1、概述

1.1、Hadoop 的概念

Hadoop是一个应用java语言实现的开源、可靠、可扩展的分布式并行计算框架软件框架，大量的廉价的计算机组成的集群中运行海量数据的可靠的，可扩展的分布式并行计算框架，它可以让应用程序支持上千个节点和PB级别的数据。Hadoop是项目的总称，主要是由分布式存储（HDFS）、和分布式并行计算（MapReduce）等组成。

Hadoop可以实现，使用多台廉价的普通服务器，搭建分布式的集群，将其资源整合，实现大数据的处理。

1.2、Hadoop的思想之源：Google及起源

1） Google低成本之道:大量使用普通的pc服务器，提供有冗余的集群服务。

Google面对的数据存储和计算难题

Goolge 三大论文 大数据的三驾马车

文件系统 GFS

计算 MapReduce

大表格 BigTable

2）Hadoop的起源及发展史

Doug Cutting的全文检索引擎的架构Lucene,在海量数据的处理上遇到了和goole一样的难题.

Google公开了GFS和Mapreduce思想

Doug Cutting等人用了2年业余时间实现了HDFS 和Mapreduce机制

Goolge Hadoop

文件系统 GFS -> HDFS

计算 MapReduce -> Mapreduce

大表格 BigTable -> HBase

1.3、Hadoop的四个模块(四大组件)

Hadoop Common 公共模块（支持其他Hadoop模块的公用工具，包括 FileSystem（面向通用文件系统的抽象类）、远程过程调用（RPC） 和序列化库）

Hadoop HDFS 存储数据：一个为访问应用数据提供高吞吐量的分布式文件系统。

Hadoop YARN 管理资源(Hadoop2.x）：一个全局的任务调度与资源管理的 框架,YARN是mareduce的云操作系统

MapReduce 并行计算框架：一个基于Yarn的大数据分布式并行计算框架

1.4、Hadoop的体系架构

HDFS

Namenode（一个集群只有一个active）：用于**保存元数据信息**（Namenode将元数据保存在内存中），同时会在硬盘上保留一份。元数据（Filename、副本数，各个副本所在的节点的位置）

DataNode:**用于具体保存数据的**，在HDFS上保存的数据副本数，默认是3个，副本数可以设置

SecondaryNameNode: 用于同步元数据信息。

YARN

ResourceManager：负责全局任务的调度和资源管理（硬件资源（CPU，内存，硬盘））

Nodemanager：负责所在节点上的资源

Mapreduce

分两个阶段：Map和Reduce 计算框架，编程模型

“分而治之”的思想, 分布式并行计算

2、HDFS：Hadoop Distributed File System

2.1、HDFS存储架构剖析

2.1.1、namenode

* Namenode 是一个中心服务器，单一节点（简化系统的设计和实现），**负责管理文件系统的名字空间(namespace)以及客户端对文件的访问。**
* 文件操作，**NameNode 负责文件元数据的操作**，DataNode负责处理文件内容的读写请求，跟文件内容相关的**数据流不经过NameNode**，只会询问它跟那个DataNode联系，否则NameNode会成为系统的瓶颈。
* **副本存放在哪些DataNode上由 NameNode来控制**，根据全局情况做出块放置决定，读取文件时NameNode尽量让用户先读取最近的副本，降低带块消耗和读取时延
* **Namenode全权管理数据块的复制**，它周期性地从集群中的每个Datanode接收心跳信号和块状态报告(Blockreport)。接收到心跳信号意味着该Datanode节点工作正常。块状态报告包含了一个该Datanode上所有数据块的列表。

**心跳机制：datanode每隔3秒向namenode发送心跳信息**

2.1.2、DataNode

* 一个数据块在DataNode以文件存储在磁盘上，包括两个文件，一个是数据本身,一个是元数据包括数据块的长度，块数据的校验和以及时间戳。
* DataNode启动后向NameNode注册，通过后，周期性（1小时）的向NameNode上报所有的块信息。
* 心跳是每3秒一次，心跳返回结果带有NameNode给该DataNode的命令如复制块数据到另一台机器，或删除某个数据块。如果超过10分钟没有收到某个DataNode 的心跳，则认为该节点不可用。
* 集群运行中可以安全加入和退出一些机器。
* 专门用来储存文件，文件被分为多个block块储存在datanode节点上，为了保护数据的安全，默认会有三个副本数，会随机的分配在不同datanode上（就近原则）。

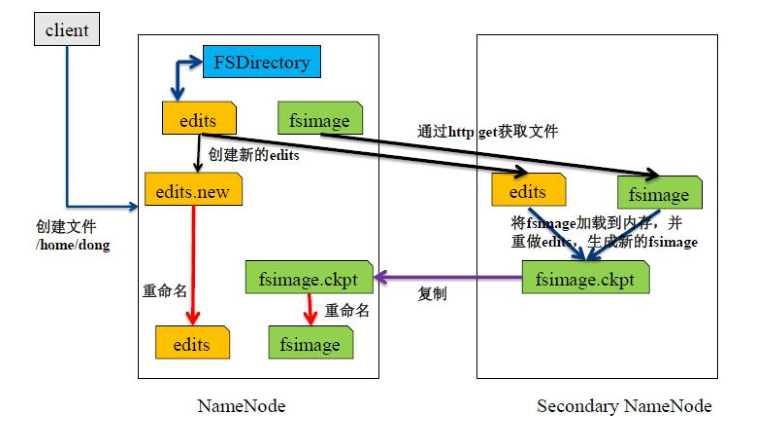
2.1.3、文件

* 文件切分成块（默认大小128M），以块为单位，每个块有多个副本存储在不同的机器上，副本数可在文件生成时指定（默认3）。
* NameNode 是主节点，存储文件的元数据如文件名，文件目录结构，文件属性（生成时间,副本数,文件权限），以及每个文件的块列表以及块所在的DataNode等等。
* DataNode 在本地文件系统存储文件块数据，以及块数据的校验和。
* 可以创建、删除、移动或重命名文件，当文件创建、写入和关闭之后不能修改文件内容。

2.1.4、数据损坏处理

* 当DataNode读取block的时候，它会计算checksum。
* 如果计算后的checksum，与block创建时值不一样，说明该block已经损坏。
* Client读取其它DataNode上的block。
* NameNode标记该块已经损坏,然后复制block达到预期设置的文件备份数。
* DataNode 在其文件创建后三周验证其checksum。

