReadLine(File  file,  String  encoding,
long pos, int num) {

        Map<String, Object> res = Maps.newHashMap();
        List<String> pins = Lists.newArrayList();
        res.put("pins", pins);
        BufferedRandomAccessFile reader = null;

        try {
            reader = new BufferedRandomAccessFile(file, "r");
            reader.seek(pos);

            for (int i = 0; i < num; i++) {
                String pin = reader.readLine();
                if (StringUtils.isBlank(pin)) {
                    break;
                }
                pins.add(new    String(pin.getBytes("8859_1"),
encoding));
            }

            res.put("pos", reader.getFilePointer());
        } catch (Exception e) {
```

```
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(reader);
        }

        return res;
    }
```

二、Linux

2.1 常用命令:df、ps、top、iotop、netstat

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

2.2 Shell 常用工具 （awk, sort, sed,cut）

awk	awk -F	指定文件拆分符
	awk -v	赋值一个用户定义变量
sort	-n	依照数值大小排序
	-r	以相反的顺序排序
	-t	定排序时所用的栏位
	-k	指定需要排序的栏位
sed	-e	直接在指令列模式上进行 sed 动作编辑
	-d	删除
	-s	查找并替换
cut	-f	取第几列
	-d	指定分割符分割列

2.2.1 问题 1：使用 Linux 命令查询 file1 中空行所在的行号答案：

2.2.2 问题 2：有文件 chengji.txt 内容如下:

张三 40

李四 50

王五 60

使用 Linux 命令计算第二列的和并输出

```
[hadoop@hadoop102 datas]$
cat chengji.txt | awk -F " " '{sum+=$2} END{print sum}'
150
```

2.2.3 问题 3：Shell 脚本里如何检查一个文件是否存在？如果不存在该如何处理？

```
#!/bin/bash
if [ -f file.txt ]; then
   echo "文件存在!"
else
   echo "文件不存在!"
fi
```

2.2.4 问题 4：用 shell 写一个脚本，对文本中无序的一列数字排序

```
[root@CentOS6-2 ~]# cat test.txt
9
8
7
6
5
4
3
2
10
1
[root@CentOS6-2 ~]
sort -n test.txt|awk '{a+=$0;print $0}END{print "SUM="a}'
1
2
3
4
5
6
7
8
9
10
SUM=55
```

2.2.5 问题 5：请用 shell 脚本写出查找当前文件夹（/home）下所有的文本文件内容中包含有字符”shen”的文件名称

```
[hadoop@hadoop102 datas]
grep -r "shen" /home | cut -d ":" -f 1

/home/hadoop/datas/sed.txt
/home/hadoop/datas/cut.txt
```

将 sed.txt 文件中 wo 替换为 ni

```
[hadoop@hadoop102 datas]$ sed 's/wo/ni/g' sed.txt
```

三、Hadoop

3.1 Hadoop 常用端口号

- dfs.namenode.http-address:50070
- 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

3.2 配置文件以及简单的 Hadoop 集群搭建

（1）配置文件：

hadoop-env.sh

core-site.xml

mapred-site.xml

hdfs-site.xml

（2）简单的集群搭建过程：

配置 SSH 免密登录

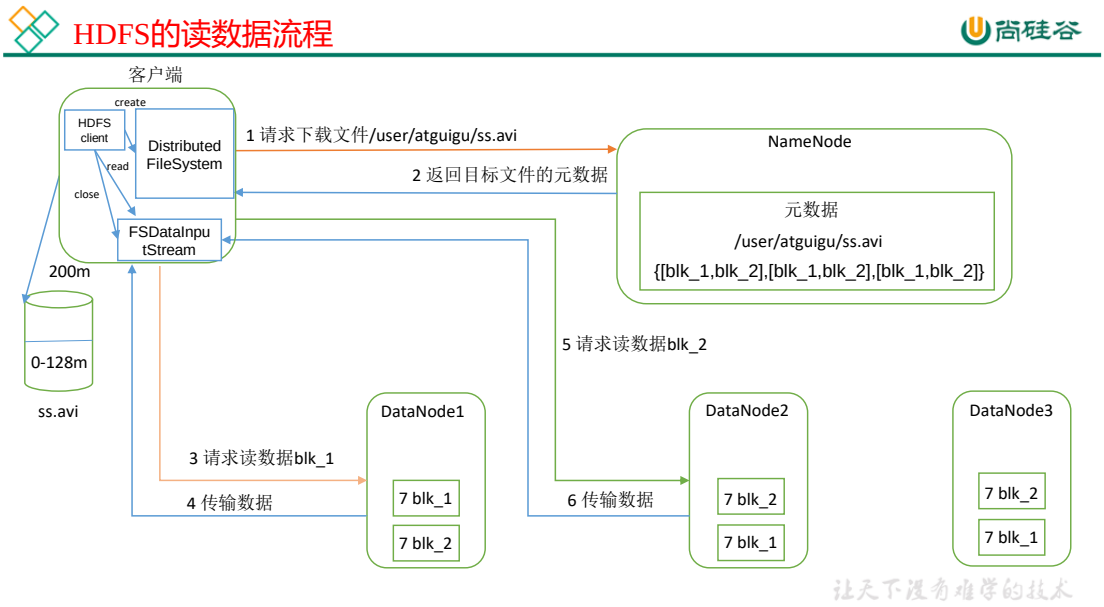
解压 hadoop 压缩包

配置 hadoop 核心文件: hadoop-env.sh，core-site.xml , mapred-site.xml ， hdfs-site.xml

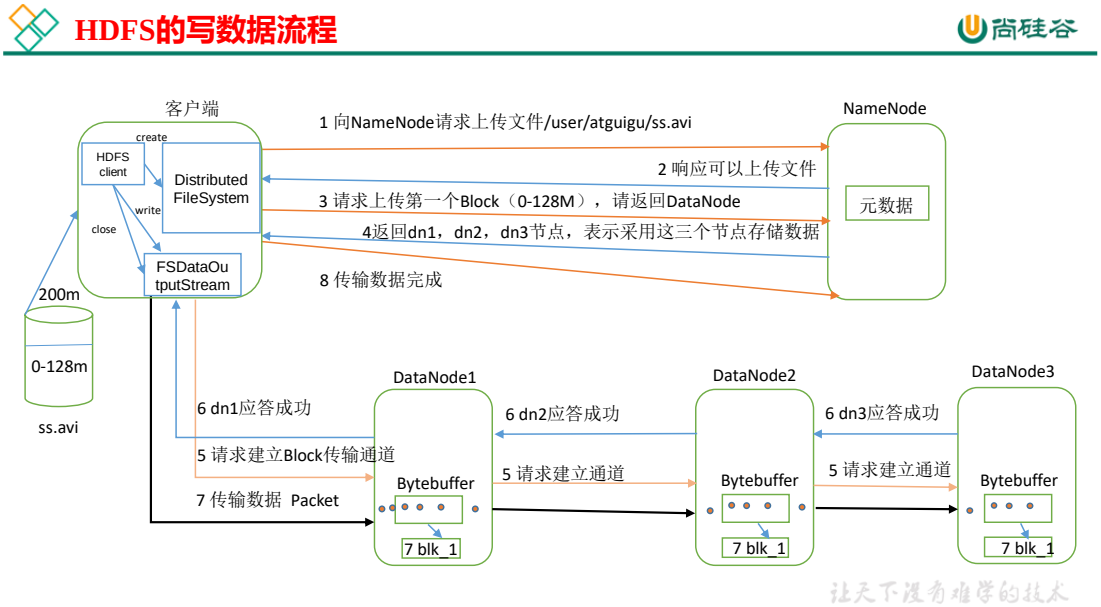
配置 hadoop 环境变量

格式化 namenode

3.3 HDFS 读流程



3.4 HDFS 写流程



3.5 MapReduce 过程发生了多少次排序

总共可能发生 4 次排序过程：

1）Map 阶段：

环形缓冲区：对 key 按照字典排序。排序手段：快速排序（能够手写快排）

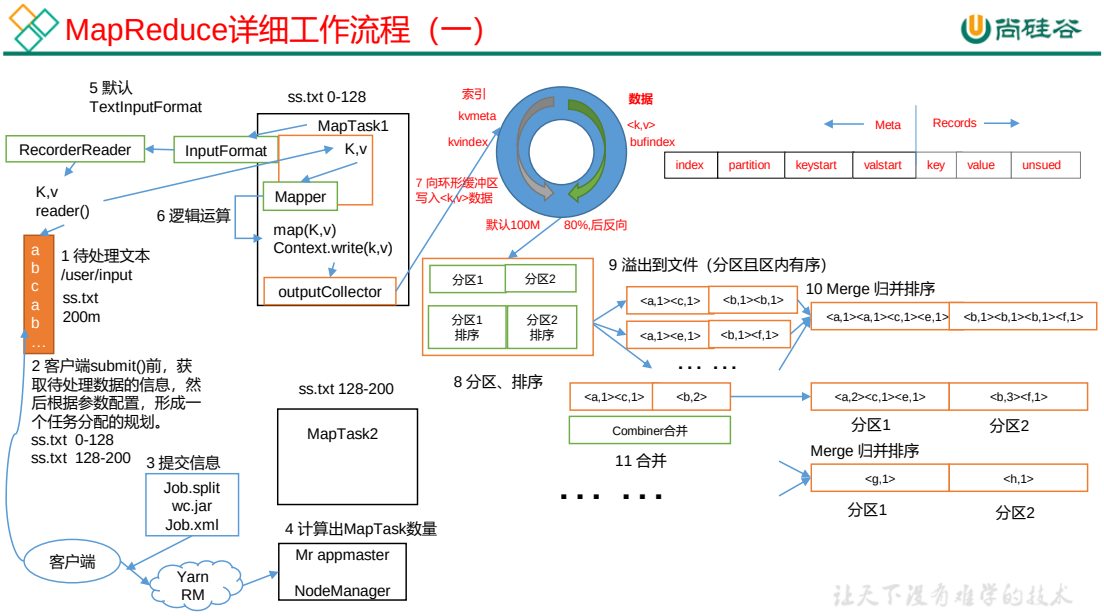
溢写到磁盘中：对多个溢写的文件进行排序。排序手段：分区归并排序（能够手写归并）

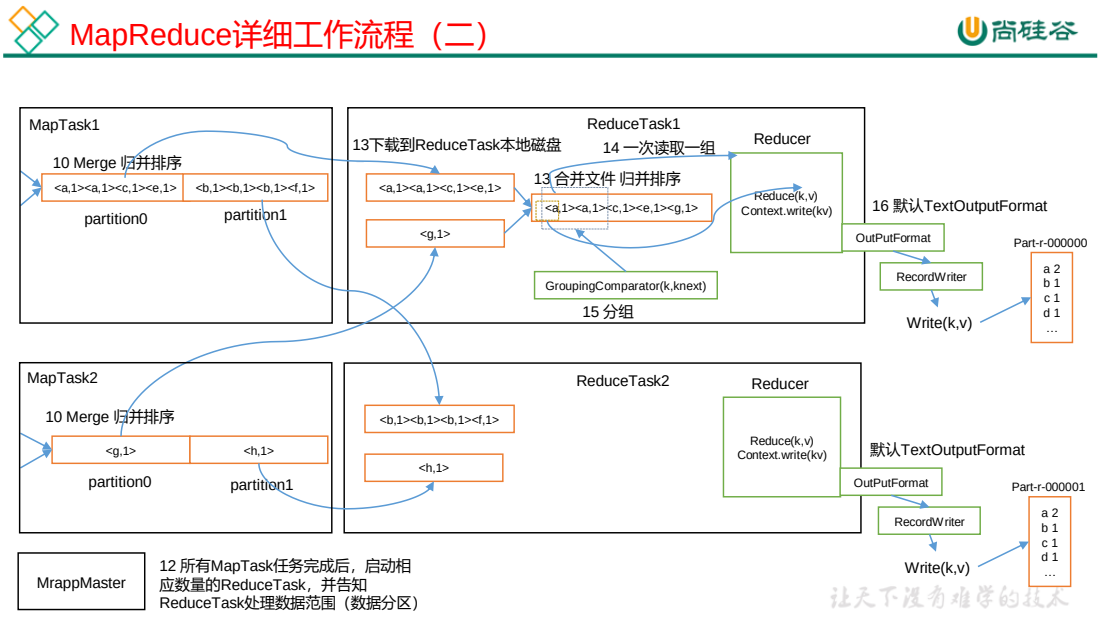
2）Reduce 阶段：

按指定分区读取到 reduce 缓存中（不够落盘）：归并排序

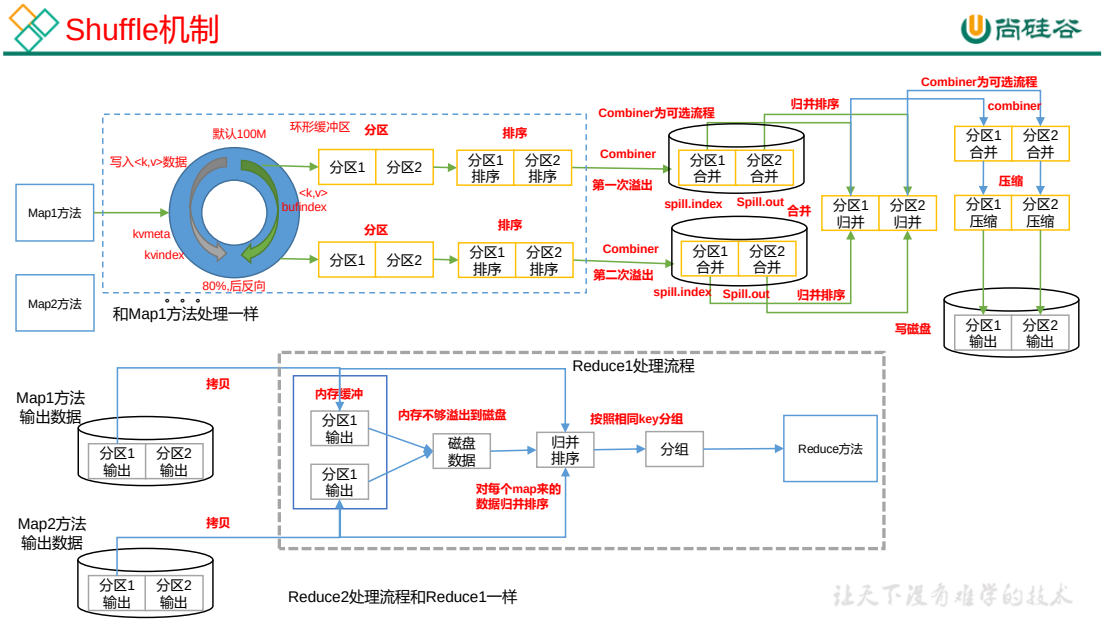
Reduce task 前分组排序：自定义

3.6 Mapreduce 的工作原理





3.7 Shuffle 机制



1）环形缓冲区:

排序方式：快排+字典序

默认溢写阈值：80%

默认大小：100M

提示：合理的调节环形缓冲区大小以及溢写阈值是一种常见的 MR 优化手段哦

2）切片机制：

a) 简单地按照文件的内容长度进行切片

b) 切片大小，默认等于 Block 大小

c) 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))

3.8 Join

Map Join:

a) Map Join 适用于一张表十分小、一张表很大的场景

b) 在 Map 端缓存多张表，提前处理业务逻辑，这样增加 Map 端业务，减少 Reduce 端数据的压力，尽可能的减少数据倾斜。

提示：Map join 是 MR 的一种很好的优化手段，大家在复习 Hadoop 优化的时候可以将 Hive 优化联系起来，因为我们数仓中使用的依旧是 MR 引擎（其他提示：ORC）

3.9 压缩

压缩格式	hadoop 自带？	算法	文件扩展名	是否可切分	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

四、YARN

4.1 调度器定义，区别

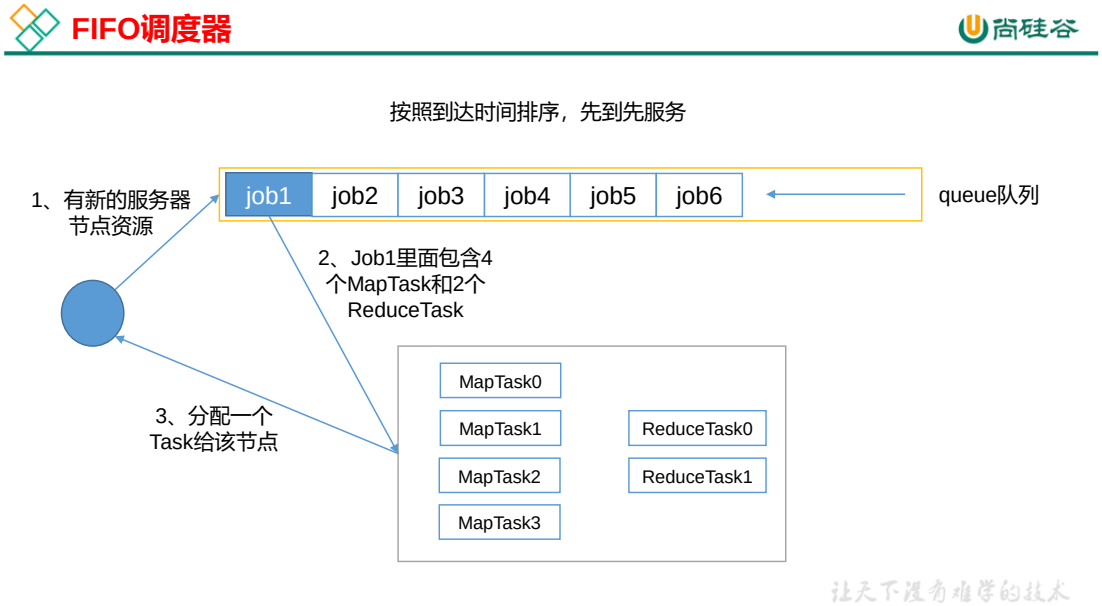
1）Hadoop 调度器重要分为三类：

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

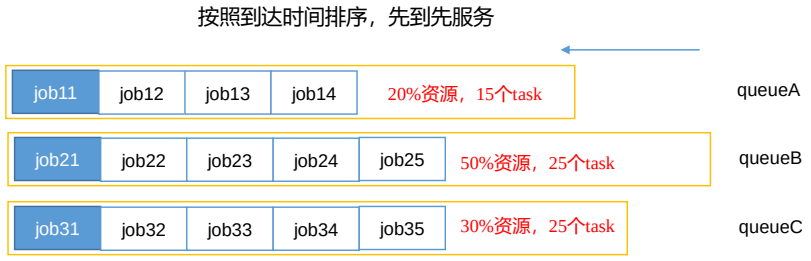
2）区别：

FIFO 调度器：先进先出



容量调度器：支持多个队列源量进行限定，选择一个正在运行的任务数与其计算资源之间比值最小的队列，队列内任务按照作业优先级、提交时间、用户资源量限制和内存限制进行排序。

容量调度器



1、支持多个队列，每个队列可配置一定的资源量，每个队列采用FIFO调度策略。

2、为了防止同一个用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源量进行限定。

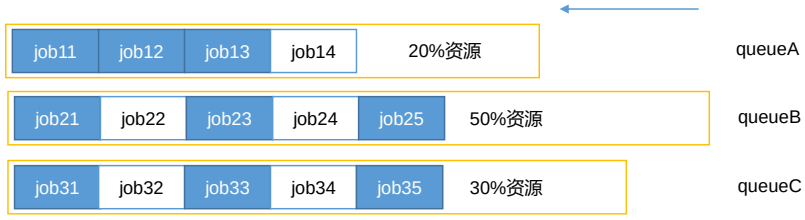
3、首先，计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个该比值最小的队列——最闲的。

4、其次，按照作业优先级和提交时间顺序，同时考虑用户资源量限制和内存限制对队列内任务排序。

5、三个队列同时按照任务的先后顺序依次执行，比如，job11、job21和job31分别排在队列最前面，先运行，也是并行运行。

公平调度器：支持多队列多用户，每个队列中的资源量可以配置，同一队列中的作业公平共享队列中所有资源，每个队列中的 job 按照优先级分配资源，优先级越高分配越多，在资源有限的情况下，每个 job 理想获得的计算资源与真实获得的计算资源的差值叫做缺额，同一队列中，job 的资源缺额越大，越优先执行，可以多个任务同时运行。

公平调度器



支持多队列多用户，每个队列中的资源量可以配置，同一队列中的作业公平共享队列中所有资源。

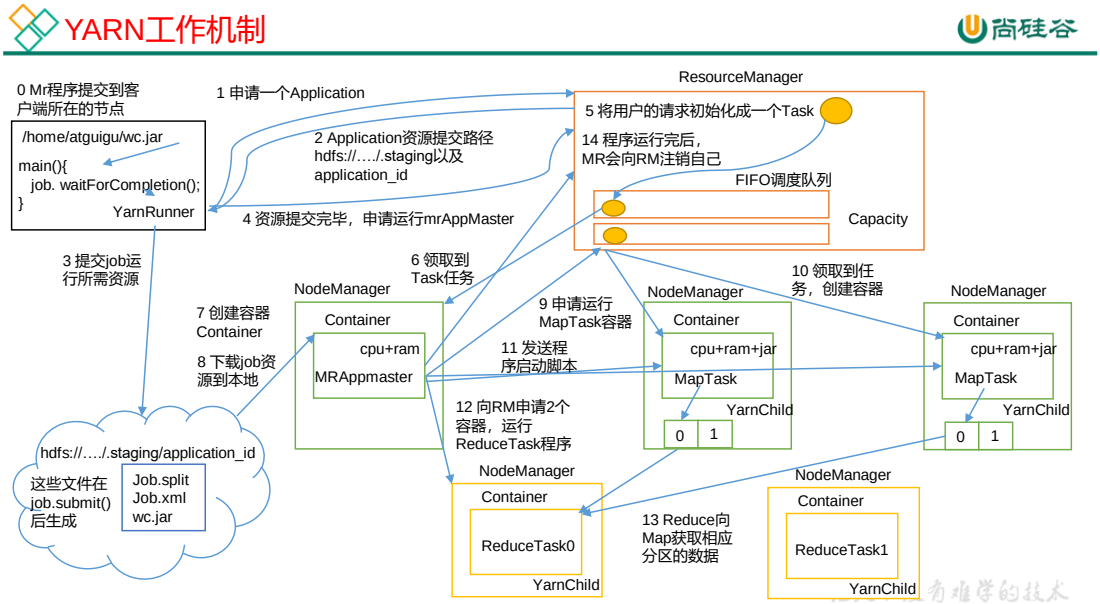
比如有三个队列：queueA、queueB和queueC，每个队列中的job按照优先级分配资源，优先级越高分配的资源越多，但是每个 job 都会分配到资源以确保公平。

在资源有限的情况下，每个job理想情况下获得的计算资源与实际获得的计算资源存在一种差距，这个差距就叫做缺额。

在同一个队列中，job的资源缺额越大，越先获得资源优先执行。作业是按照缺额的高低来先后执行的，而且可以看到上图有多个作业同时运行。

\

4.2 YARN 的 job 提交流程



五、Hadoop 优化

6 条-10 条

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。

（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

六、Zookeeper

6.1 选举机制

半数机制（Paxos 协议）：集群中半数以上机器存活，集群可用。所以 Zookeeper 适合装在奇数台机器上。

当集群开启的数量在半数以上时，就会将 Leader 给选出来，例如，有 id 为 1，2，3 三台机子，按顺序启动，第一台开启时，Zookeeper 的日志会报错，因为启动数量没有达到集一半:有 id 为 1，2，3 三台机子，按顺序启动 1.启了两台，总共三台），数量多于一半，然后根据 ID 的大小选出 Leader，则 2 号当选；2.3 号启动时，Leader 已经存在，则只能当小弟了。

6.2 常用命令

ls	查看子节点
get	获取节点信息
create	创建节点

七、Hive

7.1 Hive 的架构

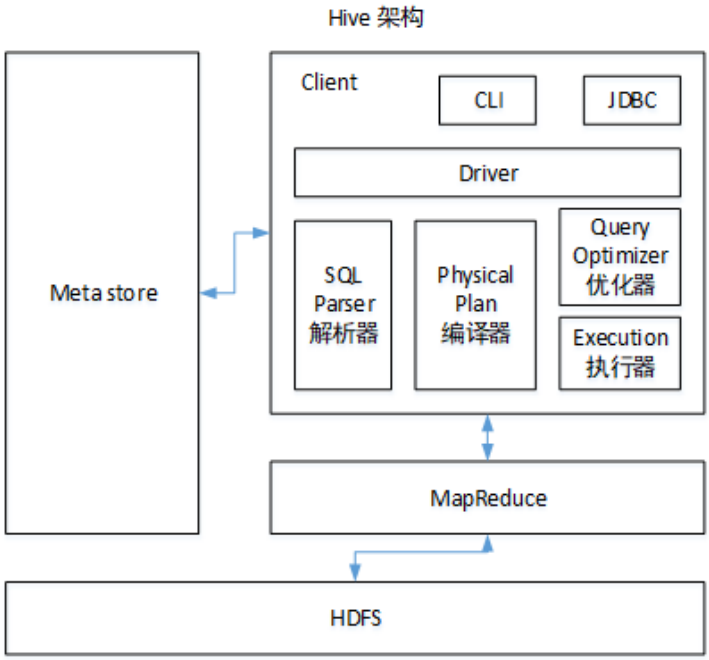
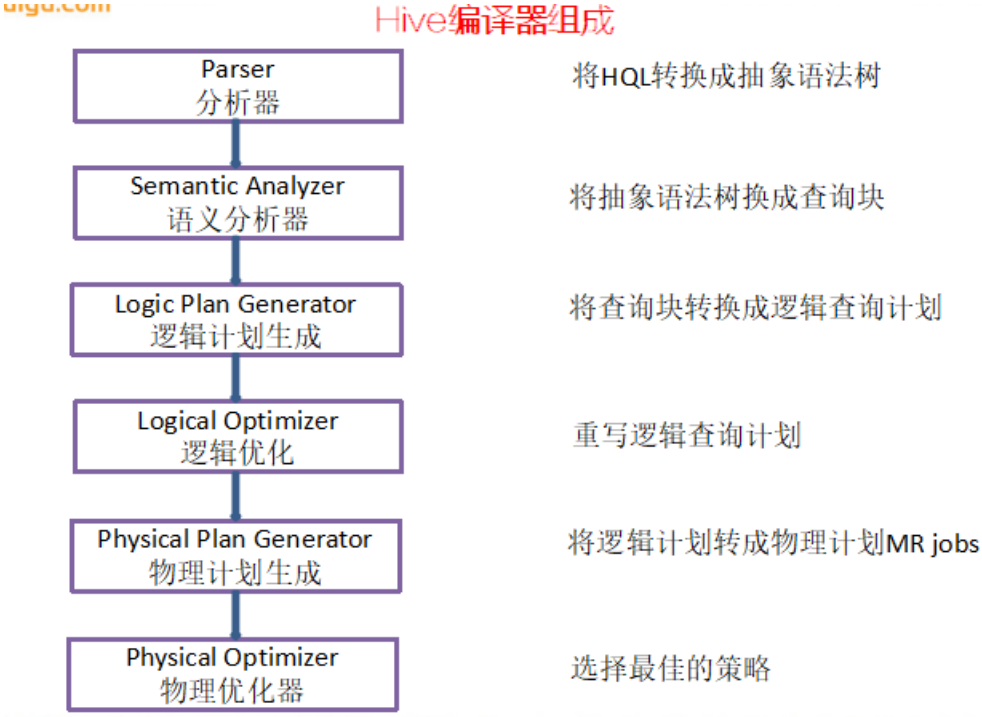


图 6-1 Hive 架构原理



7.2 Hive 和数据库比较

由于 Hive 采用了类似 SQL 的查询语言 HQL(Hive Query Language)，因此很容易

将 Hive 理解为数据库。其实从结构上来看，Hive 和数据库除了拥有类似的查询语言，再无类似之处。

7.2.1 查询语言

由于 SQL 被广泛的应用在数据仓库中，因此，专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便的使用 Hive 进行开发。

7.2.2 数据存储位置

Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。

7.2.3 数据更新

由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的。因此，Hive 中不建议对数据的改写，所有的数据都是在加载的时候确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使用 INSERT INTO … VALUES 添加数据，使用 UPDATE … SET 修改数据。

7.2.4 索引

Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入， Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。

7.2.5 执行

Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的。而数据库通常有自己的执行引擎。

7.2.6 执行延迟

Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，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。

7.2.7 可扩展性

由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性是和 Hadoop 的可扩展性是一致的（世界上最大的 Hadoop 集群在 Yahoo!，2009 年的规模在 4000 台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。

7.2.8 数据规模

由于 Hive 建立在集群上并可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。

7.3 内部表和外部表

a) 默认创建的表都是管理表，有时也被称为内部表。因为这种表，Hive 会（或多或少地）控制着数据的生命周期。Hive 默认情况下会将这些表的数据存储在由配置项 hive.metastore.warehouse.dir(例如，/user/hive/warehouse)所定义的目录的子目录下。

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。

b) Hive 并非认为其完全拥有这份数据。删除该表并不会删除掉这份数据，不过描述表的元数据信息会被删除掉

提示：抓住核心，外部表删除后不会影响原始数据

7.4 排序区别

a) Sort By：每个 Reducer 内部有序；

b) Order By：全局排序，只有一个 Reducer；

c) 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。但是排序只能是倒序排序，不能指定排序规则为 ASC 或者 DESC。

d) Distrbute By：类似 MR 中 partition，进行分区，结合 sort by 使用，Hive 要求 DISTRIBUTE BY 语句要写在 SORT BY 语句之前。

7.5 窗口函数

a) 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

b) CURRENT ROW：当前行

c) n PRECEDING：往前 n 行数据

d) n FOLLOWING：往后 n 行数据

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

f) LAG(col,n)：往前第 n 行数据

g) LEAD(col,n)：往后第 n 行数据

h) 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。

7.6 自定义 UDF 函数

回忆手机 APP 项目：UDF 在该项目中主要用来将时间戳转化为固定格式

7.7 优化

优化 6-10 条

7.7.1 Map Join

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。

7.7.2 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。

7.7.3 采用分桶技术

7.7.4 采用分区技术

7.7.5 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；

7.7.6 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。

7.7.7 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；

7.7.8 常用参数

（常用的一些可设置参数，具体数值按照需要进行调整！）

