一、 课程计划

目录

二、 HDFS 基本原理

1. NameNode 概述

a、 NameNode 是 HDFS 的核心。

b、 NameNode 也称为 Master。

c、 NameNode 仅存储 HDFS 的元数据：文件系统中所有文件的目录树，并跟踪整个集群中的文件。

d、 NameNode 不存储实际数据或数据集。数据本身实际存储在 DataNodes 中。

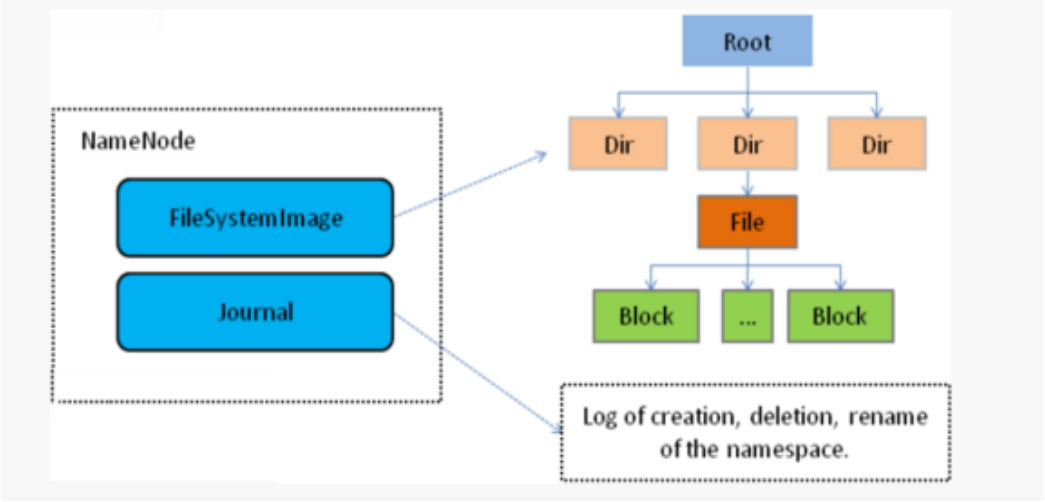
e、 NameNode 知道 HDFS 中任何给定文件的块列表及其位置。使用此信息 NameNode 知道如何从块中构建文件。

