大数据工程师面试题

1. 选择题

1.1. 下面哪个程序负责 HDFS 数据存储。

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

答案 C datanode

1.2. HDfS 中的 block 默认保存几份？

a)3 份 b)2 份 c)1 份 d)不确定

答案 A 默认 3 份

1.3. 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

答案 D

1.4. HDFS 默认 Block Size

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

答案：B

1.5. 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

答案：C 磁盘

首先集群的目的是为了节省成本，用廉价的 pc 机，取代小型机及大型机。小型机和大型机有什么特点？

1.cpu 处理能力强

2.内存够大，所以集群的瓶颈不可能是 a 和 d

3.如果是互联网有瓶颈，可以让集群搭建内网。每次写入数据都要通过网络（集群是内网），然后还要写入 3 份数据，所以 IO 就会打折扣。

1.6.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

答案 C。

1.7.下列哪项可以作为集群的管理？

a)Puppet b)Pdsh c)Cloudera Manager d)Zookeeper

答案 ABD

具体可查看什么是 Zookeeper，Zookeeper 的作用是什么，在 Hadoop 及 hbase 中具体作用是什么。

1.8.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

答案 B

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。具体查看 HDFS 体系结构简介及优缺点。

1.9.下列哪个是 Hadoop 运行的模式

a)单机版 b)伪分布式 c)分布式

答案 ABC 单机版,伪分布式只是学习用的。

2. 面试题

2.1. Hadoop 的核心配置是什么？

Hadoop 的核心配置通过两个 xml 文件来完成：1，hadoop-default.xml；2，hadoop-site.xml。这些文件都使用 xml 格式，因此每个 xml 中都有一些属性，包括名称和值，但是当下这些文件都已不复存在。

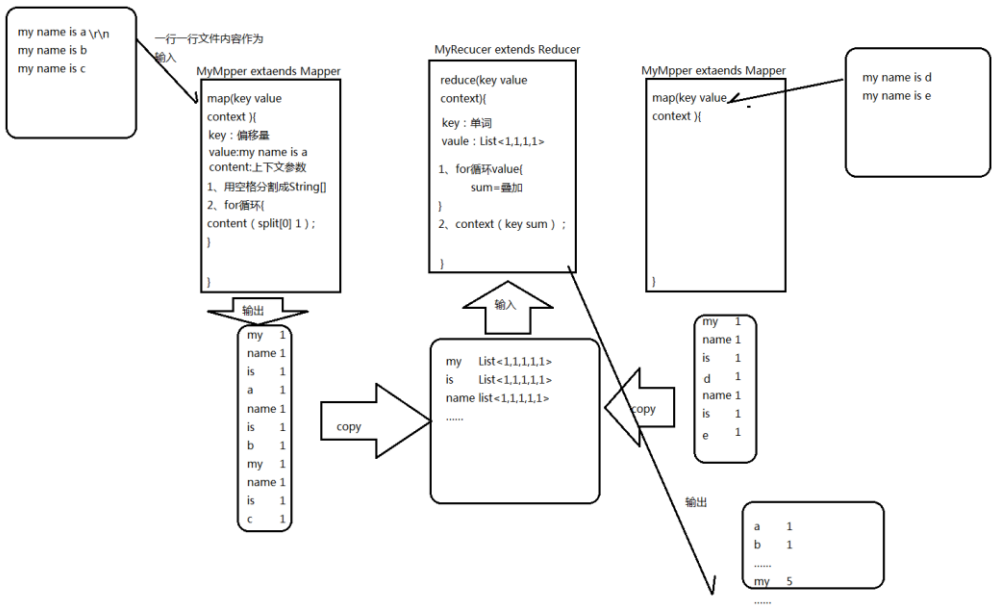
2.2. 那当下又该如何配置？

Hadoop 现在拥有 3 个配置文件：1，core-site.xml；2，hdfs-site.xml；3，mapred-site.xml。这些文件都保存在 conf/子目录下。

2.3. “jps”命令的用处？

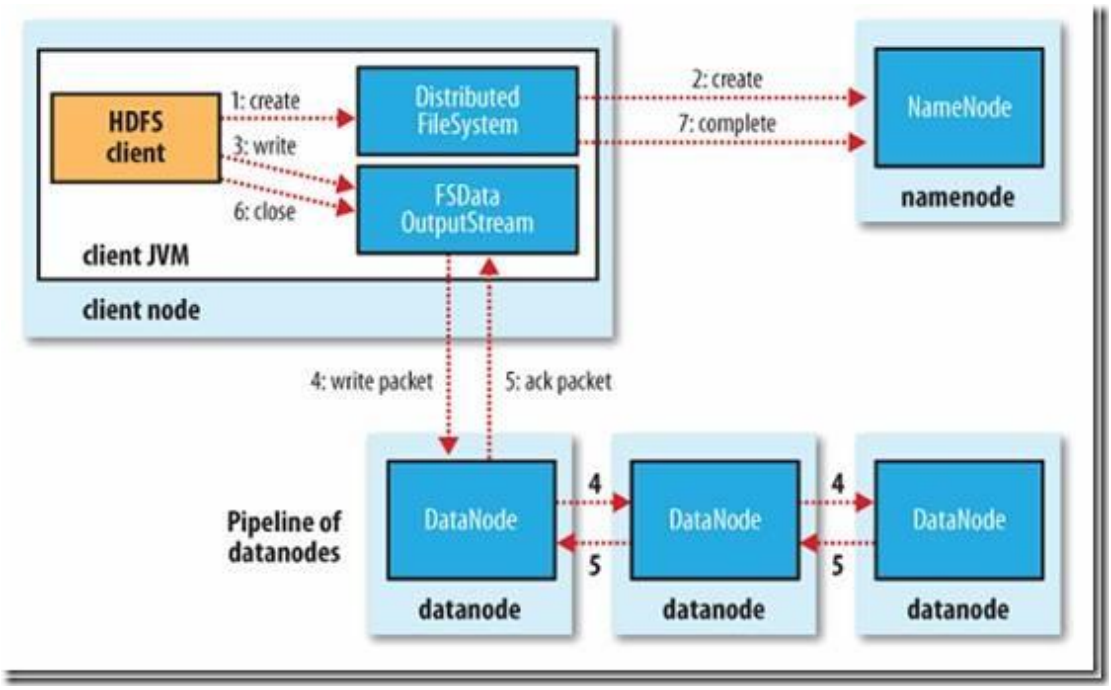
这个命令可以检查 Namenode、Datanode、Task Tracker、 Job Tracker 是否正常工作。

2.4. mapreduce 的原理?



2.5. HDFS 存储的机制?

2.5.1. hdfs 写流程



流程：

1、 client 链接 namenode 存数据

2、 namenode 记录一条数据位置信息（元数据），告诉 client 存哪。

3、 client 用 hdfs 的 api 将数据块（默认是 64M）存储到 datanode 上。

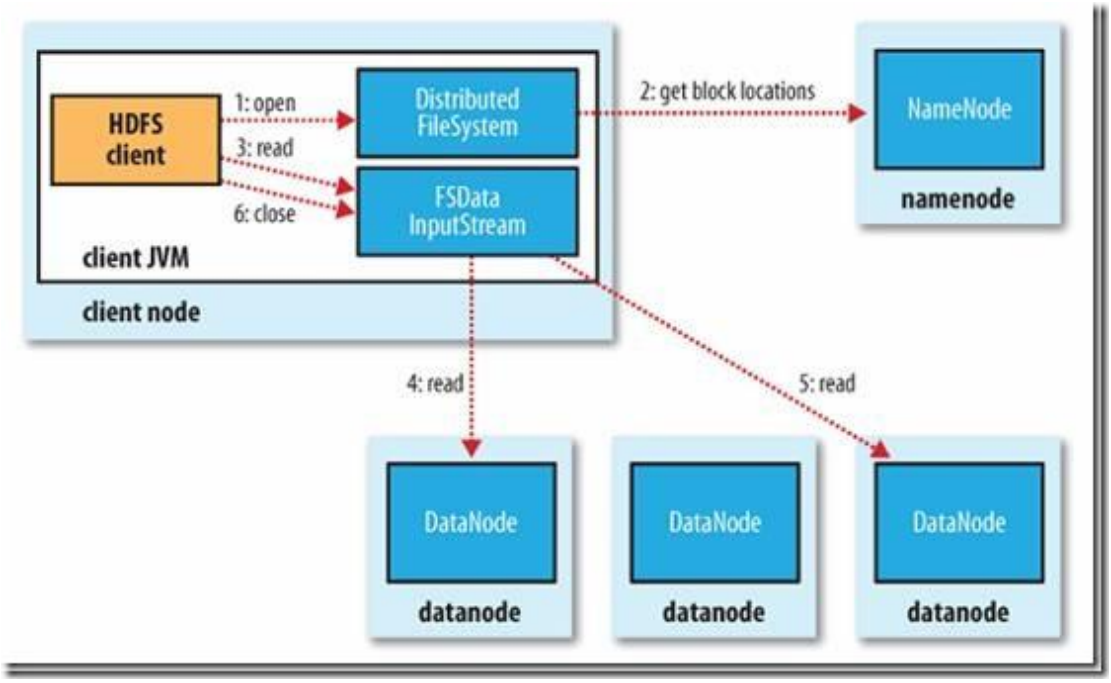
4、 datanode 将数据水平备份。并且备份完将反馈 client。

5、 client 通知 namenode 存储块完毕。

6、 namenode 将元数据同步到内存中。

7、 另一块循环上面的过程。

2.5.2. 读流程



流程：

1、 client 链接 namenode，查看元数据，找到数据的存储位置。

2、 client 通过 hdfs 的 api 并发读取数据。

3、 关闭连接。

2.6. 举一个简单的例子说明 mapreduce 是怎么来运行的 ？

wordcount 的例子

2.7. 用 mapreduce 来实现下面需求？

现在有 10 个文件夹,每个文件夹都有 1000000 个 url.现在让你找出 top1000000url。

解答：topk

(还可以用 treeMap，到 1000000 了每来一个都加进去，删掉最小的)

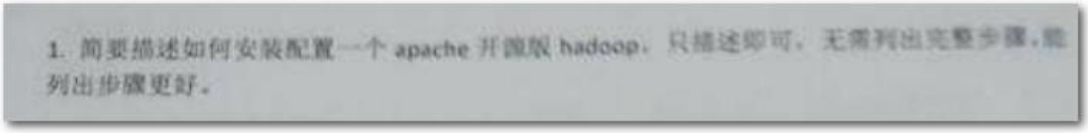
2.8. hadoop 中 Combiner 的作用?

combiner 是 reduce 的实现，在 map 端运行计算任务，减少 map 端的输出数据。

作用就是优化。

但是 combiner 的使用场景是 mapreduce 的 map 和 reduce 输入输出一样。

2.9. 简述 hadoop 安装



1.创建 hadoop 帐户。

2.setup.改 IP。

3.安装 java，并修改/etc/profile 文件，配置 java 的环境变量。

4.修改 Host 文件域名。

5.安装 SSH，配置无密钥通信。

6.解压 hadoop。

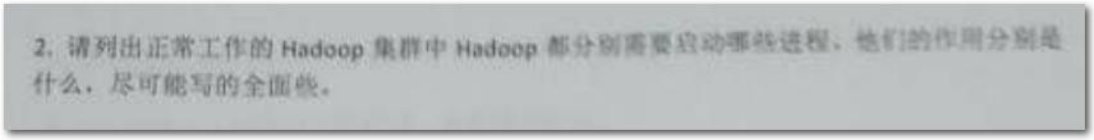
7.配置 conf 文件下 hadoop-env.sh、core-site.sh、mapre-site.sh、hdfs-site.sh。

