

大数据与hadoop

- 大数据部门业务流程
- 大数据部门组织架构
- Hadoop和它的版本
- Hadoop的优势
- Hadoop的组成

- MapReduce架构概述
- HDFS架构概述
- YARN架构概述

- 大数据技术生态体系

环境搭建

- Hadoop运行环境搭建
- Hadoop目录结构
- Hadoop运行模式

- 本地模式

- 例1：官方Grep案例
- 例2：官方WordCount案例

- 伪分布式模式

- 准备工作
- 配置集群
- 启动HDFS并运行MR程序
- 为什么不能一直格式化NameNode？
- 启动yarn并运行MR程序

- 配置历史服务器

- 配置日志聚集

- 配置文件说明

- 完全分布式模式

- 准备工作
- 集群分发脚本
- 配置集群
- 集群单点启动
- 集群SSH配置
- 群起集群
- 集群启动/停止方式总结
- 集群时间同步

HDFS操作

- HDFS概述
- HDFS产生背景
- HDFS定义与使用场景
- HDFS优缺点

- 优点

- 缺点

- HDFS组成架构

- HDFS文件块大小

- HDFS的shell操作

- HDFS客户端环境搭建

- HDFS的API操作

- 文件上传

- 文件下载

- 文件删除

- 更改文件名

- 文件详情查看

- 文件和文件夹判断

- HDFS的IO流操作

- 文件上传

文件下载

定位文件读取

HDFS机理

向HDFS读写数据的过程

向HDFS写数据的过程

网络拓扑-节点距离计算

机架感知(副本存储节点选择)

读HDFS数据的过程

NameNode和SecondaryNameNode

NN和2NN工作机制

Fsimage和Edits解析

checkpoint时间设置

NameNode故障处理

集群安全模式

NameNode多目录配置

DataNode

DataNode工作机制

数据完整性

掉线时限参数设置

服役新数据节点

添加白名单

添加黑名单

datanode多目录配置

HDFS 2.X新特性

集群间数据拷贝distcp

小文件归档

回收站

快照管理

MapReduce

MapReduce概述

MapReduce定义

MapReduce优缺点

MapReduce的核心思想

MapReduce三类实例进程

数据序列化类型

MapReduce编程规范

Wordcount案例运行

WordcountMapper类

WordcountReducer类

Driver驱动类：WordcountDriver

在集群上测试

Hadoop序列化

序列化概论

自定义bean对象Writable

序列化案例

需求分析

bean对象

Mapper类

Reducer类

Driver驱动类

InputFormat数据输入

数据切片的概念

Job提交流程

FileInputFormat切片过程

FileInputFormat切片参数设置

CombineTextInputFormat切片机制

CombineTextInputFormat案例

FileInputFormat实现类

- TextInputFormat
- KeyValueTextInputFormat
- NLineInputFormat
- 自定义InputFormat
 - 需求分析
 - 自定义InputFormat类
 - RecordReader类：
 - SequenceFileMapper类
 - SequenceFileReducer类
 - SequenceFileDriver类
- shuffle
 - partition分区
 - 排序概述
 - 全排序
 - 需求分析
 - FlowBean类
 - Mapper类
 - Reducer类
 - Driver类
 - 区内排序
 - Combiner合并
 - 需求分析
 - WordcountCombiner类
 - 辅助排序和二次排序
 - 需求分析
 - OrderBean类
 - OrderSortMapper类
 - OrderSortGroupingComparator类
 - OrderSortReducer类
 - OrderSortDriver类
- MapTask和ReduceTask
 - maptask工作机制
 - reducetask工作机制
- OutputFormat数据输出
 - OutputFormat接口实现类
 - 自定义OutputFormat案例
 - 需求分析
 - FilterMapper类
 - FilterReducer类
 - 自定义OutputFormat类
 - FilterDriver类
- Join
 - Reduce Join
 - 案例分析
 - TableBean类
 - TableMapper类
 - TableReducer类
 - TableDriver类
 - Map Join
 - 驱动类
 - DistributedCacheMapper类
- 计数器
- 数据清洗
 - 需求分析
 - LogMapper类
 - LogDriver类
- 数据压缩
 - 压缩方式

- Gzip压缩
- Bzip2压缩
- Lzo压缩
- Snappy压缩
- 压缩位置选择
- 压缩参数设置
 - 输入压缩阶段参数
 - mapper输出端压缩参数
 - reducer输出段压缩参数
- 压缩案例
 - 数据流的压缩和解压缩
 - map输出端压缩
 - reduce输出端压缩

Yarn

- Yarn资源调度器
- Yarn基本架构
- Yarn工作流程
- 资源调度器
 - 先进先出调度器 (FIFO)
 - 容量调度器 (Capacity Scheduler)
 - 公平调度器 (Fair Scheduler)
- 推测执行
 - 推测执行算法原理

优化

- hadoop企业优化
- 速度低的原因
- MapReduce优化方法
 - 数据输入
 - map阶段
 - reduce阶段
 - IO传输
 - 数据倾斜问题
- 常用的调优参数
 - MR相关参数 (mapred-default.xml)
 - yarn相关参数 (yarn-default.xml)
 - Shuffle性能优化 (mapred-default.xml)
 - 容错相关参数 (mapred-default.xml)
- 小文件问题

案例

- 扩展案例
 - 倒排索引案例 (多job串联)
 - 需求分析
 - job1的OneIndexMapper类
 - job1的OneIndexReducer类
 - job1的OneIndexDriver类
 - job2的TwoIndexMapper类
 - job2的TwoIndexReducer类
 - job2的TwoIndexDriver类
 - TopN案例 (巧用treemap)
 - 需求分析
 - FlowBean类
 - TopNMapper类
 - TopNReducer类
 - TopNDriver类
 - 共同好友案例
 - 需求分析
 - OneShareFriendsMapper类
 - OneShareFriendsReducer类

OneShareFriendsDriver类
TwoShareFriendsMapper类
TwoShareFriendsReducer类
TwoShareFriendsDriver类

hadoop HA

高可用HA概述
HDFS手动故障转移
环境配置
启动HDFS-HA集群
HDFS自动故障处理
工作要点
工作机制
环境配置
启动HDFS-HA集群
YARN-HA配置
环境配置
启动集群
HDFS Federation架构设计

大数据与hadoop

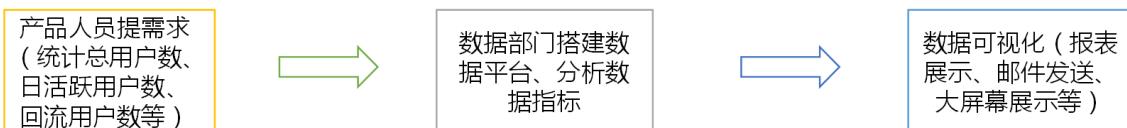
大数据技术主要解决海量数据的存储和计算问题。

数据存储单位（从小到大）：bit、Byte、KB、MB、GB、TB、PB、EB、ZB。

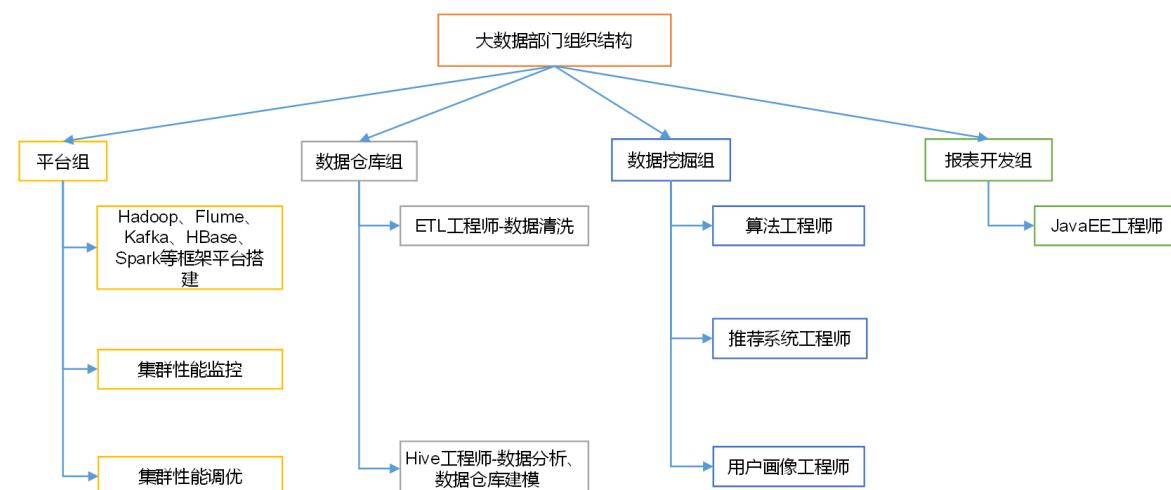
大数据的特点：量大、产生速度快、数据多样性、低价值密度（数据总量越高，价值密度越低）

常见大数据应用场景：仓库选址分析、零售商品分区分析、旅游运营优化、商品广告推荐、风险评估、推荐优质客户、房地产营销运营、机器人、自动驾驶。

大数据部门业务流程



大数据部门组织架构



Hadoop和它的版本

Hadoop是一个由Apache基金会所开发的分布式系统基础框架。Hadoop有一个更广泛的含义，就是hadoop生态圈。

Hadoop三大发行版本：Apache、Cloudera、Hortonworks。

Apache Hadoop：

官网地址：<http://hadoop.apache.org/releases.html>

下载地址：<https://archive.apache.org/dist/hadoop/common/>

Cloudera Hadoop：

官网地址：<https://www.cloudera.com/downloads/cdh/5-10-0.html>

下载地址：<http://archive-primary.cloudera.com/cdh5/cdh/5/>

Hortonworks Hadoop

官网地址：<https://hortonworks.com/products/data-center/hdp/>

下载地址：<https://hortonworks.com/downloads/#data-platform>

Apache版本最原始（最基础）的版本，对于入门学习最好。

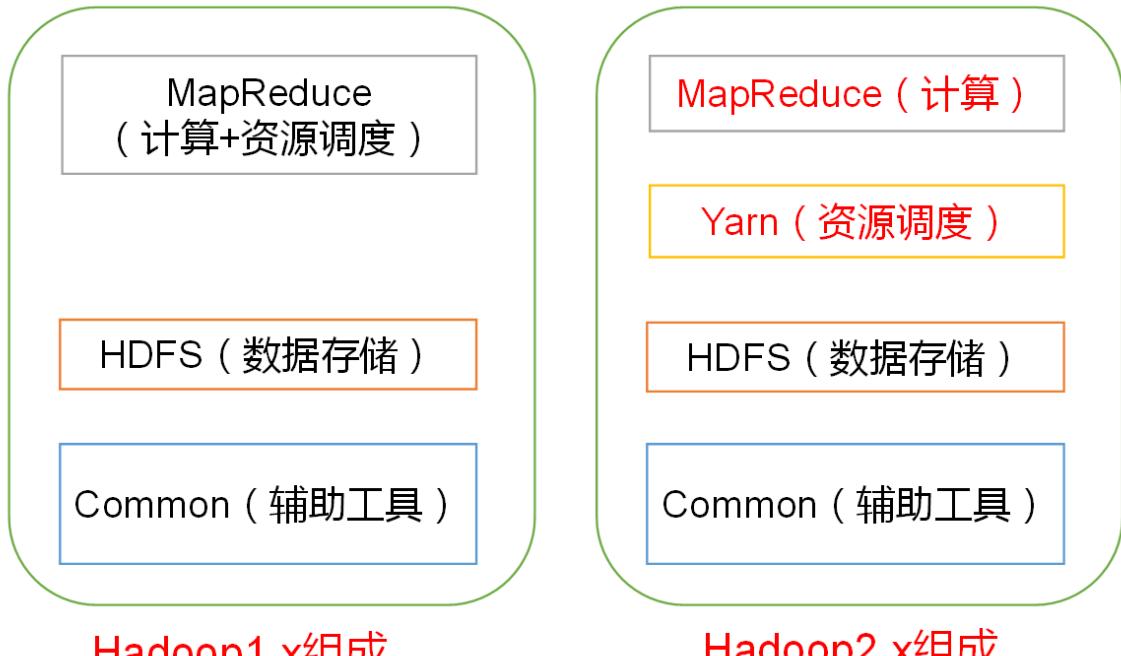
Cloudera版本（CDH版）在大型企业中应用的多，使用免费，维护收费，它搭建起来更方便。

Hadoop的优势

- 1、高可靠性：底层维护多个数据副本，即使某个计算元素或存储出现故障，也不会导致数据的丢失。
- 2、高扩展性：在集群间分配任务数据，方便的扩展数以千计的节点。
- 3、高效性：在MapReduce的思想下，Hadoop是并行工作的，处理任务的速度快。
- 4、高容错性：能够自动将失败的任务重新分配。

Hadoop的组成

Hadoop1.x和Hadoop2.x的组成不同：

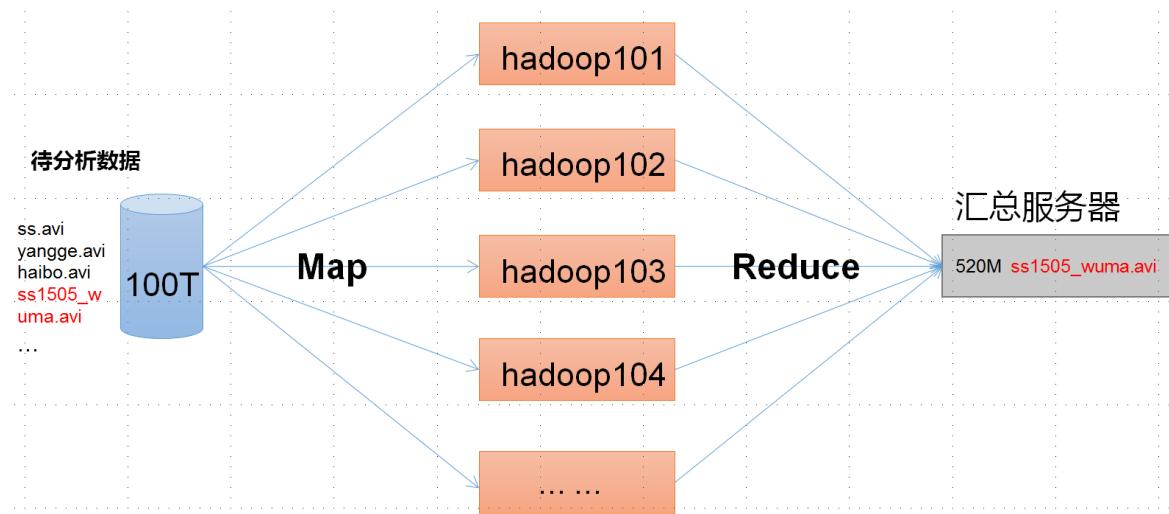


在Hadoop1.x中，MapReduce同时处理计算和资源调度，耦合性较大，在Hadoop2.x中，Yarn负责处理资源调度，而MapReduce只负责计算，降低了耦合性。

MapReduce架构概述

它将计算过程分为两个阶段：Map和Reduce

- 1) Map阶段并行处理输入数据
- 2) Reduce阶段对Map结果进行汇总



HDFS架构概述

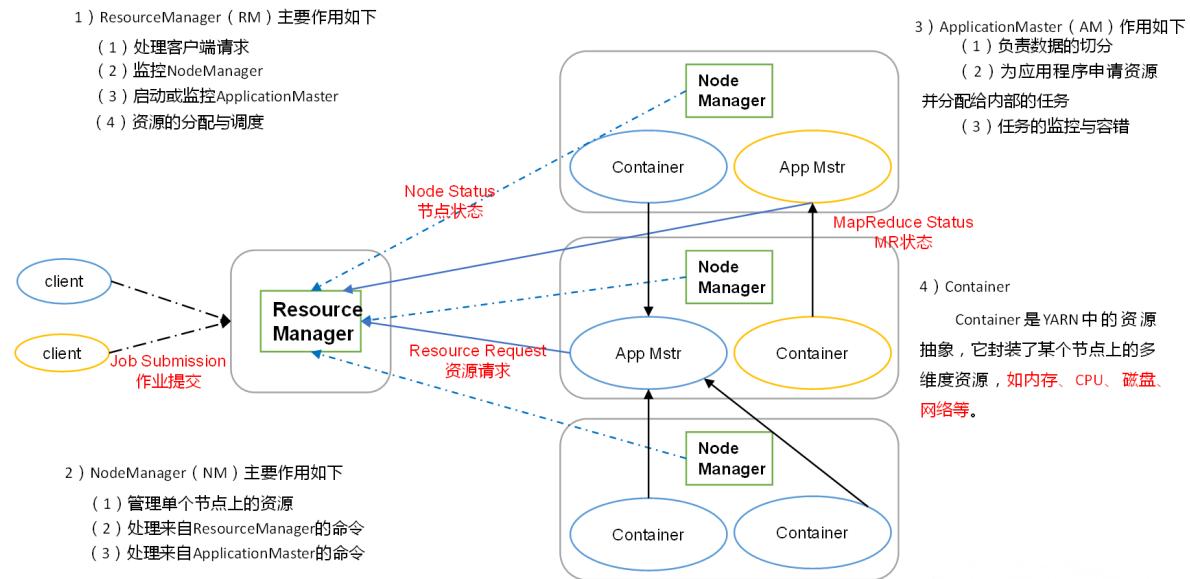
全称为Hadoop Distributed File System，用于文件存储。内部有三种node：

- 1) NameNode (nn) : 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和块所在的DataNode等。它相当于目录。
- 2) DataNode(dn) : 在本地文件系统存储文件块数据，以及块数据的校验和。它相当于文件的数据本身。
- 3) Secondary NameNode(2nn) : 用来监控HDFS状态的辅助后台程序，每隔一段时间获取HDFS元数据的快照。它的作用是辅助处理。

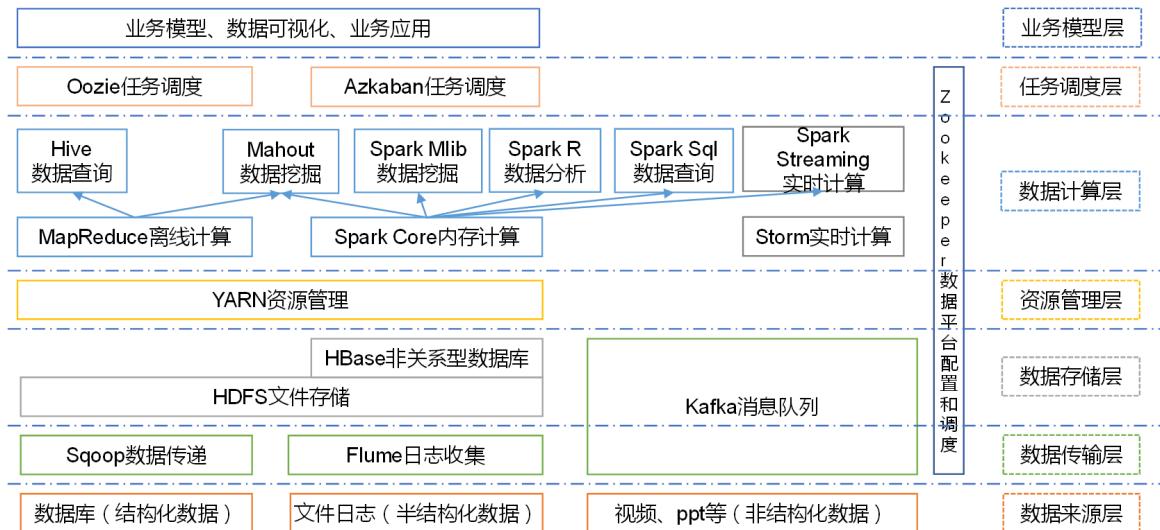
YARN架构概述

ResourceManager负责接收客户端的请求，负责资源的总调度。

NodeManager负责管理单个节点上的资源。



大数据技术生态体系



图中涉及的技术名词解释如下：

1) Sqoop : Sqoop是一款开源的工具，主要用于在Hadoop、Hive与传统的数据库(MySql)间进行数据的传递，可以将一个关系型数据库（例如：MySQL，Oracle等）中的数据导进到Hadoop的HDFS中，也可以将HDFS的数据导进到关系型数据库中。

2) Flume : Flume是Cloudera提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。

3) Kafka : Kafka是一种高吞吐量的分布式发布订阅消息系统，有如下特性：

(1) 通过O(1)的磁盘数据结构提供消息的持久化，这种结构对于即使数以TB的消息存储也能够保持长时间的稳定性能。

(2) 高吞吐量：即使是非常普通的硬件Kafka也可以支持每秒数百万的消息。

(3) 支持通过Kafka服务器和消费机集群来分区消息。

(4) 支持Hadoop并行数据加载。

4) Storm : Storm用于“连续计算”，对数据流做连续查询，在计算时就将结果以流的形式输出给用户。

5) Spark : Spark是当前最流行的开源大数据内存计算框架。可以基于Hadoop上存储的大数据进行计算。

6) Oozie : Oozie是一个管理Hadoop作业(job)的工作流程调度管理系统。

7) Hbase : HBase是一个分布式的、面向列的开源数据库。HBase不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。

8) Hive : Hive是基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的SQL查询功能，可以将SQL语句转换为MapReduce任务进行运行。其优点是学习成本低，可以通过类SQL语句快速实现简单的MapReduce统计，不必开发专门的MapReduce应用，十分适合数据仓库的统计分析。

10) R语言 : R是用于统计分析、绘图的语言和操作环境。R是属于GNU系统的一个自由、免费、源代码开放的软件，它是一个用于统计计算和统计制图的优秀工具。

11) Mahout : Apache Mahout是个可扩展的机器学习和数据挖掘库。

12) ZooKeeper : Zookeeper是Google的Chubby一个开源的实现。它是一个针对大型分布式系统的可靠协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。ZooKeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

环境搭建

Hadoop运行环境搭建

准备一个磁盘空间20G，内存2G，2核的虚拟机。

1、修改配置文件 `vim /etc/udev/rules.d/70-persistent-net.rules` :

```
# This file was automatically generated by the /lib/udev/write_net_rules
# program, run by the persistent-net-generator.rules rules file.
#
# You can modify it, as long as you keep each rule on a single
# line, and change only the value of the NAME= key.

# PCI device 0x1022:0x2000 (pcnet32)
# PCI device 0x1022:0x2000 (vmxnet)
# PCI device 0x1022:0x2000 (vmxnet)

# PCI device 0x1022:0x2000 (vmxnet)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:0c:29:c0:4e:e7", ATTR{type}=="1", KERNEL=="eth*", NAME="eth0"
~
```

改为eth0，删除多余配置，将这里的ATTR复制，要求配置文件/etc/sysconfig/network-scripts/ifcfg-eth0中的mac地址与这个一致（当虚拟机是克隆的需要核对这一步，如果虚拟机是新建安装的一般不会出现mac地址不一致的情况，配置文件中也无需修改）

2、然后配置静态IP，修改主机名（主机名最好能区分IP地址的网络号），设置从域名到IP地址的映射，需要修改/etc/hosts文件：（这里必须把本机的映射加进去，否则后面示例会报错）

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.1.100 hadoop100
192.168.1.101 hadoop101
192.168.1.102 hadoop102
192.168.1.103 hadoop103
192.168.1.104 hadoop104
192.168.1.105 hadoop105
192.168.1.106 hadoop106
192.168.1.107 hadoop107
192.168.1.108 hadoop108
```

这样就可以在局域网内通过域名访问对方了。（如果主机能ping通虚拟机，但虚拟机ping不通windows，这是windows防火墙的原因，只要在防火墙的高级设置-入站规则文件和打印共享（回显请求 - ICMPv4-In），注意是公用的，然后打开即可）

3、然后创建一个普通用户yinyunmoyi，给它赋予root权限，修改文件/etc/sudoers加入一行：

```
yinyunmoyi ALL=(ALL) ALL
```

4、安装JDK

首先要卸载原来的jdk，查看当前jdk：`rpm -qa | grep java`

然后卸载jdk：`rpm -e --nodeps jdk名`

在/opt目录下创建module、software文件夹，前者是编译完之后的文件，后者是源码压缩包存放位置。

将源码包放入software中，解压源码包，解压位置放在module中：

```
tar -zxvf jdk-8u144-linux-x64.tar.gz -C /opt/module/
```

此时在module下就会出现java的文件夹，它的绝对路径是：`/opt/module/jdk1.8.0_144`

在/etc/profile文件末尾配置环境变量：

```
#JAVA_HOME
export JAVA_HOME=/opt/module/jdk1.8.0_144
export PATH="$PATH:$JAVA_HOME/bin"
```

然后执行`source /etc/profile`让配置文件生效。

最后就可以执行`java -version`来查看java版本，可以查到表示安装成功，如果不可以重启。

5、安装hadoop

首先将源码包放入software中，然后解压到module中：

```
tar -zxvf hadoop-2.7.2.tar.gz -C /opt/module/
```

在/etc/profile文件末尾配置环境变量：

```
##HADOOP_HOME
export HADOOP_HOME=/opt/module/hadoop-2.7.2
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
```

执行`source /etc/profile`让配置文件生效。

最后可以执行hadoop命令查看是否安装完成。

Hadoop目录结构

其中重要的有：

- (1) bin目录：存放对Hadoop相关服务 (HDFS,YARN) 进行操作的脚本
- (2) etc目录：Hadoop的配置文件目录，存放Hadoop的配置文件
- (3) lib目录：存放Hadoop的本地库 (对数据进行压缩解压缩功能)
- (4) sbin目录：存放启动或停止Hadoop相关服务的脚本
- (5) share目录：存放Hadoop的依赖jar包、文档、和官方案例
- (6) logs目录：存放hadoop-atguigu-datanode-hadoop.atguigu.com.log和hadoop-atguigu-namenode-hadoop.atguigu.com.log，可以查看日志。

Hadoop运行模式

Hadoop运行模式包括：本地模式、伪分布式模式以及完全分布式模式。前两种一般测试用，后一种是实际的开发环境。

本地模式

例1：官方Grep案例

- 1、创建在hadoop-2.7.2文件下面创建一个input文件夹：`mkdir input`
- 2、将Hadoop的xml配置文件复制到input：`cp etc/hadoop/*.xml input`
- 3、执行share目录下的MapReduce程序：

```
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar grep  
input output 'dfs[a-z.]+'
```

- 4、执行结束后hadoop目录中就会出现一个output文件夹：

```
[root@hadoop130 output]# ll  
总用量 4  
-rw-r--r--. 1 root root 11 1月 30 07:28 part-r-00000  
-rw-r--r--. 1 root root 0 1月 30 07:28 _SUCCESS
```

其中_SUCCESS是程序运行的成果标志，另一个文件是运行结果：

```
[root@hadoop130 output]# cat part-r-00000  
1         dfsadmin
```

这个程序是用来统计目标文件夹中符合正则表达式的词及相应个数的，执行语句的组成一般是hadoop开头，然后跟执行的jar包，每个命令都有输入值和输出值（注意输出的目录不能存在，否则会报错），正则表达式是命令的参数。

例2：官方WordCount案例

- 1、创建在hadoop-2.7.2文件下面创建一个wcinput文件夹：`mkdir wcinput`
- 2、在wcinput文件下创建一个wc.input文件：`cd wcinput`、`touch wc.input`
- 3、编辑wc.input文件，输入以下内容：

```
hadoop yarn  
hadoop mapreduce  
atguigu  
atguigu
```

4、执行程序：

```
hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount  
wcinput wcoutput
```

5、hadoop目录中出现一个wcoutput目录，其中有成功标记和运行结果：

```
[root@hadoop130 hadoop-2.7.2]# cd wcoutput/  
[root@hadoop130 wcoutput]# ll  
总用量 4  
-rw-r--r--. 1 root root 38 1月 30 07:39 part-r-00000  
-rw-r--r--. 1 root root 0 1月 30 07:39 _SUCCESS
```

运行结果：

```
[root@hadoop130 wcoutput]# cat part-r-00000  
atguigu 2  
hadoop 2  
mapreduce 1  
yarn 1
```

这个程序是用来统计目标文件夹中的词频的，命令格式和之前一样，hadoop开头，跟jar包，有输入和输出文件。

伪分布式模式

准备工作

1、Windows下主机名和IP映射设置

在c:/windows/system32/drivers/etc找到hosts文件，添加域名和IP的对应关系。

2、关闭selinux：`vim /etc/selinux/config`

改为`SELINUX=disabled`。

3、关闭linux的防火墙：`service iptables stop`

配置集群

1、修改hadoop-env.sh：`vim etc/hadoop/hadoop-env.sh`，将JAVAHOME改为具体的路径：

```
export JAVA_HOME=/opt/module/jdk1.8.0_144
```

2、修改core-site.xml：`vim etc/hadoop/core-site.xml`，将以下代码放入标签中：

```

<!-- 指定HDFS中NameNode的地址 -->
<property>
<name>fs.defaultFS</name>
<value>hdfs://hadoop130:9000</value>
</property>

<!-- 指定Hadoop运行时产生文件的存储目录 -->
<property>
<name>hadoop.tmp.dir</name>
<value>/opt/module/hadoop-2.7.2/data/tmp</value>
</property>

```

第一个标签指定的是NameNode的地址，默认是file://，代表本地模式，现在要使用分布模式必须改为hdfs://。后面的是主机名和端口号。

第二个标签指定的是文件的存储目录。

3、修改hdfs-site.xml：`vim etc/hdfs-site.xml`，将以下代码放入标签中：

