HDFS

HDFS文件块大小

默认大小在hadoop2.x为128MB，老版本为64MB（在hdfs-site.xml中，增加全局参数dfs.block.size）

如何得来：

- 默认寻址时间约为10ms，即查找到目标的时间为10ms
- 寻址时间是传输时间的1%时，为最佳状态，在此状态下，传输时间为1000ms=1s
- 目前一般机器的传输速率为100MB/s（机械硬盘，如果是固态硬盘则更高），所以传输大小约为100MB（主要看硬件性能好不好，主要决定因素）

考虑因素

- 减少硬盘寻道时间(disk seek time)
- 减少Namenode内存消耗
- Map崩溃问题
- 监管时间问题
- 问题分解问题
- 约束Map输出

块不能设置的太小

- 同样的传输数据总量，块小的话，则块的数量就多，就导致增加寻址时间，由于数据块在硬盘上非连续存储，所以随机寻址较慢，读越多的数据块就增加了总的硬盘寻时间。（主要原因）
- 减少NameNode内存消耗。由于NameNode记录着DataNode中的数据块信息，若数据块大小设置过小，则数据块数量增多，需要维护的数据块信息就会增多，从而消耗NameNode的内存。

块不能设置的太大

- MapReduce中的map任务通常一次处理一个块中的数据，因此，如果任务数太少（少于集群中的节点数量）且任务处理量大，作业的运行速度就会变慢。
- Map崩溃问题：系统需要重新启动，启动过程需要重新加载数据，数据块越大，数据加载时间越长，系统恢复过程越长。
- 监管时间问题。**主节点监管其他节点的情况，每个节点会周期性的与主节点进行汇报通信。**倘若某一个节点保持沉默的时间超过一个预设的时间间隔，主节点会记录这个节点状态为死亡，并将该节点的数据转发给别的节点。而这个“预设时间间隔”是从数据块的角度大致估算的。（加入对64MB的数据块，我可以假设你10分钟之内无论如何也能解决完了吧，超过10分钟还没反应，那我就认为你出故障或已经死了。）64MB大小的数据块，其时间尚可较为精准地估计，如果我将数据块大小设为640MB甚至上G，那这个“预设的时间间隔”便不好估算，估长估短对系统都会造成不必要的损失和资源浪费。

Shell操作常用命令

```
#-ls 显示目录信息 -R递归查看
#-cp ：从HDFS 的一个路径拷贝到HDFS 的另一个路径
#-mv：在HDFS 目录中移动文件
#-mkdir：在HDFS 上创建目录，-p创建多级路径
#-cat：显示文件内容
```

```
#-tail：显示一个文件的末尾
#-rm：删除文件或文件夹 -R 递归删除
#-rmdir：删除空目录

#-appendToFile：追加一个文件到已经存在的文件末尾
hadoop fs -appendToFile liubei.txt /sanguo/shuguo/kongming.txt


#-moveFromLocal：从本地剪切粘贴到HDFS
hadoop fs -moveFromLocal ./kongming.txt /sanguo/shuguo

#-chgrp 、-chmod、-chown：Linux 文件系统中的用法一样，修改文件所属权限
hadoop fs -chmod 666 /sanguo/shuguo/kongming.txt
hadoop fs -chown atguigu:atguigu /sanguo/shuguo/kongming.txt

#-copyFromLocal -put：从本地文件系统中拷贝文件到HDFS 路径去
hadoop fs -put README.txt /

#-copyToLocal -get：从HDFS 拷贝到本地
hadoop fs -get /sanguo/shuguo/kongming.txt ./

#-getmerge：合并下载多个文件
hadoop fs -getmerge /user/atguigu/test/* ./zaiyiqi.txt

#-du 统计文件夹的大小信息
hadoop fs -du -h /user/atguigu/test

#-setrep：设置HDFS 中文件的副本数量
#这里设置的副本数只是记录在NameNode 的元数据中，是否真的会有这么多副本，还得看DataNode 的数量。因为目前只有3 台设备，最多也就3 个副本，只有节点数的增加到10台时，副本数才能达到10。
hadoop fs -setrep 10 /sanguo/shuguo/kongming.txt
```

开发常用方法

```
// 1 获取文件系统
Configuration configuration = new Configuration();
configuration.set("dfs.replication", "2");
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration, "atguigu");
// 2 方法操作

// 3 关闭资源
fs.close();
```

- fs.copyFromLocalFile()：文件上传
  - Path from 文件来源地址
  - Path to 文件目的地址
  - 配置参数优先级排序：客户端代码中设置的值 > ClassPath 下的用户自定义配置文件 > 然后是服务器的默认配置
- fs.copyToLocalFile()：文件下载
  - boolean delSrc 指是否将原文件删除
  - Path src 指要下载的文件路径
  - Path dst 指将文件下载到的路径

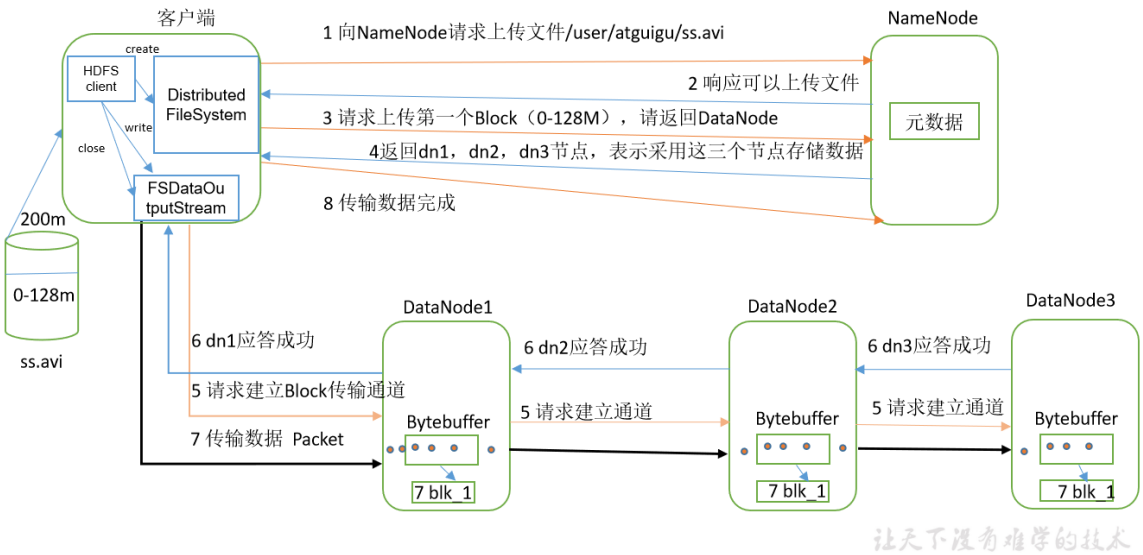
    - boolean useRawLocalFileSystem 是否开启文件校验
- fs.delete(Path p, boolean recursive)：删除文件或文件夹
    - recursive：当删除的是递归文件夹时，需要设置为true，否则抛异常，其它情况则无所谓
- fs.rename(Path before, Path after)：文件更名
- fs.listFiles(Path p, boolean recursive)：查看文件详情，recursive设置是否递归查看

```
RemoteIterator<LocatedFileStatus> listFiles = fs.listFiles(new
Path("/zzh"), true);
while (listFiles.hasNext()) {
    LocatedFileStatus fileStatus = listFiles.next();

    // 输出详情
    // 文件名称
    System.out.println(fileStatus.getPath().getName());
    // 长度
    System.out.println(fileStatus.getLen());
    // 权限
    System.out.println(fileStatus.getPermission());
    // 分组
    System.out.println(fileStatus.getGroup());

    // 获取存储的块信息
    BlockLocation[] blockLocations = fileStatus.getBlockLocations();
    for(BlockLocation blockLocation : blockLocations) {
        // 获取块存储的主机节点
        String[] hosts = blockLocation.getHosts();

        for(String host : hosts) {
            System.out.println(host);
        }
    }
    System.out.println("----------------分割线-------------------");
}
```

- fs.listStatus(Path p)：获取文件状态

```
FileStatus[] fileStatuses = fs.listStatus(new Path("/zzh"));
for(FileStatus fileStatus : fileStatuses) {
    // getPath().getName()是常用的获取文件名方式
    if (fileStatus.isFile()) {
        System.out.println("f:" + fileStatus.getPath().getName());
    } else if (fileStatus.isDirectory()) {
        System.out.println("d:" + fileStatus.getPath().getName());
    }
}
```