8.配置 hadoop 的环境变量。

9.Hadoop namenode -format

10.Start-all

2.10. 请列出 hadoop 进程名



namenode：管理集群，并记录 datanode 文件信息。

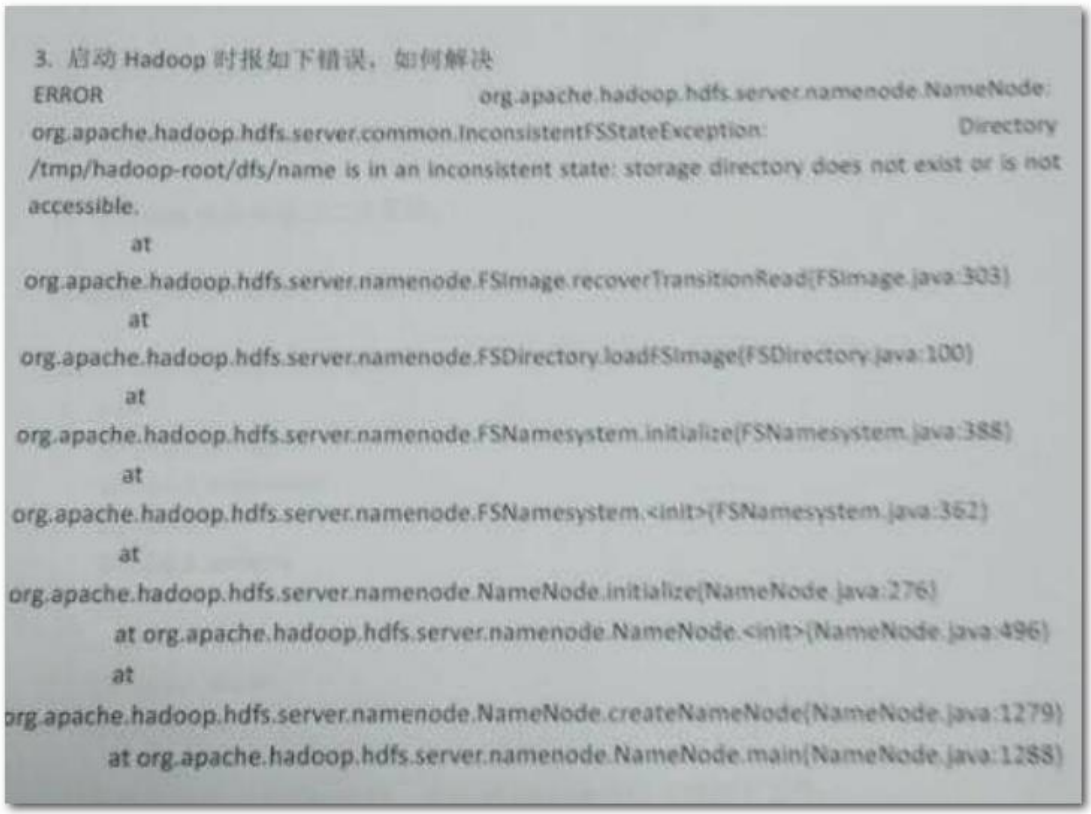
Secondname:可以做冷备，对一定范围内数据做快照性备份。

Datanode:存储数据

Jobtracker :管理任务，并将任务分配给 tasktracker。

Tasktracker:任务执行方。

2.11. 解决下面的错误

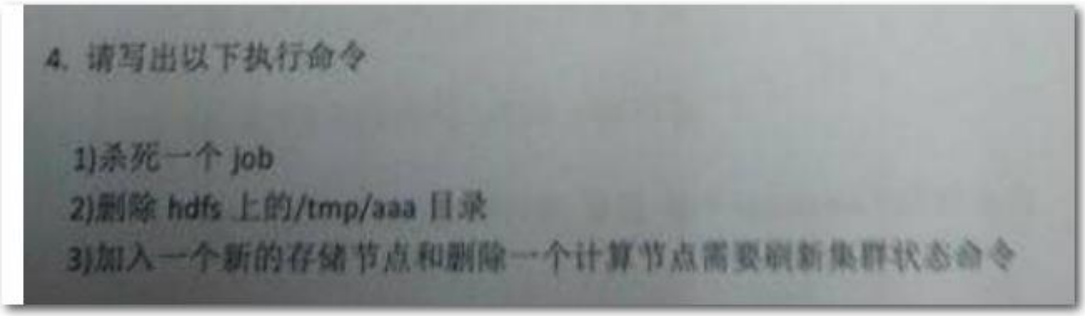


1、 权限问题，可能曾经用 root 启动过集群。(例如 hadoop 搭建的集群,是 tmp/hadoop-hadoop/.....)

2、 可能是文件夹不存在

3、 解决: 删掉 tmp 下的那个文件,或改成当前用户

2.12. 写出下面的命令



hadoop job -list　拿到 job-id　,hadoop job -kill job-id

Hadoop fs -rmr /tmp/aaa

加新节点时：

Hadoop-daemon.sh start datanode

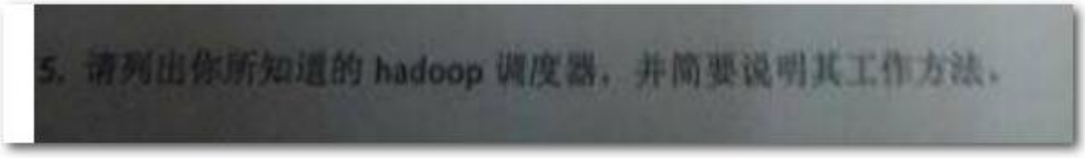
Hadoop-daemon.sh start tasktracker

删除时：

Hadoop mradmin -refreshnodes

Hadoop dfsadmin -refreshnodes

2.13. 简述 hadoop 的调度器

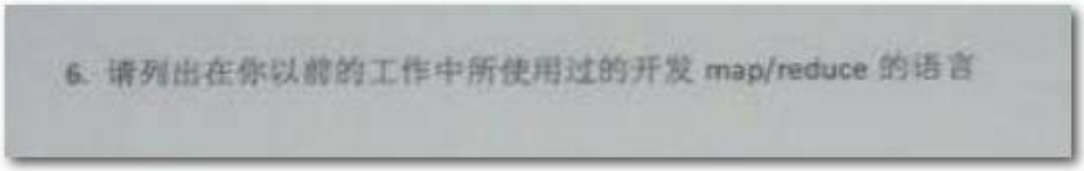


Fifo schedular :默认，先进先出的原则

Capacity schedular :计算能力调度器，选择占用最小、优先级高的先执行，依此类推。

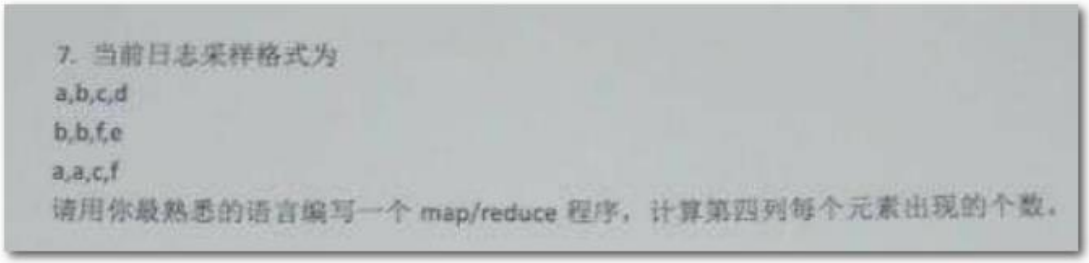
Fair schedular:公平调度，所有的 job 具有相同的资源。

2.14. 列出你开发 mapreduce 的语言



java

2.15. 书写程序

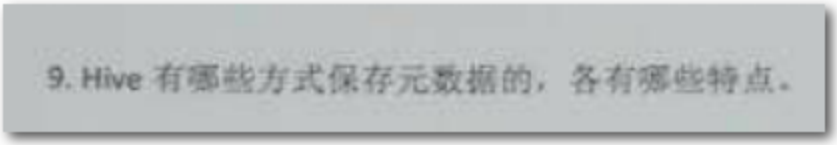


wordcount

2.16. 不同语言的优缺点

hadoop 是 java 写的，java 的集成效果最好，并且平台环境统一。

2.17. hive 有哪些保存元数据的方式，个有什么特点。



1、 内存数据库 derby，安装小，但是数据存在内存，不稳定

2、 mysql 数据库，数据存储模式可以自己设置，持久化好，查看方便。

2.18. combiner 和 partition 的作用

combiner 是 reduce 的实现，在 map 端运行计算任务，减少 map 端的输出数据。

作用就是优化。

但是 combiner 的使用场景是 mapreduce 的 map 输出结果和 reduce 输入输出一样。

partition 的默认实现是 hashpartition，是 map 端将数据按照 reduce 个数取余，进行分区，不同的 reduce 来 copy 自己的数据。

partition 的作用是将数据分到不同的 reduce 进行计算，加快计算效果。

2.19. hive 内部表和外部表的区别

内部表：加载数据到 hive 所在的 hdfs 目录，删除时，元数据和数据文件都删除

外部表：不加载数据到 hive 所在的 hdfs 目录，删除时，只删除表结构。

2.20. hbase 的 rowkey 怎么创建好？列族怎么创建比较好？

hbase 存储时，数据按照 Row key 的字典序(byte order)排序存储。设计 key 时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

一个列族在数据底层是一个文件，所以将经常一起查询的列放到一个列族中，列族尽量少，减少文件的寻址时间。

2.21. 用 mapreduce 怎么处理数据倾斜问题？

数据倾斜：map /reduce 程序执行时，reduce 节点大部分执行完毕，但是有一个或者几个 reduce 节点运行很慢，导致整个程序的处理时间很长，这是因为某一个 key 的条数比其他 key 多很多（有时是百倍或者千倍之多），这条 key 所在的 reduce 节点所处理的数据量比其他节点就大很多，从而导致某几个节点迟迟运行不完，此称之为数据倾斜。

用 hadoop 程序进行数据关联时，常碰到数据倾斜的情况，这里提供一种解决方法。

自己实现 partition 类，用 key 和 value 相加取 hash 值：

方式 1：

源代码：

```
public int getPartition(K key, V value,
                        int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
```

修改后

```
public int getPartition(K key, V value,
                        int numReduceTasks) {
    return ((key).hashCode()+value.hashCode()) &
Integer.MAX_VALUE) % numReduceTasks;
  }
```

方式 2：

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
private int aa= 0;
  /** Use {@link Object#hashCode()} to partition. */
  public int getPartition(K key, V value,
                          int numReduceTasks) {
```

```
    return (key.hashCode()+(aa++) & Integer.MAX_VALUE) %
numReduceTasks;
  }
```

2.22. hadoop 框架中怎么来优化

（1） 从应用程序角度进行优化。由于 mapreduce 是迭代逐行解析数据文件的，怎样在迭代的情况下，编写高效率的应用程序，是一种优化思路。

（2） 对 Hadoop 参数进行调优。当前 hadoop 系统有 190 多个配置参数，怎样调整这些参数，使 hadoop 作业运行尽可能的快，也是一种优化思路。

（3） 从系统实现角度进行优化。这种优化难度是最大的，它是从 hadoop 实现机制角度，发现当前 Hadoop 设计和实现上的缺点，然后进行源码级地修改。该方法虽难度大，但往往效果明显。

（4）linux 内核参数调整

2.22.1. 从应用程序角度进行优化

（1） 避免不必要的 reduce 任务

如果 mapreduce 程序中 reduce 是不必要的，那么我们可以在 map 中处理数据, Reducer 设置为 0。这样避免了多余的 reduce 任务。

（2） 为 job 添加一个 Combiner

为 job 添加一个 combiner 可以大大减少 shuffle 阶段从 map task 拷贝给远程 reduce task 的数据量。一般而言，combiner 与 reducer 相同。

（3） 根据处理数据特征使用最适合和简洁的 Writable 类型

Text 对象使用起来很方便，但它在由数值转换到文本或是由 UTF8 字符串转换到文本时都是低效的，且会消耗大量的 CPU 时间。当处理那些非文本的数据时，可以使用二进制的 Writable 类型，如 IntWritable， FloatWritable 等。二进制 writable 好处：避免文件转换的消耗；使 map task 中间结果占用更少的空间。

（4） 重用 Writable 类型

很多 MapReduce 用户常犯的一个错误是，在一个 map/reduce 方法中为每个输出都创建 Writable 对象。例如，你的 Wordcout mapper 方法可能这样写：

```
public void map(...) {
```

```
    …

    for (String word : words) {
        output.collect(new Text(word), new IntWritable(1));
    }
}
```

这样会导致程序分配出成千上万个短周期的对象。Java 垃圾收集器就要为此做很多的工作。更有效的写法是：

```
class MyMapper … {
    Text wordText = new Text();
    IntWritable one = new IntWritable(1);
    public void map(...) {
        for (String word: words) {
            wordText.set(word);
            output.collect(wordText, one);
        }
    }
}
```

（5） 使用 StringBuffer 而不是 String

当需要对字符串进行操作时，使用 StringBuffer 而不是 String，String 是 read-only 的，如果对它进行修改，会产生临时对象，而 StringBuffer 是可修改的，不会产生临时对象。

2.22.2. 对参数进行调优

查看 linux 的服务，可以关闭不必要的服务

ntsysv

停止打印服务

#/etc/init.d/cups stop

#chkconfig cups off

关闭 ipv6

#vim /etc/modprobe.conf

添加内容

alias net-pf-10 off

alias ipv6 off

调整文件最大打开数

查看： ulimit -a 结果：open files (-n) 1024

临时修改： ulimit -n 4096

持久修改：

vi /etc/security/limits.conf 在文件最后加上：