f、 NameNode 并不持久化存储每个文件中各个块所在的 DataNode 的位置信息，这些信息会在系统启动时从数据节点重建。

g、 NameNode 对于 HDFS 至关重要，当 NameNode 关闭时，HDFS / Hadoop 集群无法访问。

h、 NameNode 是 Hadoop 集群中的单点故障。

i、 NameNode 所在机器通常会配置有大量内存（RAM）。



2. DataNode 概述

a、 DataNode 负责将实际数据存储在 HDFS 中。

b、 DataNode 也称为 Slave。

c、 NameNode 和 DataNode 会保持不断通信。

d、 DataNode 启动时，它将自己发布到 NameNode 并汇报自己负责持有的块列表。

e、 当某个 DataNode 关闭时，它不会影响数据或群集的可用性。NameNode 将安排由其他 DataNode 管理的块进行副本复制。

f、 DataNode 所在机器通常配置有大量的硬盘空间。因为实际数据存储在 DataNode 中。

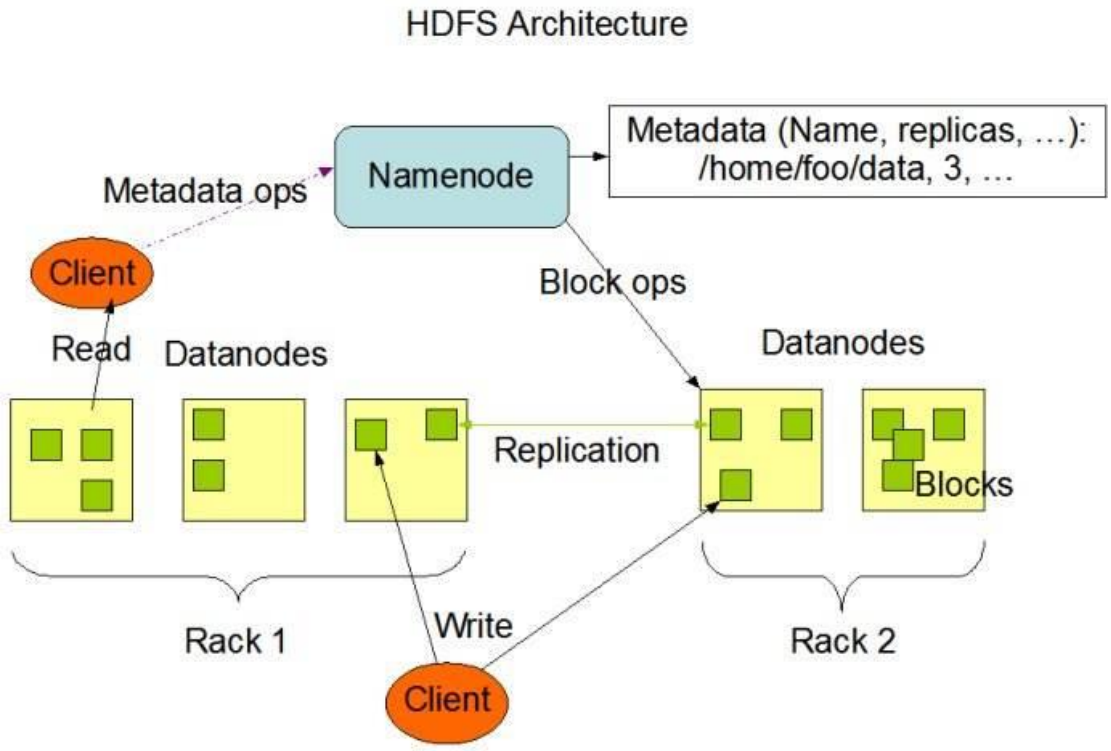
g、 DataNode 会定期（dfs.heartbeat.interval 配置项配置，默认是 3 秒）向 NameNode 发送心跳，如果 NameNode 长时间没有接受到 DataNode 发送的心跳， NameNode 就会认为该 DataNode 失效。

h、 block 汇报时间间隔取参数 dfs.blockreport.intervalMsec,参数未配置的话默认为 6 小时.

3． HDFS 的工作机制

NameNode 负责管理整个文件系统元数据；DataNode 负责管理具体文件数据块存储；Secondary NameNode 协助 NameNode 进行元数据的备份。

HDFS 的内部工作机制对客户端保持透明，客户端请求访问 HDFS 都是通过向 NameNode 申请来进行。



3.1. HDFS 写数据流程

详细步骤解析：

1、 client 发起文件上传请求，通过 RPC 与 NameNode 建立通讯，NameNode 检查目标文件是否已存在，父目录是否存在，返回是否可以上传；

2、 client 请求第一个 block 该传输到哪些 DataNode 服务器上；

3、 NameNode 根据配置文件中指定的备份数量及机架感知原理进行文件分配，返回可用的 DataNode 的地址如：A，B，C；

注：Hadoop 在设计时考虑到数据的安全与高效，数据文件默认在 HDFS 上存放三份，存储策略为本地一份，同机架内其它某一节点上一份，不同机架的某一节点上一份。

4、 client 请求 3 台 DataNode 中的一台 A 上传数据（本质上是一个 RPC 调用，建立 pipeline），A 收到请求会继续调用 B，然后 B 调用 C，将整个 pipeline 建立完成，后逐级返回 client；

5、 client 开始往 A 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（默认 64K），A 收到一个 packet 就会传给 B，B 传给 C；A 每传一个 packet 会放入一个应答队列等待应答。

6、 数据被分割成一个个 packet 数据包在 pipeline 上依次传输，在 pipeline 反方向上，逐个发送 ack（命令正确应答），最终由 pipeline 中第一个 DataNode 节点 A 将 pipeline ack 发送给 client；

7、 当一个 block 传输完成之后，client 再次请求 NameNode 上传第二个 block 到服务器。

详细步骤图：

3.2. HDFS 读数据流程

详细步骤解析：

1、 Client 向 NameNode 发起 RPC 请求，来确定请求文件 block 所在的位置；
2、 NameNode 会视情况返回文件的部分或者全部 block 列表，对于每个 block，NameNode 都会返回含有该 block 副本的 DataNode 地址；
3、 这些返回的 DN 地址，会按照集群拓扑结构得出 DataNode 与客户端的距离，然后进行排序，排序两个规则：网络拓扑结构中距离 Client 近的排靠前；心跳机制中超时汇报的 DN 状态为 STALE，这样的排靠后；
4、 Client 选取排序靠前的 DataNode 来读取 block，如果客户端本身就是 DataNode，那么将从本地直接获取数据；
5、 底层上本质是建立 Socket Stream（FSDataInputStream），重复的调用父类 DataInputStream 的 read 方法，直到这个块上的数据读取完毕；
6、 当读完列表的 block 后，若文件读取还没有结束，客户端会继续向 NameNode 获取下一批的 block 列表；
7、 读取完一个 block 都会进行 checksum 验证，如果读取 DataNode 时出现错误，客户端会通知 NameNode，然后再从下一个拥有该 block 副本的 DataNode 继续读。
8、 read 方法是并行的读取 block 信息，不是一块一块的读取；NameNode 只是返回 Client 请求包含块的 DataNode 地址，并不是返回请求块的数据；
9、 最终读取来所有的 block 会合并成一个完整的最终文件。

