MapReduce的思想核心是“分而治之”，适用于大量复杂的任务处理场景（大规模数据处理场景）。

* Map负责“分”，即把复杂的任务分解为若干个“简单的任务”来并行处理。可以进行拆分的前提是这些小任务可以并行计算，彼此间几乎没有依赖关系。
* Reduce负责“合”，即对map阶段的结果进行全局汇总。

MapReduce处理的数据类型是<key, value>

Map: (k1, v1) -> (k2, v2)

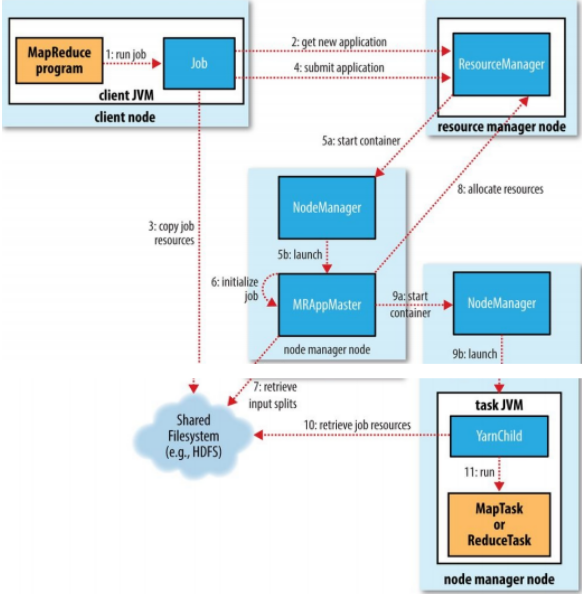
Reduce(k2, [v2,v2...v2]) -> (k3, v3)

参考：

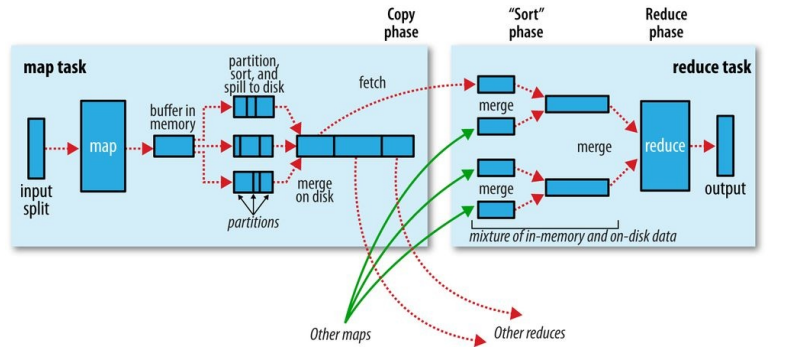
<https://www.cnblogs.com/luengmingbiao/p/11333618.html>