HDFS写流程

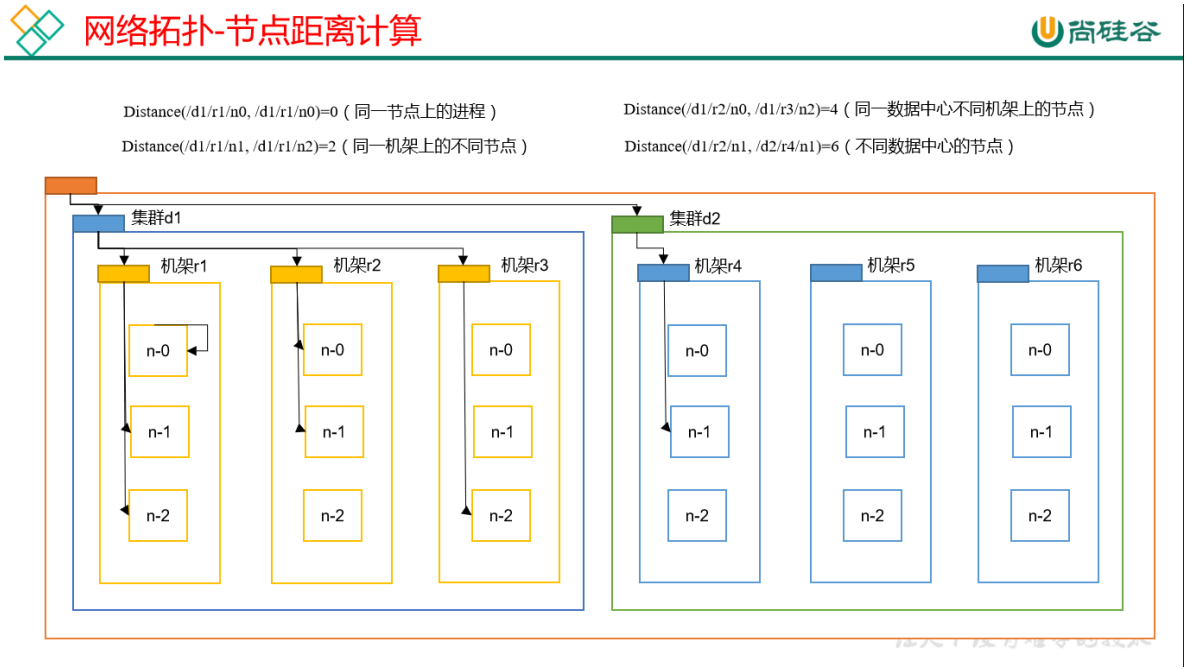


1. 客户端通过Distributed FileSystem模块向NameNode请求上传文件，NameNode检查目标文件是否已存在，父目录是否存在。
2. NameNode返回是否可以上传。
3. 客户端请求第一个 Block上传到哪几个DataNode服务器上。
4. NameNode返回3个DataNode节点，分别为dn1、dn2、dn3。
5. 客户端通过FSDataOutputStream模块请求dn1上传数据，dn1收到请求会继续调用dn2，然后dn2调用dn3，将这个通信管道建立完成。
6. dn1、dn2、dn3逐级应答客户端。
7. 客户端开始往dn1上传第一个Block（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）。

网络拓扑-节点距离计算

在HDFS写数据的过程中，NameNode会选择距离待上传数据最近距离的DataNode接收数据。

节点距离：两个节点到达最近的共同祖先的距离总和。

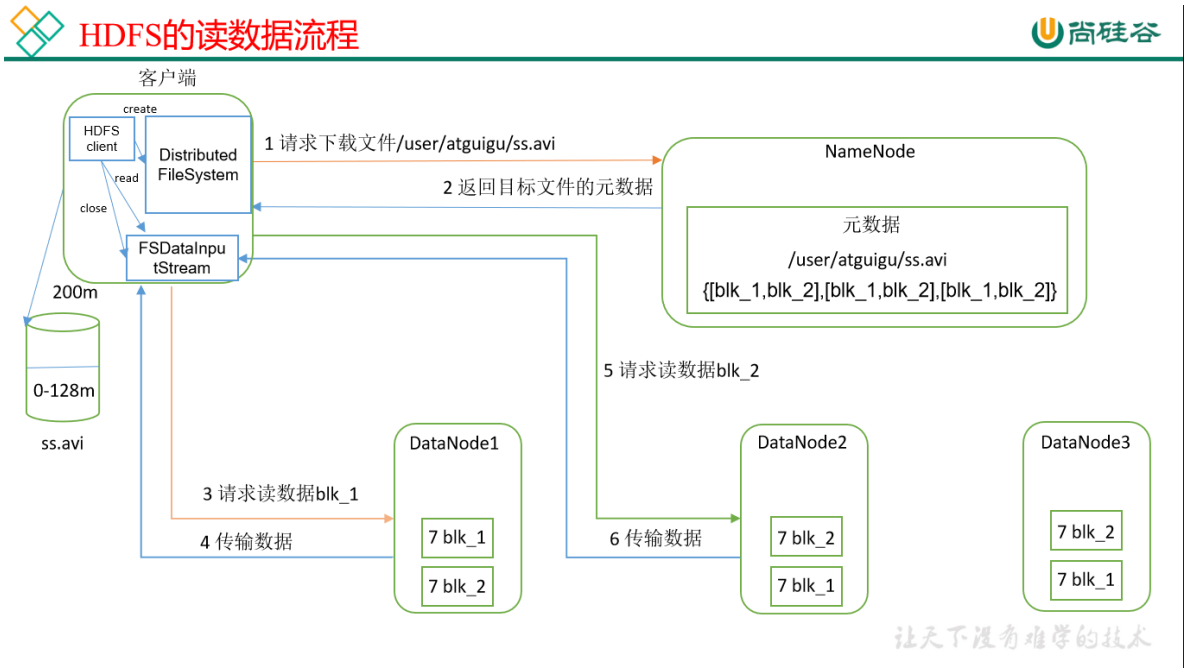


机架感知

假设有三个副本

- 第一个副本在client所处的节点
- 第二个副本在与client相同机架不同节点上
- 第三个副本在与client不同机架的节点上

HDFS读流程



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。

NameNode

SecondaryNameNode

DataNode

MapReduce

优缺点

优点

- 易于编程
- 良好的扩展性
- 高容错性
- 适合PB级别以上海量数据的**离线处理**

缺点

- 不擅长实时计算
- 不擅长流式计算

  MapReduce的输入数据集是静态的
- 不擅长DAG计算

  每个MapReduce作业的输出结果都会写入到磁盘，会造成大量的磁盘IO，导致性能非常的底下

编程规范

程序分为三个阶段

Mapper阶段

- 用户自定义的Mapper要继承父类
- Mapper的输入数据是 $<K,V>$ 对的形式（类型可自定义）
- Mapper的业务逻辑写在map()中
- Mapper的输出数据是 $<K,V>$ 对的形式（类型可自定义）
- map()方法（MapTask进程）对每个 $<K,V>$ 调用一次

Reducer阶段

- 用户自定义的Reducer要继承父类
- Reducer的输入数据对应着Mapper的输出数据类型
- Reducer的业务逻辑写在reduce()中
- RecuderTask进程对每一组相同 k 的 $<K,V>$ 组调用一次reduce()方法

Driver

```
// 1、获取job对象
Configuration conf = new Configuration();
Job job = Job.getInstance(conf);

// 2、设置jar存储位置
```

```
job.setJarByClass(WordCountDriver.class);

// 3、关联Mapper和Reducer类
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);

// 4、设置Mapper阶段输出数据的key和value类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);

// 5、设置最终数据输出的key和value类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

// 6、设置输入路径和输出路径
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 7、提交job
boolean result = job.waitForCompletion(true);
```

序列化

- 数据紧凑

  > 带宽是集群中信息传递的最宝贵的资源，所以我们必须设法缩小传递信息的大小。为了更好的控制序列化整个流程使用Writable对象，java序列化过程中会保存类的所有信息以及依赖等，Hadoop序列化不需要。