详细步骤图：

三、 HDFS 的应用开发

1. HDFS 的 JAVA API 操作

HDFS 在生产应用中主要是客户端的开发，其核心步骤是从 HDFS 提供的 api 中构造一个 HDFS 的访问客户端对象，然后通过该客户端对象操作（增删改查）HDFS 上的文件。

1.1. 搭建开发环境

创建 Maven 工程，引入 pom 依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.7.4</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>2.7.4</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.7.4</version>
    </dependency>
</dependencies>
```

配置 windows 平台 Hadoop 环境

在 windows 上做 HDFS 客户端应用开发，需要设置 Hadoop 环境,而且要求是 windows 平台编译的 Hadoop,不然会报以下的错误:

Failed to locate the winutils binary in the hadoop binary path java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

为此我们需要进行如下的操作：

A、在 windows 平台下编译 Hadoop 源码（可以参考资料编译，但不推荐）

B、使用已经编译好的 Windows 版本 Hadoop：

hadoop-2.7.4-with-windows.tar.gz

C、解压一份到 windows 的任意一个目录下

D、在 windows 系统中配置 HADOOP_HOME 指向你解压的安装包目录

E、在 windows 系统的 path 变量中加入 HADOOP_HOME 的 bin 目录

1.2. 构造客户端对象

在 java 中操作 HDFS，主要涉及以下 Class：

Configuration：该类的对象封转了客户端或者服务器的配置;

FileSystem：该类的对象是一个文件系统对象，可以用该对象的一些方法来对文件进行操作，通过 FileSystem 的静态方法 get 获得该对象。

```
FileSystem fs = FileSystem.get(conf)
```

get 方法从 conf 中的一个参数 fs.defaultFS 的配置值判断具体是什么类型的文件系统。如果我们的代码中没有指定 fs.defaultFS，并且工程 classpath 下也没有给定相应的配置，conf 中的默认值就来自于 hadoop 的 jar 包中的 core-default.xml，默认值为：file:///，则获取的将不是一个 DistributedFileSystem 的实例，而是一个本地文件系统的客户端对象。

1.3. 示例代码

```
Configuration conf = new Configuration();
//这里指定使用的是 hdfs 文件系统
conf.set("fs.defaultFS", "hdfs://node-21:9000");

//通过如下的方式进行客户端身份的设置
System.setProperty("HADOOP_USER_NAME", "root");

//通过 FileSystem 的静态方法获取文件系统客户端对象
FileSystem fs = FileSystem.get(conf);
//也可以通过如下的方式去指定文件系统的类型 并且同时设置用户身份
//FileSystem fs = FileSystem.get(new URI("hdfs://node-21:9000"), conf, "root");

//创建一个目录
fs.create(new Path("/hdfsbyjava-ha"), false);

//上传一个文件
fs.copyFromLocalFile(new Path("e:/hello.sh"), new Path("/hdfsbyjava-ha"));

//关闭我们的文件系统
 fs.close();
```

其他更多操作如文件增删改查请查看实例代码。

Stream 流形式操作

```
public void testUpload() throws Exception {
            FSDataOutputStream outputStream = fs.create(new Path("/1.txt"), true);
            FileInputStream inputStream = new FileInputStream("D:\\1.txt");
            IOUtils.copy(inputStream, outputStream);

