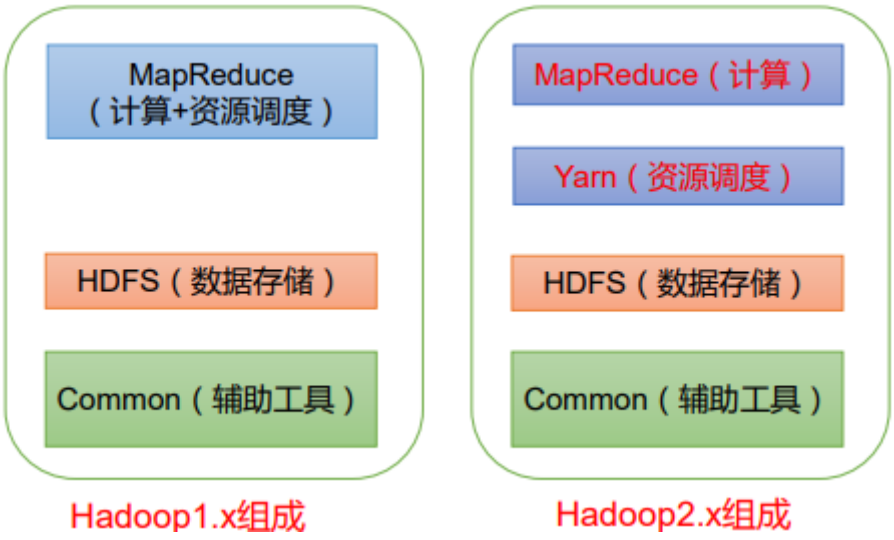
# Hadoop基础----------

分布式系统基础架构，主要是为了解决**海量数据的存储**和**海量数据的分析计算问题**

## Hadoop的特点

- 高可靠性：Hadoop底层维护多个数据副本，即使Hadoop某个计算元素或存储出现故障，也不会导致数据的丢失
- 高扩展性：集群分配任务数据，可方便扩展数以千计的节点
- 高效性：并行工作，加快任务处理速度
- 高容错性：能够自动将失败的任务重新分配

## Hadoop的1.x,2.x,3.x




- 1.x：HDFS、MapReduce、Common
- 2.x和3.x：HDFS、MapReduce、Yarn、Common

1.x时的MapReduce同时处理业务逻辑运算和资源的调度，耦合性比较大；

2.x增加了Yarn，负责资源的调度

## Hadoop的运行模式

- 单机：一台机器上运行服务，几乎不用做任何配置，仅限于调试用途。没有分布式文件系统，直接读写本地操作系统的文件系统
- 伪分布式：一台机器上启动集群相关的节点，模拟分布式运行的各个节点
- 完全分布式：正常的Hadoop集群

## Hadoop的序列化

- 序列化：将内存中的对象——>字节序列（或其他数据传输协议），以便于存储到磁盘（持久化）和网络传输
- 反序列化：将收到的字节序列（或其他数据传输协议）->内存中的对象

- 为什么要序列化：序列化可以存储“活的”对象，将“活的”对象发送到远程计算机
- 为什么不用Java的序列化：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（如：校验信息，Header，继承体系），不便于在网络中高效传输。Hadoop因此开发了一套自己的序列化机制（Writable）
- Hadoop序列化的特点：
    - （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）互操作：支持多语言的交互

# Hadoop的压缩

| 压缩格式 | 算法 | 文件扩展名 | 是否可切分 |
| --- | --- | --- | --- |
| DEFLATE | DEFLATE | .deflate | 否 |
| Gzip | DEFLATE | .gz | 否 |
| bzip2 | bzip2 | .bz2 | 是 |
| LZO | LZO | .lzo | 是 |
| Snappy | Snappy | .snappy | 否 |

为了支持多种压缩/解压算法，Hadoop引入了编码/解码器，如下所示：

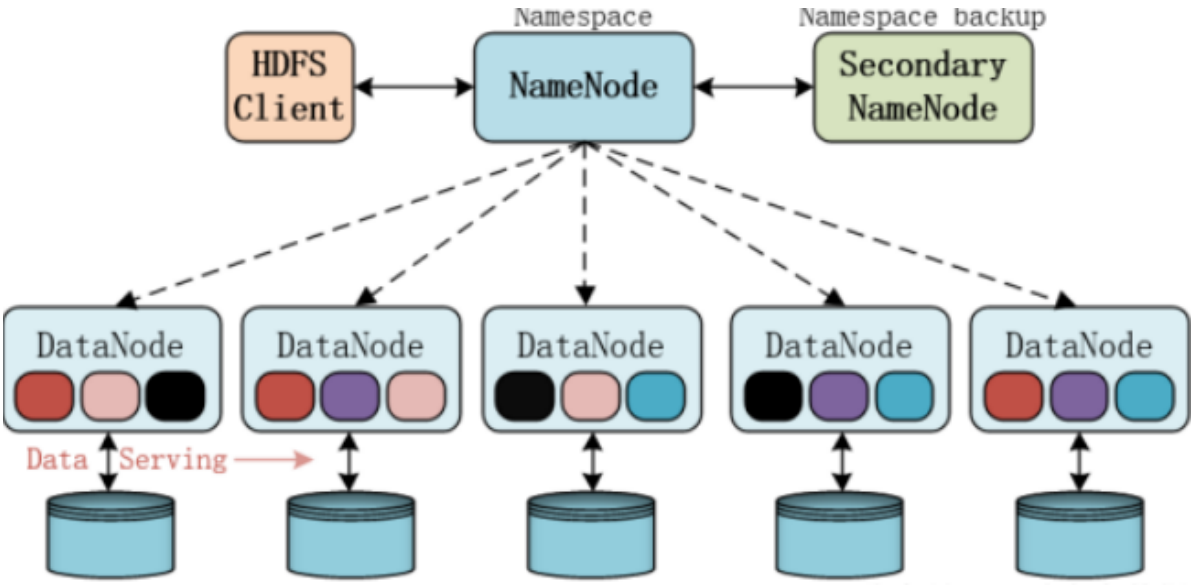
| **压缩格式** | **对应的编码/解码器** |
| --- | --- |
| DEFLATE | org.apache.hadoop.io.compress.DefaultCodec |
| gzip | org.apache.hadoop.io.compress.GzipCodec |
| bzip2 | org.apache.hadoop.io.compress.BZip2Codec |
| LZO | com.hadoop.compression.lzo.LzopCodec |
| Snappy | org.apache.hadoop.io.compress.SnappyCodec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

# Hadoop自带的小文件处理方案

- Hadoop Archive：一个高效地将小文件放入HDFS块中的文件存档工具，能够将多个小文件打包成一个HAR文件，允许对文件进行透明访问
- Sequence file：由一系列的二进制key/value组成。如果以key为小文件名，value为文件内容，则可以将大批小文件合并成一个大文件
- CombineFileInputFormat：一种新的inputformat，用于将多个文件合并成一个单独的split。

# HDFS区域----------

# HDFS组件（重点）




四个部分：HDFS Client、NameNode、DataNode、Secondary NameNode

1）HDFS Client

```
1、文件切分。文件上传HDFS的时候，Client将文件切分成一个个block，进行存储
2、于NN交互，获取文件位置信息
3、与DN交互，读/写数据
4、Client提供一些命令来管理HDFS
```

2）NameNode（NN）：Master

```
1、管理HDFS的命名空间
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（DN）：Slave

```
1、存储实际的数据块
2、执行数据库的读/写操作
```

4）Secondary NameNode（NN2）

```
1、辅助NN，分担其工作量
2、定期合并Fsimage和Edites，并推送给NN
3、在紧急情况下，辅助恢复NN
```

# HDFS的Block

**Block的概念**

首先，磁盘有一个Block size的概念，它是磁盘读/写数据的最小单位。文件系统的块通常是磁盘块的整数倍。

HDFS也有Block的概念，像磁盘中的文件系统一样，HDFS中的文件按块大小进行分解，HDFS中一个块只存储一个文件的内容

HDFS块抽象后的好处：
1、一个文件的大小，可以大于网络中任意一个硬盘的大小
2、使用抽象块而非整个文件作为存储单元，简化系统设计
3、用块数据进行备份，提供数据容错能力、系统可用性

HDFS在2.7.2版本之前默认是64M，2.7.3版本及之后是128M