- 对象可重用

  > JDK的反序列化会不断地创建对象，这肯定会造成一定的系统开销，但是在hadoop反序列化中，能重复的利用一个对象的readField方法来重新产生不同的对象。

- 可扩展性

  > hadoop自己写序列化很容易，可以通过实现hadoop的Writable接口实现序列化，或者实现WritableComparable接口实现可比较大小的序列化对象。

- 互操作

  > 支持多语言的交互

步骤

- 必须实现 $Writable$ 接口
- 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

```
public FlowBean() {
    super();
}
```

- 重写序列化方法

```java
@Override
public void write(DataOutput out) throws IOException {
    out.writeLong(upFlow);
    out.writeLong(downFlow);
    out.writeLong(sumFlow);
}
```

- 重写反序列化方法

```java
@Override
public void readFields(DataInput in) throws IOException {
    upFlow = in.readLong();
    downFlow = in.readLong();
    sumFlow = in.readLong();
}
```

- 注意反序列化的顺序和序列化的顺序完全一致
- 要想把结果显示在文件中，需要重写toString()，可用"\t"分开，方便后续用。
- 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 过程要求对key 必须能排序。

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

并行度

数据块：Block 是HDFS 物理上把数据分成一块一块。

数据切片：数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。

- 一个job的Map阶段并行度由客户端在提交job时的切片数决定
- 每一个split切片分配一个MapTask并行实例处理
- 默认情况下，切片大小=BlockSize
- 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

  若数据集下有两个文件，t1(200M)，t2(100M)，切片大小为128M，则切片情况为：文件t1切分为2块切片（128M，72M），文件t2切分为1块（100M），总共3块。

> 实际上，在切片时，不是刚好按切片大小在进行切片，在文件剩余大小 >= 切片大小 的 1.1 倍时，才会切片，并且切128M。如文件大小为129M，依然是算一个切片

作业提交源码部分解析

```java
//进入 waitForCompletion()
//进入 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);
// 6）提交Job,返回提交状态
status = submitClient.submitJob(jobId,
submitJobDir.toString(), job.getCredentials());
```

FileInputFormat切片机制

FileInputFormat.class -> getSplits()：

切片大小：

```
// blockSize 在本地模型运行时为32M，在YARN模式运行时为128M
Math.max(minSize, Math.min(maxSize, blockSize))
```

判断条件：

```
  (double)bytesRemaining / (double)splitSize > 1.1D
```

添加切片信息 splitSize为切片大小：

```
  splits.add(this.makeSplit(path, length - bytesRemaining, splitSize,
blkLocations[blkIndex].getHosts(), blkLocations[blkIndex].getCachedHosts()));
```

最后一块切片，

```
  if (bytesRemaining != 0L) {
      blkIndex = this.getBlockIndex(blkLocations, length - bytesRemaining);
      // 添加切片信息
      splits.add(this.makeSplit(path, length - bytesRemaining,
bytesRemaining,  blkLocations[blkIndex].getHosts(),
blkLocations[blkIndex].getCachedHosts()));
      }
```

CombineTextInputFormat切片机制

背景

框架默认的TextInputFormat切片机制是对任务按文件规划切片，不管文件多小，都会是一个单独的切片，都会交给一个MapTask，**这样如果有大量小文件，就会产生大量的MapTask**，处理效率极其低下。

应用场景

CombineTextInputFormat用于小文件过多的场景，它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个MapTask处理。

虚拟存储切片最大值设置

CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);// 4m

注意：虚拟存储切片最大值设置最好根据实际的小文件大小情况来设置具体的值。

切片机制

生成切片过程包括：虚拟存储过程和切片过程二部分。

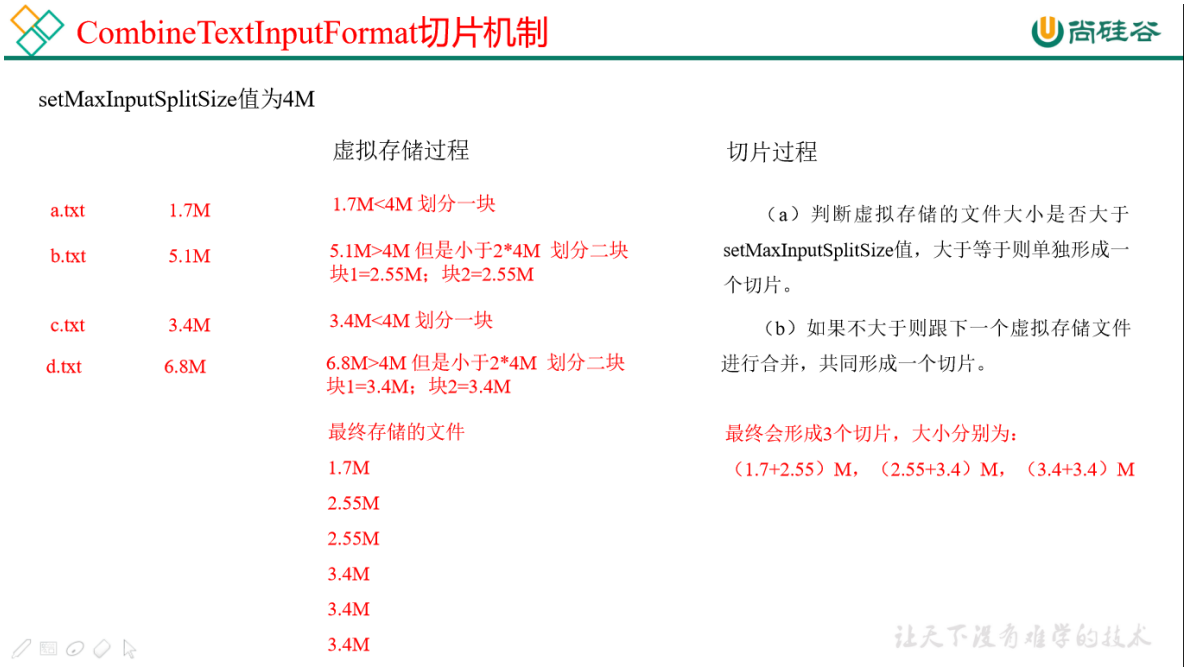
虚拟存储过程

将输入目录下所有文件大小，依次和设置的setMaxInputSplitSize值比较，如果不大于设置的最大值，逻辑上划分一个块。如果输入文件大于设置的最大值且大于两倍，那么以最大值切割一块；当剩余数据大小超过设置的最大值且不大于最大值2倍，此时将文件均分成2个虚拟存储块（防止出现太小切片）。

```
// maxSize默认为4M
//-----对每一个读入文件调用：
// 10M的文件会被划分为 4M 3M 3M
do {
    if (maxSize == 0) {
        myLength = left;
    } else {
        if (left > maxSize && left < 2 * maxSize) {
            myLength = left / 2;
        } else {
            myLength = Math.min(maxSize, left);
        }
    }
    OneBlockInfo oneblock = new OneBlockInfo(stat.getPath(),myOffset,
myLength, locations[i].getHosts(),locations[i].getTopologyPaths());
    left -= myLength;
    myOffset += myLength;
    blocksList.add(oneblock);
} while (left > 0);

```

切片过程

- 判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。
- 如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。




```
// 如果不设置InputFormat，它默认用的是TextInputFormat.class
job.setInputFormatClass(CombineTextInputFormat.class);

//虚拟存储切片最大值设置4m
CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);
```

自定义InputFormat

需要自定义两个类分别继承InputFormat和RecordFormat，并实现相关方法

InputFormat

```
public abstract class InputFormat<K, V> {
    public abstract List<InputSplit> getSplits(JobContext context);
    RecordReader<K, V> createRecordReader(InputSplit split,
TaskAttemptContext context);
}
//这些方法展示了InputFormat类的两个功能:
//  将输入文件切分为map处理所需的split
//  创建RecordReader类，它将从一个split生成键值对序列
```

RecordReader

```java
public abstract class RecordReader<Key, Value> implements Closeable {
    public abstract void initialize(InputSplit split, TaskAttemptContext context);
    public abstract boolean nextKeyValue() throws IOException, InterruptedException;
    public abstract Key getCurrentKey() throws IOException, InterruptedException;
    public abstract Value getCurrentValue() throws IOException, InterruptedException;
    public abstract float getProgress() throws IOException, InterruptedException;;
    public abstract close() throws IOException;
}
// 为每个split创建一个RecordReader实例,该实例调用getNextKeyValue并返回一个布尔值
// 组合使用InputFormat和RecordReader可以将任何类型的输入数据转换为MapReduce所需的键值对
```

