大数据面试题总结

V 1.0

总结作者	蓦然
公众号	旧时光大数据

公众号二维码

联系我（微信）

时间：2021 年 2 月 23 日

Hadoop面试题

一、Hadoop面试题——基础

1、集群的最主要瓶颈

磁盘IO

2、Hadoop运行模式

单机版、伪分布式模式、完全分布式模式

3、Hadoop生态圈的组件并做简要描述

1）Zookeeper：是一个开源的分布式应用程序协调服务,基于zookeeper可以实现同步服务，配置维护，命名服务

2）Flume：一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统

3）Hbase：是一个分布式的、面向列的开源数据库, 利用Hadoop HDFS作为其存储系统

4）Hive：基于Hadoop的一个数据仓库工具，可以将结构化的数据档映射为一张数据库表，并提供简单的sql 查询功能，可以将sql语句转换为MapReduce任务进行运行

5）Sqoop：将一个关系型数据库中的数据导进到Hadoop的 HDFS中，也可以将HDFS的数据导进到关系型数据库中

4、解释“hadoop”和“hadoop 生态系统”两个概念

Hadoop是指Hadoop框架本身；hadoop生态系统，不仅包含hadoop，还包括保证hadoop框架正常高效运行其他框架，比如zookeeper、Flume、Hbase、Hive、Sqoop等辅助框架。

5、请列出正常工作的Hadoop集群中Hadoop都分别需要启动哪些进程，它们的作用分别是什么?

1）NameNode
它是hadoop中的主服务器，管理文件系统名称空间和对集群中存储的文件的访问，保存有metadata

2）SecondaryNameNode
它不是namenode的冗余守护进程，而是提供周期检查点和清理任务。帮助NN合并editslog，减少NN启动时间。

3）DataNode
它负责管理连接到节点的存储（一个集群中可以有多个节点）。每个存储数据的节点运行一个datanode守护进程。

4）ResourceManager（JobTracker）
JobTracker负责调度DataNode上的工作。每个DataNode有一个TaskTracker，它们执行实际工作。

5）NodeManager（TaskTracker）
执行任务

6）DFSZKFailoverController
高可用时它负责监控NN的状态，并及时的把状态信息写入ZK。它通过一个独立线程周期性的调用NN上的一个特定接口来获 取NN的健康状态。FC也有选择谁作为Active NN的权利，因为最多只有两个节点，目前选择策略还比较简单（先到先得，轮换）。

7）JournalNode
高可用情况下存放namenode的editlog文件。

二、Hadoop面试题——HDFS

1、 HDFS 中的 block 默认保存几份?

默认保存3份

2、HDFS 默认 BlockSize 是多大?

Hadoop 2.7.2版本及之前默认64MB，Hadoop 2.7.3版本及之后默认128M

3、负责HDFS数据存储的是哪一部分?

DataNode负责数据存储

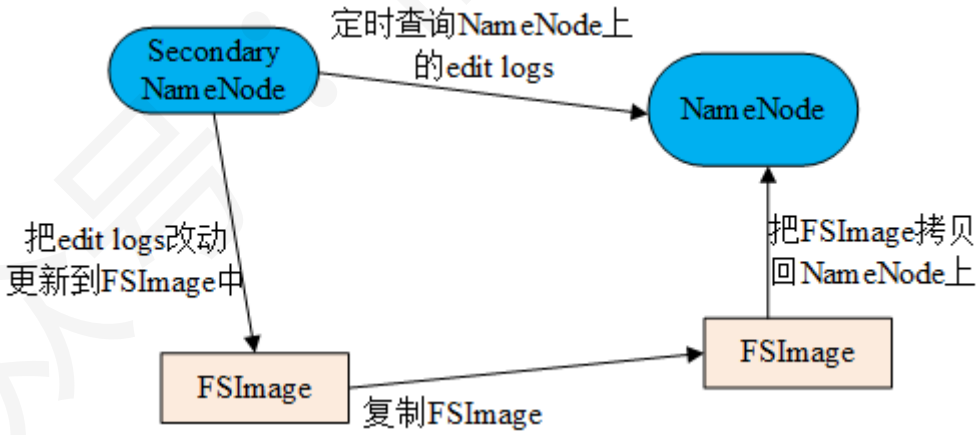
4、SecondaryNameNode的目的是什么?

它的目的是帮助NameNode合并编辑日志，减少NameNode 启动时间

详细介绍:

2NN节点主要是辅助NameNode节点，防止单点故障的发生，不过它并非是NN节点的热备，也就是说当NN挂掉之后，2NN节点并不能马上替换NN并提供服务。

当NN节点重启的时候，2NN节点有助于更新NN，使其启动更快，但是在实际应用中，NN节点的重启不是很常见的，而NN节点长时间的运行会导致EditLogs文件特别大，当NN节点需要重启时，会导致加载EditLogs花费过多的时间，或者是当NN节点在集群运行过程中突然出现故障问题，而EditLogs日志上的数据还没有及时合并到FSImage上，这就会导致文件的丢失。2NN的出现就是为了缓解NN节点的这个问题，帮助NN节点定期性的将EditLogs日志上的信息同步到FSImage文件上，得到最新的FSImage文件，在一些紧急情况下，辅助恢复NN节点。**2NN工作原理图**如下图所示。



1）2NN会定期从NN节点拷贝相应的EditLogs日志，将它加载到内存合并到FSImage镜像文件。

2）2NN上的FSImage更新之后，生成FSImage.chkpoint文件，之后将新的镜像文件拷贝到NN节点。

3）NN节点将FSImage.chkpoint文件重新命名为FSImage，更新后的FSImage文件可以防止一些意外，这样就能节约NN节点下次的启动所需时间。

5、文件大小设置，增大有什么影响?

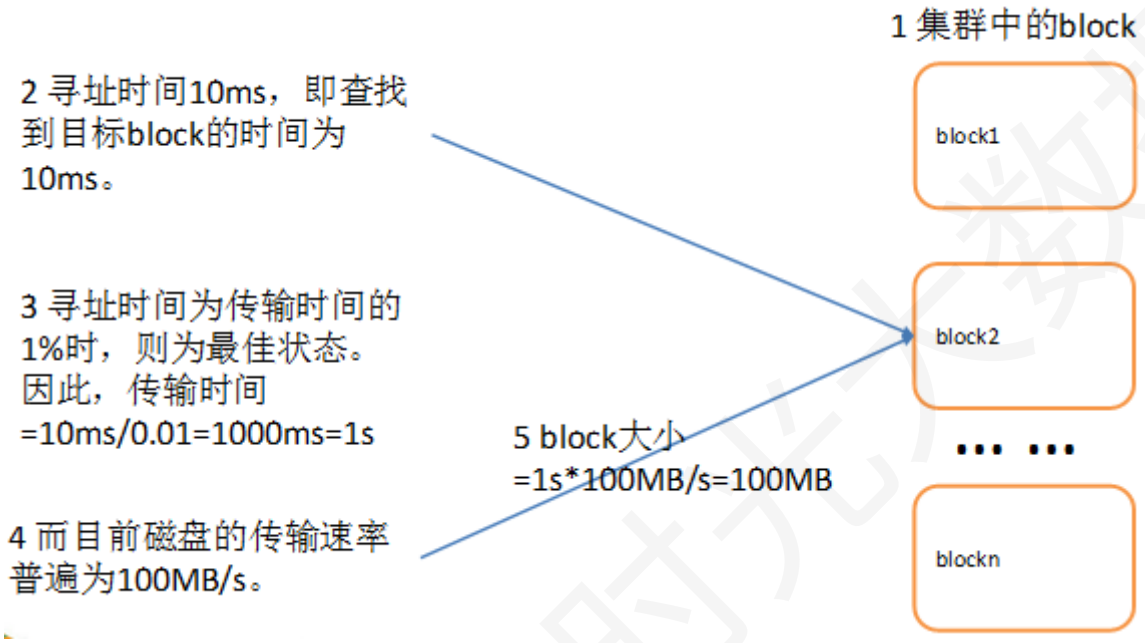
HDFS中的文件在物理上是分块存储（block），块的大小可以通过配置参数(dfs.blocksize)来规定，Hadoop 2.7.2版本及之前默认64MB，Hadoop 2.7.3版本及之后默认128M。