```
<property>
    <name>dfs.blocksize</name>
    <value>134217728</value>
</property>
```

**Block块大小的设置**

寻址时间：HDFS中找到目标文件Block块所花费的时间

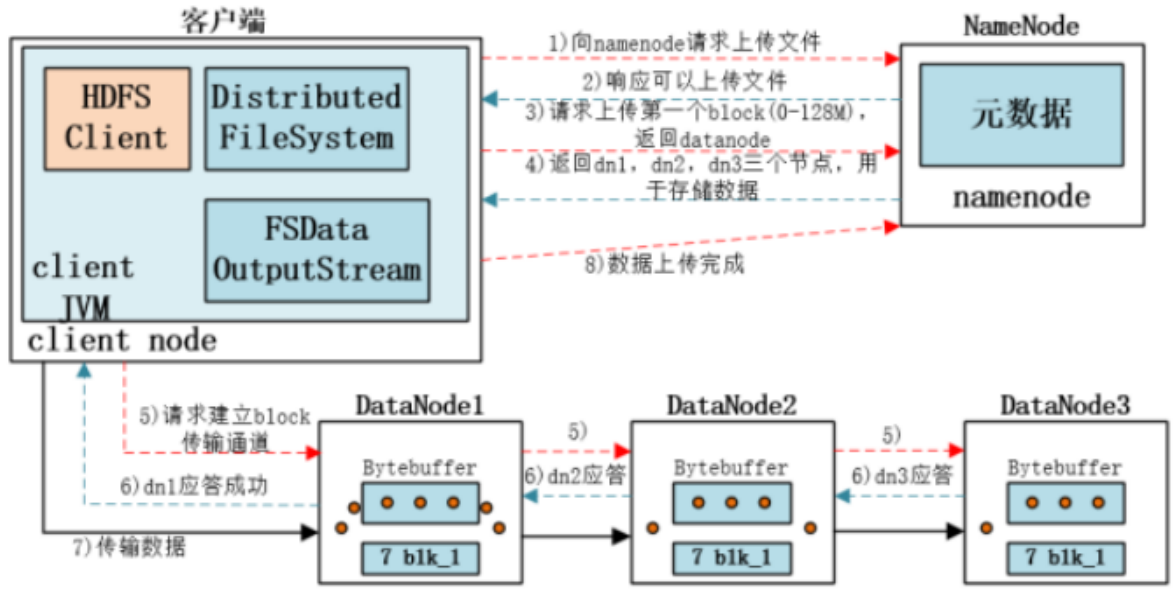
- 文件块越大，寻址时间越短，磁盘传输时间越长
- 文件块越小，寻址时间越长，磁盘传输数据越短

Block块合理的设置：
块设置过大：磁盘传输数据明显大于寻址时间，程序处理数据非常慢
MapReduce中的map任务通常一次只处理一个块中的数据，块过大运行速度慢
块设置过小：寻址时间增长，程序一直在找Block块的开始位置
大量小文件会占有NN中的内存

合适：寻址时间占传输数据的1%

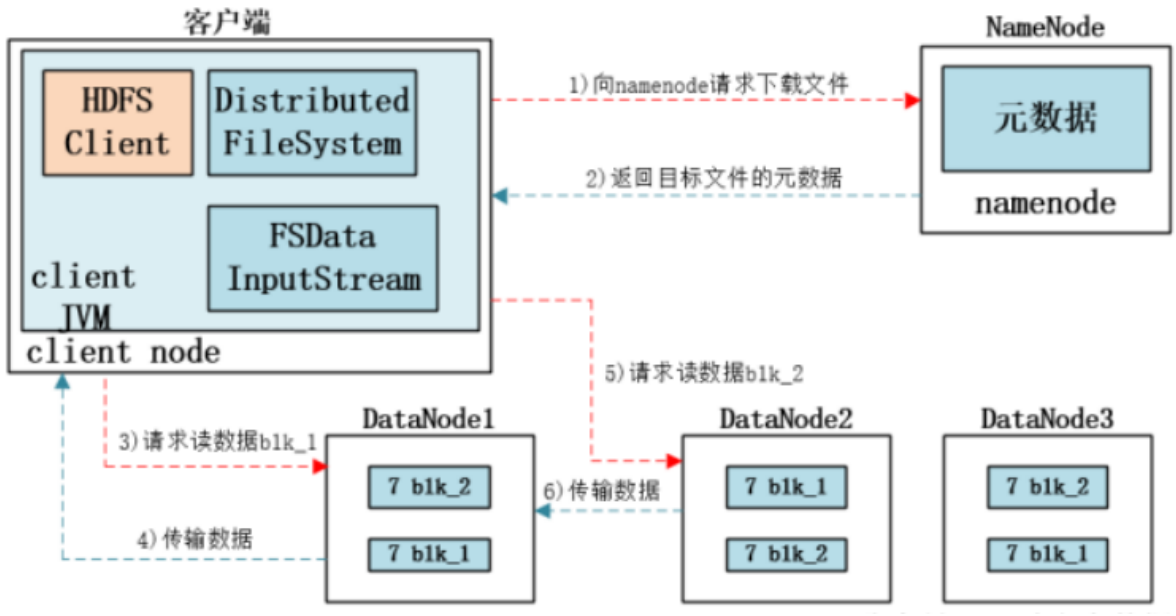
# HDFS写流程（重点）

HDFS写数据过程



1、Client通过Distributed FileSystem模块向NN请求上传文件，NN检查目标文件是否已存在，父目录是否存在
2、NN返回是否可以上传
3、Client请求第一个block上传哪几个DN服务器上
4、NN返回3个DN节点，分别为DN1、DN2、DN3
5、Client通过FSDataOutputStream模块请求DN1建立传输通道，DN1收到请求会继续调用DN2,DN2继续调用DN3
6、DN3应答DN2，DN2应答DN1，DN1应答Client
7、Client开始往DN1上传第一个block(先从磁盘上读取数据放到一个本地磁盘缓存)，以packet为单位，DN1收到一个packet就会上传给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传输完成后，Client会再次请求NN上传第二个block的服务器，重复执行3-7

# HDFS读流程（重点）




1、Client通过Distributed FileSystem模块向NN请求下载文件，NN通过查询元数据，找到文件的DN地址
2、Client挑选一台最近的DN，请求读取数据
3、DN开始传输数据给Client(从磁盘里读取数据输入流，以packet为单位做校验)
4、Client以packet为单位接收，先在本地缓存，然后写入目标文件

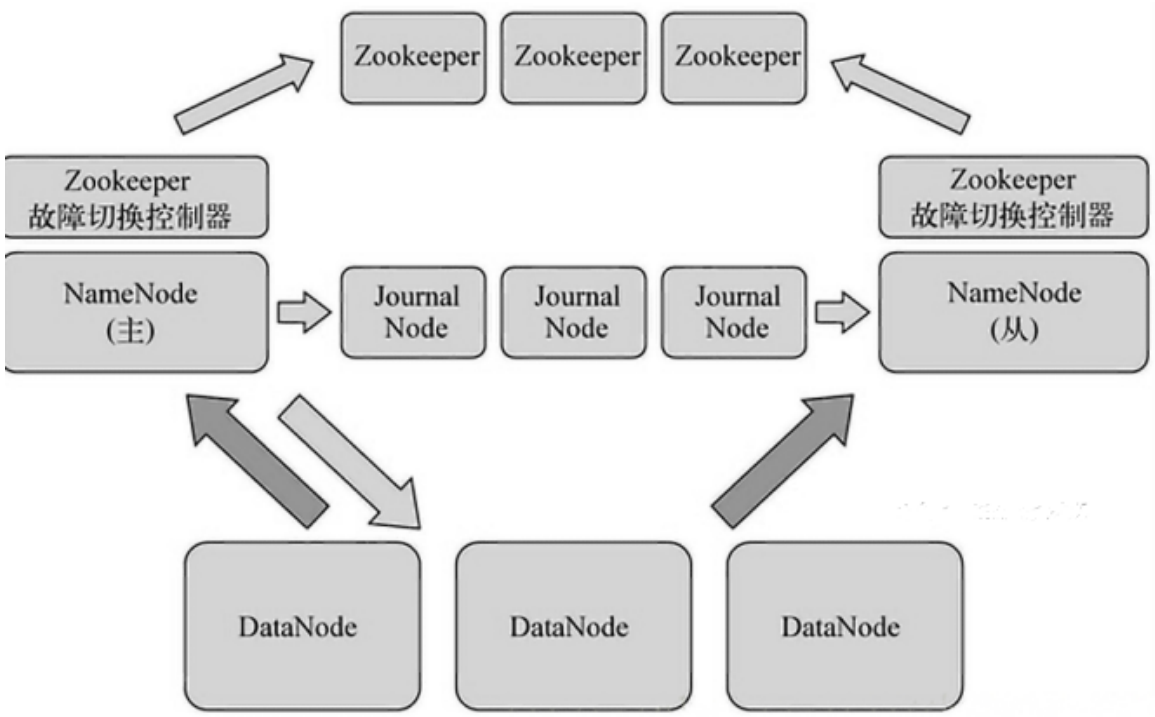
# DN节点数据完整性

1、当DN读取Block时，计算CheckSum
2、当计算后的CheckSum和创建Block时值不一样，说明Block损坏，到其他DN上读取Block

DN在文件创建后周期性验证CheckSum
常见的校验算法：crc(32)，md5(128)，sha1(160)

# HDFS HA实现

集群同时运行两个NN，一个作为活动的NN（Active），一个作为备份的NN（Standby）。备份的NN的命名空间与活动的NN是实时同步的，当活动的NN发生故障而停止服务时，备份NN可以立即切换为活动状态，而不影响HDFS集群服务。




活动的NN负责执行所有修改命名空间以及删除备份数据库的操作，而备份的NN则执行同步操作，以保持与活动节点命名空间的一致性。

为了使备份节点与活动节点的状态能够同步一致，两个节点需要同一组独立运行的节点（JournalNodes，JNS）通信。当Active NN执行了修改命名空间的操作时，会定期将执行的操作记录在editlog中，并写入JNS的多数节点中。而Standby NN会一直监听JNS上editlog的变化（变化后，读取最新的editlog并与当前的命名空间合并）