分区

对每个map()方法写入的数据进行分区

HashPartitioner

默认分区器

```java
public int getPartition(K key, V value, int numReduceTasks) {
    return (key.hashCode() & 2147483647) % numReduceTasks;
}
// 2147483647 Integer.MAX_VALUE 2^63 - 1 (011111...111)
// 与操作后，保证哈希值是正数

// numReduceTasks: 默认值为1 可以通过job.setNumReduceTasks(num)来设置
```

自定义分区器

```java
public class ProvincePartitioner extends Partitioner<Text, FlowBean> {
    @Override
    public int getPartition(Text key, FlowBean value, int i) {
        String prePhoneNum = key.toString().substring(0, 3);
        switch (prePhoneNum) {
            case "136": return 0;
            case "137": return 1;
            case "138": return 2;
            case "139": return 3;
            default: return 4;
        }
    }
}
// 只需要继承Partitioner类，并且重写getPartition方法即可
// 分区号必须从零开始，逐一累加

// 最好保证自定义的分区数reduceTask 跟 设置的分区数相同 numPartition，通常取余数
// 假设设置的分区数 numPartition = 5
// 当 reduceTask = 1时， 最终只生成一个分区，所有数据保存在里面
// 当 1 <  reduceTask < numPartition时，会有一部分分区数据没位置存储，报错
```

```
// 当 reduceTask == numPartition时，正常存储，按照自定义的逻辑存储在各个分区中
// 当 reduceTask > numPartition时，会有空的分区文件
```

排序

如果是自定义的类作为key键值，则需要实现 $WritableComparable$ 接口，因为MapReduce 默认对key排序

- 全排序

  ReduceTask值 = 1

- 部分排序（分区内排序）

  ReduceTask值 > 1s

Combiner

每一个map都可能会产生大量的本地输出，Combiner的作用就是对map端的输出先做一次合并，以减少在map和reduce节点之间的数据传输量，以提高网络IO性能，是MapReduce的一种优化手段之一。

Combiner在每一个MapTask所在的节点上运行，对每一个MapTask的输出结果进行局部汇总。

```
// 如果Combiner跟Reducer的逻辑相同，那么可以将Reducer绑定为Combiner，不需要再重写
job.setCombinerClass(WordcountReducer.class);
```

> 如果当前集群在很繁忙的情况下job就是设置了也不会执行Combiner。
>
> Conbiner的适用场景比如说在汇总统计时，就可以使用Conbiner，但是在求平均数的时候就不适用了。

使用时机

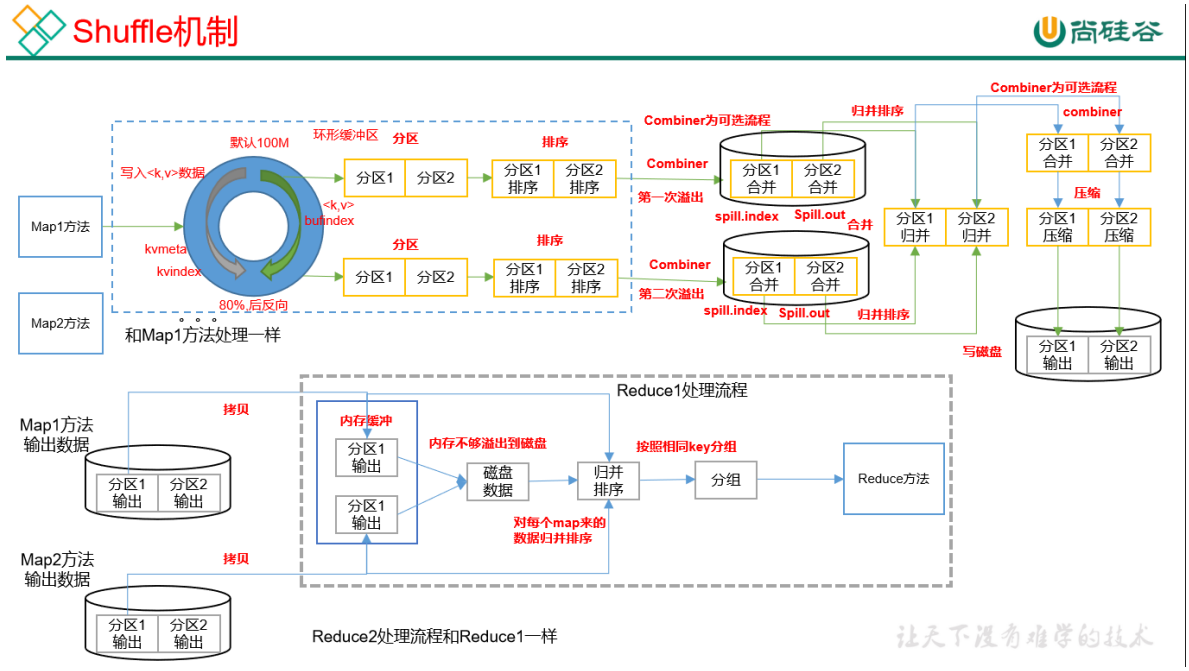
- Combiner不能影响最终输出结果
- Conbiner的适用场景比如说在汇总统计时，就可以使用Conbiner，但是在求平均数的时候就是适合适用了。

Combiner会在map端的哪个时期执行

实际上，Conbiner函数的执行时机可能会在map的merge操作完成之前，也可能在merge之后执行，这个**时机由配置参数min.num.spill.for.combine(该值默认为3)**，也就是说在map端产生的spill文件最少有min.num.spill.for.combine的时候，Conbiner函数会在merge操作合并最终的本机结果文件之前执行，否则在merge之后执行。通过这种方式，就可以在spill文件很多并且需要做conbine的时候，减少写入本地磁盘的数据量，同样也减少了对磁盘的读写频率，可以起到优化作业的目的。