```
SET hive.optimize.skewjoin = true;
SET hive.skewjoin.key = 100000;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET mapred.reducer.tasks = 50;
// Hive 中间结果压缩和压缩输出
SET hive.exec.compress.output = true; -- 默认 false
SET hive.exec.compress.intermediate = true; -- 默认 false
SET mapred.output.compression.codec = org.apache.hadoop.io.compress.SnappyCodec; --
默认 org.apache.hadoop.io.compress.DefaultCodec
SET mapred.output.compression.type = BLOCK; -- 默认 BLOCK
```

```
// 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该值时，启动一个独立的 map-reduce 任务进行文件 merge
// 设置 map 和 reduce 数量
SET mapred.max.split.size = 256000000;
SET mapred.min.split.size = 64000000;
SET mapred.min.split.size.per.node = 64000000;
SET mapred.min.split.size.per.rack = 64000000;
SET hive.input.format = org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
SET hive.exec.reducers.bytes.per.reducer = 256000000; -- 默认 64M，每个 reducer 处理的数据量大小
// 设置数据倾斜和并行化
SET hive.exec.parallel = true; -- 并行执行
SET hive.exec.parallel.thread.number = 16;
SET mapred.job.reuse.jvm.num.tasks = 10;
SET hive.exec.dynamic.partition = true;
SET hive.optimize.cp = true; -- 列裁剪
SET hive.optimize.pruner = true; -- 分区裁剪
SET hive.groupby.skewindata = true; -- groupby 数据倾斜
SET hive.exec.mode.local.auto = true; --本地执行
SET hive.exec.mode.local.auto.input.files.max = 10; //map 数默认是 4，当 map 数小于 10 就会启动任务本地执行
SET hive.exec.mode.local.auto.inputbytes.max = 128000000 --默认是 128M
//关闭以下两个参数来完成关闭 Hive 任务的推测执行
SET mapred.map.tasks.speculative.execution=false;
SET mapred.reduce.tasks.speculative.execution=false;
```

7.8 情景题---》做题

7.8.1 蚂蚁森林植物申领统计

7.8.1.1 背景说明

下表记录了用户每天的蚂蚁森林低碳生活领取的记录流水。

```
table_name: user_low_carbon
user_id      data_dt          low_carbon
用户       日期         减少碳排放（g）
```

```
蚂蚁森林植物换购表，用于记录申领环保植物所需要减少的碳排放量
table_name:  plant_carbon
plant_id  plant_name    low_carbon
```

```
植物编号        植物名      换购植物所需要的碳
```

7.8.1.2 原始数据样例

user_low_carbon：

```
user_id   date_dt   low_carbon
u_001     2017/1/1  10
u_001     2017/1/2  150
u_001     2017/1/2  110
u_001     2017/1/2  10
u_001     2017/1/4  50
u_001     2017/1/4  10
u_001     2017/1/6  45
u_001     2017/1/6  90
u_002     2017/1/1  10
u_002     2017/1/2  150
u_002     2017/1/2  70
u_002     2017/1/3  30
u_002     2017/1/3  80
u_002     2017/1/4  150
u_002     2017/1/5  101
u_002     2017/1/6  68
…
```

plant_carbon：

```
plant_id  plant_name   plant_carbon
p001      梭梭树         17
p002      沙柳           19
p003      樟子树         146
p004      胡杨           215
…
```

1.创建表

```
create table user_low_carbon(user_id String,data_dt String,low_carbon int) row format delimited fields terminated by '\t';

create table plant_carbon(plant_id string,plant_name string,low_carbon int) row format delimited fields terminated by '\t';
```

2.加载数据

```
load data local inpath "/opt/module/data/low_carbon.txt" into table user_low_carbon;

load data local inpath "/opt/module/data/plant_carbon.txt" into table plant_carbon;
```

3.设置本地模式

```
set hive.exec.mode.local.auto=true;
```

7.8.1.2 题目一

蚂蚁森林植物申领统计
问题：假设 2017 年 1 月 1 日开始记录低碳数据（user_low_carbon），假设 2017 年 10 月 1 日之前满足申领条件的用户都申领了一颗 p004-胡杨，
剩余的能量全部用来领取“p002-沙柳” 。
统计在 10 月 1 日累计申领“p002-沙柳” 排名前 10 的用户信息；以及他比后一名多领了几颗沙柳。
得到的统计结果如下表样式：

```
user_id  plant_count  less_count(比后一名多领了几颗沙柳)
u_101    1000         100
u_088    900          400
u_103    500          …
```

SQL 流程

（1）先获取在 10 月 1 日前 low_carbon 总和最大的 11 个人

```
select user_id,sum(low_carbon) low_carbon_sum
from user_low_carbon
where        datediff(regexp_replace(data_dt,        "/",        "-
"),regexp_replace('2017/10/1', "/", "-")) < 0
group by user_id
order by low_carbon_sum
desc limit 11;t1
```

（2）查询出"胡杨"所需的低碳量

```
select low_carbon from plant_carbon where plant_id='p004';t2
```

（3）查询出"沙柳"所需的低碳量

```
select low_carbon from plant_carbon where plant_id='p002';t3
```

（4）计算出在申领一颗"胡杨"后可申领的"沙柳"棵数

```
select                              user_id,round((t1.low_carbon_sum-
t2.low_carbon)/t3.low_carbon) plant_count,
from t1,t2,t3;t4
```

（5）将每一行的下一个申领棵数放在当前行

```
select   user_id,plant_count,lead(plant_count,1,0)   over(sort   by
plant_count desc) as leadCount from t4;t5
```

（6）计算最终的差集（前十名比下一名多多少棵）

```
select user_id,(plant_count-leadCount) from t5 limit 10;
```

（7）最终 Sql

```
SELECT
    user_id,
    plant_count,
    (plant_count - leadCount)
FROM
```

```
    (
        SELECT
            user_id,
            plant_count,
            lead (plant_count, 1, 0) over (sort BY plant_count
DESC) AS leadCount
        FROM
            (
                SELECT
                    user_id,
                    round(
                        (
                            t1.low_carbon_sum - t2.low_carbon
                        ) / t3.low_carbon
                    ) plant_count
                FROM
                    (
                        SELECT
                            user_id,
                            sum(low_carbon) low_carbon_sum
                        FROM
                            user_low_carbon
                        WHERE
                            datediff(
                                regexp_replace (data_dt, "/", "-
"),
                                regexp_replace ('2017/10/1', "/",
"-")
                            ) < 0
                        GROUP BY
                            user_id
                        ORDER BY
                            low_carbon_sum DESC
                        LIMIT 11
                    ) t1,
                    (
                        SELECT
                            low_carbon
                        FROM
                            plant_carbon
                        WHERE
                            plant_id = 'p004'
                    ) t2,
                    (
                        SELECT
                            low_carbon
                        FROM
                            plant_carbon
                        WHERE
                            plant_id = 'p002'
                    ) t3
            ) t4
    ) t5
LIMIT 10;
```

（8）结果展示

```
+----------+--------------+-------+--+
| user_id  | plant_count  |  _c2  |
```

```
+----------+--------------+-------+--+
| u_007    | 66.0         | 2.0   |
| u_013    | 64.0         | 10.0  |
| u_008    | 54.0         | 7.0   |
| u_005    | 47.0         | 1.0   |
| u_010    | 46.0         | 2.0   |
| u_014    | 44.0         | 5.0   |
| u_011    | 39.0         | 1.0   |
| u_009    | 38.0         | 6.0   |
| u_006    | 32.0         | 9.0   |
| u_002    | 23.0         | 1.0   |
+----------+--------------+-------+--+
```

7.8.1.3 题目二

蚂蚁森林低碳用户排名分析
问题：查询 user_low_carbon 表中每日流水记录，条件为：
用户在 2017 年，连续三天（或以上）的天数里，
每天减少碳排放（low_carbon）都超过 100g 的用户低碳流水。
需要查询返回满足以上条件的 user_low_carbon 表中的记录流水。
例如用户 u_002 符合条件的记录如下，因为 2017/1/2~2017/1/5 连续四天的碳排放量之和都大于等于 100g：

```
seq（key） user_id data_dt  low_carbon
xxxxx10    u_002  2017/1/2  150
xxxxx11    u_002  2017/1/2  70
xxxxx12    u_002  2017/1/3  30
xxxxx13    u_002  2017/1/3  80
xxxxx14    u_002  2017/1/4  150
xxxxx14    u_002  2017/1/5  101
```

备注：统计方法不限于 sql、procedure、python, java 等

第一种解法：

SQL 流程

（1）按照用户及时间聚合，计算每个人每天的低碳量（2017 年）

```
select    user_id,data_dt,sum(low_carbon)    low_carbon_sum    from
user_low_carbon
where substring(data_dt,1,4)='2017'
group BY user_id,data_dt
having low_carbon_sum>100;t1
```

（2）将每一条数据的前后各两条数据的时间放置在一行，默认值为（1970/7/1）

```
select user_id,
       data_dt,
       lag(data_dt,2,"1970/7/1")  over(partition  by  user_id)  as
lag2Date,
       lag(data_dt,1,"1970/7/1")  over(partition  by  user_id)  as
lag1Date,
       lead(data_dt,1,"1970/7/1")  over(partition  by  user_id)  as
lead1Date,
```

```
       lead(data_dt,2,"1970/7/1")  over(partition  by  user_id)  as
lead2Date
from t1;t2
```

（3）计算每一天数据时间与前后两条数据之间的差值

```
select user_id,
       data_dt,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lag2Date, "/", "-")) lag2,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lag1Date, "/", "-")) lag1,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lead1Date, "/", "-")) lead1,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lead2Date, "/", "-")) lead2
from (select user_id,
       data_dt,
       lag(data_dt,2,"1970/7/1")  over(partition  by  user_id)  as
lag2Date,
       lag(data_dt,1,"1970/7/1")  over(partition  by  user_id)  as
lag1Date,
       lead(data_dt,1,"1970/7/1")  over(partition  by  user_id)  as
lead1Date,
       lead(data_dt,2,"1970/7/1")  over(partition  by  user_id)  as
lead2Date
from t2;t3
```

（4）取出最终需要的值，连续 3 天的（user_id,data_dt）

```
select user_id,data_dt
from t3
where (lag2=2 and lag1 =1) or (lag1 =1 and lead1 = -1) or(lead1=-
1 and lead2 = -2);t4
```

（5）与原表 Join 得到最终需要的结果

```
select t5.user_id,t5.data_dt,t5.low_carbon
from user_low_carbon t5
join t4
where t4.user_id = t5.user_id and t4.data_dt = t5.data_dt;
```

（6）最终 Sql

```
select t5.user_id,t5.data_dt,t5.low_carbon
from user_low_carbon t5
join (select user_id,data_dt
from (select user_id,
       data_dt,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lag2Date, "/", "-")) lag2,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lag1Date, "/", "-")) lag1,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lead1Date, "/", "-")) lead1,
       datediff(regexp_replace(data_dt,          "/",          "-
"),regexp_replace(lead2Date, "/", "-")) lead2
from (select user_id,
       data_dt,
       lag(data_dt,2,"1970/7/1")  over(partition  by  user_id)  as
```