2.2、Namenode启动流程



**注：NameNode数据存放在两个地方，一个是内存，一个是本地磁盘。在磁盘中的存储分为fsimage（镜像文件）、edits（编辑日志文件）。**

2.2.1、Namenode启动过程

1）第一次启动hdfs，我们需要格式化hdfs文件系统。

**hdfs bin/hdfs namenode -format**

格式化hdfs的后我们会生成fsimage镜像文件，存储文件系统的元数据。

2）格式化hdfs后，启动NameNode。首先将fsimage（镜像）载入内存，并执行（replay）编辑日志editlog的的各项操作；获取文件的元数据如文件名，文件目录结构，文件属性（生成时间、副本数、文件权限）以及每个文件的块列表和块的位置。

3）一旦在内存中建立文件系统元数据映射，则创建一个新的fsimage文件（这个过程不需SecondaryNameNode）和一个空的editlog；

4）启动DataNode，启动DataNode首先会向NameNode去注册，然后会向NameNode发送一些块的报告。NameNode为了核对dataNode上的块(数据)是否有缺失情况。（在安全模式下）

5）此刻namenode运行在安全模式。即NameNode的文件系统对于客服端来说是只读的。(显示目录，显示文件内容等。写、删除、重命名都会失败)；

6）当用户用文件系统对文件进行创建、上传、读取、删除等操作的时候，NameNode会将这些操作写入编辑日志文件edits中。由此看出edits这个文件存放的是文件系统中元数据的变化信息。

7）第二次启动hdfs，启动NameNode ,nameNode首先读取镜像文件fsimage和编辑日志文件edits，这时候，nameNode 拥有完整的元数据，加载到内存，然后创建一个新的fsimage文件，并创建一个新的edits文件。然后接着走第一次启动的2.3.4.5.6步骤。

2.2.2、元数据合并流程

NameNode的元数据信息先往edits文件中写，当edits文件达到一定的阈值**(3600秒或大小到64M)的**时候，会开启合并的流程。

1）当开始合并的时候，SecondaryNameNode会把edits和fsimage拷贝到所在服务器所在内存中，合并生成名为fsimage.ckpt的文件。

2）将fsimage.ckpt文件拷贝到NameNode上，删除原有的fsimage，并将fsimage.ckpt重命名为fsimage。

3）当SecondaryNameNode将edits和fsimage拷贝走之后，NameNode会立刻生成一个edits.new文件，用于记录新来的元数据，当合并完成之后，原有的edits文件才会被删除，并将edits.new文件重命名为edits文件，开启下一轮流程。

配置合并的阈值

<property>

<name>dfs.namenode.checkpoint.period</name>

<value>3600</value>

</property>

<property>

<name>fs.checkpoint.size</name>

<value>67108864</value>

</property>

2.2.3、安全模式：safemode

Namenode的一种状态，启动时会自动进入安全模式，在安全模式，文件系统不允许有任何修改，“只读不写”。目的，是在系统启动时检查各个DataNode上数据的有效性。

1）Namenode启动会自动进入安全模式。

2）手动进入安全模式。

bin/hdfs dfsadmin -safemode enter 进入安全模式

bin/hdfs dfsadmin -safemode leave 离开安全模式

3）阀值=0.999

datanode会周期性的报告block信息。

Namenode会计算block的损坏率，

当阀值<0.999f时系统会进入安全模式，HDFS只读不写。

配置安全模式临界值,默认

<property>

<name>dfs.namenode.safemode.threshold-pct</name>

<value>0.999f</value>

</property>

2.3、HDFS的特点

2.3.1、HDFS的优点

1）**处理超大文件：**这里的超大文件通常是指百MB、数百TB大小的文件。目前在实际应用中，HDFS已经能用来存储管理PB级的数据了。

2）**流式的访问数据：**HDFS的设计建立在更多地响应"一次写入、多次读取"任务的基础上。这意味着一个数据集一旦由数据源生成，就会被复制分发到不同的存储节点中，然后响应各种各样的数据分析任务请求。在多数情况下，分析任务都会涉及数据集中的大部分数据，也就是说，对HDFS来说，请求读取整个数据集要比读取一条记录更加高效。

3）**运行于廉价的商用机器集群上：**Hadoop设计对硬件需求比较低，只须运行在低廉的商用硬件集群上，而无需昂贵的高可用性机器上。廉价的商用机也就意味着大型集群中出现节点故障情况的概率非常高。这就要求设计HDFS时要充分考虑数据的可靠性，安全性及高可用性。

2.3.2、HDFS的缺点

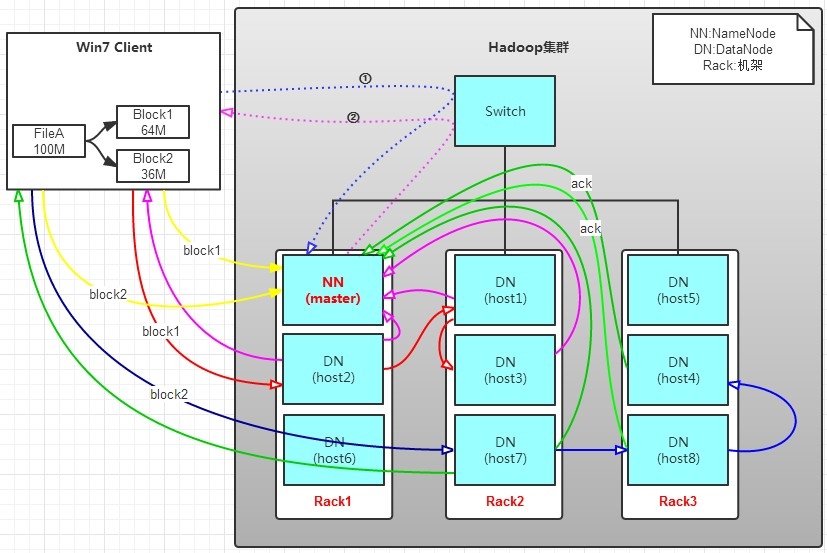
**1）不适合低延迟数据访问：**如果要处理一些用户要求时间比较短的低延迟应用请求，则HDFS不适合。HDFS是为了处理大型数据集分析任务的，主要是为达到高的数据吞吐量而设计的，这就可能要求以高延迟作为代价。