```
* soft nofile 65535
* hard nofile 65535
* soft nproc 65535
* hard nproc 65535
```

修改 linux 内核参数

vi /etc/sysctl.conf

添加

net.core.somaxconn = 32768

#web 应用中 listen 函数的 backlog 默认会给我们内核参数的 net.core.somaxconn 限制到 128，而 nginx 定义的 NGX_LISTEN_BACKLOG 默认为 511，所以有必要调整这个值。

调整 swap 分区什么时候使用：

查看：cat /proc/sys/vm/swappiness

设置：vi　/etc/sysctl.conf

在这个文档的最后加上这样一行: vm.swappiness=10

表示物理内存使用到 90%（100-10=90）的时候才使用 swap 交换区

关闭 noatime

vi /etc/fstab

/dev/sda2　/data　ext3　noatime,nodiratime　0 0

设置 readahead buffer

blockdev --setra READAHEAD 512 /dev/sda

一下是修改 mapred-site.xml 文件

修改最大槽位数

槽位数是在各个 tasktracker 上的 mapred-site.xml 上设置的，默认都是 2

<property>

<name>mapred.tasktracker.map.tasks.maximum</name>　#++++map task 的最大数

<value>2</value>

</property>

<property>

<name>mapred.tasktracker.reduce.tasks.maximum</name> #++++reducetask 的最大数

<value>2</value>

</property>

调整心跳间隔

集群规模小于 300 时，心跳间隔为 300 毫秒

mapreduce.jobtracker.heartbeat.interval.min 心跳时间

mapred.heartbeats.in.second 集群每增加多少节点，时间增加下面的值

mapreduce.jobtracker.heartbeat.scaling.factor 集群每增加上面的个数，心跳增多少

启动带外心跳

mapreduce.tasktracker.outofband.heartbeat 默认是 false

配置多块磁盘

mapreduce.local.dir

配置 RPC hander 数目

mapred.job.tracker.handler.count 默认是 10，可以改成 50，根据机器的能力

配置 HTTP 线程数目

tasktracker.http.threads 默认是 40，可以改成 100 根据机器的能力

选择合适的压缩方式

以 snappy 为例：

```
<property>
        <name>mapred.compress.map.output</name>
        <value>true</value>
    </property>
  <property>
        <name>mapred.map.output.compression.codec</name>
          <value>org.apache.hadoop.io.compress.SnappyCodec</value>
   </property>
```

启用推测执行机制

推测执行(Speculative Execution)是指在分布式集群环境下，因为程序 BUG，负载不均衡或者资源分布不均等原因，造成同一个 job 的多个 task 运行速度不一致，有的 task 运行速度明显慢于其他 task（比如：一个 job 的某个 task 进度只有 10%，而其他所有 task 已经运行完毕），则这些 task 拖慢了作业的整体执行进度，为了避免这种情况发生，Hadoop 会为该 task 启动备份任务，让该 speculative task 与原始 task 同时处理一份数据，哪个先运行完，则将谁的结果作为最终结果。

推测执行优化机制采用了典型的以空间换时间的优化策略，它同时启动多个相同 task（备份任务）处理相同的数据块，哪个完成的早，则采用哪个 task 的结果，这样可防止拖后腿 Task 任务出现，进而提高作业计算速度，但是，这样却会占用更多的资源，在集群资源紧缺的情况下，设计合理的推测执行机制可在多用少量资源情况下，减少大作业的计算时间。

mapred.map.tasks.speculative.execution 默认是 true

mapred.rduce.tasks.speculative.execution 默认是 true

设置是失败容忍度

mapred.max.map.failures.percent 作业允许失败的 map 最大比例 默认值 0，即 0%

mapred.max.reduce.failures.percent 作业允许失败的 reduce 最大比例 默认值 0，即 0%

mapred.map.max.attemps 失败后最多重新尝试的次数 默认是 4

mapred.reduce.max.attemps 失败后最多重新尝试的次数 默认是 4

启动 jvm 重用功能

mapred.job.reuse.jvm.num.tasks 默认值 1，表示只能启动一个 task，若为-1，表示可以最多运行数不限制

设置任务超时时间

mapred.task.timeout 默认值 600000 毫秒，也就是 10 分钟。

合理的控制 reduce 的启动时间

mapred.reduce.slowstart.completed.maps 默认值 0.05 表示 map 任务完成 5%时，开始启动 reduce 任务

跳过坏记录

当任务失败次数达到该值时，才会进入 skip mode，即启用跳过坏记录数功能,也就是先试几次，不行就跳过

mapred.skip.attempts.to.start.skipping 默认值 2

map 最多允许跳过的记录数

mapred.skip.map.max.skip.records 默认值 0，为不启用

reduce 最多允许跳过的记录数

mapred.skip.reduce.max.skip.records 默认值 0，为不启用

换记录存放的目录

mapred.skip.out.dir 默认值${mapred.output.dir}/_logs/

2.23. 我们开发 job 时，是否可以去掉 reduce 阶段。

可以。设置 reduce 数为 0 即可。

2.24. datanode 在什么情况下不会备份

datanode 在强制关闭或者非正常断电不会备份。

2.25. combiner 出现在那个过程

出现在 map 阶段的 map 方法后。

2.26. hdfs 的体系结构

hdfs 有 namenode、secondraynamenode、datanode 组成。

为 n+1 模式

namenode 负责管理 datanode 和记录元数据

secondraynamenode 负责合并日志

datanode 负责存储数据

2.27. 3 个 datanode 中有一个 datanode 出现错误会怎样？

这个 datanode 的数据会在其他的 datanode 上重新做备份。

2.28. 描述一下 hadoop 中，有哪些地方使用了缓存机制，作用分别是什么？

在 mapreduce 提交 job 的获取 id 之后，会将所有文件存储到分布式缓存上，这样文件可以被所有的 mapreduce 共享。

2.29. 如何确定 hadoop 集群的健康状态

通过页面监控,脚本监控。

2.30. 生产环境中为什么建议使用外部表？

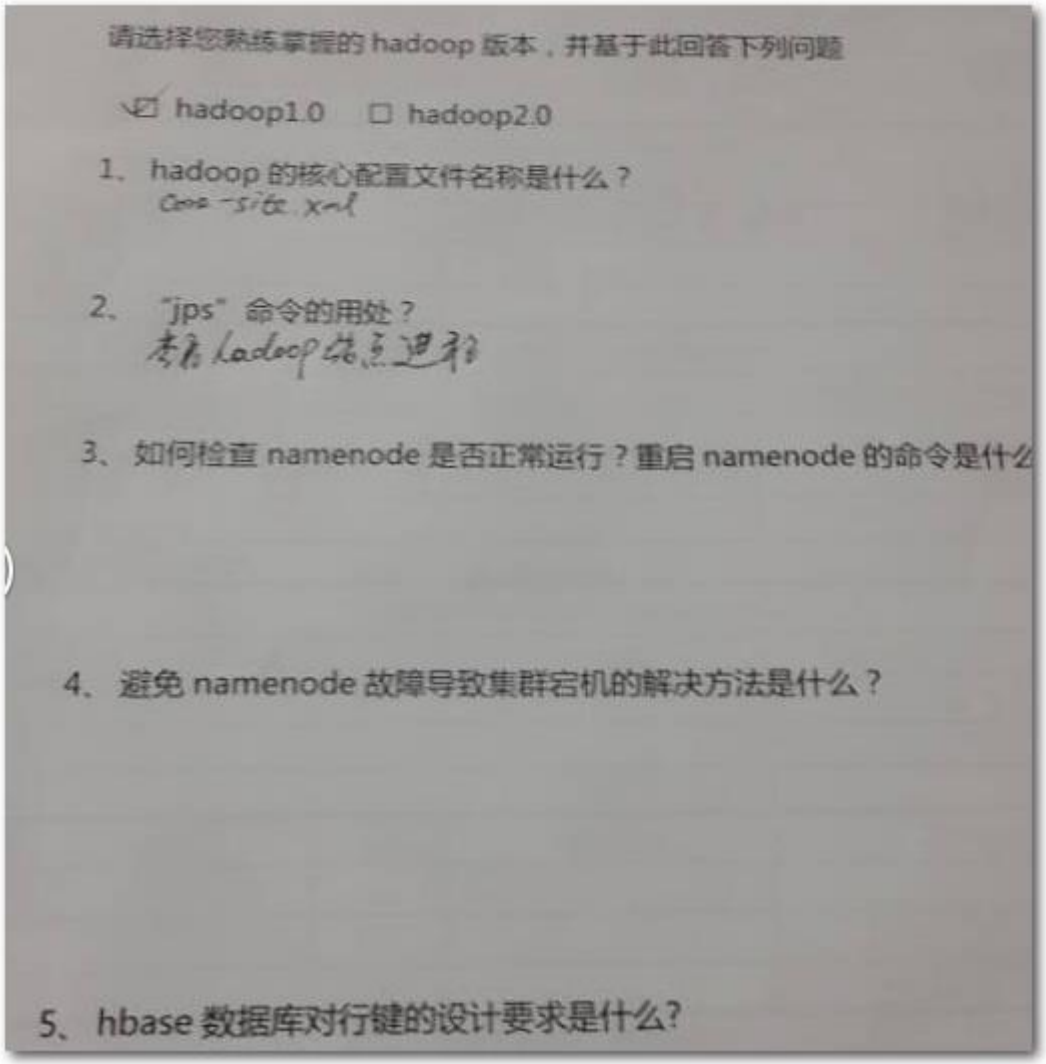
1、因为外部表不会加载数据到 hive，减少数据传输、数据还能共享。

2、hive 不会修改数据，所以无需担心数据的损坏

3、 删除表时，只删除表结构、不删除数据。

3. 15 期新增

3.1. 新增



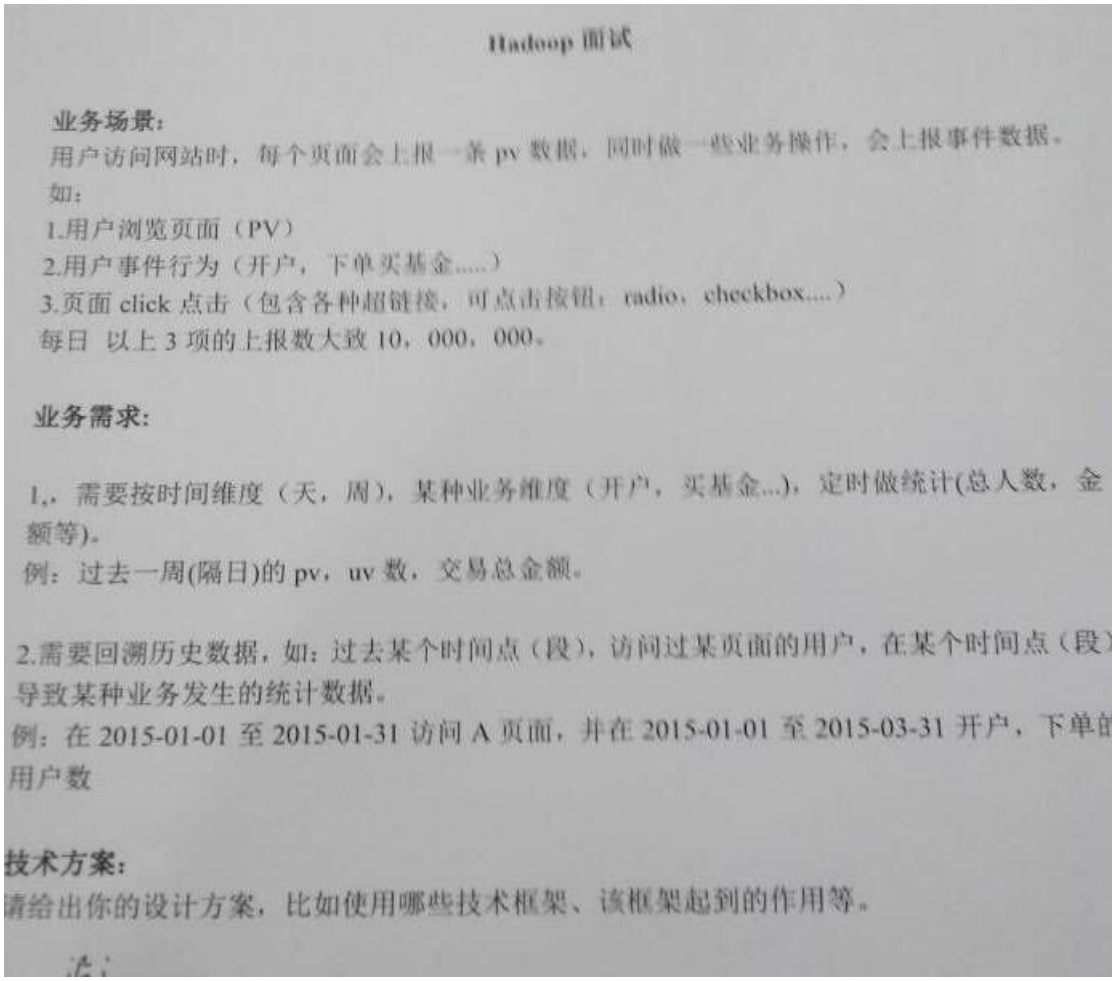
4、 通过节点信息和浏览器查看，通过脚本监控

hadoop-deamon.sh start namenode

hdfs-deamon.sh start namenode

5、 自己书写脚本监控重启

6、 行健以字典序排列，设计时充分利用这个特点，将经常一起查询的行健设计在一起，例如时间戳结尾，用户名开头（位置相关性）



Hadoop 面试

业务场景：

用户访问网站时，每个页面会上报一条 pv 数据，同时做一些业务操作，会上报事件数据。

如：

1.用户浏览页面（PV）

2.用户事件行为（开户，下单买基金.....）

3.页面 click 点击（包含各种超链接，可点击按钮：radio，checkbox....）

每日 以上 3 项的上报数大致 10，000，000。

业务需求:

1,，需要按时间维度（天，周），某种业务维度（开户，买基金...），定时做统计(总人数，金额等)。

例：过去一周(隔日)的 pv，uv 数，交易总金额。

2.需要回溯历史数据，如：过去某个时间点（段），访问过某页面的用户，在某个时间点（段）导致某种业务发生的统计数据。

例：在 2015-01-01 至 2015-01-31 访问 A 页面，并在 2015-01-01 至 2015-03-31 开户，下单的用户数

技术方案：

请给出你的设计方案，比如使用哪些技术框架、该框架起到的作用等。

1、 用 hive 分析业务数据即可

2、 将数据导入到 hive 中

sql 的设计思路：多表关联

1、 找到所有在 2015-01-01 到 2015-01-31 时间内访问 A 页面的用户

2、 在这些用户中删选在 2015-01-01 到 2015-03-31 下单的用户

3、 统计总数

3.2. 你们数据库怎么导入 hive 的,有没有出现问题

在导入 hive 的时候，如果数据库中有 blob 或者 text 字段，会报错，解决方案在 sqoop 笔记中

3.3. 公司技术选型可能利用 storm 进行实时计算,讲解一下 storm

描述下 storm 的设计模式，是基于 work、excutor、task 的方式运行代码，由 spout、bolt 组成等等

3.4. 一个 datanode 宕机,怎么一个流程恢复

将 datanode 数据删除，重新当成新节点加入即可。

3.5. Hbase 的特性,以及你怎么去设计 rowkey 和 columnFamily ,怎么去建一个 table

hbase 是列式数据库，rowkey 是字典序的，设计时的规则同上。

每个列族是一个文件，将经常一起查询的列放到同一个列族中，减少文件的寻址时间。

3.6. Redis,传统数据库,hbase,hive 每个之间的区别

redis：分布式缓存，强调缓存，内存中数据

传统数据库：注重关系

hbase：列式数据库，无法做关系数据库的主外键，用于存储海量数据，底层基于 hdfs

hive：数据仓库工具，底层是 mapreduce。不是数据库，不能用来做用户的交互存储

3.7. shuffle 阶段,你怎么理解的

shuffle 的过程说清楚，目的说清楚

3.8. Mapreduce 的 map 数量 和 reduce 数量 怎么确定 ,怎么配置

map 的数量有数据块决定，reduce 数量随便配置。

3.9. 唯一难住我的是他说实时计算,storm 如果碰上了复杂逻辑,需要算很长的时间,你怎么去优化,怎么保证实时性

3.10. Hive 你们用的是外部表还是内部表,有没有写过 UDF,hive 的版本

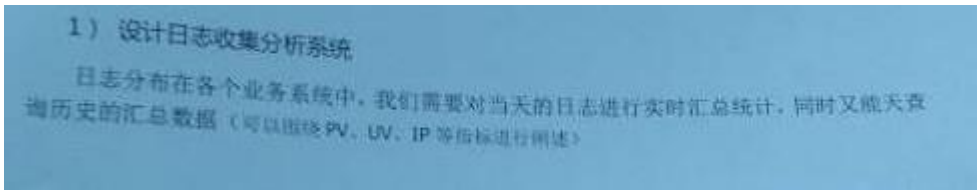
外部表和内部表的区别

3.11. Hadoop 的版本

1.04、1.20 都为稳定版，是两个常用的 hadoop1 版本。

3.12. 实时流式计算 的结果内容有哪些,你们需要统计出来么

3.13.

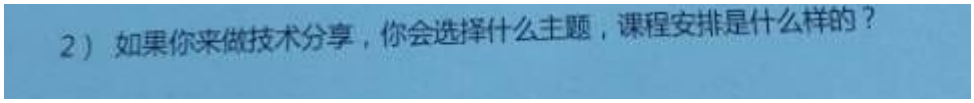


1、 通过 flume 将不同系统的日志收集到 kafka 中

2、 通过 storm 实时的处理 PV、UV、IP

3、 通过 kafka 的 consumer 将日志生产到 hbase 中。

4、 通过离线的 mapreduce 或者 hive，处理 hbase 中的数据

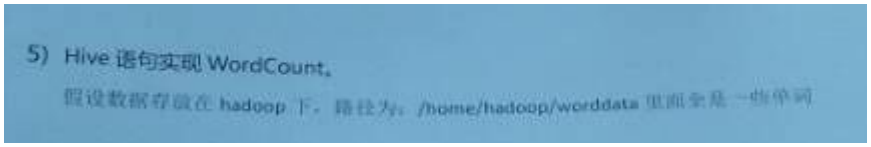


大体分为 3 个部分:

1、 离线 hadoop 技术分享（mapreduce、hive）

2、 nosql 数据库 hbase 分享

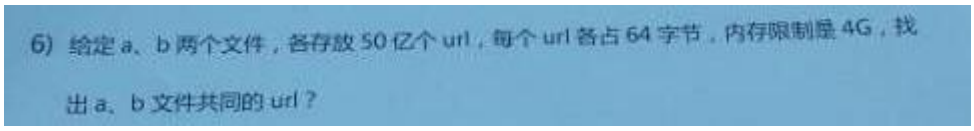
3、 实时流计算分享



1、 建表

2、 分组（group by）统计 wordcount

select word,count(1) from table1 group by word;



可以估计每个文件的大小为 50 亿×64=298G，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

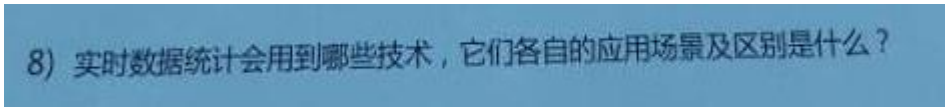
1、 将文件存储到 hdfs 中，这样每个文件为 64M 或者是 128M

2、 分别对两个文件的 url 进行去重、排序输出，这样能排除 a 文件中相同的 url，b 文件也一样

3、 对 a、b 两个文件处理后的结果进行 wordcount，并且在 reduce 中判断单词个数，个数为 2 的时候输出，这样就找到了 a、b 文件中的相同 url。

4、 此计算步骤中的每一步加载到内存中的文件大小都不会超过 64M，远远小于 4G。

topk，强调使用 treemap 是为了节省内存计算空间。



flume：日志收集系统，主要用于系统日志的收集

kafka：消息队列，进行消息的缓存和系统的解耦

storm：实时计算框架，进行流式的计算。

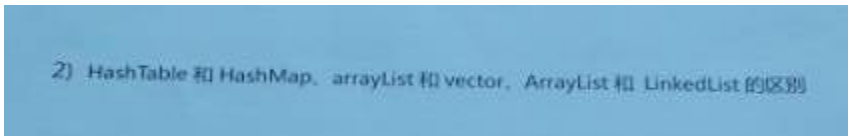
简单地说，就是一个变量和常量的关系。StringBuffer 对象的内容可以修改；而 String 对象一

旦产生后就不可以被修改，重新赋值其实是两个对象。

StringBuilder：线程非安全的

StringBuffer：线程安全的

当我们在字符串缓冲去被多个线程使用是，JVM 不能保证 StringBuilder 的操作是安全的，虽然他的速度最快，但是可以保证 StringBuffer 是可以正确操作的。当然大多数情况下就是我们是在单线程下进行的操作，所以大多数情况下是建议用 StringBuilder 而不用 StringBuffer 的，就是速度的原因。



1 HashMap 不是线程安全的

hastmap 是一个接口 是 map 接口的子接口，是将键映射到值的对象，其中键和值都是对象，并且不能包含重复键，但可以包含重复值。HashMap 允许 null key 和 null value，而 hashtable 不允许。

2 HashTable 是线程安全的一个 Collection。

HashMap 是 Hashtable 的轻量级实现（非线程安全的实现），他们都完成了 Map 接口，主要区别在于 HashMap 允许空（null）键值（key）,由于非线程安全，效率上可能高于 Hashtable。HashMap 允许将 null 作为一个 entry 的 key 或者 value，而 Hashtable 不允许。 HashMap 把 Hashtable 的 contains 方法去掉了，改成 containsvalue 和 containsKey。因为 contains 方法容易让人引起误解。 Hashtable 继承自 Dictionary 类，而 HashMap 是 Java1.2 引进的 Map interface 的一个实现。 最大的不同是，Hashtable 的方法是 Synchronize 的，而 HashMap 不是，在多个线程访问 Hashtable 时，不需要自己为它的方法实现同步，而 HashMap 就必须为之提供外同步。 Hashtable 和 HashMap 采用的 hash/rehash 算法都大概一样，所以性能不会有很大的差