```
lag2Date,
       lag(data_dt,1,"1970/7/1")  over(partition  by  user_id)  as
lag1Date,
       lead(data_dt,1,"1970/7/1")  over(partition  by  user_id)  as
lead1Date,
       lead(data_dt,2,"1970/7/1")  over(partition  by  user_id)  as
lead2Date
from  (select  user_id,data_dt,sum(low_carbon)  low_carbon_sum  from
user_low_carbon
where substring(data_dt,1,4)='2017'
group BY user_id,data_dt
having low_carbon_sum>100)t1)t2)t3
where (lag2=2 and lag1 =1) or (lag1 =1 and lead1 = -1) or(lead1=-
1 and lead2 = -2))t4
where t4.user_id = t5.user_id and t4.data_dt = t5.data_dt;

SELECT
    t5.user_id,
    t5.data_dt,
    t5.low_carbon
FROM
    user_low_carbon t5
JOIN (
    SELECT
        user_id,
        data_dt
    FROM
        (
            SELECT
                user_id,
                data_dt,
                datediff(
                    regexp_replace (data_dt, "/", "-"),
                    regexp_replace (lag2Date, "/", "-")
                ) lag2,
                datediff(
                    regexp_replace (data_dt, "/", "-"),
                    regexp_replace (lag1Date, "/", "-")
                ) lag1,
                datediff(
                    regexp_replace (data_dt, "/", "-"),
                    regexp_replace (lead1Date, "/", "-")
                ) lead1,
                datediff(
                    regexp_replace (data_dt, "/", "-"),
                    regexp_replace (lead2Date, "/", "-")
                ) lead2
            FROM
                (
                    SELECT
                        user_id,
                        data_dt,
                        lag   (data_dt,   2,   "1970/7/1")   over
(PARTITION BY user_id) AS lag2Date,
                        lag   (data_dt,   1,   "1970/7/1")   over
(PARTITION BY user_id) AS lag1Date,
                        lead   (data_dt,   1,   "1970/7/1")   over
(PARTITION BY user_id) AS lead1Date,
```

```
                        lead   (data_dt,   2,   "1970/7/1")   over
(PARTITION BY user_id) AS lead2Date
                    FROM
                        (
                            SELECT
                                user_id,
                                data_dt,
                                sum(low_carbon) low_carbon_sum
                            FROM
                                user_low_carbon
                            WHERE
                                substring(data_dt, 1, 4) = '2017'
                            GROUP BY
                                user_id,
                                data_dt
                            HAVING
                                low_carbon_sum > 100
                        ) t1
                ) t2
        ) t3
    WHERE
        (lag2 = 2 AND lag1 = 1)
    OR (lag1 = 1 AND lead1 = - 1)
    OR (lead1 =- 1 AND lead2 = - 2)
) t4
WHERE
    t4.user_id = t5.user_id
AND t4.data_dt = t5.data_dt;
```

(7) 结果展示

```
+-------------+-------------+----------------+--+
| t5.user_id  | t5.data_dt  | t5.low_carbon  |
+-------------+-------------+----------------+--+
| u_002       | 2017/1/2    | 150            |
| u_002       | 2017/1/2    | 70             |
| u_002       | 2017/1/3    | 30             |
| u_002       | 2017/1/3    | 80             |
| u_002       | 2017/1/4    | 150            |
| u_002       | 2017/1/5    | 101            |
| u_005       | 2017/1/2    | 50             |
| u_005       | 2017/1/2    | 80             |
| u_005       | 2017/1/3    | 180            |
| u_005       | 2017/1/4    | 180            |
| u_005       | 2017/1/4    | 10             |
| u_008       | 2017/1/4    | 260            |
| u_008       | 2017/1/5    | 360            |
| u_008       | 2017/1/6    | 160            |
| u_008       | 2017/1/7    | 60             |
| u_008       | 2017/1/7    | 60             |
| u_009       | 2017/1/2    | 70             |
| u_009       | 2017/1/2    | 70             |
| u_009       | 2017/1/3    | 170            |
| u_009       | 2017/1/4    | 270            |
| u_010       | 2017/1/4    | 90             |
| u_010       | 2017/1/4    | 80             |
| u_010       | 2017/1/5    | 90             |
| u_010       | 2017/1/5    | 90             |
| u_010       | 2017/1/6    | 190            |
```

```
| u_010       | 2017/1/7    | 90              |
| u_010       | 2017/1/7    | 90              |
| u_011       | 2017/1/1    | 110             |
| u_011       | 2017/1/2    | 100             |
| u_011       | 2017/1/2    | 100             |
| u_011       | 2017/1/3    | 120             |
| u_013       | 2017/1/2    | 150             |
| u_013       | 2017/1/2    | 50              |
| u_013       | 2017/1/3    | 150             |
| u_013       | 2017/1/4    | 550             |
| u_013       | 2017/1/5    | 350             |
| u_014       | 2017/1/5    | 250             |
| u_014       | 2017/1/6    | 120             |
| u_014       | 2017/1/7    | 270             |
| u_014       | 2017/1/7    | 20              |
+-------------+-------------+-----------------+--+
```

第二种解法：

SQL 流程

（1）按照用户及时间聚合，计算每个人每天的低碳量（2017 年）并给每一条数据打标签（同一个用户不同时间排序）

```
select user_id,data_dt,
sum(low_carbon) low_carbon_sum,
row_number() over(partition by user_id order by data_dt) as rn
from user_low_carbon
where substring(data_dt,1,4)='2017'
group BY user_id,data_dt
having low_carbon_sum>100;t1
```

（2）获取每一条数据时间跟标签之间的差值

```
select
user_id,data_dt,date_sub(to_date(regexp_replace(data_dt,"/","-")),rn) diffDate from t1;
```

（3）按照所获得的差值聚合，得到同一个用户下相同差值的个数

```
select user_id,data_dt,count(*) over(partition by user_id,diffDate) diffDateCount from t2;t3
```

（4）过滤出相同差值个数在 3 及以上的数据

```
select user_id,data_dt from t3 where diffDateCount>=3;
```

（5）与原表 Join 得到最终需要的结果

```
select t5.user_id,t5.data_dt,t5.low_carbon
from user_low_carbon t5
join t4
where t4.user_id = t5.user_id and t4.data_dt = t5.data_dt
order by t5.user_id,t5.data_dt;
```

（6）最终 Sql

```
SELECT
    t5.user_id,
    t5.data_dt,
    t5.low_carbon
FROM
    user_low_carbon t5
JOIN (
    SELECT
        user_id,
        data_dt
    FROM
        (
            SELECT
                user_id,
                data_dt,
                count(*) over (
                    PARTITION BY user_id,
                    diffDate
                ) diffDateCount
            FROM
                (
                    SELECT
                        user_id,
                        data_dt,
                        date_sub(
                             to_date (
                                regexp_replace (data_dt, "/", "-")
                            ),
                            rn
                        ) diffDate
                    FROM
                        (
                            SELECT
                                user_id,
                                data_dt,
                                sum(low_carbon) low_carbon_sum,
                                row_number () over (
                                    PARTITION BY user_id
                                    ORDER BY
                                        data_dt
                                ) AS rn
                            FROM
                                user_low_carbon
                            WHERE
                                substring(data_dt, 1, 4) = '2017'
                            GROUP BY
                                user_id,
                                data_dt
                            HAVING
                                low_carbon_sum > 100
                        ) t1
                ) t2
        ) t3
    WHERE
        diffDateCount >= 3
) t4
WHERE
    t4.user_id = t5.user_id
```

```
AND t4.data_dt = t5.data_dt
ORDER BY
    t5.user_id,
    t5.data_dt;
```

(7) 结果展示

```
+-------------+-------------+----------------+--+
| t5.user_id  | t5.data_dt  | t5.low_carbon  |
+-------------+-------------+----------------+--+
| u_002       | 2017/1/2    | 150            |
| u_002       | 2017/1/2    | 70             |
| u_002       | 2017/1/3    | 30             |
| u_002       | 2017/1/3    | 80             |
| u_002       | 2017/1/4    | 150            |
| u_002       | 2017/1/5    | 101            |
| u_005       | 2017/1/2    | 50             |
| u_005       | 2017/1/2    | 80             |
| u_005       | 2017/1/3    | 180            |
| u_005       | 2017/1/4    | 180            |
| u_005       | 2017/1/4    | 10             |
| u_008       | 2017/1/4    | 260            |
| u_008       | 2017/1/5    | 360            |
| u_008       | 2017/1/6    | 160            |
| u_008       | 2017/1/7    | 60             |
| u_008       | 2017/1/7    | 60             |
| u_009       | 2017/1/2    | 70             |
| u_009       | 2017/1/2    | 70             |
| u_009       | 2017/1/3    | 170            |
| u_009       | 2017/1/4    | 270            |
| u_010       | 2017/1/4    | 90             |
| u_010       | 2017/1/4    | 80             |
| u_010       | 2017/1/5    | 90             |
| u_010       | 2017/1/5    | 90             |
| u_010       | 2017/1/6    | 190            |
| u_010       | 2017/1/7    | 90             |
| u_010       | 2017/1/7    | 90             |
| u_011       | 2017/1/1    | 110            |
| u_011       | 2017/1/2    | 100            |
| u_011       | 2017/1/2    | 100            |
| u_011       | 2017/1/3    | 120            |
| u_013       | 2017/1/2    | 150            |
| u_013       | 2017/1/2    | 50             |
| u_013       | 2017/1/3    | 150            |
| u_013       | 2017/1/4    | 550            |
| u_013       | 2017/1/5    | 350            |
| u_014       | 2017/1/5    | 250            |
| u_014       | 2017/1/6    | 120            |
| u_014       | 2017/1/7    | 270            |
| u_014       | 2017/1/7    | 20             |
+-------------+-------------+----------------+--+
```

7.8.2 求出连续三天有销售记录的店铺

1） 原始数据

```
A,2017-10-11,300
A,2017-10-12,200
```

```
A,2017-10-13,100
A,2017-10-15,100
A,2017-10-16,300
A,2017-10-17,150
A,2017-10-18,340
A,2017-10-19,360
B,2017-10-11,400
B,2017-10-12,200
B,2017-10-15,600
C,2017-10-11,350
C,2017-10-13,250
C,2017-10-14,300
C,2017-10-15,400
C,2017-10-16,200
D,2017-10-13,500
E,2017-10-14,600
E,2017-10-15,500
D,2017-10-14,600
```

2） 分析：给每个用户一个编号,用日期减去编号,如果是同一天,那么就是连续的

```
A,2017-10-11,300,1,2017-10-10
A,2017-10-12,200,2,2017-10-10
A,2017-10-13,100,3,2017-10-10
A,2017-10-15,100,4,2017-10-11
A,2017-10-16,300,5,2017-10-11
A,2017-10-17,150,6,2017-10-11
A,2017-10-18,340,7,2017-10-11
A,2017-10-19,360,8,2017-10-11
B,2017-10-11,400
B,2017-10-12,200
B,2017-10-15,600
C,2017-10-11,350
C,2017-10-13,250
C,2017-10-14,300
C,2017-10-15,400
C,2017-10-16,200
D,2017-10-13,500
E,2017-10-14,600
E,2017-10-15,500
D,2017-10-14,600
```

1:建表，加载数据

```
create table t_jd(shopid string,dt string,sale int)
row format delimited fields terminated by ',';
```

```
0: jdbc:hive2://hadoop01:10000> create table t_jd(shopid string,dt string,sale int)
0: jdbc:hive2://hadoop01:10000> row format delimited fields terminated by ',';
No rows affected (18.94 seconds)
0: jdbc:hive2://hadoop01:10000>
```



```
load data local inpath '/root/sale.dat' into table t_jd;
```

```
0: jdbc:hive2://hadoop01:10000> load data local inpath '/root/sale.dat' into table t_jd;
INFO  : Loading data to table default.t_jd from file:/root/sale.dat
INFO  : Table default.t_jd stats: [numFiles=1, totalSize=340]
No rows affected (11.366 seconds)
0: jdbc:hive2://hadoop01:10000>
```


2:打编号

```
select shopid,dt,sale,
row_number() over(partition by shopid order by dt) as rn
from t_jd;
```

```
0: jdbc:hive2://hadoop01:10000> select shopid,dt,sale,
0: jdbc:hive2://hadoop01:10000> row_number() over(partition by shopid order by dt) as rn
0: jdbc:hive2://hadoop01:10000> from t_jd;
```


结果：

shopid	dt	sale	rn
A	2017-10-11	300	1
A	2017-10-12	200	2
A	2017-10-13	100	3
A	2017-10-15	100	4
A	2017-10-16	300	5
A	2017-10-17	150	6
A	2017-10-18	340	7
A	2017-10-19	360	8
B	2017-10-11	400	1
B	2017-10-12	200	2
B	2017-10-15	600	3
C	2017-10-11	350	1
C	2017-10-13	250	2
C	2017-10-14	300	3
C	2017-10-15	400	4
C	2017-10-16	200	5
D	2017-10-13	500	1
D	2017-10-14	600	2
E	2017-10-14	600	1
E	2017-10-15	500	2

3 根据编号，生成连续日期

```
select shopid,dt,sale,rn,
date_sub(to_date(dt),rn)
from
(select shopid,dt,sale,
row_number() over(partition by shopid order by dt) as rn
from t_jd) tmp;
```

```
0: jdbc:hive2://hadoop01:10000> select shopid,dt,sale,rn,
0: jdbc:hive2://hadoop01:10000> date_sub(to_date(dt),rn)
0: jdbc:hive2://hadoop01:10000> from
0: jdbc:hive2://hadoop01:10000> (select shopid,dt,sale,
0: jdbc:hive2://hadoop01:10000> row_number() over(partition by shopid order by dt) as rn
0: jdbc:hive2://hadoop01:10000> from t_jd) tmp;
```


结果：

shopid	dt	sale	rn	_c4
A	2017-10-11	300	1	2017-10-10
A	2017-10-12	200	2	2017-10-10
A	2017-10-13	100	3	2017-10-10
A	2017-10-15	100	4	2017-10-11
A	2017-10-16	300	5	2017-10-11
A	2017-10-17	150	6	2017-10-11
A	2017-10-18	340	7	2017-10-11
A	2017-10-19	360	8	2017-10-11
B	2017-10-11	400	1	2017-10-10
B	2017-10-12	200	2	2017-10-10
B	2017-10-15	600	3	2017-10-12
C	2017-10-11	350	1	2017-10-10
C	2017-10-13	250	2	2017-10-11
C	2017-10-14	300	3	2017-10-11
C	2017-10-15	400	4	2017-10-11
C	2017-10-16	200	5	2017-10-11
D	2017-10-13	500	1	2017-10-12
D	2017-10-14	600	2	2017-10-12
E	2017-10-14	600	1	2017-10-13
E	2017-10-15	500	2	2017-10-13

4 分组，求 count

```
select shopid,count(1) as cnt
from
(select shopid,dt,sale,rn,
date_sub(to_date(dt),rn) as flag
from
(select shopid,dt,sale,row_number() over(partition by shopid
order by dt) as rn
from t_jd) tmp) tmp2
group by shopid,flag;
```

结果：

```
+---------+------+--+
| shopid  | cnt  |
+---------+------+--+
| A       | 3    |
| A       | 5    |
| B       | 2    |
| B       | 1    |
| C       | 1    |
| C       | 4    |
| D       | 2    |
| E       | 2    |
+---------+------+--+
```


5 筛选出连续天数大于等于 3 的

```
select shopid from
(select shopid,count(1) as cnt
from
(select shopid,dt,sale,rn,
date_sub(to_date(dt),rn) as flag
from
(select shopid,dt,sale,
row_number() over(partition by shopid order by dt) as rn
from t_jd) tmp) tmp2
group by shopid,flag) tmp3
where tmp3.cnt>=3;
```

```
0: jdbc:hive2://hadoop01:10000> select shopid from
0: jdbc:hive2://hadoop01:10000> (select shopid,count(1) as cnt
0: jdbc:hive2://hadoop01:10000> from
0: jdbc:hive2://hadoop01:10000> (select shopid,dt,sale,rn,
0: jdbc:hive2://hadoop01:10000> date_sub(to_date(dt),rn) as flag
0: jdbc:hive2://hadoop01:10000> from
0: jdbc:hive2://hadoop01:10000> (select shopid,dt,sale,
0: jdbc:hive2://hadoop01:10000> row_number() over(partition by shopid order by dt) as rn
0: jdbc:hive2://hadoop01:10000> from t_jd) tmp) tmp2
0: jdbc:hive2://hadoop01:10000> group by shopid,flag) tmp3
0: jdbc:hive2://hadoop01:10000> where tmp3.cnt>=3;
```


结果：

```
+---------+--+
| shopid  |
+---------+--+
| A       |
| A       |
| C       |
+---------+--+
```


6 去重

```
select distinct shopid from
(select shopid,count(1) as cnt
from
(select shopid,dt,sale,rn,
date_sub(to_date(dt),rn) as flag
from
(select shopid,dt,sale,
row_number() over(partition by shopid order by dt) as rn
from t_jd) tmp) tmp2
group by shopid,flag) tmp3
where tmp3.cnt>=3;
```

```
0: jdbc:hive2://hadoop01:10000> select distinct shopid from
0: jdbc:hive2://hadoop01:10000> (select shopid,count(1) as cnt
0: jdbc:hive2://hadoop01:10000> from
0: jdbc:hive2://hadoop01:10000> (select shopid,dt,sale,rn,
0: jdbc:hive2://hadoop01:10000> date_sub(to_date(dt),rn) as flag
0: jdbc:hive2://hadoop01:10000> from
0: jdbc:hive2://hadoop01:10000> (select shopid,dt,sale,
0: jdbc:hive2://hadoop01:10000> row_number() over(partition by shopid order by dt) as rn
0: jdbc:hive2://hadoop01:10000> from t_jd) tmp) tmp2
0: jdbc:hive2://hadoop01:10000> group by shopid,flag) tmp3
0: jdbc:hive2://hadoop01:10000> where tmp3.cnt>=3;
```

结果：