shuffle机制

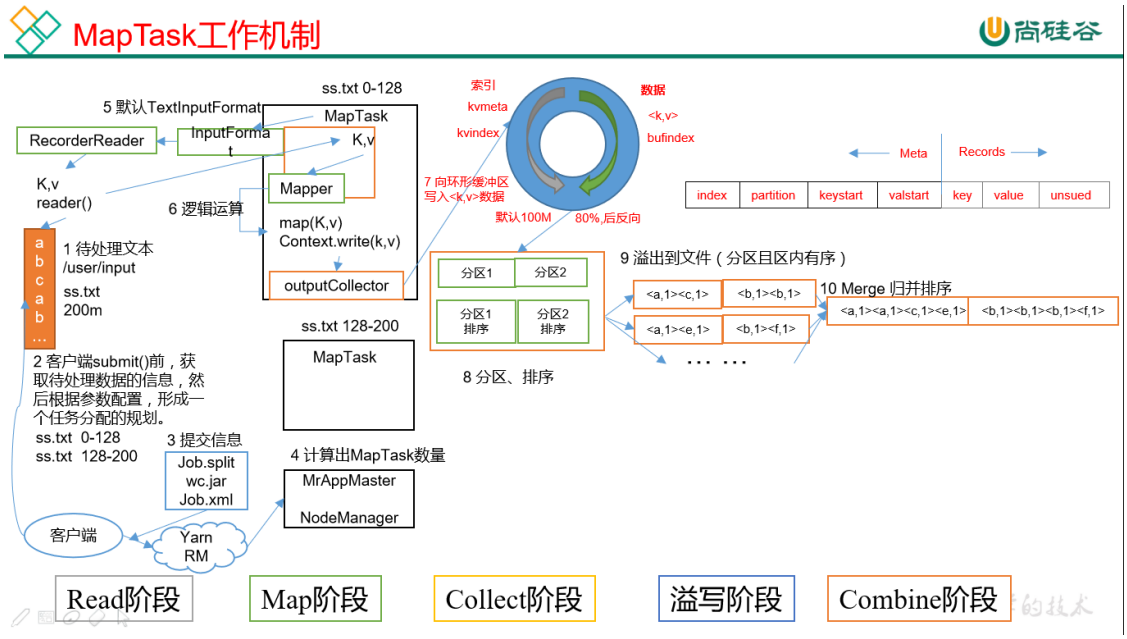
Map方法之后，Reduce方法之前的数据处理过程称之为Shuffle。



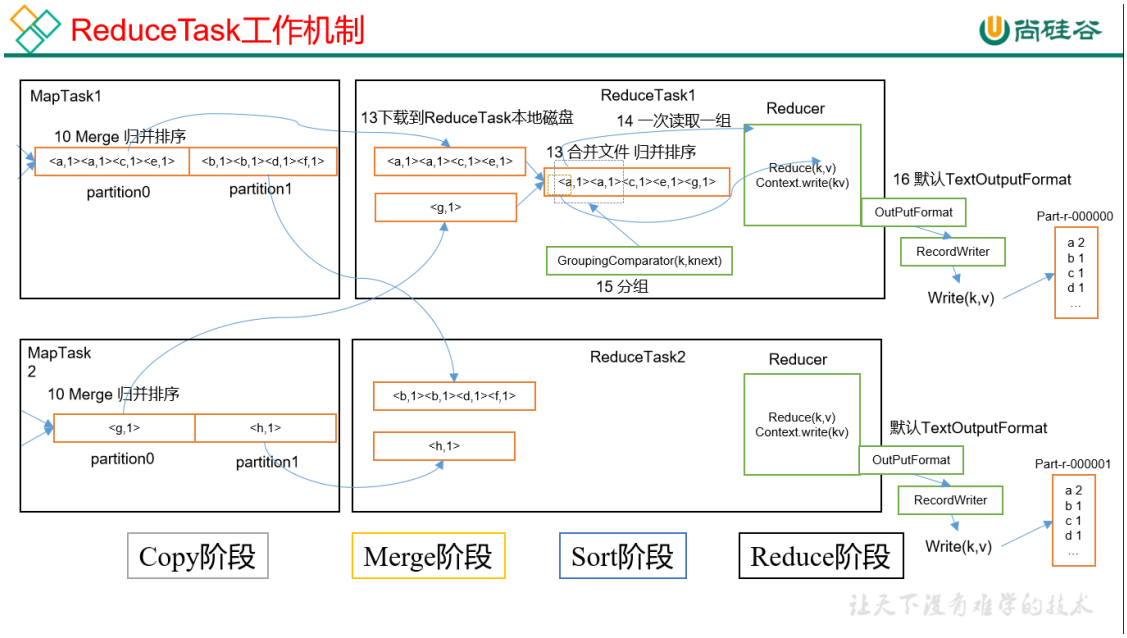
个人总结MR工作流程

- Driver端配置好job信息，提交job
- 提交过程中，会把job.split(切片信息)，job.xml(配置信息)，job.jar(项目jar包，集群模式才提交)提交至YARN资源管理器。
- 资源管理器根据split中的切片个数分配MapTask数量
- Map阶段：针对每个MapTask，对每条由InputFormat分割的数据，执行map()方法，由Collector收集器收集map()里的write(key, value)写出的数据（**数据经过 *Partitioner* 分区器分配好的分区信息记录到缓冲区的元数据区中**）写入环形缓冲区中（**数据量达到80%将会溢写到磁盘中，如果配置了 *Combiner* 的话，就在本地对溢写出的文件先进行一次key相同的值合并，再对文件归并排序合成一个大文件**）

> 环形缓冲区缓冲数据的过程可以通过Debug进入 map()方法中的context.write()方法查看。
> kvmeta.put()



- Reduce阶段：等待所有的MapTask执行完后，根据配置信息中的分区数，YARN分配ReduceTask数量，针对每个ReduceTask，通过shuffle，读取Map阶段写入磁盘的文件数据，根据分区号，将数据分配到相对应的分区中（**数据进行了混洗**）。针对每个分区，将key相同（**若是Bean类作为键，需要实现** $WritableComparable$ **接口，因为MapReduce 默认对key排序，若只需要Bean中的部分属性作为判断key相等条件，则需要继承** $WritableComparator$ **类来实现分组**）的value作为一组key-values，传递到reduce()方法中执行，并由context.write(k, v)写出。



- 根据OutputFormat方式写出到磁盘中

join

reduce 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。即：map阶段的主要任务是对不同文件中的数据打标签。

在reduce阶段：reduce函数获取key相同的来自File1和File2文件的value list， 然后对于同一个key，对File1和File2中的数据进行join（笛卡尔乘积），即：reduce阶段进行实际的连接操作。

缺点

由于合并的操作是在Reduce阶段完成的，所以Reduce端的处理压力大，MapTask节点的运算负载则很低，资源利用率不高，而且Reduce阶段极易产生数据倾斜。

map join

使用场景

Map Join适用于一张表十分小、一张表很大的场景。

优点

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。

具体方法

- 在Mapper的setup阶段，将文件读取到缓存集合中。
- 在驱动函数中加载缓存。

  在Driver缓存普通文件到Task运行节点。

```
// 加载缓存数据
job.addCacheFile(new URI("file:///e:/input/inputcache/pd.txt"));
// Map端Join的逻辑不需要Reduce阶段，设置reduceTask数量为0
job.setNumReduceTasks(0);
// 不需要设置Mapper数据类型，直接设置最终输出类型即可
```

数据清洗

在运行核心业务MapReduce程序之前，往往要先对数据进行清洗，清理掉不符合用户要求的数据。清理的过程往往只需要运行Mapper程序，不需要运行Reduce程序。

通常也伴随着使用计数器，

计数器常用计数器组、计数器名称的方式进行统计

```
contenxt.getCounter("groupName", "counterName").increment(num);
```

控制台上显示的键值对就是计数器的应用，每段首行的是组名，下面的就是不同的计数器名

DistributedCacheDriver

```
2020-01-09 16:30:29,406 INFO [org.apache.hadoop.mapreduce.Job] -
    File System Counters
        FILE: Number of bytes read=251
        FILE: Number of bytes written=280191
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
    Map-Reduce Framework
        Map input records=6
        Map output records=6
        Input split bytes=100
        Spilled Records=0
        Failed Shuffles=0
        Merged Map outputs=0
        GC time elapsed (ms)=11
        Total committed heap usage (bytes)=163053568
    File Input Format Counters
        Bytes Read=64
    File Output Format Counters
        Bytes Written=114
```

数据压缩

MR支持的压缩编码

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

Hadoop的编码/解码器

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

> Snappy压缩性能官方说明：On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

压缩参数配置

参数	默认值	阶段	建议
io.compression.codecs （在core-site.xml中配置）	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec	输入压缩	Hadoop使用文件扩展名判断是否支持某种编解码器
mapreduce.map.output.compress（在mapred-site.xml中配置）	false	mapper输出	这个参数设为true启用压缩
mapreduce.map.output.compress.codec（在mapred-site.xml中配置）	org.apache.hadoop.io.compress.DefaultCodec	mapper输出	企业多使用LZO或Snappy编解码器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress（在mapred-site.xml中配置）	false	reducer输出	这个参数设为true启用压缩
mapreduce.output.fileoutputformat.compress.codec（在mapred-site.xml中配置）	org.apache.hadoop.io.compress. DefaultCodec	reducer输出	使用标准工具或者编解码器，如gzip和bzip2
mapreduce.output.fileoutputformat.compress.type（在mapred-site.xml中配置）	RECORD	reducer输出	SequenceFile输出使用的压缩类型：NONE和BLOCK

数据流的压缩和解压缩

```
1 public class TestCompress {
2
3     public static void main(String[] args) throws Exception {
```

```
4
compress("e:/hello.txt","org.apache.hadoop.io.compress.BZip2Codec");
5   //      decompress("e:/hello.txt.bz2");
6       }
7       // 1、压缩
8       private static void compress(String filename, String method) throws
Exception {
9
10          // （1）获取输入流
11          FileInputStream fis = new FileInputStream(new File(filename));
12          Class codecClass = Class.forName(method);
13          CompressionCodec codec = (CompressionCodec)
ReflectionUtils.newInstance(codecClass, new Configuration());
14          // （2）获取输出流
15          FileOutputStream fos = new FileOutputStream(new File(filename +
codec.getDefaultExtension()));
16          CompressionOutputStream cos = codec.createOutputStream(fos);
17          // （3）流的对拷
18          IOUtils.copyBytes(fis, cos, 1024*1024*5, false);
19          // （4）关闭资源
20          cos.close();
21          fos.close();
22          fis.close();
23      }
24
25      // 2、解压缩
26      private static void decompress(String filename) throws
FileNotFoundException, IOException {
27
28          // （0）校验是否能解压缩
29          CompressionCodecFactory factory = new CompressionCodecFactory(new
Configuration());
30          CompressionCodec codec = factory.getCodec(new Path(filename));
31          if (codec == null) {
32              System.out.println("cannot find codec for file " + filename);
33              return;
34          }
35          // （1）获取输入流
36          CompressionInputStream cis = codec.createInputStream(new
FileInputStream(new File(filename)));
37          // （2）获取输出流
38          FileOutputStream fos = new FileOutputStream(new File(filename +
".decoded"));
39          // （3）流的对拷
40          IOUtils.copyBytes(cis, fos, 1024*1024*5, false);
41          // （4）关闭资源
42          cis.close();
43          fos.close();
44      }
45  }
```