```

<!-- 指定HDFS副本的数量 -->
<property>
<name>dfs.replication</name>
<value>1</value>
</property>

```

HDFS副本的默认数量是3，这里改为1。

启动HDFS并运行MR程序

1、格式化NameNode（第一次启动时格式化，千万不要重复格式化，否则后面会出错，只能重启）：

`bin/hdfs namenode -format`

2、启动NameNode：`sbin/hadoop-daemon.sh start namenode`

3、启动DataNode：`sbin/hadoop-daemon.sh start datanode`

启动完成后用jps命令查看是否开启namenode和datanode。然后web端就可以访问hadoop130:50070

在HDFS中可以创建目录、查看文件、上传或下载文件：

创建浏览目录：`bin/hdfs dfs -mkdir -p /user/yinyun/input`，创建后就能在HDFS文件系统中的Browse Directory中查看：

/								Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
drwxr-xr-x	root	supergroup	0 B	2020/1/31 上午1:08:51	0	0 B	user	

查看HDFS根目录下有什么文件，这些命令和linux中的执行方式一样：`bin/hdfs dfs -ls /`

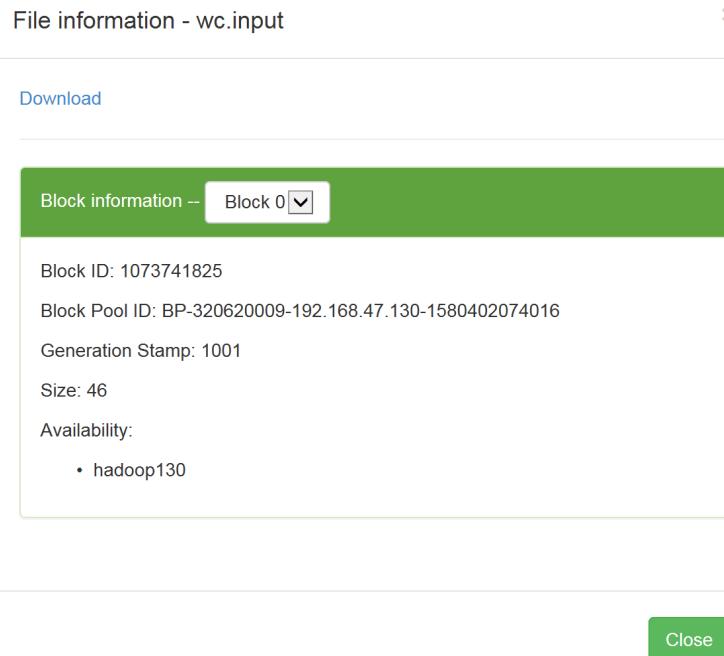
把本地文件上传到HDFS中：`bin/hdfs dfs -put wcinput/wc.input /user/yinyun/input`：

Browse Directory

/user/yinyun/input								Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
-rw-r--r--	root	supergroup	46 B	2020年1月31日 1:59:33	1	128 MB	wc.input	

Replication的意思是备份值是1，BlockSize的值是128MB，它是一个块存储的上限值，点击Name会出现块信息，如果文件很大则用多个块，还可以在这个页面下载文件（还可以在linux中查看文件）：

```
bin/hdfs dfs -cat /user/yinyun/.... ) :
```



执行命令并将结果放入HDFS中：

```
hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount  
/user/yinyun/input /user/yinyun/output
```

HDFS内的文件路径：/opt/module/hadoop-2.7.2/data/tmp/dfs/data/current/BP-938951106-192.168.10.107-1495462844069/current/finalized/subdir0/subdir0

为什么不能一直格式化NameNode？

在data/tmp/dfs/name/current/VERSION中记录着clusterID，在data/tmp/dfs/data/current/VERSION页记录着一个clusterID，当启动NameNode再启动DataNode后，会给两者分配相同的clusterID，但如果此时格式化NameNode就会给它分配新的clusterID，两者clusterID不一致就会报错。

格式NameNode时，一定要先停止两个node进程，然后再删除data数据和log日志，然后再格式化NameNode。

启动yarn并运行MR程序

首先需要配置/etc/hadoop/下的几个配置文件：

1、配置yarn-env.sh，去掉注释配置具体的JAVAHOME：

```
export JAVA_HOME=/opt/module/jdk1.8.0_144
```

2、配置yarn-site.xml：

```
<!-- Reducer获取数据的方式 -->
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>

<!-- 指定YARN的ResourceManager的地址 -->
<property>
<name>yarn.resourcemanager.hostname</name>
<value>hadoop130</value>
</property>
```

指定reducer获取数据的方式是shuffle，指定resourceManager放在哪一个服务器，注意主机名要改成自己的。

3、配置：mapred-env.sh：

```
export JAVA_HOME=/opt/module/jdk1.8.0_144
```

4、对mapred-site.xml.template重新命名为mapred-site.xml：

```
mv mapred-site.xml.template mapred-site.xml
```

并加入：

```
<!-- 指定MR运行在YARN上 -->
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

默认MR运行在local，这里要修改为运行在yarn。

5、保证NameNode和DataNode已经启动，然后启动ResourceManager：

```
sbin/yarn-daemon.sh start resourcemanager
```

最后启动NodeManager：

```
sbin/yarn-daemon.sh start nodemanager
```

最后可以用jps命令查看这四个是否已经启动，此时已经可以访问<http://hadoop130:8088/cluster>，这是任务执行进度查看的页面。

要运行MR程序首先要删除output目录：

```
bin/hdfs dfs -rm -R /user/yinyun/output
```

执行MR程序：

```
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount /user/yinyun/input /user/yinyun/output
```

，在8088端口处可以查看程序运行：



All Applications

Logged in as: dr.who

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
2	0	1	1	2	3 GB	8 GB	0 B	2	8	0	1	0	0	0	0

Scheduler Metrics

Scheduler Type		Scheduling Resource Type		Minimum Allocation		Maximum Allocation	
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>		<memory:8192, vCores:8>			

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes
application_1580427339159_0002	root	word count	MAPREDUCE	default	Fri Jan 31 17:26:38 +0800 2020	N/A	RUNNING	UNDEFINED		ApplicationMaster	0
application_1580427339159_0001	root	word count	MAPREDUCE	default	Fri Jan 31 17:15:10 +0800 2020	Fri Jan 31 17:27:31 +0800 2020	FINISHED	SUCCEEDED		History	N/A

Showing 1 to 2 of 2 entries First Previous 1 Next Last

红框处从左到右分别是运行程序的ID，程序的使用者，程序名，类型，队列，开始和结束时间，状态，最后状态，进度。

配置历史服务器

在上一个页面中任务的history无法点进去，这里配置历史服务器让它能够进入

1、配置mapred-site.xml：`vim etc/hadoop/mapred-site.xml`：

```
<!-- 历史服务器端地址 -->
<property>
<name>mapreduce.jobhistory.address</name>
<value>hadoop130:10020</value>
</property>
<!-- 历史服务器web端地址 -->
<property>
<name>mapreduce.jobhistory.webapp.address</name>
<value>hadoop130:19888</value>
</property>
```

2、启动历史服务器：`sbin/mr-jobhistory-daemon.sh start historyserver`

可以用jps查看历史服务器是否启动。这样点进去就能跳转到任务信息页面，此时端口号为19888：

MapReduce Job job_1580427339159_0001

Logged in as: dr.who

Job Overview

Job Name:	word count
User Name:	root
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Fri Jan 31 17:15:10 CST 2020
Started:	Fri Jan 31 17:26:42 CST 2020
Finished:	Fri Jan 31 17:27:30 CST 2020
Elapsed:	47sec

Diagnostics:

Average Map Time	28sec
Average Shuffle Time	10sec
Average Merge Time	0sec
Average Reduce Time	1sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Fri Jan 31 17:25:58 CST 2020	hadoop130:8042	logs

Task Type

Map	Total	Complete
1	1	1
Reduce		

Attempt Type

Maps	Failed	Killed	Successful
0	0	1	1
Reduces			

左侧有概况、计数信息、配置信息、map和reduce的情况。右侧有logs记录了运行日志，当日志聚集配置完成后能够查看。

配置日志聚集

日志聚集概念：应用运行完成以后，将程序运行日志信息上传到HDFS系统上。

1、配置yarn-site.xml : `vim etc/hadoop/yarn-site.xml` :

```
<!-- 日志聚集功能使能 -->
<property>
<name>yarn.log-aggregation-enable</name>
<value>true</value>
</property>

<!-- 日志保留时间设置7天 -->
<property>
<name>yarn.log-aggregation.retain-seconds</name>
<value>604800</value>
</property>
```

将日志聚集功能打开，默认是false。日志保留时间设置，单位为秒。

2、NodeManager、ResourceManager和HistoryManager关闭然后再重新打开，关闭命令就是开启命令中的start改为stop即可：

`sbin/yarn-daemon.sh stop nodemanager`、`sbin/yarn-daemon.sh stop resourcemanager`、

`sbin/mr-jobhistory-daemon.sh stop historyserver`

3、删除输出文件：`bin/hdfs dfs -rm -R /user/yinyun/output`

4、重新运行MR程序：`hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount /user/yinyun/input /user/yinyun/output`

此时在上个页面点开logs就能进入日志页面：

The screenshot shows the Hadoop logs interface. On the left, there's a sidebar with 'Application', 'About', 'Jobs', and 'Tools'. The main area displays two log entries:

Log Type: stderr
Log Upload Time: 星期五一月 31 18:20:57 +0800 2020
Log Length: 1766

一月 31, 2020 6:20:14 下午 com.sun.jersey.guice.spi.container.GuiceComponentProviderFactory register
信息: Registering org.apache.hadoop.mapreduce.v2.app.webapp.JAXBCoContextResolver as a provider class
一月 31, 2020 6:20:14 下午 com.sun.jersey.guice.spi.container.GuiceComponentProviderFactory register
信息: Registering org.apache.hadoop.yarn.webapp.GenericExceptionHandler as a provider class
一月 31, 2020 6:20:14 下午 com.sun.jersey.guice.spi.container.GuiceComponentProviderFactory register
信息: Registering org.apache.hadoop.mapreduce.v2.app.webapp.AMWebServices as a root resource class
一月 31, 2020 6:20:14 下午 com.sun.jersey.server.impl.application.WebApplicationImpl _initiate
信息: Initiating Jersey application, version 'Jersey: 1.9 09/02/2011 11:17 AM'
一月 31, 2020 6:20:14 下午 com.sun.jersey.guice.spi.container.GuiceComponentProviderFactory getComponentProvider
信息: Binding org.apache.hadoop.mapreduce.v2.app.webapp.JAXBCoContextResolver to GuiceManagedComponentProvider with the scope "Singleton"
一月 31, 2020 6:20:15 下午 com.sun.jersey.guice.spi.container.GuiceComponentProviderFactory getComponentProvider
信息: Binding org.apache.hadoop.yarn.webapp.GenericExceptionHandler to GuiceManagedComponentProvider with the scope "Singleton"
一月 31, 2020 6:20:15 下午 com.sun.jersey.guice.spi.container.GuiceComponentProviderFactory getComponentProvider
信息: Binding org.apache.hadoop.mapreduce.v2.app.webapp.AMWebServices to GuiceManagedComponentProvider with the scope "PerRequest".
log4j:WARN No appenders could be found for logger (org.apache.hadoop.ipc.Server).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Log Type: stdout
Log Upload Time: 星期五一月 31 18:20:57 +0800 2020
Log Length: 0

Log Type: syslog
Log Upload Time: 星期五一月 31 18:20:57 +0800 2020
Log Length: 35185

Showing 4096 bytes of 35185 total. Click [here](#) for the full log.

66008807-root-wordcount-1580466015699.jhist_tmp
2020-01-31 18:20:49,002 INFO [RMCommunicator Allocator] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Before Scheduling: PendingRed;
2020-01-31 18:20:49,639 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copied to done location: hdfs:/;
2020-01-31 18:20:49,646 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copying hdfs://hadoop130:9000/tr
2020-01-31 18:20:49,712 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copied to done location: hdfs:/;
2020-01-31 18:20:49,763 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://hadoop130:9000/tr

点击[here](#)就能查看完整日志，在logs目录下一样也能查看这些日志。

配置文件说明

hadoop中默认的配置文件都在jar包中，当需要修改自定义配置时，需要在\$HADOOP_HOME/etc/hadoop中修改。默认配置文件位置：

要获取的默认文件	文件存放在Hadoop的jar包中的位置
[core-default.xml]	hadoop-common-2.7.2.jar/ core-default.xml
[hdfs-default.xml]	hadoop-hdfs-2.7.2.jar/ hdfs-default.xml

要获取的默认文件 [yarn-default.xml]	文件存放在Hadoop的jar包中的位置 hadoop-yarn-common-2.7.2.jar/yarn-default.xml
[mapred-default.xml]	hadoop-mapreduce-client-core-2.7.2.jar/mapred-default.xml

完全分布式模式

准备工作

准备三台虚拟机，配置好静态IP、关闭防火墙、在/etc/hosts中配置映射关系（在测试的windows上也要添加映射关系）：

```
[root@hadoop131 ~]# vim /etc/hosts
192.168.47.130    hadoop130
192.168.47.131    hadoop131
192.168.47.132    hadoop132
192.168.47.133    hadoop133
```

集群分发脚本

1、scp (secure copy) 安全拷贝

它用来实现服务器与服务器之间的数据拷贝。

基本格式 `scp -r 源数据 目的数据`

(1) 从本机拷到对方，在hadoop101上，将hadoop101中/opt/module目录下的软件拷贝到hadoop102上：

```
scp -r /opt/module root@hadoop102:/opt/module
```

拷贝到对方服务器时注意，登录的用户必须有在对应目录写的权限

(2) 从对方拷到本机，在hadoop103上，将hadoop101服务器上的/opt/module目录下的软件拷贝到hadoop103上：

```
sudo scp -r atguigu@hadoop101:/opt/module root@hadoop103:/opt/module
```

拷贝到当前服务器时也要注意当前用户必须有在对应目录写的权限，用普通用户执行时可以用sudo权限

(3) 在本机操作，在其他两台服务器之间实现通信，在hadoop103上操作将hadoop101中/opt/module目录下的软件拷贝到hadoop104上。

```
scp -r atguigu@hadoop101:/opt/module root@hadoop104:/opt/module
```

注意文件拷贝完成后，要修改所有者和所属组，配置文件拷贝完成后要source一下。

2、rsync远程同步

rsync和scp区别：用rsync做文件的复制要比scp的速度快，rsync只对差异文件做更新。scp是把所有文件都复制过去。它还能支持复制符号链接。

基本格式：`rsync -rvl 要拷贝的文件 目的`，rvl分别代表递归、显示拷贝过程、拷贝符号链接。

把hadoop101机器上的/opt/software目录同步到hadoop102服务器的root用户下的/opt/目录：

```
rsync -rvl /opt/software/ root@hadoop102:/opt/software
```

3、xsync集群分发脚本

当在一个节点改变一个文件时，想把这种改变同步到其他节点，就需要一个简单高效的脚本。期望执行命令：

`xsync` 要同步的文件 就可以完成这种同步。

在/usr/local/bin中创建脚本 `touch xsync`，然后修改：

```
#!/bin/bash
#1 获取输入参数个数，如果没有参数，直接退出
pcount=$#
if((pcount==0)); then
echo no args;
exit;
fi

#2 获取文件名称
p1=$1
fname=`basename $p1`
echo fname=$fname

#3 获取上级目录到绝对路径
pdir=`cd -P $(dirname $p1); pwd`
echo pdir=$pdir

#4 获取当前用户名
user=`whoami`

#5 循环
for(host=132; host<134; host++); do
    echo ----- hadoop$host -----
    rsync -rvl $pdir/$fname $user@hadoop$host:$pdir
done
```

basename就是取到文件名，如果输入的是绝对路径它也能取到文件名。dirname后跟绝对路径它的返回值会把绝对路径中的文件名和最后一个/拿掉，如果直接跟文件名返回值为。

然后修改文件权限：`chmod 777 xsync`，然后就可以按照对应格式执行了（脚本是要在hadoop131上执行的）

脚本写好后要放在\$PATH定义的位置，这样才能不用加完整路径直接执行，这里放到/usr/local/bin中。

配置集群

HDFS的NameNode和SecondaryNameNode占用内存比例差不多，故一般不将其布置在一个服务器上，DataNode是每一个服务器都应该部署的。yarn的ResourceManager是整个集群的manager，故将其布置在没有NameNode和SecondaryNameNode的服务器上，故集群部署计划如下：

	hadoop131	hadoop132	hadoop133
HDFS	NameNode、 DataNode	DataNode	SecondaryNameNode、 DataNode
yarn	NodeManager	ResourceManager、 NodeManager	NodeManager

在/etc/hadoop中配置几个文件：

1、 core-site.xml :

```
<!-- 指定HDFS中NameNode的地址 -->
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://hadoop131:9000</value>
</property>

<!-- 指定Hadoop运行时产生文件的存储目录 -->
<property>
    <name>hadoop.tmp.dir</name>
    <value>/opt/module/hadoop-2.7.2/data/tmp</value>
</property>
```

这里要修改NameNode的地址，将其放在指定的对应服务器上。

2、 hadoop-env.sh : 设置具体的JAVA_HOME

3、 hdfs-site.xml :

```
<property>
    <name>dfs.replication</name>
    <value>3</value>
</property>

<!-- 指定Hadoop辅助名称节点主机配置 -->
<property>
    <name>dfs.namenode.secondary.http-address</name>
    <value>hadoop133:50090</value>
</property>
```

副本值改为3，这里默认值就是3，所以删掉也可以。设置secondarynamenode的位置。

4、 yarn-env.sh : 设置具体的JAVA_HOME

5、 yarn-site.xml :

```
<!-- Reducer获取数据的方式 -->
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>

<!-- 指定YARN的ResourceManager的地址 -->
<property>
    <name>yarn.resourcemanager.hostname</name>
    <value>hadoop132</value>
</property>
```

6、 mapred-env.sh : 设置具体的JAVA_HOME

7、 mapred-site.xml : 首先将源文件修改名称：`cp mapred-site.xml.template mapred-site.xml`

然后加入：

```
<!-- 指定MR运行在Yarn上 -->
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

配置文件设置完成后用脚本同步：`xsync /opt/module/hadoop-2.7.2/`

集群单点启动

因为之前的节点都启动过NameNode，所以要先停止namenode和datanode，然后将data和logs目录删除：

`rm -rf data/ logs/`，然后在hadoop131将namenode格式化并启动，在启动datanode，在其他节点启动datanode。然后访问hadoop131:50070即可访问HDFS，这个ip就是namenode所在节点的ip。但是这种启动方式太繁琐了，当节点几百上千时几乎无法启动。

集群SSH配置

ssh可以直接连接到另一个节点：`ssh ip或域名`

NameManager的hadoop131和ResourceManager的hadoop132都需要用ssh连接其他节点进行通信，故在这两个节点上要生成秘钥对，然后向其他节点分发公钥，包括自己的节点，也要保存一份公钥。

生成秘钥对：`ssh-keygen -t rsa`，分发公钥给其他节点：`ssh-copy-id hadoop102`

群起集群

特别注意namenode那个节点一定要保证之前没有启动过namenode，如果之前启动过就必须先停止namenode和datanode，然后将data和logs目录删除，最后格式化namenode。（如果是不启动namenode的节点无需格式化namenode）

修改etc/hadoop/slaves文件在其中添加集群节点的域名：

```
hadoop131
hadoop132
hadoop133
```

注意这里不能有多余的空格和回车，否则会报错。

所有节点同步该配置文件：`xsync slaves`

启动HDFS（在NameNode节点处）：`sbin/start-dfs.sh`（有时namenode会莫名启动且jps查不到，关闭namenode即可）

启动yarn（在ResourceManager处）：`sbin/start-yarn.sh`

此时就可以访问HDFS和SecondaryNameNode，它在hadoop133:50090（注意HDFS只能在NameNode对应域名访问，SecondaryNameNode也一样必须在安装的域名访问）

集群启动/停止方式总结

分别启动/停止HDFS组件：`hadoop-daemon.sh start/stop`
`namenode/datanode/secondarynamenode`

启动/停止YARN : `yarn-daemon.sh start/stop resourcemanager/nodemanager`

整体启动/停止HDFS : (sbin下的) `start-dfs.sh / stop-dfs.sh`

整体启动/停止YARN : (sbin下的) `start-yarn.sh / stop-yarn.sh`

集群时间同步

具体方式是找一个机器，作为时间服务器，所有的机器与这台集群时间进行定时的同步，比如，每隔十分钟，同步一次时间。

1、配置时间服务器：

(1) 检查ntp是否安装：`rpm -qa | grep ntp`，如果出现以下结果说明ntp正常：

```
ntp-4.2.6p5-10.el6.centos.x86_64
fontpackages-filesystem-1.41-1.1.el6.noarch
ntpdate-4.2.6p5-10.el6.centos.x86_64
```

(2) 修改ntp配置文件：`vi /etc/ntp.conf`：删除下列语句的注释符号表明网段192.168.1的节点都能与时间服务器统一时间。

```
#restrict 192.168.1.0 mask 255.255.255.0 nomodify notrap
```

以下语句每行加#，表明不使用其他互联网上的时间。

```
server 0.centos.pool.ntp.org iburst
server 1.centos.pool.ntp.org iburst
server 2.centos.pool.ntp.org iburst
server 3.centos.pool.ntp.org iburst
```

加入下列语句，表明当该节点丢失网络连接，依然可以采用本地时间作为时间服务器为集群中的其他节点提供时间同步。

```
server 127.127.1.0
fudge 127.127.1.0 stratum 10
```

(3) 修改/etc/sysconfig/ntp，加入下列语句，表示让硬件时间与系统时间一起同步。

```
SYNC_HWCLOCK=yes
```

(4) 重新启动ntp服务：`service ntpd status`、`service ntpd start`。

(5) 设置ntp服务开机启动：`chkconfig ntpd on`

2、配置其他节点

设置定时任务：`crontab -e`：

```
*/10 * * * * /usr/sbin/ntpdate 时间服务器域名或IP
```

修改节点时间 : date -s "2020-2-26 10:12:30"

HDFS操作

HDFS概述

HDFS产生背景

随着数据量越来越大，在一个操作系统存不下所有的数据，那么就分配到更多的操作系统管理的磁盘中，但是不方便管理和维护，迫切需要一种系统来管理多台机器上的文件，这就是分布式文件管理系统。HDFS只是分布式文件管理系统中的一种。

HDFS定义与使用场景

HDFS (Hadoop Distributed File System)，它是一个文件系统，用于存储文件，通过目录树来定位文件；其次，它是分布式的，由很多服务器联合起来实现其功能，集群中的服务器有各自的角色。

HDFS的使用场景：适合一次写入，多次读出的场景，且不支持文件的修改。适合用来做数据分析，并不适合用来做网盘应用。

HDFS优缺点

优点

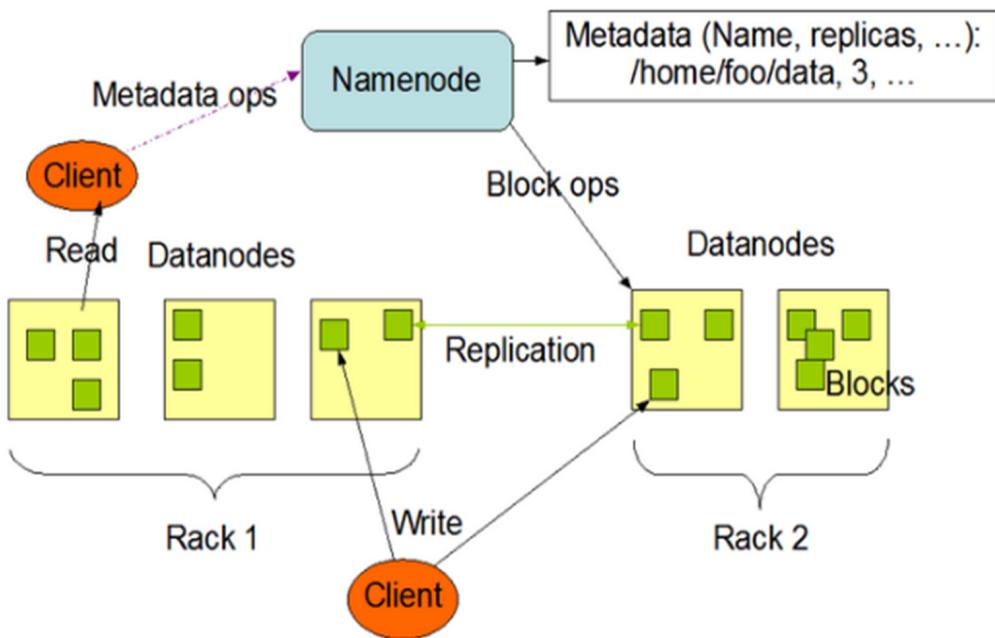
- 1、高容错性。数据自动保存多个副本，即使一个副本损坏其他依然可用，且副本还可以通过其他副本自动恢复。
- 2、适合处理大数据。处理数据规模达到GB/TB甚至PB，处理文件数量可达百万/千万/亿。
- 3、对单个节点的配置要求不高，可以布置在廉价机器上。

缺点

- 1、不适合低延时数据访问，比如毫秒级的存储数据，是做不到的。
- 2、无法高效的对大量小文件进行存储。
 - (1) 存储大量小文件的话，它会占用NameNode大量的内存来存储文件目录和块信息。这样是不可取的，因为NameNode的内存总是有限的；
 - (2) 小文件存储的寻址时间会超过读取时间，它违反了HDFS的设计目标。
- 3) 不支持并发写入、文件随机修改。
 - (1) 一个文件只能有一个写，不允许多个线程同时写；
 - (2) 仅支持数据append (追加)，不支持文件的随机修改。

HDFS组成架构

HDFS Architecture



1、NameNode (nn) : 就是Master , 它是一个主管、管理者。

(1) 管理HDFS的名称空间；

(2) 配置副本策略；

(3) 管理数据块 (Block) 映射信息；

(4) 处理客户端读写请求。

2、DataNode : 就是Slave。NameNode下达命令 , DataNode执行实际的操作。

(1) 存储实际的数据块；

(2) 执行数据块的读/写操作。

3、Client : 就是客户端。

(1) 文件切分。文件上传HDFS的时候 , Client将文件切分成一个一个的Block , 然后进行上传；

(2) 与NameNode交互 , 获取文件的位置信息；

(3) 与DataNode交互 , 读取或者写入数据；

(4) Client提供一些命令来管理HDFS , 比如NameNode格式化；

(5) Client可以通过一些命令来访问HDFS , 比如对HDFS增删查改操作；

4、Secondary NameNode : 并非NameNode的热备。当NameNode挂掉的时候 , 它并不能马上替换NameNode并提供服务。

(1) 辅助NameNode , 分担其工作量 , 比如定期合并Fsimage和Edits , 并推送给NameNode ；

(2) 在紧急情况下 , 可辅助恢复NameNode。

HDFS文件块大小

HDFS中的文件在物理上是分块存储 (Block) , 块的大小可以通过修改配置文件hdfs-default.xml的配置参数(dfs.blocksize)来规定 , 默认大小在Hadoop2.x版本中是128M , 老版本中是64M。

默认设置为128M的原因：在大量block中寻找一个特定block的时间大约是10ms，有研究表明寻址时间为传输时间的1%时整体效率最高，即传输速度为1000ms时最好，再根据磁盘的传输速率一般为100MB/s，故一个block的大小设置为100M较为合适，故最终定为128M。

文件块大小的设置主要取决于磁盘传输效率，如果磁盘传输效率较高，如200MB/s，那么block的大小应设置为256M，如果磁盘传输效率较低也可调整。

文件块的大小设置的不能过大，否则会导致传输时间过长，也不能过小，否则会导致文件存储效率低且寻址时间变长。

HDFS的shell操作

基本语法为 `bin/hadoop fs` 具体命令 OR `bin/hdfs dfs` 具体命令，dfs是fs的实现类。

(1) -help：输出这个命令参数，可以用这种方法查看帮助

```
hadoop fs -help rm
```

(2) -ls：显示目录信息

```
hadoop fs -ls /
```

(3) -mkdir：在HDFS上创建目录

```
hadoop fs -mkdir -p /sanguo/shuguo
```

(4) -moveFromLocal：从本地剪切粘贴到HDFS

```
hadoop fs -moveFromLocal ./kongming.txt /sanguo/shuguo
```

(5) -appendToFile：追加一个本地文件到已经存在的文件末尾

```
hadoop fs -appendToFile liubei.txt /sanguo/shuguo/kongming.txt
```

(6) -cat：显示文件内容

```
hadoop fs -cat /sanguo/shuguo/kongming.txt
```

(7) -chgrp、-chmod、-chown：Linux文件系统中的用法一样，修改文件所属权限

```
hadoop fs -chmod 666 /sanguo/shuguo/kongming.txt
```

```
hadoop fs -chown atguigu:atguigu /sanguo/shuguo/kongming.txt
```

(8) -copyFromLocal：从本地文件系统中拷贝文件到HDFS路径去

```
hadoop fs -copyFromLocal README.txt /
```

(9) -copyToLocal：从HDFS拷贝到本地

```
hadoop fs -copyToLocal /sanguo/shuguo/kongming.txt ./
```

(10) -cp：从HDFS的一个路径拷贝到HDFS的另一个路径

```
hadoop fs -cp /sanguo/shuguo/kongming.txt /zhuge.txt
```

(11) -mv：在HDFS目录中移动文件

```
hadoop fs -mv /zhuge.txt /sanguo/shuguo/
```

(12) -get：等同于copyToLocal，就是从HDFS下载文件到本地

```
hadoop fs -get /sanguo/shuguo/kongming.txt ./
```

(13) -getmerge : 合并下载多个文件到本地 , 比如HDFS的目录 /user/atguigu/test 下有多个文件:log.1, log.2, log.3,...

