Day04_MapReduce原理和基本操作

大数据-张军锋 Day04 MapReduce原理

FileSystem补充

获取FileSystem

```
用URI的形式连接
public static FileSystem fileSystem;

static{
    try {
        //fileSystem = FileSystem.get(CONF);
        URI uri = new URI("hdfs://master:9000");
        fileSystem = FileSystem.get(uri, CONF);
    } catch (Exception e) {
        System.out.println("无法连接HDFS,请检查配置");
        e.printStackTrace();
    }
}
```

newInstance和get方法取得FileSystem的区别:

> newInstance每次被调用时都会重新实例化一个FileSystem对象
> get在被调用时首先会检查jvm内有没有实例化的System对象,如果有就直接获取,如果没有就初始化一个,保证System对象为单例对象

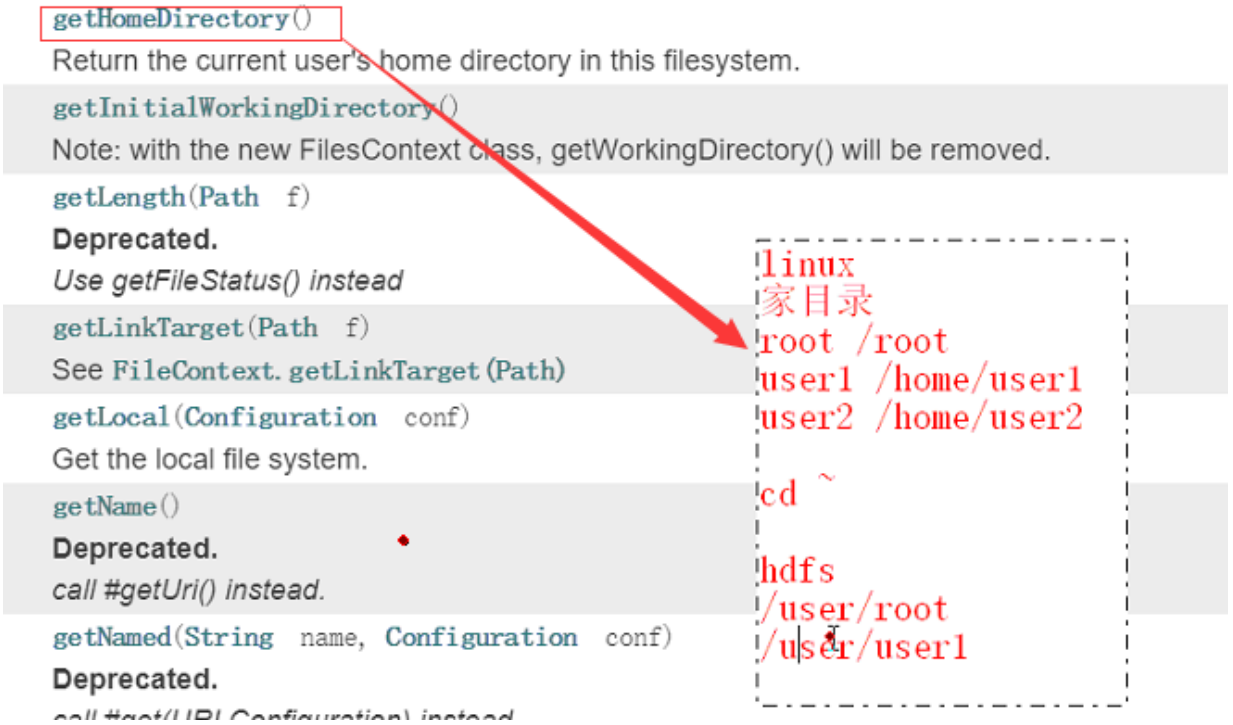
其他方法:

> immutable不可变的 getAllStatistics HDFS的统计信息 getAclStatus(Path path) 文件权限
> ACL访问控制权限
> rename重命名和剪切