思考：为什么块的大小不能设置的太小，也不能设置的太大？

HDFS的块比磁盘的块大，其目的是为了最小化寻址开销。如果块设置得足够大，从磁盘传输数据的时间会明显大于定位这个块开始位置所需的时间。因而，传输一个由多个块组成的文件的时间取决于磁盘传输速率。

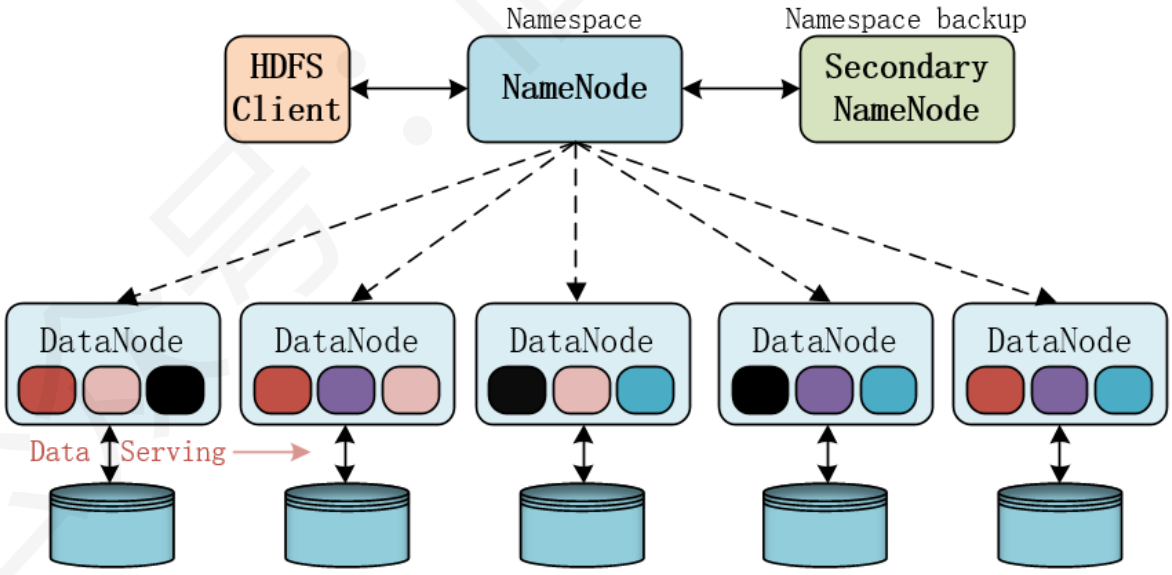
如果寻址时间约为10ms，而传输速率为100MB/s，为了使寻址时间仅占传输时间的1%，我们要将块大小设置约为100MB。默认的块大小128MB。

