# HADOOP

## HDFS

### HDFS读流程

1. client向Namenode发送请求，Namenode会检查读权限，同时检查文件是否存在，如果都存在且正常，则会返回三个block块所在的datanode位置；
2. Client会向datanode节点请求，逐步请求datanode1,2,3，建立数据读取通道；
3. 假如在读取过程中发生错误，会尝试读取其他的副本，逐步往下，直到都尝试失败，会报读取异常；
4. 直到读取完成，释放连接；

### HDFS写流程

1. Client会向NN发送请求，Namenode检查写权限，同时向Datanode确认空间以及位置，检查通过会给客户端返回对应的datanode地址（取datanode节点与副本节点最大值）；
2. Cient连接datanode地址，同时会将数据按照block切分，生成两个队列，一个数据队列，一个ack 确认队列；
3. 从数据队列里开始写数据，把数据从硬盘读到缓存，发送到packet，发送完毕chunck验证，然后写完成之后，会做ack确认，确认之后，会从ack确认队列把对应block块移除；
4. 开始第二个副本的复制，从第一个副本复制，依次类推，复制第三个队列；复制失败会重试；（重试次数都是默认3次）
5. 开始第二个block的写入，依次内推；如果在写入过程中出现问题，首先会尝试向第二个datanode写入，同时会把这个节点向namenode反馈，标记为失败的节点，如果第二个节点还不不行，就尝试往第三个节点写入；同时如果在做ack确认时候出现异常，则会把原来写入的block删除，重新写入一次，在做ack确认；
6. 直到写入完成，释放连接；

### HDFS小文件问题

实践证明，文件系统中1000w个文件会占用3G的Namenode内存（生产中一般是12G左右，根服务器配置有关，不能超过24G），内存占用越大，性能越低，会导致性能低下（JVM GC等）

如果出现小文件过多的问题，核心是想办法合并小文件，主要是通过MAPfile，或者sequenceFile，？？？

SequenceFile 无序存储，按照键值对的方式存储，一般对小文件可以使用这种文件合并，

即将文件名作为key，文件内容作为value序列化到大文件中。

这种文件格式有以下好处：

支持压缩，且可定制为基于Record或Block压缩（Block级压缩性能较优）

本地化任务支持：因为文件可以被切分，因此MapReduce任务时数据的本地化情况应该是非常好的。

对key、value的长度进行了定义，(反)序列化速度非常快。

缺点是需要一个合并文件的过程，文件较大，且合并后的文件将不方便查看，

必须通过遍历查看每一个小文件。

MapFile 会对key建立索引文件，value按key顺序存储（简言之就是排序后的SequenceFile）

通过观察其目录结构可以看到MapFile由两部分组成，分别是data和index

index作为文件的数据索引，主要记录了每个Record的key值，以及该Record在文件中的偏移位置。

需注意的是，MapFile并不会把所有Record都记录到index中去，默认情况下每隔128条记录存储一个索引映射。

MapFile的KeyClass一定要实现WritableComparable接口,即Key值是可比较的。

基于MapFile的结构有:

ArrayFile 像我们使用的数组一样，key值为序列化的数字

SetFile 他只有key，value为不可变的数据

BloomMapFile 在 MapFile 的基础上增加了一个 /bloom 文件，包含的是二进制的过滤表，

在每一次写操作完成时，会更新这个过滤表。

### HDFS文件格式

### 序列化与反序列化（了解）

序列化（Serialization）是指把结构化对象转化为字节流。简单理解就是把内存的东西写到磁盘就是序列化；

反序列化（Deserialization）是序列化的逆过程。即把字节流转回结构化对象。

简单理解就是把磁盘的东西读到内存是反序列化；

紧凑：高效使用存储空间。

快速：读写数据的额外开销小

可扩展：可透明地读取老格式的数据

互操作：支持多语言的交互

hadoop的序列化比java的序列化更高效。快捷

ps：Hadoop的序列化格式：Writable

序列化在分布式环境的两大作用：（进程间通信），（永久存储）。

Writable

write 是把每个对象序列化到输出流

readFields是把输入流字节反序列化

在这里注意如果使用ArrayWritable，需要构造函数，

class LongArrayWritable extends ArrayWritable{

public LongArrayWritable() {

super(LongWritable.class);