- getFileStatus(Path f) : 返回给定路径的文件状态信息
- ListFileStatus(Path f)：返回给定路径下所有文件状态信息
- GetHomeDirectory() :获取家目录

获取用户的家目录

HDFS和Linux的区别:

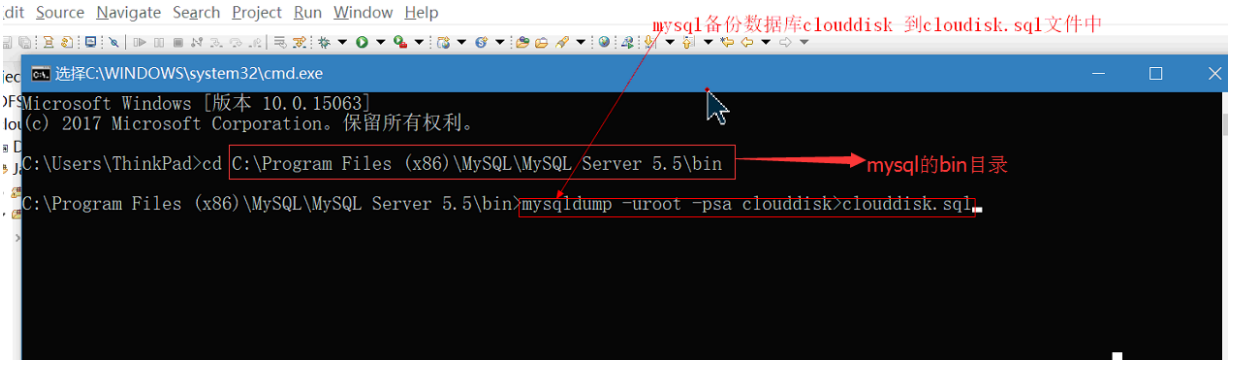


HDFS优势为廉价

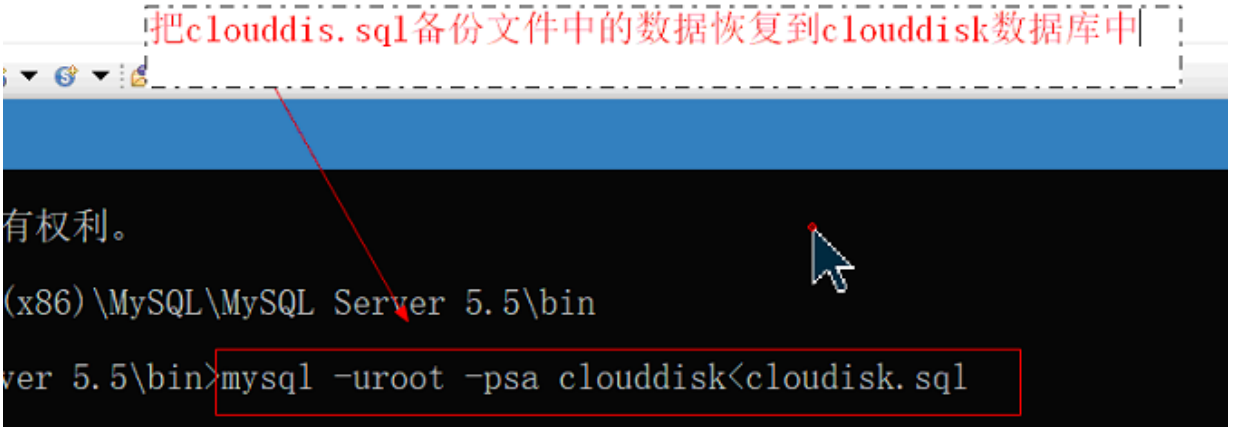
导入项目

导入数据库

1.mysql备份数据库

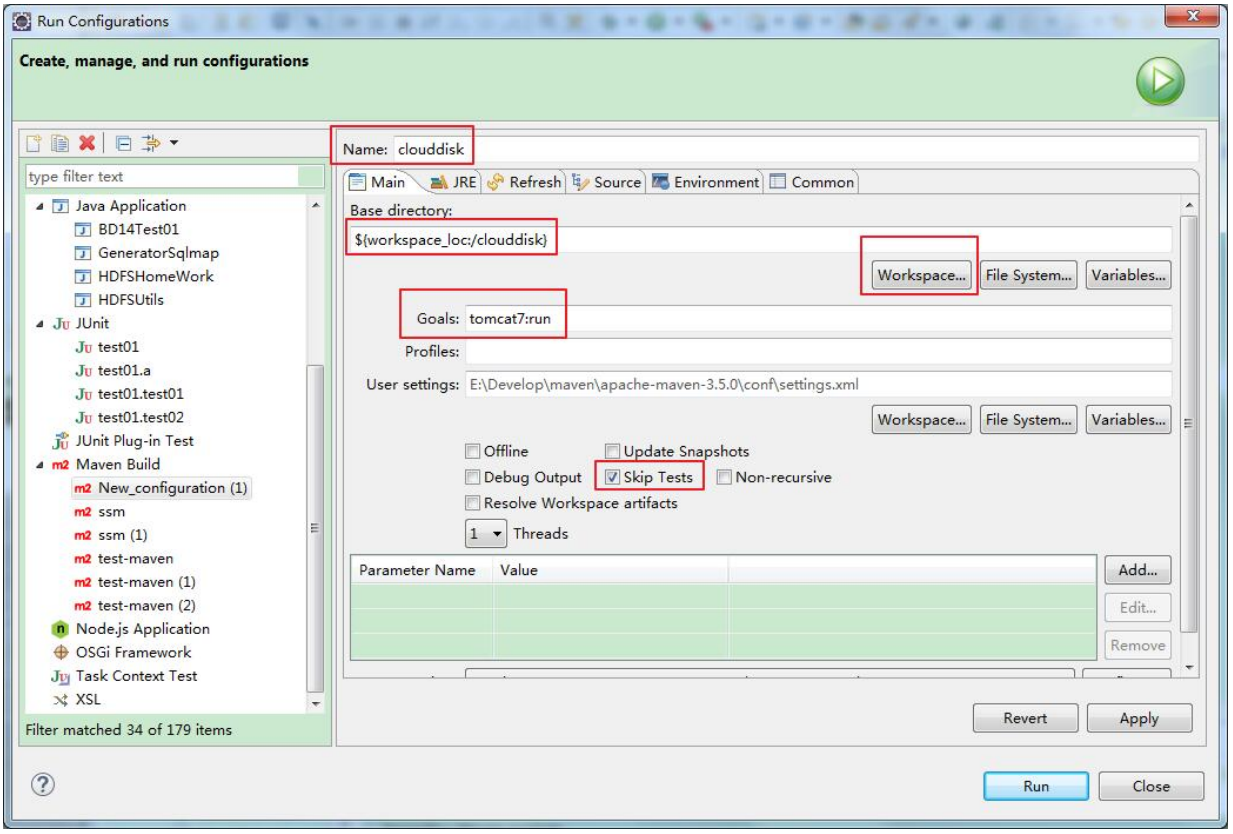


2.恢复到数据库



3.启动数据库

4.runconfig找到Maven Build



MapReduce

原理

map和reduce的运行原理

map：

1.加载数据源和解析数据源(抽取,转换)
2.哪里有数据就到哪里启动map,然后加载本地的数据文件
3.map把数据解析后发送给reduce,以key,value的形式发送,以key作为条件发送

reduce:聚合计算

1.由mapreduce引擎随机分配
2.数量由开发者决定,默认为1
3.reduce没有本地性
4.reduce处理key,则只要key相同的都会到同一个reduce节点

wordCount的运行过程

wordCount:

1.在节点上启动Map任务
2.加载之前也是key,value对,加载解析(解析为一个一个的单词)
3.以key,value的形式发送,必须保证同一个单词必须在一个reduce中
4.解析出来的单词作为key,简化计算则设置value为1
5.分配给reduce一个标号
6.获取key的Hash值key,hashcode()%reduce_number得到余数,标号分给对应标号的reduce
7.reduce会对key,value进行排序,分组和合并,都是根据key来操作,根据ascii进行排序分组合并
8.再聚合:聚合由程序定义(根据key聚合)
9.保存到HDFS上

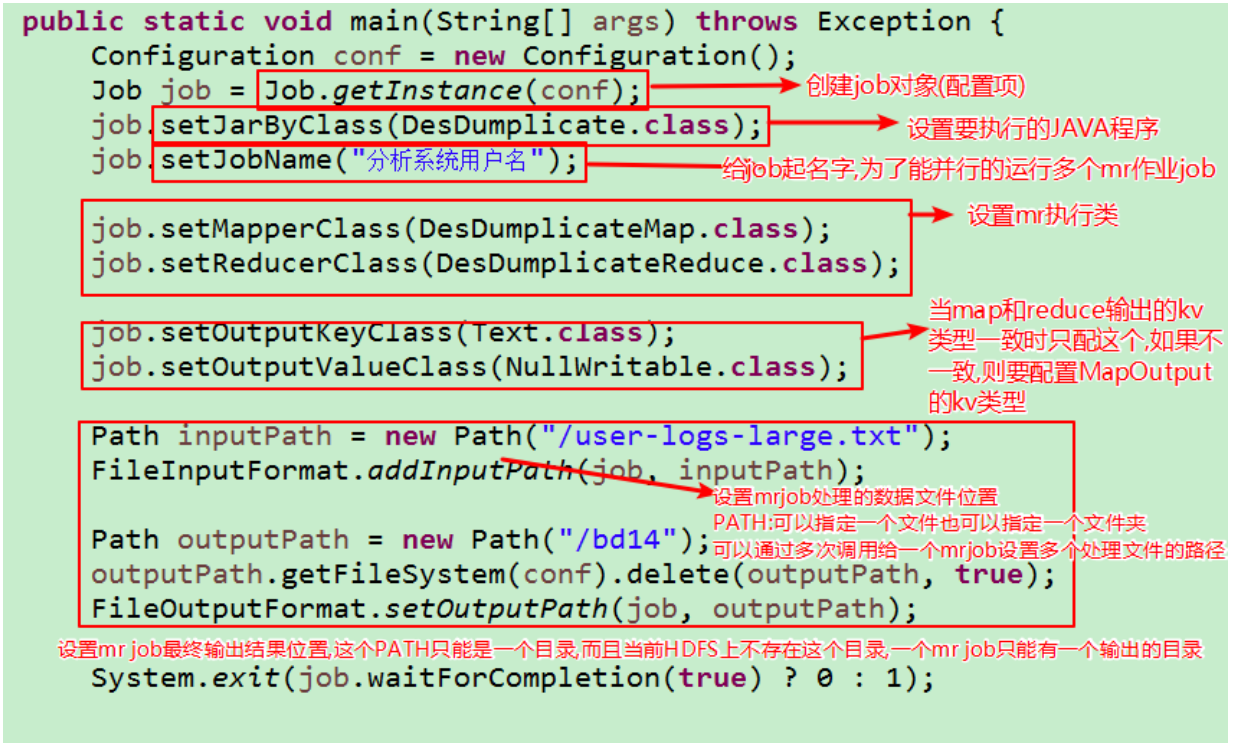
job配置

yarn:

- 可以并行的运行多个mr作业job
    - 则job需要设置名称

当map的输出kv类型和myjob最终的输出kv类型一致,则可以不用配置MapOutput的kv类型,如果不一致则必须要配置

设置mrjob处理的数据文件位置
PATH:可以指定一个文件也可以指定一个文件夹
可以通过多次调用给一个mrjob设置多个处理文件的路径



map,reduce和shuffel

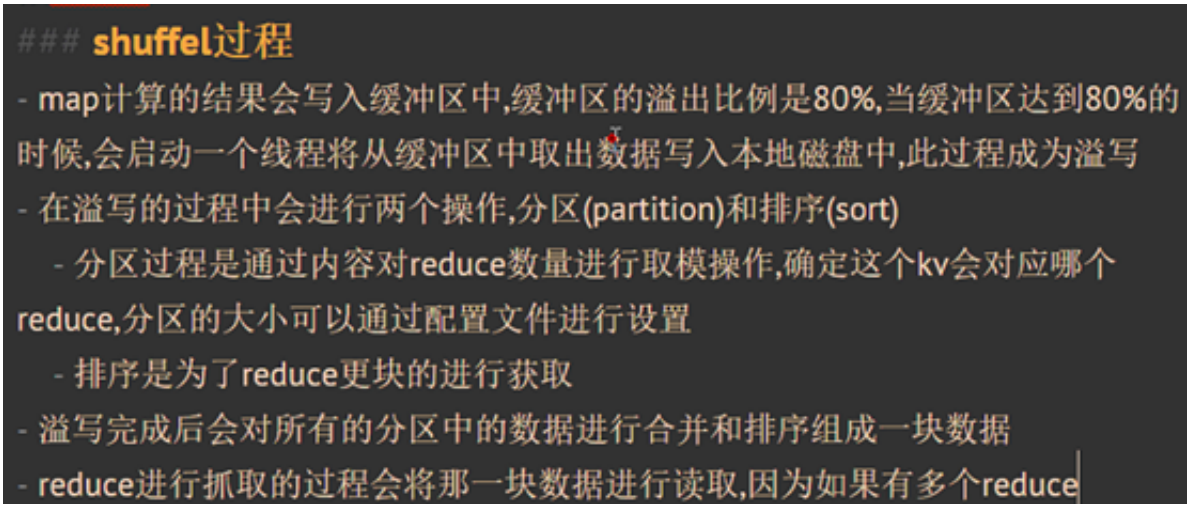
map节点:(map端)

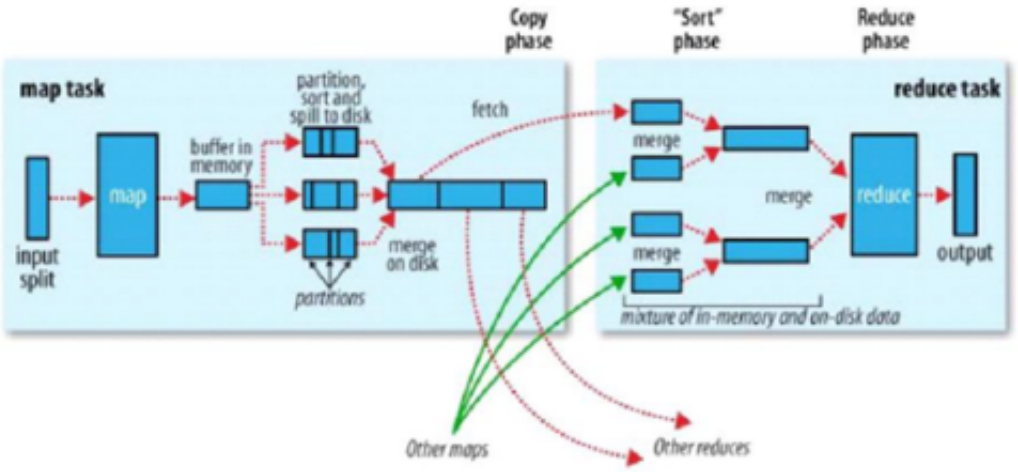
- 默认有几个block就有几个map
- split叫分片(分片为逻辑概念),block为分块(分块为物理概念)
- 默认情况map的数量由分片决定的,分片由物理的分块决定的
- 在实际过程中,自定义情况下由inputformat来决定的,可以自定义inputformat,也就是可以自定义分片
- 加载分片以kv对到内存中(k为每一行的偏移量,v就是该行的数据内容)(偏移量为)
- 传到map程序中,对每一个**kv**对都会调用map方法
- 输出kv对,先保存到map端的本地文件系统中(写入时会进行排序)

reduce节点:(reduce端)