    }
```

2. 案例：shell 定时采集数据至 HDFS

上线的网站每天都会产生日志数据。假如有这样的需求：要求在凌晨 24 点开始操作前一天产生的日志文件，准实时上传至 HDFS 集群上。

该如何实现？实现后能否实现周期性上传需求？如何定时？

2.1. 技术分析

HDFS SHELL:

hadoop fs -put //*满足上传文件，不能满足定时、周期性传入。*

Linux crontab:

crontab -e

0 0 * * * /shell/uploadFile2Hdfs.sh //每天凌晨 12：00 执行一次

2.2. 实现流程

一般日志文件生成的逻辑由业务系统决定，比如每小时滚动一次，或者一定大小滚动一次，避免单个日志文件过大不方便操作。

比如滚动后的文件命名为 access.log.x,其中 x 为数字。正在进行写的日志文件叫做 access.log。这样的话，如果日志文件后缀是 1\2\3 等数字，则该文件满足需求可以上传，就把该文件移动到准备上传的工作区间目录。工作区间有文件之后，可以使用 hadoop put 命令将文件上传。

2.3. 代码实现

```
#读取日志文件的目录，判断是否有需要上传的文件
echo "log_src_dir:"$log_src_dir
ls $log_src_dir | while read fileName
do
    if [[ "$fileName" == access.log.* ]]; then
    # if [ "access.log" = "$fileName" ];then
        date=`date +%Y_%m_%d_%H_%M_%S`
        #将文件移动到待上传目录并重命名
        #打印信息
        echo "moving $log_src_dir$fileName to $log_toupload_dir"xxxxx_click_log_$fileName
        mv $log_src_dir$fileName $log_toupload_dir"xxxxx_click_log_$fileName"$date
        #将待上传的文件path写入一个列表文件willDoing
        echo $log_toupload_dir"xxxxx_click_log_$fileName"$date >> $log_toupload_dir"willD
    fi
```

四、 初识 MapReduce

1. MapReduce 计算模型介绍

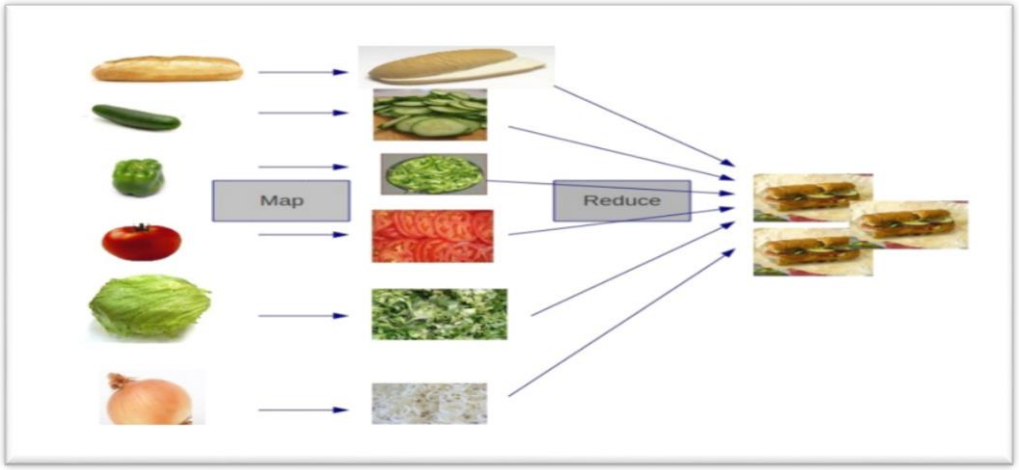
1.1. 理解 MapReduce 思想

MapReduce 思想在生活中处处可见。或多或少都曾接触过这种思想。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）。即使是发布过论文实现分布式计算的谷歌也只是实现了这种思想，而不是自己原创。

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。

这两个阶段合起来正是 MapReduce 思想的体现。



图：MapReduce 思想模型

还有一个比较形象的语言解释 MapReduce：

我们要数图书馆中的所有书。你数 1 号书架，我数 2 号书架。这就是“Map”。我们人越多，数书就更快。

现在我们到一起，把所有人的统计数加在一起。这就是“Reduce”。

1.2. Hadoop MapReduce 设计构思

MapReduce 是一个分布式运算程序的编程框架，核心功能是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 Hadoop 集群上。

既然是做计算的框架，那么表现形式就是有个输入（input），MapReduce 操作这个输入（input），通过本身定义好的计算模型，得到一个输出（output）。