```
hadoop fs -getmerge /user/atguigu/test/* ./zaiyiqi.txt
```

(14) -put : 等同于copyFromLocal

```
hadoop fs -put ./zaiyiqi.txt /user/atguigu/test/
```

(15) -tail : 显示一个文件的末尾

```
hadoop fs -tail /sanguo/shuguo/kongming.txt
```

(16) -rm : 删除文件或文件夹

```
hadoop fs -rm /user/atguigu/test/jinlian2.txt
```

(17) -rmdir : 删除空目录

```
hadoop fs -rmdir /test
```

(18) -du统计文件夹的大小信息

查看总大小 : `hadoop fs -du -s -h /user/atguigu/test`

查看每一个 : `hadoop fs -du -h /user/atguigu/test`

(19) -setrep : 设置HDFS中文件的副本数量

```
hadoop fs -setrep 10 /sanguo/shuguo/kongming.txt
```

如果设置副本数小于datanode的数量 , 那么真实副本数就等于datanode的个数 , 如果此时增加节点个数 , 新节点也会备份该文件 , 直到达到副本数 , 在HDFS中点击文件可以查看该文件到底备份在哪个节点上。

HDFS客户端环境搭建

- 1、得到win7编译过的hadoop jar包。
- 2、配置HADOOP_HOME环境变量 , 就是hadoop的解压路径 : `E:\hadoop-2.7.2`。
- 3、配置Path环境变量 , 在末尾追加 : `%HADOOP_HOME%\bin`。
- 4、创建maven工程 , 并在pom.xml中添加 :

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.8.2</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.7.2</version>
    </dependency>
```

```

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.7.2</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>2.7.2</version>
</dependency>
</dependencies>

```

5、在项目的src/main/resources目录下，新建一个文件，命名为“log4j.properties”，在文件中填入：

```

log4j.rootLogger=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
log4j.appender.logfile=org.apache.log4j.FileAppender
log4j.appender.logfile.File=target/spring.log
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n

```

6、创建HdfsClient类：

```

package hadoop;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.junit.jupiter.api.Test;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

public class HdfsClient {
    public static void main(String[] args) throws IOException,
URISyntaxException, InterruptedException {
        // 1 获取文件系统
        Configuration configuration = new Configuration();
        // 配置在集群上运行
        // configuration.set("fs.defaultFS", "hdfs://hadoop131:9000");
        // FileSystem fs = FileSystem.get(configuration);

        FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

        // 2 创建目录
        fs.mkdirs(new Path("/1108"));

        System.out.println("ok!");
        // 3 关闭资源
        fs.close();
    }
}

```

获得文件系统对象的方式有两种，第一种已经被注释，用第一种方式设置configuration时，set方法的两个参数是配置文件core-site文件中的NameNode的地址来作为键和值：

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

 http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
<!-- 指定HDFS中NameNode的地址 -->
<property>
<name>fs.defaultFS</name>
<value>hdfs://hadoop131:9000</value>
</property>

<!-- 指定Hadoop运行时产生文件的存储目录 -->
<property>
    <name>hadoop.tmp.dir</name>
    <value>/opt/module/hadoop-2.7.2/data/tmp</value>
</property>

</configuration>
~
```

用第一种方式时需要在运行时传给jvm一个参数，这个参数指定了登入的用户名。第二种方式直接将用户名作为参数传入方法中，比较简单，在代码中完成创建目录，在HDFS中就能看见该目录。运行此程序可能报错，解决方法如下：

- 1、可能因为JAVA_HOME有空格，解决方法：在dos下运行 `mklink /J C:\java "C:\Program Files\Java\jdk1.8.0_201"` 创建软连接，然后设置JAVA_HOME为c:\java。
- 2、环境变量HADOOP_HOME不要设置成多级目录。

HDFS的API操作

文件上传

```
// 1 获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 上传文件
fs.copyFromLocalFile(new Path("e:/banzhang.txt"), new
Path("/banzhang.txt"));

// 3 关闭资源
fs.close();

System.out.println("over");
```

文件上传时可以用代码指定副本数 : `configuration.set("dfs.replication", "2");`

也可以用配置文件指定副本数 , 在resources目录下建立一个hdfs-site.xml文件 :

```
<?xml version="1.0" encoding="UTF-8"?>
<xm1-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

此外 , 指定副本数的方法还可以在集群中设置配置文件。这几种设置方式的优先级顺序是 : 代码 > 客户端配置文件 > 集群配置文件。

文件下载

```
// 1 获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 执行下载操作
// boolean delsrc 指是否将原文件删除
// Path src 指要下载的文件路径
// Path dst 指将文件下载到的路径
// boolean useRawLocalFileSystem 是否开启文件校验
fs.copyToLocalFile(false, new Path("/banzhang.txt"), new
Path("e:/banhua.txt"), true);

// 3 关闭资源
fs.close();
```

注意这里是可以更改文件名的 , 两个path不一致也能运行。

上面这种方式通过FileSystem对象调用copyToLocalFile方法来实现 , 第一个参数是指是否将原文件删除 , 这里用false代表下载 , 如果是true代表剪切 , 最后一个参数代表是否开启本地模式 , 默认是false相当于关闭 , 当关闭本地模式时下载文件会产生一个crc文件 , 它是和安全校验有关的文件 , 改成true之后下载时不会产生该文件 , 有些计算机必须改成false , 因为计算机本身有一定的安全性要求。

也可以直接调用两参数的copyToLocalFile方法 , 效果相同 , 但是不能控制其他两个参数。

文件删除

```

// 1 获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 执行删除
fs.delete(new Path("/0508/"), true);

// 3 关闭资源
fs.close();

```

如果删除的是文件夹，`delete`方法的第二个参数必须是`true`，它代表递归。

更改文件名

```

// 1 获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 修改文件名称
fs.rename(new Path("/banzhang.txt"), new Path("/banhua.txt"));

// 3 关闭资源
fs.close();

```

文件详情查看

查看文件名称、权限、长度、块信息（就是文件存在哪个节点上）。

```

// 1获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 获取文件详情
RemoteIterator<LocatedFileStatus> listFiles = fs.listFiles(new Path("/"),
true);

while(listFiles.hasNext()){
    LocatedFileStatus status = listFiles.next();

    // 输出详情
    // 文件名称
    System.out.println(status.getPath().getName());
    // 长度
    System.out.println(status.getLength());
    // 权限
    System.out.println(status.getPermission());
    // 分组
    System.out.println(status.getGroup());

    // 获取存储的块信息
    BlockLocation[] blockLocations = status.getBlockLocations();
}

```

```

for (BlockLocation blockLocation : blockLocations) {

    // 获取块存储的主机节点
    String[] hosts = blockLocation.getHosts();

    for (String host : hosts) {
        System.out.println(host);
    }
}

System.out.println("-----班长的分割线-----");
}

// 3 关闭资源
fs.close();

```

通过FileSystem对象的listFiles方法得到迭代器RemotelIterator，然后遍历迭代器，取出其中的文件属性，块信息中可能有多个主机名，此时也需要遍历来操作。

文件和文件夹判断

```

// 1 获取文件配置信息
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 判断是文件还是文件夹
FileStatus[] listStatus = fs.listStatus(new Path("/"));

for (FileStatus filestatus : listStatus) {

    // 如果是文件
    if (filestatus.isFile()) {
        System.out.println("f:" + filestatus.getPath().getName());
    } else {
        System.out.println("d:" + filestatus.getPath().getName());
    }
}

// 3 关闭资源
fs.close();

```

判断方式主要是FileStatus调用isFile方法，根据返回值判断是否是文件。

HDFS的IO流操作

操作文件的API底层都是用流操作的。

文件上传

```

// 1 获取文件系统
Configuration configuration = new Configuration();

```

```
Filesystem fs = Filesystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 创建输入流
FileInputStream fis = new FileInputStream(new File("e:/banhua.txt"));

// 3 获取输出流
FSDataOutputStream fos = fs.create(new Path("/banhua.txt"));

// 4 流对拷
IOUtils.copyBytes(fis, fos, configuration);

// 5 关闭资源
IOUtils.closeStream(fos);
IOUtils.closeStream(fis);
fs.close();
```

想要把文件上传到HDFS，那么输出流就要用FileSystem来创建，输入流直接用FileInputStream类，两个流之间的转换可以通过IOUtils类的copyBytes方法。

文件下载

```
// 1 获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = Filesystem.get(new URI("hdfs://hadoop131:9000"),
configuration, "root");

// 2 获取输入流
FSDataInputStream fis = fs.open(new Path("/banhua.txt"));

// 3 获取输出流
FileOutputStream fos = new FileOutputStream(new File("e:/banhua.txt"));

// 4 流的对拷
IOUtils.copyBytes(fis, fos, configuration);

// 5 关闭资源
IOUtils.closeStream(fos);
IOUtils.closeStream(fis);
fs.close();
```

想要把文件从HDFS上下载下来，那么输出流就用FileOutputStream类，输入流是用FileSystem创建的。

定位文件读取

当日志文件存入HDFS中，文件过大时可能会分块存储（默认一块128M），有时只需下载最新的日志文件，此时就需要用到文件的分块下载。

下载第一块：

```
// 1 获取文件系统
Configuration configuration = new Configuration();
```

```

Filesystem fs = Filesystem.get(new URI("hdfs://hadoop102:9000"),
configuration, "atguigu");

// 2 获取输入流
FSDataInputStream fis = fs.open(new Path("/hadoop-2.7.2.tar.gz"));

// 3 创建输出流
FileOutputStream fos = new FileOutputStream(new File("e:/hadoop-
2.7.2.tar.gz.part1"));

// 4 流的拷贝
byte[] buf = new byte[1024];

for(int i =0 ; i < 1024 * 128; i++){
    fis.read(buf);
    fos.write(buf);
}

// 5关闭资源
IUtils.closeStream(fis);
IUtils.closeStream(fos);
fs.close();

```

利用字节数组定量读取128M的文件。

下载第二块（对本例来说是最后一块）：

```

// 1 获取文件系统
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"),
configuration, "atguigu");

// 2 打开输入流
FSDataInputStream fis = fs.open(new Path("/hadoop-2.7.2.tar.gz"));

// 3 定位输入数据位置
fis.seek(1024*1024*128);

// 4 创建输出流
FileOutputStream fos = new FileOutputStream(new File("e:/hadoop-
2.7.2.tar.gz.part2"));

// 5 流的对拷
IUtils.copyBytes(fis, fos, configuration);

// 6 关闭资源
IUtils.closeStream(fis);
IUtils.closeStream(fos);

```

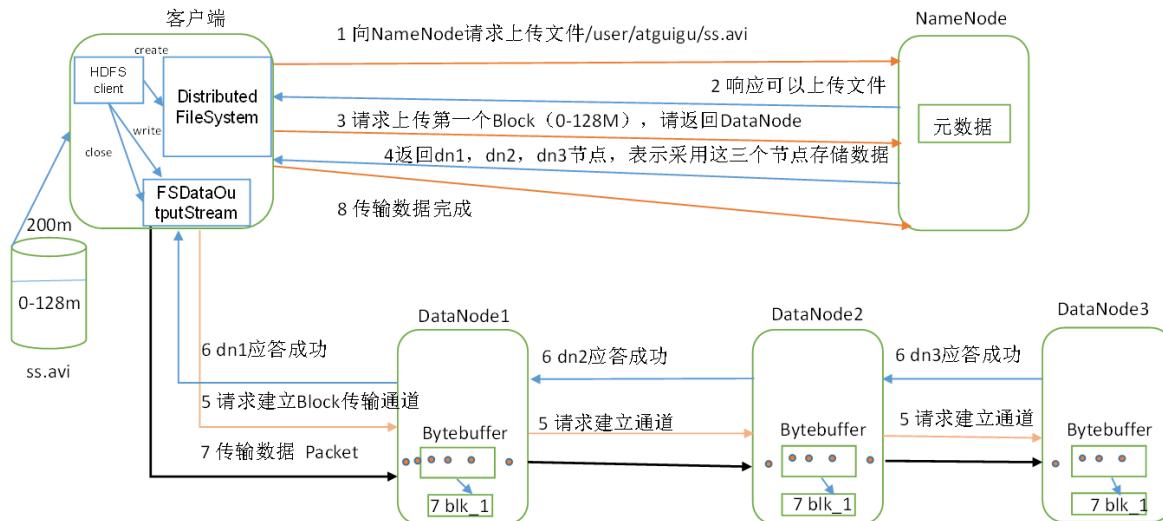
这里要用seek方法来进行定位，从128M后开始读。

两块文件的合并在dos中可以用定向的方法：`type part2 >> part1`，运行完毕后part1就是合并后的文件。

HDFS机理

向HDFS读写数据的过程

向HDFS写数据的过程



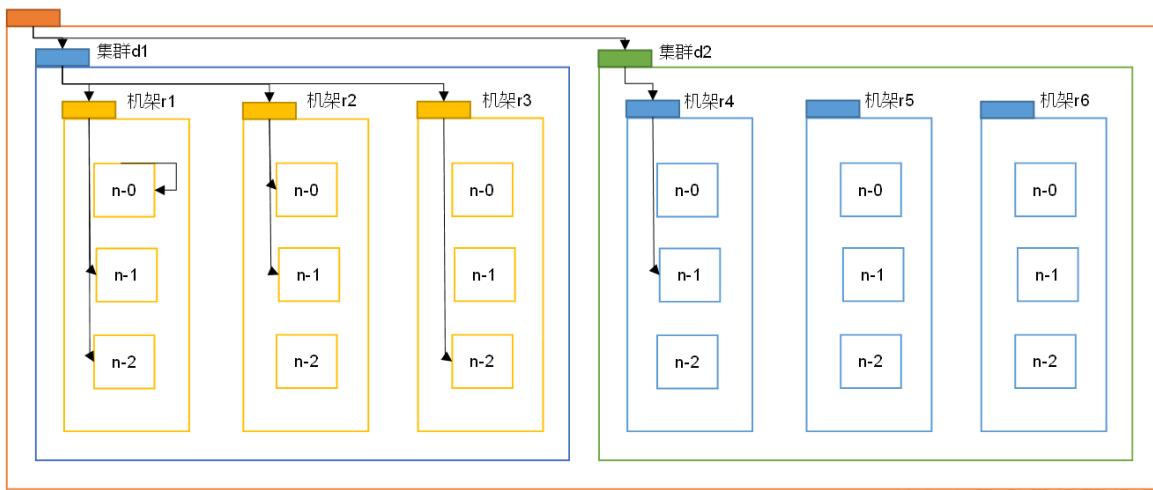
- 1) 客户端通过Distributed FileSystem模块向NameNode请求上传文件 , NameNode检查目标文件是否已存在 , 父目录是否存在。
 - 2) NameNode返回是否可以上传。
 - 3) 客户端请求第一个 Block上传到哪几个DataNode服务器上。 (传输数据时是要把文件分块传输的)
 - 4) NameNode返回3个DataNode节点 , 分别为dn1、 dn2、 dn3。
 - 5) 客户端通过FSDataOutputStream模块请求dn1上传数据 , dn1收到请求会继续调用dn2 , 然后dn2调用dn3 , 将这个通信管道建立完成。 (传输数据时优先向距离近、负载小的datanode传递数据)
 - 6) dn1、 dn2、 dn3逐级应答客户端。
 - 7) 客户端开始往dn1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 , dn1收到一个Packet就会传给dn2 , dn2传给dn3 ; dn1每传一个packet会放入一个应答队列等待应答。
- 传输数据的基本单位是packet , 一个节点接收到paket时就传递到下一个datanode , 同时写入节点的bytebuffer队列中等待本地序列化。上传完成后倒序清空bytebuffer中的数据。
- 8) 当一个Block传输完成之后 , 客户端再次请求NameNode上传第二个Block的服务器。 (重复执行3-7步)。

网络拓扑-节点距离计算

一般是用网关和服务器搭建起局域网 , 一个机架就是最小范围的局域网 , 而集群可能由几个机架组成。计算网络节点距离时 , 按照两个节点到达最近的共同祖先的距离总和来计算 , 如下图的四个例子 :

Distance(/d1/r1/n0, /d1/r1/n0)=0 (同一节点上的进程)
Distance(/d1/r1/n1, /d1/r1/n2)=2 (同一机架上的不同节点)

Distance(/d1/r2/n0, /d1/r3/n2)=4 (同一数据中心不同机架上的节点)
Distance(/d1/r2/n1, /d2/r4/n1)=6 (不同数据中心的节点)



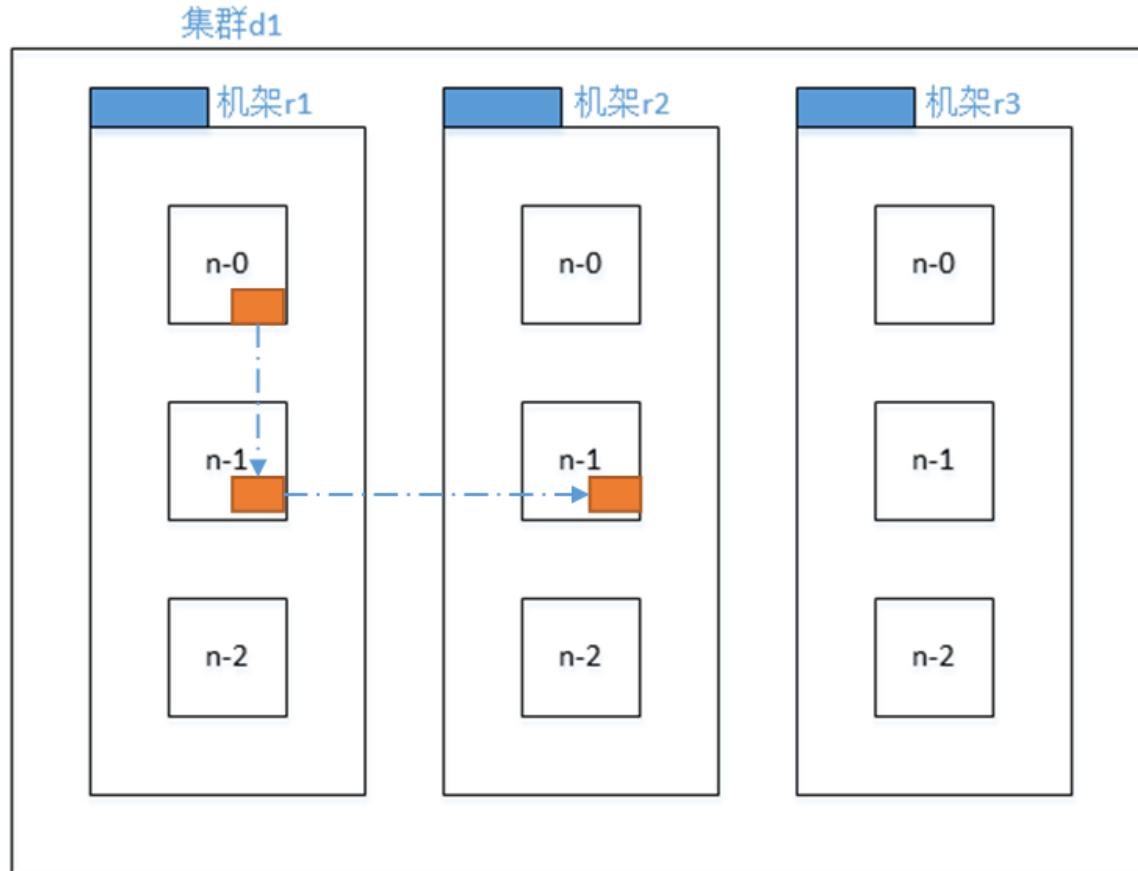
机架感知 (副本存储节点选择)

默认三个副本存放节点的位置：

第一个副本在Client所处的节点上。如果客户端在集群外，随机选一个。

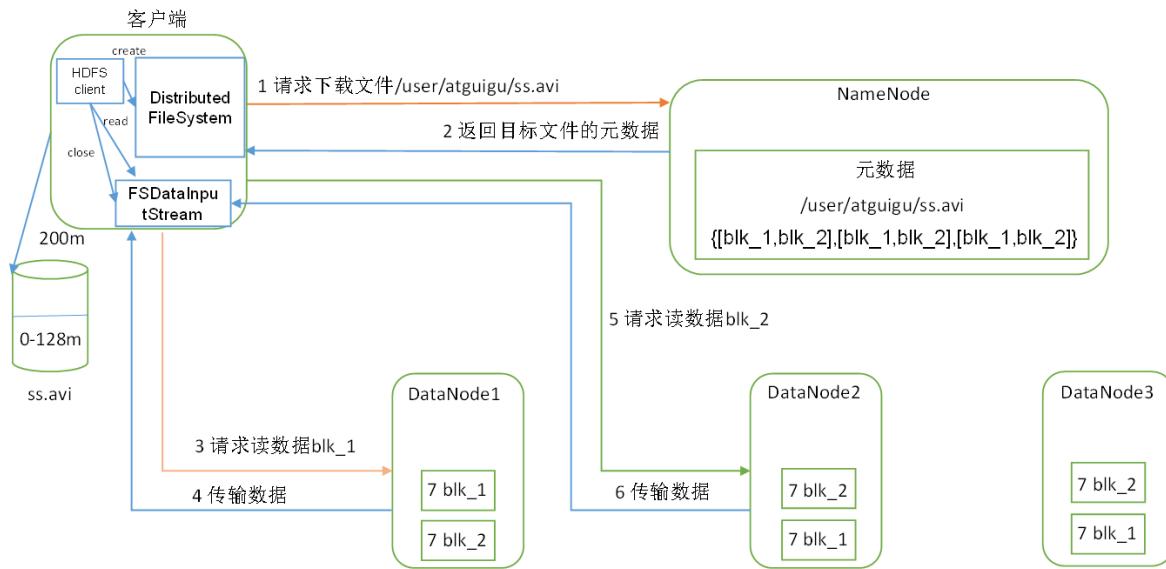
第二个副本和第一个副本位于相同机架，随机节点。

第三个副本位于不同机架，随机节点。



副本存放节点的选择兼顾了传输速度和安全性两方面，两个副本在一个机架上时传输距离短，速度快，两个副本在不同机架上时，某个机架的崩溃不会影响另一个机架，增加了抗风险能力。

读HDFS数据的过程



- 1) 客户端通过Distributed FileSystem向NameNode请求下载文件 , NameNode通过查询元数据 , 找到文件块所在的DataNode地址。
- 2) 挑选一台DataNode (就近原则 , 然后随机) 服务器 , 请求读取数据。
- 3) DataNode开始传输数据给客户端 (从磁盘里面读取数据输入流 , 以Packet为单位来做校验) 。
- 4) 客户端以Packet为单位接收 , 先在本地缓存 , 然后写入目标文件。如果一个文件有多个文件块在其他节点时 , 输入流会在请求完第一个节点后请求第二个节点。

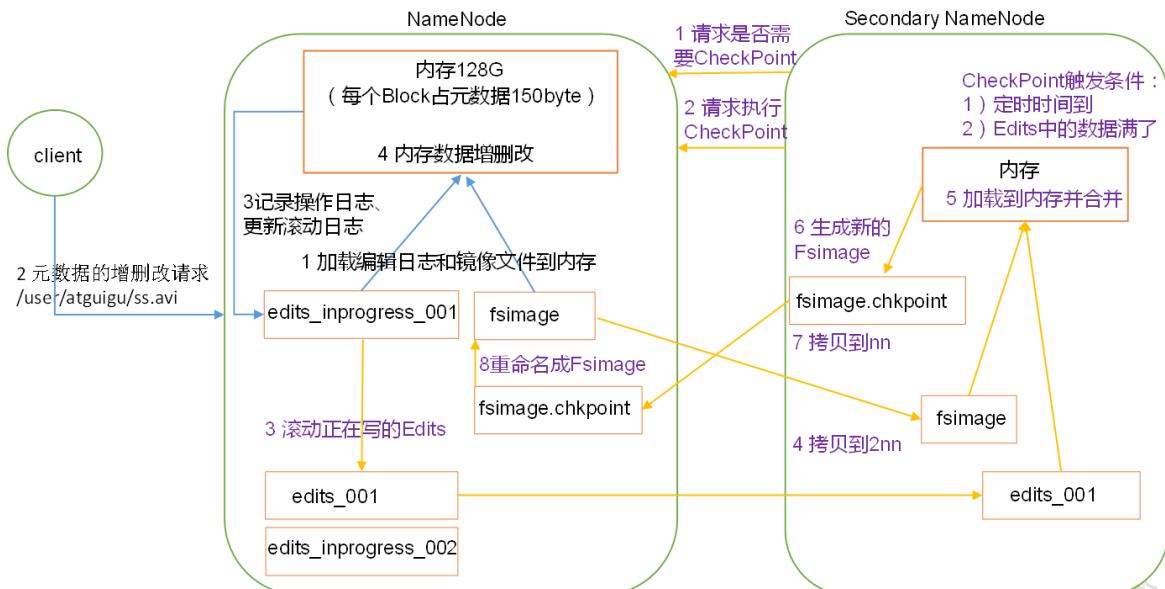
NameNode和SecondaryNameNode

NN和2NN工作机制

NameNode存储着元数据，这些元数据存在内存中，被称为镜像文件FsImage，因为存在内存中所以读的速度很快，但是安全性不能保证，编辑日志Edits文件可以解决这个问题，每次更新数据时修改内存中的元数据并将修改追加到编辑日志（只追加文件速度快），这样一旦NameNode节点断点，还可以合并FsImage和Edits，保护数据。

长时间运行会导致编辑日志Edits过大，于是就需要经常进行合成元数据并清空编辑日志的操作，这个工作由SecondaryNameNode承担，专门用于FsImage和Edits的合并。

运行时NN和2NN工作机制：



1、第一阶段：NameNode启动

- (1) 第一次启动NameNode格式化后，创建Fsimage和Edits文件。如果不是第一次启动，直接加载编辑日志和镜像文件到内存。
- (2) 客户端对元数据进行增删改的请求。
- (3) NameNode记录操作日志，更新滚动日志。
- (4) NameNode在内存中对数据进行增删改。

2、第二阶段：Secondary NameNode工作

- (1) Secondary NameNode询问NameNode是否需要CheckPoint。直接带回NameNode是否检查结果。
- (2) Secondary NameNode请求执行CheckPoint。
- (3) NameNode滚动正在写的Edits日志。
- (4) 将滚动前的编辑日志和镜像文件拷贝到Secondary NameNode。
- (5) Secondary NameNode加载编辑日志和镜像文件到内存，并合并。
- (6) 生成新的镜像文件fsimage.chkpoint。（为了避免冲突生成一个空的编辑日志）
- (7) 拷贝fsimage.chkpoint到NameNode。
- (8) NameNode将fsimage.chkpoint重新命名成fsimage。

Fsimage和Edits解析

NameNode被格式化之后，将在/opt/module/hadoop-2.7.2/data/tmp/dfs/name/current目录中产生镜像文件，每次NameNode启动的时候都会将Fsimage文件读入内存，加载Edits里面的更新操作，保证内存中的元数据信息是最新的、同步的，可以看成NameNode启动的时候就将Fsimage和Edits文件进行了合并。

Fsimage文件不能直接打开，要使用 `hdfs oiv -p XML -i fsimage_000000000000000025 -o /opt/module/hadoop-2.7.2/fsimage.xml` 来将文件转换为xml文件输出到hadoop的安装目录下，转换成xml文件后就能打开了。Fsimage中没有记录块所对应DataNode，这是因为DataNode要定时上报数据块信息，这是为了出现节点失联时能够及时更新。

Edits文件也不能直接打开，要使用 `hdfs oev -p XML -i edits_000000000000000012-000000000000000013 -o /opt/module/hadoop-2.7.2/edits.xml` 来将文件转换为xml文件输出到hadoop的安装目录下。

checkpoint时间设置

CheckPoint默认每隔一小时执行一次，这个默认配置可以在hdfs-default.xml中：

```
<property>
  <name>dfs.namenode.checkpoint.period</name>
  <value>3600</value>
</property>
```

一分钟检查一次操作次数，当操作次数达到1百万时，checkpoint执行一次：

```

<property>
  <name>dfs.namenode.checkpoint.txns</name>
  <value>1000000</value>
<description>操作动作次数</description>
</property>

<property>
  <name>dfs.namenode.checkpoint.check.period</name>
  <value>60</value>
<description> 1分钟检查一次操作次数</description>
</property >

```

NameNode故障处理

主要是利用SecondaryNameNode有NameNode的数据，将SecondaryNameNode中的数据拷贝到NameNode中。

集群安全模式

NameNode刚启动时，要合并内存中镜像文件和编辑日志，再重新生成新的镜像文件和空的编辑日志，这个过程一直处于安全模式，此时NameNode的文件系统对于客户端是只读的，因为合并期间进行写操作可能会导致出错。

在系统的正常操作期间，NameNode会在内存中保留所有块位置的映射信息。在安全模式下，各个DataNode会向NameNode发送最新的块列表信息，NameNode了解到足够多的块位置信息之后，即可高效运行文件系统。

如果满足“最小副本条件”，NameNode会在30秒钟之后就退出安全模式。所谓的最小副本条件指的是在整个文件系统中99.9%的块满足最小副本级别（默认值：dfs.replication.min=1），也就是绝大多数文件已经有了至少一个备份，在启动一个刚刚格式化的HDFS集群时，因为系统中还没有任何块，所以NameNode不会进入安全模式。

相关命令：

- (1) bin/hdfs dfsadmin -safemode get (功能描述：查看安全模式状态，HDFS中的overview页面也可以查看safemode是否开启。)
- (2) bin/hdfs dfsadmin -safemode enter (功能描述：进入安全模式状态)
- (3) bin/hdfs dfsadmin -safemode leave (功能描述：离开安全模式状态)
- (4) bin/hdfs dfsadmin -safemode wait (功能描述：等待安全模式状态，这个命令一般用于执行脚本时，脚本执行到该命令就进入阻塞状态等待安全模式，系统进入安全模式后继续执行)