- map执行完就会启动reduce
- 不指定就是1
- reduce到map端找到相应的自己的数据,拿到自己内存中
- 先写入reduce端的磁盘,写时合并和排序
- 再发送到reduce上执行,调用reduce方法,每一个**key**调用一次方法
- 通过outputformat类输出出去(保存数据)

shuffel:洗牌,混洗(整个mr中效率最低的过程)





map上的:

- 两次磁盘IO和一次网络IO为shuffe
- 由mr引擎来完成的
- 从map输出开始,kv对保存在内存的缓存区,内存空间可以自己设定(缓冲区有两个参数:缓冲区大小和缓冲区溢出比例)
- 当缓存区大小超过溢出比例时,会额外启动一个线程,从缓冲区中的数据写到本地磁盘,这个过程叫 溢写:
  1.partition 分区(确定每个kv该到哪个reduce)
  2.sort 根据key进行排序
- 当map排序好后就会再合并成一个文件,再发送到reduce上

reduce上的:

- reduce抓取数据,根据key进行合并

数据处理分析:

创建map类

```
public static class WordCountMap extends Mapper<LongWritable, Text,
Text, IntWritable> {
    private String[] infos;
    private Text okey = new Text();
    private final IntWritable oValue = new IntWritable(1);

    // 加载，转换，解析，抽取
    // context 传递mr执行过程的参数数据,通过writer方法可以将数据传给reduce
    // LongWritable key, Text value
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWri
table, Text, Text, IntWritable>.Context context)
            throws IOException, InterruptedException {
        // 解析一行数据，转换成一个单词组成的数组
        infos = value.toString().split("\\s");
        for (String i : infos) {
            // 把单词形成一个kv对发送给reduce
            okey.set(i);
            context.write(okey, oValue);
        }
    }
}
```

创建reducer

```
public static class WordCountReducer extends Reducer<Text, IntWrita
ble, Text, IntWritable> {
    private IntWritable oValue = new IntWritable();

    /**
     * Text key eg:hadoop(1,1),key是hadoop Iterable<IntWritable> val
ues
     * eg:hadoop(1,1),values是(1,1)
     *
     */
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
            Reducer<Text, IntWritable, Text, IntWritable>.Context c
ontext)
            throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        // 输出kv(单词，单词计数)
        oValue.set(sum);
        context.write(key, oValue);
    }
}
```

编写job

```java
// 组装一个job到mr引擎上执行
public static void main(String[] args) throws Exception {
    // 构建一个Configuration,用啦配置hdfs的位置和mr的各项参数
    Configuration conf = new Configuration();
    // 构建job对象
    Job job = Job.getInstance(conf);
    job.setJarByClass(WordCount.class);
    job.setJobName("第一个mr作业: wordcount");

    // 配置mr执行类
    job.setMapperClass(WordCountMap.class);
    job.setReducerClass(WordCountReducer.class);

    // 设置输出kv类型
    // Map
    // job.setMapOutputKeyClass(Text.class);
    // job.setMapOutputValueClass(IntWritable.class);

    // reducer
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    // 设置数据源(待处理)
    Path inputPath = new Path("/README.txt");
    FileInputFormat.addInputPath(job, inputPath);

    // 设置目标数据的存放位置
    Path outputPath = new Path("/bd14/output/wordcount");
    // 删除存在的文件
    outputPath.getFileSystem(conf).delete(outputPath, true);
    FileOutputFormat.setOutputPath(job, outputPath);

    // 启动作业，分布式计算交给mr引擎
    // 参数true代表打印日志
    boolean result = job.waitForCompletion(true);
    System.exit(result ? 0 : 1);
}
```