```
JN（Journal Node）：运行在N台独立的物理机器上，将editlog文件保存在本地磁盘上，对外提供RPC接口QJournalProtocol以执行远程读写editlog文件的功能
QJM（Quorum Journal Manager）：运行在NN上，通过调用RPC接口QJournalProtocol中的方法向JN发送、同步editlog
```

DN同时向Active和Standby NN两个节点发送心跳及块汇报信息，一旦Active NN发送故障，Standby NN就可以马上切换，这就是热备。

当HA架构中出现两个NN同时修改命名空间（脑裂现象），会造成HDFS集群数据库丢失，预防脑裂的三个隔离机制

- 共享存储隔离：同一时间只允许一个NN向JNS写入editlog
- 客户端隔离：同一时间只允许一个NN响应Client的请求
- DN隔离：同一时间只允许一个NN向DN下发各节点指令

# HDFS数据的一致性

- **NN机制**：NN在工作时将元数据缓存在内存中，同时备份到磁盘fsimage中。对元数据信息修改的操作会追加到editlog文件中。NN2定期或者etitlog达到一定数量后复制合并fsimage、editlog为一个新的fsimage，在推送给NN。
- **心跳机制**：DN定期发送元数据信息给NN，默认时3秒一次
- **安全模型**：HDFS初始化阶段会进入安全(safe)模式，此时NN不允许操作。NN同DN进行安全检查，当安全的数据块比值达到阈值才会推出安全模式。
- **回滚机制**：在hdfs升级或者数据写入时，相关的数据会被保留备份。成功则更新备份，失败则使用备份
- **安全校验**：避免网络传输造成的数据错误问题，HDFS采用了校验和机制。各个NN之间数据备份和读取需要通过校验，校验不通过则重新备份

- **回收站**：当数据文件从hdfs删除时，文件转存于/trash目录在。在超过规定的时间fs.trash.interval，NN和DN会将该文件的元数据删除

# HDFS的列式存储、行式存储

- 行式存储：一条数据保存为一行，读取一行中的任何值都需要把整行数据都读取处理，如：Sequence Files，Map File，Avro Data Files，磁盘读取开销比较大
- 列式存储：整个文件被切割为若干列数据，每一列中数据保存在一起，如：Parquet，RC Files，ORC Files，Carbon Data，IndexR，会占有更多的内存空间，需要将行数据缓存起来

```
写入：
    行存储一次完成，效率高，保证数据的完整性
    列存储把一行记录拆分成单列保存，写入次数多。
读取：
    行存储将一行数据完全读出，如果只需要其中几列数据，存在冗余列
    列存储每次读取一段或者全部，存储的数据是同质的，使得数据解析容易。
```

# MapReduce区域----------

MapReduce是一个**分布式运算程序的编程框架**，核心功能实将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在一个Hadoop集群上

# MapReduce优缺点

优点：

- 1、易于编程，用户只关心业务逻辑；实现框架的接口
- 2、良好的扩展性，可以动态增加服务器，解决计算资源不够问题
- 3、高容错性，任何一台机器挂掉，可以将任务转移到其他节点
- 4、适合海量数据计算（TP/PB），几千台服务器共同计算

缺点：

- 1、不擅长实时计算，Mysql
- 2、不擅长流式计算，Sparkstreaming flink
- 3、不擅长DAG有向无环图计算，spark

# MapReduce进程

一个完整的MapReduce在分布式运行时有三类实例进程：

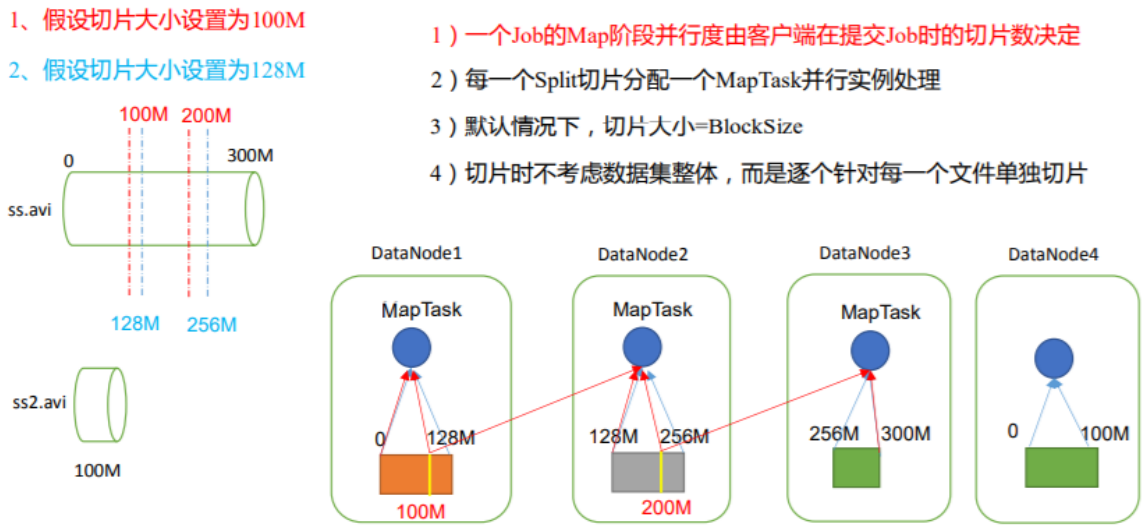
- MrAppMaster：负责整个程序的过程调度及状态协调
- Map Task：负责Map阶段的整个数据处理流程
- ReduceTask：负责Reduce阶段的整个数据处理流程

# InputFormat数据输入

## 1、切片与MapTask并行度机制

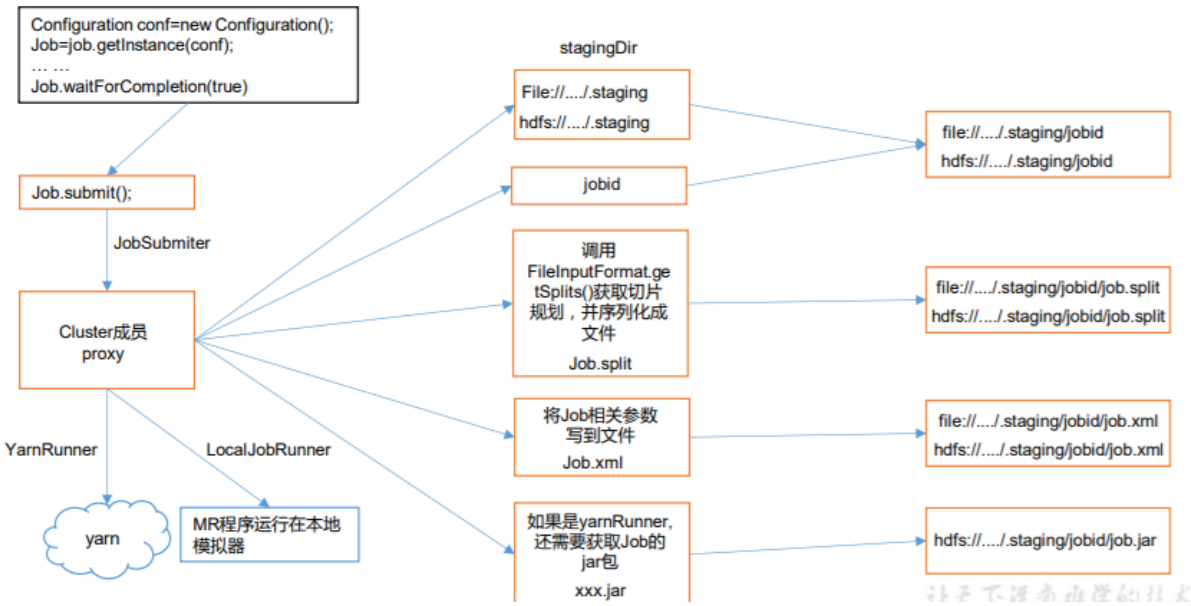
- 问题：MapTask的并行度决定Map阶段的任务处理并发度，进而影响到整个Job的处理速度
- MapTask并行度决定机制
    - 数据块：Block是HDFS物理上把数据分成一块一块，**数据块是HDFS存储数据单位**

- 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。**数据切片是MapReduce程序计算输入数据的单位**，一个切片会对应启动一个MapTask




## 2、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）判断是本地运行环境还是yarn集群运行环境
            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());
```




