

一 大数据概论

1.1 大数据概念



一、大数据概念

大数据（big data）：指无法在一定时间范围内用常规软件工具进行捕捉、管理和处理的数据集合，是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。

主要解决，海量数据的存储和海量数据的分析计算问题。

按顺序给出数据存储单位：bit、Byte、KB、MB、GB、TB、PB、EB、ZB、YB、BB、NB、DB。

1Byte = 8bit 1K = 1024Byte 1MB = 1024K
1G = 1024M 1T = 1024G 1P = 1024T

才存
100T



1.2 大数据特点



二、大数据特点

1、Volume（大量）：

截至目前，人类生产的所有印刷材料的数据量是200PB，而历史上全人类总共说过的话的数据量大约是5EB。当前，典型个人计算机硬盘的容量为TB量级，而一些大企业的数据量已经接近EB量级。



二、大数据特点

2、Velocity（高速）：

这是大数据区别于传统数据挖掘的最显著特征。根据IDC的“数字宇宙”的报告，预计到2020年，全球数据使用量将达到35.2ZB。在如此海量的数据面前，处理数据的效率就是企业的生命。

天猫双十一：2017年3分01秒，天猫交易额超过100亿



二、大数据特点

3、Variety（多样）：

这种类型的多样性也让数据被分为结构化数据和非结构化数据。相对于以往便于存储的以数据库/文本为主的结构化数据，非结构化数据越来越多，包括网络日志、音频、视频、图片、地理位置信息等，这些多类型的数据对数据的处理能力提出了更高要求。



订单数据

id	用户	日期	购买商品	购买数量
1001	canglaoshi	20170710-9:10:10	面膜	2
1002	xiaozelaoshi	20170710-9:11:20	化妆品	3
1003	boduolaoshi	20170710-9:22:50	内衣	4
1004	sslaoshi	20170710-10:12:20	海狗人参丸	100



网络日志



二、大数据特点

4、Value（低价值密度）：

价值密度的高低与数据总量的大小成反比。比如，在一天监控视频中，我们只关心宋老师晚上在床上健身那一分钟，如何快速对有价值数据“提纯”成为目前大数据背景下待解决的难题。