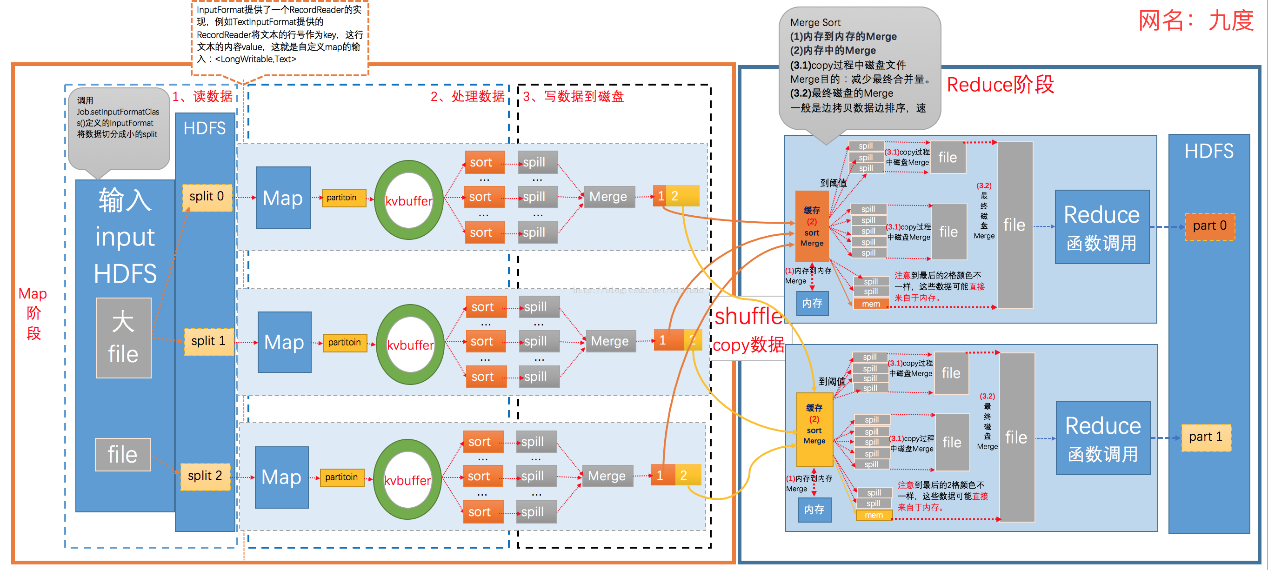
<https://www.jianshu.com/p/9e4d01b74600>

MapReduce在Yarn上运行机制



MapReduce执行过程





**MapReduce的八大步骤:**

Map阶段:

* 第一步: 通过FileInputFormat读取文件, 解析文件成为key, value对, 输出到第二步.
* 第二步: 自定义Map逻辑, 处理key1, value1, 将其转换为key2, value2, 输出到第三步.

Shuffle阶段:

* 第三步: 对key2, value2进行分区.
* 第四步: 对不同分区内的数据按照相同的key进行排序.
* 第五步: 分组后的数据进行规约(combine操作)，降低数据的网络拷贝（可选步骤）
* 第六步: 对排序后的数据, 将相同的key的value数据放入一个集合中, 作为value2.

Reduce阶段:

* 第七步: 对多个map的任务进行合并, 排序. 自定义reduce逻辑, 处理key2, value2, 将其转换为key3, value3, 进行输出.
* 第八步: 通过FileOutputFormat输出处理后的数据, 保存到文件.

1. FileInputFormat

默认是TextInputFormat

自定义InputFormat

public class MyInputFormat extends FileInputFormat

重写RecordReader方法

自定义RecordReader

public class MyRecordReader extends RecordReader

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.addInputPath(job, new Path("file:///D:\\input"));

2. Mapper

自定义Mapper

public class MyMapper extends Mapper

重写map方法

 job.setMapperClass(MyMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

3. 分区

自定义Partitioner

public class MyPartitioner extends Partitioner

重写getPartition方法

job.setPartitionerClass(MyPartitioner.class);

job.setNumReduceTasks(4);

4. 排序

4.1. 自定义类实现WritableComparable

重写

readFields

write

compareTo

4.2. 自定义类继承WritableComparator

重写compare

job.setSortComparatorClass

5. Combiner

Combiner组件的父类就是 Reducer

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.

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.

自定义一个 combiner 继承 Reducer，重写 reduce 方法

在 job 中设置 job.setCombinerClass(CustomCombiner.class)

6.分组

自定义类继承WritableComparator

public class MyGroupComparator extends WritableComparator

重写compare

job.setGroupingComparatorClass(MyGroupComparator.class);

7. Reducer

自定义Reducer

public class MyReducer extends Reducer

重写reduce方法

job.setReducerClass(MyReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

8. FileOutputFormat

默认是TextOutputFormat

自定义OutputFormat

public class MyOutputFormat extends FileOutputFormat

实现RecordWriter方法

自定义RecordWriter

public class MyRecordWriter extends RecordWriter

job.setOutputFormatClass(TextOutputFormat.class);

TextOutputFormat.setOutputPath(job, new Path("file:///D:\\output"));

