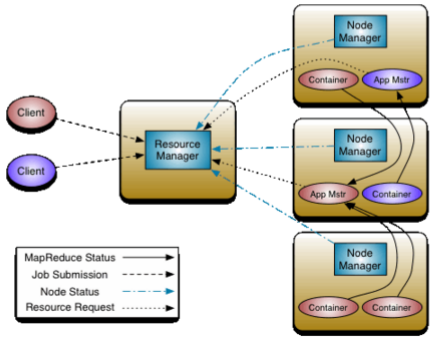
2.4.1、yarn的介绍

The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager (*RM*) and per-application ApplicationMaster (*AM*). An application is either a single job or a DAG of jobs.

The ResourceManager and the NodeManager form the data-computation framework. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.

The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.



思想：yarn的基本思想是将资源管理和作业调度/监视功能划分为单独的守护进程。其思想是拥有一个全局ResourceManager (RM)和每个应用程序的ApplicationMaster (AM)。应用程序可以是单个作业，也可以是一组作业

ResourceManager和NodeManager构成数据计算框架。ResourceManager是在系统中的所有应用程序之间仲裁资源的最终权威。NodeManager是每台机器的框架代理，负责监视容器的资源使用情况(cpu、内存、磁盘、网络)，并向ResourceManager/Scheduler报告相同的情况

每个应用程序ApplicationMaster实际上是一个特定于框架的库，它的任务是与ResourceManager协商资源，并与NodeManager一起执行和监视任务

启动：yarn-daemon.sh start RM

yarn-daemon.sh start NM

start-yarn.sh

stop-yarn.sh

2.4.2、yarn的安装

yarn属于hadoop的一个组件，不需要再单独安装程序，hadoop中已经存在

配置文件的设置

yarn也是一个集群，有主节点和从节点。

```
配置YARN集群：
cp mapred-site.xml.template mapred-site.xml
vi mapred-site.xml

<configuration>
<!--用于执行MapReduce作业的运行时框架-->
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
</configuration>

vi yarn-site.xml
```

```
<!--配置resourcemanager的主机-->
<property>
<name>yarn.resourcemanager.hostname</name>
<value>mini1</value>
</property>
<!--NodeManager上运行的附属服务-->
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>

<!--配置resourcemanager的scheduler的内部通讯地址-->
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value>mini1:8030</value>
</property>

<!--配置resoucemanager的资源调度的内部通讯地址-->
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>mini1:8031</value>
</property>

<!--配置resourcemanager的内部通讯地址-->
<property>
<name>yarn.resourcemanager.address</name>
<value>mini1:8032</value>
</property>

<!--配置resourcemanager的管理员的内部通讯地址-->
<property>
<name>yarn.resourcemanager.admin.address</name>
<value>mini1:8033</value>
</property>

<!--配置resourcemanager的web ui 的监控页面-->
<property>
<name>yarn.resourcemanager.webapp.address</name>
<value>mini1:8088</value>
</property>

配置slaves
```