Map输出端采用压缩

在Driver加入配置信息

```
// 开启map端输出压缩
configuration.setBoolean("mapreduce.map.output.compress", true);
// 设置map端输出压缩方式
configuration.setClass("mapreduce.map.output.compress.codec",
BZip2Codec.class, CompressionCodec.class);
```

Reduce输出端采用压缩

在Driver加入配置信息

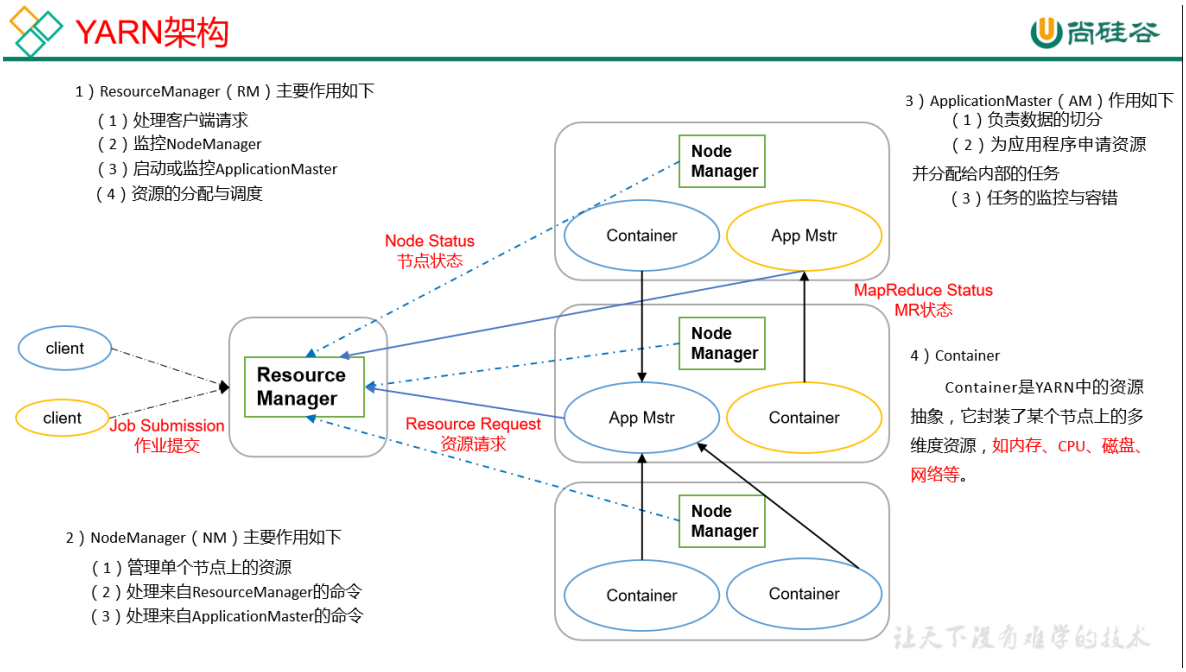
```
// 设置reduce端输出压缩开启
FileOutputFormat.setCompressOutput(job, true);
// 设置压缩的方式
FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);
```

> Mapper和Reducer保持不变

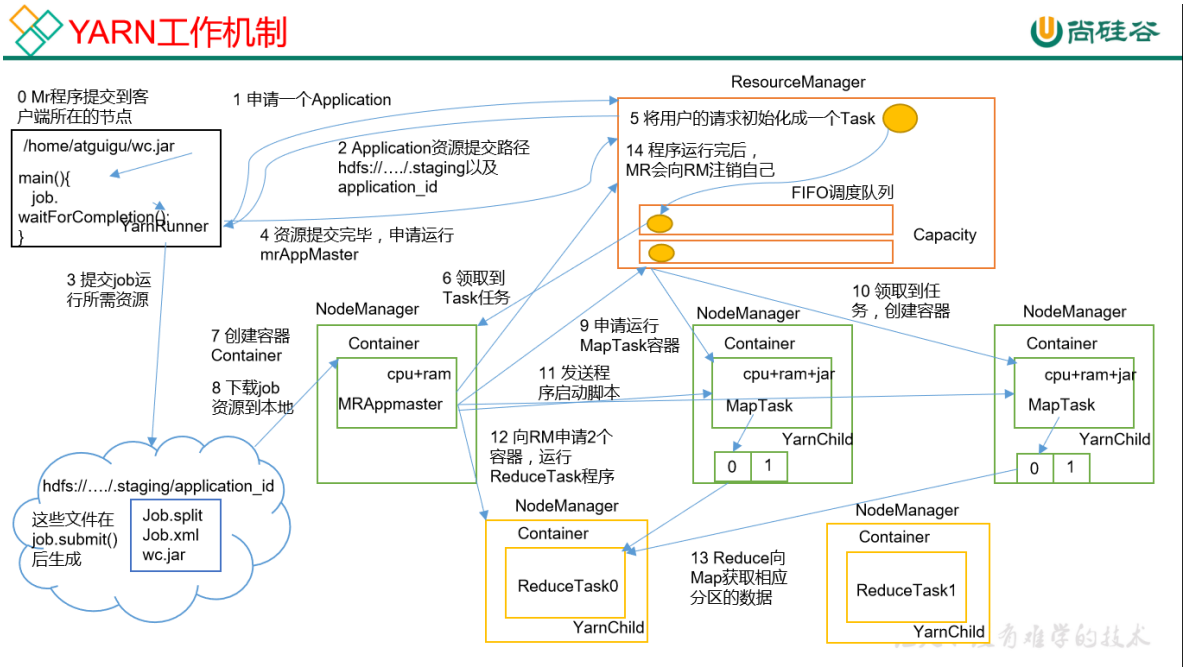
Yarn

Yarn是一个**资源调度平台**，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而MapReduce等运算程序则相当于运行于操作系统之上的应用程序。

Yarn架构



Yarn工作流程



1. MR程序提交到客户端所在的节点。
2. YarnRunner向ResourceManager申请一个Application。
3. RM将该应用程序的资源路径返回给YarnRunner。
4. 该程序将运行所需资源提交到HDFS上。
5. 程序资源提交完毕后，申请运行mrAppMaster。
6. RM将用户的请求初始化成一个Task。
7. 其中一个NodeManager领取到Task任务。
8. 该NodeManager创建容器Container，并产生MRAppmaster。
9. Container从HDFS上拷贝资源到本地。
10. MRAppmaster向RM 申请运行MapTask资源。
11. RM将运行MapTask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器。
12. MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动MapTask，MapTask对数据分区排序。
13. MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask。
14. ReduceTask向MapTask获取相应分区的数据。
15. 程序运行完毕后，MR会向RM申请注销自己。