## 3、FileInputFormat切片机制

切片机制：

- 1.简单地按照文件的内容长度进行切片
- 2.切片大小，默认等于Block大小
- 3.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

切片源码解析：

```
(1)源码中计算切片大小的公式
Math.max(minSize,Math.min(maxSize,blockSize))
mapreduce.input.fileinputformat.split.minsize=1 //默认值为1
mapreduce.input.fileinputformat.split.maxsize=Long.MAXValue //默认值Long.MAXValue

(2)切片大小设置
maxsize(切片最大值)：参数如果调得比blockSize小，则会让切片变小，此时切片大小=maxsize
minsize(切片最小值)：参数调的比blockSize大，让切片变大,此时切片大小=minSize

(3)获取切片信息API
String name = inputSplit.getPath().getName();   //获取切片的文件名称
FileSplit inputSplit = (FileSplit) contxt.getInputSplit();  //根据文件类型获取切片信
息
```

## 4、FileInputFormat

FileInputFormat常见的接口实现类：

```
TextInputFormat、KeyValueTextInputFormat、NLineInputFormat、CombineTextInputFormat
和自定义InputFormat等
```

- TextInputFormat：FileInputFormat默认实现类，按行读取每条记录。
  - 键：存储该行在整个文件中的起始字节偏移量，LongWritable类型
  - 值：改行内容，不包括任何行终止符（换行符和回车符），Text类型

## 5、CombineTextInputFormat切片机制

说明：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，处理效率及其低下。

(1)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，多个小文件交给一个MapTask处理

(2)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

(3)切片机制，生成切片过程包括：虚拟存储过程和切片过程两部分

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）。

例如setMaxInputSplitSize值为4M，输入文件大小为8.02M，则先逻辑上分成一个4M。剩余的大小为4.02M，如果按照4M逻辑划分，就会出现0.02M的小的虚拟存储文件，所以将剩余的4.02M文件切分成（2.01M和2.01M）两个文件

切片过程：

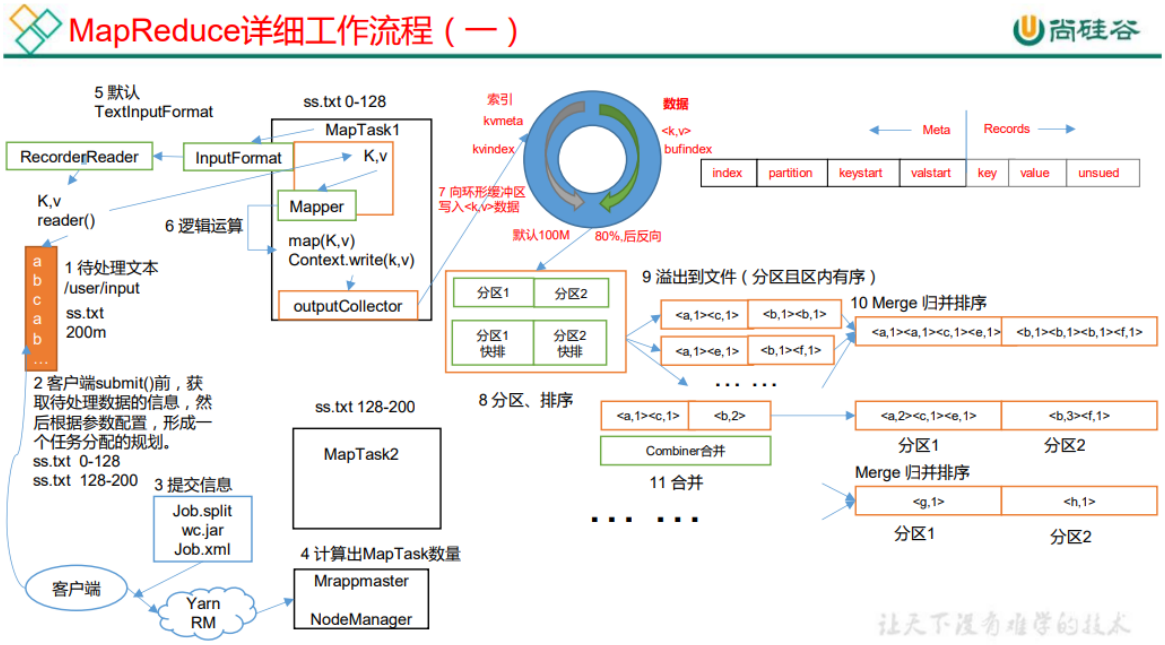
（a）判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。