NameNode多目录配置

NameNode的本地目录可以配置成多个，且每个目录存放内容相同，增加了一定的可靠性。

配置时要修改配置文件hdfs-site.xml，故首先要停止集群。

1、首先在布置NameNode的节点关闭yarn和hdfs：

在布置NameNode的节点执行：`sbin/stop-dfs.sh`

在布置ResourceManager的节点执行：`sbin/stop-yarn.sh`，然后删除数据目录：`rm -rf data/ logs/`

其他节点也要删除数据目录，然后修改hdfs-site.xml：

```

<property>
    <name>dfs.namenode.name.dir</name>
    <value>file://${hadoop.tmp.dir}/dfs/name1,file://${hadoop.tmp.dir}/dfs/name2</value>
</property>

```

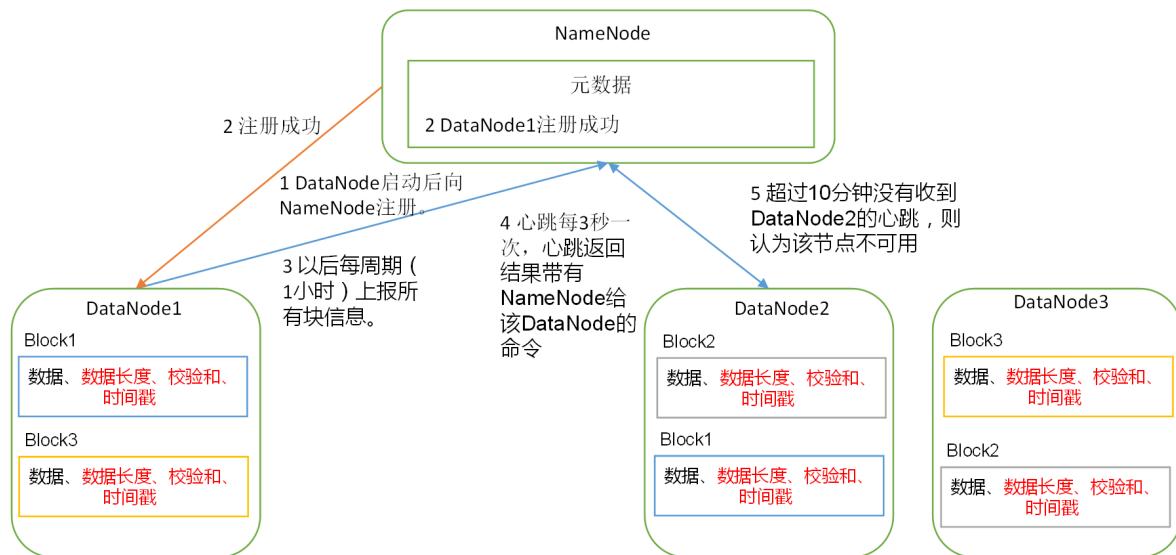
将目录位置设置成两个。然后将配置文件分发到各服务器：`xsync hdfs-site.xml`

2、格式化NameNode，只需要在布置NameNode的节点格式化即可：`bin/hdfs namenode -format`，然后启动集群：`sbin/start-dfs.sh`、`start-yarn.sh`。

然后在目录`/opt/module/hadoop-2.7.2/data/tmp/dfs/name1`和`name2`中就有完全相同的本地目录了。

DataNode

DataNode工作机制



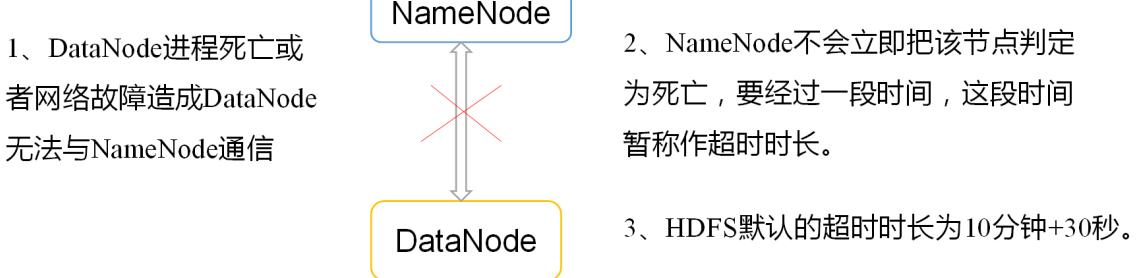
- 1) 一个数据块在DataNode上以文件形式存储在磁盘上，包括两个文件，一个是数据本身，一个是元数据包括数据块的长度，块数据的校验和，以及时间戳。
- 2) DataNode启动后向NameNode注册，通过后，周期性（1小时）的向NameNode上报所有的块信息。
- 3) 心跳是每3秒一次，心跳返回结果带有NameNode给该DataNode的命令如复制块数据到另一台机器，或删除某个数据块。如果超过10分钟没有收到某个DataNode的心跳，则认为该节点不可用。
- 4) 集群运行中可以安全加入和退出一些机器。

数据完整性

保证数据完整性的原理：传输前用校验算法计算出crc校验位，传输后用同样的算法计算，对比前后校验位是否相同。

当DataNode读取Block的时候，它会计算CheckSum，与block中的校验位对比。Client读取DataNode上的Block时也会进行校验。文件在HDFS中创建完成后，也会周期性的检验校验位。

掉线时限参数设置



4、如果定义超时时间为TimeOut，则超时时长的计算公式为：

$\text{TimeOut} = 2 * \text{dfs.namenode.heartbeat.recheck-interval} + 10 * \text{dfs.heartbeat.interval}$

而默认的`dfs.namenode.heartbeat.recheck-interval`大小为5分钟，`dfs.heartbeat.interval`默认为3秒。

要设置的参数主要有两个，都在`hdfs-site.xml`中，前者单位为毫秒，后者单位为秒。

```

<property>
    <name>dfs.namenode.heartbeat.recheck-interval</name>
    <value>300000</value>
</property>
<property>
    <name>dfs.heartbeat.interval</name>
    <value>3</value>
</property>

```

服役新数据节点

随着公司业务的增长，数据量越来越大，原有的数据节点的容量已经不能满足存储数据的需求，需要在原有集群基础上动态添加新的数据节点。具体步骤：

- 1、克隆一台新的虚拟机，配置好主机名和IP地址。
- 2、删除原来HDFS文件系统留存的文件（`/opt/module/hadoop-2.7.2/data`和`log`）。
- 3、source一下配置文件：`source /etc/profile`。
- 4、在新节点启动datanode：`sbin/hadoop-daemon.sh start datanode`。

启动nodemanager：`sbin/yarn-daemon.sh start nodemanager`。

如果数据不平衡，可以用以下命令来平衡集群：`sbin/start-balancer.sh`

添加白名单

添加到白名单的主机节点，都允许访问NameNode，不在白名单的主机节点，都会被退出。具体步骤：

- 1、在NameNode的`/opt/module/hadoop-2.7.2/etc/hadoop`目录下创建`dfs.hosts`文件，添加主机名称，如：

```

hadoop102
hadoop103
hadoop104

```

这就是白名单。

- 2、然后修改配置文件`hdfs-site.xml`：

```
<property>
    <name>dfs.hosts</name>
    <value>/opt/module/hadoop-2.7.2/etc/hadoop/dfs.hosts</value>
</property>
```

然后将配置文件分发到各节点。

3、刷新NameNode：`hdfs dfsadmin -refreshNodes`

更新ResourceManager节点：`yarn rmadmin -refreshNodes`

添加黑名单

在黑名单上面的主机都会被强制退出。具体步骤：

1、在NameNode的`/opt/module/hadoop-2.7.2/etc/hadoop`目录下创建`dfs.hosts.exclude`文件：

```
hadoop105
```

写入即将添加进黑名单的主机名。

2、在NameNode的`hdfs-site.xml`配置文件中增加`dfs.hosts.exclude`属性，然后分发到其他节点。

```
<property>
    <name>dfs.hosts.exclude</name>
    <value>/opt/module/hadoop-2.7.2/etc/hadoop/dfs.hosts.exclude</value>
</property>
```

3、刷新NameNode、刷新ResourceManager：

`hdfs dfsadmin -refreshNodes`、`yarn rmadmin -refreshNodes`

检查Web浏览器，退役节点的状态为decommission in progress（退役中），说明数据节点正在复制块到其他节点，等待退役节点状态为decommissioned（所有块已经复制完成），停止该节点及节点资源管理器。注意：如果副本数是3，服役的节点小于等于3，是不能退役成功的，需要修改副本数后才能退役。

datanode多目录配置

DataNode也可以配置成多个目录，每个目录存储的数据不一样。即：数据不是副本。

修改`hdfs-site.xml`：

```
<property>
    <name>dfs.datanode.data.dir</name>
    <value>file://${hadoop.tmp.dir}/dfs/data1,file://${hadoop.tmp.dir}/dfs/data2</value>
</property>
```

HDFS 2.X新特性

集群间数据拷贝distcp

两个远程主机之间的文件复制可以用scp，有时需要用到集群间数据拷贝，一般发生在从测试集群到生产集群之间的数据迁徙：

```
bin/hadoop distcp hdfs://hadoop102:9000/user/atguigu/hello.txt  
hdfs://hadoop103:9000/user/atguigu/hello.txt
```

小文件归档

HDFS存储小文件是有弊端的，再小的文件也是按块存储，块在namenode中会占用内存，大量的小文件会使系统运行效率降低。（但文件占用的磁盘空间和块大小无关，即使块是128M，但占用的磁盘空间和文件大小一样）

解决小文件问题的手段之一就是归档，将多个小文件合并成一个归档文件，能在单独访问小文件的同时将信息整体记录在namenode上，减少了namenode内存的占用。归档的具体步骤：

1、需要启动YARN进程：`start-yarn.sh`

2、把`/user/atguigu/input`目录里面的所有文件归档成一个叫`input.har`的归档文件，并把归档后文件存储到`/user/atguigu/output`路径下，输出路径对应文件夹必须设置成不存在：

```
bin/hadoop archive -archiveName input.har -p /user/atguigu/input  
/user/atguigu/output
```

3、查看归档后的小文件：`hadoop fs -lsr har:///user/atguigu/output/input.har`

4、解归档：`hadoop fs -cp har:/// user/atguigu/output/input.har/* /user/atguigu`

回收站

开启回收站功能后，删除文件后，不超过存活时间时可以将文件恢复。设置回收站功能时，有两个关键参数，一个是`fs.trash.interval`，它是文件的存活时间，默认为0（文件删除后立即过期），另一个是`fs.trash.checkpoint.interval`，它是检查回收站的间隔时间，默认为0（为0相当于该值和存活时间相同），它主要是检查回收站文件有没有到存活时间。要求设置时`fs.trash.checkpoint.interval<=fs.trash.interval`。

设置步骤：

1、修改`core-site.xml`，设置存活时间为1分钟，这里是为了便于测试，实际开发时这个时间可以设置的长一点：

```
<property>  
    <name>fs.trash.interval</name>  
    <value>1</value>  
</property>  
  
<property>  
    <name>hadoop.http.staticuser.user</name>  
    <value>root</value>  
</property>
```

后面的配置是为了设置进入回收站的用户名，默认是`dr.who`，不配置就无法进入回收站。

2、查看回收站：`/user/atguigu/.Trash/`

3、恢复回收站数据：`hadoop fs -mv /user/atguigu/.Trash/Current/user/atguigu/input /user/atguigu/input`，相当于把文件剪切到实际目录中。

4、清空回收站：`hadoop fs -expunge`，这里清空后还会留下一个很小的标记文件。

快照管理

快照是对目录做一个备份，它不会进行全量备份，只会记录源文件的变化。

- (1) `hdfs dfsadmin -allowSnapshot 路径` (功能描述：开启指定目录的快照功能)
- (2) `hdfs dfsadmin -disallowSnapshot 路径` (功能描述：禁用指定目录的快照功能，默认是禁用)
- (3) `hdfs dfs -createSnapshot 路径` (功能描述：对目录创建快照)
- (4) `hdfs dfs -createSnapshot 路径 名称` (功能描述：指定名称创建快照)
- (5) `hdfs dfs -renameSnapshot 路径 旧名称 新名称` (功能描述：重命名快照)
- (6) `hdfs lsSnapshottableDir` (功能描述：列出当前用户所有可快照目录)
- (7) `hdfs snapshotDiff 路径1 路径2` (功能描述：比较两个快照目录的不同之处)
- (8) `hdfs dfs -deleteSnapshot <path> <snapshotName>` (功能描述：删除快照)

MapReduce

MapReduce概述

MapReduce定义

MapReduce是一个分布式运算程序的编程框架，是用户开发“基于Hadoop的数据分析应用”的核心框架。

MapReduce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，并发运行在一个Hadoop集群上。

MapReduce优缺点

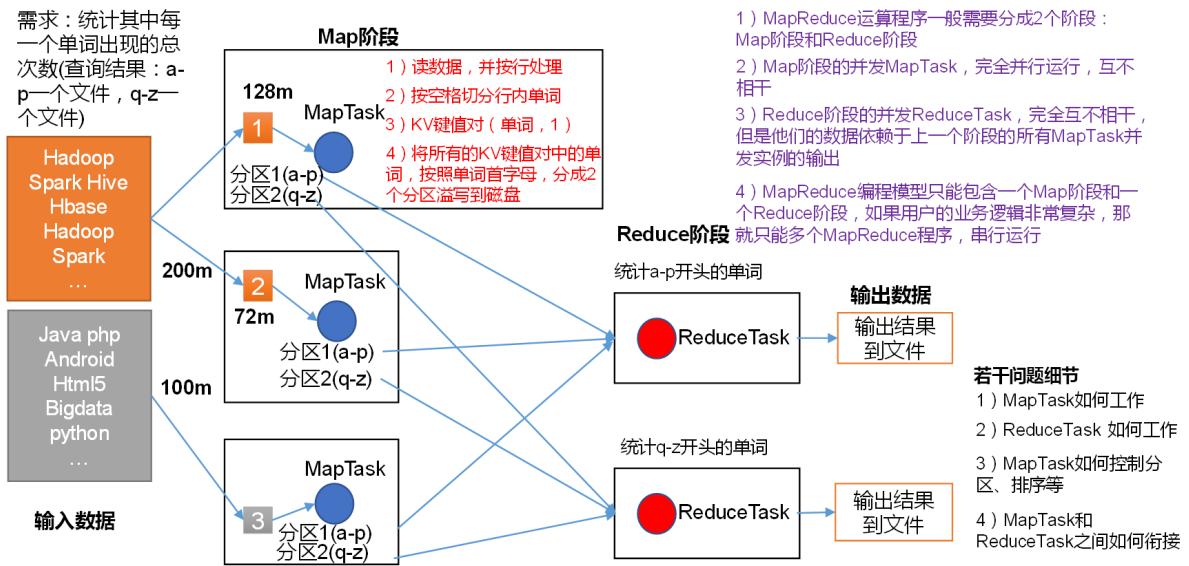
优点：

- 1、易于编程，只要实现一些借口，就能像编写串行程序一样编写分布式程序。
- 2、拓展性好，单纯的增加节点数能有效提升计算能力。
- 3、高容错性，节点的失效不影响任务执行的结果，它可以将计算任务自动转移到另一个节点上。
- 4、适合PB级以上的海量数据的离线处理

缺点：

- 1、不擅长实时计算，很难在毫秒或者秒级返回结果。
- 2、不擅长流式计算，输入数据集只能是静态的，不能是动态的。
- 3、不擅长有向图计算，不适合处理多个程序存在依赖关系，后一个应用程序的输入为前一个的输出，这种把输出结果写入磁盘的任务会导致计算性能低下。

MapReduce的核心思想



- 1) 分布式的运算程序往往需要分成至少2个阶段。将文件分块发到各节点，然后进行MapTask。
- 2) 第一个阶段的MapTask并发实例，完全并行运行，互不相干。MapTask形成KV对(单词, 1)，然后把结果分成两个分区写入磁盘，注意输出结果这里次数是1，因为还没有到合并阶段。
- 3) 第二个阶段的ReduceTask并发实例互不相干，但是他们的数据依赖于上一个阶段的所有MapTask并发实例的输出。将磁盘中的各分区分别统计，将输出写入文件。
- 4) MapReduce编程模型只能包含一个Map阶段和一个Reduce阶段，如果用户的业务逻辑非常复杂，那就只能多个MapReduce程序，串行运行。

MapReduce三类实例进程

MrAppMaster：负责整个程序的过程调度及状态协调。

MapTask：负责Map阶段的整个数据处理流程。

ReduceTask：负责Reduce阶段的整个数据处理流程。

数据序列化类型

在MapReduce中，有很多基本数据类型被封装：

Java类型	Hadoop Writable类型
boolean	BooleanWritable
byte	ByteWritable
int	IntWritable
float	FloatWritable
long	LongWritable
double	DoubleWritable
String	Text
map	MapWritable
array	ArrayWritable

MapReduce编程规范

1. Mapper阶段

- (1) 用户自定义的Mapper要继承自己的父类
- (2) Mapper的输入数据是KV对的形式 (KV的类型可自定义)
- (3) Mapper中的业务逻辑写在map()方法中
- (4) Mapper的输出数据是KV对的形式 (KV的类型可自定义)
- (5) map()方法 (MapTask进程) 对每一个<K,V>调用一次

2. Reducer阶段

- (1) 用户自定义的Reducer要继承自己的父类
- (2) Reducer的输入数据类型对应Mapper的输出数据类型，也是KV
- (3) Reducer的业务逻辑写在reduce()方法中
- (4) ReduceTask进程对每一组相同k的<k,v>组调用一次reduce()方法

3. Driver阶段

相当于YARN集群的客户端，用于提交我们整个程序到YARN集群，提交的是封装了MapReduce程序相关运行参数的job对象。

Wordcount案例运行

首先准备maven和hadoop环境、java环境，然后开始创建类：

WordcountMapper类

```
package hadoop1;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class wordcountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    Text k = new Text();
    IntWritable v = new IntWritable(1);

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        // 1 获取一行
        String line = value.toString();

        // 2 切割
        String[] words = line.split(" ");

        // 3 输出
        for (String word : words) {
```

```

        k.set(word);
        context.write(k, v);
    }
}

}

```

mapper泛型的四个参数分别代表输入的键和值，输出的键和值。输入的键是文件的偏移量，输入的值是一行的文本，输出的键是文本，输出的值是文本出现的次数。重写map方法，方法的参数分别为输入的键和值，以及context，提取出一行，然后切割，吧文本装入text中，次数为默认的1，最后都装入context中。为了避免重复创建对象，把一些对象提到方法外。读取一行执行一次map方法，都执行完之后reduce。经过map后的数据都是<单词，1>。

用户根据业务需求实现mapper中三个方法：主要逻辑map()、初始化setup()、最后执行cleanup()

WordcountReducer类

```

package hadoop1;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordcountReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
    int sum;
    IntWritable v = new IntWritable();

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {

        // 1 累加求和
        sum = 0;
        for (IntWritable count : values) {
            sum += count.get();
        }

        // 2 输出
        v.set(sum);
        context.write(key, v);
    }
}

```

Reducer泛型的四个参数是reduce的输入键（单词）和值（对应这个单词的多个词频）、输出值的键（单词）和值（对该单词的最终词频）。reduce方法的三个参数分别是输入键和值、context，最后的结果要放到context中，词频统计通过遍历values累加。

和mapper一样，用户根据业务需求实现Reducer中三个方法：reduce()、setup()、cleanup()

Driver驱动类：WordcountDriver

```

package hadoop1;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class WordcountDriver {
    public static void main(String[] args) throws IOException,
    ClassNotFoundException, InterruptedException {

        // 1 获取配置信息以及封装任务
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 设置jar加载路径
        job.setJarByClass(WordcountDriver.class);

        // 3 设置map和reduce类
        job.setMapperClass(WordcountMapper.class);
        job.setReducerClass(WordcountReducer.class);

        // 4 设置map输出
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        // 5 设置最终输出kv类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 6 设置输入和输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 7 提交，传入true提交完成会打印一些信息
        boolean result = job.waitForCompletion(true);

        System.exit(result ? 0 : 1);
    }
}

```

然后就可以运行该main方法了，运行时要指定运行两个运行参数，分别是数据文件路径和输出文件的目录：

C:\Users\greedy\Desktop\hello.txt C:\Users\greedy\Desktop\output，注意这里数据也可以是目录，输出目录必须是存在的。

在集群上测试

首先用maven打包jar包，在pom.xml中引入：

```

<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.3.2</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-assembly-plugin </artifactId>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
                <archive>
                    <manifest>
                        <mainClass>com.atguigu.mr.WordcountDriver</mainClass>
                    </manifest>
                </archive>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

mainClass处要换成主驱动类。然后点击maven projects->项目名->Lifecycle->package。在项目目录的target下就会产生打包好的jar包，这里选择没有依赖的jar包即可运行，传入linux中后运行：

```

hadoop jar wc.jar com.atguigu.wordcount.WordcountDriver /user/atguigu/input
/usr/atguigu/output 第三个参数是jar包路径，然后跟主驱动类名，接下来是输入值和输出值，这里输出目录也不能为空。

```

Hadoop序列化

序列化概论

序列化就是把内存中的对象，转换成字节序列（或其他数据传输协议）以便于存储到磁盘（持久化）和网络传输。反序列化就是将收到字节序列（或其他数据传输协议）或者是磁盘的持久化数据，转换成内存中的对象。为了方便传输和持久化，需要序列化反序列化对象。

由于java序列化功能比较重量级，故使用hadoop序列化，它比较紧凑、速度快、可扩展（随着通信协议的升级而升级）、支持多语言的交互。

自定义bean对象Writable

具体实现bean对象序列化步骤如下：

- 1、必须实现Writable接口
- 2、反序列化时，需要反射调用空参构造函数，所以必须有空参构造
- 3、重写序列化方法write和反序列化方法readFields，注意序列化方法和反序列化方法的顺序必须一致。

序列化案例

需求分析

统计每一个手机号耗费的总上行流量、下行流量、总流量，注意只需要三列数据，且需要叠加相同手机号的数据，具体文件如下：

1	13736230513	192.196.100.1	www.atguigu.com	2481	24681	200
2	13846544121	192.196.100.2		264 0	200	
3	13956435636	192.196.100.3		132 1512	200	
4	13966251146	192.168.100.1		240 0	404	
5	18271575951	192.168.100.2	www.atguigu.com	1527	2106	200
6	84188413	192.168.100.3	www.atguigu.com	4116	1432	200
7	13590439668	192.168.100.4		1116	954	200
8	15910133277	192.168.100.5	www.hao123.com	3156	2936	200
9	13729199489	192.168.100.6		240 0	200	
10	13630577991	192.168.100.7	www.shouhu.com	6960	690	200
11	15043685818	192.168.100.8	www.baidu.com	3659	3538	200
12	15959002129	192.168.100.9	www.atguigu.com	1938	180	500
13	13560439638	192.168.100.10		918 4938	200	
14	13470253144	192.168.100.11		180 180	200	
15	13682846555	192.168.100.12	www.qq.com	1938	2910	200
16	13992314666	192.168.100.13	www.gaga.com	3008	3720	200
17	13509468723	192.168.100.14	www.qinghua.com	7335	110349	404
18	18390173782	192.168.100.15	www.sogou.com	9531	2412	200
19	13975057813	192.168.100.16	www.baidu.com	11058	48243	200
20	13768778790	192.168.100.17		120 120	200	
21	13568436656	192.168.100.18	www.alibaba.com	2481	24681	200
22	13568436656	192.168.100.19		1116	954	200

第二列是手机号，倒数第二和第三列是上行流量、下行流量。

期望输出格式：

13560436666	1116	954	2070
手机号码	上行流量	下行流量	总流量

对于这个需求来说，map输出数据的键对应手机号，值必须同时对应上行流量和下行流量，此时应该把值设置为bean对象。

bean对象

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;

// 1 实现Writable接口
```

```
public class FlowBean implements Writable{  
  
    //上行流量和下行流量、总流量  
    private long upFlow;  
    private long downFlow;  
    private long sumFlow;  
  
    //2 反序列化时，需要反射调用空参构造函数，所以必须有  
    public FlowBean() {  
        super();  
    }  
  
    public FlowBean(long upFlow, long downFlow) {  
        super();  
        this.upFlow = upFlow;  
        this.downFlow = downFlow;  
        this.sumFlow = upFlow + downFlow;  
    }  
  
    //3 写序列化方法  
    @Override  
    public void write(DataOutput out) throws IOException {  
        out.writeLong(upFlow);  
        out.writeLong(downFlow);  
        out.writeLong(sumFlow);  
    }  
  
    //4 反序列化方法  
    //5 反序列化方法读顺序必须和写序列化方法的写顺序必须一致，序列化是按照先进先出的原则进行的  
    @Override  
    public void readFields(DataInput in) throws IOException {  
        this.upFlow = in.readLong();  
        this.downFlow = in.readLong();  
        this.sumFlow = in.readLong();  
    }  
  
    // 6 编写toString方法，方便后续直接用\t切割  
    @Override  
    public String toString() {  
        return upFlow + "\t" + downFlow + "\t" + sumFlow;  
    }  
  
    public long getUpFlow() {  
        return upFlow;  
    }  
  
    public void setUpFlow(long upFlow) {  
        this.upFlow = upFlow;  
    }  
  
    public long getDownFlow() {  
        return downFlow;  
    }  
  
    public void setDownFlow(long downFlow) {  
        this.downFlow = downFlow;  
    }  
}
```

```

public long getSumFlow() {
    return sumFlow;
}

public void setSumFlow(long sumFlow) {
    this.sumFlow = sumFlow;
}
}

```

Mapper类

```

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean>{

    FlowBean v = new FlowBean();
    Text k = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        // 1 获取一行
        String line = value.toString();

        // 2 切割字段
        String[] fields = line.split("\t");

        // 3 封装对象
        // 取出手机号码
        String phoneNum = fields[1];

        // 取出上行流量和下行流量，因为每一行的内容个数不一致，所以这里采用从后向前取的方式
        long upFlow = Long.parseLong(fields[fields.length - 3]);
        long downFlow = Long.parseLong(fields[fields.length - 2]);

        k.set(phoneNum);
        v.set(downFlow, upFlow);

        // 4 写出
        context.write(k, v);
    }
}

```

Reducer类

```

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {

    @Override

```

```

protected void reduce(Text key, Iterable<FlowBean> values, Context
context) throws IOException, InterruptedException {

    long sum_upFlow = 0;
    long sum_downFlow = 0;

    // 1 遍历所用bean，将其中的上行流量，下行流量分别累加，相同号码如果出现在多行就靠这个
    循环累加
    for (FlowBean flowBean : values) {
        sum_upFlow += flowBean.getUpFlow();
        sum_downFlow += flowBean.getDownFlow();
    }

    // 2 封装对象
    FlowBean resultBean = new FlowBean(sum_upFlow, sum_downFlow);

    // 3 写出
    context.write(key, resultBean);
}
}

```

Driver驱动类

```

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FlowsumDriver {

    public static void main(String[] args) throws IllegalArgumentException,
    IOException, ClassNotFoundException, InterruptedException {

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[] { "e:/input/inputflow", "e:/output1" };

        // 1 获取配置信息，或者job对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 6 指定本程序的jar包所在的本地路径
        job.setJarByClass(FlowsumDriver.class);

        // 2 指定本业务job要使用的mapper/Reducer业务类
        job.setMapperClass(FlowCountMapper.class);
        job.setReducerClass(FlowCountReducer.class);

        // 3 指定mapper输出数据的kv类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

        // 4 指定最终输出的数据的kv类型
        job.setOutputKeyClass(Text.class);
    }
}

```

```

        job.setOutputValueClass(FlowBean.class);

        // 5 指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 7 将job中配置的相关参数, 以及job所用的java类所在的jar包, 提交给yarn去运行
        boolean result = job.waitForCompletion(true);
        System.exit(result ? 0 : 1);
    }
}

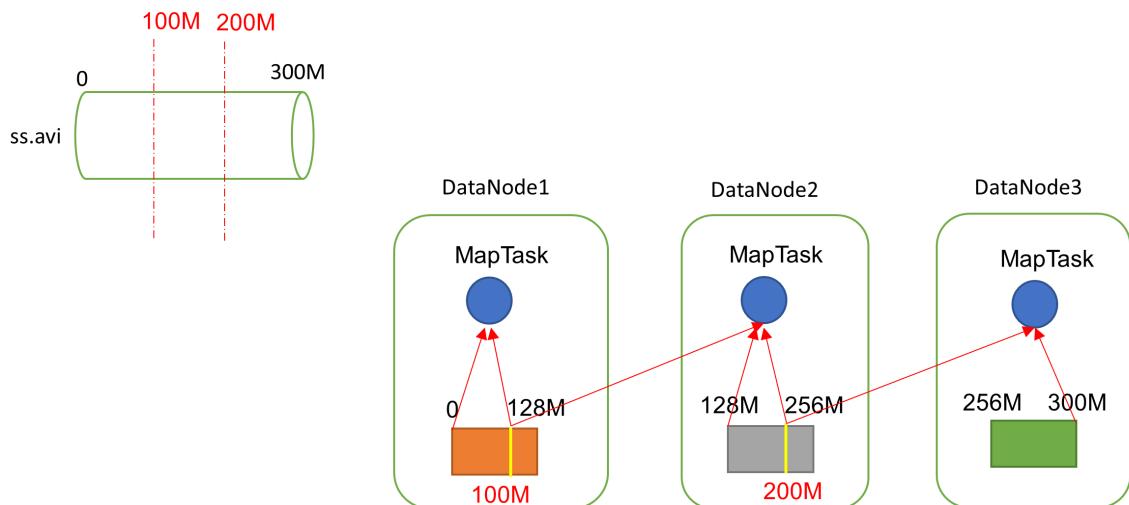
```