资源调度器

- FIFO资源调度器
- Capacity资源调度器
- Fair资源调度器

任务的推测执行

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。

思考：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？

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。

执行推测任务的前提条件

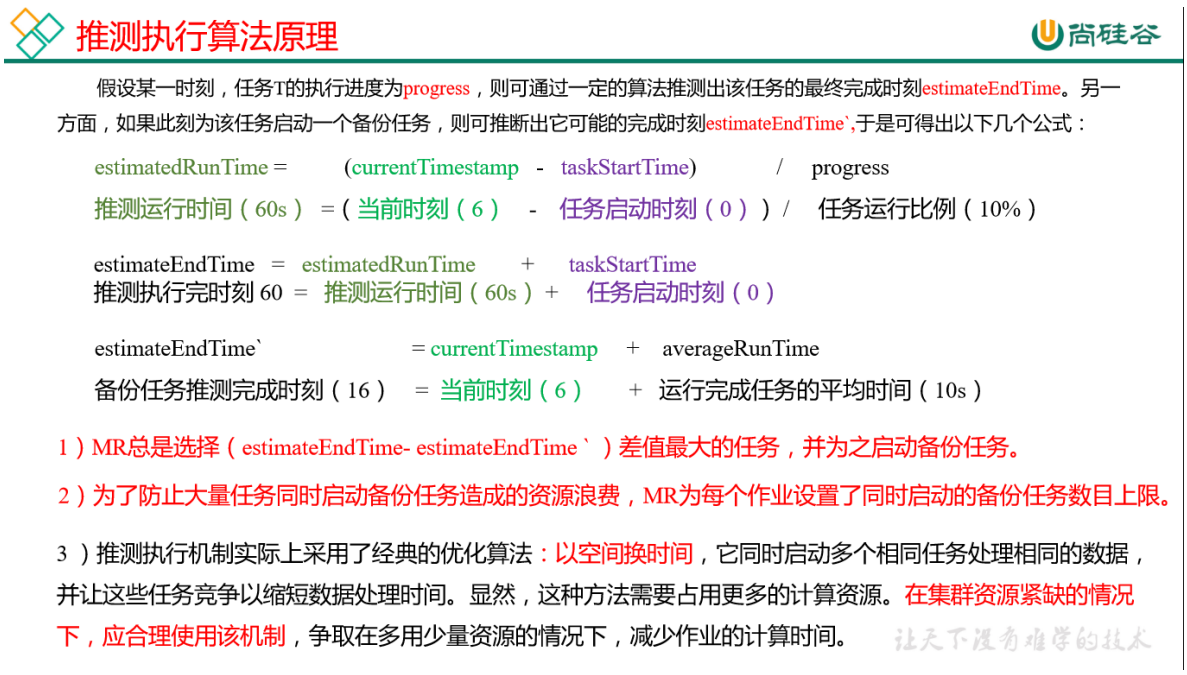
- 每个Task只能有一个备份任务
- 当前Job已完成的Task必须不小于0.05（5%）
- 开启推测执行参数设置。mapred-site.xml文件中默认是打开的。

```
<property>
    <name>mapreduce.map.speculative</name>
    <value>true</value>
    <description>If true, then multiple instances of some map tasks may be executed in parallel.</description>
</property>
<property>
    <name>mapreduce.reduce.speculative</name>
    <value>true</value>
    <description>If true, then multiple instances of some reduce tasks may be executed in parallel.</description>
</property>
```

不能启用推测执行机制情况

- 任务间存在严重的负载倾斜；
- 特殊任务，比如任务向数据库中写数据。

推测执行算法原理



失败

任务运行失败

Hadoop优化

MapReduce相关

MapReduce 程序效率的瓶颈

- 计算机性能

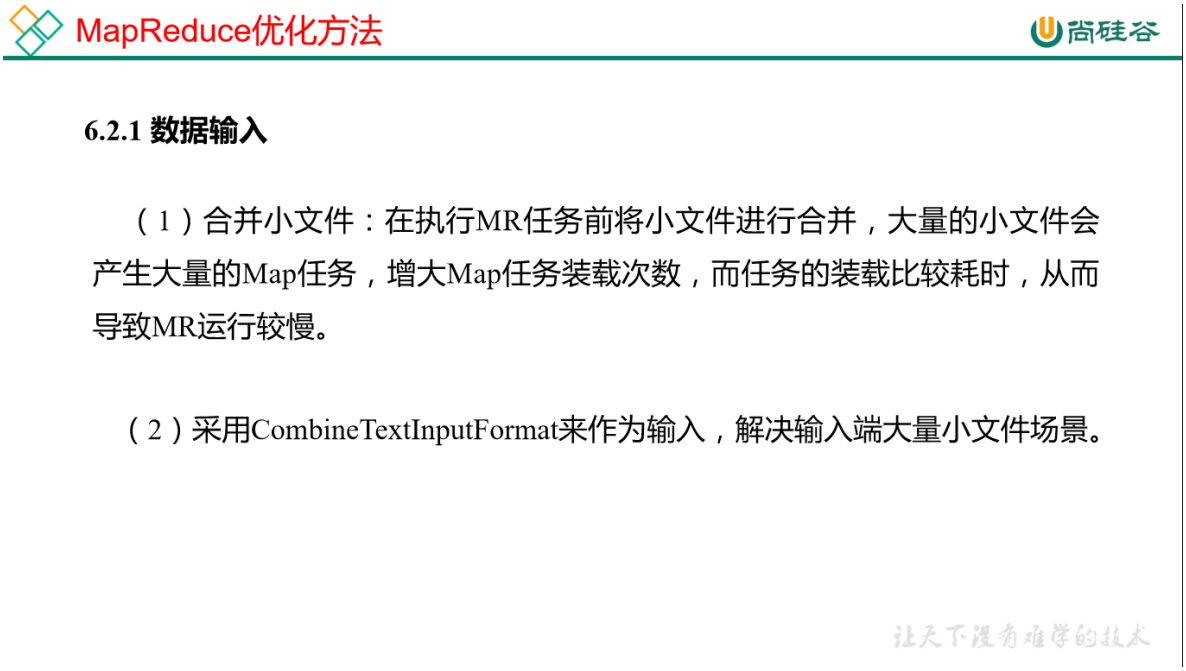
  CPU、内存、磁盘健康、网络（用钱就能解决，购买设备）

- I/O操作
  - 数据倾斜
  - Map和Reduce数设置不合理
  - Map运行时间太长，导致Reduce等待过久
  - 小文件过多
  - 大量的不可分块的超大文件
  - Spill次数过多
  - Merge次数过多（涉及到文件的分区排序）

优化方法

MapReduce优化方法主要从六个方面考虑：数据输入、Map阶段、Reduce阶段、IO传输、数据倾斜问题和常用的调优参数。

数据输入



Map阶段

6.2.2 Map阶段

（1）减少溢写（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）减少合并（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，**不影响业务逻辑前提下，先进行Combine处理**，减少 I/O。

Reduce阶段

6.2.3 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。

6.2.3 Reduce阶段

（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，中间多次写磁盘->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：mapreduce.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

6.2.4 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LZO压缩编码器。

2）使用SequenceFile二进制文件。

数据倾斜

2．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

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。且其中某几个专业词汇较多。那么就可以自定义分区将这这些专业词汇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倾斜。在可能的情况下，Combine的目的就是聚合并精简数据。

方法4：采用Map Join，尽量避免Reduce Join。

常用的调优参数

资源相关参数

- 以下参数是在用户自己的MR应用程序中配置就可以生效（mapred-default.xml）

配置参数	参数说明
mapreduce.map.memory.mb	一个MapTask可使用的资源上限（单位:MB），默认为1024。如果MapTask实际使用的资源量超过该值，则会被强制杀死。
mapreduce.reduce.memory.mb	一个ReduceTask可使用的资源上限（单位:MB），默认为1024。如果ReduceTask实际使用的资源量超过该值，则会被强制杀死。
mapreduce.map.cpu.vcores	每个MapTask可使用的最多cpu core数目，默认值: 1
mapreduce.reduce.cpu.vcores	每个ReduceTask可使用的最多cpu core数目，默认值: 1
mapreduce.reduce.shuffle.parallelcopies	每个Reduce去Map中取数据的并行数。默认值是5
mapreduce.reduce.shuffle.merge.percent	Buffer中的数据达到多少比例开始写入磁盘。默认值0.66
mapreduce.reduce.shuffle.input.buffer.percent	Buffer大小占Reduce可用内存的比例。默认值0.7
mapreduce.reduce.input.buffer.percent	指定多少比例的内存用来存放Buffer中的数据，默认值是0.0

- 应该在YARN启动之前就配置在服务器的配置文件中才能生效（yarn-default.xml）

配置参数	参数说明

配置参数	参数说明
yarn.scheduler.minimum-allocation-mb	给应用程序Container分配的最小内存，默认值：1024
yarn.scheduler.maximum-allocation-mb	给应用程序Container分配的最大内存，默认值：8192
yarn.scheduler.minimum-allocation-vcores	每个Container申请的最小CPU核数，默认值：1
yarn.scheduler.maximum-allocation-vcores	每个Container申请的最大CPU核数，默认值：32
yarn.nodemanager.resource.memory-mb	给Containers分配的最大物理内存，默认值：8192