```
+---------+---+
| shopid  |
+---------+---+
| A       |
| C       |
+---------+---+
```

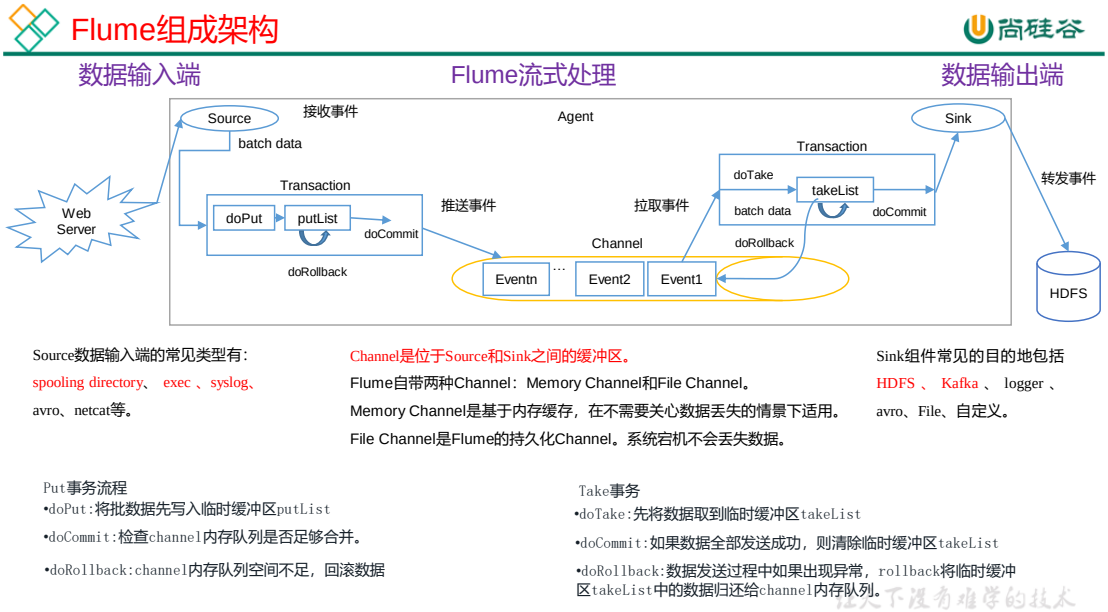

八、Flume

8.1 Flume 组成，Put 事务，Take 事务

Flume 三大组件为：Agent、Channel、Sink

Flume 的传输单元为 Event，Event 里面有 Header+ Body ，Header 为可选消息头。

Agent 与 Channel 之间为 Put 事务、Channel 与 Sink 之间为 Take 事务



8.2 拦截器

自定义拦截器主要目的是把日志分类，拦截器为每个 Event 加上一个 Header，这 Header 就是日志的类型，当日志传输到 Kafka 时候，通过 Key(Header)便知道这属于哪种类型的。

自定义拦截器的操作：

1）项目 Pom 文件添加 Flume 依赖

2）寻找 Flume 拦截器类 TimestampInterceptor（系统时间拦截器） 把这个类的代码 copy 到自定义类，然后按需求修改，打成 jar 包修改 jar 包名称为 app_logs_flume.jar 然后放到/opt/module/flume/lib 目录下 。

3）在 Flume 的配置文件中指定拦截器的类型

a1.sources.r1.interceptors = i1　a1.sources.r1.interceptors.i1.type = 自定义类全类名

8.3 监控器 Ganglia

字段（图表名称）	字段含义
EventPutAttemptCount	source 尝试写入 channel 的事件总数量
EventPutSuccessCount	成功写入 channel 且提交的事件总数量
EventTakeAttemptCount	sink 尝试从 channel 拉取事件的总数量。这不意味着每次事件都被返回，因为 sink 拉取的时候 channel 可能没有任何数据。
EventTakeSuccessCount	sink 成功读取的事件的总数量
StartTime	channel 启动的时间（毫秒）
StopTime	channel 停止的时间（毫秒）
ChannelSize	目前 channel 中事件的总数量
ChannelFillPercentage	channel 占用百分比
ChannelCapacity	channel 的容量

主要简单记一下 Ganglia 里面能看到什么

8.4 自定义 MySqlSource，实时读 MySql 数据

如果需要实时监控 MySQL，从 MySQL 中获取数据传输到 HDFS 或者其他存储框架，所以此时需要我们自己实现 MySQLSource。

根据官方说明自定义 MySqlSource 需要继承 AbstractSource 类并实现 Configurable 和 PollableSource 接口，然后根据业务编写逻辑代码。

官网说明：https://flume.apache.org/FlumeDeveloperGuide.html#source

8.5 为什么使用双层 Flume

实现负载均衡和容灾

第一层 flume 地实现数据采集，第二层做数据的聚合，并通过 Flume 拓扑结构中的 Sinkgroup 实现了 Load Balance，充分利用了资源，当第二层的 Flume 故障超过 30s 后自动切换组内 Sink。

8.6 Flume 源码修改

如果使用 0.8 版本 Kafka 并配套 1.6 版本 Flume，由于 Flume 1.6 版本没有 Taildir Source 组件，因此，需要将 Flume 1.7 中的 Taildir Source 组件源码编译打包后，放入

Flume1.6 安装目录的 lib 文件目录下

在 TaildirSource 的源码中，通过 inode 和文件名来进行唯一文件的判定，也就是说，一旦文件改名，那么 TaildirSource 会将它当成新文件而重新读取，这是不合理的，会造成数据的重复读取。

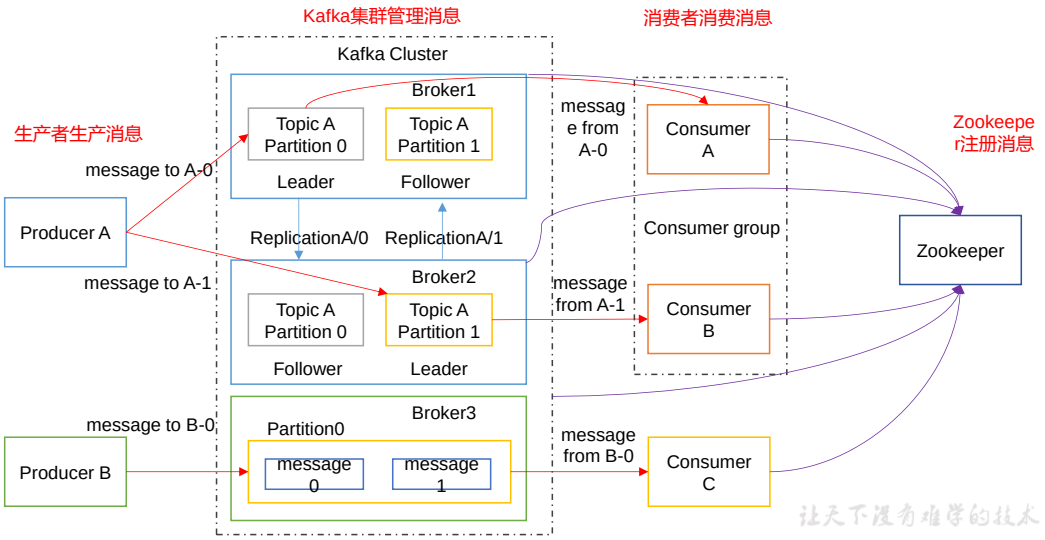
为了解决这个问题，我们对 ReliableTaildirEventReader.java 文件进行修改，具体是对此文件中的 updateTailFiles 函数和 loadPositionFile 函数进行修改。

```
//updateTailFiles
//重命名后，Map 中的文件名还是老的文件名，因此使用 openFIle 重新创建 TailFile 用来替换原数据

       if (!tf.getPath().equtAbsolutePath())) {

          tf = openFile(f, hede, tf.getPos());

        }

        tf.setNeedTail(update


//loadPositionFile

了 updatePos 能够顺利更新 pos，应该传入 tf.getPath()，即新文件名，tailfile 与 tailfile 自身的文件名的比较必然是相等的

      if (tf != null && tf.Pos(tf.getPath(), inode, pos)) {

        tailFiles.pde, tf);

      } else {

        logger.info("Missing + path + ", inode: " + inode + "," + pos);

    }
```

九、Kafka

9.1 组成

生产者不需要在集群，只需要记住消费过程



9.2 几个分区

10 个分区，10 个消费者

9.3 几个副本

2 个

9.4 Kafka 丢不丢数据

1）常见 Kafka 丢数据情况：

（1）如果 auto.commit.enable=true，当 consumer fech 了一数据但还没有完全处理 1.掉的时候，刚好到 commit interval 出发了提交 offset 作，接着 consumer crash 掉了。这时已经 etch 的数据还没有处理完成但已经被 commit 掉，因此没有机再次被处理，数据丢失。

（2）网络负载很高或者磁盘很忙写入失败的情况下，没有自动重试重发消息。没有做限速处理，超出了网络带宽限速。kafk2.a 一定要配置上消息重试的机制，并且重试的时间间隔一定要长一些，默认 1 秒钟符合生产环境（网络中断时间有可能超过 1 秒）。

（3）如果磁盘坏了，会丢失已经落盘的数据

（4）单批数据的长度超过限制会丢失数据，报 kafka.common.Mess3.ageSizeTooLargeExce4.ption 异常

2）解决

（1）partition leader 在未副本数 follows 的备份时就宕机的情况，即使选举出了新的 leade1.r 但是已经 push 的数据因为未备份就丢失了！

（2）kafka 是多副本的你配置了同步复制之后。多个副本的数据都在 PageCache 里面，

出现多个副本同时挂掉 2.的概率比 1 个副本挂掉概率就很小了。（官方推荐是通过副本来保证数据的完整）

（3）kafka 的数据一开始就是存储在 PageCache 上的，定期 flush 到磁盘上的，也就是说，不是每个消息都被 3.存储在磁盘了，如果出现断电或者机器故障等，PageCache 上的数据就丢。

（4）可以通过 log.flush.interval.messages 和 log.flush.interval.ms 来 4.配置 flush 间隔，interval 大丢的数据多些，小会影响性能但在 0.本，可以通过 replica 机制保证数据不丢，代价就是需要更多资源，尤其是磁盘资源，kafka 当前支持 GZip 和 Snappy 压缩，来缓解这个问题 是否使用 replica 取决于在可靠性和资源代价之间的 balance

9.5 Kafka 保存数据持久化，默认保存多久

保存 7 天，可以自己设置

9.6 offset 在 Zookeeper 存在什么路径下

---->consumer--

9.7 多少个 Topic

多少个日志类型就多少个 Topic

9.8 Kafka 高级 API，低级 API

9.9 数据量少

10g，1000 万条，每分钟 1000 万/24/60=0.69 万条，
平均每分钟：6900 条
低谷每分钟：2000 条
高峰每分钟：6900 条*（2-10 倍）=14 万条-70 万条

每平均秒钟：115 条
每条日志大小：0.5k-1k
每秒多少数据量：57.5k-115k

9.10 Kafka 挂掉

- Flume 记录
- 日志有记录
- 短期

9.11 isa

ISR（In-Sync Replicas），副本同步队列。

ISR 中包括 leader 和 follower。副本数对 Kafka 的吞吐率是有一定的影响，但极大的增强了可用性。默认情况下 Kafka 的 replica 数量为 1，即每个 partition 都有一个唯一的 leader，为了确保消息的可靠性。

通常应用中将其值(由 broker 的参数 offsets.topic.replication.factor 指定)大小设置为大于 1，比如 3。 所有的副本（replicas）统称为 Assigned Replicas，即 AR。ISR 是 AR 中的一个子集，由 leader 维护 ISR 列表，follower 从 leader 同步数据有一些延迟（包括延迟时间 replica.lag.time.max.ms 和延迟条数

replica.lag.max.messages 两个维度, 当前的 0.10.x 及以上版本中只支持

replica.lag.time.max.ms 这个维度），任意一个超过阈值都会把 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。

AR=ISR+OSR

9.12 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。当以下事件发生时，Kafka 将会进行一次分区分配：

- 同一个 Consumer Group 内新增消费者
- 消费者离开当前所属的 Consumer Group，包括 shuts down 或 crashes
- 订阅的主题新增分区

将分区的所有权从一个消费者移到另一个消费者称为重新平衡（rebalance），如何 rebalance 就涉及到下面提到的分区分配策略。下面我们将详细介绍 Kafka 内置的两种分区分配策略。本文假设我们有个名为 T1 的主题，其包含了 10 个分区，然后我们有两个消费者（C1，C2）来消费这 10 个分区里面的数据，而且 C1 的 num.streams = 1，C2 的 num.streams = 2。

• Range strategy

Range 策略是对每个主题而言的，首先对同一个主题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。在我们的例子里面，排完序的分区将会是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9；消费者线程排完序将会是 C1-0, C2-0, C2-1。然后将 partitions 的个数除于消费

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。

在我们的例子里面，我们有 10 个分区，3 个消费者线程， 10 / 3 = 3，而且除不尽，那么消费者线程 C1-0 将会多消费一个分区，所以最后分区分配的结果看起来是这样的：

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

假如我们有 11 个分区，那么最后分区分配的结果看起来是这样的：

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6, 7 分区

C2-1 将消费 8, 9, 10 分区

假如我们有 2 个主题(T1 和 T2)，分别有 10 个分区，那么最后分区分配的结果看起来是这样的：

C1-0 将消费 T1 主题的 0, 1, 2, 3 分区以及 T2 主题的 0, 1, 2, 3 分区

C2-0 将消费 T1 主题的 4, 5, 6 分区以及 T2 主题的 4, 5, 6 分区

C2-1 将消费 T1 主题的 7, 8, 9 分区以及 T2 主题的 7, 8, 9 分区

可以看出，C1-0 消费者线程比其他消费者线程多消费了 2 个分区，这就是 Range strategy 的一个很明显的弊端。

• RoundRobin strategy

使用 RoundRobin 策略有两个前提条件必须满足：

- 同一个 Consumer Group 里面的所有消费者的 num.streams 必须相等；
- 每个消费者订阅的主题必须相同。

所以这里假设前面提到的 2 个消费者的 num.streams = 2。RoundRobin 策略的工作原理：将所有主题的分区组成 TopicAndPartition 列表，然后对 TopicAndPartition 列表按照 hashCode 进行排序

最后按照 round-robin 风格将分区分别分配给不同的消费者线程。

在我们的例子里面，假如按照 hashCode 排序完的 topic-partitions 组依次为 T1-5, T1-3, T1-0, T1-8, T1-2, T1-1, T1-4, T1-7, T1-6, T1-9，我们的消费者线程排序为 C1-0, C1-1, C2-0, C2-1，最后分区分配的结果为：

C1-0 将消费 T1-5, T1-2, T1-6 分区；

C1-1 将消费 T1-3, T1-1, T1-9 分区；

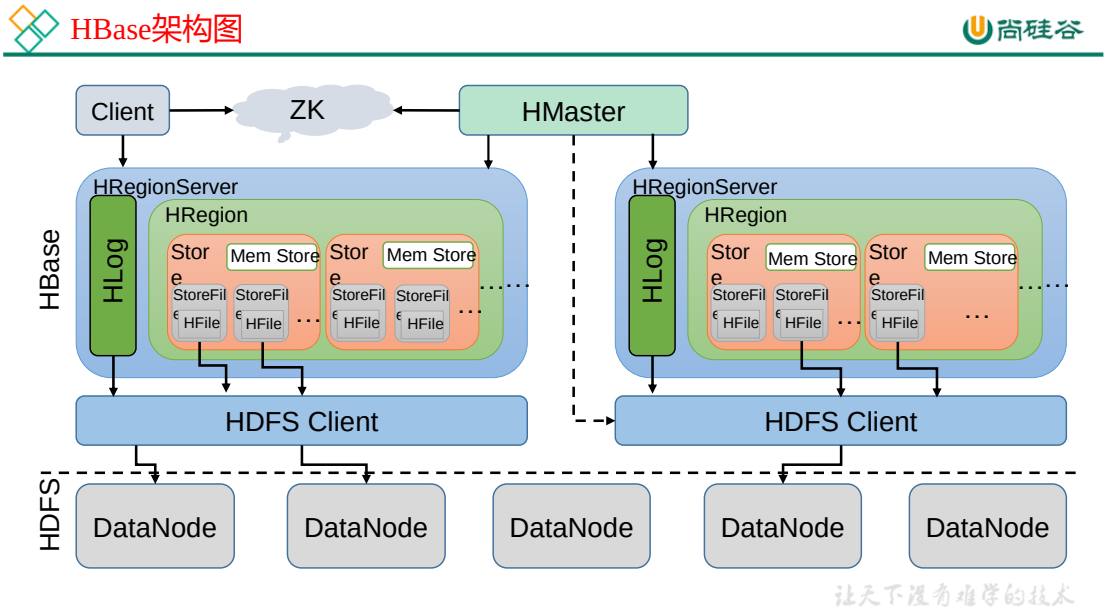
C2-0 将消费 T1-0, T1-4 分区；

C2-1 将消费 T1-8, T1-7 分区；

9.13 Kafka Monitor, Kafka Manager

十、HBase

10.1 Hbase 存储结构



10.2 rowkey 设计原则

1）rowkey 长度原则

rowkey 是一个二进制码流，可以为任意字符串，最大长度为 64kb，实际应用中一般为 10-100bytes，它以 byte[]形式保存，一般设定成定长。

一般越短越好，不要超过 16 个字节，注意原因如下：

1、目前操作系统都是 64 位系统，内存 8 字节对齐，控制在 16 字节，8 字节的整数倍利用了操作系统的最佳特性。

2、hbase 将部分数据加载到内存当中，如果 rowkey 过长，内存的有效利用率就会下降。

2）rowkey 散列原则

如果 rowkey 按照时间戳的方式递增，不要将时间放在二进制码的前面，建议将 rowkey 的高位字节采用散列字段处理，由程序随即生成。低位放时间字段，这样将提高数据均衡分布，各个 regionServer 负载均衡的几率。

如果不进行散列处理，首字段直接使用时间信息，所有该时段的数据都将集中到一个 regionServer 当中，这样当检索数据时，负载会集中到个别 regionServer 上，造成热点问题，会降低查询效率。

3）rowkey 唯一原则

必须在设计上保证其唯一性，rowkey 是按照字典顺序排序存储的，因此，设计 rowkey 的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。但是这里的量不能太大，如果太大需要拆分到多个节点上去。

10.3 RowKey 如何设计

一条数据的唯一标识就是 rowkey，那么这条数据存储于哪个分区，取决于 rowkey 处度上防止数据倾斜。接下来我们就谈一谈 rowkey 常用的设计方法。

(1）生成随机数、hash、散列值

比如：

原本 rowKey 为 1001 的，SHA1 后变成：dd01903921ea24941c26a48f2cec24e0bb0e8cc7

原本 rowKey 为 3001 的，SHA1 后变成：49042c54de64a1e9bf0b33e00245660ef92dc7bd

原本 rowKey 为 501 的，SA1 后变成：7b61dec07e02c188790670af43e717f0f46e8913

在做此操作之前，一般我们会选择从据集中抽取样本，来决定什么样的 rowKey 来 Hash 后作为每个分区的临界值。

（2）字符串反转现在，假定需要查询满足条件 q1=0 and q2=02 的 Sample 记录，分析查询字段和索引匹配情况可知应使用索引 a，也就是说我们首先确定了索引名，于是在 Region 1 上进行 scan 的区间将从主数据全集收窄至[0000-a, 0000-b)，接着拼接查询字段的值，我们得到了索引键：0102，scan 区间又进一步收窄为[0000-a-0102, 0000-a-0103)，于是我们可以很快地找到 0000-a-0102-0000|63af51b2 这条索引，进而得到了索引值，也就是目标数据的 RowKey：0000|63af51b2，通过在 Region 内执行 Get 操作，最终得到了目标数据。

需要特别说明的是这个 Get 操作是在本 Region 上执行的，这和通过 HTable 发出的 Get 有很大的不同，它专门用于获取 Region 的本地数据，其执行效率是非常高的，这也是为什么我们一定要将索引和它的主数据放在同一张表的同一个 Region 上的原因。

d q2=02 的 Sample 记录，分析查询字段和索引匹配情况可知应使用索引 a，也就是说我们首先确定了索引名，于是在 Region 1 上进行 scan 的区间将从主数据全集收窄至[0000-a, 0000-b)，接着拼接查询字段的值，我们得到了索引键：0102，scan 区间又进一步收窄为[0000-a-0102, 0000-a-0103)，于是我们可以很快地找到 0000-a-0102-0000|63af51b2 这条索引，进而得到了索引值，也就是目标数据的 RowKey：0000|63af51b2，通过在 Region 内执行 Get 操作，最终得到了目标数据。需要特别说明的是这个 Get 操作是在本 Region 上执行的，这和通过 HTable 发出的 Get 有很大的不同，它专门用于获取 Region 的本地数据，其执行效率是非常高的，这也是为什么我们一定要将索引和它的主数据放在同一张表的同一个 Region 上的原因。

10.4 二级索引

10.5 Sqoop

从哪导导哪去

- MySql 导入到 HDFS
- MySql 导入到 Hive
- HDFS 导入到 MySql
- Hive 导入到 MySql

实际生产环境正常一天跑 500-1000 个 Task

十一、Scala

11.1 元组

1）元组的创建

```
val tuple1 = (1, 2, 3, "heiheihei")
println(tuple1)
```

2）元组数据的访问，注意元组元素的访问有下划线，并且访问下标从 1 开始，而不是 0

```
val value1 = tuple1._4
println(value1)
```

3）元组的遍历

```
方式 1：
for (elem <- tuple1.productIterator  ) {
   print(elem)
}
println()
方式 2：
tuple1.productIterator.foreach(i => println(i))
tuple1.produIterator.foreach(print(_))
```

11.2 隐私转换

隐式转换函数是以 implicit 关键字声明的带有单个参数的函数。这种函数将会自动应用，将值从一种类型转换为另一种类型。

```
implicit def a(d: Double) = d.toInt
//不加上边这句你试试
val i1: Int = 3.5
println(i1)
```

11.3 函数式编程理解

1、Scala 中函数的地位：一等公民

2、Scala 中的匿名函数(函数字面量)

3、Scala 中的高阶函数

4、Scala 中的闭包

5、Scala 中的部分应用函数

6、Scala 中的柯里化函数

11.4 样例类

case class Person(name:String,age:Int)

一 般使用在 ds=df.as[Person]

11.5 柯里化

函数编程中，接受多个参数的函数都可以转化为接受单个参数的函数，这个转化过程就叫柯里化，柯里化就是证明了函数只需要一个参数而已。其实我们刚的学习过程中，已经涉及到了柯里化操作，所以这也印证了，柯里化就是以函数为主体这种思想发展的必然产生的结果。

（1）柯里化的示例

```
def mul(x: Int, y: Int) = x * y
println(mul(10, 10))
def mulCurry(x: Int) = (y: Int) => x * y
println(mulCurry(10)(9))
def mulCurry2(x: Int)(y:Int) = x * y
println(mulCurry2(10)(8))
```

（2）柯里化的应用

比较两个字符串在忽略大小写的情况下是否相等，注意，这里是两个任务：

- 全部转大写（或小写）
- 比较是否相等

针对这两个操作，我们用一个函数去处理的思想，其实无意间也变成了两个函数处理的思想。示例如下：