InputFormat数据输入

数据切片的概念

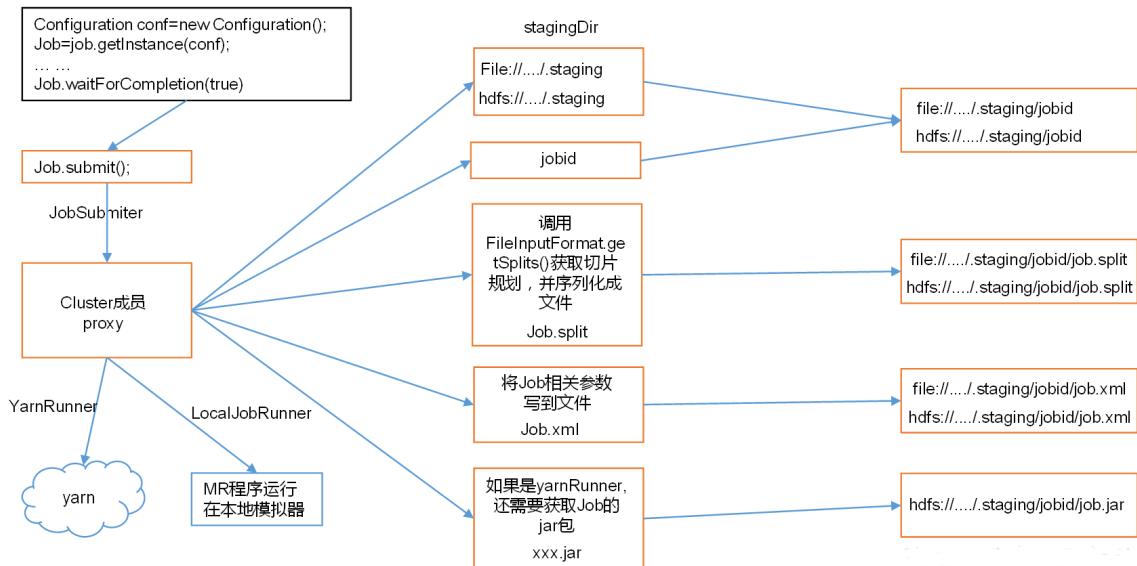
数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。

300M的文件设置切片大小为100M，不会三个节点一个存100M，而是根据block的大小切分文件，数据切片和maptask相关，第一个100M会被第一个maptask处理，第二个100M会被第二个maptask处理，每一个切片都会交给对应maptask处理，故切片数决定了并行程度，但如果切片大小>block大小，就会导致节点之间IO频繁从而降低运行效率，默认切片大小=block大小。



同时处理多个文件时，切片操作针对的是单个文件而不是整体数据，多个小文件会导致切片数大量增加。

Job提交流程



首先调用waitForCompletion方法提交job，会根据程序运行在集群还是运行在本地调用不同的Runner，然后进行stagingDir，也就是创建提交资源的路径（上面是本地，下面是集群模式），同时为job分配一个id，保证job的唯一性，两者拼接成完整的路径。然后获取切片信息，并将切片信息文件写入刚创建的路径中。获取job相关参数，也写入之前的路径中。如果程序运行在集群，就要获取job的jar包，将其放在hdfs中。

FileInputFormat切片过程

- (1) 程序先找到你数据存储的目录。
- (2) 开始遍历处理（规划切片）目录下的每一个文件
- (3) 遍历第一个文件ss.txt
 - a) 获取文件大小fs.sizeOf(ss.txt)
 - b) 计算切片大小
 $computeSplitSize(\max(\minSize, \min(maxSize, blockSize))) = blockSize = 128M$
 - c) 默认情况下，切片大小=blocksize
 - d) 开始切，形成第1个切片：ss.txt—0:128M 第2个切片ss.txt—128:256M 第3个切片ss.txt—256M:300M
(每次切片时，都要判断切完剩下的部分是否大于块的1.1倍，不大于1.1倍就划分一块切片)
 - e) 将切片信息写到一个切片规划文件中
 - f) 整个切片的核心过程在getSplit()方法中完成
 - g) **InputSplit只记录了切片的元数据信息**，比如起始位置、长度以及所在的节点列表等。
- (4) 提交切片规划文件到YARN上，YARN上的MrAppMaster就可以根据切片规划文件计算开启MapTask个数。

本地模式block默认是32M，切片也是32M。（每次切片时都要判断剩余是否大于块的1.1倍，如果不大于就只划分一块切片）

FileInputFormat切片参数设置

(1) 源码中计算切片大小的公式

```
Math.max(minSize, Math.min(maxSize, blockSize));  
mapreduce.input.fileinputformat.split.minsize=1 默认值为1  
mapreduce.input.fileinputformat.split.maxsize= Long.MAXValue 默认值Long.MAXValue  
因此，默认情况下，切片大小=blocksize。
```

(2) 切片大小设置

maxsize (切片最大值) : 参数如果调得比blockSize小，则会让切片变小，而且就等于配置的这个参数的值。

minsize (切片最小值) : 参数调的比blockSize大，则可以让切片变得比blockSize还大。

(3) 获取切片信息API

```
// 获取切片的文件名称  
String name = inputSplit.getPath().getName();  
// 根据文件类型获取切片信息  
FileSplit inputSplit = (FileSplit) context.getInputSplit();
```

CombineTextInputFormat切片机制

框架默认的TextInputFormat切片机制是对任务按文件规划切片，不管文件多小，都会是一个单独的切片，都会交给一个MapTask，这样如果有大量小文件，就会产生大量的MapTask，处理效率极其低下。CombineTextInputFormat用于小文件过多的场景，它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个MapTask处理。

首先应该设置切片最大值，CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);也就是4M，实际参数的大小根据具体小文件的大小决定。

生成切片过程包括：虚拟存储过程和切片过程二部分：

setMaxInputSplitSize值为4M

虚拟存储过程			切片过程
a.txt	1.7M	1.7M<4M 划分一块	(a) 判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。
b.txt	5.1M	5.1M>4M 但是小于2*4M 划分二块 块1=2.55M; 块2=2.55M	
c.txt	3.4M	3.4M<4M 划分一块	(b) 如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。
d.txt	6.8M	6.8M>4M 但是小于2*4M 划分二块 块1=3.4M; 块2=3.4M	
最终存储的文件			最终会形成3个切片，大小分别为：
1.7M			(1.7+2.55) M, (2.55+3.4) M, (3.4+3.4) M
2.55M			
3.4M			
3.4M			
3.4M			

上图中表示有4个文件，切片最大值为4M，在虚拟存储阶段如果文件大小<4M，那么就直接划分一块，如果大于4M但不大于8M，但均分后文件小于4M则均分成两份。虚拟存储的逻辑是，将输入目录下所有文件大小，依次和设置的setMaxInputSplitSize值比较，如果不大于设置的最大值，逻辑上划分一个块。如果输入文件大于设置的最大值且大于两倍最大值，那么以最大值切割一块；当剩余数据大小超过设置的最大值且不大于最大值2倍，此时将文件均分成2个虚拟存储块（防止出现太小切片）。如果文件大小为8.04M，那么就会分为4M/2.02M/2.02M三个。

在切片过程中处理切分后的文件，按顺序合并文件，如果合并后文件大小小于4M则将其合并为一个切片，否则就划分成两个切片，上例中最后得到3个切片。

CombineTextInputFormat案例

将输入的大量小文件合并成一个切片统一处理，输入数据为一个目录，目录下有多个小文件。程序运行时可以通过日志观察切片数，number of splits，想要以CombineTextInputFormat模式运行需要在驱动类中添加：

```
// 如果不设置InputFormat，它默认用的是TextInputFormat.class  
job.setInputFormatClass(CombineTextInputFormat.class);  
  
// 虚拟存储切片最大值设置4m  
CombineTextInputFormat.setMaxInputSplitsize(job, 4194304);
```

如果想合并的更多，就应该把切片最大值继续调大。

FileInputFormat实现类

针对不同的文件类型，FileInputFormat负责读取文件转换成数据。FileInputFormat常见的接口实现类包括：TextInputFormat、KeyValueTextInputFormat、NLineInputFormat、CombineTextInputFormat和自定义InputFormat等。

TextInputFormat

TextInputFormat是默认的FileInputFormat实现类。按行读取每条记录。键是存储该行在整个文件中的起始字节偏移量，LongWritable类型。值是这行的内容，不包括任何行终止符（换行符和回车符），Text类型。

文件：

```
Rich learning form  
Intelligent learning engine  
Learning more convenient  
From the real demand for more close to the enterprise
```

读出的数据：

```
(0,Rich learning form)  
(19,Intelligent learning engine)  
(47,Learning more convenient)  
(72,From the real demand for more close to the enterprise)
```

KeyValueTextInputFormat

每一行均为一条记录，被分隔符分割为key，value。可以通过在驱动类中设置conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, "\t");来设定分隔符。默认分隔符是tab (\t)。以下是一个示例，输入是一个包含4条记录的分片。其中—>表示一个（水平方向的）制表符：

文件：

```
line1 -->Rich learning form  
line2 -->Intelligent learning engine  
line3 -->Learning more convenient  
line4 -->From the real demand for more close to the enterprise
```

读出的数据：

```
(line1,Rich learning form)
(line2,Intelligent learning engine)
(line3,Learning more convenient)
(line4,From the real demand for more close to the enterprise)
```

应用该格式读取需要在驱动类中加入：

```
// 设置切割符
conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, " ");
// 设置输入格式
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

NLineInputFormat

读到的键值对和TextInputFormat相同，切片机制有所不同。

如果使用NlineInputFormat，代表每个map进程处理的InputSplit不再按Block块去划分，而是按NlineInputFormat指定的行数N来划分。即输入文件的总行数/N=切片数，如果不整除，切片数=商+1。

应用该格式读取需要在驱动类中加入：

```
// 设置每个切片InputSplit中划分三条记录
NLineInputFormat.setNumLinesPerSplit(job, 3);

// 使用NLineInputFormat处理记录数
job.setInputFormatClass(NLineInputFormat.class);
```

自定义InputFormat

在企业开发中，Hadoop框架自带的InputFormat类型不能满足所有应用场景，需要自定义InputFormat来解决实际问题。自定义InputFormat步骤如下：

- (1) 自定义一个类继承FileInputFormat。
- (2) 改写RecordReader，实现一次读取一个完整文件封装为KV。
- (3) 在输出时使用SequenceFileOutPutFormat输出合并文件。

需求分析

输入数据为三个数据文件，输出文件为一个SequenceFile文件，将多个小文件合并成一个SequenceFile文件（SequenceFile文件是Hadoop用来存储二进制形式的key-value对的文件格式），SequenceFile里面存储着多个文件，存储的形式为文件路径+名称为key，文件内容为value。这是避免处理大量小文件的应对措施之一。

自定义InputFormat类

首先创建自定义InputFormat类，需要继承FileInputFormat类，然后重写isSplitable方法（返回一个布尔值，指定文件是否切片）和createRecordReader方法（返回一个RecordReader类）：

```

package hadoop2;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import java.io.IOException;

//text对应文件名称，BytesWritable为字节流。
public class wholeFileInputformat extends FileInputFormat<Text, BytesWritable>{

    @Override
    protected boolean isSplitable(JobContext context, Path filename) {
        return false;
    }

    @Override
    public RecordReader<Text, BytesWritable> createRecordReader(InputSplit
split, TaskAttemptContext context) throws IOException, InterruptedException {

        wholeRecordReader recordReader = new wholeRecordReader();
        recordReader.initialize(split, context);

        return recordReader;
    }
}

```

RecordReader类：

要设置该类的构造方法，且要重写nextKeyValue方法，在这个方法中提取要输入格式中的key和value，也就是Text和BytesWritable，执行时nextKeyValue会被执行多次，执行次数和具体返回值有关，但在本例中文件没有被切片，故只执行一次，第二次执行同一个对象的nextKeyValue方法时就会返回false，然后执行map。

```

package hadoop2;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

```

```
//泛型类型和wholeFileInputFormat相同。只要读取一个新文件，就会创建一个新的该类对象
public class wholeRecordReader extends RecordReader<Text, BytesWritable>{

    private Configuration configuration;
    //文件切片
    private Filesplit split;

    private boolean isProgress= true;
    private BytesWritable value = new BytesWritable();
    private Text k = new Text();

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context) throws
IOException, InterruptedException {
        //初始化将两个参数传到类中
        this.split = (Filesplit)split;
        configuration = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {

        if (isProgress) {

            // 1 定义缓存区，切片就是文件路径
            byte[] contents = new byte[(int)split.getLength()];

            FileSystem fs = null;
            FSDataInputStream fis = null;

            try {
                // 2 根据切片获取路径，再根据路径获取文件系统
                Path path = split.getPath();
                fs = path.getFileSystem(configuration);

                // 3 读取数据
                fis = fs.open(path);

                // 4 读取文件内容
                IOUtils.readFully(fis, contents, 0, contents.length);

                // 5 输出文件内容，最终吧切片的数据设置到value中
                value.set(contents, 0, contents.length);

            } // 6 获取文件路径及名称
            String name = split.getPath().toString();

            // 7 设置输出的key值，最终将文件路径及名称设置到key中
            k.set(name);

        } catch (Exception e) {

        }finally {
            IOUtils.closeStream(fis);
        }

        isProgress = false;
    }
}
```

```

        return true;
    }
    //这里返回false的意思是数据已经读完，不会再执行map了，这个方法只会执行一次，然后执行
    map方法。
    return false;
}

@Override
public Text getCurrentKey() throws IOException, InterruptedException {
    return k;
}

@Override
public BytesWritable getCurrentValue() throws IOException,
InterruptedException {
    return value;
}

@Override
public float getProgress() throws IOException, InterruptedException {
    return 0;
}

@Override
public void close() throws IOException {
}
}

```

nextKeyValue会被多次执行，下段代码是mapper类的run方法：

```

public void run(Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>.Context context) throws
IOException, InterruptedException {
    this.setup(context);

    try {
        while(context.nextKeyValue()) {
            this.map(context.getCurrentKey(), context.getCurrentValue(),
context);
        }
    } finally {
        this.cleanup(context);
    }
}

```

SequenceFileMapper类

mapper的泛型前两个类型就是输入类的输出，即Text和BytesWritable，结合需求可以发现map和reduce阶段几乎不需要其他逻辑，只需要传递输入值即可：

```

package hadoop2;

import java.io.IOException;
import org.apache.hadoop.io.BytesWritable;

```

```

import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class SequenceFileMapper extends Mapper<Text, BytesWritable, Text,
BytesWritable>{

    @Override
    protected void map(Text key, BytesWritable value, Context context) throws
IOException, InterruptedException {

        context.write(key, value);
    }
}

```

SequenceFileReducer类

```

package hadoop2;

import java.io.IOException;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class SequenceFileReducer extends Reducer<Text, BytesWritable, Text,
BytesWritable> {

    @Override
    protected void reduce(Text key, Iterable<BytesWritable> values, Context
context) throws IOException, InterruptedException {

        context.write(key, values.iterator().next());
    }
}

```

SequenceFileDriver类

这里注意要引入输入类WholeFileInputformat.class，把输出类设置为SequenceFileOutputFormat.class。

```

package hadoop2;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;

public class SequenceFileDriver {

```

```
public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {

    // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
    args = new String[] { "C:\\\\users\\\\greedy\\\\Desktop\\\\input",
    "C:\\\\users\\\\greedy\\\\Desktop\\\\output" };

    // 1 获取job对象
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf);

    // 2 设置jar包存储位置、关联自定义的mapper和reducer
    job.setJarByClass(SequenceFileDriver.class);
    job.setMapperClass(SequenceFileMapper.class);
    job.setReducerClass(SequenceFileReducer.class);

    // 7设置输入的inputFormat
    job.setInputFormatClass(WholeFileInputformat.class);

    // 8设置输出的outputFormat
    job.setOutputFormatClass(SequenceFileOutputFormat.class);

    // 3 设置map输出端的kv类型
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(BytesWritable.class);

    // 4 设置最终输出端的kv类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(BytesWritable.class);

    // 5 设置输入输出路径
    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // 6 提交job
    boolean result = job.waitForCompletion(true);
    System.exit(result ? 0 : 1);
}
```

shuffle

shuffle是map方法后，reduce方法前处理数据的流程。

partition分区

程序运行时默认将运行结果装入一个文件中，如果想将不同的结果输出到不同的文件则需要设置多个分区。在HashPartitioner类的getPartition方法中会返回对应的方法号，默认返回的是如下结果，也就是将数控制在int范围内再取余，numReduceTasks默认是1，也就是结果默认是0，默认的分区只有一个，分区号是0。

```

public class HashPartitioner<K, V> extends Partitioner<K, V> {

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}

```

自定义分区步骤：

(1) 自定义类继承Partitioner，重写getPartition()方法，如果现在有一个需求，手机号136、137、138、139开头都分别放到一个独立的4个文件中，其他开头的放到一个文件中，那么该方法就应该为：

```

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {

        // 1 获取电话号码的前三位
        String preNum = key.toString().substring(0, 3);

        int partition = 4;

        // 2 判断是哪个省
        if ("136".equals(preNum)) {
            partition = 0;
        } else if ("137".equals(preNum)) {
            partition = 1;
        } else if ("138".equals(preNum)) {
            partition = 2;
        } else if ("139".equals(preNum)) {
            partition = 3;
        }

        return partition;
    }
}

```

partitioner对应的泛型就是map的输出类型，和getPartition的参数相同，方法的返回值只能从0开始，0/1/2..

(2) 在Job驱动中，设置自定义Partitioner，还要根据自定义Partitioner的逻辑设置相应数量的ReduceTask：

```

// 指定自定义数据分区
job.setPartitionerClass(ProvincePartitioner.class);

// 同时指定相应数量的reduce task
job.setNumReduceTasks(5);

```

ReduceTask的数量和getPartition的结果数对结果的影响：

如果ReduceTask的数量> getPartition的结果数，则会多产生几个空的输出文件part-r-000xx；

如果1<ReduceTask的数量<getPartition的结果数，则有一部分分区数据无处安放，会Exception；

如果ReduceTask的数量=1，则不管MapTask端输出多少个分区文件，最终结果都交给这一个ReduceTask，最终也就只会产生一个结果文件 part-r-00000；

只有两者相同时才能设置分区成功。

排序概述

排序是MapReduce框架中最重要的操作之一。

MapTask和ReduceTask均会对数据按照key进行排序。该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。

对于MapTask，它会将处理的结果暂时放到环形缓冲区中，当环形缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次快速排序，并将这些有序数据溢写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行归并排序。

对于ReduceTask，它从每个MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则溢写磁盘上，否则存储在内存中。如果磁盘上文件数目达到一定阈值，则进行一次归并排序以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据溢写到磁盘上。当所有数据拷贝完毕后，ReduceTask统一对内存和磁盘上的所有数据进行一次归并排序。

要实现排序bean对象做为key传输，需要实现WritableComparable接口重写compareTo方法，就可以实现排序。

全排序

需求分析

输入的数据包括手机号，上行流量、下行流量和总流量，要求整合成一个文件，文件中要保留的数据有手机号、总流量，每条数据按照总流量大小从大到小排列。

输入数据：

```
13736230513 2481    24681    27162
13846544121 264 0    264
13956435636 132 1512    1644
13509468723 7335    110349    117684
. . . . .
```

输出数据：

```
13509468723 7335    110349    117684
13736230513 2481    24681    27162
13956435636 132 1512    1644
13846544121 264 0    264
. . . . .
```

FlowBean类

内部构造三个属性，分别是上行流量、下行流量和总流量，然后建立构造方法，创建序列化方法和反序列化方法，最后还要创建compareTo方法。注意要写toString方法，它决定了bean向文件写的格式。

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;

public class FlowBean implements WritableComparable<FlowBean> {
```

```
private long upFlow;
private long downFlow;
private long sumFlow;

// 反序列化时，需要反射调用空参构造函数，所以必须有
public FlowBean() {
    super();
}

public FlowBean(long upFlow, long downFlow) {
    super();
    this.upFlow = upFlow;
    this.downFlow = downFlow;
    this.sumFlow = upFlow + downFlow;
}

public void set(long upFlow, long downFlow) {
    this.upFlow = upFlow;
    this.downFlow = downFlow;
    this.sumFlow = upFlow + downFlow;
}

public long getSumFlow() {
    return sumFlow;
}

public void setSumFlow(long sumFlow) {
    this.sumFlow = sumFlow;
}

public long getUpFlow() {
    return upFlow;
}

public void setUpFlow(long upFlow) {
    this.upFlow = upFlow;
}

public long getDownFlow() {
    return downFlow;
}

public void setDownFlow(long downFlow) {
    this.downFlow = downFlow;
}

/**
 * 序列化方法
 * @param out
 * @throws IOException
 */
@Override
public void write(DataOutput out) throws IOException {
    out.writeLong(upFlow);
    out.writeLong(downFlow);
    out.writeLong(sumFlow);
}
```

```

    /**
     * 反序列化方法 注意反序列化的顺序和序列化的顺序完全一致
     * @param in
     * @throws IOException
     */
    @Override
    public void readFields(DataInput in) throws IOException {
        upFlow = in.readLong();
        downFlow = in.readLong();
        sumFlow = in.readLong();
    }

    @Override
    public String toString() {
        return upFlow + "\t" + downFlow + "\t" + sumFlow;
    }

    @Override
    public int compareTo(FlowBean bean) {

        int result;

        // 按照总流量大小，倒序排列
        if (sumFlow > bean.getSumFlow()) {
            result = -1;
        } else if (sumFlow < bean.getSumFlow()) {
            result = 1;
        } else {
            result = 0;
        }

        return result;
    }
}

```

Mapper类

mapper类的泛型设置为：输入类型为LongWritable（偏移量）和Text（一行的字符串），输出类型为FlowBean（流量bean）和Text（手机号），想要排序必须将bean设置为键。

```

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class FlowCountSortMapper extends Mapper<LongWritable, Text, FlowBean,
Text>{

    FlowBean bean = new FlowBean();
    Text v = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        // 1 获取一行
        String line = value.toString();

```

```

    // 2 截取
    String[] fields = line.split("\t");

    // 3 封装对象
    String phoneNbr = fields[0];
    long upFlow = Long.parseLong(fields[1]);
    Long downFlow = Long.parseLong(fields[2]);

    bean.set(upFlow, downFlow);
    v.set(phoneNbr);

    // 4 输出
    context.write(bean, v);
}
}

```

Reducer类

输出类型为Text（手机号）和FlowBean（流量bean），可能存在多个手机号拥有相同流量的情况，所以要遍历所有的values，然后用context去写。

```

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FlowCountSortReducer extends Reducer<FlowBean, Text, Text,
FlowBean>{

    @Override
    protected void reduce(FlowBean key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {

        // 循环输出，避免总流量相同情况
        for (Text text : values) {
            context.write(text, key);
        }
    }
}

```

Driver类

```

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FlowCountSortDriver {

    public static void main(String[] args) throws ClassNotFoundException,
IOException, InterruptedException {

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
    }
}

```

```

args = new String[]{"e:/output1", "e:/output2"};

// 1 获取配置信息，或者job对象实例
Configuration configuration = new Configuration();
Job job = Job.getInstance(configuration);

// 2 指定本程序的jar包所在的本地路径
job.setJarByClass(FlowCountSortDriver.class);

// 3 指定本业务job要使用的mapper/Reducer业务类
job.setMapperClass(FlowCountSortMapper.class);
job.setReducerClass(FlowCountSortReducer.class);

// 4 指定mapper输出数据的kv类型
job.setMapOutputKeyClass(FlowBean.class);
job.setMapOutputValueClass(Text.class);

// 5 指定最终输出的数据的kv类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FlowBean.class);

// 6 指定job的输入原始文件所在目录
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}

```

区内排序

输入数据包括手机号、上行流量、下行流量、总流量，要求按照手机号前三位不同将数据分到不同文件，同时每个文件内部按照总流量从大到小排列。

输入数据：

13509468723	7335	110349	117684
13975057813	11058	48243	59301
13568436656	3597	25635	29232
13736230513	2481	24681	27162
18390173782	9531	2412	11943
13630577991	6960	690	7650
15043685818	3659	3538	7197
13992314666	3008	3720	6728
15910133277	3156	2936	6092
13560439638	918	4938	5856
84188413	4116	1432	5548
13682846555	1938	2910	4848
18271575951	1527	2106	3633
15959002129	1938	180	2118
13590439668	1116	954	2070
13956435636	132	1512	1644
13470253144	180	180	360
13846544121	264	0	264

```
13966251146 240 0 240  
13768778790 120 120 240  
13729199489 240 0 240  
. . . . .
```

输出数据：

```
13630577991 6960 690 7650  
13682846555 1938 2910 4848
```

```
13736230513 2481 24681 27162  
13768778790 120 120 240  
13729199489 240 0 240
```

```
13846544121 264 0 264
```

```
13975057813 11058 48243 59301  
13992314666 3008 3720 6728  
13956435636 132 1512 1644  
13966251146 240 0 240
```

```
13509468723 7335 110349 117684  
13568436656 3597 25635 29232  
18390173782 9531 2412 11943  
15043685818 3659 3538 7197  
15910133277 3156 2936 6092  
. . . . .
```

解决方案就是分区的同时创建一个排序类实现WritableComparable。

Combiner合并

Combiner是重要的提高效率的组件，它独立于mapper和reduce存在，combiner在每一个maptask所在的节点运行，负责每一个MapTask的输出进行局部汇总，以减小网络传输量。在combiner运行结束后再执行reduce，Reducer是接收全局所有Mapper的输出结果，Combiner组件的父类就是Reducer。

Combiner能够应用的前提是不能影响最终的业务逻辑，举例来说，如果想计算多个数字的平均值，不能在每个mapper后求平均值然后最后再平均，这样会导致错误的结果：

Mapper	Reducer
3 5 7 ->(3+5+7)/3=5	(3+5+7+2+6)/5=23/5 不等于 (5+4)/2=9/2
2 6 ->(2+6)/2=4	

Combiner的输出kv和输入kv应该跟mapper和Reduce对应起来。

需求分析

典型的wordcount案例，统计文件中的词频。

输入数据：

```
banzhang ni hao
xihuan hadoop banzhang
banzhang ni hao
xihuan hadoop banzhang
```

输出数据：

```
banzhang 4
ni 2
hao 2
xihuan 2
Hadoop 2
```

按照原来的处理方式，mapper处理完数据输出的kv是<banzhang,1>，也就是说词频总是1没有进行合并，这种方式效率很低，如果能在maptask处就将同一个词的词频合并，就能减少大量网络开销。根据Combiner的原理来进行合并处理。

可以通过查看控制台的Combine input records和Combine output records查看是否进行combine。

WordcountCombiner类

自定义一个Combiner继承Reducer，重写Reduce方法，注意Reducer的泛型分别是mapper的输出kv和reduce的输入kv。

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordcountCombiner extends Reducer<Text, IntWritable, Text,
IntWritable>{

    IntWritable v = new IntWritable();

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {

        // 1 汇总
        int sum = 0;

        for(IntWritable value : values){
            sum += value.get();
        }

        v.set(sum);

        // 2 写出
        context.write(key, v);
    }
}
```

```
}
```

接收到同一个key的多个value，在combine阶段就将其累加，然后写入context中。

然后在驱动类中添加该combiner：

```
// 指定需要使用combiner，以及用哪个类作为combiner的逻辑  
job.setCombinerClass(wordcountCombiner.class);
```

还有一种更简单的方案，就是直接把reducer当做combiner引入，因为reducer和combiner的逻辑完全相同。

```
// 指定需要使用Combiner，以及用哪个类作为Combiner的逻辑  
job.setCombinerClass(wordcountReducer.class);
```

辅助排序和二次排序

需求分析

每行输入数据都有三个字段，分别是订单id，商品id，和商品价格，要求筛选出每个订单中价格最高的商品价格。

输入数据：

```
0000001 Pdt_01 222.8  
0000002 Pdt_05 722.4  
0000001 Pdt_02 33.8  
0000003 Pdt_06 232.8  
0000003 Pdt_02 33.8  
0000002 Pdt_03 522.8  
0000002 Pdt_04 122.4
```

输出数据：

```
1 222.8  
2 722.4  
3 232.8
```

首先想到的就是排序，要进行排序就要把要排序的内容设置为key，这里其实是要进行两次排序，输出数据是按照订单号排序的，其次数据内部还应该按照价格排序，然后取最高的价格输出，这种排序逻辑内包含两次的就被称为**二次排序**。

现在问题是如果按照上述的二次排序操作，输出数据会把所有数据都输出，而不是只输出三个，所以这里应该使用**辅助排序**来基于排序筛选掉不需要的key。

OrderBean类

需要一个bean类将订单号和价格封装在一起进行排序，在compareTo逻辑中进行了两次排序：

```
import java.io.DataInput;
```

```
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;

public class OrderBean implements WritableComparable<OrderBean> {

    private int order_id; // 订单id号
    private double price; // 价格

    public OrderBean() {
        super();
    }

    public OrderBean(int order_id, double price) {
        super();
        this.order_id = order_id;
        this.price = price;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeInt(order_id);
        out.writeDouble(price);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        order_id = in.readInt();
        price = in.readDouble();
    }

    @Override
    public String toString() {
        return order_id + "\t" + price;
    }

    public int getOrder_id() {
        return order_id;
    }

    public void setOrder_id(int order_id) {
        this.order_id = order_id;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    // 二次排序
    @Override
    public int compareTo(OrderBean o) {

        int result;
```

```

        if (order_id > o.getOrder_id()) {
            result = 1;
        } else if (order_id < o.getOrder_id()) {
            result = -1;
        } else {
            // 价格倒序排序
            result = price > o.getPrice() ? -1 : 1;
        }

        return result;
    }
}

```

OrderSortMapper类

mapper类的输出kv类型为OrderBean和NullWritable，这是因为key的bean就已经包含了要输出的全部数据，所以这里value设置为空：

```

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class OrderMapper extends Mapper<LongWritable, Text, OrderBean,
NullWritable> {

    OrderBean k = new OrderBean();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        // 1 获取一行
        String line = value.toString();

        // 2 截取
        String[] fields = line.split("\t");