块的大小：10ms x 100 x 100M/s = 100M，如图



增加文件块大小，需要增加磁盘的传输速率。

6、HDFS组成架构（☆☆☆☆☆）



架构主要由四个部分组成，分别为HDFS Client、NameNode、DataNode和Secondary NameNode。下面我们分别介绍这四个组成部分。

1）Client：就是客户端

（1）文件切分。文件上传HDFS的时候，Client将文件切分成一个一个的Block，然后进行存储；

（2）与NameNode交互，获取文件的位置信息；

（3）与DataNode交互，读取或者写入数据；

（4）Client提供一些命令来管理HDFS，比如启动或者关闭HDFS；

（5）Client可以通过一些命令来访问HDFS；

2）NameNode：就是Master，它是一个主管、管理者

（1）管理HDFS的名称空间；

（2）管理数据块（Block）映射信息；

（3）配置副本策略；

（4）处理客户端读写请求。

3）DataNode：就是Slave。NameNode下达命令，DataNode执行实际的操作

（1）存储实际的数据块；

（2）执行数据块的读/写操作。

4）Secondary NameNode：并非NameNode的热备。当NameNode挂掉的时候，它并不能马上替换NameNode并提供服务

（1）辅助NameNode，分担其工作量；

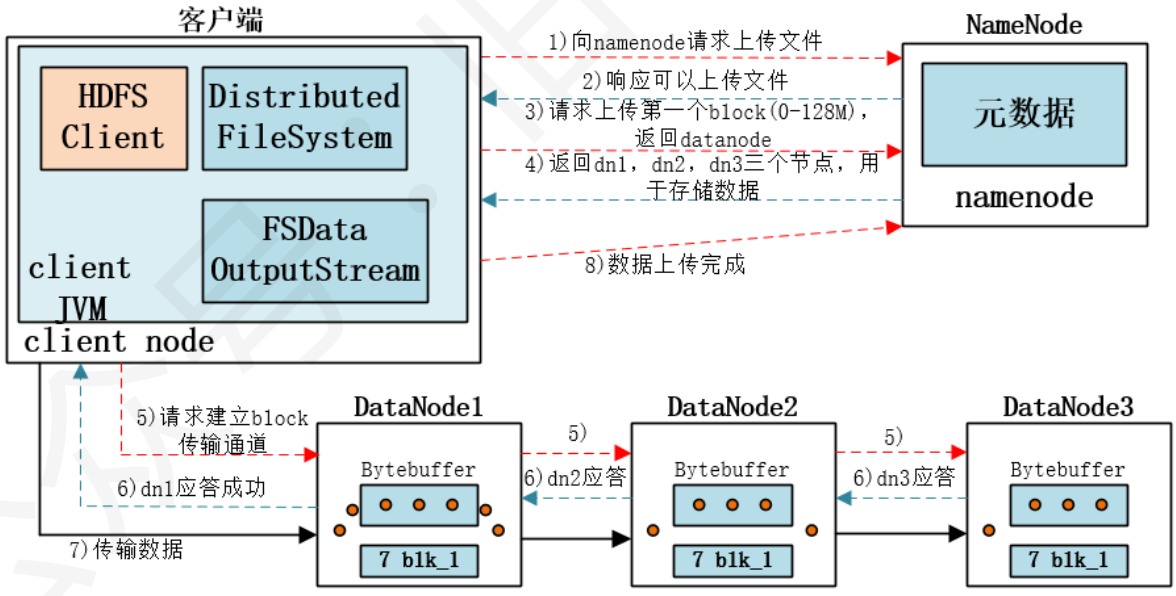
（2）定期合并Fsimage和Edits，并推送给NameNode；

（3）在紧急情况下，可辅助恢复NameNode。

7、HDFS的存储机制（☆☆☆☆☆）

HDFS存储机制，包括HDFS的**写入数据过程**和**读取数据过程**两部分

HDFS写数据过程



1）客户端通过Distributed FileSystem模块向NameNode请求上传文件，NameNode检查目标文件是否已存在，父目录是否存在。

2）NameNode返回是否可以上传。

3）客户端请求第一个 block上传到哪几个datanode服务器上。

4）NameNode返回3个datanode节点，分别为dn1、dn2、dn3。

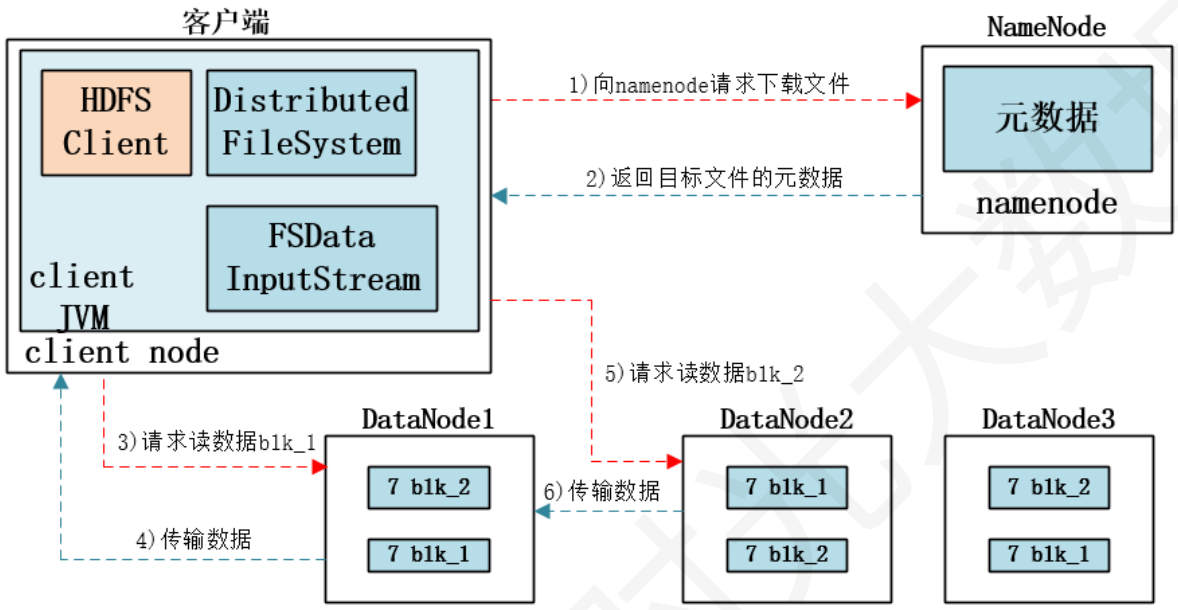
5）客户端通过FSDataOutputStream模块请求dn1上传数据，dn1收到请求会继续调用dn2，然后dn2调用dn3，将这个通信管道建立完成。

6）dn1、dn2、dn3逐级应答客户端。

7）客户端开始往dn1上传第一个block（先从磁盘读取数据放到一个本地内存缓存），以packet为单位，dn1收到一个packet就会传给dn2，dn2传给dn3；dn1每传一个packet会放入一个应答队列等待应答。

8）当一个block传输完成之后，客户端再次请求NameNode上传第二个block的服务器。（重复执行3-7步）。

HDFS读数据过程



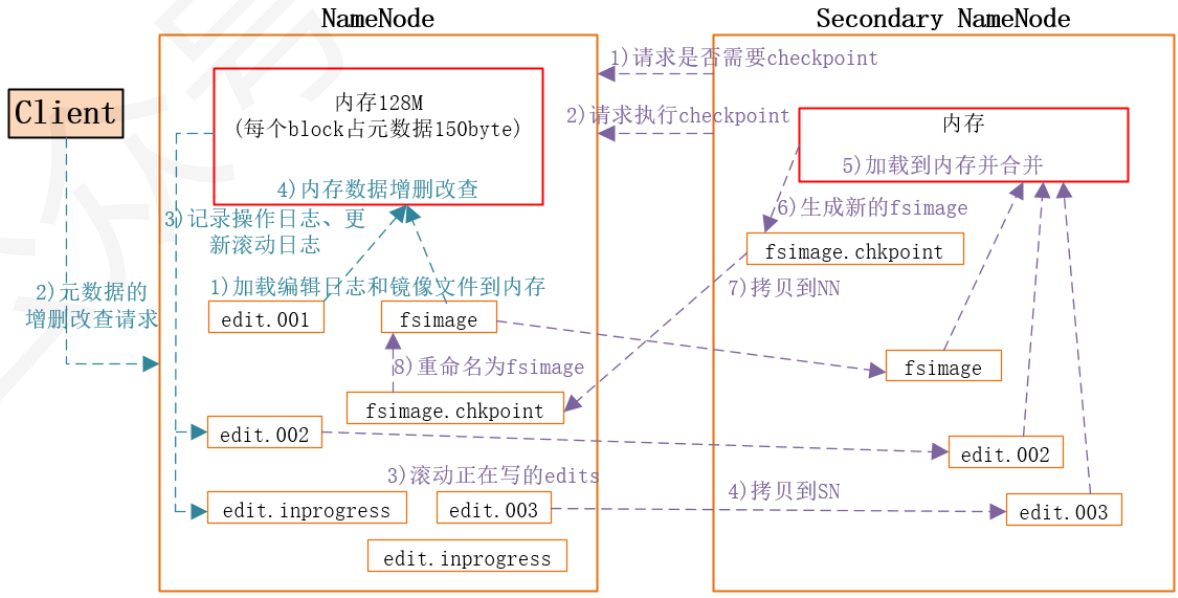
1）客户端通过Distributed FileSystem向NameNode请求下载文件，NameNode通过查询元数据，找到文件块所在的DataNode地址。