```
val a = Array("Hello", "World")
val b = Array("hello", "world")
println(a.corresponds(b)(_.equalsIgnoreCase(_)))
其中 corresponds 函数的源码如下：
def corresponds[B](that: GenSeq[B])(p: (A,B) => Boolean): Boolean = {

val i = this.iterator

val j = that.iterator

while (i.hasNext && j.hasNex  t)

  if (!p(i.next(), j.next()    ))

    return fals
!i.hasNext && !j.hasNext
}
```

尖叫提示：不要设立柯里化存在义这样的命题，柯里化，是面向函数思想的必然产生结果。

11.6 闭包

一个函数把外部的那些不属于自己的对象也包含(闭合)进来。

案例 1：

```
def minusxy(x: Int) = (y: Int) => x - y
这就是一个闭包：
1) 匿名函数(y: Int) => x -y 嵌套在 minusxy 函数中。
2) 匿名函数(y: Int) => x -y 使用了该匿名函数之外的变量 x
3) 函数 minusxy 返回了引用了局部变量的匿名函数
```

案例 2

```
def minusxy(x: Int) = (y: Int) => x - y
val f1 = minusxy(10)
val f2 = minusxy(10)
println(f1(3) + f2(3))
此处 f1,f2 这两个函数就叫闭包。
```

十二、Spark:阶段考试题

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

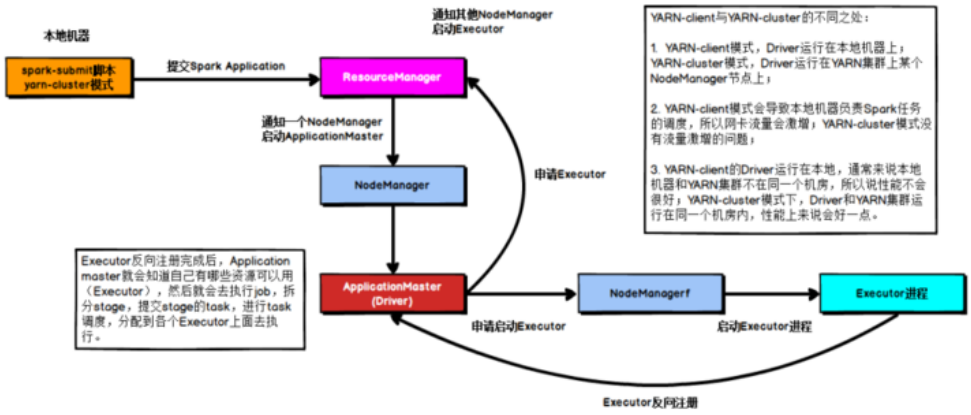


图 5-1 YARN-Cluster 任务提交流程

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

当 shuffle read task 的数量小于等于 spark.shuffle.sort.

bypassMergeThreshold 参数的值时（默认为 200），就会启用 bypass 机制。

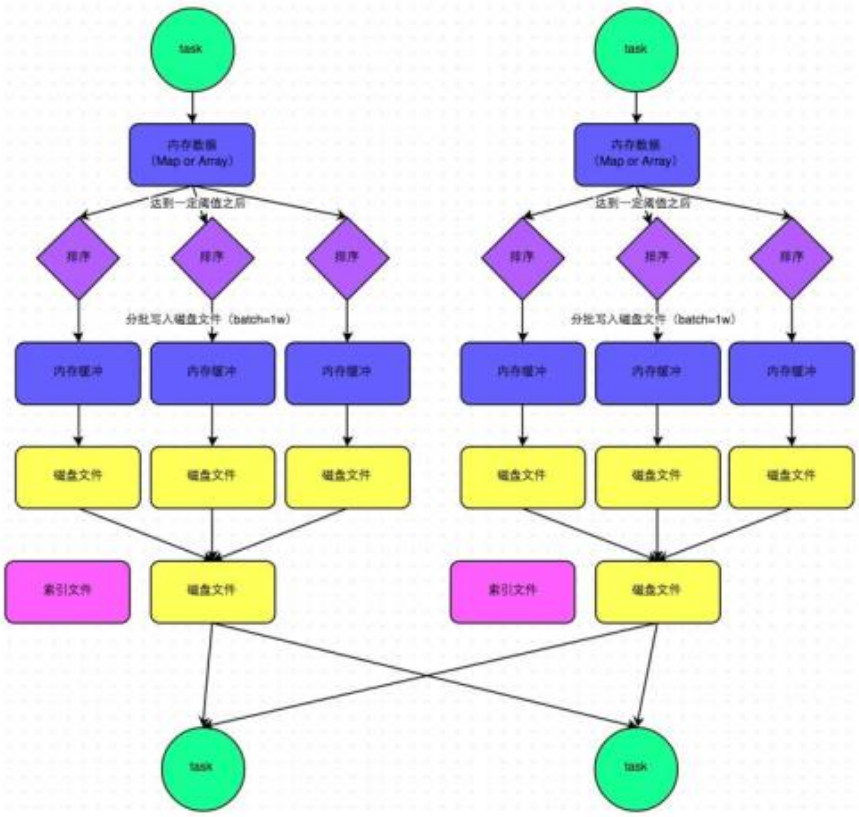


图 6-4 普通运行机制的 SortShuffleManager 工作原理

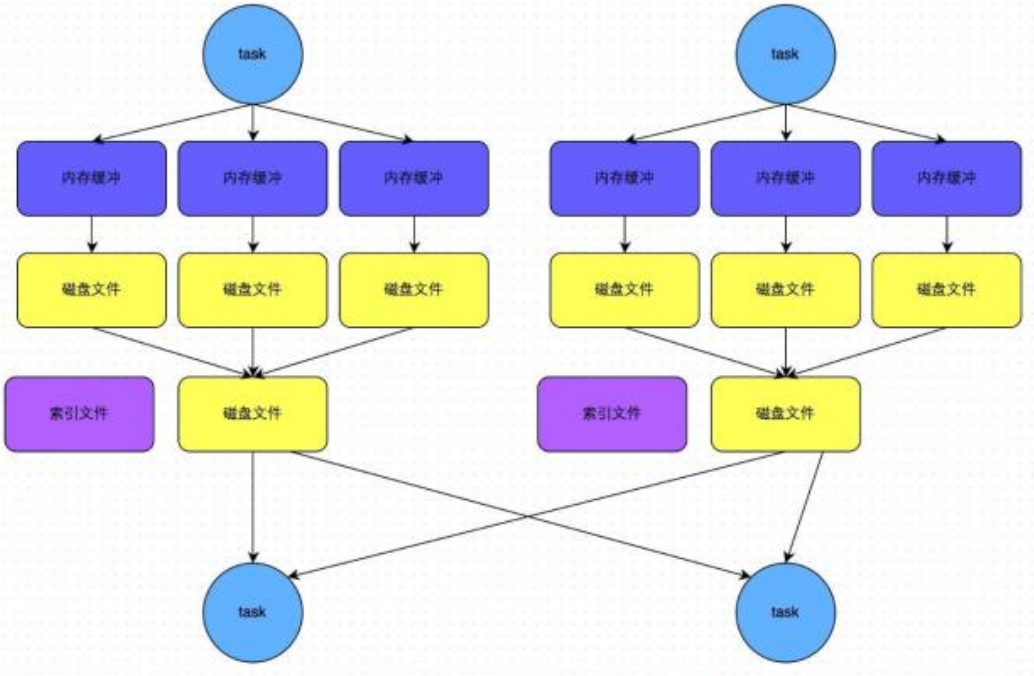


图 6-5 bypass 运行机制的 SortShuffleManager 工作原理

12.3 Spark 有几种部署方式？请分别简要论述

1）Local:运行在一台机器上，通常是练手或者测试环境。

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

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

4）Mesos：国内大环境比较少用。

12.4 如何理解 Spark 中的血统概念（RDD）？

RDD 在 Lineage 依赖方面分为两种 Narrow Dependencies 与 Wide Dependencies 用来解决数据容错时的高效性。

Narrow Dependencies 是指父 RDD 的每一个分区最多被一个子 RDD 的分区所用，表现为一个父 RDD 的分区对应于一个子 RDD 的分区或多个父 RDD 的分区对应于一个子 RDD 的分区，也就是说一个父 RDD 的一个分区不可能对应一个子 RDD 的多个分区。

Wide Dependencies 是指子 RDD 的分区依赖于父 RDD 的多个分区或所有分区，也就是说存在一个父 RDD 的一个分区对应一个子 RDD 的多个分区。对与 Wide Dependencies，这种计算的输入和输出在不同的节点上，lineage 方法对与输入节点完好，而输出节点宕机时，通过重新计算，这种情况下，这种方法容错是有效的，否则无效，因为无法重试，需要向上其祖先追溯看是否可以重试（这就是 lineage，血统的意思），Narrow Dependencies 对于数据的重算开销要远小于 Wide Dependencies 的数据重算开销。

12.5 简述 Spark 的宽窄依赖，以及 Spark 如何划分 stage，每个 stage 又根据什么决定 task 个数?

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

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

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

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

（2）mapPartitions(func)：类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]。假设有 N 个

元素，有 M 个分区，那么 map 的函数的将被调用 N 次,而 mapPartitions 被调用 M 次,一个函数一次处理所有分区。

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

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

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

对相同 K，把 V 合并成一个集合。

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

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

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

…

根据自身情况选择比较熟悉的算子加以介绍。

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

（1）reduce：

（2）collect:

（3）first：

（4）take：

（5）aggregate：

（6）countByKey：

（7）foreach：

（8）saveAsTextFile：

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

reduceBykey：

groupByKey：

…ByKey:

12.9 Spark 常用算子 reduceByKey 与 groupByKey 的区别，哪一种更具优势？

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

groupByKey：按照 key 进行分组，直接进行 shuffle。

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

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

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

共享变量出现的原因：

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

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

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

都是做 RDD 持久化的

cache()与 persist()：

会被重复使用的(但是)不能太大的 RDD 需要 cache。cache 只使用 memory，写磁盘的话那就叫 checkpoint 了。 哪些 RDD 需要 checkpoint？运算时间很长或运算量太大才能得到的 RDD，computing chain 过长或依赖其他 RDD 很多的 RDD。 实际上，将 ShuffleMapTask 的输出结果存放到本地磁盘也算是 checkpoint，只不过这个 checkpoint 的主要目的是去 partition 输出数据。

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。 也就是说需要 checkpoint 的 RDD 会被计算两次。因此，在使用 rdd.checkpoint() 的时候，建议加上 rdd.cache()， 这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。

persist()与 checkpoint()

深入一点讨论，rdd.persist(StorageLevel.DISK_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）

而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉（ 话说怎么 remove checkpoint 过的 RDD？ ），是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。

同时：cache 跟 persist 不会截断血缘关系，checkPoint 会截断血缘关系

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

12.13 简述 SparkSQL 中 RDD、DataFrame、DataSet 三者的区别与联系?

1）RDD

优点:

编译时类型安全

编译时就能检查出类型错误

面向对象的编程风格

直接通过类名点的方式来操作数据

缺点:

序列化和反序列化的性能开销

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

GC 的性能开销 ， 频繁的创建和销毁对象, 势必会增加 GC

2）DataFrame

DataFrame 引入了 schema 和 off-heap

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

off-heap : 意味着 JVM 堆以外的内存, 这些内存直接受操作系统管理（而不是 JVM）。Spark 能够以二进制的形式序列化数据(不包括结构)到 off-heap 中， 当要操作数据时，就直接操作 off-heap 内存。由于 Spark 理解 schema，所以知道该如何操作。

off-heap 就像地盘，schema 就像地图，Spark 有地图又有自己地盘了，就可以自己说了算了，不再受 JVM 的限制，也就不再收 GC 的困扰了。

通过 schema 和 off-heap，DataFrame 解决了 RDD 的缺点，但是却丢了 RDD 的优点。DataFrame 不是类型安全的，API 也不是面向对象风格的。

3）DataSet

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

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

4）RDD 和 DataSet

DataSet 以 Catalyst 逻辑执行计划表示，并且数据以编码的二进制形式被存储，不需要反序列化就可以执行 sorting、shuffle 等操作。

DataSet 创立需要一个显式的 Encoder，把对象序列化为二进制，可以把对象的 scheme 映射为 Spark SQL 类型，然而 RDD 依赖于运行时反射机制。

DataSet 比 RDD 性能要好很多。

5）DataFrame 和 DataSet

Dataset 可以认为是 DataFrame 的一个特例，主要区别是 Dataset 每一个 record 存储的是一个强类型值而不是一个 Row。因此具有如下三个特点：

- DataSet 可以在编译时检查类型
- DataSet 是面向对象的编程接口。
- 后面版本 DataFrame 会继承 DataSet，DataFrame 是面向 Spark SQL 的接口。

DataFrame 和 DataSet 可以相互转化，df.as[ElementType]这样可以把 DataFrame 转化为 DataSet，ds.toDF()这样可以把 DataSet 转化为 DataFrame。

12.14 SparkSQL 中 join 操作与 left join 操作的区别？

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

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

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

一、基于 Receiver 的方式

这种方式使用 Receiver 来获取数据。Receiver 是使用 Kafka 的高层次 Consumer API 来实现的。receiver 从 Kafka 中获取的数据都是存储在 Spark Executor 的内存中的（如果突然

数据暴增，大量 batch 堆积，很容易出现内存溢出的问题），然后 Spark Streaming 启动的 job 会去处理那些数据。

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

需要注意的要点

Kafka 中的 topic 的 partition，与 Spark 中的 RDD 的 partition 是没有关系的。所以，在 1、KafkaUtils.createStream()中，提高 partition 的数量，只会增加一个 Receiver 中，读取 partition 的线程的数量。不会增加 Spark 处理数据的并行度。

可以创建多个 Kafka 输入 DStream，使用不同的 consumer group 和 topic，来通过多个 receiver 并行接收数据。

如果基于容错的文件系统，比如 HDFS，启用了预写日志机制，接收到的数据都会被复制一份到预写日志中。因此，在 KafkaUtils.createStream()中，设置的持久化级别是 StorageLevel.MEMORY_AND_DISK_SER。

二、基于 Direct 的方式

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

优点如下：

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

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

一次且仅一次的事务机制。

三、对比:

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

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

在实际生产环境中大都用 Direct 方式

12.16 SparkStreaming 读取 Kafka 中数据时，如何有效的对 offset 进行手动维护？

在 spark streaming 读取 kafka 的数据中，spark streaming 提供了两个接口读取 kafka 中的数据，分别是 KafkaUtils.createDstream，KafkaUtils.createDirectStream，

KafkaUtils.createDstream 自动把 offset 更新到 zk 中，默认会丢数据，效率低，后者不会经过 zk，效率更高，需要自己手动维护 offse，通过维护护 offset 写到 zk 中，保障数据零丢失，只处理一次，**KafkaUtils.createDirectStream** 我把 zk 的端口改成了 9999，防止和 kakfa 自带的 zk 的端口产生冲突，下面我写了一些测试代码，经自己测试数据没任何问题，即使 spark streaming 挂了，另一方往 topic 中写数据，下次启动 streaming 程序也能读取，做到数据零丢失，不同的 group.id 下只读取一次。

12.17 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。

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

12.18 请手写出 wordcount 的 Spark 代码实现（Scala）

```
val conf: SparkConf = new SparkConf().setMaster("local[*]").setAppName("WordCount")

val sc = new SparkContext(conf)

sc.textFile("/input")
.flatMap(_.split(" "))
.map((_,1))
.reduceByKey(_+_)
.saveAsTextFile("/output")
sc.stop()
```

12.19 如何使用 Spark 实现 topN 的获取（描述思路或使用伪代码）

方法 1：

（1）按照 key 对数据进行聚合（groupByKey）

（2）将 value 转换为数组，利用 scala 的 sortBy 或者 sortWith 进行排序（mapValues）

数据量太大，会 OOM。

方法 2：

（1）取出所有的 key

（2）对 key 进行迭代，每次取出一个 key 利用 spark 的排序算子进行排序

方法 3：

（1）自定义分区器，按照 key 进行分区，使不同的 key 进到不同的分区

（2）对每个分区运用 spark 的排序算子进行排序

12.20 Spark 提交作业参数

参考答案：

https://blog.csdn.net/gamer_gyt/article/details/79135118

1）在提交任务时的几个重要参数

executor-cores —— 每个 executor 使用的内核数，默认为 1

num-executors —— 启动 executors 的数量，默认为 2

executor-memory —— executor 内存大小，默认 1G

driver-cores —— driver 使用内核数，默认为 1

driver-memory —— driver 内存大小，默认 512M

2）边给一个提交任务的样式