- Shuffle性能优化的关键参数，应在YARN启动之前就配置好（mapred-default.xml）

配置参数	参数说明
mapreduce.task.io.sort.mb	Shuffle的环形缓冲区大小，默认100m
mapreduce.map.sort.spill.percent	环形缓冲区溢出的阈值，默认80%

容错相关参数(MapReduce性能优化)

配置参数	参数说明
mapreduce.map.maxattempts	每个Map Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。
mapreduce.reduce.maxattempts	每个Reduce Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。
mapreduce.task.timeout	Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个Task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该Task处于Block状态，可能是卡住了，也许永远会卡住，为了防止因为用户程序永远Block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是600000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示是“AttemptID:attempt_14267829456721_123456_m_000224_0 Timed out after 300 secsContainer killed by the ApplicationMaster.”。

HDFS小文件优化方法

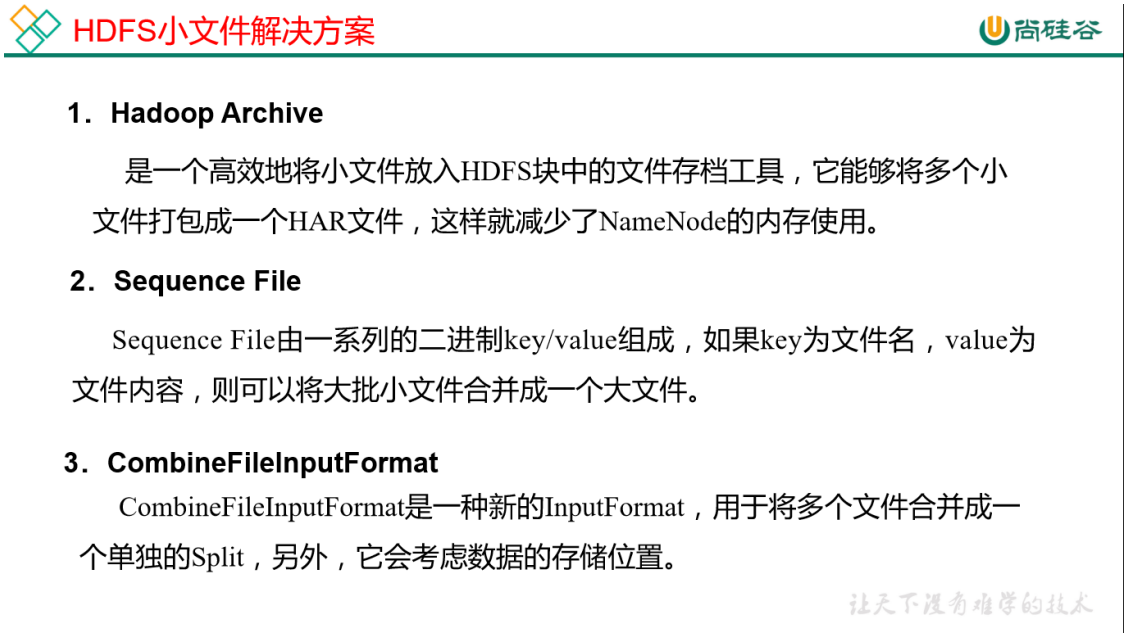
HDFS小文件弊端

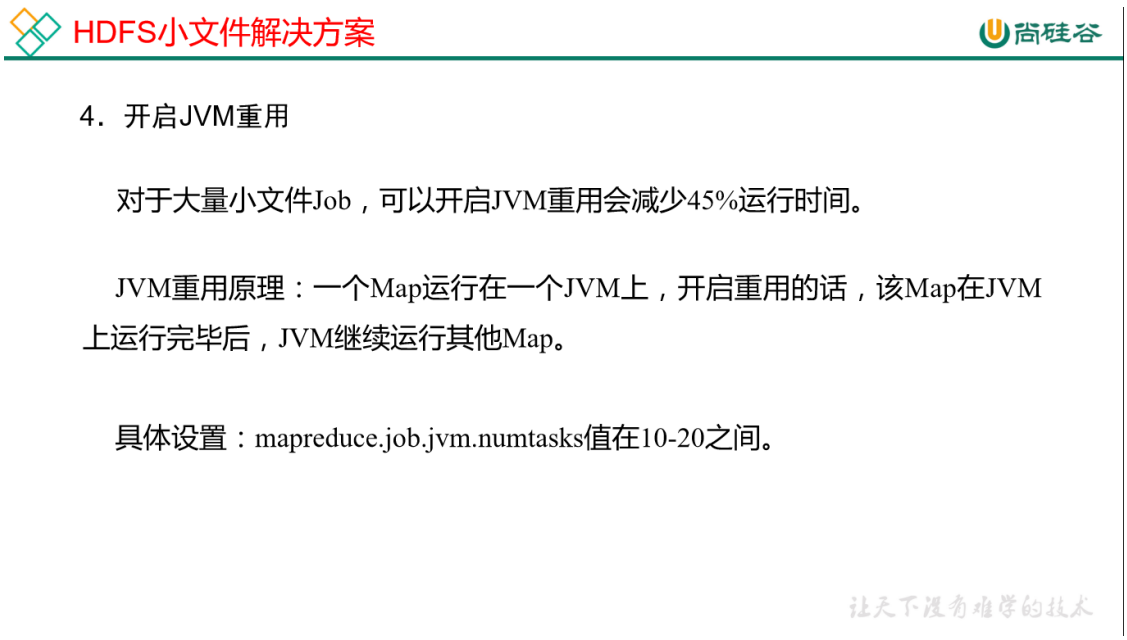
HDFS上每个文件都要在NameNode上建立一个索引，这个索引的大小约为150byte，这样当小文件比较多的时候，就会产生很多的索引文件，一方面会大量占用NameNode的内存空间，另一方面就是索引文件过大使得索引速度变慢。

解决方案

- 在数据采集的时候，就将小文件或小批数据合成大文件再上传HDFS。

- 在业务处理之前，在HDFS上使用MapReduce程序对小文件进行合并。
- 在MapReduce处理时，可采用CombineTextInputFormat提高效率。





> JVM重用技术不是指同一Job的两个或两个以上的task可以同时运行于同一JVM上，而是排队按顺序执行。s
>
> 如果task属于不同的job，那么JVM重用机制无效，不同job的task需要不同的JVM来运行。
>
> 重用缺点：开启JVM重用将会一直占用使用到的task插槽，以便进行重用，直到任务完成后才能释放。如果某个“不平衡”的job中有几个reduce task 执行的时间要比其他reduce task消耗的时间多得多的话，那么保留的插槽就会一直空闲着却无法被其他的job使用，直到所有的task都结束了才会释放。
>
> 插槽应该可以理解为一个task占用的内存空间吧。

坑

reduce()中的values Iterable只能遍历一次

```
protected void reduce(Text key, Iterable<TableBean> values, Context
context){
    for (TableBean value : values) {
        if ("order".equals(value.getFlag())) {
            TableBean temp = new TableBean();
            try {
                // 需要深拷贝，否则直接添加value的话，value会一直都是第一个数据
                BeanUtils.copyProperties(temp, value);
            } catch (IllegalAccessException | InvocationTargetException e)
{
                e.printStackTrace();
            }
            orderBeans.add(temp);
        } else {
            try {
                BeanUtils.copyProperties(pdBean, value);
            } catch (IllegalAccessException | InvocationTargetException e)
{
                e.printStackTrace();
            }
        }
    }
}
```

虽然reduce方法会反复执行多次，但key和value相关的对象只有两个，reduce会反复重用这两个对象。所以如果要保存key或者value的结果，只能将其中的值取出另存或者重新clone一个对象（例如 Text store = new Text(value) 或者 String a = value.toString() ），而不能直接赋引用。因为引用从始至终都是指向同一个对象，你如果直接保存它们，那最后它们都指向最后一个输入记录。会影响最终计算结果而出错。

即每次迭代对应的值在此次reduce时内存中是一个实例，每次执行时都是对value进行赋值

HDFS块 切片

假设块大小为128MB，现有文件129MB

- 块大小是严格切分的，切分为128MB，1MB
- 切片是逻辑上的，切分时大于1.1倍才切分，所以129MB不切分