}

}

### hadoop的压缩codec:

【使用命令hadoop checknative -a 检查本地压缩库是否存在】

如果不存在，替换native文件，然后再次验证，就会成功

Codec为压缩，解压缩的算法实现。在Hadoop中，codec由CompressionCode的实现来表示。

支持的算法压缩比较：

原来的文件8.3g 压缩后： 压缩速度； 解压速度：

gzip： 1.8 17.5m/s 58m/s

bzip2： 1.1 2.4 9.5

## MapReduce

### MapReduce核心原理

Mapreduce核心是map跟reduce，map就像映射一一对应的意思在单个节点做map映射操作；reduce就是聚合，在hadoop分布式系统中会将多个节点的数据做汇总聚合。

### MapReduce过程

#### Map阶段

1. input阶段

主要是读取数据，确定map个数，如何确定？首先一般情况下是根据block一一对应，特殊情况文件不能分割（bzip2。），就是默认一个maptask，这里还有个公式就是判断具体的map数，max？？？

1. 复写Map函数

Map会调用多次，多次是几次呢？多次就是maptask的数量。

Set基本环境；这个只会调用一次；

要重点掌握mapreduce书写流程，以wordcount为例，\*\*\*\*\*

#### Shuffle过程

写入缓存区（默认100M）bufferarray，当环形缓存区到达0.8（阈值）的比例时候，开始溢写磁盘，生成maptask个文件，然后会做排序（快排）、合并，等待reducetask来拉取；

1.每个map有一个环形内存缓冲区，用于存储任务的输出。默认大小100MB（io.sort.mb属性），

一旦达到阀值0.8（io.sort.spill.percent）,一个后台线程把内容写到(spill)磁盘的指定目录（mapred.local.dir）

下的新建的一个溢出写文件。

2.写磁盘前，要partition,sort。如果有combiner，combine排序后数据。

3.等最后记录写完，合并全部溢出写文件为一个分区且排序的文件。

1.Reducer通过Http方式得到输出文件的分区。

2.TaskTracker为分区文件运行Reduce任务。复制阶段把Map输出复制到Reducer的内存或磁盘。

一个Map任务完成，Reduce就开始复制输出。

3.排序阶段合并map输出。然后走Reduce阶段。

#### Reduce过程

将reducetask来拉取过来的数据继续做排序合并（归并排序），执行redcue函数操作；

### MapReduce源码过程

？？？(先看主要函数、方法，回头再细讲，提高课程)

### MapRecude调优

调优跟随上面的源码一起

1. 配置合适的reduce数量
2. Map数量不能配置（他只能块数量一一对应）
3. 。。。

1.如果存在大量的小数据文件，可以使用SequenceFile、自定义的CombineFileInputFormat

2.推测执行在整个集群上关闭，特定需要的作业单独开启，一般可以省下约5%~10%的集群资源

mapred.map.task.speculative.execution=true;

mapred.reduce.task.speculative.execution=false;

3.开启jvm重用

mapred.job.reuse.jvm.num.tasks=-1

4.增加InputSplit大小

mapred.min.split.size=268435456

5.增大map输出的缓存

io.sort.mb=300

6.增加合并spill文件数量

io.sort.factor=50

7.map端输出压缩，推荐LZO压缩算法

mapred.compress.map.output=true;

mapred.map.output.compression.codec=com.hadoop.compression.lzo.LzoCodec;

8.增大shuffle复制线程数

mapred.reduce.parallel.copies=15

### MapReduce输入的处理类

FileInputFormat是所有以文件作为数据源的InputFormat实现的基类，FileInputFormat保存作为job输入的所有文件，

并实现了对输入文件计算splits的方法。

InputFormat 负责处理MR的输入部分.有三个作用:

验证作业的输入是否规范.