**2）无法高效存储大量小文件：**因为Namenode把文件系统的元数据放置在内存中，所以文件系统所能容纳的文件数目是由Namenode的内存大小来决定。一般来说，每一个文件、文件夹和Block需要占据150字节左右的空间，所以，如果你有100万个文件，每一个占据一个Block，你就至少需要300MB内存。当前来说，数百万的文件还是可行的，当扩展到数十亿时，对于当前的硬件水平来说就没法实现了。还有一个问题就是，因为Map task的数量是由splits来决定的，所以用MR处理大量的小文件时，就会产生过多的Maptask，线程管理开销将会增加作业时间。举个例子，处理10000M的文件，若每个split为1M，那就会有10000个Maptasks，会有很大的线程开销；若每个split为100M，则只有100个Maptasks，每个Maptask将会有更多的事情做，而线程的管理开销也将减小很多。

**3）不支持多用户写入及任意修改文件：**在HDFS的一个文件中只有一个写入者，而且写操作只能在文件末尾完成，即只能执行追加操作。目前HDFS还不支持多个用户对同一文件的写操作，以及在文件任意位置进行修改。

2.4、HDFS文件读写流程

2.4.1、HDFS文件写流程



有一个文件FileA，100M大小。Client将FileA写入到HDFS上。

HDFS按默认配置。HDFS分布在三个机架上Rack1，Rack2，Rack3。

**a.** Client将FileA按64M分块。分成两块，block1和Block2;

**b.** Client向nameNode发送写数据请求，如图蓝色虚线①——>。

**c.** NameNode节点，记录block信息。并返回可用的DataNode，

如粉色虚线②———>。Block1: host2,host1,host3

     Block2: host7,host8,host4

    原理：NameNode具有RackAware机架感知功能，这个可以配置。

若client为DataNode节点，那存储block时，规则为：副本1，同client的节点上；副本2，不同机架节点上；副本3，同第二个副本机架的另一个节点上；其他副本随机挑选。

若client不为DataNode节点，那存储block时，规则为：副本1，随机选择一个节点上；副本2，不同副本1的机架上；副本3，同副本2相同的另一个节点上；其他副本随机挑选。

**d.** client向DataNode发送block1；发送过程是以流式写入。

    流式写入过程，

1. 将64M的block1按64k的package划分;
2. 然后将第一个package发送给host2;
3. host2接收完后，将第一个package发送给host1，同时client 向host2发送第二个package；
4. host1接收完第一个package后，发送给host3，同时接收host2 发来的第二个package。
5. 以此类推，如图红线实线所示，直到将block1发送完毕。
6. host2,host1,host3向NameNode，host2向Client发送通知，说 “消息发送完了”。如图粉红颜色实线所示。
7. client收到host2发来的消息后，向namenode发送消息，说我写 完了。这样就真完成了。如图黄色粗实线
8. 发送完block1后，再向host7，host8，host4发送block2，如图 蓝色实线所示。
9. 发送完block2后，host7,host8,host4向NameNode，host7向 Client发送通知，如图浅绿色实线所示。
10. client向NameNode发送消息，说我写完了，如图黄色粗实线。。。这样就完毕了。

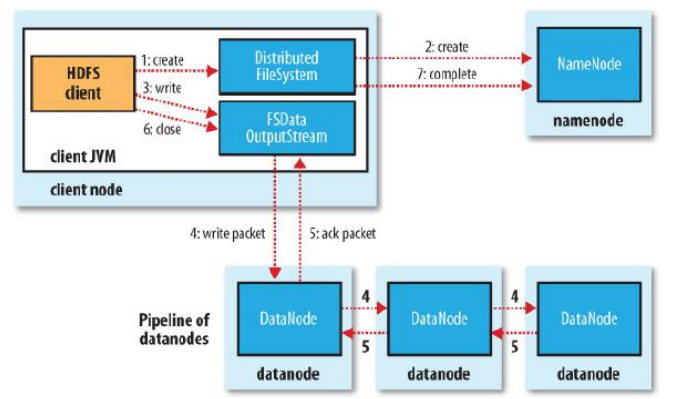
**分析，**通过写过程，我们可以了解到：

**①**写1T文件，我们需要3T的存储，3T的网络流量贷款。

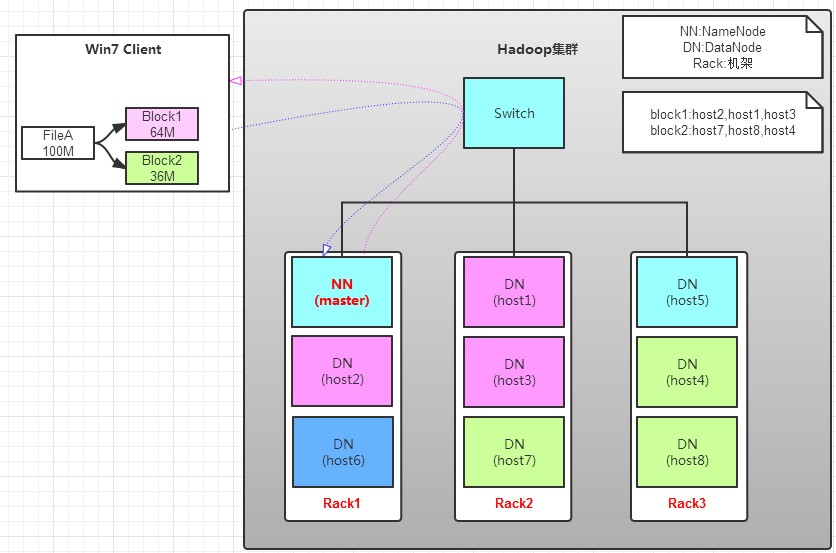
**②**在执行读或写的过程中，NameNode和DataNode通过HeartBeat 进行保存通信，确定DataNode活着。如果发现DataNode宕机， 就将宕机的DataNode上的数据，放到其他节点去。读取时，