1.3 大数据应用场景

三、大数据能干啥

1、物流仓储：大数据分析系统助力商家精细化运营、提升销量、节约成本。



三、大数据应用场景

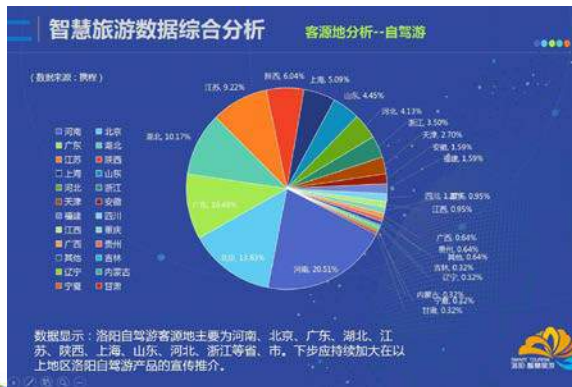
2、零售：分析用户消费习惯，为用户购买商品提供方便，从而提升商品销量。

经典案例，子尿布+啤酒。



三、大数据应用场景

3、旅游：深度结合大数据能力与旅游行业需求，共建旅游产业智慧管理、智慧服务和智慧营销的未来。



三、大数据应用场景

4、商品广告推荐：给用户推荐可能喜欢的商品

我选了一种药，又推荐了8种，太棒了，么么哒！



三、大数据应用场景

5、保险：海量数据挖掘及风险预测，助力保险行业精准营销，提升精细化定价能力。

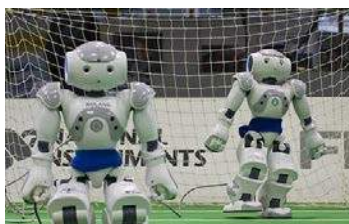
6、金融：多维度体现用户特征，帮助金融机构推荐优质客户，防范欺诈风险。

7、房产：大数据全面助力房地产行业，打造精准投策与营销，选出更合适的地，建造更合适的楼，卖给更合适的人。



三、大数据应用场景

8、人工智能：



1.4 大数据发展前景

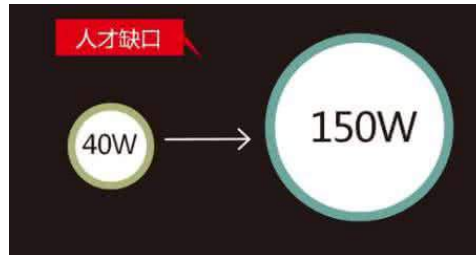
四、大数据发展前景

1、党的十八届五中全会提出“实施国家大数据战略”，国务院印发《促进大数据发展行动纲要》，大数据技术和应用处于创新突破期，国内市场需求处于爆发期，我国大数据产业面临重要的发展机遇。



四、大数据发展前景

2、国际数据公司IDC预测，到2020年，企业基于大数据计算分析平台的支出将突破5000亿美元。目前，我国大数据人才只有46万，未来3到5年人才缺口达150万之多。



人才缺口计算

$$150w - 40w = 110w$$

$$110w / 5 \text{年} = 22w/\text{年}$$

$$22w / 12 \text{月} = 1.83w/\text{月}$$

自古不变的真理：先入行者吃肉，后入行者喝汤，最后到的买单！

四、大数据发展前景

3、2017年北京大学、中国人民大学、北京邮电大学等25所高校成功申请开设大数据课程。



4、大数据属于高新技术，大牛少，升职竞争小；

四、大数据发展前景

5、在北京大数据开发工程师的平均薪水已经到了17800元（数据统计来职友集），而且目前还保持强劲的发展势头。



四、大数据发展前景

6、智联招聘网站上的大数据工程师薪水如下

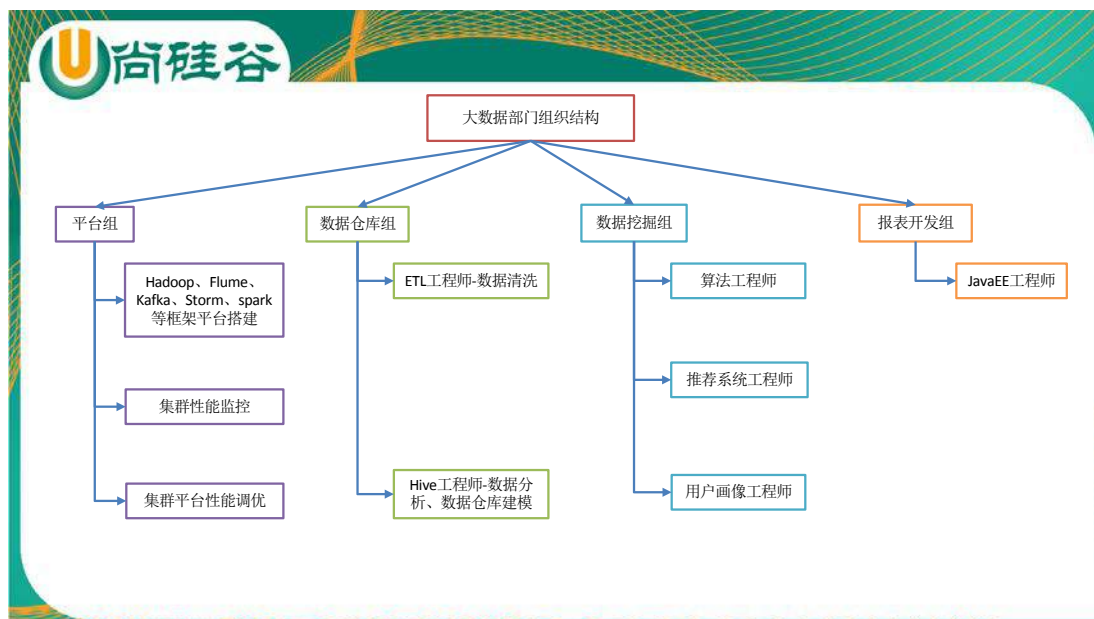
<input type="checkbox"/> 软件工程师 (Java/大数据方向)		中金数据系统有限公司	15001-20000	北京	03-11	▼
<input checked="" type="checkbox"/> 全国各地大区大数据高级客户经理	97%	广州市冠升网络科技有限公司	20000-24999	北京	03-11	▼
<input type="checkbox"/> 大数据算法工程师	79%	北京三好互动教育科技有限公司	15001-20000	北京	03-11	▼
<input type="checkbox"/> 大数据开发工程师	66%	北京腾信软创信息技术有限公司	15000-20000	北京	03-11	▼
<input type="checkbox"/> 高级PHP开发工程师 (大数据)		万科链家 (北京) 装饰工程有限公司	15000-30000	北京	03-11	▼
<input type="checkbox"/> 大数据分析工程师		北京百通无限网络技术有限公司	15001-20000	北京	03-11	▼
<input type="checkbox"/> 高级咨询顾问-财务、大数据、金融、信息化方向		远光软件股份有限公司北京分公司	20000-40000	北京	03-11	▼
<input type="checkbox"/> 大数据系统/算法工程师/数据工程师		北京知趣科技有限公司	15001-20000	北京	03-11	▼
<input type="checkbox"/> 大数据分析工程师		中金云金融 (北京) 大数据科技股份有限公司	12000-18000	北京-朝阳区	03-11	▼
<input type="checkbox"/> 大数据平台架构师 (java)		中金云金融 (北京) 大数据科技股份有限公司	20000-28000	北京	03-11	▼
<input type="checkbox"/> 大数据工程师-2237		完美世界 (北京) 软件有限公司 BIST	15000-25000	北京-朝阳区	03-11	▼