2）挑选一台DataNode（就近原则，然后随机）服务器，请求读取数据。

3）DataNode开始传输数据给客户端（从磁盘里面读取数据输入流，以packet为单位来做校验）。

4）客户端以packet为单位接收，先在本地缓存，然后写入目标文件。

8、Secondary NameNode工作机制（☆☆☆☆☆）



1）**第一阶段：NameNode启动**

（1）第一次启动NameNode格式化后，创建fsimage和edits文件。如果不是第一次启动，直接加载编辑日志和镜像文件到内存。

（2）客户端对元数据进行增删改的请求。

（3）NameNode记录操作日志，更新滚动日志。

（4）NameNode在内存中对数据进行增删改查。

2）**第二阶段：Secondary NameNode工作**

（1）Secondary NameNode询问NameNode是否需要checkpoint。直接带回NameNode是否检查结果。

（2）Secondary NameNode请求执行checkpoint。

（3）NameNode滚动正在写的edits日志。

（4）将滚动前的编辑日志和镜像文件拷贝到Secondary NameNode。

（5）Secondary NameNode加载编辑日志和镜像文件到内存，并合并。

（6）生成新的镜像文件fsimage.chkpoint。

（7）拷贝fsimage.chkpoint到NameNode。

（8）NameNode将fsimage.chkpoint重新命名成fsimage。

9、NameNode与SecondaryNameNode 的区别与联系？（☆☆☆☆☆）

机制流程看第6题

区别

1）NameNode负责管理整个文件系统的元数据，以及每一个路径（文件）所对应的数据块信息。

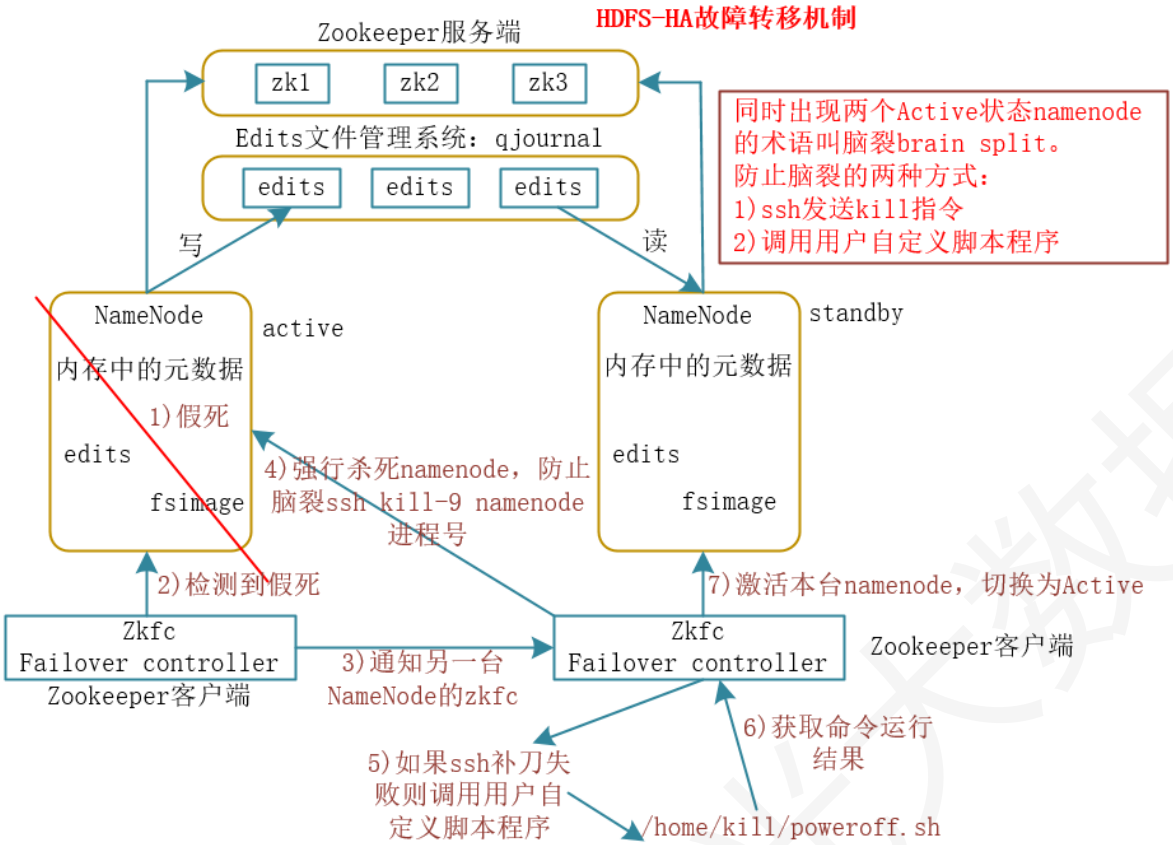
2）SecondaryNameNode主要用于定期合并命名空间镜像和命名空间镜像的编辑日志。

联系

1）SecondaryNameNode中保存了一份和namenode一致的镜像文件（fsimage）和编辑日志（edits）。

2）在主namenode发生故障时（假设没有及时备份数据），可以从SecondaryNameNode恢复数据。

10、HAnamenode 是如何工作的？（☆☆☆☆☆）



ZKFailoverController主要职责

1）健康监测：周期性的向它监控的NN发送健康探测命令，从而来确定某个NameNode是否处于健康状态，如果机器宕机，心跳失败，那么zkfc就会标记它处于一个不健康的状态。

2）会话管理：如果NN是健康的，zkfc就会在zookeeper中保持一个打开的会话，如果NameNode同时还是Active状态的，那么zkfc还会在Zookeeper中占有一个类型为短暂类型的znode，当这个NN挂掉时，这个znode将会被删除，然后备用的NN，将会得到这把锁，升级为主NN，同时标记状态为Active。

3）当宕机的NN新启动时，它会再次注册zookeper，发现已经有znode锁了，便会自动变为Standby状态，如此往复循环，保证高可靠，需要注意，目前仅仅支持最多配置2个NN。

4）master选举：如上所述，通过在zookeeper中维持一个短暂类型的znode，来实现抢占式的锁机制，从而判断那个NameNode为Active状态。

三、Hadoop面试题——MapReduce

1、谈谈Hadoop序列化和反序列化及自定义bean对象实现序列化?

序列化和反序列化

1）**序列化**就是把内存中的对象，转换成字节序列（或其他数据传输协议）以便于存储（持久化）和网络传输。

2）**反序列化**就是将收到字节序列（或其他数据传输协议）或者是硬盘的持久化数据，转换成内存中的对象。

为什么要序列化

一般来说，“活的”对象只生存在内存里，关机断电就没有了。而且“活的”对象只能由本地的进程使用，不能被发送到网络上的另外一台计算机。然而序列化可以存储“活的”对象，可以将“活的”对象发送到远程计算机

为什么不用Java序列化

Java的序列化是一个重量级序列化框架（Serializable），一个对象被序列化后，会附带很多额外的信息（各种校验信息，header，继承体系等），不便于在网络中高效传输。所以，hadoop自己开发了一套序列化机制（Writable），精简、高效。

Hadoop序列化特点

1）**紧凑**：高效使用存储空间

2）**快速**：读写数据的额外开销小

3）**可扩展**：随着通讯协议的升级而可升级

4）**互操作**：支持多语言的交互

自定义bean对象要想序列化传输步骤及注意事项

1）必须实现Writable接口

2）反序列化时，需要反射调用空参构造函数，所以必须有空参构造

3）重写序列化方法

4）重写反序列化方法

5）注意反序列化的顺序和序列化的顺序完全一致

6）要想把结果显示在文件中，需要重写toString()，且用"\t"分开，方便后续用

7）如果需要将自定义的bean放在key中传输，则还需要实现comparable接口，因为mapreduce框中的shuffle过程一定会对key进行排序

2、FileInputFormat切片机制（☆☆☆☆☆）

job提交流程源码详解

```
waitForCompletion()
submit();
// 1、建立连接
    connect();
        // 1）创建提交job的代理
        new Cluster(getConfiguration());
            // （1）判断是本地yarn还是远程
            initialize(jobTrackAddr, conf);
// 2、提交job
submitter.submitJobInternal(Job.this, cluster)
    // 1）创建给集群提交数据的Stag路径
    Path jobStagingArea = JobSubmissionFiles.getStagingDir(cluster, conf);
    // 2）获取jobid ，并创建job路径
    JobID jobId = submitClient.getNewJobID();
    // 3）拷贝jar包到集群
    copyAndConfigureFiles(job, submitJobDir);
    rUploader.uploadFiles(job, jobSubmitDir);
    // 4）计算切片，生成切片规划文件
    writeSplits(job, submitJobDir);
    maps = writeNewSplits(job, jobSubmitDir);
    input.getSplits(job);
    // 5）向Stag路径写xml配置文件
    writeConf(conf, submitJobFile);
    conf.writeXml(out);
```

```
25      // 6）提交job,返回提交状态
26      status = submitClient.submitJob(jobId, submitJobDir.toString(),
job.getCredentials());
```