要读其他节点去。

**③**挂掉一个节点，没关系，还有其他节点可以备份；甚至，挂掉 某一个机架，也没关系；其他机架上，也有备份。



2.4.2、HDFS文件读流程



如图所示，client要从datanode上，读取FileA。而FileA由block1和block2组成。

那么，读操作流程为：

**a.**client向namenode发送读请求。

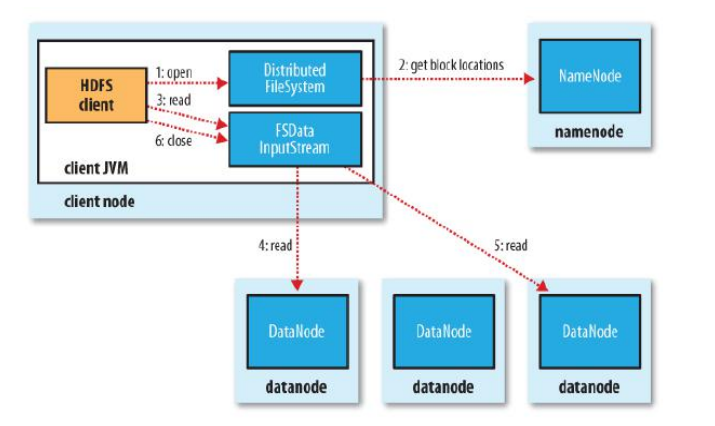
**b.**namenode查看Metadata信息，返回fileA的block的位置。

    block1:host2,host1,host3  block2:host7,host8,host4

**c.**block的位置是有先后顺序的，先读block1，再读block2。而且block1去host2上读取；然后block2，去host7上读取；

上面例子中，client位于机架外，那么如果client位于机架内某个DataNode上，例如,client是host6。那么读取的时候，遵循的规律是：

**优选读取本机架上的数据**。



2.5、HDFS Java API

2.6、HDFS数据安全性如何保证

1) 存储在HDFS系统上的文件，会分割成128M大小的block存储在不同的节点上，block的副本数默认3份，也可配置成更多份；

2) 第一个副本一般放置在与client（客户端）所在的同一节点上（若客户端无datanode，则随机放），第二个副本放置到与第一个副本同一机架的不同节点，第三个副本放到不同机架的datanode节点，当取用时遵循就近原则；

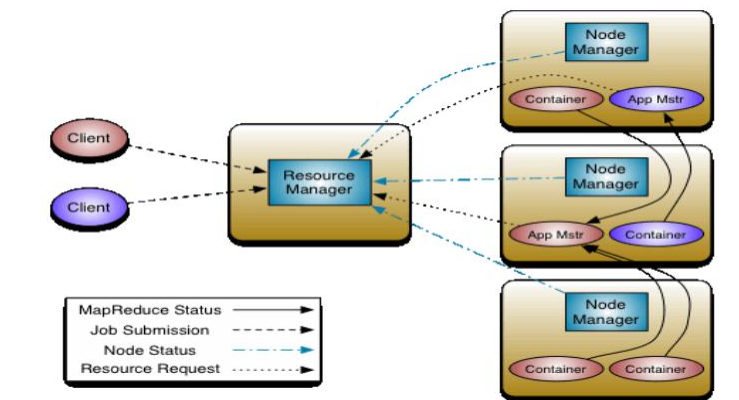
3) datanode已block为单位，每3s报告心跳状态，做10min内不报告心跳状态则namenode认为block已死掉，namonode会把其上面的数据备份到其他一个datanode节点上，保证数据的副本数量；

4) datanode会默认每小时把自己节点上的所有块状态信息报告给namenode

5) 采用safemode模式：datanode会周期性的报告block信息。Namenode会计算block的损坏率，当阀值<0.999f时系统会进入安全模式，HDFS只读不写。 HDFS元数据采用secondaryname备份或者HA备份。

3、Yarn

3.1、Yarn架构



1）ResourceManager

在YARN中，ResourceManager的主要作用是管理和分配资源。它的主要职责有：

①处理客户端请求；

②跟踪各个可用的NodeManager节点及其可用资源；

* ③分配可用资源给各个应用及其任务；
* ④监控各个ApplicationMaster的健康状况，并负责启动和重启失败的 ApplicationMaster。

实际上，ResourceManager也可以拆分成两个主要组成部分：Scheduler 和 ApplicationsManager。

Scheduler根据一定的限制（资源容量、队列、数据位置、ACL等）负责给各个在集群运行的应用分配资源。它是一个纯资源的调度器，不会去对应用的状态执行监控或是跟踪。并且，它不保障当任务因为应用或硬件出现故障而导致失败时去重启失败的任务。它仅基于应用所需要的资源来执行调度。

Scheduler还是一个可插拔的组件，当前Hadoop提供了几种不同的现成的调度器，如CapacityScheduler和FairScheduler等，它们根据不同的算法在不同的队列和应用间分配集群的资源。

而ApplicationsManager负责接受任务提交，协商用于启动应用的ApplicationMaster所需的第一个container，并且提供当AM出现故障时重启AM container的服务。

2）NodeManager

NodeManager是MRv1架构中TaskTracker的一种更加普通和高效的版本。它没有固定数量的map和reduce插槽，而是拥有许多动态创建的资源容器。容器的大小取决于它所包含的资源量（内存、CPU、磁盘和网络IO）。目前，仅支持内存和CPU。未来可使用cgroups来控制磁盘和网络IO。NodeManager的主要职责有：⑴ 单个节点上的资源管理；⑵ 处理来自ResourceManager的命令；⑶ 处理来自ApplicationMaster的命令。

3）ApplicationMaster

当一个应用程序从client提交到ResourceManager，RM就会启动一个称为ApplicationMaster的轻量级进程来协调应用程序内的所有任务的执行，它管理着该应用程序的整个生命周期。每个特定的在集群运行的应用程序都会由一个特定的AM来管理。它的主要职责有：

① 对数据进行切分

② 为应用程序向ResourceManager申请资源，并分配给内部任务

③ 监视任务执行状况，重启失败的任务

④ 在任务完成后注销容器

ApplicationMaster和属于它的应用程序的任务，都是在受NodeManager控制的资源容器container中运行的。

4）Container

在YARN中，资源是以容器container的形式提供给各个应用程序的任务。它是一个抽象的概念，主要包含内存、CPU、磁盘和网络IO。每个特定任务的执行都是在一个给定的container中完成。container的主要特性有：

* (1) 运行不同类型的任务（包含ApplicationMaster本身）.
* (2) 动态创建，具有不同的大小（内存、cpu等）.