```
public static void main(String args[]) { HashTable h=new HashTable(); h.put("用户 1",new Integer(90)); h.put("用户 2",new Integer(50)); h.put("用户 3",new Integer(60)); h.put("用户 4",new Integer(70)); h.put("用户 5",new Integer(80)); Enumeration e=h.elements(); while(e.hasMoreElements()){ System.out.println(e.nextElement()); }
```

总结：

hashmap	线程不安全	允许有 null 的键和值	效率高一点、	方法不是 Synchronize 的要提供外同步	有 containsvalue 和 containsKey 方法

hashtable	线程安全	不允许有 null 的键和值	效率稍低、	方法是是 Synchronize 的	有 contains 方法方法

Vector & ArrayList

1） Vector 的方法都是同步的(Synchronized),是线程安全的(thread-safe)，而 ArrayList 的方法不是，由于线程的同步必然要影响性能，因此,ArrayList 的性能比 Vector 好。

2） 当 Vector 或 ArrayList 中的元素超过它的初始大小时,Vector 会将它的容量翻倍,而 ArrayList 只增加 50%的大小，这样,ArrayList 就有利于节约内存空间。

linkedlist & ArrayList

ArrayList 采用的是数组形式来保存对象的，这种方式将对象放在连续的位置中，所以最大的缺点就是插入删除时非常麻烦

LinkedList 采用的将对象存放在独立的空间中，而且在每个空间中还保存下一个链接的索引但是缺点就是查找非常麻烦 要从第一个索引开始

Hashtable 和 HashMap 类有三个重要的不同之处。第一个不同主要是历史原因。Hashtable 是基于陈旧的 Dictionary 类的，HashMap 是 Java 1.2 引进的 Map 接口的一个实现。

也许最重要的不同是 Hashtable 的方法是同步的，而 HashMap 的方法不是。这就意味着，虽然你可以不用采取任何特殊的行为就可以在一个多线程的应用程序中用一个 Hashtable，但你必须同样地为一个 HashMap 提供外同步。一个方便的方法就是利用 Collections 类的静态的 synchronizedMap()方法，它创建一个线程安全的 Map 对象，并把它作为一个封装的对象来返回。这个对象的方法可以让你同步访问潜在的 HashMap。这么做的结果就是当你不需要同步时，你不能切断 Hashtable 中的同步（比如在一个单线程的应用程序中），而且同步增加了很多处理费用。

第三点不同是，只有 HashMap 可以让你将空值作为一个表的条目的 key 或 value。HashMap 中只有一条记录可以是一个空的 key，但任意数量的条目可以是空的 value。这就是说，如果在表中没有发现搜索键，或者如果发现了搜索键，但它是一个空的值，那么 get()将返回 null。如果有必要，用 containKey()方法来区别这两种情况。

一些资料建议，当需要同步时，用 Hashtable，反之用 HashMap。但是，因为在需要时，HashMap 可以被同步，HashMap 的功能比 Hashtable 的功能更多，而且它不是基于一个陈旧的类的，所以有人认为，在各种情况下，HashMap 都优先于 Hashtable。

关于 Properties

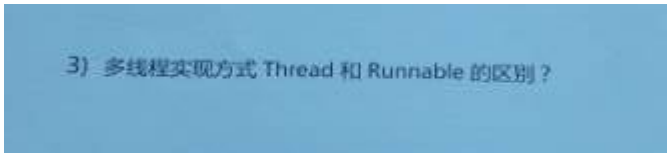
有时侯，你可能想用一个 hashtable 来映射 key 的字符串到 value 的字符串。DOS、Windows 和 Unix 中的环境字符串就有一些例子，如 key 的字符串 PATH 被映射到 value 的字符串

C:\WINDOWS;C:\WINDOWS\SYSTEM。Hashtables 是表示这些的一个简单的方法，但 Java 提供了另外一种方法。

Java.util.Properties 类是 Hashtable 的一个子类，设计用于 String keys 和 values。Properties 对象的用法同 Hashtable 的用法相象，但是类增加了两个节省时间的方法，你应该知道。

Store()方法把一个 Properties 对象的内容以一种可读的形式保存到一个文件中。Load()方法正好相反，用来读取文件，并设定 Properties 对象来包含 keys 和 values。

注意，因为 Properties 扩展了 Hashtable，你可以用超类的 put()方法来添加不是 String 对象的 keys 和 values。这是不可取的。另外，如果你将 store()用于一个不包含 String 对象的 Properties 对象，store()将失败。作为 put()和 get()的替代，你应该用 setProperty()和 getProperty()，它们用 String 参数。



在 java 中可有两种方式实现多线程，一种是继承 Thread 类，一种是实现 Runnable 接口；Thread 类是在 java.lang 包中定义的。一个类只要继承了 Thread 类同时覆写了本类中的 run()方法就可以实现多线程操作了，但是一个类只能继承一个父类，这是此方法的局限。

在 java 中可有两种方式实现多线程，一种是继承 Thread 类，一种是实现 Runnable 接口；Thread 类是在 java.lang 包中定义的。一个类只要继承了 Thread 类同时覆写了本类中的 run()方法就可以实现多线程操作了，但是一个类只能继承一个父类，这是此方法的局限。

下面看例子：