3、在一个运行的Hadoop 任务中，什么是InputSplit？（☆☆☆☆☆）

FileInputFormat源码解析(input.getSplits(job))

1）找到你数据存储的目录

2）开始遍历处理（规划切片）目录下的每一个文件

3）遍历第一个文件（假设为ss.txt）

（1）获取文件大小fs.sizeOf(ss.txt);

（2）计算切片大小
computeSliteSize(Math.max(minSize,Math.min(maxSize,blocksize)))=blocksize=128M

（3）默认情况下，切片大小=blocksize

（4）开始切，形成第1个切片：ss.txt—0:128M 第2个切片ss.txt—128:256M 第3个切片ss.txt—256M:300M（每次切片时，都要判断切完剩下的部分是否大于块的1.1倍，不大于1.1倍就划分一块切片）

（5）将切片信息写到一个切片规划文件中

（6）整个切片的核心过程在getSplit()方法中完成。

（7）数据切片只是在逻辑上对输入数据进行分片，并不会再磁盘上将其切分成分片进行存储。InputSplit只记录了分片的元数据信息，比如起始位置、长度以及所在的节点列表等。

（8）注意：block是HDFS上物理上存储的存储的数据，切片是对数据逻辑上的划分。

4） 提交切片规划文件到yarn上，yarn上的MrAppMaster就可以根据切片规划文件计算开启maptask个数。

4、如何判定一个job的map和reduce的数量?

1）map数量

splitSize=max{minSize,min{maxSize,blockSize}}

map数量由处理的数据分成的block数量决定default_num = total_size / split_size;

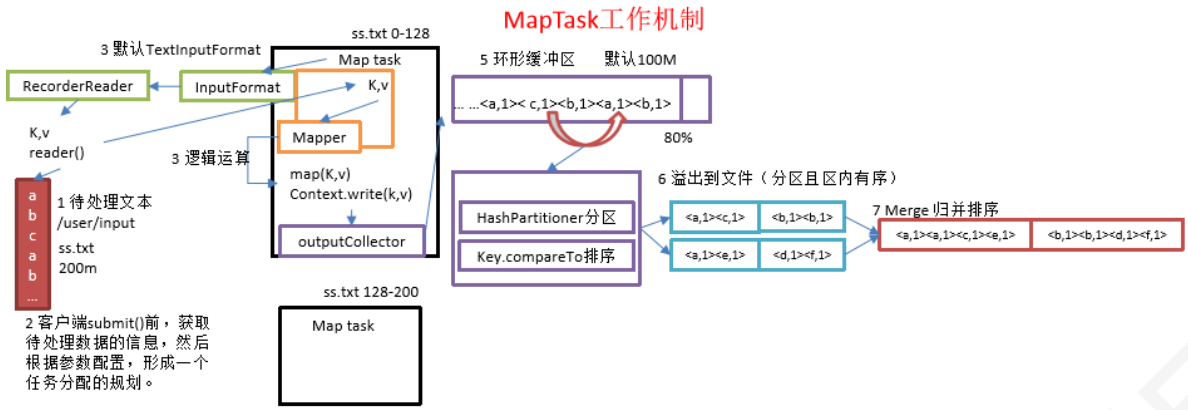
2）reduce数量

reduce的数量job.setNumReduceTasks(x);x 为reduce的数量。不设置的话默认为 1。

5、 Maptask的个数由什么决定?

一个job的map阶段MapTask并行度（个数），由客户端提交job时的切片个数决定。

6、MapTask和ReduceTask工作机制（☆☆☆☆☆）（也可回答MapReduce工作原理）



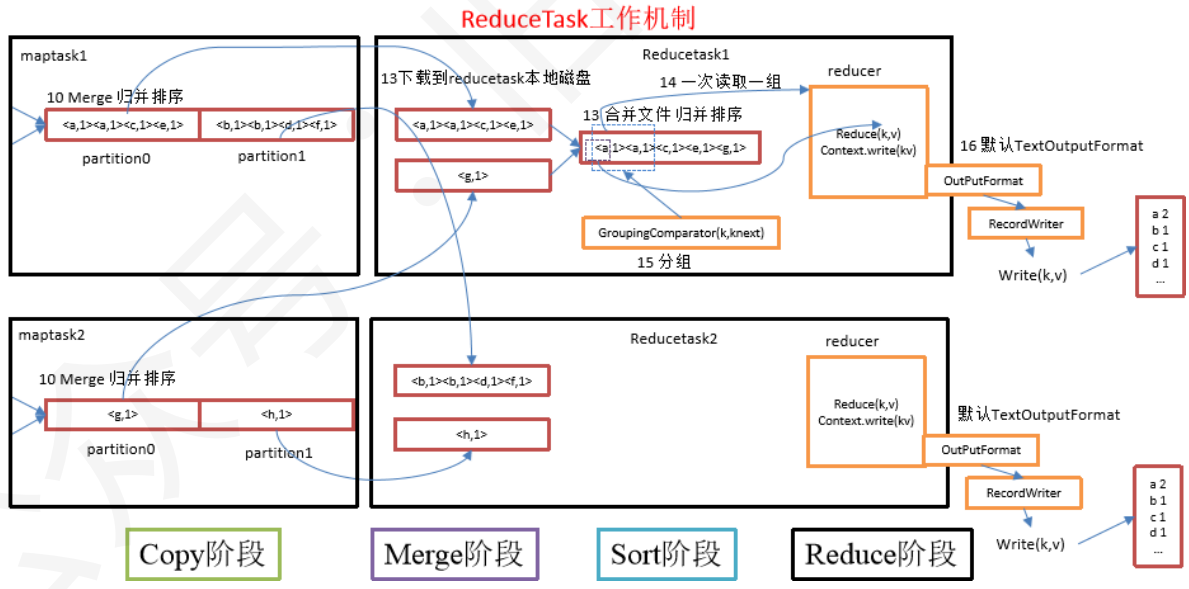
Read阶段：Map Task通过用户编写的RecordReader，从输入InputSplit中解析出一个个key/value。

Map阶段：该节点主要是将解析出的key/value交给用户编写map()函数处理，并产生一系列新的key/value。

Collect收集阶段：在用户编写map()函数中，当数据处理完成后，一般会调用OutputCollector.collect()输出结果。在该函数内部，它会将生成的key/value分区（调用Partitioner），并写入一个环形内存缓冲区中。

Spill阶段：即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必要时对数据进行合并、压缩等操作。

Combine阶段：当所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。



Copy阶段：ReduceTask从各个MapTask上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。

Merge阶段：在远程拷贝数据的同时，ReduceTask启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

Sort阶段：按照MapReduce语义，用户编写reduce()函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个MapTask已经实现对自己的处理结果进行了局部排序，因此，ReduceTask只需对所有数据进行一次归并排序即可。

Reduce阶段：reduce()函数将计算结果写到HDFS上。

7、描述mapReduce有几种排序及排序发生的阶段（☆☆☆☆☆）

排序的分类

1）部分排序：MapReduce根据输入记录的键对数据集排序。保证输出的每个文件内部排序。

2）全排序：如何用Hadoop产生一个全局排序的文件？最简单的方法是使用一个分区。但该方法在处理大型文件时效率极低，因为一台机器必须处理所有输出文件，从而完全丧失了MapReduce所提供的并行架构。