对许多开发者来说，自己完完全全实现一个并行计算程序难度太大，而 MapReduce 就是一种简化并行计算的编程模型，降低了开发并行应用的入门门槛。

Hadoop MapReduce 构思体现在如下的三个方面：

- 如何对付大数据处理：分而治之

对相互间不具有计算依赖关系的大数据，实现并行最自然的办法就是采取分而治之的策略。并行计算的第一个重要问题是如何划分计算任务或者计算数据以便对划分的子任务或数据块同时进行计算。不可分拆的计算任务或相互间有依赖关系的数据无法进行并行计算！

- 构建抽象模型：Map 和 Reduce

MapReduce 借鉴了函数式语言中的思想，用 Map 和 Reduce 两个函数提供了高层的并行编程抽象模型。

Map: 对一组数据元素进行某种重复式的处理；

Reduce: 对 Map 的中间结果进行某种进一步的结果整理。

MapReduce 中定义了如下的 Map 和 Reduce 两个抽象的编程接口，由用户去编程实现:

map: (k1; v1) → [(k2; v2)]

reduce: (k2; [v2]) → [(k3; v3)]

Map 和 Reduce 为程序员提供了一个清晰的操作接口抽象描述。通过以上两个编程接口，大家可以看出 MapReduce 处理的数据类型是<key,value>键值对。

- 统一构架，隐藏系统层细节

如何提供统一的计算框架，如果没有统一封装底层细节，那么程序员则需要考虑诸如数据存储、划分、分发、结果收集、错误恢复等诸多细节；为此，MapReduce 设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面

的处理细节。

MapReduce 最大的亮点在于通过抽象模型和计算框架把需要做什么(what need to do)与具体怎么做(how to do)分开了，为程序员提供一个抽象和高层的编程接口和框架。程序员仅需要关心其应用层的具体计算问题，仅需编写少量的处理应用本身计算问题的程序代码。如何具体完成这个并行计算任务所相关的诸多系统层细节被隐藏起来,交给计算框架去处理：从分布代码的执行，到大到数千小到单个节点集群的自动调度使用。

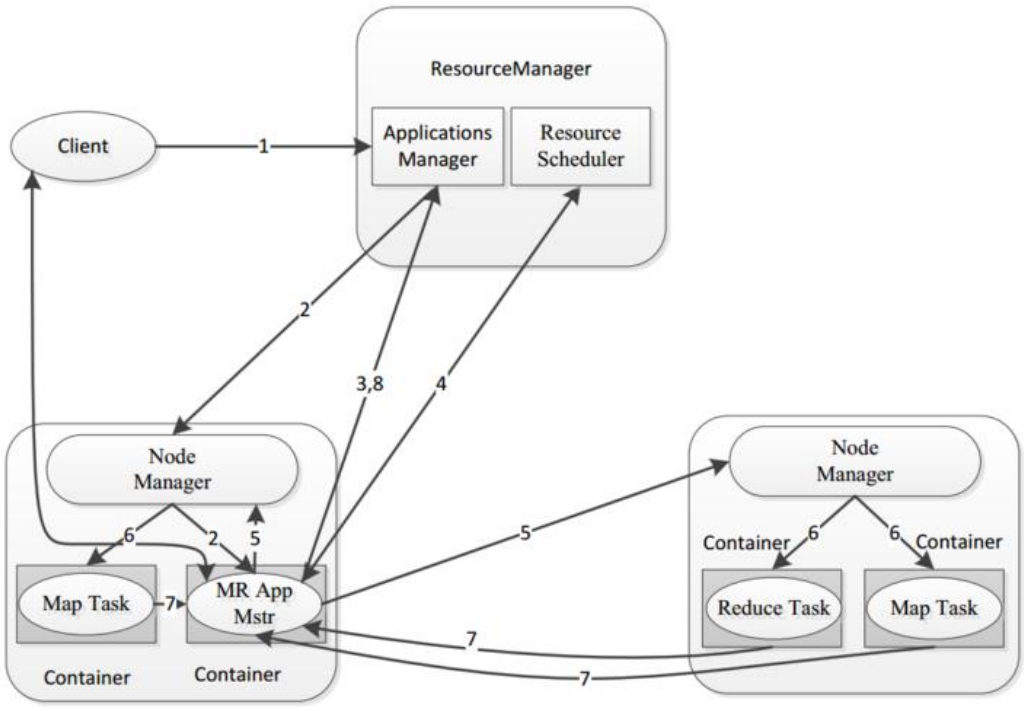
1.3. MapReduce 框架结构

一个完整的 mapreduce 程序在分布式运行时有三类实例进程：

1、MRAppMaster：负责整个程序的过程调度及状态协调

2、MapTask：负责 map 阶段的整个数据处理流程

3、ReduceTask：负责 reduce 阶段的整个数据处理流程



图：MapReduce 2.0 运行流程图

2. MapReduce 编程规范及示例编写

2.1. 编程规范

（1）**用户编写的程序分成三个部分：**Mapper，Reducer，Driver(提交运行 mr 程序的客户端)

（2）Mapper 的输入数据是 KV 对的形式（KV 的类型可自定义）

（3）Mapper 的输出数据是 KV 对的形式（KV 的类型可自定义）

（4）Mapper 中的业务逻辑写在 map()方法中

（5）map()方法（maptask 进程）对每一个<K,V>调用一次

（6）Reducer 的输入数据类型对应 Mapper 的输出数据类型，也是 KV

（7）Reducer 的业务逻辑写在 reduce()方法中

（8）Reducetask 进程对每一组相同 k 的<k,v>组调用一次 reduce()方法

（9）用户自定义的 Mapper 和 Reducer 都要继承各自的父类

（10）整个程序需要一个 Drvier 来进行提交，提交的是一个描述了各种必要信息的 job 对象

2.2. WordCount 示例编写

需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

(1)定义一个 mapper 类