```
1. package org.thread.demo;
2. class MyThread extends Thread{
3. private String name;
4. public MyThread(String name) {
5. super();
6. this.name = name;
7. }
8. public void run(){
9. for(int i=0;i<10;i++){
10. System.out.println("线程开始: "+this.name+",i="+i);
```

```
11.}
12.}
13.}
14.package org.thread.demo;
15.public class ThreadDemo01 {
16.public static void main(String[] args) {
17.MyThread mt1=new MyThread("线程 a");
18.MyThread mt2=new MyThread("线程 b");
19.mt1.run();
20.mt2.run();
21.}
22.}
```

但是，此时结果很有规律，先第一个对象执行，然后第二个对象执行，并没有相互运行。在 JDK 的文档中可以发现，一旦调用 start()方法，则会通过 JVM 找到 run()方法。下面启动 start()方法启动线程：

```
1. package org.thread.demo;
2. public class ThreadDemo01 {
3. public static void main(String[] args) {
4. MyThread mt1=new MyThread("线程 a");
5. MyThread mt2=new MyThread("线程 b");
6. mt1.start();
7. mt2.start();
8. }
9. };
```

这样程序可以正常完成交互式运行。那么为啥非要使用 start();方法启动多线程呢？

在 JDK 的安装路径下，src.zip 是全部的 java 源程序，通过此代码找到 Thread 中的 start()方法的定义，可以发现此方法中使用了 private native void start0();其中 native 关键字表示可以调用操作系统的底层函数，那么这样的技术成为 JNI 技术（java Native Interface）

Runnable 接口

在实际开发中一个多线程的操作很少使用 Thread 类，而是通过 Runnable 接口完成。

```
1. public interface Runnable{
2. public void run();
3. }
```

例子：

```
1. package org.runnable.demo;
2. class MyThread implements Runnable{
3. private String name;
4. public MyThread(String name) {
5. this.name = name;
6. }
7. public void run(){
8. for(int i=0;i<100;i++){
9. System.out.println("线程开始: "+this.name+",i="+i);
10.}
11.}
12.};
```

但是在使用 Runnable 定义的子类中没有 start()方法，只有 Thread 类中才有。此时观察 Thread 类，有一个构造方法：public Thread(Runnable targer)此构造方法接受 Runnable 的子类实例，也就是说可以通过 Thread 类来启动 Runnable 实现的多线程。（start()可以协调系统的资源）：

```
1. package org.runnable.demo;
2. import org.runnable.demo.MyThread;
3. public class ThreadDemo01 {
4. public static void main(String[] args) {
5. MyThread mt1=new MyThread("线程 a");
6. MyThread mt2=new MyThread("线程 b");
7. new Thread(mt1).start();
8. new Thread(mt2).start();
9. }
10.}
```

两种实现方式的区别和联系：

在程序开发中只要是多线程肯定永远以实现 Runnable 接口为主，因为实现 Runnable 接口相比继承 Thread 类有如下好处：

- 避免点继承的局限，一个类可以继承多个接口。
- 适合于资源的共享

以卖票程序为例，通过 Thread 类完成：

```
1. package org.demo.dff;
2. class MyThread extends Thread{
3. private int ticket=10;
```

```
4. public void run(){
5. for(int i=0;i<20;i++){
6. if(this.ticket>0){
7. System.out.println("卖票: ticket"+this.ticket--);
8. }
9. }
10. }
11. };
```

下面通过三个线程对象，同时卖票：

```
1. package org.demo.dff;
2. public class ThreadTicket {
3. public static void main(String[] args) {
4. MyThread mt1=new MyThread();
5. MyThread mt2=new MyThread();
6. MyThread mt3=new MyThread();
7. mt1.start();//每个线程都各卖了 10 张，共卖了 30 张票
8. mt2.start();//但实际只有 10 张票，每个线程都卖自己的票
9. mt3.start();//没有达到资源共享
10. }
11. }
```

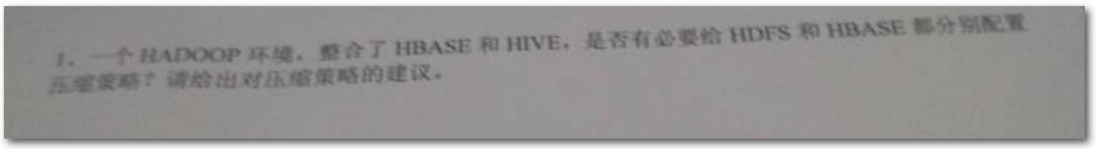
如果用 Runnable 就可以实现资源共享，下面看例子：

```
1. package org.demo.runnable;
2. class MyThread implements Runnable{
3. private int ticket=10;
4. public void run(){
5. for(int i=0;i<20;i++){
6. if(this.ticket>0){
7. System.out.println("卖票: ticket"+this.ticket--);
8. }
9. }
10. }
11. }
12.package org.demo.runnable;
13.public class RunnableTicket {
14.public static void main(String[] args) {
15.MyThread mt=new MyThread();
16.new Thread(mt).start();//同一个 mt，但是在 Thread 中就不可以，如果用同一
17.new Thread(mt).start();//个实例化对象 mt，就会出现异常
18.new Thread(mt).start();
19.}
```

```
20.};
```

虽然现在程序中有三个线程，但是一共卖了 10 张票，也就是说使用 Runnable 实现多线程可以达到资源共享目的。

3.14.



hdfs 在存储的时候不会将数据进行压缩，如果想进行压缩，我们可以在向 hdfs 上传数据的时候进行压缩。

1、 采用压缩流

```
//压缩文件
    public static void compress(String codecClassName) throws Exception{
        Class<?> codecClass = Class.forName(codecClassName);
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        CompressionCodec codec =
(CompressionCodec)ReflectionUtils.newInstance(codecClass, conf);
        //指定压缩文件路径
        FSDataOutputStream outputStream = fs.create(new
Path("/user/hadoop/text.gz"));
        //指定要被压缩的文件路径
        FSDataInputStream in = fs.open(new
Path("/user/hadoop/aa.txt"));
        //创建压缩输出流
        CompressionOutputStream out =
codec.createOutputStream(outputStream);
        IOUtils.copyBytes(in, out, conf);
        IOUtils.closeStream(in);
```

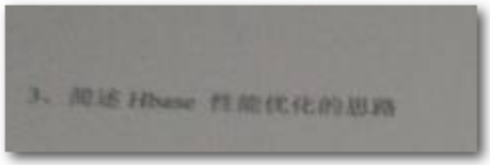
```
            IOUtils.closeStream(out);
    }
```

2、 采用序列化文件

```
    public void testSeqWrite() throws Exception {


            Configuration conf = new Configuration();// 创建配置信息
            conf.set("fs.default.name", "hdfs://master:9000");// hdfs 默认路径
            conf.set("hadoop.job.ugi", "hadoop,hadoop");// 用户和组信息
            String uriin = "hdfs://master:9000/ceshi2/";// 文件路径
            FileSystem fs = FileSystem.get(URI.create(uriin), conf);// 创建 filesystem
            Path path = new Path("hdfs://master:9000/ceshi3/test.seq");// 文件名
            IntWritable k = new IntWritable();// key，相当于 int
            Text v = new Text();// value，相当于 String
            SequenceFile.Writer w = SequenceFile.createWriter(fs, conf, path,
                    k.getClass(), v.getClass());// 创建 writer
            for (int i = 1; i < 100; i++) {// 循环添加
                k.set(i);
                v.set("abcd");
                w.append(k, v);
            }
            w.close();
            IOUtils.closeStream(w);// 关闭的时候 flush
            fs.close();
    }
```

hbase 为列存数据库，本身存在压缩机制，所以无需设计。



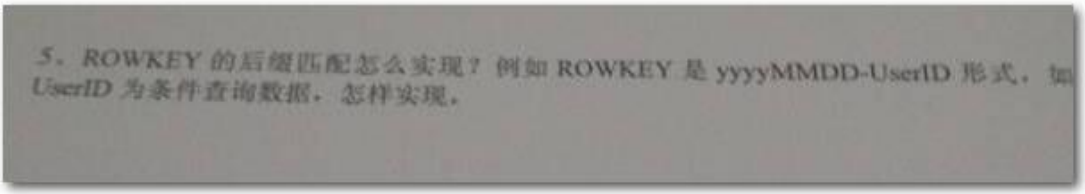
1、 在库表设计的时候，尽量考虑 rowkey 和 columnfamily 的特性

2、 进行 hbase 集群的调优：见 hbase 调优

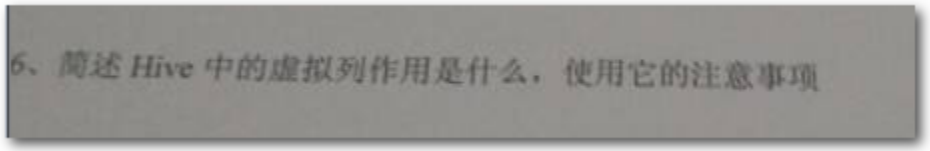
hbase 的 filter 是通过 scan 设置的，所以是基于 scan 的查询结果进行过滤。

1、 在进行订单开发的时候，我们使用 rowkeyfilter 过滤出某个用户的所有订单

2、 在进行云笔记开发时，我们使用 rowkey 过滤器进行 redis 数据的恢复。



使用 rowkey 过滤器实现



Hive 提供了三个虚拟列：

INPUT__FILE__NAME

BLOCK__OFFSET__INSIDE__FILE

ROW__OFFSET__INSIDE__BLOCK

但 ROW__OFFSET__INSIDE__BLOCK 默认是不可用的，需要设置 hive.exec.rowoffset 为 true 才可以。可以用来排查有问题的输入数据。

INPUT__FILE__NAME, mapper 任务的输出文件名。

BLOCK__OFFSET__INSIDE__FILE, 当前全局文件的偏移量。对于块压缩文件，就是当前块的文件偏移量，即当前块的第一个字节在文件中的偏移量。

hive> SELECT INPUT__FILE__NAME, BLOCK__OFFSET__INSIDE__FILE, line

> FROM hive_text WHERE line LIKE '%hive%' LIMIT 2;

har://file/user/hive/warehouse/hive_text/folder=docs/

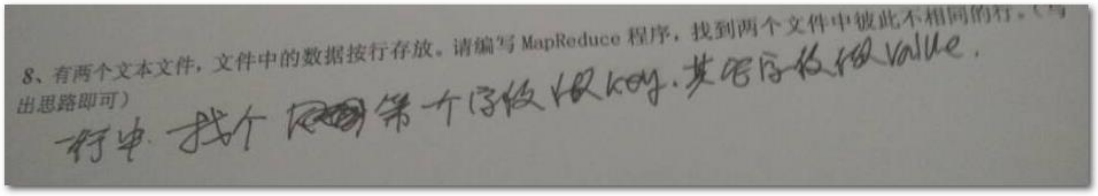
data.har/user/hive/warehouse/hive_text/folder=docs/README.txt 2243

har://file/user/hive/warehouse/hive_text/folder=docs/

data.har/user/hive/warehouse/hive_text/folder=docs/README.txt 3646

1、 将小文件打成 har 文件存储

2、 将小文件序列化到 hdfs 中



写个 mapreduce 链　用依赖关系，一共三个 mapreduce，第一个处理第一个文件，第二个处理第二个文件，第三个处理前两个的输出结果，第一个 mapreduce 将文件去重，第二个 mapreduce 也将文件去重，第三个做 wordcount，wordcount 为 1 的结果就是不同的

4. 共同朋友

思路：例如 A，他的朋友是 B\C\D\E\F\，那么 BC 的共同朋友就是 A。所以将 BC 作为 key，将 A 作为 value，在 map 端输出即可！其他的朋友循环处理。

```
import java.io.IOException;

import java.util.Set;

import java.util.StringTokenizer;

import java.util.TreeSet;



import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;
```

```
import org.apache.hadoop.io.Text;
9.
import org.apache.hadoop.mapreduce.Job;
10. import org.apache.hadoop.mapreduce.Mapper;
11. import org.apache.hadoop.mapreduce.Reducer;
12. import org.apache.hadoop.mapreduce.Mapper.Context;
13. import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
14. import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
15. import org.apache.hadoop.util.GenericOptionsParser;
16.
17. public class FindFriend {
www.aboutyun.com
18.
19.                 public static class ChangeMapper extends Mapper<Object, Text, Text,
Text>{
20.                         @Override
21.                         public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
22.                                                 StringTokenizer itr = new
StringTokenizer(value.toString());
23.                                 Text owner = new Text();
24.                                 Set<String> set = new TreeSet<String>();
25.                                 owner.set(itr.nextToken());
26.                                 while (itr.hasMoreTokens()) {
27.                                         set.add(itr.nextToken());
28.                                 }
29.                                 String[] friends = new String[set.size()];
30.                                 friends = set.toArray(friends);
```