2.4.3、MR的概念

一种分布式运算程序，分为map和reduce两个阶段

map阶段会有一个实体程序，不需要自己开发，用户只需要维护map方法就可以

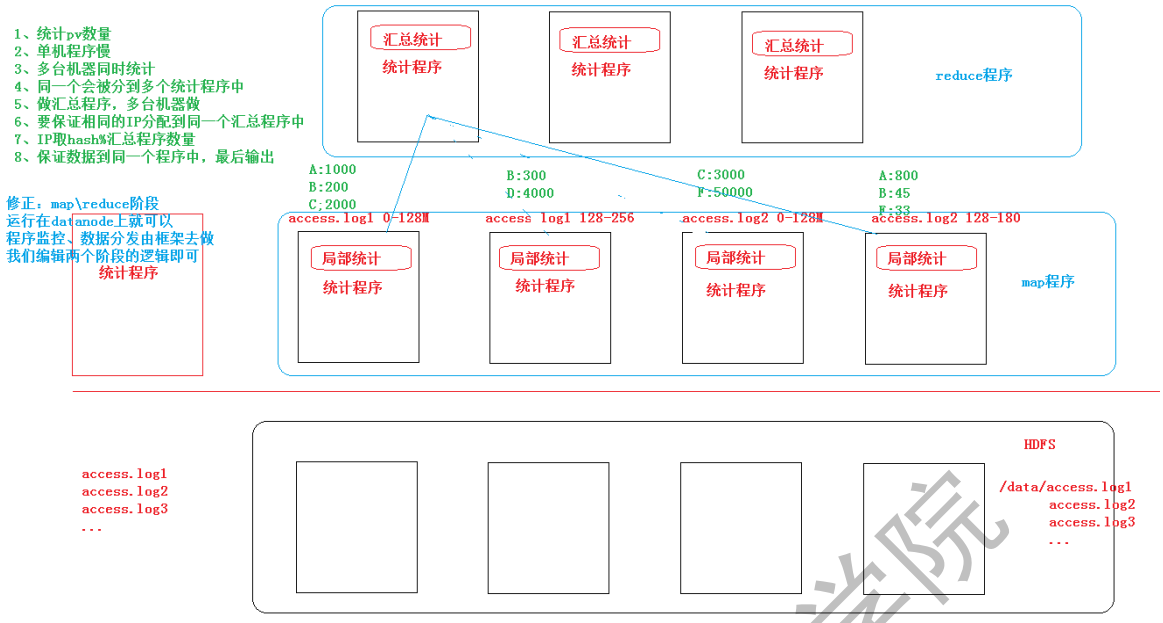
默认情况下map程序每读取一行数据就会调用一次map()方法，而且会将这一行数据的起始偏移量作为key,这一行数据作为value返回给框架，然后由框架写出context.write(key,value)

reduce阶段也有一个实体程序，不需要自己开发，用户只需要维护reduce方法就可以

reduce程序会收到map输出的中间结果数据，而且相同key的数据会到达同一个reduce程序实例，每个reduce程序会处理多个key的数据。reduce程序会将自己收集的数据按照key相同进行分组，对一组数据调用一次reduce方法，并且将参数传给reduce（key,迭代器values,context）,然后写出

2.4.4、MR原型解析

图解



集群上会运行多个程序，不会把所有资源都分配给你，各个部门的程序都需资源，那我们就需要一个资源分配和调度程序，帮我们分配cpu,内存和作业的调度。这个程序就是yarn