5）Client

Client，即YARN客户端，通过它可以向RM发送各种不同的请求，用于向RM提交不同类型的应用程序，并且还可以通过它向RM发送一些控制命令，如查询应用的状态，停止应用等。

3.2、Yarn的基本流程（执行流程）

1）client向yarn提交job，首先找ResourceManager分配资源，

2）ResourceManager开启一个Container,在Container中运行一个Application manager

3）Application manager找一台nodemanager启动Application master，计算任务所需的计算

4）Application master向Application manager（Yarn）申请运行任务所需的资源

5）Resource scheduler将资源封装发给Application master

6）Application master将获取到的资源分配给各个nodemanager

7）各个nodemanager得到任务和资源开始执行map task

8）map task执行结束后，开始执行reduce task

9）map task和 reduce task将执行结果反馈给Application master

10）Application master将任务执行的结果反馈Application manager

4、MapReduce

4.1、MapReduce的架构和应用场景

1）MapReduce将计算过程分为两个阶段，Map和Reduce

Map阶段并行处理输入数据；

Reduce阶段对Map结果进行归总；

2）shuffle连接Map和Reduce两个阶段

Map Task将数据写到本地磁盘；

Reduce Task从每个Map Task上读取一份数据；

3）仅适合离线批处理

具有很好的容错性和扩展性；

适合简单的批处理任务；

4）缺点明显

启动开销大、过多使用磁盘导致效率低下等。

4.2、WordCount实例

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class WC extends Configured implements Tool {

public static class MyMapper extends

Mapper<LongWritable,Text,Text,IntWritable>{

private Text mapOutputKey = new Text();

private IntWritable mapOutputValue = new IntWritable();

/\*\*

\* key 偏移量

\* value 一行字符串

\* context 上下文对象

\*/

@Override

protected void map(LongWritable key, Text value,

Mapper<LongWritable, Text, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

String val = value.toString();

String[] words = val.split(" ");

for (String word : words) {

mapOutputKey.set(word);

mapOutputValue.set(1);

// 通过上下文对象,最终传给我们的reduce

context.write(mapOutputKey, mapOutputValue);

}

}

}

public static class MyReducer extends

Reducer<Text,IntWritable,Text,IntWritable>{

private IntWritable reduceOutValue = new IntWritable();

/\*\*

\* key 输入的单词

\* values 存有单词出现的次数的集合

\* context 上下文对象

\*/

@Override

protected void reduce(Text key, Iterable<IntWritable> values,

Reducer<Text, IntWritable, Text, IntWritable>.Context

context) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

reduceOutValue.set(sum);

//把单词作为key,单词出现的次数作为值输出

context.write(key, reduceOutValue);

}

}

public int run(String[] args) throws Exception {

//1 创建job

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

job.setJarByClass(getClass());

//2 设置输入的路径

Path in = new Path(args[0]);

FileInputFormat.setInputPaths(job, in);

//3 设置输出的路径

Path out = new Path(args[1]);

FileOutputFormat.setOutputPath(job, out);

//4 设置map信息

job.setMapperClass(MyMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

//5设置reduce的信息

job.setReducerClass(MyReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

//6 提交任务

FileSystem fs = out.getFileSystem(conf);

if(fs.exists(out)) {

fs.delete(out, true);

}

boolean success = job.waitForCompletion(true);

return success?0:1;

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

//如果没有给参数,分析默认的

if(args==null|| args.length==0) {

args = new String[] {

"hdfs://hostname:8020/input/wordcount.txt",

"hdfs://hostname:8020/output"

};

}

int success = ToolRunner.run(conf, new WC(), args);

System.out.println(success);

}

}

4.3、mapreduce（map和reduce个数）

1）map的个数：

在map阶段读取数据前，FileInputFormat会将输入文件分割成split。 split的个数决定了map的个数。由任务切片spilt决定的，默认情况下一个split的大小就是block。

影响map个数，即split个数的因素主要有：   
 (1) HDFS块的大小，即HDFS中dfs.block.size的值。如果有一个输入文件为1024m，当块为256m时，会被划分为4个split；当块为128m时，会被划分为8个split。   
 (2) 文件的大小。当块为128m时，如果输入文件为128m，会被划分为1个split；当块为256m，会被划分为2个split。   
 (3) 文件的个数。FileInputFormat按照文件分割split，并且只会分割大文件，即那些大小超过HDFS块的大小的文件。如果HDFS中dfs.block.size设置为64m，而输入的目录中文件有100个，则划分后的split个数至少为100个。   
 (4) splitsize的大小。分片是按照splitszie的大小进行分割的，一个split的大小在没有设置的情况下，默认等于hdfs block的大小。但应用程序可以通过两个参数来对splitsize进行调节。

map个数的计算公式如下：   
 splitsize=max(minimumsize,min(maximumsize,blocksize))。如果没有 设置minimumsize和maximumsize，splitsize的大小默认等于blocksize。

**maxSize:由配置参数mapred.max.spilt.size确定,已经不考虑用户设定的maptask个数;  
 minSize:inputSplit的最小值,由配置参数mapred.min.spilt.size确定,默认值为1;**

**注：对于大文件，一般选择split=block，如果split<block，则会增加map执行的并发度，并且会造成在节点之间拉取数据。**

**对于小文件，默认一个文件启动一个map，这样会启动多个map，增大节点资源消耗，此时可以使用使用CombineFileInputFormat将多个小文件加入到一个split，并适当增大split的大小，这样会减少map启动的个数，减少并发度，减少资源消耗。**