```
31.
32.                                     for(int i=0;i<friends.length;i++){
33.                                         for(int j=i+1;j<friends.length;j++){
34.                                                                         String outputkey = friends[i]+friends[j];
35.                                                                         context.write(new Text(outputkey),owner);
36.                                         }
37.                                     }
38.                             }
39.                 }
40.
41.                 public static class FindReducer extends Reducer<Text,Text,Text,Text>
{
42.                                 public void reduce(Text key, Iterable<Text> values,
43.                                                 Context context) throws IOException, InterruptedException {
44.                                         String    commonfriends ="";
www.aboutyun.com
45.                                     for (Text val : values) {
46.                                         if(commonfriends == ""){
47.                                                 commonfriends = val.toString();
48.                                         }else{
49.                                                 commonfriends = commonfriends+":"+val.toString();
50.                                         }
51.                                       }
52.                                     context.write(key, new
```

```
Text(commonfriends));
53.                                     }
54.                 }
55.
56.
57.             public static void main(String[] args) throws IOException,
58.             InterruptedException, ClassNotFoundException {
59.
60.                 Configuration conf = new Configuration();
61.                                     String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
62.                 if (otherArgs.length < 2) {
63.                     System.err.println("args error");
64.                     System.exit(2);
65.                 }
66.                 Job job = new Job(conf, "word count");
67.                 job.setJarByClass(FindFriend.class);
68.                 job.setMapperClass(ChangeMapper.class);
69.                 job.setCombinerClass(FindReducer.class);
70.                 job.setReducerClass(FindReducer.class);
71.                 job.setOutputKeyClass(Text.class);
72.                 job.setOutputValueClass(Text.class);
73.                 for (int i = 0; i < otherArgs.length - 1; ++i) {
www.aboutyun.com
74.                     FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
75.                 }
76.                 FileOutputFormat.setOutputPath(job,
77.                     new Path(otherArgs[otherArgs.length - 1]));
```

```
78.                     System.exit(job.waitForCompletion(true) ? 0 : 1);
79.
80.                 }
81.
82. }
```

结果：

```
1. AB          E:C:D
2. AC          E:B
3. AD          B:E
4. AE          C:B:D
5. BC          A:E
6. BD          A:E
7. BE          C:D:A
8. BF          A
9. CD          E:A:B
10. CE          A:B
11. CF          A
12. DE          B:A
13. DF          A
14. EF          A
```

5. 基站逗留时间

1 使用 Hive 或者自定义 MR 实现如下逻辑

product_no	lac_id	moment	start_time	user_id	county_id	staytime	city_id
13429100031	22554	8	2013-03-11 08:55:19.151754088	571	571	282	571
13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	103	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100140	26642	9	2013-03-11 09:02:19.151754088	571	571	18	571
13429100082	22691	8	2013-03-11 08:57:32.151754088	571	571	287	571

字段解释：

product_no：用户手机号；

lac_id：用户所在基站；

start_time：用户在此基站的开始时间；

staytime：用户在此基站的逗留时间。

需求：

根据 lac_id 和 start_time 知道用户当时的位置，根据 staytime 知道用户各个基站的逗留时长。根据轨迹合并连续基站的 staytime。

最终得到每一个用户按时间排序在每一个基站驻留时长

期望：

期望输出举例：

13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	103	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571

思路：

将数据导入 hive 表中，查询时，用电话号码和时间排序即可！

6. 脚本替换

2.1 请随意使用各种类型的脚本语言实现：批量将指定目录下的所有文件中的$HADOOP_HOME$替换成/home/ocetl/app/hadoop

脚本：随意命名为 aaa.sh

```
#!/bin/bash
ls $1 | while read line
do
sed -i 's,\$HADOOP_HOME\$,\/home\/aa,g' $1$line
echo $1$line
done
```

脚本执行命令：替换/home/hadoop/test/下的所有文件

./aaa.sh /home/hadoop/test/

7. 一键执行

脚本：

vi runRemoteCmd.sh