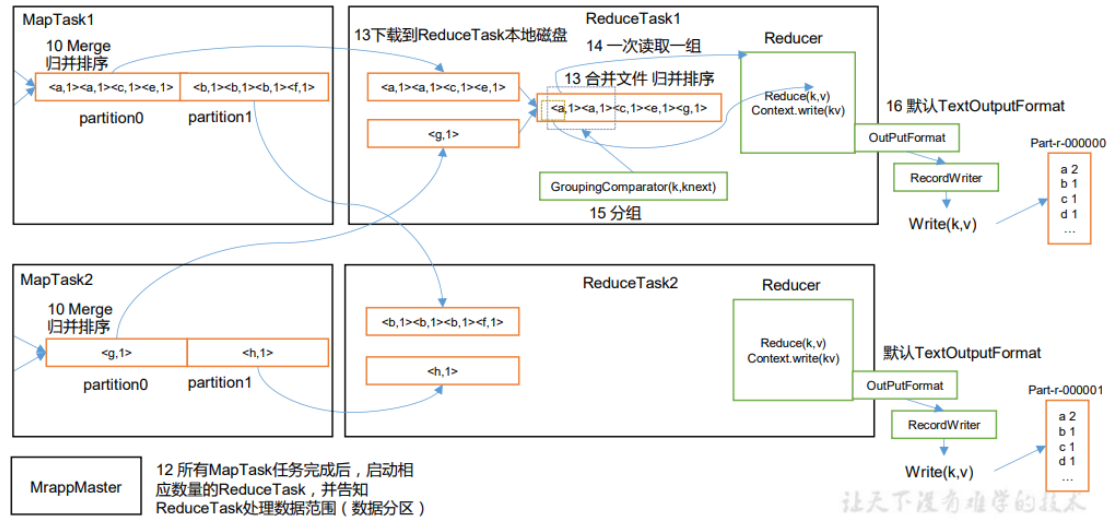
（b）如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。

（c）测试举例：有4个小文件大小分别为1.7M、5.1M、3.4M以及6.8M这四个小文件，则虚拟存储之后形成6个文件块，大小分别为：1.7M，（2.55M、2.55M），3.4M以及（3.4M、3.4M）

最终会形成3个切片，大小分别为：（1.7+2.55）M，（2.55+3.4）M，（3.4+3.4）M

# MapReduce工作流程（重点）

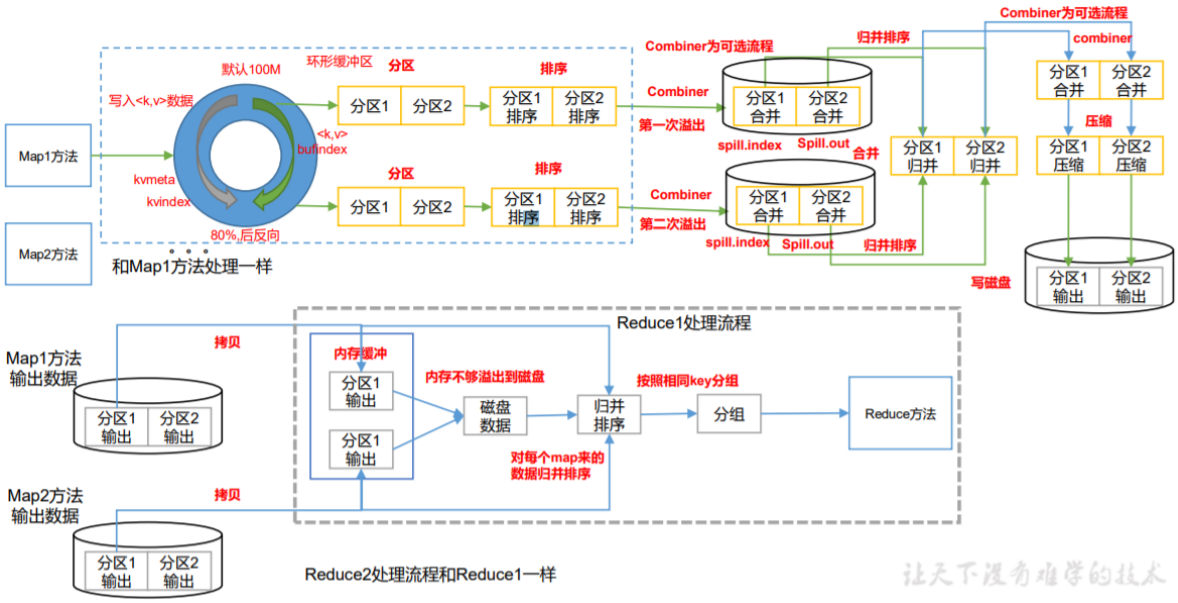





# Shffle机制（重点）

## 1、Shffle机制大纲流程

**Map方法之后，Reduce方法之前的数据处理过程称之为Shuffle**




(1)MapTask收集我们的map方法输出kv对，放到内存缓冲区中
(2)从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件
(3)多个溢出文件会被合并成大的溢出文件
(4)在溢出过程及合并的过程中，都要调用Partition进行分区和针对key进行排序
(5)ReduceTask根据自己的分区号，去各个MapTask机器上取相应的结果分区数据
(6)ReduceTask会抓取到同一个分区的来自不同MapTask的结果文件，ReduceTask会将这些文件再进行合并(归并排序)
(7)合并成大文件后。Shuffle的过程也就结束了，后面进入ReduceTask的逻辑运算过程(从文件中取出一个一个的键值对Group，调用用户自定义的reduce()方法)

注意：(1)Shuffle中的缓冲区大小会影响到MapReduce程序的执行效率，原则上说，缓冲区越大，磁盘io的次数越少，执行速度就越快。
(2)缓冲区的大小可以通过参数调整，参数：mapreduce.task.io.sort.mb默认100M

## 2、Partition分区

1.问题引出：
要求将统计结果按照条件输出到不同文件中(分区)

2.默认Partitioner分区
```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
        }
}
```
默认分区是根据key的hashCode对ReduceTask个数取模得到的，用户没法控制哪个key存储到哪个分区

3.自定义Partition步骤
(1)自定义类继承Partitioner，重写getPartition方法
```
public class CustomPartitioner extends Partitioner<Text, FlowBean> {
    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {
    // 控制分区代码逻辑
    … …
        return partition;
    }
}
```
(2)在Job驱动中，设置自定义Partitioner
```
job.setPartitionerClass(CustomPartitioner.class)
```
(3)自定义Partition后，要根据自定义Partitioner的逻辑设置相应数量的ReduceTask
```
job.setNumReduceTasks(5); //输出到五个文件,分区设置为5
```

4.分区总结
(1)如果ReduceTask的数量 > getPartition的结果数，则会多产出几个空的输出文件part-r-000xx;
(2)如果1 < ReduceTask的数量 < getPartition的结果数，则有一部分分区数据无处安放，会Exception;
(3)如果ReduceTask的数量=1，则不管MapTask端出数多少个分区文件，最终结果都交给这一个ReduceTask，最终也就只会产出一个结果文件part-r-00000;
(4)分区号必须从零开始，逐一累加

5.案例分析
例如：假设自定义分区数为5，则
(1)job.setNumReduceTasks(1);　　会正常运行，只不过会产生一个输出文件
(2)job.setNumReduceTasks(2);　　会报错
(3)job.setNumReduceTasks(6);　　大于5，程序会正常运行，会产生空文件

## 3、WritableComparable排序

排序是MapReduce框架中最重要的操作之一

MapTask和ReduceTask均会对数据按照key进行排序。该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。(默认排序是按照字典顺序排序，且实现该排序的方法是快速排序)

对于MapTask，它会将处理的结果暂时存放到环形缓冲区中，当环形缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次快速排序，并将这些有序数据溢写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行归并排序

对于ReduceTask，它每个MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则溢写到磁盘上，否则存储在内存中。如果磁盘上文件数目达到一定阈值，则进行一次归并排序生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据溢写到磁盘上。当所有数据拷贝完后，ReduceTask统一对内存和磁盘上的所有数据进行一次归并排序

排序分类：

(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有序

(2)全排序：最终输出结果只有一个文件，且文件内部有序。实现方式是只设置一个ReduceTask，但该方法在处理大型文件时效率极低，因为一台机器处理所有文件，完全丧失了MapReduce所提供的并行架构。

(3)辅助排序(GroupingComparator分组)：在Reduce端对key进行分组，在接收的key为bean对象时，想让一个或几个字段相同的key进入到同一个reduce方法时，可以采用分组排序