        // 3 封装对象
        k.setOrder_id(Integer.parseInt(fields[0]));
        k.setPrice(Double.parseDouble(fields[2]));

        // 4 写出
        context.write(k, NullWritable.get());
    }
}

```

OrderSortGroupingComparator类

这是辅助排序的核心类，需要继承WritableComparator，然后重写compare方法，在该方法中对bean进行排序，重要的是排序中那些返回0的部分，这里认为id号相同的bean就认为是同一个bean，这样在进入reduce之前，保留下来的就只有第一个bean，也就是价格最高的订单。

```

import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class OrderGroupingComparator extends WritableComparator {

    protected OrderGroupingComparator() {
        super(OrderBean.class, true);
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {

        OrderBean aBean = (OrderBean) a;
        OrderBean bBean = (OrderBean) b;

        int result;
        if (aBean.getOrder_id() > bBean.getOrder_id()) {
            result = 1;
        } else if (aBean.getOrder_id() < bBean.getOrder_id()) {
            result = -1;
        } else {
            result = 0;
        }

        return result;
    }
}

```

除此之外还需要创建一个构造将比较对象的类传给父类，指定比较的bean。

OrderSortReducer类

```

import java.io.IOException;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Reducer;

public class OrderReducer extends Reducer<OrderBean, NullWritable, OrderBean,
NullWritable> {

    @Override
    protected void reduce(OrderBean key, Iterable<NullWritable> values, Context
context) throws IOException, InterruptedException {

        context.write(key, NullWritable.get());
    }
}

```

OrderSortDriver类

注意这里要进入辅助排序的类：

```

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;

```

```
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class OrderDriver {

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

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[]{"e:/input/inputorder" , "e:/output1"};

        // 1 获取配置信息
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 2 设置jar包加载路径
        job.setJarByClass(OrderDriver.class);

        // 3 加载map/reduce类
        job.setMapperClass(OrderMapper.class);
        job.setReducerClass(OrderReducer.class);

        // 4 设置map输出数据key和value类型
        job.setMapOutputKeyClass(OrderBean.class);
        job.setMapOutputValueClass(NullWritable.class);

        // 5 设置最终输出数据的key和value类型
        job.setOutputKeyClass(OrderBean.class);
        job.setOutputValueClass(NullWritable.class);

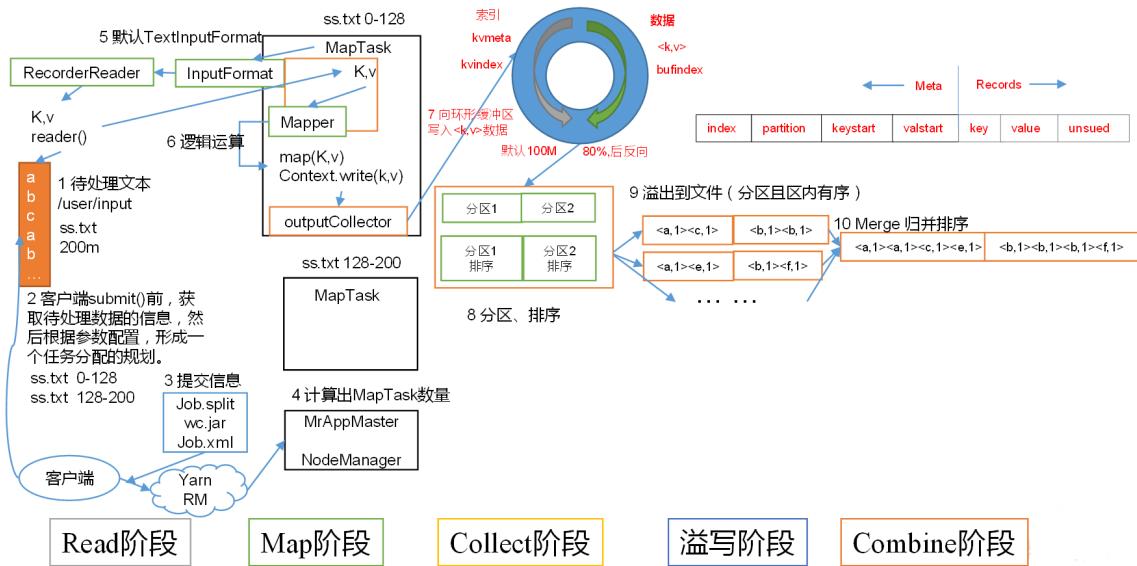
        // 6 设置输入数据和输出数据路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 8 设置reduce端的分组
        job.setGroupingComparatorClass(OrderGroupingComparator.class);

        // 7 提交
        boolean result = job.waitForCompletion(true);
        System.exit(result ? 0 : 1);
    }
}
```

MapTask和ReduceTask

maptask工作机制



(1) Read阶段：MapTask通过用户编写的RecordReader，从输入InputSplit中解析出一个个key/value。

(2) Map阶段：该节点主要是将解析出的key/value交给用户编写map()函数处理，并产生一系列新的key/value。

(3) Collect收集阶段：在用户编写map()函数中，当数据处理完成后，一般会调用OutputCollector.collect()输出结果。在该函数内部，它会将生成的key/value分区（调用Partitioner），并写入一个环形内存缓冲区中，缓冲区分成索引和数据两部分。

(4) Spill阶段：即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必要时对数据进行合并、压缩等操作。

溢写阶段详情：

步骤1：利用快速排序算法对缓存区内的数据进行排序，排序方式是，先按照分区编号Partition进行排序，然后按照key进行排序。这样，经过排序后，数据以分区为单位聚集在一起，且同一分区所有数据按照key有序。

步骤2：按照分区编号由小到大依次将每个分区中的数据写入任务工作目录下的临时文件output/spillN.out (N表示当前溢写次数) 中。如果用户设置了Combiner，则写入文件之前，对每个分区中的数据进行一次聚集操作。

步骤3：将分区数据的元信息写到内存索引数据结构SpillRecord中，其中每个分区的元信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存索引大小超过1MB，则将内存索引写到文件output/spillN.out.index中。

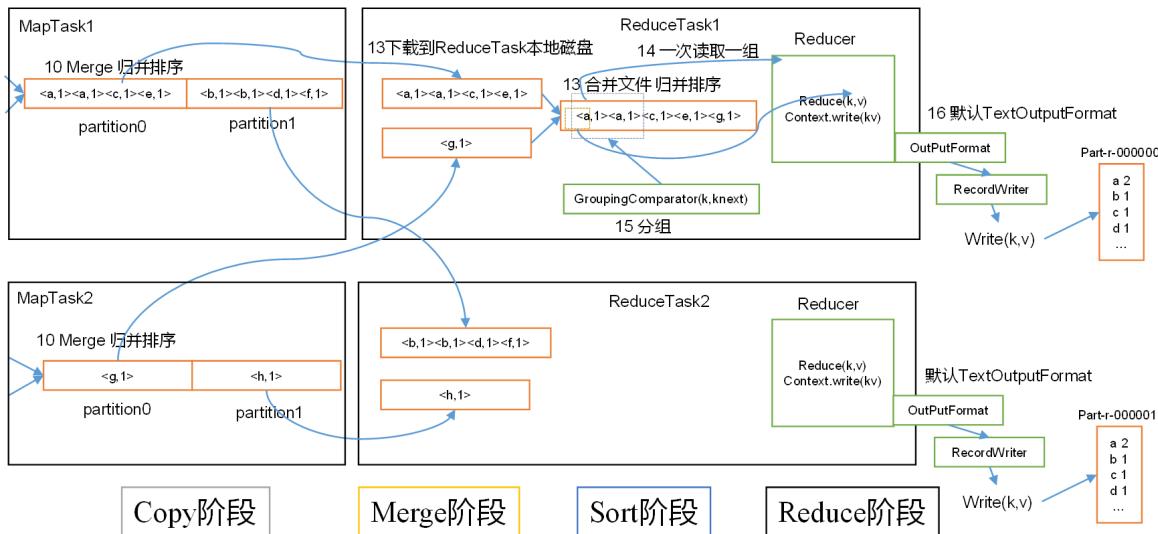
(5) Combine阶段：当所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

当所有数据处理完后，MapTask会将所有临时文件合并成一个大文件，并保存到文件output/file.out中，同时生成相应的索引文件output/file.out.index。

在进行文件合并过程中，MapTask以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式。每轮合并io.sort.factor (默认10) 个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

让每个MapTask最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

reducetask工作机制



(1) Copy阶段：ReduceTask从各个MapTask上远程拷贝一片数据，不同分区的数据交给不同的reducetask来处理，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。

(2) Merge阶段：在远程拷贝数据的同时，ReduceTask启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

(3) Sort阶段：按照MapReduce语义，用户编写reduce()函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个MapTask已经实现对自己的处理结果进行了局部排序，因此，ReduceTask只需对所有数据进行一次归并排序即可。这个过程中还可以进行对key进行分组，一组的key发到同一个reduce方法中。

(4) Reduce阶段：reduce()函数将计算结果写到HDFS上。

ReduceTask的并行度同样影响整个Job的执行并发度和执行效率，但与MapTask的并发数由切片数决定不同，ReduceTask数量的决定是可以直接手动设置：

```
// 默认值是1，手动设置为4
job.setNumReduceTasks(4);
```

具体设置多少个reducetask，要根据集群性能决定，默认reducetask为1，也就是输出文件个数是1；如果reducetask设置为0，表示没有reduce阶段，输出文件个数和map个数一致；如果分区数不是1，但reducetask是1，最后输出文件个数还是1，执行分区的前提是判断reducetask的个数；当进行的业务需求是汇总，可能会出现reducetask只能设置为1个的情况。

注意如果数据不均匀，就会在reducetask阶段产生数据倾斜，导致某个节点的压力过大。

OutputFormat数据输出

OutputFormat接口实现类

OutputFormat是MapReduce输出的基类，所有实现MapReduce输出都实现了 OutputFormat接口。

1、文本输出TextOutputFormat

默认的输出格式是TextOutputFormat，它把每条记录写为文本行。它的键和值可以是任意类型，因为TextOutputFormat调用toString()方法把它们转换为字符串。

2、SequenceFileOutputFormat

将SequenceFileOutputFormat输出作为后续 MapReduce任务的输入，这便是一种好的输出格式，因为它的格式紧凑，很容易被压缩。

3、自定义OutputFormat

可以定制最终文件的输出路径和输出格式，实现自定义OutputFormat的步骤分两步：

(1) 自定义一个类继承FileOutputFormat

(2) 改写RecordWriter，具体改写输出数据的方法write()

自定义OutputFormat案例

需求分析

过滤输入的log日志，包含atguigu的网站和不包含atguigu的网站输出不同文件。

输入数据：

```
http://www.baidu.com  
http://www.google.com  
http://cn.bing.com  
http://www.atguigu.com  
http://www.sohu.com  
http://www.sina.com  
http://www.sin2a.com  
http://www.sin2desa.com  
http://www.sindsafa.com
```

输出数据：

```
http://www.atguigu.com
```

```
http://cn.bing.com  
http://www.baidu.com  
http://www.google.com  
http://www.sin2a.com  
http://www.sin2desa.com  
http://www.sina.com  
http://www.sindsafa.com  
http://www.sohu.com
```

FilterMapper类

这里因为需要的数据全部在value中，key是不需要的（偏移量），所以传递给reduce的数据只有value，kv对中的值设置为空。

```
import java.io.IOException;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.NullWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;
```

```

public class FilterMapper extends Mapper<LongWritable, Text, Text, NullWritable>
{
    @Override
    protected void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException {
        // 写出
        context.write(value, NullWritable.get());
    }
}

```

FilterReducer类

输出文件时所有的网址都集中在一行输出，所以这里在reduce中给每个key都加\r\n。

```

import java.io.IOException;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FilterReducer extends Reducer<Text, NullWritable, Text,
NullWritable> {

    Text k = new Text();

    @Override
    protected void reduce(Text key, Iterable<NullWritable> values, Context
context) throws IOException, InterruptedException {
        // 1 获取一行
        String line = key.toString();

        // 2 拼接
        line = line + "\r\n";

        // 3 设置key
        k.set(line);

        // 4 输出
        context.write(k, NullWritable.get());
    }
}

```

自定义OutputFormat类

返回一个recordwriter类，把job传入。

```

import java.io.IOException;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

```

```
public class FilterOutputFormat extends FileOutputFormat<Text, NullWritable>{

    @Override
    public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext
job) throws IOException, InterruptedException {

        // 创建一个RecordWriter
        return new FilterRecordWriter(job);
    }
}
```

```
import java.io.IOException;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

public class FilterRecordWriter extends RecordWriter<Text, NullWritable> {

    FSDataOutputStream atguiguOut = null;
    FSDataOutputStream otherOut = null;

    public FilterRecordWriter(TaskAttemptContext job) {

        // 1 获取文件系统
        FileSystem fs;

        try {
            fs = FileSystem.get(job.getConfiguration());

            // 2 创建输出文件路径
            Path atguiguPath = new Path("e:/atguigu.log");
            Path otherPath = new Path("e:/other.log");

            // 3 创建输出流
            atguiguOut = fs.create(atguiguPath);
            otherOut = fs.create(otherPath);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void write(Text key, NullWritable value) throws IOException,
InterruptedException {

        // 判断是否包含“atguigu”输出到不同文件
        if (key.toString().contains("atguigu")) {
            atguiguOut.write(key.toString().getBytes());
        } else {
            otherOut.write(key.toString().getBytes());
        }
    }
}
```

```

    }

    @Override
    public void close(TaskAttemptContext context) throws IOException,
    InterruptedException {

        // 关闭资源
        IOUtils.closeStream(atguiguOut);
        IOUtils.closeStream(otherOut);
    }
}

```

在recordwriter建立构造方法创建两个输出流，write方法中根据key中是否含atguigu将key写入不同输出流，最后用close关闭资源。这里继承的FileOutputFormat类的泛型是reduce输出的kv类型。

FilterDriver类

注意这里要设置自定义的输出类型，指定输出类型的数据输入和输出路径，这里的输出路径是用来输出运行标志文件的，真正的数据结果依然在自定义输出类中指定。

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FilterDriver {

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

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[] { "e:/input/inputoutputformat", "e:/output2" };

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(FilterDriver.class);
        job.setMapperClass(FilterMapper.class);
        job.setReducerClass(FilterReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(NullWritable.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        // 要将自定义的输出格式组件设置到job中
        job.setOutputFormatClass(FilterOutputFormat.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));

        // 虽然我们自定义了outputformat，但是因为我们的outputformat继承自
        fileoutputformat
        // 而fileoutputformat要输出一个_SUCCESS文件，所以，在这还得指定一个输出目录
    }
}

```

```
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Join

Reduce Join

当需要处理多个表或文件的join操作时，此时hadoop的数据流程如下：

Map端的主要工作：为来自不同表或文件的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，最后进行合并就ok了。

案例分析

输入数据为一个订单表和一个商品表。

订单表（订单表id、商品id也就是pid、商品数量）

1001	01	1
1002	02	2
1003	03	3
1004	01	4
1005	02	5
1006	03	6

商品表（pid、商品名）

01	小米
02	华为
03	格力

要求进行join操作，将商品信息表中数据根据商品pid合并到订单数据表中，输出结果应该为：

1001	小米	1
1004	小米	4
1002	华为	2
1005	华为	5
1003	格力	3
1006	格力	6

TableBean类

这个bean类要有两个表全部的属性，包括订单id、商品id、商品数、商品名，还要再加一个标记位指定bean到底来自哪张表，这样一共5个属性。toString方法只涉及三个属性，就是最后要输出的那三个属性。

```
import java.io.DataInput;
```

```
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;

public class TableBean implements Writable {

    private String order_id; // 订单id
    private String p_id;      // 产品id
    private int amount;       // 产品数量
    private String pname;     // 产品名称
    private String flag;      // 表的标记

    public TableBean() {
        super();
    }

    public TableBean(String order_id, String p_id, int amount, String pname,
String flag) {

        super();

        this.order_id = order_id;
        this.p_id = p_id;
        this.amount = amount;
        this.pname = pname;
        this.flag = flag;
    }

    public String getFlag() {
        return flag;
    }

    public void setFlag(String flag) {
        this.flag = flag;
    }

    public String getOrder_id() {
        return order_id;
    }

    public void setOrder_id(String order_id) {
        this.order_id = order_id;
    }

    public String getP_id() {
        return p_id;
    }

    public void setP_id(String p_id) {
        this.p_id = p_id;
    }

    public int getAmount() {
        return amount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }
}
```

```

    }

    public String getPname() {
        return pname;
    }

    public void setPname(String pname) {
        this.pname = pname;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(order_id);
        out.writeUTF(p_id);
        out.writeInt(amount);
        out.writeUTF(pname);
        out.writeUTF(flag);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        this.order_id = in.readUTF();
        this.p_id = in.readUTF();
        this.amount = in.readInt();
        this.pname = in.readUTF();
        this.flag = in.readUTF();
    }

    @Override
    public String toString() {
        return order_id + "\t" + pname + "\t" + amount + "\t" ;
    }
}

```

TableMapper类

maper输出的key是连接字段pid，输出的value为bean。在map阶段还需要给不同的数据打上标记位，这就需要在执行map方法前就取到文件名，这里重写了setup，通过拿到文件切片得到文件名称，然后在map方法中判断文件名确定到底是订单表还是商品表，根据商品名的不同执行不同的逻辑。

```

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class TableMapper extends Mapper<LongWritable, Text, Text, TableBean>{

    String name;
    TableBean bean = new TableBean();
    Text k = new Text();

    @Override
    protected void setup(Context context) throws IOException,
    InterruptedException {
        // 1 获取输入文件切片
    }
}

```

```

FileSplit split = (FileSplit) context.getInputSplit();

// 2 获取输入文件名称
name = split.getPath().getName();
}

@Override
protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

    // 1 获取输入数据
    String line = value.toString();

    // 2 不同文件分别处理
    if (name.startsWith("order")) { // 订单表处理

        // 2.1 切割
        String[] fields = line.split("\t");

        // 2.2 封装bean对象
        bean.setOrder_id(fields[0]);
        bean.setP_id(fields[1]);
        bean.setAmount(Integer.parseInt(fields[2]));
        bean.setPname("");
        bean.setFlag("order");

        k.set(fields[1]);
    } else { // 产品表处理

        // 2.3 切割
        String[] fields = line.split("\t");

        // 2.4 封装bean对象
        bean.setP_id(fields[0]);
        bean.setPname(fields[1]);
        bean.setFlag("pd");
        bean.setAmount(0);
        bean.setOrder_id("");

        k.set(fields[0]);
    }

    // 3 写出
    context.write(k, bean);
}
}

```

对于订单表bean来说，flag设置为order，商品名设置为空字符串；对于商品表bean来说，flag设置为pd，订单id设置为空字符串，订单中的商品数量设置为0。

TableReducer类

在reduce阶段，需要合并来自不同表的数据。reduce的输出类型kv分别为bean和null。因为一个pid可能对应多个订单对象，只对应一个商品对象，所以这里建立了一个订单对象集合ArrayList<TableBean>。经过map阶段的处理，所有key相同的数据都会汇聚到一个reduce方法中来，也就是对应同一个pid的多个bean，根据bean的标记位区分到底是商品bean还是订单bean，分别装入对应的对象中。

这里遍历values得到的是一个引用，必须将引用中的数据拷贝到另外一个对象中，如果直接将引用装入集合，那么集合中的元素都是最后一个引用指向的元素。

最后进行表的拼接，对所有订单bean进行遍历，将商品bean的商品名属性设置到订单bean中，然后将合并后的订单bean写出。（这里遍历的是订单bean的原因是因为订单表是要输出的数据基础）

```
import java.io.IOException;
import java.util.ArrayList;
import org.apache.commons.beanutils.BeanUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TableReducer extends Reducer<Text, TableBean, TableBean,
NullWritable> {

    @Override
    protected void reduce(Text key, Iterable<TableBean> values, Context context)
throws IOException, InterruptedException {

        // 1准备存储订单的集合
        ArrayList<TableBean> orderBeans = new ArrayList<>();

        // 2 准备bean对象
        TableBean pdBean = new TableBean();

        for (TableBean bean : values) {

            if ("order".equals(bean.getFlag())) {// 订单表

                // 拷贝传递过来的每条订单数据到集合中
                TableBean orderBean = new TableBean();

                try {
                    BeanUtils.copyProperties(orderBean, bean);
                } catch (Exception e) {
                    e.printStackTrace();
                }

                orderBeans.add(orderBean);
            } else {// 产品表

                try {
                    // 拷贝传递过来的产品表到内存中
                    BeanUtils.copyProperties(pdBean, bean);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }

        // 3 表的拼接
        for(TableBean bean:orderBeans){

            bean.setPname (pdBean.getPname());

            // 4 数据写出去
        }
    }
}
```

```

        context.write(bean, NullWritable.get());
    }
}
}

```

TableDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TableDriver {

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

        // 0 根据自己电脑路径重新配置
        args = new String[]{"e:/input/inputtable","e:/output1"};

        // 1 获取配置信息，或者job对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 指定本程序的jar包所在的本地路径
        job.setJarByClass(TableDriver.class);

        // 3 指定本业务job要使用的Mapper/Reducer业务类
        job.setMapperClass(TableMapper.class);
        job.setReducerClass(TableReducer.class);

        // 4 指定Mapper输出数据的kv类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(TableBean.class);

        // 5 指定最终输出的数据的kv类型
        job.setOutputKeyClass(TableBean.class);
        job.setOutputValueClass(NullWritable.class);

        // 6 指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

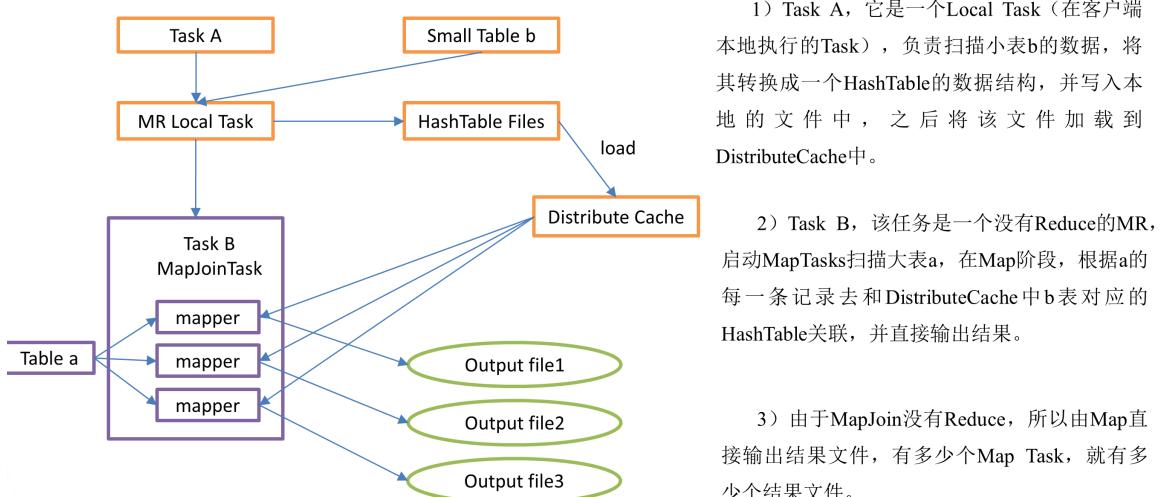
        // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行
        boolean result = job.waitForCompletion(true);
        System.exit(result ? 0 : 1);
    }
}

```

缺点：这种方式中，合并的操作是在Reduce阶段完成，Reduce端的处理压力太大，Map节点的运算负载则很低，资源利用率不高，且在Reduce阶段极易产生数据倾斜。解决方案就是在Map端实现数据合并。

Map Join

对于上个例子，在map端实现数据合并需要先把商品表加载到内存中，然后读取订单表直接完成字符串的拼接，这种方法必须要求商品表是小表，如果该表很大则无法事先读取到内存中，故这种join方法适用于一张表大，一张表小的情况。



具体步骤：

- (1) 在Mapper的setup阶段，将文件读取到缓存集合中。
- (2) 在驱动函数中加载缓存。

驱动类

```

import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class DistributedCacheDriver {

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

        // 0 根据自己电脑路径重新配置
        args = new String[]{"e:/input/inputtable2", "e:/output1"};

        // 1 获取job信息
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 设置加载jar包路径
        job.setJarByClass(DistributedCacheDriver.class);

        // 3 关联map
        job.setMapperClass(DistributedCacheMapper.class);

        // 4 设置最终输出数据类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        // 5 设置输入输出路径
    }
}

```

```

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // 6 加载缓存数据
    job.addCacheFile(new URI("file:///e:/input/inputcache/pd.txt"));

    // 7 Map端Join的逻辑不需要Reduce阶段，设置reduceTask数量为0
    job.setNumReduceTasks(0);

    // 8 提交
    boolean result = job.waitForCompletion(true);
    System.exit(result ? 0 : 1);
}
}

```

在驱动类中要读取小表的缓存，注意这里没有指定mapper输出的kv类型，因为此时没有reduce，最终输出就是mapper的输出。

DistributedCacheMapper类

在setup方法中读取文件获得输入流，然后逐行读取解析，以pid为key，商品名为value装入hashmap中，在map方法中读取订单表，然后进行拼接最后写入context。

```

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;
import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class DistributedCacheMapper extends Mapper<LongWritable, Text, Text, NullWritable>{

    Map<String, String> pdMap = new HashMap<>();

    @Override
    protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context context) throws IOException, InterruptedException {

        // 1 获取缓存的文件
        URI[] cacheFiles = context.getCacheFiles();
        String path = cacheFiles[0].getPath().toString();

        BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(path), "UTF-8"));

        String line;
        while(StringUtils.isNotEmpty(line = reader.readLine())){

            // 2 切割
            String[] fields = line.split("\t");

```

```

        // 3 缓存数据到集合
        pdMap.put(fields[0], fields[1]);
    }

    // 4 关流
    reader.close();
}

Text k = new Text();

@Override
protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

    // 1 获取一行
    String line = value.toString();

    // 2 截取
    String[] fields = line.split("\t");

    // 3 获取产品id
    String pId = fields[1];

    // 4 获取商品名称
    String pdName = pdMap.get(pId);

    // 5 拼接
    k.set(line + "\t" + pdName);

    // 6 写出
    context.write(k, NullWritable.get());
}
}

```

计数器

Hadoop为每个作业维护若干内置计数器，以描述多项指标。例如，某些计数器记录已处理的字节数和记录数，使用户可监控已处理的输入数据量和已产生的输出数据量。

运行hadoop程序时，控制台显示的如下部分就是计数器：

```

Map-Reduce Framework
Map input records=3
Map output records=3
Map output bytes=334
Map output materialized bytes=358
Input split bytes=326
Combine input records=0
Combine output records=0
Reduce input groups=3
Reduce shuffle bytes=358
Reduce input records=3
Reduce output records=3
Spilled Records=6
Shuffled Maps =3

```

```
Failed shuffles=0
Merged Map outputs=3
GC time elapsed (ms)=8
Total committed heap usage (bytes)=1539833856
```

在程序中加入计数语句就可以在控制台或日志中显示，加入计数器的方法主要有两种：

(1) 采用枚举的方式统计计数：enum MyCounter{MALFORORMED, NORMAL}

```
//对枚举定义的自定义计数器加1
context.getCounter(MyCounter.MALFORORMED).increment(1);
```

(2) 采用计数器组、计数器名称的方式统计

```
context.getCounter("counterGroup", "counter").increment(1);
```

最后在控制台上就会显示计数次数：

```
counterGroup
    counter=1
```

数据清洗

在运行核心业务MapReduce程序之前，往往要先对数据进行清洗，清理掉不符合用户要求的数据。清理的过程往往只需要运行Mapper程序，不需要运行Reduce程序。

需求分析

去除日志中字段长度小于等于11的日志。最后输出时要求每行数据长度都大于11。

LogMapper类

封装了一个parseLog方法，用来判断本行数据是否合理，如果不合理返回false，map方法就直接返回，返回true再执行下面，将数据写入context中。引入计数器，在map阶段统计成功的条数和被过滤掉的条数。

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class LogMapper extends Mapper<LongWritable, Text, Text, NullWritable>{

    Text k = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
```

```

    // 1 获取1行数据
    String line = value.toString();

    // 2 解析日志
    boolean result = parseLog(line, context);

    // 3 日志不合法退出
    if (!result) {
        return;
    }

    // 4 设置key
    k.set(line);

    // 5 写出数据
    context.write(k, NullWritable.get());
}

// 2 解析日志
private boolean parseLog(String line, Context context) {

    // 1 截取
    String[] fields = line.split(" ");

    // 2 日志长度大于11的为合法
    if (fields.length > 11) {

        // 系统计数器
        context.getCounter("map", "true").increment(1);
        return true;
    } else {
        context.getCounter("map", "false").increment(1);
        return false;
    }
}
}

```

LogDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class LogDriver {

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

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[] { "e:/input/inputlog", "e:/output1" };

        // 1 获取job信息
        Configuration conf = new Configuration();

```

```

        Job job = Job.getInstance(conf);

        // 2 加载jar包
        job.setJarByClass(LogDriver.class);

        // 3 关联map
        job.setMapperClass(LogMapper.class);

        // 4 设置最终输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        // 设置reducetask个数为0
        job.setNumReduceTasks(0);

        // 5 设置输入和输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 6 提交
        job.waitForCompletion(true);
    }
}

```

这里没有reduce阶段，故reducetask设置成0.