```
#!/bin/bash
$1
ssh -q hadoop@slave1 "$1"
ssh -q hadoop@slave2 "$1"
```

执行命令

```
./runRemoteCmd.sh "ls -l"
```

8. 大数据面试汇总

1. 讲解一下 MapReduce 的一些基本流程

   任务提交流程，任务运行流程

2. 你们数据库怎么导入 hive 的,有没有出现问题

   使用 sqoop 导入，我们公司的数据库中设计了 text 字段，导致导入的时候出现了缓存不够的情况（见云笔记），开始解决起来感觉很棘手，后来查看了 sqoop 的文档，加上了 limit 属性，解决了。

3. 公司技术选型可能利用 storm 进行实时计算,讲解一下 storm

   从 storm 的应用，代码书写，运行机制讲

4. 问你 java 集合类的数据结构,比如 hashmap

   看 java 面试宝典

5. 问你知不知道 concurrent 包下的东西,例如 concurrenthashmap

   看 java 面试宝典

6. 公司最近主要在自然语言学习去开发,有没有接触过

   没有用过

9. 面试问题:

1. 从前到后从你教育背景(学过哪些课)到各个项目你负责的模块,问的很细(本以为他是物理学博士,但是所有的技术都懂)

2. hadoop 的 namenode 宕机,怎么解决

   先分析宕机后的损失，宕机后直接导致 client 无法访问，内存中的元数据丢失，但是硬盘中的元数据应该还存在，如果只是节点挂了，重启即可，如果是机器挂了，重启机器后看节点是否能重启，不能重启就要找到原因修复了。但是最终的解决方案应该是在设计集群的初期就考虑到这个问题，做 namenode 的 HA。

3. 一个 datanode 宕机,怎么一个流程恢复

   Datanode 宕机了后，如果是短暂的宕机，可以实现写好脚本监控，将它启动起来。如果是长时间宕机了，那么 datanode 上的数据应该已经被备份到其他机器了，那这台 datanode 就是一台新的 datanode 了，删除他的所有数据文件和状态文件，重新启动。

4. Hbase 的特性,以及你怎么去设计 rowkey 和 columnFamily ,怎么去建一个 table

   因为 hbase 是列式数据库，列非表 schema 的一部分，所以在设计初期只需要考虑 rowkey 和 columnFamily 即可，rowkey 有位置相关性，所以如果数据是练习查询的，最好对同类数据加一个前缀，而每个 columnFamily 实际上在底层是一个文件，那么文件越小，查询越快，所以讲经常一起查询的列设计到一个列簇，但是列簇不宜过多。

5. Redis,传统数据库,hbase,hive 每个之间的区别(问的非常细)

   Redis 是缓存，围绕着内存和缓存说

   Hbase 是列式数据库，存在 hdfs 上，围绕着数据量来说

   Hive 是数据仓库，是用来分析数据的，不是增删改查数据的。

6. 公司之后倾向用 spark 开发,你会么(就用 java 代码去写)

   会，spark 使用 scala 开发的，在 scala 中可以随意使用 jdk 的类库，可以用 java 开发，但是最好用原生的 scala 开发，兼容性好，scala 更灵活。

10. 面试问题:

1. 笔试: java 基础(基本全忘,做的很烂,复习大数据连单例都忘了怎么写)

   复习 java 面试宝典

2. 开始介绍项目,直接用大数据项目介绍,项目经理也懂大数据

3. Mapreduce 一些流程,经过哪些步骤

   Map—combiner—partition—sort—copy—sort—grouping—reduce

4. 说下对 hadoop 的一些理解,包括哪些组件

   详谈 hadoop 的应用，包括的组件分为三类，分别说明 hdfs，yarn，mapreduce

5. 详细讲解下你流式实时计算的项目部署以及收集的结果情况

   讲解 storm 集群的部署方案，项目的大小，使用的 worker 数，数据收集在 hbase 或者 hdfs，好处是什么

6. 你的数据库是不是很大么,有没有分表,分区,你是怎么实现的

   数据库的分表在设计初期是按照月份进行拆分的，不同的月份查询不同的表。分区没弄过。

7. 开始问 java 的一些东西(从各种框架原理到各种复杂 SQL)

8. 多线程,并发,垃圾回收机制,数据结构(问这些,基本觉得看你是不是高级程序员了)

   多线程要知道操作方式，线程安全的锁，并且要知道 lock 锁

   垃圾回收机制需要详细了解（见云笔记），主要从内存划分，垃圾回收主要的工作区域，垃圾回收器的种类，各有什么优缺点，用在哪里合适。

   数据结构基本的要知道，复杂的参考相关的书籍。

11. 面试问题:

1. BI 小组的 3 个年轻学生一起技术面试(一个是南开博士

2. 数据量多少,集群规模多大,型号

   一般中型的电商或者互联网企业，日志量每天在 200-500M 左右，集群规模在 30-50 台左右，机器一般为 dell 的 2000 左右的服务器，型号不定

   大型的互联网公司据网上资料显示，日志量在 GP-PB 不等，集群规模在 500-4000 不等，甚至更多，机器型号不确定。

3. 项目,mapreduce

   介绍整个 mapreduce 项目流程，数据采集—数据聚合—数据分析—数据展示等

4. 实时流式计算框架,几个人,多长时间,细节问题,包括讲 flume ,kafka ,storm 的各个的组件组成,你负责那一块,如果需要你搭建你可以完成么?

5. 你觉得 spark 可以完全替代 hadoop 么?

12. 面试问题:

1. 一些传统的 hadoop 问题,mapreduce 他就问 shuffle 阶段,你怎么理解的

   Shuffle 意义在于将不同 map 处理后的数据进行合理分配，让 reduce 处理，从而产生了排序、分区。

2. Mapreduce 的 map 数量 和 reduce 数量 怎么确定 ,怎么配置

   Map 无法配置，reduce 随便配置

3. 唯一难住我的是他说实时计算,storm 如果碰上了复杂逻辑,需要算很长的时间,你怎么去优化

   拆分复杂的业务到多个 bolt 中，这样可以利用 bolt 的 tree 将速度提升

4. Hive 你们用的是外部表还是内部表,有没有写过 UDF(当然吹自己写过了),hive 的版本

   外部表，udf，udaf 等，hive 版本为 1.0

5. Hadoop 的版本

   如果是 1.0 版本就说 1.2，如果是 2.0 版本，就说 2.6 或者 2.7

   1.2 为官方稳定版本，2.7 为官方稳定版本。

   Apache Hadoop 2.7.1 于美国时间 2015 年 07 月 06 日正式发布，本版本属于稳定版本，是自 Hadoop 2.6.0 以来又一个稳定版，同时也是 Hadoop 2.7.x 版本线的第一个稳定版本，也是 2.7 版本线的维护版本，变化不大，主要是修复了一些比较严重的 Bug

6. 实时流式计算的结果内容有哪些,你们需要统计出来么(我就说 highchart 展示)

   简单介绍日志监控、风控等结果内容，统计出来显示在报表或者邮件中。

7. 开始问 java 相关,包括 luecne,solr(倒排索引的原理),框架呀,redis 呀

13. 京东商城 - 大数据

（1）Java 篇

1、JVM，GC（算法，新生代，老年代），JVM 结构

2、hashcode，hashMap，list，hashSet，equals（结构原理），A extends B（类的加载顺序）

1.父类静态代码块；

2.子类静态代码块；

3.父类非静态代码块；

4.父类构造函数；

5.子类非静态代码块；

6.子类构造函数；

3、多线程，主线程，次线程，唤醒，睡眠

略

4、常见算法：冒泡算法，排序算法，二分查找，时间复杂度

略

（2）Flume 篇

1、数据怎么采集到 Kafka，实现方式

使用官方提供的 flumeKafka 插件，插件的实现方式是自定义了 flume 的 sink，将数据从 channle 中取出，通过 kafka 的 producer 写入到 kafka 中，可以自定义分区等。

2、flume 管道内存，flume 宕机了数据丢失怎么解决

1、Flume 的 channel 分为很多种，可以将数据写入到文件

2、防止非首个 agent 宕机的方法数可以做集群或者主备

3、flume 配置方式，flume 集群（问的很详细）

Flume 的配置围绕着 source、channel、sink 叙述，flume 的集群是做在 agent 上的，而非机器上。

4、flume 不采集 Nginx 日志，通过 Logger4j 采集日志，优缺点是什么？

优点：Nginx 的日志格式是固定的，但是缺少 sessionid，通过 logger4j 采集的日志是带有 sessionid 的，而 session 可以通过 redis 共享，保证了集群日志中的同一 session 落到不同的 tomcat 时，sessionId 还是一样的，而且 logger4j 的方式比较稳定，不会宕机。

缺点：不够灵活，logger4j 的方式和项目结合过于紧密，而 flume 的方式比较灵活，拔插式比较好，不会影响项目性能。

5、flume 和 kafka 采集日志区别，采集日志时中间停了，怎么记录之前的日志。

Flume 采集日志是通过流的方式直接将日志收集到存储层 ,而 kafka 试讲日志缓存在 kafka 集群，待后期可以采集到存储层。

Flume 采集中间停了，可以采用文件的方式记录之前的日志，而 kafka 是采用 offset 的方式记录之前的日志。

（3）Kafka 篇

1、容错机制

分区备份，存在主备 partition

2、同一 topic 不同 partition 分区

？？？？

3、kafka 数据流向

Producer → leader partition → follower partition(半数以上) →consumer

4、kafka+spark-streaming 结合丢数据怎么解决？

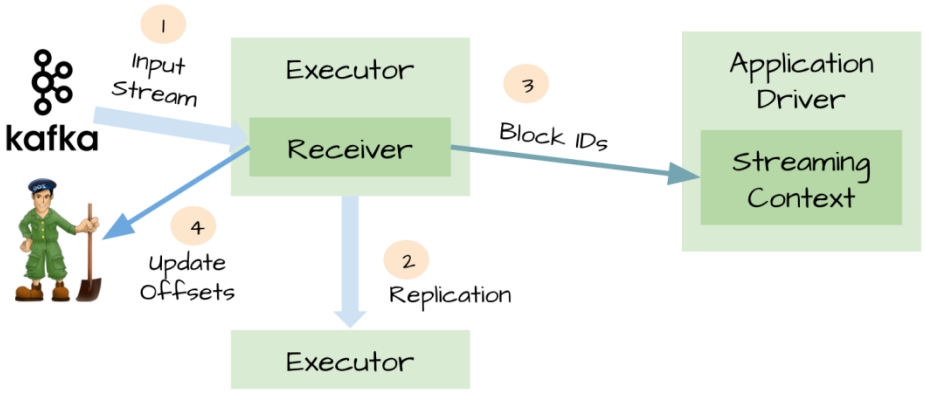
spark streaming 从 1.2 开始提供了数据的零丢失，想享受这个特性，需要满足如下条件：

1. 数据输入需要可靠的 sources 和可靠的 receivers
2. 应用 metadata 必须通过应用 driver checkpoint

3. WAL（write ahead log）

13.1. 可靠的 sources 和 receivers

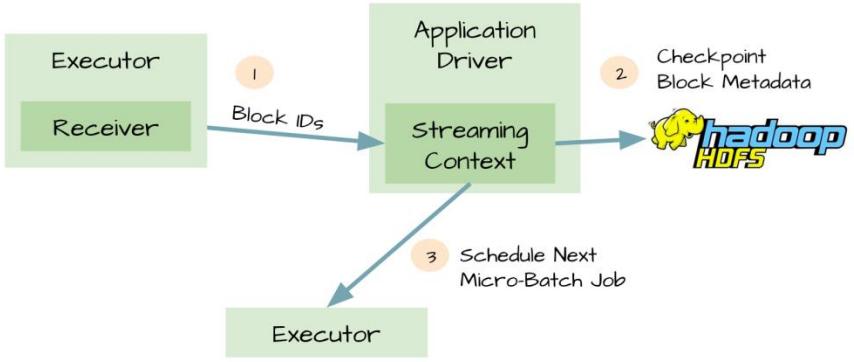
spark streaming 可以通过多种方式作为数据 sources（包括 kafka），输入数据通过 receivers 接收，通过 replication 存储于 spark 中（为了 faultolerance，默认复制到两个 spark executors），如果数据复制完成，receivers 可以知道（例如 kafka 中更新 offsets 到 zookeeper 中）。这样当 receivers 在接收数据过程中 crash 掉，不会有数据丢失，receivers 没有复制的数据，当 receiver 恢复后重新接收。



13.2. metadata checkpoint

可靠的 sources 和 receivers，可以使数据在 receivers 失败后恢复，然而在 driver 失败后恢复是比较复杂的，一种方法是通过 checkpoint metadata 到 HDFS 或者 S3。metadata 包括：

- configuration
- code
- 一些排队等待处理但没有完成的 RDD（仅仅是 metadata，而不是 data）