(4)二次排序：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

## 4、Combiner合并

1.Combiner是MR程序中Mapper和Reducer之外的一种组件
2.Combiner组件的父类是Reducer
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是在接收全局所有Mapper的输出结果
4.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
5.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与Reducer的输入kv类型要对应
使用场景(求和√，求均值×)
求均值不行的例子：

| Mapper | Reducer |
| --- | --- |
| 3 5 7 ->5 | 3 5 7 2 6 -> 23/5 正确的 |
| 2 6 ->4 | 4 5 -> 9/2 使用场景错误导致不同的输出 |

6.自定义Combiner实现步骤
(a)自定义一个Combiner继承Reducer，重写Reduce方法

```
public class WordCountCombiner extends Reducer<Text, IntWritable,
Text,IntWritable> {
        private IntWritable outV = new IntWritable();
        @Override
        protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable value : values) {
                sum += value.get();
            }

             outV.set(sum);
            context.write(key,outV);
        }
    }
```

(b)在Job驱动类中设置

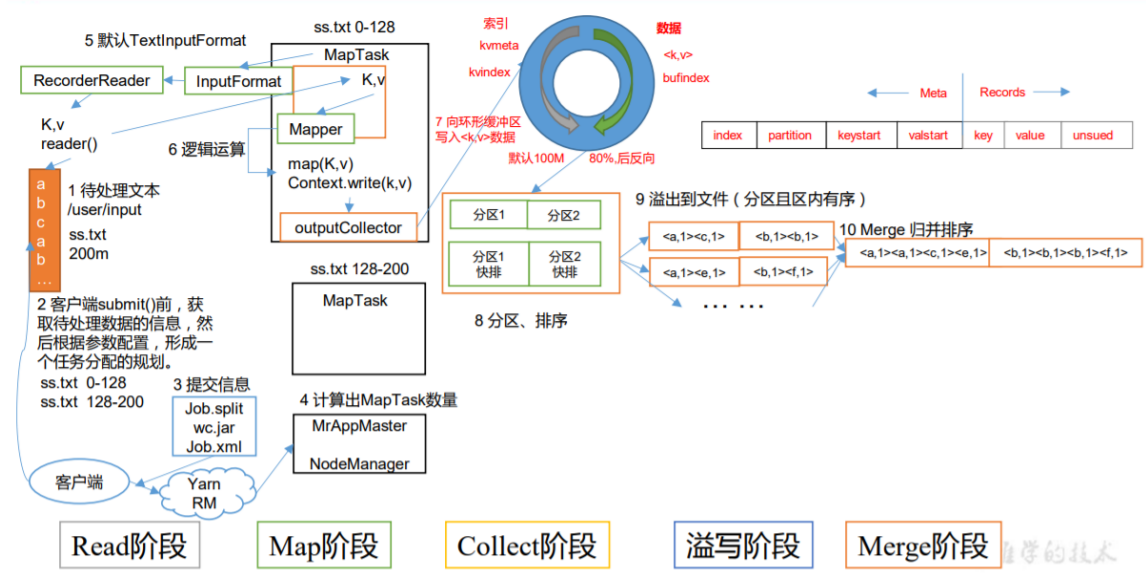
```
 job.setCombinerClass(WordCountCombiner.class);
```

# OutputFormat数据输出

OutputFormat是MapReduce输出的基类，所有实现MapReduce输出都实现了OutputFormat接口，默认格式TextOutputFormat

```
FileOutputFormat(含
TextOutputFormat,SequenceFileOutputFormat,MapFileOutputFormat),FilterOutputForma
t,DBOutputFormat

自定义OutPutFormat
1.应用场景：输出数据到MySQL/HBase/Elasticsearch等存储框架中
2.自定义OutputFormat步骤
    a.自定义一个类继承FileOutputFormat
    b.改写RecordWriter,具体改写输出数据的方法write()
```

# MapTask工作机制（重点）

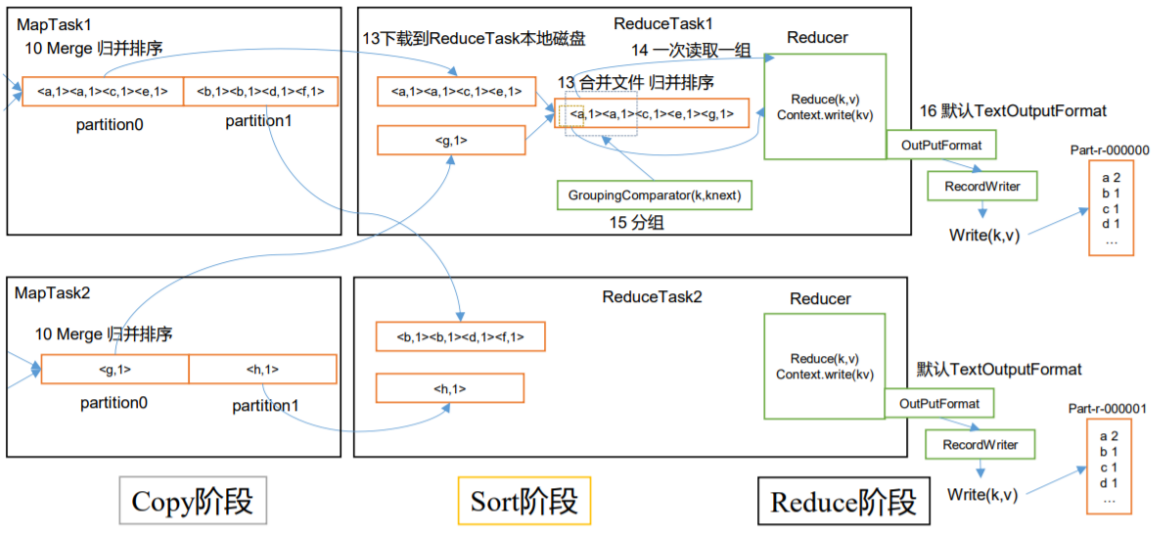


(1)Read阶段：MapTask通过InputFormat获得的RecordReader，从输入InputSplit中解析出一个个key/value
(2)Map阶段：该节点主要是将解析出的key/value交给用户编写map()函数处理，并产生一系列新的key/value
(3)Collect收集阶段：在用户编写map()函数中，当数据处理完成后，一般会调用OutputCollector.collect()输出结果。在该函数内部，它将会生成的key/value分区(调用Partition)，并写入一个环形内存缓冲区中
(4)Spill(溢写)阶段：当环形缓冲区满后，MapReducer会将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

步骤1：利用快速排序算法对缓冲区的数据进行排序，排序方式是，先按照分区编号Partition进行排序，然后按照key进行排序。这样，经过排序后，数据以分区为单位聚集在一起，且同一分区内所有数据按照key有序
步骤2：按照分区编号由小到大依次将每个分区中的数据写入任务工作目录下的临时文件output/spillN.out(N表示当前溢写次数)中。如果用户设置Combiner，则写入文件之前，对每个分区中数据进行一次聚集操作。
步骤3：将分区数据的元信息写到内存索引数据结构SpillRecord中，其中每个分区的元信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存索引大小1MB，则将内存索引写到文件output/spillN.out.index中

(5)Merge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。
当所有数据处理完后，MapTask会将所有临时文件合并成一个大文件，并保持到文件output/file.out中，同时生成相应的索引文件output/file.out.index
在进行文件合并过程中，MapTask以分区为单位进行合并，对于某个分区，它将采用多轮递归合并的方式。每轮合并mapreduce.task.io.facyor(默认10)个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。
让MapTask最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销

# ReduceTask机制（重点）




(1)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。
(2)Sort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。按照MapReduce语义，用户编写reduce()函数输入数据时按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
(3)Reduce阶段：reduce()函数将计算结果写到HDFS上