把输入文件切分成InputSplit.

提供RecordReader 的实现类，把InputSplit读到Mapper中进行处理.

TextInputFormat是默认的处理类，处理普通文本文件

文件中每一行作为一个记录，他将每一行在文件中的起始偏移量作为key，每一行的内容作为value。

默认以\n或回车键作为一行记录。TextInputFormat继承了FileInputFormat。

自定义输入格式

1）继承FileInputFormat基类。

2）重写里面的getSplits(JobContext context)方法。

3）重写createRecordReader(InputSplit split,TaskAttemptContext context)方法。

MapReduce其他输入类

DBInputFormat

CombineFileInputFormat

相对于大量的小文件来说，hadoop更合适处理少量的大文件。

CombineFileInputFormat可以缓解这个问题，它是针对小文件而设计的。

KeyValueTextInputFormat

当输入数据的每一行是两列，并用tab分离的形式的时候，

KeyValueTextInputformat处理这种格式的文件非常适合。

NLineInputformat

NLineInputformat可以控制在每个split中数据的行数。

SequenceFileInputformat?

当输入文件格式是sequencefile的时候，要使用SequenceFileInputformat作为输入。

### MapReduce的输出

TextOutputFormat

默认的输出格式，key和value中间值用tab隔开的。

DBOutputFormat

SequenceFileOutputformat

将key和value以sequencefile格式输出。

SequenceFileAsOutputFormat

将key和value以原始二进制的格式输出。

MapFileOutputFormat

将key和value写入MapFile中。由于MapFile中的key是有序的，所以写入的时候必须保证记录是按key值顺序写入的。

MultipleOutputFormat

默认情况下一个reducer会产生一个输出，但是有些时候我们想一个reducer产生多个输出，MultipleOutputFormat和MultipleOutputs可以实现这个功能。

Map任务的数量？

一个InputSplit对应一个Map task。

InputSplit的大小是由Math.max(minSize, Math.min(maxSize, blockSize))决定。

单节点一般10—100个map task。

map task执行时长不建议低于1分钟，否则效率低。

Partitioner编程

Partitioner是partitioner的基类，如果需要定制partitioner也需要继承该类。

job的reduce task数量默认是1，由参数mapreduce.job.reduces设定。

默认partitioner是HashPartitioner。

自定义partitioner类，需要设置job.setNumReduceTasks(Integer。parsint（args【1】）)

如果设置job.setNumReduceTasks(0)，那么不会执行reduce任务。同时，mapper的输出也不会排序、分组、合并。

排序编程：

方式一：继承WritableComparator

方式二：自定义k2类型，实现writeablecomparable，复写序列化，反序列化

在map和reduce阶段进行排序时，比较的是k2。v2是不参与排序比较的。如果要想让v2也进行排序，需要把k2和v2组装成新的类，作为k2，才能参与比较。

//无参构造方法，切记，切记

public MySort() {

super(K2.class, true);

}

分组编程：

job.setGroupingComparatorClass(...);

//无参构造方法，切记，切记

public MyGroup() {

super(K2.class, true);

}

Combiners编程

每一个map可能会产生大量的输出，combiner的作用就是在map端对输出先做一次合并，以减少传输到reducer的数据量

combiner最基本是实现本地key的归并，combiner具有类似本地的reduce功能。

注意：Combiner的输出是Reducer的输入，Combiner绝不能改变最终的计算结果。所以从我的想法来看，Combiner只应该用于那种Reduce的输入key/value与输出key/value类型完全一致，

且不影响最终结果的场景。比如累加，最大值等。

缓存：

1.加载文件job.addCacheFile(new URI("/hello#hel"));在这里hel是连接文件，

在map或者reduce中就可以直接使用hel作为文件名。

2.加载jar包job.addArchiveToClassPath(new Path("/mj.jar"));