MapReduce计数器

内置计数器列表

|  |  |
| --- | --- |
| MapReduce任务计数器 | org.apache.hadoop.mapreduce.TaskCounter |
| 文件系统计数器 | org.apache.hadoop.mapreduce.FileSystemCounter |
| FileInputFormat计数器 | org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter |
| FileOutputFormat计数器 | org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter |
| 作业计数器 | org.apache.hadoop.mapreduce.JobCounter |

自定义Counter

Counter counter = context.getCounter("MR\_COUNT", "MyRecordCounter");       counter.increment(1L);

Reduce端Join

Map端Join

job.addCacheFile(new URI("hdfs://node01:8020/cache\_file/product.txt"));

private Map<String, String> pdDatas=new HashMap<String, String>();

//在map之前手动读取pd.txt中的内容

@Override

protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context context)

throws IOException, InterruptedException {

//从分布式缓存中读取数据

URI[] files = context.getCacheFiles();

for (URI uri : files) {

BufferedReader reader = new BufferedReader(new FileReader(new File(uri)));

String line="";

//循环读取pd.txt中的每一行

while(StringUtils.isNotBlank(line=reader.readLine())) {

String[] words = line.split("\t");

pdDatas.put(words[0], words[1]);

}

reader.close();

}

Hadoop优化

1. 减少map task个数，合并小文件

CombineFileInputFormat

2. 使用Combiner

3. 使用压缩

mapreduce.map.output.compress --->设置是否启动map输出的压缩机制，默认为false。在需要减少网络传输的时候，可以设置为true。

mapreduce.map.output.compress.codec --->设置map输出压缩机制，默认为org.apache.hadoop.io.compress.DefaultCodec

4.mapreduce参数调优

1. mapreduce.task.io.sort.factor --->mr程序进行合并排序的时候，打开的文件数量，默认为10个.

2. mapreduce.task.io.sort.mb ---> mr程序进行合并排序操作的时候或者mapper写数据的时候，内存大小，默认100M

3. mapreduce.map.sort.spill.percent ---> mr程序进行flush操作的阀值，默认0.80。

4. mapreduce.reduce.shuffle.parallelcopies --->mr程序reducer copy数据的线程数，默认5。

5. mapreduce.reduce.shuffle.input.buffer.percent --->reduce复制map数据的时候指定的内存堆大小百分比，默认为0.70，适当的增加该值可以减少map数据的磁盘溢出，能够提高系统能。

6. mapreduce.reduce.shuffle.merge.percent --->reduce进行shuffle的时候，用于启动合并输出和磁盘溢写的过程的阀值，默认为0.66。如果允许，适当增大其比例能够减少磁盘溢写次数，提

高系统性能。同mapreduce.reduce.shuffle.input.buffer.percent一起使用。

7. mapreduce.task.timeout --->mr程序的task执行情况汇报过期时间，默认600000(10分钟)，设置为0表示不进行该值的判断。

Uber模式

针对多个小作业，开启uber模式，mapreduce会将所有的task任务放在一个JVM中完成，就需要每个task都去申请资源，启动一个Container容器，

而是多个task申请一份资源，资源会重复的利用，这样的话可以节省cpu及网络Io，磁盘Io的消耗，节省了job运行的时间。

Hadoop中对Uber模式的定义：

1：mapreduce.job.ubertask.enable=true，首先开启Uber模式，默认是false

2：mapreduce.job.ubertask.maxmaps map任务数的阀值9，map的数量<=9

mapreduce.job.ubertask.maxreduces reduce任务数的阀值1，reduce<=1

3：所有的输入文件的总长度<=默认的块的大小（128M）

4：mapreduce.map.memory.mb(默认是1024)<=内存需求(内存需求的大小由yarn.app.mapreduce.am.resource.mb来决定，默认1536M)

5：cpu<=yarn.app.mapreduce.am.resource.cpu-vcores(默认1)

6：采用非链式方式运行MR

数据倾斜

1. 在map中做部分聚合

2. 使用两个job，第一个job把key随机分布到reduce中进行部分聚合，第二个job正常按照相同key聚合。

3. Map join