# MapTask & ReduceTask源码解析

```
=================== MapTask ===================
context.write(k, NullWritable.get()); //自定义的 map 方法的写出，进入
    output.write(key, value);
        //MapTask727 行，收集方法，进入两次
        collector.collect(key, value,partitioner.getPartition(key, value,
partitions));
            HashPartitioner(); //默认分区器
        collect() //MapTask1082 行 map 端所有的 kv 全部写出后会走下面的 close 方法
            close() //MapTask732 行
                collector.flush() // 溢出刷写方法，MapTask735 行，提前打个断点，进入
                    sortAndSpill() //溢写排序，MapTask1505 行，进入
                        sorter.sort() QuickSort //溢写排序方法，MapTask1625 行，进入
                    mergeParts(); //合并文件，MapTask1527 行，进入
                collector.close(); //MapTask739 行,收集器关闭,即将进入 ReduceTask
=================== ReduceTask ===================
if (isMapOrReduce()) //reduceTask324 行，提前打断点
initialize() // reduceTask333 行,进入
init(shuffleContext); // reduceTask375 行,走到这需要先给下面的打断点
    totalMaps = job.getNumMapTasks(); // ShuffleSchedulerImpl 第 120 行，提前打断点
        merger = createMergeManager(context); //合并方法，Shuffle 第 80 行
            // MergeManagerImpl 第 232 235 行，提前打断点
            this.inMemoryMerger = createInMemoryMerger(); //内存合并
            this.onDiskMerger = new OnDiskMerger(this); //磁盘合并
rIter = shuffleConsumerPlugin.run();
        eventFetcher.start(); //开始抓取数据，Shuffle 第 107 行，提前打断点
        eventFetcher.shutDown(); //抓取结束，Shuffle 第 141 行，提前打断点
        copyPhase.complete(); //copy 阶段完成，Shuffle 第 151 行
        taskStatus.setPhase(TaskStatus.Phase.SORT); //开始排序阶段，Shuffle 第 152
行
    sortPhase.complete(); //排序阶段完成，即将进入 reduce 阶段 reduceTask382 行
reduce(); //reduce 阶段调用的就是我们自定义的 reduce 方法，会被调用多次
    cleanup(context); //reduce 完成之前，会最后调用一次 Reducer 里面的 cleanup 方法
```

# MapReduce数据倾斜

1、数据倾斜现象

```
数据倾斜是数据的key的分化严重不均，造成一部分数据很多，一部分数据很少
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、数据倾斜产生的原因

```
（1）Hadoop框架的特性
        Job数多的作业运行效率会相对比较低
        count(distinct)、group by、join等操作，触发了Shuffle动作，导致全部相同key的值聚
集在一个或几个节点上。
（2）具体原因
        key分布不均匀，某一个key的条数远超于其他key的条数
        业务数据自带的特性
        建表时不考虑全面
        某些HQL语句自身存在数据倾斜问题
```

3、数据倾斜解决方案

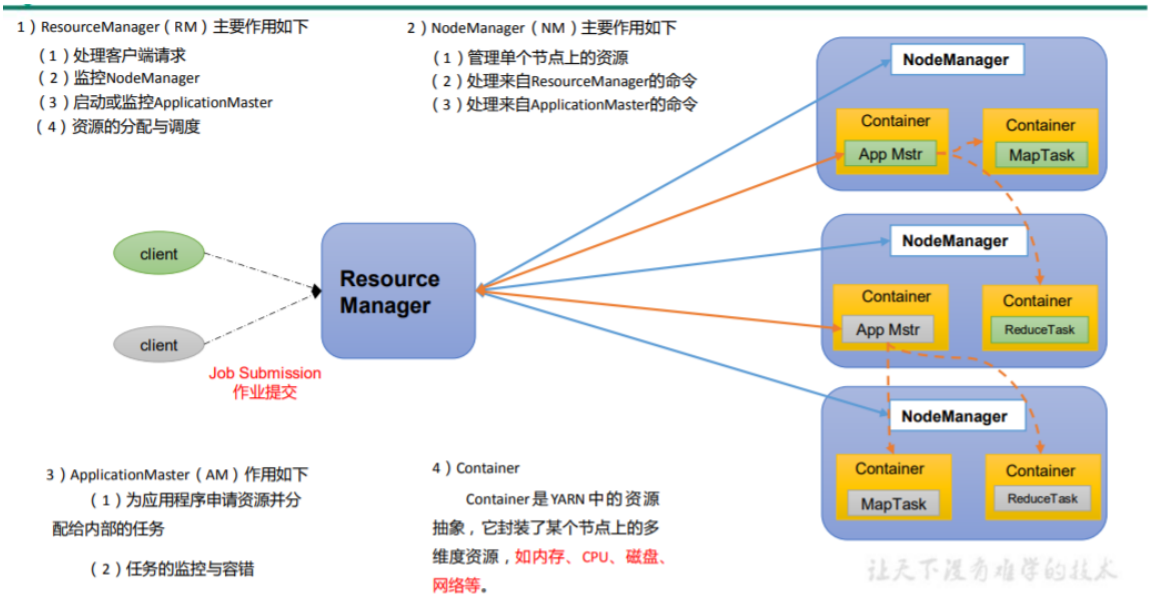
```
从业务和数据方面解决数据倾斜
    有损的方法：找到异常数据
    无损的方法：对分布不均匀的数据，进行单独计算，首先对key做一层hash，把数据打散，让它的并行度变大，之后进行汇集