替代方案：首先创建一系列排好序的文件；其次，串联这些文件；最后，生成一个全局排序的文件。主要思路是使用一个分区来描述输出的全局排序。例如：可以为待分析文件创建3个分区，在第一分区中，记录的单词首字母a-g，第二分区记录单词首字母h-n, 第三分区记录单词首字母o-z。

3）辅助排序（GroupingComparator分组）：Mapreduce框架在记录到达reducer之前按键对记录排序，但键所对应的值并没有被排序。甚至在不同的执行轮次中，这些值的排序也不固定，因为它们来自不同的map任务且这些map任务在不同轮次中完成时间各不相同。一般来说，大多数MapReduce程序会避免让reduce函数依赖于值的排序。但是，有时也需要通过特定的方法对键进行排序和分组等以实现对值的排序。

4）二次排序：在自定义排序过程中，如果compareTo中的判断条件为两个即为二次排序。

自定义排序WritableComparable

bean对象实现WritableComparable接口重写compareTo方法，就可以实现排序

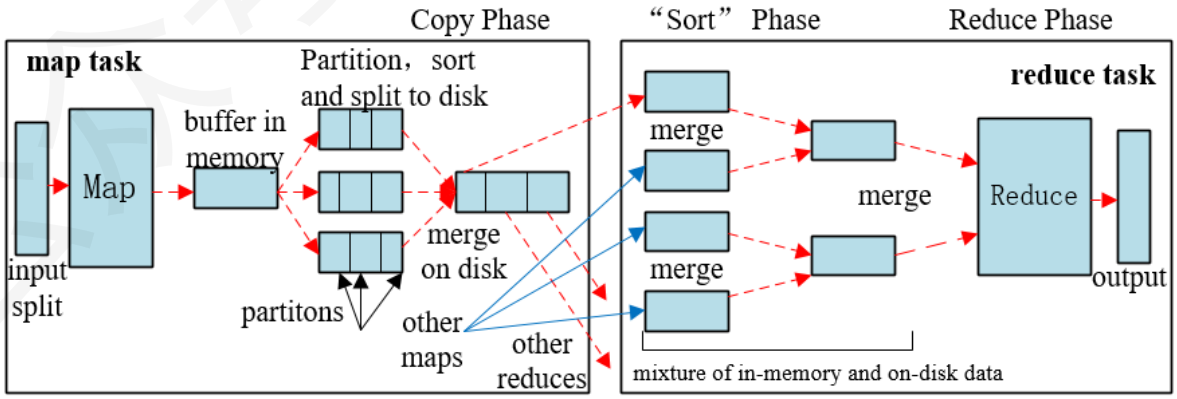
```
@Override
public int compareTo(FlowBean o) {
    // 倒序排列，从大到小
    return this.sumFlow > o.getSumFlow() ? -1 : 1;
}
```

排序发生的阶段

1）一个是在map side发生在spill后partition前。

2）一个是在reduce side发生在copy后 reduce前。

8、描述mapReduce中shuffle阶段的工作流程，如何优化shuffle阶段（☆☆☆☆☆）



分区，排序，溢写，拷贝到对应reduce机器上，增加combiner，压缩溢写的文件。

9、描述mapReduce中combiner的作用是什么，一般使用情景，哪些情况不需要，及和reduce的区别?

1）Combiner的意义就是对每一个maptask的输出进行局部汇总，以减小网络传输量。

2）Combiner能够应用的前提是不能影响最终的业务逻辑，而且，Combiner的输出kv应该跟reducer的输入kv类型要对应起来。

3）Combiner和reducer的区别在于运行的位置。

Combiner是在每一个maptask所在的节点运行；

Reducer是接收全局所有Mapper的输出结果。

10、如果没有定义partitioner，那数据在被送达reducer前是如何被分区的?

如果没有自定义的partitioning，则默认的partition算法，即根据每一条数据的key的hashcode值模运算（%）reduce的数量，得到的数字就是“分区号“。

11、MapReduce 出现单点负载多大，怎么负载平衡？（☆☆☆☆☆）

通过Partitioner实现

12、MapReduce 怎么实现 TopN？（☆☆☆☆☆）

可以自定义groupingcomparator，对结果进行最大值排序，然后再reduce输出时，控制只输出前n个数。就达到了topn输出的目的。

13、Hadoop的缓存机制（Distributedcache）（☆☆☆☆☆）

分布式缓存一个最重要的应用就是在进行join操作的时候，如果一个表很大，另一个表很小，我们就可以将这个小表进行广播处理，即每个计算节点上都存一份，然后进行map端的连接操作，经过我的实验验证，这种情况下处理效率大大高于一般的reduce端join，广播处理就运用到了分布式缓存的技术。

DistributedCache将拷贝缓存的文件到Slave节点在任何Job在节点上执行之前，文件在每个Job中只会被拷贝一次，缓存的归档文件会被在Slave节点中解压缩。将本地文件复制到HDFS中去，接着Client会通过addCacheFile() 和addCacheArchive()方法告诉DistributedCache在HDFS中的位置。当文件存放到文地时，JobClient同样获得DistributedCache来创建符号链接，其形式为文件的URI加fragment标识。当用户需要获得缓存中所有有效文件的列表时，JobConf 的方法 getLocalCacheFiles() 和 getLocalArchives()都返回一个指向本地文件路径对象数组。

14、如何使用mapReduce实现两个表的join?（☆☆☆☆☆）

reduce side join：

在map阶段，map函数同时读取两个文件File1和File2，为了区分两种来源的key/value数据对，对每条数据打一个标签（tag）,比如：tag=0 表示来自文件File1，tag=2 表示来自文件File2。

Map side join：

Map side join是针对以下场景进行的优化：两个待连接表中，有一个表非常大，而另一个表非常小，以至于小表可以直接存放到内存中。这样，我们可以将小表复制多份，让每个map task 内存中存在一份（比如存放到hash table 中），然后只扫描大表：对于大表中的每一条记录key/value，在hash table 中查找是否有相同的key 的记录，如果有，则连接后输出即可。

15、什么样的计算不能用mr来提速?

1）数据量很小。

2）繁杂的小文件。

3）索引是更好的存取机制的时候。

4）事务处理。

5）只有一台机器的时候。

16、ETL是哪三个单词的缩写

Extraction-Transformation-Loading的缩写，中文名称为数据提取、转换和加载。

四、Hadoop面试题——YARN

1、简述Hadoop1与Hadoop2 的架构异同

1）Hadoop2.x加入了yarn解决了资源调度的问题。

2）Hadoop2.x加入了对zookeeper的支持实现比较可靠的高可用。

2、为什么会产生 yarn,它解决了什么问题，有什么优势?