```
spark-submit \
  --master local[5]  \
  --driver-cores 2   \
  --driver-memory 8g \
  --executor-cores 4 \
  --num-executors 10 \
  --executor-memory 8g \
  --class PackageName.ClassName XXXX.jar \
  --name "Spark Job Name" \
  InputPath      \
  OutputPath
```

十三、JavaSE

13.1 hashMap 底层源码，数据结构

底层结构：jdk7:数组+链表 jdk8：数组+链表+红黑树

HashMap 中维护了 Node 类型的数组 table,初始为 null

1）创建对象时，将加载因子 loadFactor 初始化为 0.75,其他成员保持默认值

2）添加元素时，相当于 putVal 方法,需要先将元素的 key 哈希值获取出来，并且运算得出在数组中存放索引。

如果该索引处没有其他元素，则可以直接存放

如果该索引处有其他元素，则需要先判断是否相等，

如果相等，则覆盖

如果不相等，则继续判断，是否为树结构或链表结构，根据不同结构进行不同处理

3）如果需要扩容，则进行对应的扩容。

如果第一次添加，则扩容 table 的 capacity 为 16，临界值 threshold 为 12.

如果其他次扩容，则扩容 table 的 capacity 为 2 倍，临界值 threshold 为 2 倍.

4）当链表中节点数>=7 && capacity>=64 则将链表变成树结构

jdk7 和 jdk8 的对比：

	结构	table 数据类型	初始容量
jdk7	数组+链表	Entry	16
jdk8	数组+链表+红黑数	Node	0

jdk7 中创建对象时，则初始化 table 容量为 16（饿汉式）

jdk8 中创建对象时，并没有初始化，而是第一次添加元素初始化 table 容量为 16（懒汉式）

13.2 java 自带有哪几种线程池

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。

13.3 HashMap 和 HashTable 区别

HashMap:线程不安全的，key 和 value 可以是 null

Hashtable:线程安全的，key 和 values 不可以是 null，否则会报空指针异常

13.4 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。

13.5 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

13.6 Final,Finally,Finalize

final：修饰符（关键字）有三种用法：如果一个类被声明为 final，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。将变量声明为 final，可以保证它们在使用中不被改变，被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为 final 的方法也同样只能使用，不能在子类中被重写。

finally：通常放在 try…catch 的后面构造总是执行代码块，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。

13.7 ==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

十四、Redis

14.1 缓存穿透

1）缓存系统定义：

按照 KEY 去查询 VALUE,当 KEY 对应的 VALUE 一定不存在的时候并对 KEY 并发请求量很大的时候，就会对后端造成很大的压力。（查询一个必然不存在的数据。比如文章表，查询一个不存在的 id，每次都会访问 DB，如果有人恶意破坏，很可能直接对 DB 造成影响。）

由于缓存不命中，每次都要查询持久层。从而失去缓存的意义。

2）解决方法：

（1）缓存层缓 存空值。

缓存太多空值，占用更多空间。（优化：给个空值过 期时间）

存储层更新代码了，缓存层还是空值。（优化：后台设置时主动删除空值，并缓存把值进去）

（2）将数据库中所有的查询条件，放布隆过滤器中。当一个查询请求来临的时候，先经过布隆过滤器进行查，如果请求存在这个条件中，那么继续执行，如果不在，直接丢弃。

3）备注 ：

比如数据库中有 10000 个条件，那么布隆过滤器的容量 size 设置的要稍微比 10000 大一些，比如 120 00.

对于误判率的设置，根据实际项目，以及硬件设施来具体定。但一定不能设置为 0，并且误判率设置的越小，哈希函数跟数组长度都会更多跟更长，那么对硬件，内存中间的要求就会相应 的高

private st atic BloomFilter<Inte ger> bloomFi lt er = BloomFilter.create(Funnels.integerFue l(), s ize, 000 01) ；

有了 siz 跟误判率，那么布隆过滤器会产相应的哈希函数跟数组。

综上：我们可以利用布隆过滤器，将 redis 缓存击穿制在一个可容的范围内。

14.2 哨兵模式

如果 Master 异常，则会进行 Master-Slave 切换，将其中一 Slae 作为 Master，将之前的 Master 作为 Slave

下线：

①主观下线：Subjectively Down，简称 SDOWN，指的是当前 Sentinel 实例对某个 redis 服务器做出的下线判断。

②客观下线：Objectively Down， 简称 ODOWN，指的是多个 Sentinel 实例在对 Master Server 做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的 Master Server 下线判断，然后开启 failover.

工作原理：

①每个 Sentinel 以每秒钟一次的频率向它所知的 Master，Slave 以及其他 Sentinel 实例发送一个 PING 命令；

②如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线；

③如果一个 Master 被标记为主观下线，则正在监视这个 Master 的所有 Sentinel 要以每秒一次的频率确认 Master 的确进入了主观下线状态；

④当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认 Master 的确进入了主观下线状态， 则 Master 会被标记为客观下线；

⑤在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有 Master，Slave 发送 INFO 命令

⑥当 Master 被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次；

⑦若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除；

若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除；

14.3 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

14.4 持久化

1）RDB 持久化:

每隔一段时间，将内存中的数据集写到磁盘

Redis 会单独创建（fork）一个子进程来进行持久化，会先将数据写入到个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何 IO 操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那 RDB 方式要比 AOF 方式更加的高效。

保存策略：

save 9　00 1　 900 秒内如果至少有 1 个 key 的值变化，则保存

save 300　10　 300 秒内如果至少有 10 个 key 的值变化，则保存

save 60 1 0000　60 秒内如果至 10000 个 key 的值变化，则保存

2）AOF：以日志形式记录每个更新（(总结、改）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：

（1）appendfsync always：每次产生一条新的修改数据的命令都执行保存操作；效率低，但是安全！

（2）appendfsync everysec：每秒执行一次保存操作。如果在未保存当前秒内操作时发生了断电，仍然会导致一部分数据丢失（即 1 秒钟的数据）。

（3）appendfsync no：从不保存，将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 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：

可读的日志文本，通过操作 AOF

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

14.5 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。

Redis 不支持悲观锁。Redis 作为缓存服务器使用时，以操作为主，很少写操作，相应的操作被打断的几率较少。不采用悲观锁是为了防止降低性能。

14.6 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。

十五、MySql

15.1 行锁，表锁

	MyISAM	InnoDB
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

15.2 索引

数据结构：B+Tree

一般来说能够达到 range 就可以算是优化了

口诀：

全职匹配我最爱，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。

十六、JVM

16.1 GC

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

16.2 GC 的简单理解

JVM 的区域分为两类 堆、非堆（Perm Gen 永久代 / Java 8 改为 元空间 Metaspace），堆区：Eden Space（伊甸园）、Survivor Space(幸存者区 0/1)、Tenured Gen（养老区）新生代：Eden Space（伊甸园）+ Survivor Space 0 + Survivor Space 1 养老代：Tenured Gen

a) 一个对象被创建后，就会出现在 Eden 区，每过一段时间 GC 就会过来进行回收，当对象没有引会被回收，有引用就会进入 Survivor Space 幸存者区

b) 在幸存者区里面有两个区域，一个是有对象的区域，另外一个是没对象的区域，有且仅有一个区域有对象。当 GC 到来时，有引用的对象都会转移到另外一个幸存者区，剩下没有引用的全部回收，当达对象到 16 次的没回收时候进入养老区

c) 在养老区内，GC 来的次数比较少，但是当 GC 来的时候，没有引用的对象一样会被清除。

d) 总结：新生代空间比较小，GC 频率高，养老代空间比较大，GC 频率低

十七 面试中项目讲解思路

1）自我介绍

自我介绍准备 2 个版本

（1）短的（1-2 分钟）。体现的一定全是你的优势，（一本，年龄 25，计算机专业）

- 学习方面

  喜欢研究新技术、官网、论坛，社区
- 能加班，能吃苦
- 团队之间感情深厚
- 工作经验

（2）长的：一直聊

基本上会直接进入到技术、和项目层面的面试。

2）面试交互（40 分钟以上，面试成功的概率高达 80%。20 分钟以下，面试失败的概率 80%）

（1）只问技术（大公司，可不可培养---》解决问题方案，思路）---》试探性扩展，看面试

通常：面试官从简历技术第一条开始，从上往下问技术。

- 面试策略：

当你正常答完面试官的问题后，可以尝试着扩展。比如问完 hadoop 的 shuffle 后，尝试着说说 hadoop 的优化、hadoop 的调度策略。围绕着 hadoop 技术都扩展完。你还可以往其他框架上扩展。比如从 hadoop 跳转到 spark（框架的相识度）等。或者从 flume 跳转到 kafka（数据采集的通道）等。而且在聊技术框架的时候，最好要结合项

目去说。比如问你 hadoop 的优化技术，要说有哪些优化手段，你们在项目中采用了哪些手段，哪个效果更好些等。最后绝大多数面试官是能接受扩展的。如果发现面试官不喜欢你扩展，就立即停止。

简单的问题不允许不会，特别生僻的问题，如果不会就直接说之前开发中没遇到过，可以回去研究一下。不要在你不会的问题上纠缠太久。让整个面试过程中你会的问题比例尽可能的高。

- 复习策略：

简历上写的每一项技术都必须是掌握的。评测自己掌握没掌握的标准是，针对每一个技术点，问自己 3 个技术问题。如果能很快想起来，说明你会了。比如：hadoop 的 shuffle、hadoop 优化、hadoop 的调度策略等。任何技术点，尽可能的把自己会的都聊出来。

（2）只问项目（中小型公司）

来了就干活

技术加项目（中小型公司）

项目

2-3 句话简单介绍项目是做什么的，优势（跟竞争对手优势）

项目用到哪些框架（app 项目，观察会议室有没有白板，没有自带白纸，提前备好，至少在家写 3 遍）

画一个说一下知识点

十八 企业中项目相关问题

18.1 每天的离线数据要处理多长时间

8 小时左右

18.2 京东：Spark Executor 具体配置

spark.executor.extraJavaOptions -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=70 -XX:MaxHeapFreeRatio=70 -XX:+CMSClassUnloadingEnabled -XX:OnOutOfMemoryError='kill -9 %p'

spark.hadoop.yarn.timeline-service.enabled false

spark.executor.memory 10240M

spark.executor.cores

memory 和 core 的数据基本是 yarn 的 nodeManager 在节点上申请的资源的 1 倍或 1/2,1/3，这样能充分利用计算机的资源。

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

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

18.4 一个项目，从拿到需求到得到报表统计结果，经过哪些实际的工作流程

- 先与产品讨论，看报表的各个数据从哪些埋点中取
- 将取得逻辑过程设计好，与产品确定后开始开发
- 开发出报表 sql 脚本，并且跑几天的历史数据，观察结果
- 将报表放入调度任务中，第二天给产品看结果。
- 周期性将表结果导出或是导入后台数据库，生成可视化报表

18.5 工作中用到过哪些 shell 脚本，shell 脚本的具体功能

- 我们的数仓的脚本都是 shell 的。
- Flume 的统一部署用 shell
- 一些 spark 任务，用 shell 执行，并且监控结果

功能就是传入时间，或是地址等其他参数，给脚本执行。

18.6 数仓拿到数据之后怎么根据数据特征进行表格的设计，具体怎么维度建模

表格是指二维表吧。这个如果说数据特征，看是行为分析类型的，还是对象分析类型的。前面用 kimball 建模，适合数据量大的事实表，后面用 ER 建模，适合聚合分析。

18.7 对于具体的业务要如何设计表格，如何设计日志采集系统

看是行为分析类型的，还是对象分析类型的。前面用 kimball 建模，后面用 ER 建模

Flume 基本能满足所有的数据源全量采集（jdbc，文件，kafka）。文件的增量采集也试用，而数据库的增量采集需要监控日志（如监控 Mysql 的 binlog）。所以我们设计日志采集系统是

- 找出适合该数据源的 source，如 flume，或是 canal
- 设计中间的缓冲层，kafka 或是 flume 的 filechannel
- 设计最后的落地层，根据实时性，是 redis,Mysql，hbase 还是 hdfs 等

注：日志采集系统的后面部分看实时性，比如 elasticsearch,redis（有缓存，实时性高，可覆盖）,hdfs（原始数据结构可读性高，但文件流写入，实时低，可以覆盖），HBASE（实时高，可以覆盖），Mongodb（实时高，不可覆盖）。

18.8 常用的报表工具

只知道一个 kibana，和 sqlpad，这个不太懂，ETL 用 Kettle

18.9 Spark 任务使用什么进行提交，javaEE 界面还是脚本

Shell 脚本。

18.10 数据仓库每天跑多少张表，全跑还是跑一部分，大概什么时候跑？

每天跑一百多张表，0：00 开始跑。

18.11 谈一下你们公司的整体架构。

属于研发部，技术总监下面有各个项目组，我们属于数据组，其他还有后端项目组，基础平台等。总监上面就是副总等级别了。其他的还有产品运营部，等。

18.12 你们公司的测试环境是什么样的。

测试环境的配置是生产的一半。

18.13 测试环境的数据量大概多少。

离线跟生产是同一个数据源，我们用的是分布式对象存储系统 S3，所以测试也可以访问生产的 s3。一般拿生产数据的一半测试。Kafka 的数据也是从离线数据里面拿的，或是造数据，一般比较少。

18.14 你感觉你们公司的架构合理不，有什么改进的地方。

- 目前我们所有产品的日志都写在一个 kafka topic 里面，后面可以分开
- 从日志到 kafka 的过程，flume 其实有一层缓冲，可以通过自己写 producer 来发送消息，大大减少使用 flume 占用的内存和空间。
- 引入 kettle 做 ETL 的工作

18.15 你们公司 3 年进行过多少次的项目迭代，每一个项目具体是如何迭代的。

差不多一个月会迭代一次。就产品或我们提出优化需求，然后评估时间。每周我们都会开会做下周计划和本周总结。

有时候也会调去做一些 java 的事情。

18.16 你们公司集群的运行性能怎么样，是有什么好的改进措施。

集群是弹性的。能满足临时需要补数据的任务。性能若是 6 亿多的原始数据一起跑清洗，跑满整个队列，一个小时左右。

18.19 你个人在工作过程中，谁给你提出需求，做完之后提交给谁，通过什么提交。

运营和产品。代码提交 git，用 git 命令。结果生成数据表，导入 Mysql，通过 Hue 查询，或是导出 csv，给他们看

18.20 你们大数据组一天要提交多少 job？

整个组的话，生产环境按一个表就一个任务，然后两百多个表，加上实时的数据，差不多 300 多个任务。

18.21 数据仓库权限问题，涉及哪些业务系统的数据，有没有设置权限管理，你的权限是什么？

我们的权限是让写入 hdfs 的数据只读（原始数据只读）。开放给外部都是只读权限（GUI 或接口）

18.22 使用什么进行 ETL？有没有专门的 ETL 工具。

没有用专门的 etl 工具，我们都是手写 sql 脚本。（Kettle 可以去了解一下）

18.23 数据量非常大的时候如何更好的将数据导入 Hive？

非常大的时候不需要导入 hive，直接写入 hdfs，用 hive 外部表关联。

18.24 有没有其他的日志采集框架？（咱们的说的人太多了）

Logstash

18.25 你们部门属于公司的第几级部门，部门的职级等级，晋升规则。

这个第几级部门不是很清楚。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。

18.26 数据回溯。

我们接口层的数据一般都有保存在 S3 中，而且保存了 2 年以上的数据。可以随时回溯之前的原始数据。

18.27 从业务数据库每天采集多少条业务数据，大概多大？采集流程耗时多少时间。

全表同步 20 多张表，大概几亿数据。时间一小时左右。

18.28 每天处理多少条实时数据？

2 千万左右。

18.29 每天清洗完成的数据量有多大？

这个表太多了，没有统计过。但原始数据是 6,7 亿，清洗完成之后，出现细节表，各个埋点表，还有产品表，相当于*3，所以是十几亿的样子。

18.30 项目完后如何测试（测试使用那些监控工具），如果测试发现问题如何解决，最后如何上线

集群性能就看 ganglia 系统负载。

任务性能就看 spark history server 看应用的运行日志，用了多少 executor。

Hadoop 的任务就看 yarn 的 log 日志。

还有打点看实时任务的延时，性能调配一般在生产环境调节资源。

结果写入 kafka 或 hive 表中，查看数据。Spark 调试用 show（）或 take（），hive 调试用中间表。

上线的时候，将脚本打包，提交 git。先发邮件抄送经理和总监，运维。通过之后跟运维一起上线。

18.31 你具体每天的工作内容，具体说明每天大概做什么

新需求比如埋点或是报表来了之后，需要设计做的方案，设计完成之后跟产品讨论，再开发。

数仓的任何步骤出现问题，需要查看问题，比如日活，月活下降等。

18.32 实现一个需求大概多长时间

不一定，有长有短，长的一周，短的半天也有。

18.33 测试数据从哪里来，数据量多大

从生产环境直接拿。生产的数据有落地到 S3，测试环境可以直接访问。

18.34 项目测试时的各项测试指标

一个就是任务执行时间，这个可以看 spark 日志。多少次 shuffle，项目的延时。还有就是读取和写入的数据量。

18.35 每天离线生成的 HDFS 文件会有多少

这个没有具体统计过，但 200 多张表，hive 一般一个表 60 多个文件，spark 200 个文件，粗略估计 2w 多个文件。

18.36 Sqoop 数据导出的时候一次执行多长时间，如何进行 Sqoop 调优

Sqoop 这个没有用过

18.37 kafka 消息数据积压，kafka 消费能力不足怎么处理？

1.找出造成积压的瓶颈，是否在 kafka 消费能力不足，还是下游的数据处理不及。

2.如果是前者，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）

3.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。

18.38 一个情景处理（集群优化相关）

情景描述：

数据流向：数据源->Flume->Kafka->HDFS->Hive(只做离线处理)

总共 7 台机器，每天几亿条数据

面临问题：

数据统计主要用 HiveSQL，没有数据倾斜，小文件已经做了合并处理，开启的 JVM 重用，而且 IO 没有阻塞，内存用了不到 50%。但是还是跑的非常慢，而且数据量洪峰过来时，整个集群都会宕掉。基于这种情况有没有优化方案

解决办法：

内存利用率不够。这个一般是 yarn 的 2 个配置造成的，单个任务可以申请的最大内存大小，和 hadoop 单个节点可用内存大小。调节这两个参数能提高系统内存的利用率。

18.39 Flume 宕机

- oom 这种情况，可以靠调高 jvm 的堆内存大小。
- 采用 filechannel 而不是 memory channel
- 部署多个 flume 同时处理

18.40 Kafka 宕机

- 如果只做离线处理，可以不需要 kafka 了。多个 flume 直接写在硬盘上，使用 file roll sink 切分文件，并且压缩。再直接上传到 hdfs。大大提高效率
- 某个节点上的 broker 宕掉，查看 leader 是否过多。所有的 producer 都跟它建立连接。此时要考虑重新分配 leader，

18.41 Hadoop 宕机

- 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

- 写入文件过量造成 namenode 的宕机。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。

十九 集群与人员配置参考

19.1 不同业务对系统运行的影响

不同的业务，数据结构不同，运算效率也不同，即使数据量一样，也会因为数据结构的设计产生不同的效率影响。比如有些单个数据块 128M，我们直接产生一个 Mapper 来处理，这看起来很合理。但是有时单个数据块 128M，我们可能会设计产生 2 个或更多 Mapper 来处理，因为有些业务类型的数据块可能每一个事件（可以理解为一行数据）操作都会经历非常复杂的逻辑处理。此时需要考虑多个 Mapper。

反之一种情况就是使用一个 Mapper 处理 256M 或更多数据。

业务不同，表结构不同，Mapper 个数不同，进程数线程数不同，CPU 和内存以及网络 IO 的开销便不同，所以相同时间约定下的运算，机器配置也不同。

19.2 数据估算方法

① 选定公司后，能够从易观千帆的值此公司的日活量（选择 10 万~40 万日活的公司）。

② 然后我们需要确定一个用户一天平均产生的日志数量（一般 10~100 条，这个要根据你假设的埋点个数确定，埋点越多，用户一次访问产生的日志就越多）。

③ 然后确定一条日志的大小，可以将数据仓库项目中的埋点日志设计中的日志（APP 启动日志、首页日志、商品详情日志等任选一个）拷贝到 txt 文档中，看一条日志的大小（这是一条日志的大小，由于埋点日志的种类繁多，可以取一个平均数值）。

然后根据上面的三个参数估计你所选择的公司一天的数据量（10G~50G）：

数据总量 = 日活量（一天活跃的用户总数） * 一个用户一天平均产生日志数量 * 一条日志的平均大小

如果发现估算的数据量达不到下限 10G，可以假设埋点日志的种类特别多，或者假设一条日志记录的数据量特别大，这样可以保证数据量能够在上述范围内。

公司一天的数据量在面试过程中必问，一定要计算好。

例如：10G 数据 = 20 万日活 * 100 条 = 1000 万条*1k（或者 2000 万条 * 0.5k）

用户总量：100 万-200 万

半年数据量：10G（日活）*4（数仓翻倍）*30 天*6 个月=7T

19.3 集群配置

不同的业务，节点类型不同，资源配置不同。

以下是一个大概的集群规划，按照这个规划进行横向拓展即可，该集群规划适用于100TB 左右的数据。

一般 100 万条数据，对应的数据大小大概是 20~50M 之间（不绝对，取决于日志大小），一个 DataNode 磁盘配置大概是：2T * 7 如果数据冗余 3 份，再除以 30%。

- **SPARK 计算节点/HDFS 存储节点/HBASE 数据节点：共 13 台**

  数量：PCServer，13 台

  配置：

  Ø CPU ： 2 路 8 核 / 或 2 路 6 核

  Ø 内存：64GB（16GB*4）、

  Ø 系统盘：2.5 寸 SAS 硬盘， 600G*2 块 / 或 300G*2 块、支持 RAID0、1、10 等

  Ø 硬盘：2TB*7 块 （SAS 盘 /或企业级 SATA 盘）、支持 RAID5 即可

  Ø 网口：1GB*4 + 10Gb*2 / 或 1Gb*8

  由于 Spark 离线处理的数据来源基本全部为 Hive，而 Hive 的数据存储于 HDFS，所以 Spark 往往部署于 Hadoop 的 DataNode，Hbase 的部署同理，出发点都是数据本地化。

- **HDFS 存储中心节点（HA）：2 台**

  数量：PCServer，2 台

  配置：

  Ø CPU：2 路 8 核 / 或 2 路 6 核

  Ø 内存：64GB（16GB*4）、

  Ø 系统盘：2.5 寸 SAS 硬盘， 600G*2 块 / 或 300G*2 块、支持 RAID0、1、10 等

  Ø 硬盘：2TB*7 块 （SAS 盘 /或企业级 SATA 盘）、支持 RAID5 即可

  Ø 网口：1GB*4 + 10Gb*2 / 或 1Gb*8

- **Zookeeper 节点：3 台**

  数量：PCServer，2 台

  配置：

  Ø CPU：2 路 8 核 / 或 2 路 6 核

Ø 内存：64GB（16GB*4）、

Ø 系统盘：2.5 寸 SAS 硬盘， 600G*2 块 / 或 300G*2 块、支持 RAID0、1、10 等

Ø 硬盘：2TB*7 块 （SAS 盘 /或企业级 SATA 盘）、支持 RAID5 即可

Ø 网口：1GB*4 + 10Gb*2 / 或 1Gb*8

- **Kafka/Flume 节点：共 3 台（内存和硬盘充足的话可以混合部署在 HDFS 节点）**

数量：PCServer，2 台

配置：

Ø CPU：2 路 8 核 / 或 2 路 6 核

Ø 内存：64GB（16GB*4）、

Ø 系统盘：2.5 寸 SAS 硬盘， 600G*2 块 / 或 300G*2 块、支持 RAID0、1、10 等

Ø 硬盘：2TB*7 块 （SAS 盘 /或企业级 SATA 盘）、支持 RAID5 即可

Ø 网口：1GB*4 + 10Gb*2 / 或 1Gb*8

- **ETL 节点：2 台**

数量：PCServer，2 台

配置：

Ø CPU：2 路 8 核 / 或 2 路 6 核 、

Ø 内存：64GB（16GB*4）、

Ø 系统盘：2.5 寸 SAS 硬盘， 600G*2 块 / 或 300G*2 块、支持 RAID0、1、10 等

Ø 硬盘：2TB*7 块 （SAS 盘 /或企业级 SATA 盘）、支持 RAID5 即可

Ø 网口：1GB*4 + 10Gb*2 / 或 1Gb*8

- **交换机：4 台**

a) 建议分布到 3 个机架

b) 建议 4 台万兆交换机（或者 3 台千兆交换机 + 1 台万兆交换机）配置

以上为参考配置，一般一天数据在几十 GB 规模的公司的集群规模在 10 台左右，并且一般多个框架部署在相同的节点，因为服务器性能较高，内存和硬盘能够满足混合部署，面试前必须选定一个配置，面试必问。

19.4 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），JavaEE 1 人（有或者没有人单独负责 JavaEE，有时是有组员大数据和 JavaEE 一起做，或者大数据和前端一起做）。

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，JavaEE 1 人左右（负责对接 JavaEE 业务），前端 1 人左右（有或者没有人单独负责前端）。

中大型公司（5~20 人左右）：组长 1 人，离线 5~10 人（离线处理、数仓），实时 5 人左右，JavaEE2 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。

19.5 项目数据库与表格

基本一个项目建一个库，表格个数为初始的原始数据表格加上统计结果表格的总数。（一般 10~20 张表格）

19.6 项目中系统指标

大数据实时处理部分控制在 5 分钟之内。

所有离线数据报表控制在 8 小时之内

集群负载一般只要不宕机就没有问题，然后负载过高需要报警。一天平均的负载要在 70%以下。

HDFS 和硬盘空闲控制在 70%以下。

二十 Java 基础相关

20.1 JavaSE

1. 什么是 Java 的序列化，如何实现 Java 的序列？列举在哪些程序中见过 Java 序列化?

答：Java 中的序列化机制能够将一个实例对象（只序列化对象的属性值，而不会去序列化什么所谓的方法。）的状态信息写入到一个字节流中使其可以通过 socket 进行传输、或者持久化到存储数据库或文件系统中；然后在需要的时候通过字节流中的信息来重构一个相同的对象。一般而言，要使得一个类可以序列化，只需简单实现

java.io.Serializable 接口即可。

2. **String 和 StringBuffer、StringBuilder 的区别是什么?**

答:

1、String 类是不可变类，即一旦一个 String 对象被创建后，包含在这个对象中的字符序列是不可改变的，直至这个对象销毁。

2、 StringBuffer 类则代表一个字符序列可变的字符串，可以通过 append、insert、reverse、setChartAt、setLength 等方法改变其内容。一旦生成了最终的字符串，调用 toString 方法将其转变为 String

3、JDK1.5 新增了一个 StringBuilder 类，与 StringBuffer 相似，构造方法和方法基本相同。不同是 StringBuffer 是线程安全的，而 StringBuilder 是线程不安全的，所以性能略高。通常情况下，创建一个内容可变的字符串，应该优先考虑使用 StringBuilder

3. 不通过构造函数也能创建对象吗?

答：Java 创建对象的几种方式（重要）：

1、 用 new 语句创建对象，这是最常见的创建对象的方法。

2、 运用反射手段,调用 java.lang.Class 或者 java.lang.reflect.Constructor 类的 newInstance()实例方法。

3、 调用对象的 clone()方法。

4、运用反序列化手段，调用 java.io.ObjectInputStream 对象的 readObject()方法。

(1)和(2)都会明确的显式的调用构造函数 ；(3)是在内存上对已有对象的影印，所以不会调用构造函数；(4)是从文件中还原类的对象，也不会调用构造函数。

4. Java 的 HashMap 和 Hashtable 有什么区别 HashSet 和 HashMap 有什么区别? 使用这些结构保存的数需要重载的方法是哪些?

答:

HashMap 与 Hashtable 实现原理相同，功能相同，底层都是哈希表结构，查询速度快，在很多情况下可以互用

两者的主要区别如下

1、Hashtable 是早期 JDK 提供的接口，HashMap 是新版 JDK 提供的接口

2、Hashtable 继承 Dictionary 类，HashMap 实现 Map 接口

3、Hashtable 线程安全，HashMap 线程非安全

4、Hashtable 不允许 null 值，HashMap 允许 null 值

HashSet 与 HashMap 的区别

1、HashSet 底层是采用 HashMap 实现的。HashSet 的实现比较简单，HashSet 的绝大部分方法都是通过调用 HashMap 的方法来实现的，因此 HashSet 和 HashMap 两个集合在实现本质上是相同的。

2、HashMap 的 key 就是放进 HashSet 中对象，value 是 Object 类型的。

3、当调用 HashSet 的 add 方法时，实际上是向 HashMap 中增加了一行(key-value 对)，该行的 key 就是向 HashSet 增加的那个对象，该行的 value 就是一个 Object 类型的常量

5. 接口和抽象类的区别

相同点：
(1) 抽象类和接口均包含抽象方法，类必须实现所有的抽象方法，否则是抽象类
(2) 抽象类和接口都不能实例化，他们位于继承树的顶端，用来被其他类继承和实现

6. 静态内部类和内部类有什么区别

答:

静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。

静态内部类可以有静态成员(方法，属性)，而非静态内部类则不能有静态成员(方法，属性)。

非静态内部类能够访问外部类的静态和非静态成员。静态内部类不能访问外部类的非静态成员，只能访问外部类的静态成员。

实例化方式不同：

1) 静态内部类：不依赖于外部类的实例，直接实例化内部类对象

2) 非静态内部类：通过外部类的对象实例生成内部类对象

7. 反射的概念与作用

答:

反射的概念：

反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通过采用某种机制来实现对自己行为的描述（self-representation）和监测（examination），并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

反射机制是 Java 动态性的重要体现。我们可以通过反射机制在运行时加载编译期完全未知的类。通过反射机制，我们可以在运行时加载需要的类，从而动态的改变了程序结构，使我们的程序更加灵活、更加开放。

反射的作用：

通过反射可以使程序代码访问装载到 JVM 中的类的内部信息

1) 获取已装载类的属性信息

2) 获取已装载类的方法

3) 获取已装载类的构造方法信息

8. Java 线程的几种状态

答:

线程是一个动态执行的过程，它有一个从产生到死亡的过程，共五种状态：

新建（new Thread）

当创建 Thread 类的一个实例（对象）时，此线程进入新建状态（未被启动）。

例如：Thread t1=new Thread();

就绪（runnable）

线程已经被启动，正在等待被分配给 CPU 时间片，也就是说此时线程正在就绪队列中排队等候得到 CPU 资源。例如：t1.start();

运行（running）

线程获得 CPU 资源正在执行任务（run()方法），此时除非此线程自动放弃 CPU 资源或者有优先级更高的线程进入，线程将一直运行到结束。

死亡（dead）

当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

自然终止：正常运行 run()方法后终止

异常终止：调用 stop()方法让一个线程终止运行

堵塞（blocked）

由于某种原因导致正在运行的线程让出 CPU 并暂停自己的执行，即进入堵塞状态。

正在睡眠：用 sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间

过去可进入就绪状态。

正在等待：调用 wait()方法。（调用 motify()方法回到就绪状态）

被另一个线程所阻塞：调用 suspend()方法。（调用 resume()方法恢复）

9. int 与 Integer 有什么区别?

答:

int 是 java 提供的 8 种原始数据类型之一。Java 为每个原始类型提供了封装类，Integer 是 java 为 int 提供的封装类。int 的默认值为 0，而 Integer 的默认值为 null，即 Integer 可以区分出未赋值和值为 0 的区别，int 则无法表达出未赋值的情况，例如，要想表达出没有参加考试和考试成绩为 0 的区别，则只能使用 Integer。在 JSP 开发中，Integer 的默认为 null，所以用 el 表达式在文本框中显示时，值为空白字符串，而 int 默认的默认值为 0，所以用 el 表达式在文本框中显示时，结果为 0，所以，int 不适合作为 web 层的表单数据的类型。

在 Hibernate 中，如果将 OID 定义为 Integer 类型，那么 Hibernate 就可以根据其值是否为 null 而判断一个对象是否是临时的，如果将 OID 定义为了 int 类型，还需要在 hbm 映射文件中设置其 unsaved-value 属性为 0。

另外，Integer 提供了多个与整数相关的操作方法，例如，将一个字符串转换成整数，Integer 中还定义了表示整数的最大值和最小值的常量。

10. 描述 final、finally、finalize 的区别。

答:

final 修饰符（关键字）如果一个类被声明为 final，意味着它不能再派生出新的子类，不能作为父类被继承。将变量或方法声明为 final，可以保证它们在使用中不被改变。被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。被声

明为 final 的方法也同样只能使用，不能重载。

finally 在异常处理时提供 finally 块来执行任何清除操作。如果有 finally 的话，则不管是否发生异常，finally 语句都会被执行。

finalize 方法名。Java 技术允许使用 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清理工作。finalize() 方法是在垃圾收集器删除对象之前被调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作。

11. ==和 equals 的区别和联系

“==”是关系运算符，equals()是方法，同时他们的结果都返回布尔值；

“==”使用情况如下：

a) 基本类型，比较的是值

b) 引用类型，比较的是地址

c) 不能比较没有父子关系的两个对象

equals()方法使用如下：

a) 系统类一般已经覆盖了 equals()，比较的是内容。

b) 用户自定义类如果没有覆盖 equals()，将调用父类的 equals（比如是 Object），而 Object 的 equals 的比较是地址（return (this == obj);）

c) 用户自定义类需要覆盖父类的 equals()

注意：Object 的==和 equals 比较的都是地址，作用相同

12、HashMap 扩容机制

三个常量