解决方案：

1.在采集数据的预处理系统中将其合并成大文件，然后再上传。

2.使用Hadoop中的CombineFileInputFormat可以将多个小文件合并成大文件。

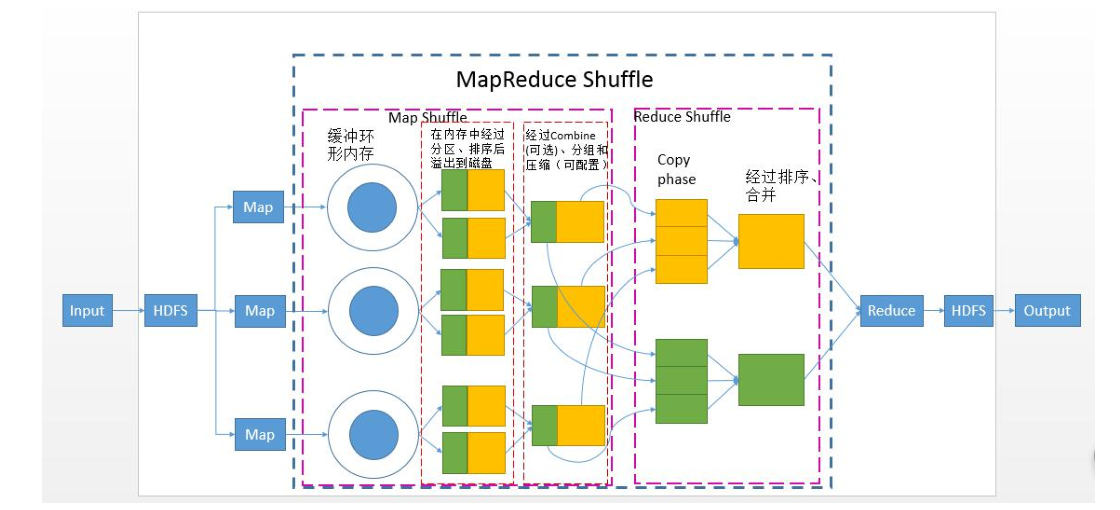
2）reduce个数：由分区个数决定。可以由用户在程序中Driver自定义job.setNumReduceTasks(3);一个ruduce对应一个结果文件partition。

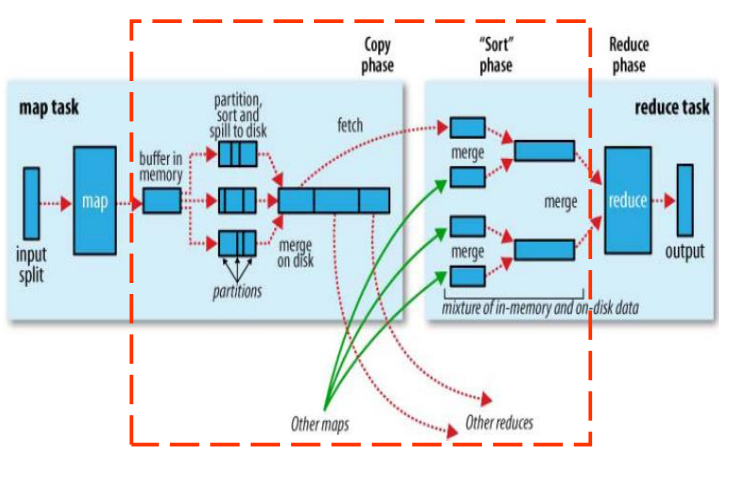
partiton（分区）：用来指定map输出的key交给哪个reuducer处理。默认是通过对map输出的key取hashcode 对指定的reduce个数求模（取余）

**(key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks = job.setNumReduceTasks(n)**

4.4、shuffle过程

从map()的输出到reduce()的输入，中间的过程被称为shuffle过程。





1）map side

(1) 在写入磁盘之前，会先写入环形缓冲区(circular memory buffer),默认100M(mapreduce.task.io.sort.mb可修改),当缓冲区内容达到80M

(mapreduce.map.sort.spill.percent可修改),缓冲区内容会被溢写到磁盘，形成一个spill file文件。

(2) 分区：在写入磁盘之前，会先进分区(partition),而partition的数量是由reducer的数量决定的。自定义设置分区个数：job.setNumReduceTasks(2)；默认是用map输出的<key,value>中key的hashcode对NumReduceTasks的个数取余，相同的分到一个区。

(3) 排序：在每一个partition中，都会有一个sort by key。

(4) combiner：**如果有combiner function，**在sort之后会执行combiner，相当于map阶段的redece。减少map输出的数据,减少了键值对；减少map输出的键值对磁盘的IO；减少reduce拉取的数据网络的IO。

(5) merge：数个spill files会合并(merge成一个分过区的排过序的文件）。

（可配置,依据经验,测试:正态分布图）减少网络IO,磁盘IO，消耗cpu等资源

mapreduce.task.io.sort.factor 一次性合并小文件的数量 默认10个

mapreduce.map.output.compress 启用压缩，默认是false

org.apache.hadoop.io.compress.DefaultCodec 默认使用的压缩算法

2）reduce side

(1) 解压缩:如果在map side 已经压缩过,在合并排序之前要先进行解压缩

(2) merger 合并：相同分区的数据合并在一起(map task 处理完成数据后,通知app master, 然后 app master 通过所有的reduce task,主动的拉去 map task的本地数据,拉取到自己分区中)

(3) 排序：对各个分区的数据进行排序

(4) 分组：将相同的key的value值,存入到一个集合中,形成一个 <key,[1,1,1,2]>，将key/values传递给reduce函数。

3）自定义分区 (词频统计,数字写在一个文件,非数字写在一个文件)

(1) 创建一个类,继承 Partitioner

(2) 设置numReduceTask

(3) 设置job.setPartitionClass()

public static class MyPartitioner extends Partitioner<Text, IntWritable> {

/\*\*

\* @return 数字 表示分区

\*/

@Override

public int getPartition(Text key, IntWritable value,int numPartitions) {

Pattern pattern = Pattern.compile("^[0-9]+$");

Matcher matcher = pattern.matcher(key.toString());

if(matcher.find()) {//数字

return 0;

}else { // 非数字

return 1;

}

}

}

job.setNumReduceTasks(2);

job.setPartitionerClass(MyPartitioner.class);

4.5、MapReduce运行流程

1) 首先对输入数据源进行切片

2) master调度worker执行map任务

3) worker读取输入源片段

4) worker执行map任务，将任务输出保存在本地

5) master调度worker执行reduce任务，reduce worker读取map任务的 输出文件

6) 执行reduce任务，将任务输出保存到HDFS

4.6、MapReducer优化

1.自定义分区：extends Partitioner 可以使用map输出的key或者value。

2.自定义分组：implements RawComparator 重点。WritableComparator.compareBytes，可以实现在字节数据层面上的比较，更加高效，分组的作用就是为了减少key value对。