Hadoop平台的解决办法
（1）join产生的数据倾斜
    场景一：大表和小表
        在多表关联情况下，将小表依次放在前面，触发reduce端减少操作次数
        使用Map Join让小表缓存到内存，在map端完成join过程，这样省掉reduce端的工作，需设值
set hive.auto.convert.join=true
            对优化的小表的大小进行设置：set
hive.mapjoin.smalltable.filesize=25000000(默认25M)
    场景二：大表和大表
        将异常值赋给一个随机值，以此来分数key，均匀分配给多个reduce执行
        key值都是有效情况下，设置参数：set
hive.exec.reducers.bytes.per.reducer=1000000000(1G)，如果join操作也产生了数据倾斜，设置：set hive.optimize.skewjoin=true;set
hive.skewjoin.key=skey_key_threshold(default=1000000)
（2）group by造成的数据倾斜
    hive.map.aggr=true(这个配置项代表在map端进行聚合，相当于Combiner)
    hive.groupby.skewindata
（3）count(distinct)或者其他参数不当造成的数据倾斜
    reduce个数太小：set mapred.reduce.tasks=800
    HQL中包含count(distinct)时：使用sum...group by来替代
```

# Yarn区域----------

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**，MapReduce等运算程序相当于运行在操作系统之上的应用程序

## Yarn组件（重点）

Yarn主要包含：ResourceManager（RM）、NodeManager（NM）、ApplicationMaster和Container模块




RM：

处理客户端请求

监控NM

启动或监控ApplicationMaster

资源的分配与调度

NM：

管理单个节点上的资源

处理来自RM的命令

处理来自ApplicationMaster的命令

ApplicationMaster：

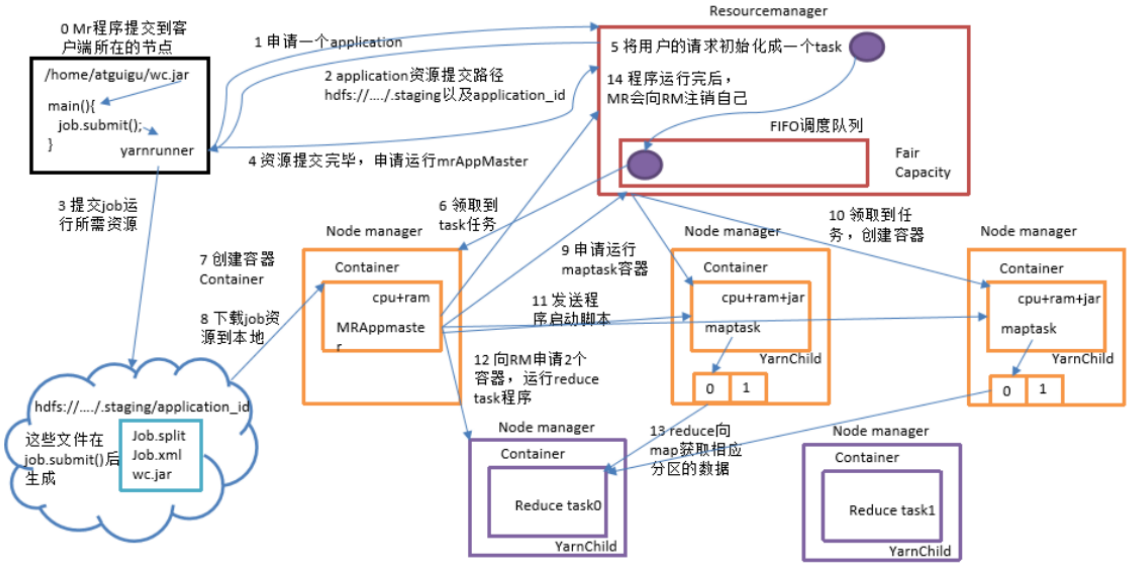
为应用程序申请资源并分配给内部的任务

任务的监督与容错

Containr：

是Yarn中的资源抽象，封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等

# Yarn工作机制（重点）




一）作业提交

1、Client调用job.waitForCompletion方法，向整个集群提交MapReduce作业

2、Client向RM申请一个作业id

3、RM给Client返回该job资源的提交路径和作业id

4、Client提交jar包、切片信息和配置文件到指定资源提交路径
5、Client提交完资源后，向RM申请运行MrAppMaster
二）作业初始化
6、当RM收到Client请求后，将该job添加到调度器中
7、某一个空闲的NM领取到该Job
8、该NM创建Container，并产生MrAppmaster
9、下载Client提交的资源到本地
三）任务分配
10、MrAppMaster向RM申请运行多个MapTask任务资源
11、RM将运行MapTask任务分配给另外两个NM,另外两个NM分别领取任务并创建容器
四）任务运行
12、MrAppmaster向两个接收到任务的NM发送程序启动脚本，这两个NM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，MrAppMaster向RM申请注销自己
五）进度和状态更新
YARN中的任务将其进度和状态返回给应用管理器，客户端每秒向应用管理器请求进度更新，展示给用户
六）作业完成
除了向应用管理器请求作业进度外，客户端每5秒都会通过调用waitForCompletion()来检查作业是否完成，时间间隔用mapreduce.client.completion.pollinterval来设置。作业完成之后，应用管理器和Container会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查

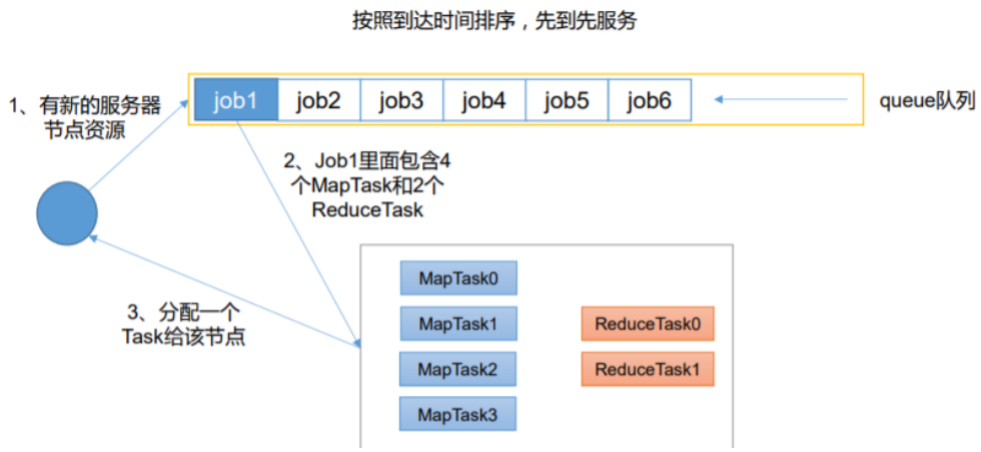
# Yarn调度器

目前Hadoop作业调度器主要有三种：FIFO、容量（Capacity Scheduler）和公平（Fair Scheduler）
Apache Hadoop3.13默认：Capacity Scheduler
CDH框架默认调度器是：Fair Scheduler

具体设置详见：yarn-default.xml文件

```
<name>yarn.resourcemanager.scheduler.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>
```

1、FIFO（先进先出调度器）




优点：简单易懂
缺点：不支持多队列，生产环境很少使用

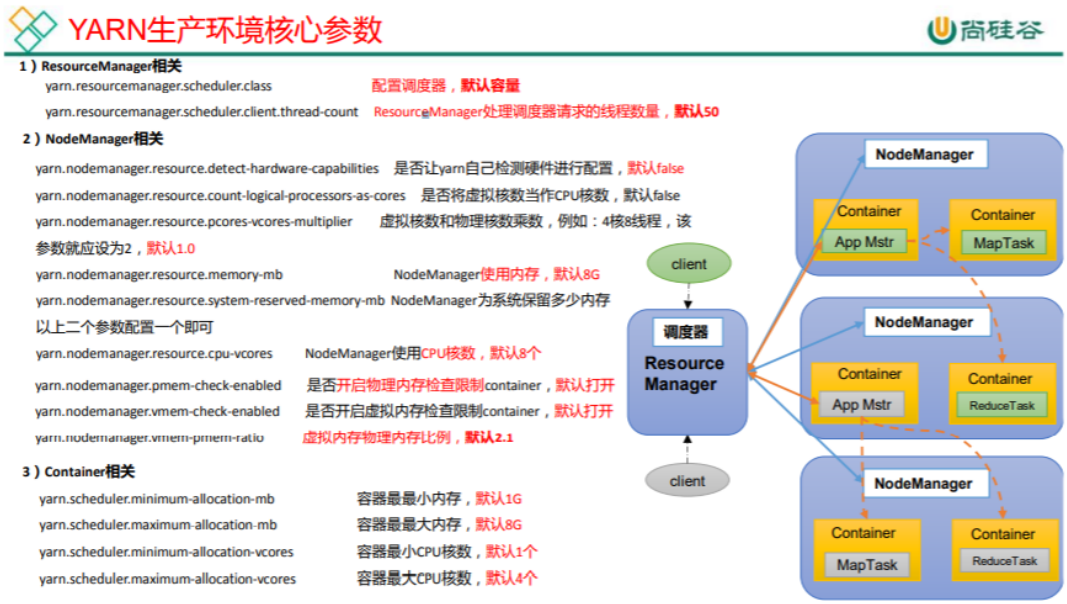
2、Capacity Scheduler（容量调度器）


Yahoo开发的多用户调度器
多队列：每个队列配置一定的资源量，每个队列采用FIFO调度策略
容量保证：管理员为每个队列设置资源上下限
灵活性：当一个队列中的资源有剩，可暂时共享给其他需要资源的队列，一旦该队列有新的应用程序提交，借调的资源会归还
多租户：支持多用户共享集群和多应用程序同时运行
　　为了防止同一个用户的作业独占队列中的资源，会对同一用户所提交的作业所占资源进行限定

3、Fair Scheduler（公平调度器）


Facebook开发的多用户调度器
与Capacity的不同点
（1）核心调度策略不同
　　Capacity：优先选择资源利用率低的队列
　　Fair：优先选择对资源的缺额比例高的队列
（2）每个队列单独设置资源的分配方式
　　Capacity：FIFO、DRF
　　Fair：FIFO、FAIR、DRF

# Yarn生产环境核心参数




# Yarn监控

配置yarn-site.xml开启日志聚合，日志聚集是Yarn提供的日志中央化管理功能，它能将运行完成的Container/任务日志上传到HDFS上，从而减轻NM负载，且提供一个中央化存储和分析机制。默认情况下，Container/任务日志存储在各个NM上

```
<property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!-- 开启日志聚合，如果没有设置，会显示3个目录 -->
    <property>
        <name>yarn.log-aggregation-enable</name>
        <value>true</value>
    </property>

    <properte>
        <name>yarn.log.server.url</name>
        <value>http://localhost:19888/jobhistory.logs</value>
    </properte>
```

配置mapred-site.xml

```
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
<property>
    <name>mapreduce.jobhistoy.address</name>
    <value>master:10020</value>
</property>
<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>master:19888</value>
</property>
```

在NN机器上执行sbin/mr-jobhistory-daemon.sh start historyserver命令