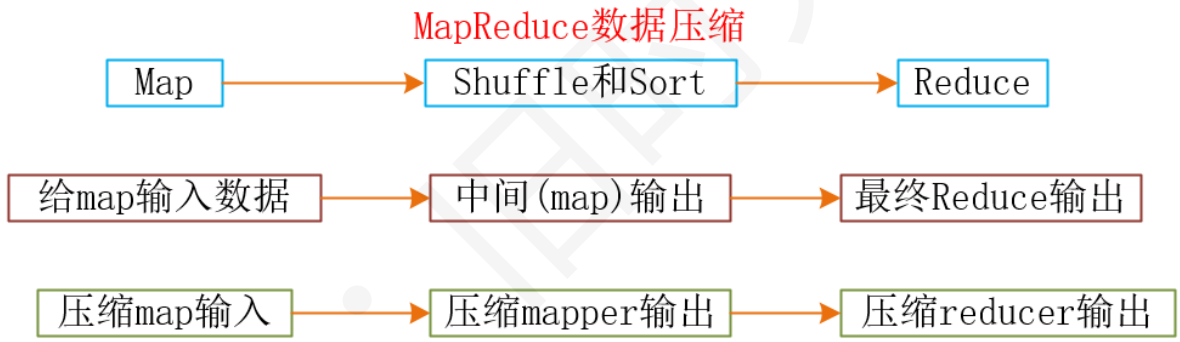
1）Yarn最主要的功能就是解决运行的用户程序与yarn框架完全解耦。

2）Yarn上可以运行各种类型的分布式运算程序（mapreduce只是其中的一种），比如mapreduce、storm程序，spark程序……

3、HDFS的数据压缩算法? （☆☆☆☆☆）

Hadoop中常用的压缩算法有bzip2、gzip、lzo、snappy，其中lzo、snappy需要操作系统安装native库才可以支持。

数据压缩的位置如下所示。



MapReduce数据压缩解析

输入端采用压缩

在有大量数据并计划重复处理的情况下，应该考虑对输入进行压缩。然而，你无须显示指定使用的编解码方式。Hadoop自动检查文件扩展名，如果扩展名能够匹配，就会用恰当的编解码方式对文件进行压缩和解压。否则，Hadoop就不会使用任何编解码器。

mapper输出端采用压缩

当map任务输出的中间数据量很大时，应考虑在此阶段采用压缩技术。这能显著改善内部数据Shuffle过程，而Shuffle过程在Hadoop处理过程中是资源消耗最多的环节。如果发现数据量大造成网络传输缓慢，应该考虑使用压缩技术。可用于压缩mapper输出的快速编解码器包括LZO或者Snappy。

reducer输出采用压缩

在此阶段启用压缩技术能够减少要存储的数据量，因此降低所需的磁盘空间。当mapreduce作业形成作业链条时，因为第二个作业的输入也已压缩，所以启用压缩同样有效。

压缩方式对比

1）Gzip压缩

优点：压缩率比较高，而且压缩/解压速度也比较快；hadoop本身支持，在应用中处理gzip格式的文件就和直接处理文本一样；大部分linux系统都自带gzip命令，使用方便。

缺点：不支持split。

应用场景：当每个文件压缩之后在130M以内的（1个块大小内），都可以考虑用gzip压缩格式。例如说一天或者一个小时的日志压缩成一个gzip文件，运行mapreduce程序的时候通过多个gzip文件达到并发。hive程序，streaming程序，和java写的mapreduce程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

2）Bzip2压缩

优点：支持split；具有很高的压缩率，比gzip压缩率都高；hadoop本身支持，但不支持native；在linux系统下自带bzip2命令，使用方便。

缺点：压缩/解压速度慢；不支持native。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，可以作为mapreduce作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持split，而且兼容之前的应用程序（即应用程序不需要修改）的情况。

3）Lzo压缩

优点：压缩/解压速度也比较快，合理的压缩率；支持split，是hadoop中最流行的压缩格式；可以在linux系统下安装lzop命令，使用方便。

缺点：压缩率比gzip要低一些；hadoop本身不支持，需要安装；在应用中对lzo格式的文件需要做一些特殊处理（为了支持split需要建索引，还需要指定inputformat为lzo格式）。

应用场景：一个很大的文本文件，压缩之后还大于200M以上的可以考虑，而且单个文件越大，lzo优点越越明显。

4）Snappy压缩

优点：高速压缩速度和合理的压缩率。

缺点：不支持split；压缩率比gzip要低；hadoop本身不支持，需要安装；

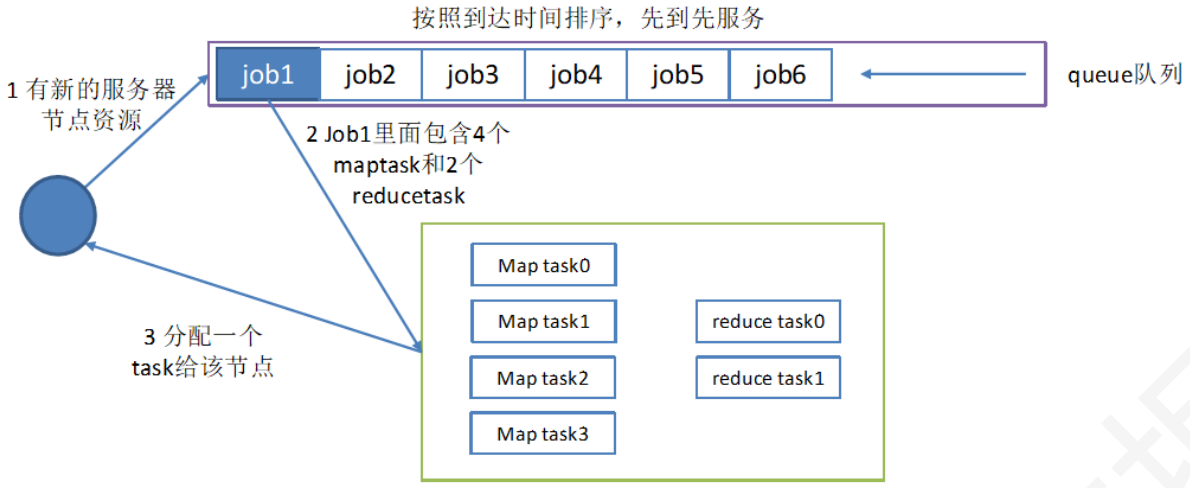
应用场景：当Mapreduce作业的Map输出的数据比较大的时候，作为Map到Reduce的中间数据的压缩格式；或者作为一个Mapreduce作业的输出和另外一个Mapreduce作业的输入。

4、Hadoop的调度器总结（☆☆☆☆☆）

Hadoop作业调度器主要有三种：**FIFO**、**Capacity Scheduler**和**Fair Scheduler**。Hadoop3.1.3默认的资源调度器是Capacity Scheduler。

具体设置详见：yarn-default.xml文件

1）先进先出调度器（FIFO）



它先按照作业的优先级高低，再按照到达时间的先后选择被执行的作业。

2）容量调度器（Capacity Scheduler）

（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。

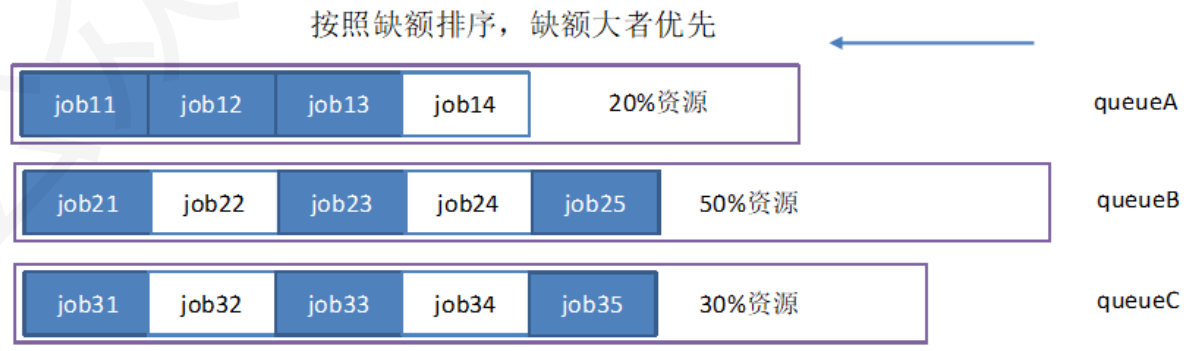
（2）为了防止同一个用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源量进行限定。