```
//首先要定义四个泛型的类型
//keyin:    LongWritable        valuein: Text
//keyout: Text                  valueout:IntWritable


public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{
        //map 方法的生命周期：    框架每传一行数据就被调用一次
        //key :   这一行的起始点在文件中的偏移量
        //value:  这一行的内容
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
{
                //拿到一行数据转换为 string
                String line = value.toString();
```

```
            //将这一行切分出各个单词
            String[] words = line.split(" ");
            //遍历数组，输出<单词，1>
            for(String word:words){
                context.write(new Text(word), new IntWritable(1));
            }
        }
}
```

(2)定义一个 reducer 类

```
    //生命周期：框架每传递进来一个 kv 组，reduce 方法被调用一次
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
            //定义一个计数器
            int count = 0;
            //遍历这一组 kv 的所有 v，累加到 count 中
            for(IntWritable value:values){
                count += value.get();
            }
            context.write(key, new IntWritable(count));
    }
}
```

(3)定义一个主类，用来描述 job 并提交 job

```
public class WordCountRunner {
    //把业务逻辑相关的信息（哪个是 mapper，哪个是 reducer，要处理的数据在哪里，输出的结果放哪里……）描述成一个 job 对象
    //把这个描述好的 job 提交给集群去运行
    public static void main(String[] args) throws Exception {
            Configuration conf = new Configuration();
            Job wcjob = Job.getInstance(conf);
            //指定我这个 job 所在的 jar 包
//          wcjob.setJar("/home/hadoop/wordcount.jar");
            wcjob.setJarByClass(WordCountRunner.class);

            wcjob.setMapperClass(WordCountMapper.class);
            wcjob.setReducerClass(WordCountReducer.class);
```

```
        //设置我们的业务逻辑 Mapper 类的输出 key 和 value 的数据类型
        wcjob.setMapOutputKeyClass(Text.class);
        wcjob.setMapOutputValueClass(IntWritable.class);
        //设置我们的业务逻辑 Reducer 类的输出 key 和 value 的数据类型
        wcjob.setOutputKeyClass(Text.class);
        wcjob.setOutputValueClass(IntWritable.class);

        //指定要处理的数据所在的位置
        FileInputFormat.setInputPaths(wcjob, "hdfs://hdp-server01:9000/wordcount/data/big.txt");
        //指定处理完成之后的结果所保存的位置
        FileOutputFormat.setOutputPath(wcjob, new Path("hdfs://hdp-server01:9000/wordcount/output/"));

        //向 yarn 集群提交这个 job
        boolean res = wcjob.waitForCompletion(true);
        System.exit(res?0:1);
    }
```

3. MapReduce 程序运行模式

3.1. 本地运行模式

（1）mapreduce 程序是被提交给 LocalJobRunner 在本地以单进程的形式运行

（2）而处理的数据及输出结果可以在本地文件系统，也可以在 hdfs 上

（3）怎样实现本地运行？写一个程序，不要带集群的配置文件

本质是程序的 conf 中是否有 mapreduce.framework.name=local 以及 yarn.resourcemanager.hostname 参数

（4）本地模式非常便于进行业务逻辑的 debug，只要在 eclipse 中打断点即可

3.2. 集群运行模式

（1）将 mapreduce 程序提交给 yarn 集群，分发到很多的节点上并发执行

（2）处理的数据和输出结果应该位于 hdfs 文件系统

（3）提交集群的实现步骤：

将程序打成 JAR 包，然后在集群的任意一个节点上用 hadoop 命令启动

hadoop jar wordcount.jar cn.itcast.bigdata.mrsimple.WordCountDriver args