```
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final int MAXIMUM_CAPACITY = 1 << 30;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

三个常量中可以看出，默认的容器大小是 16，最大长度是 2 的 30 次方，load factor 默认是 0.75，扩充的临界值是 16*0.75=12
HashMap 进行扩容
当 HashMap 中的元素个数超过数组大小乘以负载因子(loadFactor)时，就会进行数组扩容，loadFactor 的默认值为 0.75，
也就是说，默认情况下，数组大小为 16，
那么当 HashMap 中元素个数超过 16*0.75=12 的时候，
就把数组的大小扩展为 2*16=32，即扩大一倍，
然后重新计算每个元素在数组中的位置。

13、如何解决 HashMap 的线程不安全问题?

1.替换成 Hashtable，Hashtable 通过对整个表上锁实现线程安全，因此效率比较低
2.使用 Collections 类的 synchronizedMap 方法包装一下。方法如下：
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) 返回由指定映射支持的同步（线程安全的）映射
3.使用 ConcurrentHashMap，它使用分段锁来保证线程安全
通过前两种方式获得的线程安全的 HashMap 在读写数据的时候会对整个容器上锁，而 ConcurrentHashMap 并不需要对整个容器上锁，它只需要锁住要修改的部分就行了

14、什么是比特(Bit),什么是字节(Byte),什么是字符(Char),它们长度是多少,各有什么区别

Bit 最小的二进制单位 ，是计算机的操作部分 取值 0 或者 1
Byte 是计算机操作数据的最小单位由 8 位 bit 组成 取值（-128-127）
Char 是用户的可读写的最小单位，在 Java 里面由 16 位 bit 组成 取值（0-65535）
Bit 是最小单位 计算机 只能认识 0 或者 1 8 个字节 是给计算机看的字符 是看到的东西 一个字符=二个字节

15、把一个对象写入数据源或者从一个数据源读出来,用哪两个流

ObjectInputStream ObjectOutputStream

20.2 Mysql

1. 如果有很多数据插入 MYSQL 你会选择什么方式?

答: (1) 利用 mybatis 的 foreach 拼接动态 sql 或者 java 中写循环拼接，将数据分组拼接成大 sql，比如可以每 1 万行数据拼接为一个 insert 语句。

(2) 设置 Mybatis 的 sqlsession 的 ExecutorType 为 batch,如果用 jdbc 则用 executeBatch。

(3) 去掉表中的非主键索引。

(4) 取消该表自动提交。

（5）可以利用多线程执行。

1　直接在 mysql 层面，可以通过写存储过程和函数构建大批量数据插入的 pl/sql 程序，直接在 mysql 数据层面自己处理。

2　mysql 使用 load file 导入[大]批量数据

2. **如果查询很慢，你会想到的第一个方式是什么？索引是干嘛的?**

答：（1）第一个方式是建立索引

（2）索引是是帮助 MySQL 高效获取数据的数据结构。可以提高数据检索的效率，提高排序效率。

Mysql 查询慢查询流程：

1　如果 mysql 服务器变慢了，需要首先判断是 CPU 慢还是 IO 慢，需要用 top 命令检查出来高 CPU 和 IO 占用的情况是 SQL 导致还是程序导致，如果是 SQL 下一步。
2　结合业务打开 MySQL 慢查询日志抓出执行时间超过 3 秒钟以上的 SQL 语句
3　将抓出的语句用 explain 语句命令进行分析，重点看 type/key/rows/extra 等关键字段
4　按照上一步的判断，可以重新新建或者调整数据与索引

2.　如果建了一个单列索引，查询的时候查出 2 列，会用到这个单列索引吗？

答：会用到。

4. 如果建了一个包含多个列的索引，查询的时候只用了第一列，能不能用上 这个索引？查三列呢?

答：

（1）查询的时候只用了第一列，能用上这个索引

（2）查三列要看具体情况，如果这个索引刚好就是三个，那就是全部对应，自然可以全部用上。其它情况看分析。

5. **接上题，如果 where 条件后面带有一一个 i + 5 < 100 会使用到这个索引吗?**

答：不会使用到这个索引

6. **怎么看是否用到了某个索引?**

答：使用Explain分析sql，key字段是实际使用的索引

7. **like %aaa%会使用索引吗? like aaa%呢?**

答：like %aaa%不会使用索引，like aaa%能使用上索引

8. **drop、truncate、delete 的区别?**

答：（1）drop 语句将删除表的结构被依赖的约束，和表数据

（2）truncate 只删除表数据，不能回滚

（3）delete 只删除表数据，能回滚

9. **平时你们是怎么监控数据库的? 慢 SQL 是怎么排查的?**

答：（1）可以通过 mysql 日志监控数据库

（2）慢 SQL 可以通过配置慢查询日志排查

10. **你们数据库是否支持 emoji 表情，如果不支持，如何操作?**

答：（1）mysql 的 utf8 编码的一个字符最多 3 个字节，但是一个 emoji 表情为 4 个字节，所以 utf8 不支持存储 emoji 表情。

（2）可以修改 mysql 数据库的编码格式变为 utf8mb4，utf8mb4 一个字符最多能有 4 字节，能支持 emoji 表情的存储。

11. **你们的数据库单表数据量量是多少？一般多大的时候开始出现查询性能急 剧下降?**

答：（1）最大 100-200 万。

（2）一般 500 万出现查询性能急剧下降

12. **查询死掉了，想要找出执行的查询进程用什么命令？找出来之后一般你会干嘛?**

答：（1）使用 SHOW PROCESSLIST 查看进程

（2）找出后使用 kill 【id】杀掉进程

13. **读写分离是怎么做的？你认为中间件会怎么来操作？这样操作跟事务有什么关系？**

答：（1）读写分离可以通过写库和读库做主从备份，再加上数据库中间件（如 mycat）来实现。

（2）中间件会对 sql 语句进行拦截，之后判断是读操作还是写操作，分别分配到读库或写库

（3）事务内部的一切操作都会走写节点，所以读操作不要加事务。

14. **分库分表有没有做过？线上的迁移过程是怎么样的？如何确定数据是正确的？**

答：（1）分库分表表可以通过 mycat 来实现

（2）备份所有节点数据，如迁移失败后数据恢复

进行扩容、缩容操作：表迁移信息迁移计划、执行数据迁移、清理旧节点冗余数据、校验迁移是否成功等。

（3）验证数据总量是否一致

随机抽样数据验证是否一致

迁移前后统计报表是否一致

15. **什么是幂等？什么情况下需要考虑幂等？你怎么解决幂等的问题？**

答：（1）一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。

（2）在电商或者其他的项目中，处理重复的订单需要考虑幂等

（3）可通过插入数据的唯一索引、分布式锁等方式解决幂等的问题

20.3 Redis

1. 缓存穿透可以介绍一一下么？你认为应该如何解决这个问题

答：缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，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。

1、对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃。还有最常见的则是采用布隆过滤器，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、也可以采用一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

2. 你是怎么触发缓存更更新的？(比如设置超时间(被动方式)，比如更更新的时候主动 update)？如果是被动的方式如何控制多个入口同时触发某个缓存更更新?

答： [主动]需要操作人员去操作，或者定时调度、[被动]由用户触发更新、[预加载]提前加载好数据

被动更新出现并发问题，可以通过程序锁，锁住更新操作只能有一个进入 DB 查询，可以避免问题。

3. 你们用 Redis 来做什么？为什么不用其他的 KV 存储例例如 Memcached, Cassandra 等?

答：用 redis 做缓存数据库。Redis 与 Memcached 相比，几乎覆盖后者绝大部分功能，还有其独有功能：支持持久化、支持多数据类型

4. 你们用什么 Redis 客户端? Redis 高性能的原因大概可以讲一些?

答：使用 Java 的 Redis 客户端 Jedis，Redis 的高性能在于单线程和多路 IO 复用

5. 你熟悉哪些 Redis 的数据结构? zset 是干什么的? 和 set 有什么区别?

答：Redis 五大数据结构：string、set、list、hash、zset。

zset 是有序集合，最常用来实现排行榜。

和 set 的区别，set 是无序集合，zset 是有序集合

6. Redis 的 hash, 存储和获取的具体命令叫什么名字?

答：存储命令：hset <key> <field> <value>

获取命令：hget <key1> <field>

7. LPOP 和 BLPOP 的区别?

答：BLPOP 是阻塞式列表的弹出命令。 它是命令 LPOP 的阻塞版本，这是因为当给定列表内没有任何元素可供弹出的时候， 连接将被 BLPOP 命令阻塞。 当给定多个 key 参数时，按参数 key 的先后顺序依次检查各个列表，弹出第一个非空列表的头元素。

8. Redis 的有一些包含 SCAN 关键字的命令是干嘛的? SCAN 返回的数据量是固定的吗?

答：SCAN 命令及其相关的 SSCAN 命令、 HSCAN 命令和 ZSCAN 命令都用于增量地迭代，它们每次执行都只会返回少量元素。

SCAN 返回的数据量不固定。

9. Redis 中的 Lua 有没有使用过? 可以用来做什么? 为什么可以这么用?

答：使用过，可以用来解决秒杀业务的高并发请求。

Lua 可以将复杂或多步的 redis 操作，写成脚本一次性提交给 redis 执行，提升性能

Lua 脚本类似于 redis 事务有一定原子性，不会被其他命令插队，可以完成一些 redis 事务操作

10. Redis 的 Pipeline 是用来干什么的?

答：pipeline（管道）可以一次性发送多条命令并在执行完后一次性将结果返回，pipeline 通过减少客户端与 redis 的通信次数来实现降低往返延时时间。

11. Redis 持久化大概有几种方式? aof 和 rdb 的区别是什么? AOF 有什么优缺 点吗?

答：redis 持久化分两种：aof、rdb。

rdb 是以快照的方式全量备份 redis 数据，aof 是以日志方式增量备份 redis 写操作。

aof 优点：备份机制更稳健、丢失数据概率更低；可读的日志文本，通过操作 aof 文件可处理误操作。

aof 缺点：比 rdb 占用更多磁盘空间；恢复备份速度慢；如每次写都同步，有性能压力；有个别 bug，造成恢复不能。

12. Redis Replication 的大致流程是什么? bgsave 这个命令的执行过程?

答：redis Replication 流程：（1）从机向主机发起复制请求（2）主机立刻进行存盘操作，发送 rdb 文件给从机（3）从机收到 rdb 文件后进行全盘加载（4）之后每次主机写操作都会立刻发给从机，从机执行相同命令。

Bgsave 执行过程：BGSAVE 命令执行之后立即返回 OK ，然后 Redis fork 出一个新子进程，原来的 Redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出。

13. 如果有很多 KV 数据要存储到 Redis, 但是内存不足, 通过什么方式可以缩 减内存? 为什么这样可以缩小内存?

答：可以配置 redis 内存淘汰机制，通过内存淘汰机制淘汰已存储数据，存入新数据。

14. Redis 中 List, HashTable 都用用到了 ZipList, 为什么会选择它?

答：redis 初始创建 hash 表,有序集合，链表时，存储结构采用一种 ziplist 的存储结构，这种结

构内存排列更紧密, 能提高访存性能。

20.4 JVM

1. **你知道哪些或者你们线上使用什么 GC 策略略? 它有什么优势，适用于什么场景?**

1). New Generation 的 GC 策略

Serial GC。采用单线程方式，用 Copying 算法。New Generation 会再次被划分成 Eden Space 和 S0、S1，S0 和 S1 又称为 From Space 和 To Space

Copying 算法就是开辟另外一个内存空间，把扫描到可达的对象复制过去，然后把原内存空间全部清除即可。适用于存活对象较少的情况。

Parallel Scavenge。将内存空间分段来使用多线程，也是用 Copying 算法。

Parallel New 比 Parallel Scavenge 多做了与 Old Generation 使用 CMS GC 一起发生时的特殊处理。

2). Old Generation 的 GC 策略

Serial GC。也是单线程方式，但是实现是将 Mark-Sweep 和 Mark-Compact 结合了下，做了点改进。

Mark-Sweep 算法是把扫描到可达的对象都标记下来，然后把所有未标记的对象清除。但这个方法会引起内存碎片。适用于存活对象较多的情况。

Mark-Compact 算法是在 Mark-Sweep 基础上，在把内存空间整理一下，让存储连续以消除内存碎片。

Parallel Mark-Sweep、Parallel Mark-Compact。同样也是把 Old Generation 空间进行划分成 regions，只是粒度更细了。

CMS（Concurrent Mark-Sweep） GC。主要是为实现并发，就是算法使用的还是

Mark-Sweep，对于内存碎片的问题，CMS 提供了一个额外的内存碎片的整理功能，会在执行几次 Full GC 以后执行一次。

2. JAVA 类加载器包括几种？它们之间的父子关系是怎么样的？双亲委派机制是什么意思？有什么好处？

有 4 种

1). 引导类加载器(BootstrapClassLoader) 是扩展类加载器的父加载器(不是父类), 是最底层的核心类加载器, 由 C++代码写成, 属于 JVM 一部分.