数据压缩

压缩技术能够有效减少底层存储系统（HDFS）读写字节数。压缩提高了网络带宽和磁盘空间的效率。在运行MR程序时，I/O操作、网络数据传输、Shuffle和Merge要花大量的时间，尤其是数据规模很大和工作负载密集的情况下，因此，使用数据压缩显得非常重要。

鉴于磁盘I/O和网络带宽是Hadoop的宝贵资源，数据压缩对于节省资源、最小化磁盘I/O和网络传输非常有帮助。可以在任意MapReduce阶段启用压缩。不过，应该合理使用数据压缩，否则会增加数据处理速度。

压缩操作应该使用在IO密集的任务中，对于CPU密集的任务使用压缩会适得其反。

压缩方式

常用的压缩方式有以下几种：

压缩格式	hadoop是否自带	算法	文件扩展名	是否可切分	换成压缩格式后，原来的程序是否需要修改
DEFLATE	直接使用	DEFLATE	.deflate	否	和文本处理一样，不需要修改
Gzip	直接使用	DEFLATE	.gz	否	和文本处理一样，不需要修改
bzip2	直接使用	bzip2	.bz2	是	和文本处理一样，不需要修改
I7O	需要安装	I7O	I7O	是	需要建索引，还需要指定输

					入格式
压缩格式	hadoop是否需要自带	算法	文件扩展名	是否可切分	换成压缩格式后，原来的程序和文本处理一样，不需要修改是否需要修改
Snappy	需要自带	Snappy	.snappy	否	

为了支持多种压缩/解压缩算法，Hadoop引入了编码/解码器，如下表所示：

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

几种算法的大致对比如下：

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s
Snappy	8.3GB	很大	250MB/s	500MB/s

Gzip压缩

压缩率比较高，而且压缩/解压速度也比较快；Hadoop本身支持，在应用中处理Gzip格式的文件就和直接处理文本一样；大部分Linux系统都自带Gzip命令，使用方便。

但是它不支持Split，也就是说压缩后的数据是不能被切分处理的。这就导致了这种压缩适合放置map方法后的阶段，防止直接进入map方法的文件过大。

应用场景：map后，当每个文件压缩之后在130M以内的（1个块大小内），都可以考虑用Gzip压缩格式。比如说一天或者一个小时的日志压缩成一个Gzip文件。

Bzip2压缩

支持Split；具有很高的压缩率，比Gzip压缩率都高；Hadoop本身自带，使用方便，但压缩/解压速度慢。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，同时又需要支持Split，而且兼容之前的应用程序的情况。

Lzo压缩

压缩/解压速度也比较快，合理的压缩率；支持Split，是Hadoop中最流行的压缩格式；可以在Linux系统下安装lzo命令，使用方便。

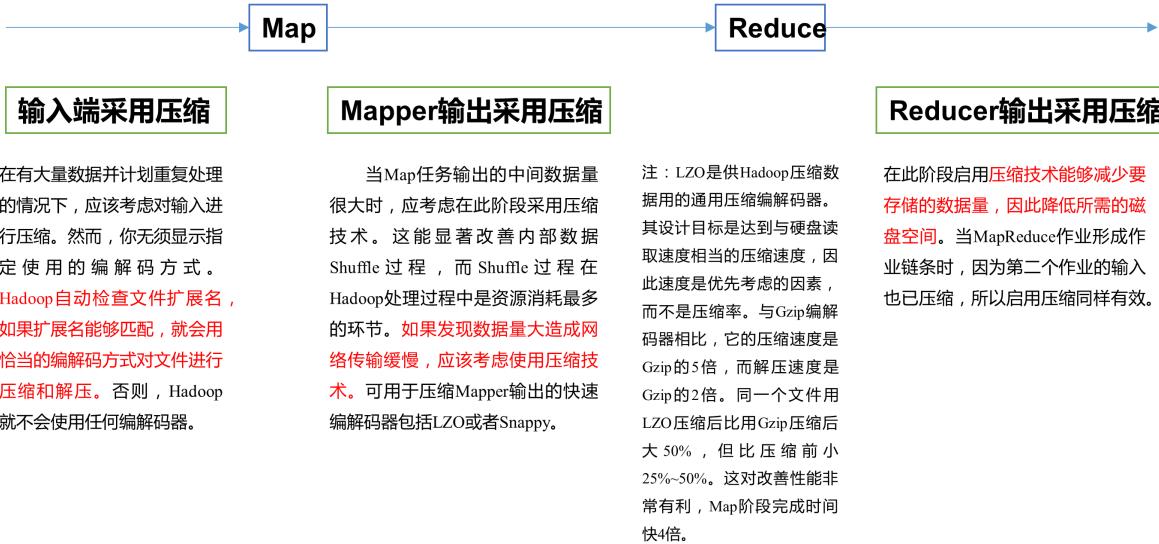
但是在应用中对Lzo格式的文件需要做一些特殊处理（为了支持Split需要建索引，还需要指定InputFormat为Lzo格式）。

Snappy压缩

高速压缩速度和合理的压缩率。但不支持Split；压缩率比Gzip要低；Hadoop本身不支持，需要安装。

应用场景：当MapReduce作业的Map输出的数据比较大的时候，作为Map到Reduce的中间数据的压缩格式；或者作为一个MapReduce作业的输出和另外一个MapReduce作业的输入。

压缩位置选择



输入给hadoop的数据也可以是压缩形式的，这就是输入端压缩。输出的数据也可以用压缩文件表达，这就是reducer输出端压缩。在map和reduce之间传递的数据也可以压缩，这就是mapper输出端压缩。

对于mapper输出端压缩可采用LZO或Snappy，因为这两个都是解压和压缩速度很快的。

在reducer输出端的压缩技术更倾向于减少要存储的数据量。

压缩参数设置

输入压缩阶段参数

在core-site.xml中配置参数io.compression.codecs，它的默认值是：

```
org.apache.hadoop.io.compress.DefaultCodec,  
org.apache.hadoop.io.compress.GzipCodec,  
org.apache.hadoop.io.compress.BZip2Codec
```

参数的作用是指明输入端能自动匹配的压缩格式，可以手动指定其他编码器，前提是该压缩命令linux必须提前安装好。

mapper输出端压缩参数

在mapred-site.xml中配置参数mapreduce.map.output.compress，它的默认值是false，表示不启用压缩，可以改为true启用压缩。

还可以在mapred-site.xml中配置参数org.apache.hadoop.io.compress.DefaultCodec，它的默认值是：

```
org.apache.hadoop.io.compress.DefaultCodec
```

它的作用是指明该阶段的压缩格式，可以手动指定其他编码器，前提是该压缩命令linux必须提前安装好。

reducer输出段压缩参数

在mapred-site.xml中配置参数mapreduce.output.fileoutputformat.compress，它的默认值是false，表示不启用压缩，可以改为true。

在mapred-site.xml中配置mapreduce.output.fileoutputformat.compress.codec，它的默认值是：

```
org.apache.hadoop.io.compress.DefaultCodec
```

它的作用是指明该阶段的压缩格式，可以手动指定其他。

在mapred-site.xml中配置mapreduce.output.fileoutputformat.compress.type，它的默认值是RECORD，表示该阶段的压缩针对行，还可以改为NONE和BLOCK（文件块），一般来说按块效率较高。

压缩案例

数据流的压缩和解压缩

压缩操作需要建立一个压缩输出流，读取一个压缩文件也需要一个特殊的解压缩输入流。

要想对正在被写入一个输出流的数据进行压缩，我们可以使用createOutputStream(OutputStreamout)方法创建一个CompressionOutputStream，将其以压缩格式写入底层的流。

相反，要想对从输入流读取而来的数据进行解压缩，则调用createInputStream(InputStreamin)函数，从而获得一个CompressionInputStream，从而从底层的流读取未压缩的数据。

创建main方法，其中调用压缩的方法和解压缩的方法：

```
public static void main(String[] args) throws Exception {
    compress("e:/hello.txt","org.apache.hadoop.io.compress.BZip2Codec");
    decompress("e:/hello.txt.bz2");
}
```

压缩方法：

根据传入的压缩格式得到对应的压缩类CompressionCodec，然后建立压缩输入流，就可以进行流的对拷。

```
// 1、压缩
private static void compress(String filename, String method) throws
Exception {

    // (1) 获取输入流
    FileInputStream fis = new FileInputStream(new File(filename));

    Class codecClass = Class.forName(method);

    CompressionCodec codec = (CompressionCodec)
ReflectionUtils.newInstance(codecClass, new Configuration());
```

```

    // (2) 获取输出流
    FileOutputStream fos = new FileOutputStream(new File(filename +
codec.getDefaultExtension()));
    CompressionOutputStream cos = codec.createOutputStream(fos);

    // (3) 流的对拷
    IOUtils.copyBytes(fis, cos, 1024*1024*5, false);

    // (4) 关闭资源
    cos.close();
    fos.close();
    fis.close();
}

```

解压方法：

首先查看是否是支持的解压类型，如果是获得解压类CompressionCodec，然后建立解压输出流，进行流的对拷。

```

// 2、解压缩
private static void decompress(String filename) throws
FileNotFoundException, IOException {

    // (0) 校验是否能解压缩
    CompressionCodecFactory factory = new CompressionCodecFactory(new
Configuration());

    CompressionCodec codec = factory.getCodec(new Path(filename));

    if (codec == null) {
        System.out.println("cannot find codec for file " + filename);
        return;
    }

    // (1) 获取输入流
    CompressionInputStream cis = codec.createInputStream(new
FileInputStream(new File(filename)));

    // (2) 获取输出流
    FileOutputStream fos = new FileOutputStream(new File(filename +
".decoded"));

    // (3) 流的对拷
    IOUtils.copyBytes(cis, fos, 1024*1024*5, false);

    // (4) 关闭资源
    cis.close();
    fos.close();
}

```

map输出端压缩

只需要在驱动类中开启map端压缩并指定压缩模式即可：

```

// 开启map端输出压缩
configuration.setBoolean("mapreduce.map.output.compress", true);
// 设置map端输出压缩方式
configuration.setClass("mapreduce.map.output.compress.codec",
BZip2Codec.class, CompressionCodec.class);

```

reduce输出端压缩

只需要在驱动类中开启reduce端压缩并指定压缩模式即可：

```

// 设置reduce端输出压缩开启
FileOutputFormat.setCompressOutput(job, true);

// 设置压缩的方式
FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);

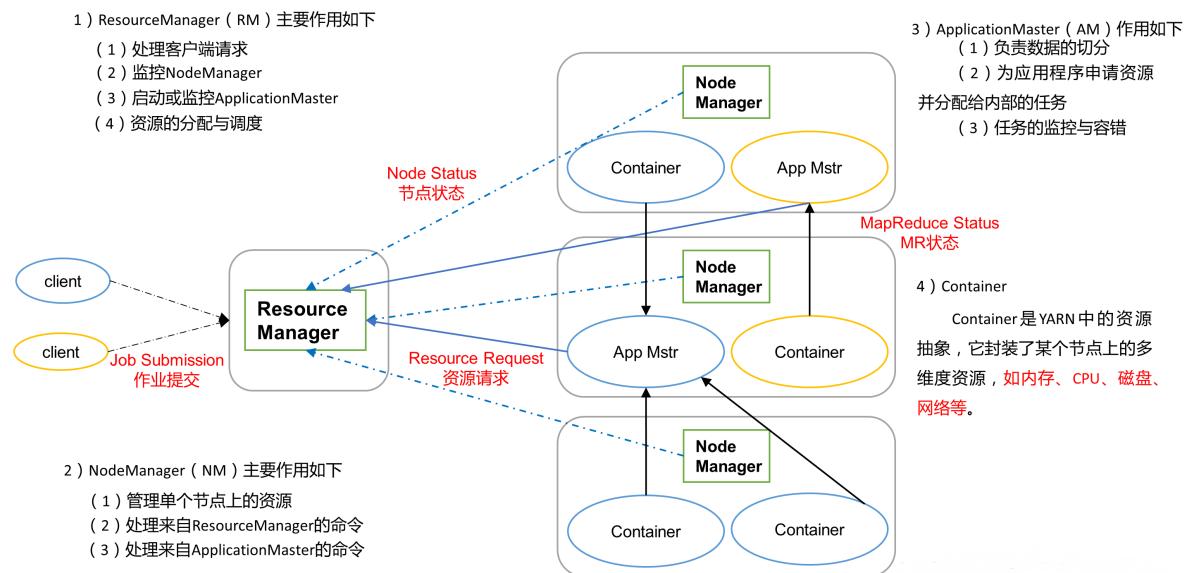
```

Yarn

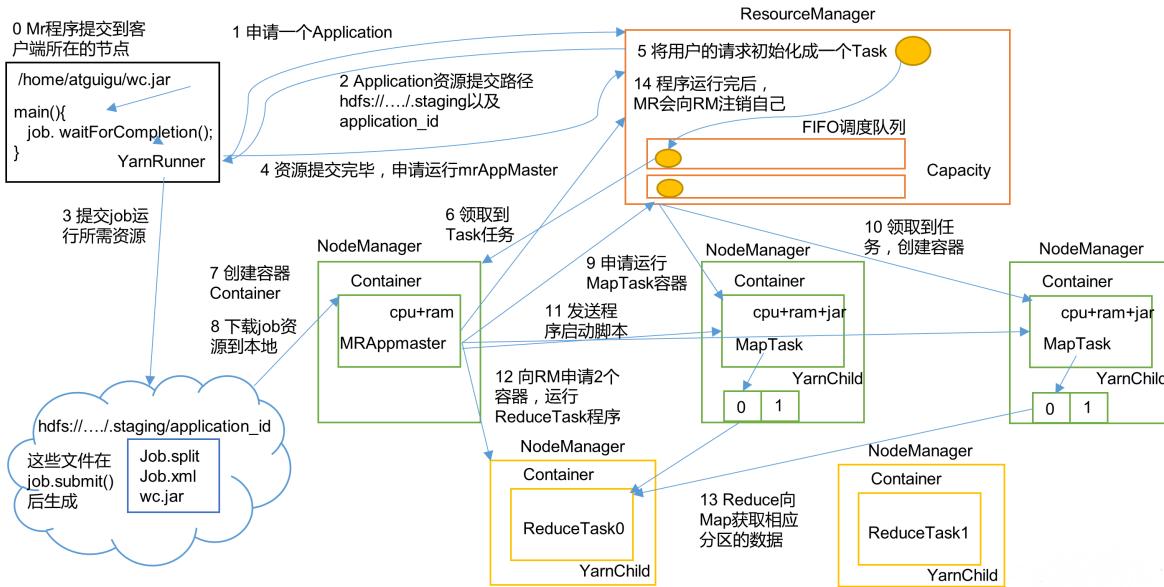
Yarn资源调度器

Yarn基本架构

YARN主要由ResourceManager、NodeManager、ApplicationMaster和Container等组件构成：



Yarn工作流程



- (1) MR程序提交到客户端所在的节点。
- (2) YarnRunner向ResourceManager申请一个Application。
- (3) RM将该应用程序的资源路径返回给YarnRunner，通常是hdfs中的一个路径。
- (4) 该程序将运行所需资源提交到HDFS上，包括jar包、切片信息和配置文件。
- (5) 程序资源提交完毕后，客户端向ResourceManager申请运行mrAppMaster。
- (6) RM将用户的请求初始化成一个Task。
- (7) 其中一个NodeManager领取到Task任务。
- (8) 该NodeManager创建容器Container，并产生MRAppmaster。
- (9) Container从HDFS上拷贝资源到本地。
- (10) MRAppmaster向RM 申请运行MapTask资源。
- (11) RM将运行MapTask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器。
- (12) MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动 MapTask，MapTask对数据分区排序。
- (13) MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask。
- (14) ReduceTask向MapTask获取相应分区的数据。
- (15) 程序运行完毕后，MR会向RM申请注销自己。

在这个过程中，YARN中的任务将其进度和状态(包括counter)返回给应用管理器，客户端每秒(通过mapreduce.client.progressmonitor.pollinterval设置)向应用管理器请求进度更新，展示给用户。

除了向应用管理器请求作业进度外，客户端每5秒都会通过调用waitForCompletion()来检查作业是否完成。时间间隔可以通过mapreduce.client.completion.pollinterval来设置。

作业完成之后，应用管理器和Container会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查。

资源调度器

Hadoop作业调度器主要有三种：FIFO、Capacity Scheduler和Fair Scheduler。Hadoop2.7.2默认的资源调度器是Capacity Scheduler。

在yarn-default.xml文件中可以设置作业调度器：

```
<property>
  <description>The class to use as the resource scheduler.</description>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>
</property>
```

先进先出调度器 (FIFO)

按照job到达的时间先到先服务。只要有空闲的资源就会从队列中取走job，并申请执行job中一个或几个task。

容量调度器 (Capacity Scheduler)



它就是多个队列的组合，每个队列都设置独立的资源量，同时防止同一个用户独占队列的资源，调度器会对同一用户提交的任务数量进行限制。

三个队列同时按照任务的先后顺序依次执行，比如，job11、job21和job31分别排在队列最前面，先运行，也是并行运行。

计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个该比值最小的队列——最闲的，然后优先提交该队列的job。

按照作业优先级和提交时间顺序，同时考虑用户资源量限制和内存限制对队列内任务排序。

公平调度器 (Fair Scheduler)



由多个队列组成，每个队列都可以设置独立的资源量，同一队列中的所有作业都共享资源。

每个队列中的job按照优先级分配资源，优先级越高分配的资源越多，但是每个 job 都会分配到资源以确保公平。

在资源有限的情况下，每个job理想情况下获得的计算资源与实际获得的计算资源存在一种差距，这个差距就叫做缺额。

在同一个队列中，job的资源缺额越大，越先获得资源优先执行。作业是按照缺额的高低来先后执行的，而且可以看到上图有多个作业同时运行。

公平调度器并发度高一些，要求机器配置也高。

推测执行

一个作业由若干个Map任务和Reduce任务构成，有可能会出现个别任务执行的极慢或卡死的情况。推测执行机制就是及时发现拖后腿的任务，比如某个任务运行速度远慢于任务平均速度。为拖后腿任务启动一个备份任务，同时运行。谁先运行完，则采用谁的结果。

执行推测任务的前提条件：

- (1) 每个Task只能有一个备份任务
- (2) 当前job已完成的Task必须不小于0.05 (5%)，不能某个job刚开始就要启动备份。
- (3) 开启推测执行参数设置。 mapred-site.xml文件中默认是打开的：

```
<property>
    <name>mapreduce.map.speculative</name>
    <value>true</value>
    <description>If true, then multiple instances of some map tasks may be
executed in parallel.</description>
</property>

<property>
    <name>mapreduce.reduce.speculative</name>
    <value>true</value>
    <description>If true, then multiple instances of some reduce tasks may be
executed in parallel.</description>
</property>
```

不能启用推测执行机制情况：

- (1) 任务间存在严重的负载倾斜；(不能再开一个巨大的任务)
- (2) 特殊任务，比如任务向数据库中写数据。(数据往数据库中写多次)

推测执行算法原理

假设某一时刻，任务T的执行进度为progress，则可通过一定的算法推测出该任务的最终完成时刻 estimateEndTime。另一方面，如果此刻为该任务启动一个备份任务，则可推断出它可能的完成时刻 estimateEndTime`，于是可得出以下几个公式：

estimatedRunTime = (currentTimestamp - taskStartTime) / progress
推测运行时间 (60s) = (当前时刻 (6) - 任务启动时刻 (0)) / 任务运行比例 (10%)

estimateEndTime = estimatedRunTime + taskStartTime
推测执行完时刻 60 = 推测运行时间 (60s) + 任务启动时刻 (0)

estimateEndTime` = currentTimestamp + averageRunTime
备份任务推测完成时刻 (16) = 当前时刻 (6) + 运行完成任务的平均时间 (10s)

上图中的第一个公式的意思是，可以根据当前任务执行进度和已用时间反推一个job的总运行时间

第二个公式是预测执行完毕的时间点=job的总运行时间+启动时间

第三个公式是如果此刻开启备份任务，备份任务执行完毕的时间点应该=当前时刻+已完成job的平均运行时间

- 1) MR总是选择 (estimateEndTime- estimateEndTime `) 差值最大的任务，并为之启动备份任务。
- 2) 为了防止大量任务同时启动备份任务造成的资源浪费，MR为每个作业设置了同时启动的备份任务数目上限。
- 3) 推测执行机制实际上采用了经典的优化算法：以空间换时间，它同时启动多个相同任务处理相同的数 据，并让这些任务竞争以缩短数据处理时间。显然，这种方法需要占用更多的计算资源。在集群资源紧缺的情况下，应合理使用该机制，争取在多用少量资源的情况下，减少作业的计算时间。

优化

hadoop企业优化

速度低的原因

MapReduce程序效率瓶颈主要有两点：

- 1、计算机性能。如CPU、磁盘情况、内存、网络情况。
- 2、IO操作过多。可能的原因有数据倾斜、map和reduce数量设置不合理、map运行时间很长导致reduce等待过久、小文件过多，大量不可分块的超大文件、溢写次数多、合并文件次数多。

MapReduce优化方法

MapReduce优化方法主要从六个方面考虑：数据输入、Map阶段、Reduce阶段、IO传输、数据倾斜问题和常用的调优参数。

数据输入

尽量在输入到程序时就避免多个小文件的情况，尽可能合成大文件然后输入，否则程序多次装载map任务会很慢。

也可以采用CombineTextInputFormat来作为输入，解决输入端大量小文件场景。它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个MapTask处理。

map阶段

(1) 减少溢写 (Spill) 次数：通过调整io.sort.mb (缓冲区的大小) 及sort.spill.percent (到达环形缓冲区多大百分比触发溢写) 参数值，增大触发Spill的内存上限，减少Spill次数，从而减少磁盘IO。

(2) 减少合并 (Merge) 次数：通过调整io.sort.factor参数，增大Merge的文件数目，减少Merge的次数，从而缩短MR处理时间。(合并溢写文件时要分批进行，可以通过调整参数增大merge文件的数目，减少merge的次数)

(3) 在Map之后，不影响业务逻辑前提下，先进行Combine处理，减少 I/O。

reduce阶段

(1) 合理设置Map和Reduce数：两个都不能设置太少，也不能设置太多。太少，会导致Task等待，延长处理时间；太多，会导致Map、Reduce任务间竞争资源，造成处理超时等错误。

(2) 设置Map、Reduce共存：调整slowstart.completedmaps参数，使Map运行到一定程度后，Reduce也开始运行，减少Reduce的等待时间。

(3) 规避使用Reduce：因为Reduce在用于连接数据集的时候将会产生大量的网络消耗。

(4) 合理设置Reduce端的Buffer：默认情况下，数据达到一个阈值的时候，Buffer中的数据就会写入磁盘，然后Reduce会从磁盘中获得所有的数据。也就是说，Buffer和Reduce是没有直接关联的，中间多次写磁盘->读磁盘的过程，既然有这个弊端，那么就可以通过参数来配置，使得Buffer中的一部分数据可以直接输送到Reduce，从而减少IO开销：mapreduce.reduce.input.buffer.percent，默认为0.0。当值大于0的时候，会保留指定比例的内存读Buffer中的数据直接拿给Reduce使用。这样一来，设置Buffer需要内存，读取数据需要内存，Reduce计算也要内存，所以要根据作业的运行情况进行调整。

IO传输

- 1) 采用数据压缩的方式，减少网络IO的时间。安装Snappy和LZO压缩编码器。
- 2) 使用SequenceFile二进制文件。（这种二进制文件比较紧凑）

数据倾斜问题

- 1、可以事先做好数据的抽样调查，设置合理的分区边界
- 2、某些特殊情况的数据较多，就抽出一个reduce单独处理
- 3、进行combine精简数据
- 4、尽量用map join不用reduce join

常用的调优参数

MR相关参数 (mapred-default.xml)

配置参数	参数说明
mapreduce.map.memory.mb	一个MapTask可使用的资源上限（单位:MB），默认为1024。如果MapTask实际使用的资源量超过该值，则会被强制杀死。
mapreduce.reduce.memory.mb	一个ReduceTask可使用的资源上限（单位:MB），默认为1024。如果ReduceTask实际使用的资源量超过该值，则会被强制杀死。
mapreduce.map.cpu.vcores	每个MapTask可使用的最多cpu core数目，默认值: 1
mapreduce.reduce.cpu.vcores	每个ReduceTask可使用的最多cpu core数目，默认值为1
mapreduce.reduce.shuffle.parallelcopies	每个Reduce去Map中取数据的并行数。默认值是5
mapreduce.reduce.shuffle.merge.percent	Buffer中的数据达到多少比例开始写入磁盘。默认值0.66
mapreduce.reduce.shuffle.input.buffer.percent	Buffer大小占Reduce可用内存的比例。默认值0.7

配置参数	参数说明
mapreduce.reduce.input.buffer.percent	指定多少比例的内存用来存放Buffer中的数据，默认值是0.0

yarn相关参数 (yarn-default.xml)

配置参数	参数说明
yarn.scheduler.minimum-allocation-mb	给应用程序Container分配的最小内存，默认值：1024
yarn.scheduler.maximum-allocation-mb	给应用程序Container分配的最大内存，默认值8192
yarn.scheduler.minimum-allocation-vcores	每个Container申请的最小CPU核数，默认值：1
yarn.scheduler.maximum-allocation-vcores	每个Container申请的最大CPU核数，默认值：32
yarn.nodemanager.resource.memory-mb	给Containers分配的最大物理内存，默认值：8192

Shuffle性能优化 (mapred-default.xml)

配置参数	参数说明
mapreduce.task.io.sort.mb	Shuffle的环形缓冲区大小，默认100m
mapreduce.map.sort.spill.percent	环形缓冲区溢出的阈值，默认80%

容错相关参数 (mapred-default.xml)

配置参数	参数说明
mapreduce.map.maxattempts	每个Map Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。
mapreduce.reduce.maxattempts	每个Reduce Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。
mapreduce.task.timeout	Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个Task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该Task处于Block状态，可能是卡住了，也许永远会卡住，为了防止因为用户程序永远Block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是600000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示是“AttemptID:attempt_14267829456721_123456_m_000224_0 Timed out after 300 secsContainer killed by the ApplicationMaster.”。

小文件问题

HDFS上每个文件都要在NameNode上建立一个索引，这个索引的大小约为150byte，这样当小文件比较多的时候，就会产生很多的索引文件，一方面会大量占用NameNode的内存空间，另一方面就是索引文件过大使得索引速度变慢。而且小文件会使map方法反复加载，导致程序运行速度慢。

解决方案：

- 1、在数据采集的时候，就将小文件或小批数据合成大文件再上传HDFS。如归档Hadoop Archive。
- 2、在业务处理之前，在HDFS上使用MapReduce程序对小文件进行合并。如将多个小文件合并成Sequence File
- 3、在MapReduce处理时，可采用CombineTextInputFormat提高效率。它可以把多个小文件合并为一个切片。
- 4、开启JVM重用。对于大量小文件Job，可以开启JVM重用会减少45%运行时间。

JVM重用：一个Map运行在一个JVM上，开启重用的话，该Map在JVM上运行完毕后，JVM继续运行其他Map。

具体设置方法：调整参数mapreduce.job.jvm.numtasks (jvm线程池的中重用jvm的个数) 值在10-20之间。

这是因为开启关闭JVM的时间比处理小文件的时间都长，故在小文件多时降低jvm的启动关闭次数可以提高效率。

案例

扩展案例

倒排索引案例（多job串联）

需求分析

有大量的文本（文档、网页），需要建立搜索索引，具体需求如下：

输入数据：

a.txt

```
atguigu pingping  
atguigu ss  
atguigu ss
```

b.txt

```
atguigu pingping  
atguigu pingping  
pingping ss
```

c.txt

```
atguigu ss  
atguigu pingping
```

输出数据：

```
atguigu c.txt-->2 b.txt-->2 a.txt-->3
pingping    c.txt-->1 b.txt-->3 a.txt-->1
ss   c.txt-->1 b.txt-->1 a.txt-->2
```

这种需求很难在一次mapreduce中完成，只能一次完成词频计数，一次合并（最后一次一定是所有词频汇聚到同一个key为单词的reduce中），第一次预期输出结果：

```
atguigu--a.txt 3
atguigu--b.txt 2
atguigu--c.txt 2
pingping--a.txt 1
pingping--b.txt 3
pingping--c.txt 1
ss--a.txt 2
ss--b.txt 1
ss--c.txt 1
```

第二次预期输出结果：

```
atguigu c.txt-->2 b.txt-->2 a.txt-->3
pingping    c.txt-->1 b.txt-->3 a.txt-->1
ss   c.txt-->1 b.txt-->1 a.txt-->2
```

所谓的串联job不是一口气执行的，而是分次执行，手动整理输出目录。

job1的OneIndexMapper类

mapper输入的kv是偏移量和文件的一行，输出的kv是"单词--文件名"和词频1，在setup中取到文件名，然后赋值给类变量，在map中取到文件的一行，然后设置kv到context。（在setup方法中可以通过context获取configuration对象，如果在驱动类启动时带一个参数，在main方法将该参数设置进configuration中，然后在setup方法中取到该值，这样就能实现启动时参数决定mapreduce程序运行）

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class OneIndexMapper extends Mapper<LongWritable, Text, Text,
IntWritable>{

    String name;
    Text k = new Text();
    IntWritable v = new IntWritable();

    @Override
    protected void setup(Context context) throws IOException,
InterruptedException {

        // 获取文件名称
        FileSplit split = (FileSplit) context.getInputSplit();

        name = split.getPath().getName();
    }
}
```

```

@Override
protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

    // 1 获取1行
    String line = value.toString();

    // 2 切割
    String[] fields = line.split(" ");

    for (String word : fields) {

        // 3 拼接
        k.set(word+"--"+name);
        v.set(1);

        // 4 写出
        context.write(k, v);
    }
}
}

```

job1的OneIndexReducer类

在reduce中进行词频的合并，这样输出的数据就是"atguigu--a.txt 3"

```

import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class OneIndexReducer extends Reducer<Text, IntWritable, Text,
IntWritable>{

    IntWritable v = new IntWritable();

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {

        int sum = 0;

        // 1 累加求和
        for(IntWritable value: values){
            sum +=value.get();
        }

        v.set(sum);