3.启用Combiner： 减少reduce端拉取的的文件（key value对）数量。

4.shuffle启用压缩：map端压缩 reduce端解压缩 平衡集群的资源（CPU 内存）和网络IO。

5.CombinerFileInputFormat来合并小文件。

6.调整shffle环形缓冲区大小以及spill溢写的阈值。

7.自定义key和value，根据业务需求。

4.7、JobTracker和TaskTracker

1）概述：Hadoop MapReduce采用Master/Slave结构。

(1)Master：是整个集群的唯一的全局管理者，功能包括：作业管理、状态监控和任务调度等，即MapReduce中的JobTracker。

(2)Slave：负责任务的执行和任务状态的回报，即MapReduce中的TaskTracker。

2）JobTracker剖析：

(1)概述：JobTracker是一个后台服务进程，启动之后，会一直监听并接收来自各个TaskTracker发送的心跳信息，包括资源使用情况和任务运行情况等信息。

(2)JobTracker的主要功能：

a.作业控制：在hadoop中每个应用程序被表示成一个作业，每个作业又被分成多个任务，JobTracker的作业控制模块则负责作业的分解和状态监控。

**最重要的是状态监控**：主要包括TaskTracker状态监控、作业状态监控和任务状态监控。主要作用：容错和为任务调度提供决策依据。

b.资源管理。

3）TaskTracker剖析：

(1)TaskTracker概述：TaskTracker是JobTracker和Task之间的桥梁：一方面，从JobTracker接收并执行各种命令：运行任务、提交任务、杀死任务等；另一方面，将本地节点上各个任务的状态通过心跳周期性汇报给JobTracker。TaskTracker与JobTracker和Task之间采用了RPC协议进行通信。

(2)TaskTracker的功能：

a.汇报心跳：Tracker周期性将所有节点上各种信息通过心跳机制汇报给JobTracker。这些信息包括两部分：

\*机器级别信息：节点健康情况、资源使用情况等。

\*任务级别信息：任务执行进度、任务运行状态等。

b.执行命令：JobTracker会给TaskTracker下达各种命令，主要包括：启动任务(LaunchTaskAction)、提交任务(CommitTaskAction)、杀死任务(KillTaskAction)、杀死作业(KillJobAction)和重新初始化(TaskTrackerReinitAction)。

5、MapReduce Join

5.1、map Join

适用场景:一个大表、一个小表的情况 25MB

在map端，小表会被加载到处理大表的block的map方法所在的NodeManager的内存中，具体方式是重写Mapper类的setup()方法，将小表的数据从HDFS读出来，保存到某一个变量中，在map()方法中，就可以读取到小表的数据和大表的数据，然后先进行join，join的结果大大减小，这样，就可以减小网络和磁盘的IO，降低reduce()方法处理数据的压力。

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.URI;

import java.util.HashMap;

import java.util.Map;

import java.io.DataInput;

import java.io.DataOutput;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.FSDataInputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Writable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class MapJoin extends Configured implements Tool {

private static String cacheFile =

"hdfs://hostname:8020/input/customer.txt";

public static class MapJoinMapper extends

Mapper<LongWritable, Text, IntWritable, Text> {

private Map<Integer,CustomData> cacheMap =

new HashMap<Integer, CustomData>();

private IntWritable outPutKey = new IntWritable();

private Text outPutValue = new Text();

/\*\*

\*此方法在每个task开始之前执行，这里主要用作从DistributedCache

\* 中取到costomer文件，并将里边记录取出放到内存中。

\*/

@Override

protected void setup(Context context) throws Exception{

//获取当前作业的DistributedCache相关文件

//Path[] distributePaths = DistributedCache.

getLocalCacheFiles(context.getConfiguration());

FileSystem fs = FileSystem.get(URI.create(cacheFile),

context.getConfiguration());

FSDataInputStream fdis = fs.open(new Path(cacheFile));

BufferedReader br =

new BufferedReader(new InputStreamReader(fdis));

String lines = null;

String[] clos = null;

while ((lines=br.readLine())!=null){

clos = lines.split(" ");

if (clos.length<3){

continue;

}

CustomData customData =

new CustomData(Integer.parseInt(clos[0]),clos[1],clos[2]);

System.err.println(customData.getCustomID());

cacheMap.put(customData.getCustomID(),customData);

}

}

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] strs = value.toString().split(" ");

//格式：cust\_id order\_id prices order\_date

if (strs.length<4){

return;

}

int joinID = Integer.valueOf(strs[0]);

System.err.println(joinID);

CustomData customer = cacheMap.get(joinID);

//如果内存中没有的用户信息没有可以join的数据，则过滤

if (customer == null){

return;

}

//cust\_id order\_id prices order\_date cust\_name phone

StringBuffer sb = new StringBuffer();

sb.append(strs[1]).append("\t")

.append(strs[2]).append("\t")

.append(strs[3]).append("\t")

.append(customer.getName()).append("\t")

.append(customer.getPhone());

outPutKey.set(joinID);

outPutValue.set(sb.toString());

context.write(outPutKey,outPutValue);

}

}

public static class MapJoinReducer extends

Reducer<IntWritable,Text,IntWritable,Text>{

@Override

protected void reduce(IntWritable key, Iterable<Text> values,

Context context) throws IOException, InterruptedException {

for (Text value: values){

System.out.println("hdfadfs");

context.write(key,value);

}

}

}

public int run(String[] args) throws Exception {

//1.创建job

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

job.setJarByClass(getClass());

//2 设置输入的路径

Path in = new Path(args[0]);

job.addCacheFile(URI.create(cacheFile));

FileInputFormat.addInputPath(job,in);

//3 设置输出的路径

Path outPath = new Path(args[1]);

FileOutputFormat.setOutputPath(job, outPath);

//4 设置map信息

job.setMapperClass(MapJoinMapper.class);

job.setMapOutputKeyClass(IntWritable.class);

job.setMapOutputValueClass(Text.class);

//5设置reduce的信息

job.setOutputKeyClass(IntWritable.class);

job.setOutputValueClass(Text.class);

job.setNumReduceTasks(0);

//6 提交

FileSystem fs = outPath.getFileSystem(conf);

if (fs.exists(outPath)) {

fs.delete(outPath, true);

}

boolean isSuccess = job.waitForCompletion(true);

return isSuccess ? 0 : 1;