这样当 driver 失败时，可以通过 metadata checkpoint，重构应用程序并知道执行到那个地方。

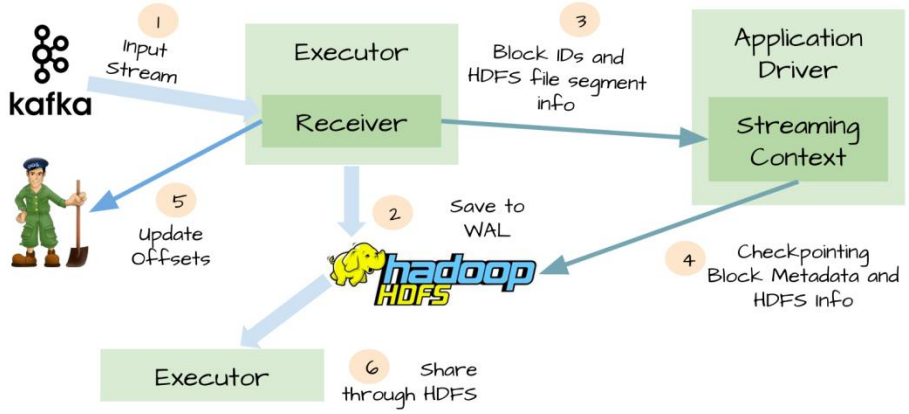
13.3. 数据可能丢失的场景

可靠的 sources 和 receivers，以及 metadata checkpoint 也不可以保证数据的不丢失，例如：

- 两个 executor 得到计算数据，并保存在他们的内存中
- receivers 知道数据已经输入
- executors 开始计算数据
- driver 突然失败
- driver 失败，那么 executors 都会被 kill 掉
- 因为 executor 被 kill 掉，那么他们内存中得数据都会丢失，但是这些数据不再被处理
- executor 中的数据不可恢复

13.4. WAL

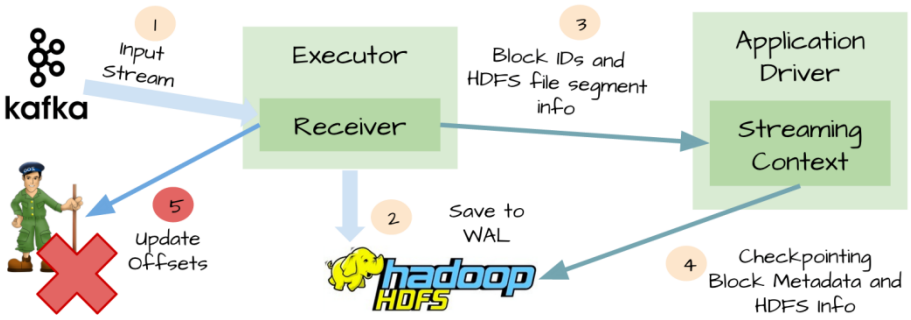
为了避免上面情景的出现，spark streaming 1.2 引入了 WAL。所有接收的数据通过 receivers 写入 HDFS 或者 S3 中 checkpoint 目录，这样当 driver 失败后，executor 中数据丢失后，可以通过 checkpoint 恢复。



13.5. At-Least-Once

尽管 WAL 可以保证数据零丢失，但是不能保证 exactly-once，例如下面场景：

- Receivers 接收完数据并保存到 HDFS 或 S3
- 在更新 offset 前，receivers 失败了



- Spark Streaming 以为数据接收成功，但是 Kafka 以为数据没有接收成功，因为 offset 没有更新到 zookeeper
- 随后 receiver 恢复了
- 从 WAL 可以读取的数据重新消费一次，因为使用的 kafka High-Level 消费 API，从 zookeeper 中保存的 offsets 开始消费

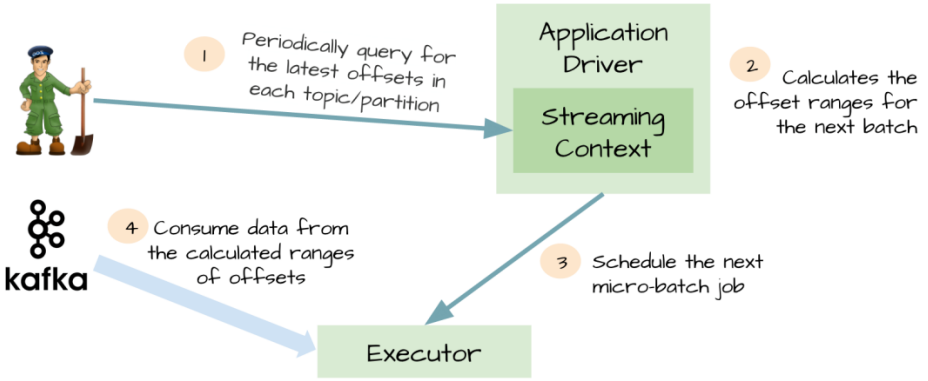
13.6. WAL 的缺点

通过上面描述，WAL 有两个缺点：

- 降低了 receivers 的性能，因为数据还要存储到 HDFS 等分布式文件系统
- 对于一些 resources，可能存在重复的数据，比如 Kafka，在 Kafka 中存在一份数据，在 Spark Streaming 也存在一份（以 WAL 的形式存储在 hadoop API 兼容的文件系统中）

13.7. Kafka direct API

为了 WAL 的性能损失和 exactly-once，spark streaming1.3 中使用 Kafka direct API。非常巧妙，Spark driver 计算下个 batch 的 offsets，指导 executor 消费对应的 topics 和 partitions。消费 Kafka 消息，就像消费文件系统文件一样。



1. 不再需要 kafka receivers，executor 直接通过 Kafka API 消费数据

2. WAL 不再需要，如果从失败恢复，可以重新消费
3. exactly-once 得到了保证，不会再从 WAL 中重复读取数据

13.8. 总结

主要说的是 spark streaming 通过各种方式来保证数据不丢失，并保证 exactly-once，每个版本都是 spark streaming 越来越稳定，越来越向生产环境使用发展。

5、kafka 中存储目录 data/dir.....topic1 和 topic2 怎么存储的，存储结构，data.....目录下有多少个分区，每个分区的存储格式是什么样的？

1、topic 是按照“主题名-分区”存储的

2、分区个数由配置文件决定

3、每个分区下最重要的两个文件是 0000000000.log 和 000000.index ,0000000.log 以默认 1G 大小回滚。

（4）Hive 篇

1、hive partition 分区

分区表，动态分区

2、insert into 和 override write 区别？

insert into：将某一张表中的数据写到另一张表中

override write：覆盖之前的内容。

3、假如一个分区的数据主部错误怎么通过 hivesql 删除 hdfs

alter table ptable drop partition (daytime='20140911',city='bj');

元数据，数据文件都删除，但目录 daytime= 20140911 还在

（5）Storm 篇

1、开发流程，容错机制

开发流程：

1、写主类（设计 spout 和 bolt 的分发机制）

2、写 spout 收集数据

3、写 bolt 处理数据，根据数据量和业务的复杂程度，设计并行度。

容错机制：采用 ack 和 fail 进行容错，失败的数据重新发送。

2、storm 和 spark-streaming：为什么用 storm 不同 spark-streaming

3、mr 和 spark 区别，怎么理解 spark-rdd

Mr 是文件方式的分布式计算框架 ,是将中间结果和最终结果记录在文件中 ,map 和 reduce 的数据分发也是在文件中。

spark 是内存迭代式的计算框架，计算的中间结果可以缓存内存，也可以缓存硬盘，但是不是每一步计算都需要缓存的。

Spark-rdd 是一个数据的分区记录集合……

4、sqoop 命令

sqoop import --connect jdbc:mysql://192.168.56.204:3306/sqoop --username hive --password hive --table jobinfo --target-dir /sqoop/test7 --inline-lob-limit 16777216 --fields-terminated-by '\t' -m 2
sqoop create-hive-table --connect jdbc:mysql://192.168.56.204:3306/sqoop --table jobinfo --username hive --password hive --hive-table sqtest --fields-terminated-by "\t" --lines-terminated-by "\n";

（6）Redis 篇

1、基本操作，存储格式

略

（7）Mysql 篇

1、mysql 集群的分布式事务

京东自主开发分布式 MYSQL 集群系统

2、mysql 性能优化（数据方面）

数据的分表、分库、分区

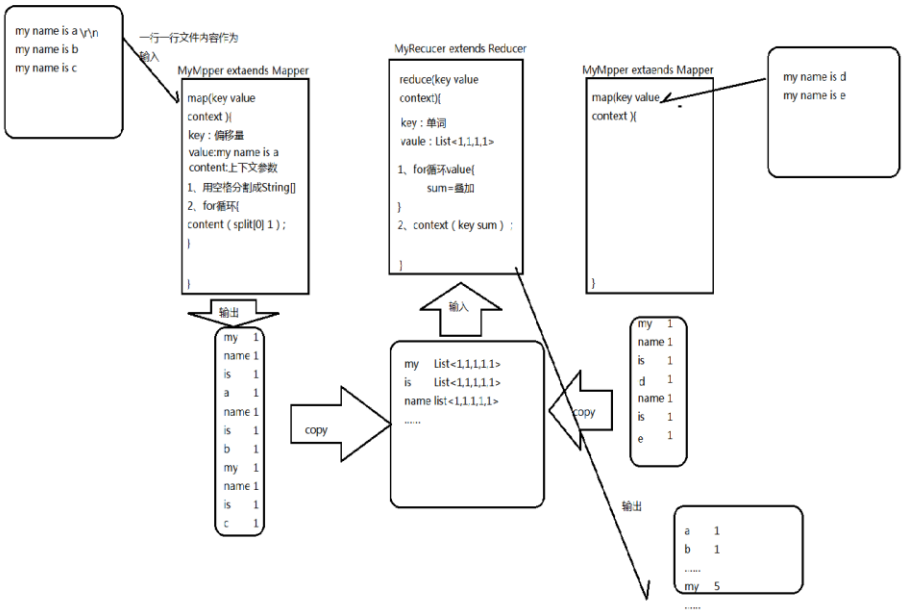
（6）Hadoop 篇

1、hadoop HA 两个 namenode 和 zk 之间的通信，zk 的选举机制？

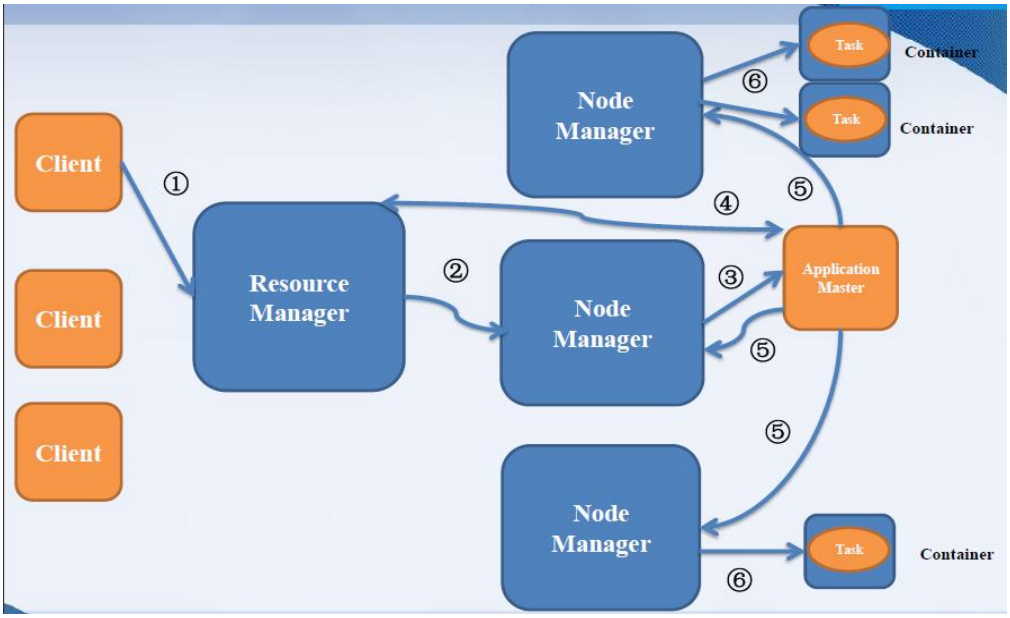
HA 是通过先后获取 zk 的锁决定谁是主

Zk 的选举机制，涉及到全新机群的选主和数据恢复的选主

2、mr 运行机制



3、yarn 流程



1) 用户向 YARN 中提交应用程序， 其中包括 ApplicationMaster 程序、启动 ApplicationMaster 的命令、用户程序等。

2) ResourceManager 为该应用程序分配第一个 Container， 并与对应的 NodeManager 通信，要求它在这个 Container 中启动应用程序的 ApplicationMaster。

3) ApplicationMaster 首先向 ResourceManager 注册， 这样用户可以直接通过 ResourceManage 查看应用程序的运行状态，然后它将为各个任务申请资源，并监控它的运行状态，直到运行结束，即重复步骤 4~7。

4) ApplicationMaster 采用轮询的方式通过 RPC 协议向 ResourceManager 申请和领取资源。

5) 一旦 ApplicationMaster 申请到资源后，便与对应的 NodeManager 通信，要求它启动任务。

6) NodeManager 为任务设置好运行环境（包括环境变量、JAR 包、二进制程序等）后，将任务启动命令写到一个脚本中，并通过运行该脚本启动任务。

7) 各个任务通过某个 RPC 协议向 ApplicationMaster 汇报自己的状态和进度，以让 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。在应用程序运行过程中，用户可随时通过 RPC 向 ApplicationMaster 查询应用程序的当前运行状态。

8) 应用程序运行完成后，ApplicationMaster 向 ResourceManager 注销并关闭自己。

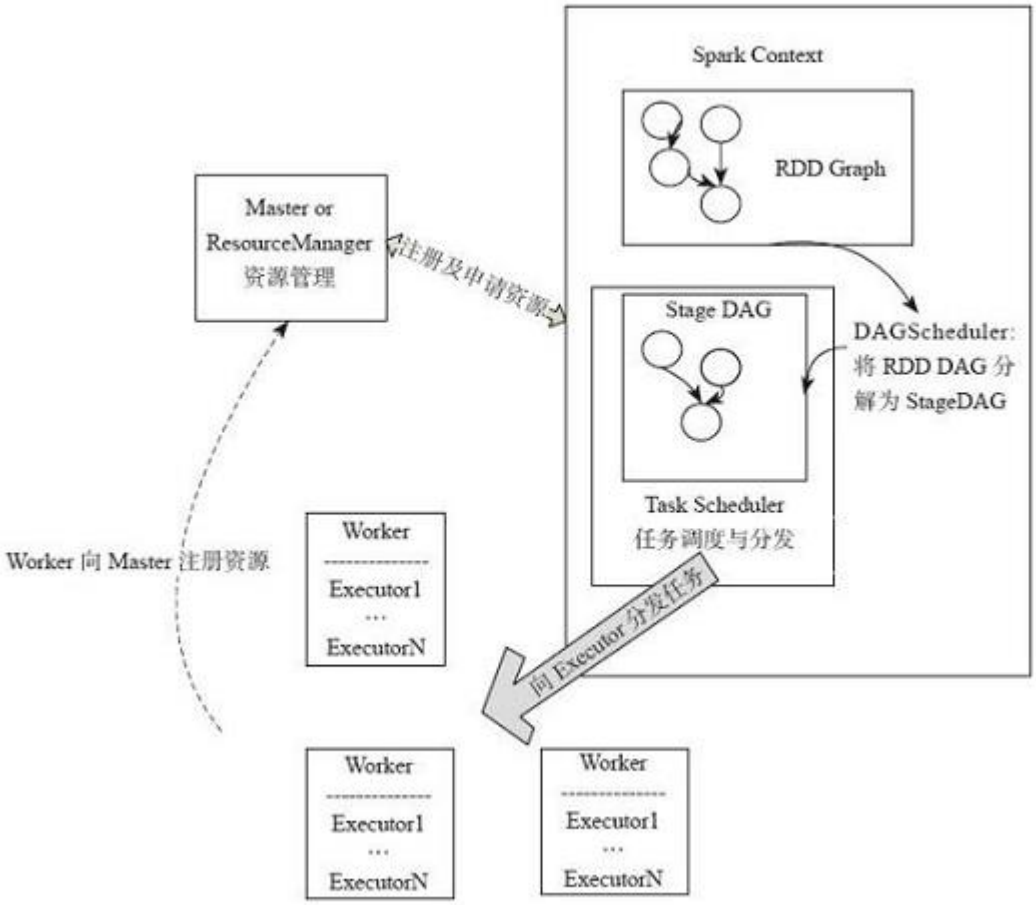
（7）Hbase

1、涉及到概念，文档

（8）Spark 篇

1、spark 原理

Spark 应用转换流程



1、 spark 应用提交后，经历了一系列的转换，最后成为 task 在每个节点上执行

2、 RDD 的 Action 算子触发 Job 的提交，生成 RDD DAG

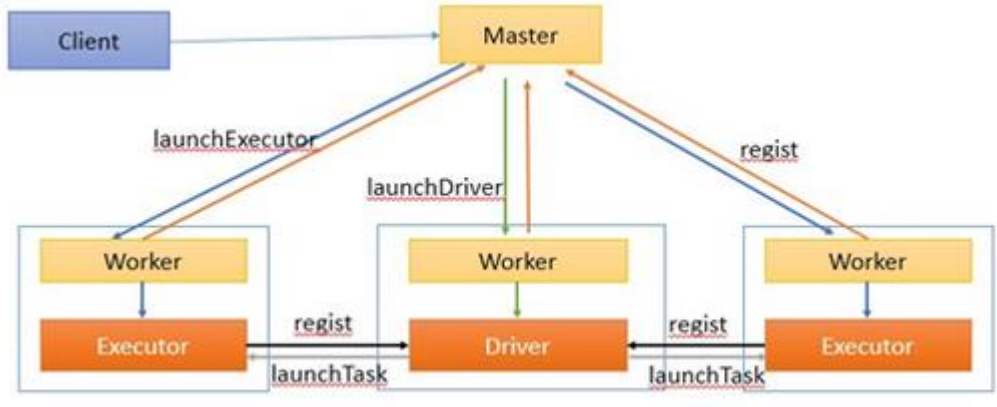
3、 由 DAGScheduler 将 RDD DAG 转化为 Stage DAG，每个 Stage 中产生相应的 Task 集合

4、 TaskScheduler 将任务分发到 Executor 执行

5、 每个任务对应相应的一个数据块，只用用户定义的函数处理数据块

Driver 运行在 Worker 上

通过 org.apache.spark.deploy.Client 类执行作业，作业运行命令如下：



作业执行流程描述：

1、客户端提交作业给 Master

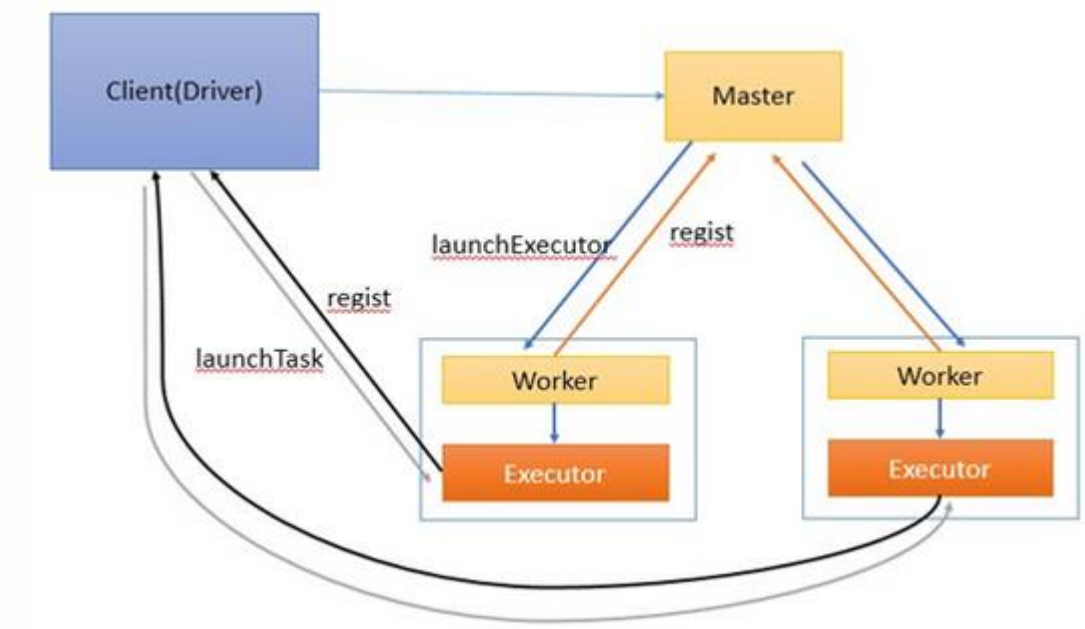
2、Master 让一个 Worker 启动 Driver，即 SchedulerBackend。Worker 创建一个 DriverRunner 线程，DriverRunner 启动 SchedulerBackend 进程。

3、另外 Master 还会让其余 Worker 启动 Exeuctor，即 ExecutorBackend。Worker 创建一个 ExecutorRunner 线程，ExecutorRunner 会启动 ExecutorBackend 进程。

4、ExecutorBackend 启动后会向 Driver 的 SchedulerBackend 注册。SchedulerBackend 进程中包含 DAGScheduler，它会根据用户程序，生成执行计划，并调度执行。对于每个 stage 的 task，都会被存放到 TaskScheduler 中，ExecutorBackend 向 SchedulerBackend 汇报的时候把 TaskScheduler 中的 task 调度到 ExecutorBackend 执行。

5、所有 stage 都完成后作业结束。

Driver 运行在客户端



作业执行流程描述：

1、客户端启动后直接运行用户程序，启动 Driver 相关的工作：DAGScheduler 和 BlockManagerMaster 等。

2、客户端的 Driver 向 Master 注册。

3、Master 还会让 Worker 启动 Exeuctor。Worker 创建一个 ExecutorRunner 线程，ExecutorRunner 会启动 ExecutorBackend 进程。

4、ExecutorBackend 启动后会向 Driver 的 SchedulerBackend 注册。Driver 的 DAGScheduler 解析作业并生成相应的 Stage，每个 Stage 包含的 Task 通过 TaskScheduler 分配给 Executor 执行。

5、所有 stage 都完成后作业结束。