主要加载 JAVA_HOME/lib/rt.jar 中的最核心的类.

2). 扩展类加载器(ExtClassLoader) 是应用程序类加载器的父加载器(不是父类), 扩展类加载器是由 java 代码写的, 主要加载 JAVA_HOME/lib/ext 下的 jar

3). 应用程序类加载器(AppClassLoader) 是自定义类加载器的父加载器(不是父类),也是由 java 代码写的, 主要加载 classpath 中指示的目录中的类文件.

4）. 自定义类加载器, 用户可以选择自行处理 class 文件如何获取, 并读取其中的数据, 再调用父类的方法完成类的初始化.

双亲委派机制是指 除引导类加载器外的所有类加载器在加载类时，都必须先把加载类的任务交给父类加载器完成, 如果父类加载器追朔到引导类加载器,

就不再向上追朔, 直接使用引导类加载器加载类, 所以,可以认为所有类的加载, 都必须要先经过引导类加载器.

如果父类加载器加载类成功,则加载完成, 并返回. 如果父类加载器加载失败, 才会把加载类的任务向下传递.

这样做的好处是核心类库的加载都是由引导类加载器完成的，防止用户加载恶意的核心类，因为引导类加载器只加载 rt.jar 中的类. 如果是其它地方的类则拒绝加载.

3. **如何自定义一个类加载器？你使用过哪些或者你在什么场景下需要一个自定义的类加载器吗?**

通常自定义类加载器需要继承 URLClassLoader，重写方法 loadClass(String name)，在方法中实现如何读取类文件的内容，并最终需要调用父类的 defineClass 来加载并初始化类

在一些需要热布署的应用中使用自定义类加载器，当有某个模块需要动态启动时，创建一个自定义类加载器，然后加载所有相关的类，并执行相关的代码

模块执行完毕后，如果需要卸载，则把类加载器所加载的所有类清空，然后把类加器本身也清空. 这样模块的所有使用到的类就会清除.

类似于 Tomcat 的工作模式，布署一个应用就是创建一个 WebAppClassLoader 对象，然后由这个类加载器加载所有资源.

4. **堆内存设置的参数是什么?**

-Xms 字节 初始化堆大小

-Xmx 字节 最大堆大小

5. **Perm Space 中保存什么数据? 会引起 OutOfMemory 吗?**

永久代中保存的是方法区数据，在 JVM 规范中这是不一定的. 方法区中包含所有类的信息和常量池，在 JDK1.7 之前，如果由于大量的调用字符串方法 intern

就会导致大量的对象创建在常量池中，有可能导致 OutOfMemory,但是在 JDK1.7 以后，这个问题消除了.

6. **做 gc 时，一个对象在内存各个 Space 中被移动的顺序是什么?**

最先在 Eden 代中，如果 GC 后仍存活，则会将对象移动到 Yong 代(青年代)，如果 GC 后仍

存活, 则会将对象移动到 Old 代(老年代).

7. **你有没有遇到过 OutOfMemory 问题？你是怎么来处理这个问题的？处理理过程中有哪些收获?**

有，在 eclipse 中编译大型项目时，如果 eclipse 配置的内存不够大，就会抛出这个问题，解决方法就是修改 eclipse/eclipse.ini 文件,

修改或添加参数-Xmx 设置堆内存的最大空间

8. **1.8 之后 Perm Space 有哪些变动？MetaSpace 大小默认是无限的么？还是你们会通过什么?方式来指定大小?**

1.8 之后永久代取消了，原来在其中保存的类信息数据被移动到元数据区，元数据区是在本地内存中保存的, 不受普通 GC 的管理.

MetaSpace 默认大小是 21m，但是它的容量是受元空间虚拟机管理的，理论上是可以很大的, 甚至可以扩展到交换区.

也可以通过设置参数来改变默认大小 -XX:MetaSpaceSize.

也可以通过设置参数来改变最大空间 -XX:MaxMetaSpaceSize

9. **Jstack 是干什么的？Jstat 呢？如果线上程序周期性地出现卡顿，你怀疑可能是 gc 导致的，你会怎么来排查这个问题？线程日志一般你会看其中的什么部分？**

Jstack 是一个命令, 可以查看运行的 java 进程的内部栈情况, 每个线程都拥有各自的栈.

Jstack 命令执行时, 需要加上一些选项和 java 进程 ID

执行这个命令时, 会自动侦测进程中有无死锁, 还会显示每个线程的当前状态(运行, 阻塞)和方法调用踪迹, 还有当前线程持有的锁.

jstat 可以查看虚拟机的内存管理情况, 比如查看各个代的 GC 工作情况, 以及 GC 的统计信息等

可以通过 jstat -options 查看详细的选项.

-class // 查看加载了多少类

-compiler // 查看编译了多少类

-gc // 查看 GC 情况

-gccapacity // 内存容量

-gccause

-gcmetacapacity // 元空间容量

-gcnew // 年轻代

-gcnewcapacity

-gcold // 老年代

-gcoldcapacity

-gcutil

-printcompilation

10. **StackOverFlow 异常有没有遇到过？一般你猜测会在什么情况下被触发？如何指定一个线程的堆栈大小？一般你们写多少？**

StackOverFlow 是栈溢出错误, 通常会在方法无限递归时会发生, 或者有可能方法调用层数太深(一般不太可能)

线程的堆栈大小设置可以使用选项 -Xss 字节数, 一般设置 50M 左右就可以.

11.说说 JVM 原理？内存泄漏与溢出的区别？何时产生内存泄漏？

JVM 原理：

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，它是整个 java 实现跨平台的最核心的部分，所有的 Java 程序会首先被编译为.class 的类文件，这种类文件可以在虚拟机上

执行，也就是说 class 并不直接与机器的操作系统相对应，而是经过虚拟机间接与操作系统交互，由虚拟机将程序解释给本地系统执行。JVM 是 Java 平台的基础，和实际的机器一样，它也有自己的指令集，并且在运行时操作不同的内存区域。JVM 通过抽象操作系统和 CPU 结构，提供了一种与平台无关的代码执行方法，即与特殊的实现方法、主机硬件、主机操作系统无关。JVM 的主要工作是解释自己的指令集（即字节码）到 CPU 的指令集或对应的系统调用，保护用户免被恶意程序骚扰。JVM 对上层的 Java 源文件是不关心的，它关注的只是由源文件生成的类文件（.class 文件）。

内存泄漏与溢出的区别：

1) 内存泄漏是指分配出去的内存无法回收了。

2) 内存溢出是指程序要求的内存，超出了系统所能分配的范围，从而发生溢出。比如用 byte 类型的变量存储 10000 这个数据，就属于内存溢出。

3) 内存溢出是提供的内存不够；内存泄漏是无法再提供内存资源。

何时产生内存泄漏：

1) 静态集合类：在使用 Set、Vector、HashMap 等集合类的时候需要特别注意，有可能会发生内存泄漏。当这些集合被定义成静态的时候，由于它们的生命周期跟应用程序一样长，这时候，就有可能会发生内存泄漏。

2) 监听器：在 Java 中，我们经常会使用到监听器，如对某个控件添加单击监听器 addOnClickListener()，但往往释放对象的时候会忘记删除监听器，这就有可能造成内存泄漏。好的方法就是，在释放对象的时候，应该记住释放所有监听器，这就能避免了因为监听器而导致的内存泄漏。

3) 各种连接：Java 中的连接包括数据库连接、网络连接和 io 连接，如果没有显式调用其 close()方法，是不会自动关闭的，这些连接就不能被 GC 回收而导致内存泄漏。一般情况下，在 try 代码块里创建连接，在 finally 里释放连接，就能够避免此类内存泄漏。

4) 外部模块的引用：调用外部模块的时候，也应该注意防止内存泄漏。如模块 A 调用了外部模块 B 的一个方法，如：public void register(Object o)。这个方法有可能就使得 A 模块持有传入对象的引用，这时候需要查看 B 模块是否提供了去除引用的方法，如 unregister()。这种情况容易忽略，而且发生了内存泄漏的话，比较难察觉，应该在编写代码过程中就应该注意此类问题。

5) 单例模式：使用单例模式的时候也有可能导致内存泄漏。因为单例对象初始化后将在 JVM 的整个生命周期内存在，如果它持有一个外部对象（生命周期比较短）的引用，那么这个外部对象就不能被回收，而导致内存泄漏。如果这个外部对象还持有其它对象的引用，那么内存泄漏会更严重，因此需要特别注意此类情况。这种情况就需要考虑下单例模式的设计会不会有问题，应该怎样保证不会产生内存泄漏问题。

20.5 多线程

1. 进程和线程之间有什么不同?

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。Java 运行环境是一个包含了不同的类和程序的单一进程。线程可以被称为轻量级进程。线程需要较少的资源来创建和驻留在进程中，并且可以共享进程中的资源。

2. 多线程编程的好处是什么?

在多线程程序中，多个线程被并发的执行以提高程序的效率，CPU 不会因为某个线程需要等待资源而进入空闲状态。多个线程共享堆内存(heap memory)，因此创建多个线程去执行一些任

务会比创建多个进程更好。举个例子，Servlets 比 CGI 更好，是因为 Servlets 支持多线程而 CGI 不支持。

3. 用户线程和守护线程有什么区别?

当我们在 Java 程序中创建一个线程，它就被称为用户线程。一个守护线程是在后台执行并且不会阻止 JVM 终止的线程。当没有用户线程在运行的时候，JVM 关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

4. 我们如何创建一个线程?

有两种创建线程的方法：一是实现 Runnable 接口，然后将它传递给 Thread 的构造函数，创建一个 Thread 对象；二是直接继承 Thread 类。

5. 有哪些不同的线程生命周期?

当我们在 Java 程序中新建一个线程时，它的状态是 New。当我们调用线程的 start()方法时，状态被改变为 Runnable。线程调度器会为 Runnable 线程池中的线程分配 CPU 时间并且讲它们的状态改变为 Running。其他的线程状态还有 Waiting，Blocked 和 Dead。

6. 可以直接调用 Thread 类的 run()方法么?

当然可以，但是如果我们调用了 Thread 的 run()方法，它的行为就会和普通的方法一样，为了在新的线程中执行我们的代码，必须使用 Thread.start()方法。

7. 如何让正在运行的线程暂停一段时间?

我们可以使用 Thread 类的 Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为 Runnable，并且根据线程调度，它将得到执行。

8. 你对线程优先级的理解是什么?

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于

线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量(从 1-10)，1 代表最低优先级，10 代表最高优先级。

9. 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)?

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

10. 在多线程中，什么是上下文切换(context-switching)?

上下文切换是存储和恢复 CPU 状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

11. 你如何确保 main()方法所在的线程是 Java 程序最后结束的线程?

我们可以使用 Thread 类的 joint()方法来确保所有程序创建的线程在 main()方法退出前结束。

12.线程之间是如何通信的?

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object 类中 wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。

13.为什么线程通信的方法 wait(), notify()和 notifyAll()被定义在 Object 类里?

Java 的每个对象中都有一个锁(monitor，也可以成为监视器) 并且 wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在 Java 的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是 Object 类的一部分，这样 Java 的每一个类都有用于线程间通信的基本方法

14. 为什么 wait(), notify()和 notifyAll()必须在同步方法或者同步块中被调用?

当一个线程需要调用对象的 wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的 notify()方法。同样的，当一个线程需要调用对象的 notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

15. 为什么 Thread 类的 sleep()和 yield()方法是静态的?

Thread 类的 sleep()和 yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

16.如何确保线程安全?

在 Java 中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用 volatile 关键字，使用不变类和线程安全类。

17. volatile 关键字在 Java 中有什么作用?

当我们使用 volatile 关键字去修饰变量的时候，所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

18. 同步方法和同步块，哪个是更好的选择?

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

19.如何创建守护线程?

使用 Thread 类的 setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在

调用 start()方法前调用这个方法，否则会抛出 IllegalThreadStateException 异常。

20. 什么是 ThreadLocal?

ThreadLocal 用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择 ThreadLocal 变量。

每个线程都会拥有他们自己的 Thread 变量，它们可以使用 get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal 实例通常是希望它们同线程状态关联起来是 private static 属性。

21. 什么是 Thread Group？为什么不建议使用它?

ThreadGroup 是一个类，它的目的是提供关于线程组的信息。

ThreadGroup API 比较薄弱，它并没有比 Thread 提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaught exception handler)。但在 Java 1.5 中 Thread 类也添加了 setUncaughtExceptionHandler(UncaughtExceptionHandler eh) 方法，所以 ThreadGroup 是已经过时的，不建议继续使用。

22. 什么是 Java 线程转储(Thread Dump)，如何得到它?

线程转储是一个 JVM 活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获取线程转储——使用 Profiler，Kill -3 命令，jstack 工具等等。我更喜欢 jstack 工具，因为它容易使用并且是 JDK 自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。

23. 什么是死锁(Deadlock)？如何分析和避免死锁?

死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上

的资源。

分析死锁，我们需要查看 Java 应用程序的线程转储。我们需要找出那些状态为 BLOCKED 的线程和他们等待的资源。每个资源都有一个唯一的 id，用这个 id 我们可以找出哪些线程已经拥有了它的对象锁。

24. 什么是 Java Timer 类？如何创建一个有特定时间间隔的任务？

java.util.Timer 是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer 类可以用安排一次性任务或者周期任务。

java.util.TimerTask 是一个实现了 Runnable 接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用 Timer 去安排它的执行。

25. 什么是线程池？如何创建一个 Java 线程池？

一个线程池管理了一组工作线程，同时它还包括了一个用于放置等待执行的任务的队列。

java.util.concurrent.Executors 提供了一个 java.util.concurrent.Executor 接口的实现用于创建线程池。

20.6 JUC

1. 什么是原子操作？在 Java Concurrency API 中有哪些原子类(atomic classes)？

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在 JDK1.5 之前我们可以使用同步技术来做到这一点。到 JDK1.5，java.util.concurrent.atomic 包提供了 int 和 long 类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

2. Java Concurrency API 中的 Lock 接口(Lock interface)是什么？对比同步它有什么优势?

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

可以使锁更公平

可以使线程在等待锁的时候响应中断

可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

可以在不同的范围，以不同的顺序获取和释放锁

3. 什么是 Executors 框架?

Executor 框架同 java.util.concurrent.Executor 接口在 Java 5 中被引入。Executor 框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。

4. 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型?

java.util.concurrent.BlockingQueue 的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出 NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

5. 什么是 Callable 和 Future?

Java 5 在 concurrency 包中引入了 java.util.concurrent.Callable 接口，它和 Runnable 接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable 接口使用泛型去定义它的返回类型。Executors 类提供了一些有用的方法去在线程池中执行 Callable 内的任务。由于 Callable 任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future 对象为我们解决了这个问题。在线程池提交 Callable 任务后返回了一个 Future 对象，使用它我们可以知道 Callable 任务的状态和得到 Callable 返回的执行结果。Future 提供了 get()方法让我们可以等待 Callable 结束并获取它的执行结果。

6. 什么是 FutureTask?

FutureTask 是 Future 的一个基础实现，我们可以将它同 Executors 使用处理异步任务。通常我们不需要使用 FutureTask 类，单当我们打算重写 Future 接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。

7.什么是并发容器的实现?

Java 集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的 next()方法将抛出 ConcurrentModificationException 异常。

并发容器支持并发的遍历和并发的更新。

主要的类有 ConcurrentHashMap, CopyOnWriteArrayList 和 CopyOnWriteArraySet

8. Executors 类是什么?

Executors 为 Executor，ExecutorService，ScheduledExecutorService，ThreadFactory 和 Callable 类提供了一些工具方法。

Executors 可以用于方便的创建线程池

9. 为什么说 ConcurrentHashMap 是弱一致性的？以及为何多个线程并发修改 ConcurrentHashMap 时不会报 ConcurrentModificationException?

二十一 模拟考试

1.1 选择题

1.1.1 HDFS

1. 下面哪个程序负责 HDFS 数据存储?

   a)NameNode b)Jobtracker c)Datanode d)secondaryNameNode e)tasktracker

2. HDfS 中的 block 默认保存几份?

   a)3 份 b)2 份 c)1 份 d)不确定

3. 下列哪个程序通常与 NameNode 在一个节点启动?

   a)SecondaryNameNode b)DataNode c)TaskTracker d)Jobtracker

   解析:

   JobTracker 对应于 NameNode

   TaskTracker 对应于 DataNode

4. HDFS 默认 Block Size

   a)32MB b)64MB c)128MB

   注:旧版本是 64MB

5. Client 端上传文件的时候下列哪项正确

   a)数据经过 NameNode 传递给 DataNode

b)Client 端将文件切分为 Block，依次上传

c)Client 只上传数据到一台 DataNode，然后由 NameNode 负责 Block 复制工作

分析：

Client 向 NameNode 发起文件写入的请求。

NameNode 根据文件大小和文件块配置情况，返回给 Client 它所管理部分 DataNode 的信息。

Client 将文件划分为多个 Block，根据 DataNode 的地址信息，按顺序写入到每一个 DataNode 块中。

6. 下面与 HDFS 类似的框架是？C

A NTFS B FAT32 C GFS D EXT3

1.1.2 集群管理

1. 下列哪项通常是集群的最主要瓶颈

a)CPU b)网络 c)磁盘 IO d)内存

2. 关于 SecondaryNameNode 哪项是正确的？

a)它是 NameNode 的热备

b)它对内存没有要求

c)它的目的是帮助 NameNode 合并编辑日志，减少 NameNode 启动时间

d)SecondaryNameNode 应与 NameNode 部署到一个节点

3. 配置机架感知的下面哪项正确

a)如果一个机架出问题，不会影响数据读写

b)写入数据的时候会写到不同机架的 DataNode 中

c)MapReduce 会根据机架获取离自己比较近的网络数据

4. 下列哪个是 Hadoop 运行的模式

a)单机版 b)伪分布式 c)分布式

5. Cloudera 提供哪几种安装 CDH 的方法

a)Cloudera manager b)Tarball c)Yum d)Rpm

1.3.1 Zookeeper 基础

1. 下面与 Zookeeper 类似的框架是？D

AProtobuf

B Java

C Kafka

D Chubby

2.1 判断题

2.1.1 集群管理

1. Ganglia 不仅可以进行监控，也可以进行告警。（正确）

2. Nagios 不可以监控 Hadoop 集群，因为它不提供 Hadoop 支持。（错误 ）

3. 如果 NameNode 意外终止，SecondaryNameNode 会接替它使集群继续工作。（错误）

4. Cloudera CDH 是需要付费使用的。（错误）

5. NameNode 负责管理 metadata，client 端每次读写请求，它都会从磁盘中读取或则会写入 metadata 信息并反馈 client 端。（错误）

6. DataNode 通过长连接与 NameNode 保持通信。错误

7. Hadoop 自身具有严格的权限管理和安全措施保障集群正常运行。（错误）

8. Slave 节点要存储数据，所以它的磁盘越大越好。（错误）

9. hadoop dfsadmin –report 命令用于检测 HDFS 损坏块。（错误）

10. Hadoop 默认调度器策略为 FIFO（错误）

11. 集群内每个节点都应该配 RAID，这样避免单磁盘损坏，影响整个节点运行。（错误）

12. Hadoop 环境变量中的 HADOOP_HEAPSIZE 用于设置所有 Hadoop 守护线程的内存。它默认是 200 GB。（错误）

13. DataNode 首次加入 cluster 的时候，如果 log 中报告不兼容文件版本，那需要 NameNode 执行Hadoopnamenode -format||操作格式化磁盘。（错误）

2.1.2 HDFS

1. Block Size 是不可以修改的。（错误）

2. Hadoop 支持数据的随机读写。（错）

3. 因为 HDFS 有多个副本，所以 NameNode 是不存在单点问题的。（错误）

2.1.3 MapReduce

1. Hadoop 是 Java 开发的，所以 MapReduce 只支持 Java 语言编写。（错误）

2. 每个 map 就是一个线程。（错误）

3. Mapreduce 的 input split 就是一个 block。（错误）