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

if( args==null|| args.length==0) {

args= new String[] {

"hdfs://hostname:8020/input/order.txt",

"hdfs://hostname:8020/output"

};

}

int status = ToolRunner.run(conf, new MapJoin(), args);

System.exit(status);

}

public static class CustomerData {

public static class CustomData implements Writable {

private int customID;

private String name;

private String phone;

public CustomData() {

}

public CustomData(int customID, String name, String phone) {

this.customID = customID;

this.name = name;

this.phone = phone;

}

public void set(int customID, String name, String phone) {

this.customID = customID;

this.name = name;

this.phone = phone;

}

public int getCustomID() {

return customID;

}

public void setCustomID(int customID) {

this.customID = customID;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPhone() {

return phone;

}

public void setPhone(String phone) {

this.phone = phone;

}

public void write(DataOutput out) throws IOException {

}

public void readFields(DataInput in) throws IOException {

}

@Override

public String toString() {

return customID + "," + name + "," + phone;

}

}

}

}

Order.txt

1001 1111 1000.0 20180101123001

1002 1112 2999.0 20180101123001

1001 1113 10.0 20180103123001

1002 1114 7888.0 20180121143801

1003 1115 100.0 20180224082709

1004 1116 99.0 20180224144610

customer.txt

1001 jack 18717771844

1002 rose 17166666666

1003 tom 13987654321

1004 lily 17512345678

5.2、reduce Join

适用场景:两个大表的情况

两张表通常都是大文件，Join的操作是在Reduce端执行，由于文件过大，载入内存是不现实的，可以在发送到reduce之后再进行join。

每次在map端得到数据之后，就打一个标签将标签和数据一起发送到reduce，在reduce端根据标签和具体的业务需求完成join。

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

/\*\*

\* 在Reudce端进行连接是MapReduce框架进行表之间join操作最为常见的模式，

\* 其具体的实现原理如下：

\* Map端的主要工作：为来自不同表（文件）的key/value对打标签以区别不同来源的记录。

\* 然后用连接字段作为key，其余部分和新加的标志作为value，最后进行输出。

\* reduce端的主要工作：在reduce端以连接字段作为key的分组已经完成，

\* 我们只需要在每一个分组当中将那些来源于不同文件的记录（在map阶段已经打标志）分开，

\* 最后进行笛卡尔只就ok了。

\*/

public class ReduceJoin extends Configured implements Tool {

public static class ReducerJoinMapper extends

Mapper<LongWritable, Text, Text, Text> {

private Text mapOutKey = new Text();

private Text mapOutValue = new Text();

@Override

protected void map(LongWritable key, Text value,

Context context) throws IOException, InterruptedException {

// 获取输入文件路径

String pathName =

((FileSplit)context.getInputSplit()).getPath().toString();

String val = null;

String line = value.toString();

String[] strs = line.split(" ");

//如果是customer信息

if (pathName.contains("customer")) {

val = strs[1]+"\t"+strs[2]+"\t0";

//如果是order

} else if (pathName.contains("order")) {

val = strs[1]+"\t"+strs[2]+"\t"+"\t"+strs[3]+"\t"+"1";

}

//设置key value

mapOutKey.set(strs[0]);

mapOutValue.set(val);

context.write(mapOutKey, mapOutValue);

}

}

public static class ReducerJoinReducer extends

Reducer<Text, Text, Text, Text> {

private List<String> leftTable = new ArrayList<String>();

private List<String> rightTable = new ArrayList<String>();

private Text outValue = new Text();

@Override

protected void reduce(Text key, Iterable<Text> values,

Context context) throws IOException, InterruptedException {

leftTable.clear();

rightTable.clear();

//遍历集合

for (Text text : values) {

//1取出标记

String val = text.toString();

String[] vals = val.split("\t");

String mark = vals[vals.length-1];

//分别添加到不同的集合

if (mark.equals("0")) {

leftTable.add(val);

} else if (mark.equals("1")) {

rightTable.add(val);

}

}

//积

for (String lt : leftTable) {

for (String rt : rightTable) {

outValue.set(lt + "\t" + rt);

context.write(key, outValue);

}

}

}

}

public int run(String[] args) throws Exception {

// 1 生成对应的job

Configuration config = super.getConf();

Job job = Job.getInstance(config, this.getClass().getSimpleName());

job.setJarByClass(getClass());

// 2 输入

Path in1 = new Path(args[0]);

FileInputFormat.setInputPaths(job, in1);

// 3 输出

Path outPath = new Path(args[1]);

FileOutputFormat.setOutputPath(job, outPath);

// 4 map

job.setMapperClass(ReducerJoinMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

/\*\*

\* shuffle 过程 按照map输出的key进行分区和分组 默认

\* PairWritable 自定义分区 new

\* PairWritable().getName() 自定义分组 new PairWritable().getName()

\*/

// 5 reducer

job.setReducerClass(ReducerJoinReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

// 6 提交

FileSystem fs = outPath.getFileSystem(config);

if (fs.exists(outPath)) {

fs.delete(outPath, true);

}

boolean isSuccess = job.waitForCompletion(true);

return isSuccess ? 0 : 1;

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

if (args == null || args.length == 0) {

args = new String[] {

"hdfs://hostname:8020/input",

"hdfs://hostname:8020/output"

};

}

int status = ToolRunner.run(conf, new ReduceJoin(), args);

System.out.println(status);

}

}

6、hadoop2.x与Hadoop1.x区别

（1）2.x比1.x从性能上有很大增强

（2）Hadoop1中的JobTracker是一个功能集中的部分，负责资源的分配和任务的分配，所以JobTracker单点出问题就会造成整个集群无法使用了，而且MapReduce模式是集成在Hadoop1中，不易分解;Hadoop2中，ResourceManager(RM)就是负责资源的分配，NodeManager(NM)是从节点上管理资源的，而ApplicationMaster(AM)就是一个负责任务分配的组件，根据不同的模式有不同的AM，因此MapReduce模式有自己独有的AM;

（3）1.x版本没有Namenode Federation功能，只能有一个NameNode；而2.x添加了，还可以有多个NameNode同时运行，每个负责集群中的一部分

（4）1.x版本没有Namenode HA功能，只能有一个NameNode；而2.x添加了，解决了namenode单点故障问题。

（5）解决对MapReduce之外的框架支持问题。