（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。

（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，是最先运行，也是同时运行。

3）公平调度器（Fair Scheduler）



支持多队列多用户，每个队列中的资源量可以配置，同一个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。

实际上，Hadoop的调度器远不止以上三种，也有很多针对新型应用的Hadoop调度器。

5、MapReduce 2.0 容错性 （☆☆☆☆☆）

1）MRAppMaster容错性

一旦运行失败，由YARN的ResourceManager负责重新启动，最多重启次数可由用户设置，默认是2次。一旦超过最高重启次数，则作业运行失败。

2）Map Task/Reduce

Task Task周期性向MRAppMaster汇报心跳；一旦Task 挂掉，则MRAppMaster将为之重新申请资源，并运行之。最多重新运行次数可由用户设置，默认4 次。

6、MapReduce推测执行算法及原理

1）作业完成时间取决于最慢的任务完成时间

一个作业由若干个Map 任务和Reduce 任务构成。因硬件老化、软件Bug 等，某些任务可能运行非常慢。

典型案例：系统中有99%的Map任务都完成了，只有少数几个Map老是进度很慢，完不成，怎么办?

2）推测执行机制

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

3）不能启用推测执行机制情况

（1）任务间存在严重的负载倾斜；

（2）特殊任务，比如任务向数据库中写数据。

4）算法原理

假设某一时刻，任务T的执行进度为progress，则可通过一定的算法推测出该任务的最终完成时刻estimateEndTime。另一方面，如果此刻为该任务启动一个备份任务，则可推断出它可能的完成时刻estimateEndTime`,于是可得出以下几个公式：

```
estimateEndTime = estimatedRunTime+taskStartTime
estimatedRunTime = (currentTimestamp-taskStartTime)/progress
estimateEndTime` = currentTimestamp+averageRunTime
```

其中，currentTimestamp为当前时刻；taskStartTime为该任务的启动时刻；averageRunTime为已经成功运行完成的任务的平均运行时间。这样，MRv2总是选择（estimateEndTime-estimateEndTime·）差值最大的任务，并为之启动备份任务。为了防止大量任务同时启动备份任务造成的资源浪费，MRv2为每个作业设置了同时启动的备份任务数目上限。

推测执行机制实际上采用了经典的算法优化方法：以空间换时间，它同时启动多个相同任务处理相同的数据，并让这些任务竞争以缩短数据处理时间。显然，这种方法需要占用更多的计算资源。在集群资源紧缺的情况下，应合理使用该机制，争取在多用少量资源的情况下，减少作业的计算时间。

五、Hadoop面试题——优化问题

1、MapReduce跑得慢的原因？（☆☆☆☆☆）

Mapreduce 程序效率的瓶颈在于两点：

1）计算机性能

CPU、内存、磁盘健康、网络

2）I/O 操作优化

（1）数据倾斜

（2）map和reduce数设置不合理

（3）reduce等待过久

（4）小文件过多

（5）大量的不可分块的超大文件

（6）spill次数过多

（7）merge次数过多等。

2、MapReduce优化方法（☆☆☆☆☆）

数据输入阶段

1）合并小文件：在执行mr任务前将小文件进行合并，大量的小文件会产生大量的map任务，增大map任务装载次数，而任务的装载比较耗时，从而导致 mr 运行较慢。

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

map阶段

1）减少spill次数：通过调整io.sort.mb及sort.spill.percent参数值，增大触发spill的内存上限，减少spill次数，从而减少磁盘 IO。

2）减少merge次数：通过调整io.sort.factor参数，增大merge的文件数目，减少merge的次数，从而缩短mr处理时间。

3）在 map 之后先进行combine处理，减少 I/O。

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）合理设置reduce端的buffer，默认情况下，数据达到一个阈值的时候，buffer中的数据就会写入磁盘，然后reduce会从磁盘中获得所有的数据。也就是说，buffer和reduce是没有直接关联的，中间多个一个写磁盘->读磁盘的过程，既然有这个弊端，那么就可以通过参数来配置，使得buffer中的一部分数据可以直接输送到reduce，从而减少IO开销：mapred.job.reduce.input.buffer.percent，默认为0.0。当值大于0的时候，会保留指定比例的内存读buffer中的数据直接拿给reduce使用。这样一来，设置buffer需要内存，读取数据需要内存，reduce计算也要内存，所以要根据作业的运行情况进行调整。

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

数据倾斜问题

1）数据倾斜现象

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

数据大小倾斜——部分记录的大小远远大于平均值。

2）如何收集倾斜数据

在reduce方法中加入记录map输出键的详细情况的功能。

```
public static final String MAX_VALUES = "skew.maxvalues";
private int maxValueThreshold;

@Override
public void configure(JobConf job) {
     maxValueThreshold = job.getInt(MAX_VALUES, 100);
}

@Override
public void reduce(Text key, Iterator<Text> values,
                   OutputCollector<Text, Text> output,
                   Reporter reporter) throws IOException {
     int i = 0;
     while (values.hasNext()) {
         values.next();
         i++;
     }
     if (++i > maxValueThreshold) {
         log.info("Received " + i + " values for key " + key);
     }
}
```

3）减少数据倾斜的方法

方法1：抽样和范围分区

可以通过对原始数据进行抽样得到的结果集来预设分区边界值。

方法2：自定义分区

另一个抽样和范围分区的替代方案是基于输出键的背景知识进行自定义分区。例如，如果map输出键的单词来源于一本书。其中大部分必然是省略词（stopword）。那么就可以将自定义分区将这部分省略词发送给固定的一部分reduce实例。而将其他的都发送给剩余的reduce实例。

方法3：Combine

使用Combine可以大量地减小数据频率倾斜和数据大小倾斜。在可能的情况下，combine的目的就是聚合并精简数据。

3、HDFS小文件优化方法（☆☆☆☆☆）

1）HDFS小文件弊端：

HDFS上每个文件都要在namenode上建立一个索引，这个索引的大小约为150byte，这样当小文件比较多的时候，就会产生很多的索引文件，一方面会大量占用namenode的内存空间，另一方面就是索引文件过大是的索引速度变慢。

2）解决的方式：

（1）Hadoop本身提供了一些文件压缩的方案。

（2）从系统层面改变现有HDFS存在的问题，其实主要还是小文件的合并，然后建立比较快速的索引。

3）Hadoop自带小文件解决方案

（1）Hadoop Archive：是一个高效地将小文件放入HDFS块中的文件存档工具，它能够将多个小文件打包成一个HAR文件，这样在减少namenode内存使用的同时。

（2）Sequence file： sequence file由一系列的二进制key/value组成，如果为key小文件名，value为文件内容，则可以将大批小文件合并成一个大文件。

（3）CombineFileInputFormat： CombineFileInputFormat是一种新的inputformat，用于将多个文件合并成一个单独的split，另外，它会考虑数据的存储位置。