        // 2 写出
        context.write(key, v);
    }
}

```

job1的OneIndexDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class OneIndexDriver {

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

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[] { "e:/input/inputoneindex", "e:/output5" };

        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);
        job.setJarByClass(OneIndexDriver.class);

        job.setMapperClass(OneIndexMapper.class);
        job.setReducerClass(OneIndexReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

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

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
    }
}

```

job2的TwoIndexMapper类

job1输出的数据用--切割，切割前是"atguigu--a.txt 3"，切割后atguigu为key，'a.txt 3'为value传递到context。

```

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TwoIndexMapper extends Mapper<LongWritable, Text, Text, Text>{

    Text k = new Text();
    Text v = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        // 1 获取1行数据

```

```

        String line = value.toString();

        // 2用“--”切割
        String[] fields = line.split("--");

        k.set(fields[0]);
        v.set(fields[1]);

        // 3 输出数据
        context.write(k, v);
    }
}

```

job2的TwoIndexReducer类

所有相同的key都汇聚到一个reduce方法中，把制表符替换成-->然后累加字符串，即可得到最后的输出。

```

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class TwoIndexReducer extends Reducer<Text, Text, Text, Text> {

    Text v = new Text();

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        // atguigu a.txt 3
        // atguigu b.txt 2
        // atguigu c.txt 2

        // atguigu c.txt-->2 b.txt-->2 a.txt-->3

        StringBuilder sb = new StringBuilder();

        // 1 拼接
        for (Text value : values) {
            sb.append(value.toString().replace("\t", "-->") + "\t");
        }

        v.set(sb.toString());

        // 2 写出
        context.write(key, v);
    }
}

```

job2的TwoIndexDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;

```

```

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

public class TwoIndexDriver {

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

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[] { "e:/input/inputtwoindex", "e:/output6" };

        Configuration config = new Configuration();
        Job job = Job.getInstance(config);

        job.setJarByClass(TwoIndexDriver.class);
        job.setMapperClass(TwoIndexMapper.class);
        job.setReducerClass(TwoIndexReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        TextInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        boolean result = job.waitForCompletion(true);
        System.exit(result?0:1);
    }
}

```

TopN案例（巧用treemap）

需求分析

输出流量使用量在前10的用户信息。

输出数据如下，一行数据分别是手机号、上行流量、下行流量和总流量

手机号	上行流量	下行流量	总流量
13470253144	180	180	360
13509468723	7335	110349	117684
13560439638	918	4938	5856
13568436656	3597	25635	29232
13590439668	1116	954	2070
13630577991	6960	690	7650
13682846555	1938	2910	4848
13729199489	240	0	240
13736230513	2481	24681	27162
13768778790	120	120	240
13846544121	264	0	264
13956435636	132	1512	1644
13966251146	240	0	240
13975057813	11058	48243	59301
13992314666	3008	3720	6728
15043685818	3659	3538	7197
15910133277	3156	2936	6092

```
15959002129 1938    180 2118
18271575951 1527    2106    3633
18390173782 9531    2412    11943
84188413     4116    1432    5548
```

输出数据：

```
13509468723 7335    110349  117684
13975057813 11058   48243   59301
13568436656 3597   25635   29232
13736230513 2481   24681   27162
18390173782 9531   2412    11943
13630577991 6960   690    7650
15043685818 3659   3538    7197
13992314666 3008   3720    6728
15910133277 3156   2936    6092
13560439638 918    4938    5856
```

从输出的数据来看，必须把三个流量封装成一个bean对象。

FlowBean类

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.WritableComparable;

public class FlowBean implements WritableComparable<FlowBean>{

    private long upFlow;
    private long downFlow;
    private long sumFlow;

    public FlowBean() {
        super();
    }

    public FlowBean(long upFlow, long downFlow) {
        super();
        this.upFlow = upFlow;
        this.downFlow = downFlow;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeLong(upFlow);
        out.writeLong(downFlow);
        out.writeLong(sumFlow);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
```

```
        upFlow = in.readLong();
        downFlow = in.readLong();
        sumFlow = in.readLong();
    }

    public long getUpFlow() {
        return upFlow;
    }

    public void setUpFlow(long upFlow) {
        this.upFlow = upFlow;
    }

    public long getDownFlow() {
        return downFlow;
    }

    public void setDownFlow(long downFlow) {
        this.downFlow = downFlow;
    }

    public long getSumFlow() {
        return sumFlow;
    }

    public void setSumFlow(long sumFlow) {
        this.sumFlow = sumFlow;
    }

    @Override
    public String toString() {
        return upFlow + "\t" + downFlow + "\t" + sumFlow;
    }

    public void set(long downFlow2, long upFlow2) {
        downFlow = downFlow2;
        upFlow = upFlow2;
        sumFlow = downFlow2 + upFlow2;
    }

    @Override
    public int compareTo(FlowBean bean) {

        int result;

        if (this.sumFlow > bean.getSumFlow()) {
            result = -1;
        } else if (this.sumFlow < bean.getSumFlow()) {
            result = 1;
        } else {
            result = 0;
        }

        return result;
    }
}
```

TopNMapper类

把需要排序的bean对象设置为mapper的输出k，v就是电话号码。在map方法中进行bean的封装，然后把bean和电话号码装入treemap中，同时检查treemap容量如果大于10就删除最小的一对kv。在cleanup中遍历treemap，然后将数据装入context中。

```
import java.io.IOException;
import java.util.Iterator;
import java.util.TreeMap;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TopNMapper extends Mapper<LongWritable, Text, FlowBean, Text>{

    // 定义一个TreeMap作为存储数据的容器（天然按key排序）
    private TreeMap<FlowBean, Text> flowMap = new TreeMap<FlowBean, Text>();
    private FlowBean kBean;

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException {

        kBean = new FlowBean();
        Text v = new Text();

        // 1 获取一行
        String line = value.toString();

        // 2 切割
        String[] fields = line.split("\t");

        // 3 封装数据
        String phoneNum = fields[0];
        long upFlow = Long.parseLong(fields[1]);
        long downFlow = Long.parseLong(fields[2]);
        long sumFlow = Long.parseLong(fields[3]);

        kBean.setDownFlow(downFlow);
        kBeansetUpFlow(upFlow);
        kBean.setSumFlow(sumFlow);

        v.set(phoneNum);

        // 4 向TreeMap中添加数据
        flowMap.put(kBean, v);

        // 5 限制TreeMap的数据量，超过10条就删除掉流量最小的一条数据
        if (flowMap.size() > 10) {
            flowMap.remove(flowMap.firstKey());
            flowMap.remove(flowMap.lastKey());
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException,
    InterruptedException {
```

```

// 6 遍历treeMap集合，输出数据
Iterator<FlowBean> bean = flowMap.keySet().iterator();

while (bean.hasNext()) {

    FlowBean k = bean.next();

    context.write(k, flowMap.get(k));
}
}
}

```

TopNReducer类

reduce的逻辑和mapper基本相同，这里主要是为了减少reduce端接受数据的量级，所以把主要逻辑在mapper实现，如果不考虑reduce端的性能，也可以去掉mapper的集合相关操作。

```

import java.io.IOException;
import java.util.Iterator;
import java.util.TreeMap;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TopNReducer extends Reducer<FlowBean, Text, Text, FlowBean> {

    // 定义一个TreeMap作为存储数据的容器（天然按key排序）
    TreeMap<FlowBean, Text> flowMap = new TreeMap<FlowBean, Text>();

    @Override
    protected void reduce(FlowBean key, Iterable<Text> values, Context
context) throws IOException, InterruptedException {

        for (Text value : values) {

            FlowBean bean = new FlowBean();
            bean.set(key.getDownFlow(), key.getUpFlow());

            // 1 向treeMap集合中添加数据
            flowMap.put(bean, new Text(value));

            // 2 限制TreeMap数据量，超过10条就删除掉流量最小的一条数据
            if (flowMap.size() > 10) {
                // flowMap.remove(flowMap.firstKey());
                flowMap.remove(flowMap.lastKey());
            }
        }
    }

    @Override
    protected void cleanup(Reducer<FlowBean, Text, Text, FlowBean>.Context
context) throws IOException, InterruptedException {

        // 3 遍历集合，输出数据
        Iterator<FlowBean> it = flowMap.keySet().iterator();
    }
}

```

```

        while (it.hasNext()) {

            FlowBean v = it.next();

            context.write(new Text(flowMap.get(v)), v);
        }
    }
}

```

TopNDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TopNDriver {

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

        args = new String[]{"e:/output1","e:/output3"};

        // 1 获取配置信息，或者job对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 6 指定本程序的jar包所在的本地路径
        job.setJarByClass(TopNDriver.class);

        // 2 指定本业务job要使用的mapper/Reducer业务类
        job.setMapperClass(TopNMapper.class);
        job.setReducerClass(TopNReducer.class);

        // 3 指定mapper输出数据的kv类型
        job.setMapOutputKeyClass(FlowBean.class);
        job.setMapOutputValueClass(Text.class);

        // 4 指定最终输出的数据的kv类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        // 5 指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行
        boolean result = job.waitForCompletion(true);
        System.exit(result ? 0 : 1);
    }
}

```

共同好友案例

需求分析

以下是博客的好友列表数据，冒号前是一个用户，冒号后是该用户的所有好友（数据中的好友关系是单向的，A是B的好友，但是B可能不是A的好友）

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

输入数据：

```
A:B,C,D,F,E,O  
B:A,C,E,K  
C:F,A,D,I  
D:A,E,F,L  
E:B,C,D,M,L  
F:A,B,C,D,E,O,M  
G:A,C,D,E,F  
H:A,C,D,E,O  
I:A,O  
J:B,O  
K:A,C,D  
L:D,E,F  
M:E,F,G  
O:A,H,I,J
```

输出数据：

```
A-B E C  
A-C D F  
A-D E F  
A-E D B C  
A-F O B C D E  
A-G F E C D  
A-H E C D O  
A-I O  
A-J O B  
A-K D C  
A-L F E D  
A-M E F  
B-C A  
B-D A E  
B-E C  
B-F E A C  
B-G C E A  
B-H A E C  
B-I A  
B-K C A  
B-L E  
B-M E  
B-O A  
C-D A F  
C-E D  
C-F D A  
C-G D F A  
C-H D A
```

```
C-I A
C-K A D
C-L D F
C-M F
C-O I A
D-E L
D-F A E
D-G E A F
D-H A E
D-I A
D-K A
D-L E F
D-M F E
D-O A
E-F D M C B
E-G C D
E-H C D
E-J B
E-K C D
E-L D
F-G D C A E
F-H A D O E C
F-I O A
F-J B O
F-K D C A
F-L E D
F-M E
F-O A
G-H D C E A
G-I A
G-K D A C
G-L D F E
G-M E F
G-O A
H-I O A
H-J O
H-K A C D
H-L D E
H-M E
H-O A
I-J O
I-K A
I-O A
K-L D
K-O A
L-M E F
```

想要整理出最终结果，如A、B的共同好友是E和C，这也就意味着E是A/B的好友，C是A/B的好友。如果第一次输出能把数据整理成x是y、z、h。。。的好友，第二步就能轻松将数据整理成最终格式。

OneShareFriendsMapper类

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

```

public class OneShareFriendsMapper extends Mapper<LongWritable, Text, Text,
Text>{

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, Text>.Context context)
        throws IOException, InterruptedException {

        // 1 获取一行 A:B,C,D,F,E,O
        String line = value.toString();

        // 2 切割
        String[] fields = line.split ":";

        // 3 获取person和好友
        String person = fields[0];
        String[] friends = fields[1].split ",";

        // 4 写出去
        for(String friend: friends){

            // 输出 <好友, 人>
            context.write(new Text(friend), new Text(person));
        }
    }
}

```

OneShareFriendsReducer类

```

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class OneShareFriendsReducer extends Reducer<Text, Text, Text, Text>{

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
context) throws IOException, InterruptedException {

        StringBuffer sb = new StringBuffer();

        //1 拼接
        for(Text person: values){
            sb.append(person).append(",");
        }

        //2 写出
        context.write(key, new Text(sb.toString()));
    }
}

```

OneShareFriendsDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class OneShareFriendsDriver {

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

        // 1 获取job对象
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 指定jar包运行的路径
        job.setJarByClass(OneShareFriendsDriver.class);

        // 3 指定map/reduce使用的类
        job.setMapperClass(OneShareFriendsMapper.class);
        job.setReducerClass(OneShareFriendsReducer.class);

        // 4 指定map输出的数据类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        // 5 指定最终输出的数据类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        // 6 指定job的输入原始所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 7 提交
        boolean result = job.waitForCompletion(true);

        System.exit(result?0:1);
    }
}

```

TwoShareFriendsMapper类

```

import java.io.IOException;
import java.util.Arrays;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TwoShareFriendsMapper extends Mapper<LongWritable, Text, Text,
Text>{

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

```

```

// A I,K,C,B,G,F,H,O,D,
// 友 人，人，人
String line = value.toString();
String[] friend_persons = line.split("\t");

String friend = friend_persons[0];
String[] persons = friend_persons[1].split(",");

Arrays.sort(persons);

for (int i = 0; i < persons.length - 1; i++) {

    for (int j = i + 1; j < persons.length; j++) {
        // 发出 <人-人, 好友> , 这样, 相同的“人-人”对的所有好友就会到同1个reduce
        中去
        context.write(new Text(persons[i] + "-" + persons[j]), new
Text(friend));
    }
}
}

```

TwoShareFriendsReducer类

```

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TwoShareFriendsReducer extends Reducer<Text, Text, Text, Text> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {

        StringBuffer sb = new StringBuffer();

        for (Text friend : values) {
            sb.append(friend).append(" ");
        }

        context.write(key, new Text(sb.toString()));
    }
}

```

TwoShareFriendsDriver类

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TwoShareFriendsDriver {

```

```
public static void main(String[] args) throws Exception {  
  
    // 1 获取job对象  
    Configuration configuration = new Configuration();  
    Job job = Job.getInstance(configuration);  
  
    // 2 指定jar包运行的路径  
    job.setJarByClass(TwoShareFriendsDriver.class);  
  
    // 3 指定map/reduce使用的类  
    job.setMapperClass(TwoShareFriendsMapper.class);  
    job.setReducerClass(TwoShareFriendsReducer.class);  
  
    // 4 指定map输出的数据类型  
    job.setMapOutputKeyClass(Text.class);  
    job.setMapOutputValueClass(Text.class);  
  
    // 5 指定最终输出的数据类型  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(Text.class);  
  
    // 6 指定job的输入原始所在目录  
    FileInputFormat.setInputPaths(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
    // 7 提交  
    boolean result = job.waitForCompletion(true);  
    System.exit(result?0:1);  
}  
}
```

hadoop HA

高可用HA概述

所谓HA (High Available) , 即高可用 (7*24小时不中断服务) 。

实现高可用最关键的策略是消除单点故障。Hadoop2.0之前，在HDFS集群中NameNode存在单点故障 (SPOF) , 也就是NameNode挂掉后无法取到数据。

NameNode主要在以下两个方面影响HDFS集群 :

- 1、NameNode机器发生意外，如宕机，集群将无法使用，直到管理员重启
- 2、NameNode机器需要升级，包括软件、硬件升级，此时集群也将无法使用

HA严格来说应该分成各个组件的HA机制 : HDFS的HA和YARN的HA。

HDFS HA功能通过配置Active/Standby两个NameNodes实现在集群中对NameNode的热备来解决上述问题。如果出现故障，如机器崩溃或机器需要升级维护，这时可通过此种方式将NameNode很快的切换到另外一台机器。

HDFS手动故障转移

不能简单的让Standby去监控active，如果连接不上就转为active，这个计划的缺陷在于如果仅仅是两个namenode之间连接不上，而active和其他node连接没有出问题，Standby转变为active就会发生splitbrain现象，集群同时出现两个namenode，情况很糟糕。

因此用一个新的JournalNode来存放namenode共享的元数据，JournalNode符合半数协议，节点数应该是奇数，和zookeeper一样，启动半数以上可以正常工作。JournalNode在每个节点上都应该布置。

环境配置

配置core-site.xml：

```
<configuration>
    <!-- 把两个NameNode) 的地址组装成一个集群mycluster -->
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://mycluster</value>
    </property>

    <!-- 指定hadoop运行时产生文件的存储目录 -->
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/opt/ha/hadoop-2.7.2/data/tmp</value>
    </property>
</configuration>
```

配置hdfs-site.xml：

```
<configuration>
    <!-- 完全分布式集群名称 -->
    <property>
        <name>dfs.nameservices</name>
        <value>mycluster</value>
    </property>

    <!-- 集群中NameNode节点都有哪些 -->
    <property>
        <name>dfs.ha.namenodes.mycluster</name>
        <value>nn1,nn2</value>
    </property>

    <!-- nn1的RPC通信地址 -->
    <property>
        <name>dfs.namenode.rpc-address.mycluster.nn1</name>
        <value>hadoop102:9000</value>
    </property>

    <!-- nn2的RPC通信地址 -->
    <property>
        <name>dfs.namenode.rpc-address.mycluster.nn2</name>
        <value>hadoop103:9000</value>
    </property>

    <!-- nn1的http通信地址 -->
    <property>
        <name>dfs.namenode.http-address.mycluster.nn1</name>
        <value>hadoop102:50070</value>
    </property>
```

```

</property>

<!-- nn2的http通信地址 -->
<property>
    <name>dfs.namenode.http-address.mycluster.nn2</name>
    <value>hadoop103:50070</value>
</property>

<!-- 存NameNode共享的元数据的文件系统JournalNode的存放位置 -->
<property>
    <name>dfs.namenode.shared.edits.dir</name>
<value>qjournal://hadoop102:8485;hadoop103:8485;hadoop104:8485/mycluster</value>
</property>

<!-- 配置隔离机制，即同一时刻只能有一台服务器对外响应 -->
<property>
    <name>dfs.ha.fencing.methods</name>
    <value>sshfence</value>
</property>

<!-- 使用隔离机制时需要ssh无秘钥登录-->
<property>
    <name>dfs.ha.fencing.ssh.private-key-files</name>
    <value>/home/atguigu/.ssh/id_rsa</value>
</property>

<!-- 声明journalnode服务器存储目录-->
<property>
    <name>dfs.journalnode.edits.dir</name>
    <value>/opt/ha/hadoop-2.7.2/data/jn</value>
</property>

<!-- 关闭权限检查，让hdfs中的权限失效-->
<property>
    <name>dfs.permissions.enable</name>
    <value>false</value>
</property>

<!-- 访问代理类：client, mycluster, active配置失败自动切换实现方式-->
<property>
    <name>dfs.client.failover.proxy.provider.mycluster</name>
<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider
</value>
</property>
</configuration>

```

(注意secondarynamenode已经不需要了，Standby就会取代它的功能)

hadoop的多个配置都可以放在一个文件里，但是这里分开主要是为了便于管理。

最后拷贝配置好的hadoop环境到其他节点

启动HDFS-HA集群

1、在各个JournalNode节点上，输入以下命令启动journalnode服务：

```
sbin/hadoop-daemon.sh start journalnode
```

2、在1号namenode，即nn1上，对其进行格式化，并启动nn1：

```
bin/hdfs namenode -format、sbin/hadoop-daemon.sh start namenode
```

3、在nn2上，同步nn1的元数据信息（不能格式化两次，为了clusterID一致）：

```
bin/hdfs namenode -bootstrapStandby
```

4、启动nn2：`sbin/hadoop-daemon.sh start namenode`

启动web端的HDFS，发现在设置active之前无法访问hdfs，因为默认都是备用的。

5、在nn1上，启动所有datanode：`sbin/hadoop-daemons.sh start datanode`

6、将nn1切换为Active：`bin/hdfs haadmin -transitionToActive nn1`

7、查看是否Active：`bin/hdfs haadmin -getServiceState nn1`

所谓手动故障处理就是当nn1故障后，重新启动nn1，然后此时两个namenode都是standby，然后再把nn2设置为active，不能nn1故障后直接设置nn2，为了防止splitbrain想启动必须和原来的active进行通信。

手动故障处理必须是在namenode都启动了才能进行的，如果active节点被彻底破坏无法启动，这样的方式也解决不了问题。

HDFS自动故障处理

工作要点

1、元数据管理方式需要改变：

内存中各自保存一份元数据；

Edits日志只有Active状态的NameNode节点可以做写操作；

两个NameNode都可以读取Edits；

共享的Edits放在一个共享存储中管理（qjournal和NFS两个主流实现）；

2、需要一个状态管理功能模块

实现了一个zkfailover，常驻在每一个namenode所在的节点，每一个zkfailover负责监控自己所在NameNode节点，利用zk进行状态标识，当需要进行状态切换时，由zkfailover来负责切换，切换时需要防止brain split现象的发生。

3、必须保证两个NameNode之间能够ssh无密码登录

4、隔离（Fence），即同一时刻仅仅有一个NameNode对外提供服务

工作机制

机制的核心在于借助第三方zookeeper来监控nn的状态，一旦连接不上就强制杀死该进程，然后启动另一个nn，这样就避免了split-brain。

自动故障转移为HDFS部署增加了两个新组件：ZooKeeper和ZKFailoverController（ZKFC）进程。

ZooKeeper是维护少量协调数据，通知客户端这些数据的改变和监视客户端故障的高可用服务。HA的自动故障转移依赖于ZooKeeper的以下功能：

1) 故障检测：集群中的每个NameNode在ZooKeeper中维护了一个持久会话，如果机器崩溃，ZooKeeper中的会话将终止，ZooKeeper通知另一个NameNode需要触发故障转移。

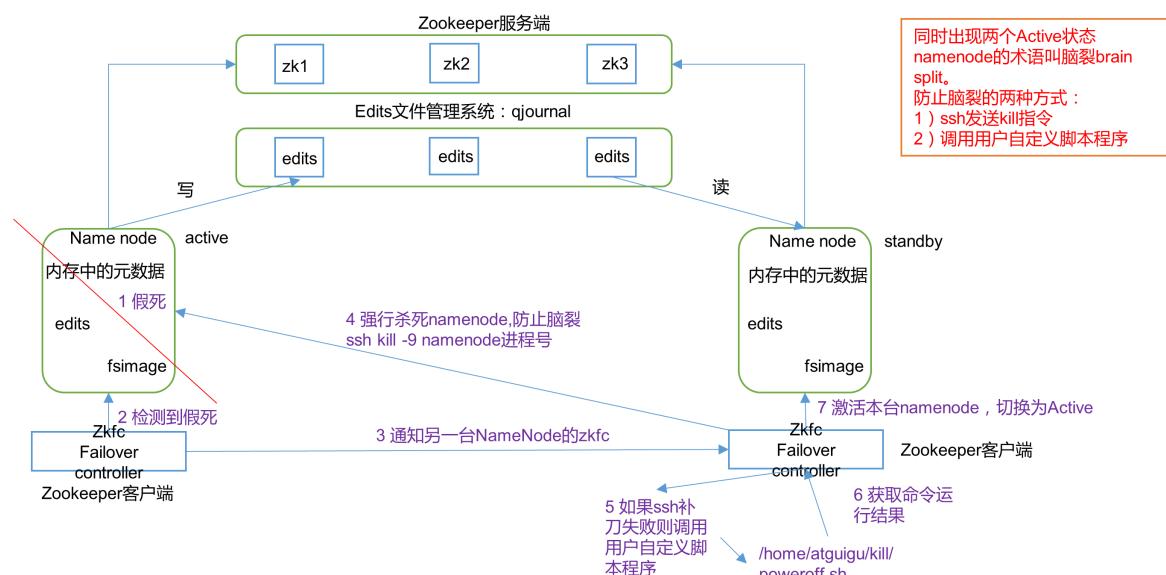
2) 现役NameNode选择：ZooKeeper提供了一个简单的机制用于唯一的选择一个节点为active状态。如果目前现役NameNode崩溃，另一个节点可能从ZooKeeper获得特殊的排外锁以表明它应该成为现役NameNode。

ZKFC是自动故障转移中的另一个新组件，是ZooKeeper的客户端，也监视和管理NameNode的状态。每个运行NameNode的主机也运行了一个ZKFC进程（它是hadoop的一个进程，不是zookeeper的，可以和zookeeper进行通信），ZKFC负责：

1) 健康监测：ZKFC使用一个健康检查命令定期地ping与之在相同主机的NameNode，只要该NameNode及时地回复健康状态，ZKFC认为该节点是健康的。如果该节点崩溃，冻结或进入不健康状态，健康监测器标识该节点为非健康的。

2) ZooKeeper会话管理：当本地NameNode是健康的，ZKFC保持一个在ZooKeeper中打开的会话。如果本地NameNode处于active状态，ZKFC也保持一个特殊的znode锁，该锁使用了ZooKeeper对短暂节点的支持，如果namenode挂掉会话终止，锁节点将自动删除，其他namenode就能争取到这把锁，用这种机制控制active只有一个。

3) 基于ZooKeeper的选择：如果本地NameNode是健康的，且ZKFC发现没有其它的节点当前持有znode锁，它将为自己获取该锁。如果成功，则它已经赢得了选择，并负责运行故障转移进程以使它的本地NameNode为Active。故障转移进程与前面描述的手动故障转移相似，首先如果必要保护之前的现役NameNode，然后本地NameNode转换为Active状态。



环境配置

在hdfs-site.xml中开启自动故障处理：

```
<property>
  <name>dfs.ha.automatic-failover.enabled</name>
  <value>true</value>
</property>
```

在core-site.xml文件中增加zookeeper的位置：

```
<property>
  <name>ha.zookeeper.quorum</name>
  <value>hadoop102:2181,hadoop103:2181,hadoop104:2181</value>
</property>
```

启动HDFS-HA集群

- (1) 关闭所有HDFS服务 : `sbin/stop-dfs.sh`
 - (2) 启动Zookeeper集群 : `bin/zkServer.sh start`
 - (3) 初始化HA在Zookeeper中状态 : `bin/hdfs zkfc -formatZK`
 - (4) 启动HDFS服务 : `sbin/start-dfs.sh`
- (5) 在各个NameNode节点上启动DFSZK Failover Controller , 先在哪台机器启动 , 哪个机器的NameNode就是Active NameNode : `sbin/hadoop-daemon.sh start zkfc`

验证 :

- (1) 将Active NameNode进程kill : `kill -9 namenode的进程id`
- (2) 将Active NameNode机器断开网络 : `service network stop`

YARN-HA配置

配置两个resourcemanager以提高稳定性。

环境配置

配置yarn-site.xml :

```
<configuration>

    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!--启用resourcemanager ha-->
    <property>
        <name>yarn.resourcemanager.ha.enabled</name>
        <value>true</value>
    </property>

    <!--声明两台resourcemanager的地址-->
    <property>
        <name>yarn.resourcemanager.cluster-id</name>
        <value>cluster-yarn1</value>
    </property>

    <property>
        <name>yarn.resourcemanager.ha.rm-ids</name>
        <value>rm1,rm2</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm1</name>
        <value>hadoop102</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm2</name>
        <value>hadoop103</value>
    </property>
```

```

<!--指定zookeeper集群的地址-->
<property>
    <name>yarn.resourcemanager.zk-address</name>
    <value>hadoop102:2181,hadoop103:2181,hadoop104:2181</value>
</property>

<!--启用自动恢复-->
<property>
    <name>yarn.resourcemanager.recovery.enabled</name>
    <value>true</value>
</property>

<!--指定resourcemanager的状态信息存储在zookeeper集群-->
<property>
    <name>yarn.resourcemanager.store.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</va
lue>
</property>

</configuration>

```

然后同步更新其他节点的配置信息。

启动集群

启动hdfs：

1、在各个JournalNode节点上，输入以下命令启动journalnode服务：

```
sbin/hadoop-daemon.sh start journalnode
```

2、在1号namenode，即nn1上，对其进行格式化，并启动nn1：

```
bin/hdfs namenode -format、sbin/hadoop-daemon.sh start namenode
```

3、在nn2上，同步nn1的元数据信息（不能格式化两次，为了clusterID一致）：

```
bin/hdfs namenode -bootstrapStandby
```

4、启动nn2：`sbin/hadoop-daemon.sh start namenode`

启动web端的HDFS，发现在设置active之前无法访问hdfs，因为默认都是备用的。

5、在nn1上，启动所有datanode：`sbin/hadoop-daemons.sh start datanode`

6、将nn1切换为Active：`bin/hdfs haadmin -transitionToActive nn1`

启动yarn：

1、在hadoop102中执行：这样就能启动一个resourcemanager和所有nodemanager：

```
sbin/start-yarn.sh
```

(2) 在hadoop103中执行：启动剩下的那个resourcemanager：

```
sbin/yarn-daemon.sh start resourcemanager
```

(3) 查看服务状态：`bin/yarn rmadmin -getServiceState rm1`

HDFS Federation架构设计

NameNode架构的局限性：

(1) Namespace (命名空间) 的限制

由于NameNode在内存中存储所有的元数据 (metadata) , 因此单个NameNode所能存储的对象 (文件+块) 数目受到NameNode所在JVM的heap size的限制。50G的heap能够存储20亿 (200million) 个对象 , 这20亿个对象支持4000个DataNode , 12PB的存储 (假设文件平均大小为40MB) 。随着数据的飞速增长 , 存储的需求也随之增长。单个DataNode从4T增长到36T , 集群的尺寸增长到8000个DataNode。存储的需求从12PB增长到大于100PB。

(2) 隔离问题

由于HDFS仅有一个NameNode , 无法隔离各个程序 , 因此HDFS上的一个实验程序就很有可能影响整个HDFS上运行的程序。

(3) 性能的瓶颈

由于是单个NameNode的HDFS架构 , 因此整个HDFS文件系统的吞吐量受限于单个NameNode的吞吐量。

HDFS Federation实际上就是同时配置多个NameNode , 然后以负载均衡的方式向NameNode集群分发任务 , 但是这种架构比较罕见 , 因为很难达到一个NameNode不够用的这种数据量级。