2.4.5、MR的核心思想

分而治之、移动计算不移动数据

2.4.6、MR的入门案例wordcount

统计单词数量思路

思路mapper端将数据读入映射成key，value键值对

获取到每一行数据后输出word：1这种形式的数据

map端完成后会将数据按照key的hash值模上reduce数量将数据分发

reduce端获取到数据后，将相同key的数据分到一组，将value存入迭代器中类似于value：{1,1,1,1}这样的数据

reduce端遍历迭代器，将value累加

reduce端将单词作为key，该单词的value的累加值做为value的输出

```
public class WordCount {
    /**
 * 框架在调用我们写的map业务方法时，会将数据作为参数（一个key，一个value）传入到map方法
 * KEYIN：是框架（maptask）要传递给map方法的输入参数中的key的数据类型
 * VALUEIN：是框架（maptask）要传递给map方法的输入参数的value的数据类型
 *
 * 在默认情况下，框架传入的key是框架从待处理的数据（文本文件）中读取到的某一行数据的起始偏移量
 * 所以，key的类型是Long
 * 框架传入的value是框架凑够待处理数据（文本文件）中读取到的某一行数据的内容，所以类型是String
 * 但是，Long或者是String是JAVA的原生数据类型，他们的序列化的效率比较低，
 * 所以，hadoop对其进行了改造，有一些替代品：LongWritable/Text
 *
 * map方法处理完数据之后需要写出一个key一个value
```

```
 * KEYOUT：是map方法处理完成后写出结果中的key的数据类型
 * VALUEOUT：是map方法处理完成后写出结果中的value的数据类型
 * 在此案例中就是Text，IntWritable
 */
    static class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
        /**
         * 原始数据：
         * hello qianfeng
         * hello gp1808
         * hello beijing
         * hello world
         *
         * 输出数据：
         * hello 1
         * qianfeng 1
         * hello 1
         * gp1808 1
         * hello 1
         * beijing 1
         *
         * 我么自己定义的map业务逻辑，maptask每读取一行数据就会调用一次map方法
         */

        /**
         * map方法是MR程序提供的，每读取一行数据就调用一次map（）方法
         * 也就是说读一行数据就要映射成一个键值对，处理完成后写出
         *
         */
        @Override
        protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
            //1、数据转换
            String line = value.toString();

            //2、根据分隔符 空格将该行的数据切分成单词数组
            String[] words = line.split(" ");

            //3、循环遍历，将单词按照<word,1>格式输出
            for (String word : words) {
                context.write(new Text(word),new IntWritable(1));
            }
        }
    }


    /**
     *  Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
     *  KEYIN，VALUEIN对应map端输出的数据key和value的类型
     *  KEYOUT,VALUEOUT是reduce阶段处理后输出的key和value的类型
     *  详细说下reduce输出的key和value的数据类型
     */
    /**
     * Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
     * reduce方法要接受的输入参数是一个key一个迭代器<T>values
     * reduce方法的调用规律：框架从map阶段的输出结果中找出所有相同key的key-value的数据组成
一组，
     * 每个key调用一次reduce方法
```

```
     */
     static class MyReducer extends Reducer<Text, IntWritable, Text,
LongWritable>{
        @Override
        protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
            /**
             * <hello,1>
             * <hello,1>
             * <hello,1>
             * <qianfeng,1>
             * <gp1808,1>
             *
             * 传入参数key：每一组相同单词的kv对的key
             * values：若干相同key的values的集合
             * 例如：[1,1,1,1,1,1,1,1,1]
             *
             */
            //定义计数器
            int count =0;
            //循环遍历values
            for (IntWritable value : values) {
                count += value.get();
            }
            //将结果输出到文件
            context.write(key,new LongWritable(count));
        }
    }

    /**
     *该类运行在hadoop客户端，main一运行，yarn的客户端就启动起来，与yarn的服务端进行通信
     * yarn的服务端负责启动mapreduce程序并使用MyMapper和MyReducer类
     * @param args
     */
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        //1、配置连接hadoop集群的参数
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS","hdfs://qianfeng");
        //2、获取job对象实例
        Job job = Job.getInstance(conf,"wordcount");
        //3、指定本业务job的路径
        job.setJarByClass(WordCount.class);
        //4、指定本业务job要使用的Mapper类
        job.setMapperClass(MyMapper.class);
        //5、指定mapper类的输出数据的kv的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        //6、指定本业务job要使用的Reducer类
        job.setReducerClass(MyReducer.class);
        //7、设置程序的最终输出结果的kv的类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        //8、设置job要处理的数据的输入源
        FileInputFormat.setInputPaths(job,new Path(args[0]));
        //9、设置job的输出目录
        FileOutputFormat.setOutputPath(job,new Path(args[1]));
        //10、提交job
```

```
        // job.submit();
        boolean b = job.waitForCompletion(true);
        System.exit(b?0:1);
    }
}
```

2.4.7、MR的分片机制

查看源码创建分片信息的集合splits，用于存放分片信息

遍历目录下文件

```
FileInputFormat.setInputDirRecursive(job,true);//递归遍历
```

循环取出每一个文件，然后做如下操作

获取文件大小及位置

计算每个分片的大小

```
FileInputFormat.setMaxInputSplitSize(job,100);
FileInputFormat.setMinInputSplitSize(job,1);
long minSize = Math.max(getFormatMinSplitSize()//1
                        , getMinSplitSize(job));//配置文件中可设置
long maxSize = getMaxSplitSize(job);//可在运行时指定-D
mapreduce.input.fileinputformat.split.maxsize=300M,默认是long的最大值
long splitSize = computeSplitSize(blockSize, minSize, maxSize);
return Math.max(minSize, Math.min(maxSize, blockSize));
```

判断文件是否可以分片（压缩格式有的可以进行分片，有的不可以，）

剩余文件的大小/分片大小>1.1时，循环执行封装分片信息的方法，具体如下

封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)

剩余文件的大小/分片大小<=1.1且 不等于0时，封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)

分片的注意事项：1.1倍的冗余。260M文件分几个片？

```
考虑Hadoop应用处理的数据集比较大，因此需要借助压缩。按照效率从高到低排列的
（1）使用容器格式文件，例如：顺序文件、RCFile、Avro数据格式支持压缩和切分文件。另外在配合使用一些快速压缩工具，例如：LZO、LZ4或者Snappy.
（2）使用支持切分压缩格式，例如gzip2
（3）在应用中将文件切分成块，对每块进行任意格式压缩。这种情况确保压缩后的数据库接近HDFS块大小。
（4）存储未压缩文件，以原始文件存储。
```

读取分片的细节：如果有多个分片

- 第一个分片读到末尾再多读一行
- 既不是第一个分片也不是最后一个分片第一行数据舍弃，末尾多读一行
- 最后一个分片舍弃第一行，末尾多读一行