1.5 大数据部门业务流程分析



1.6 大数据部门组织结构

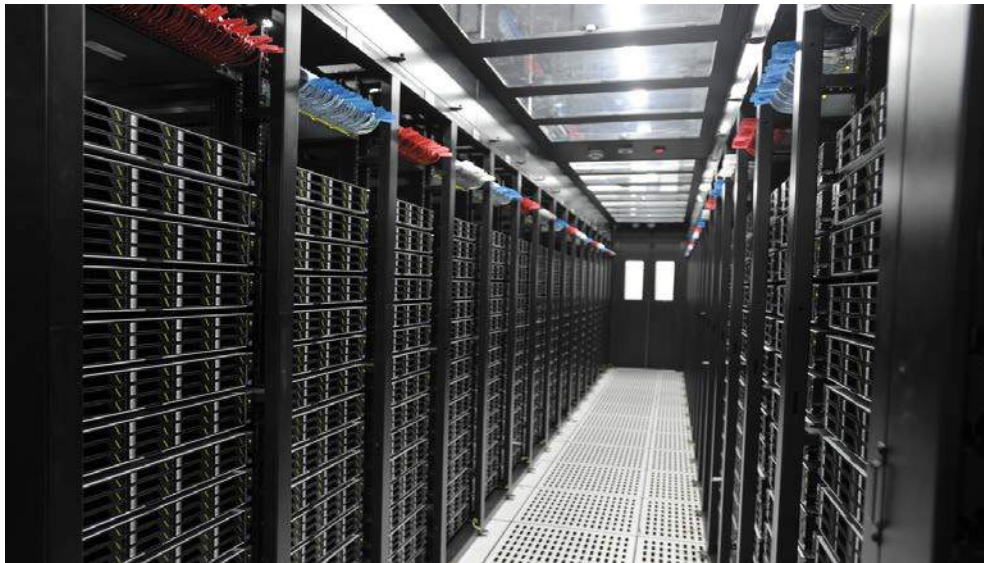
大数据部门组织结构，适用于大中型企业。



二 从 Hadoop 框架讨论大数据生态

2.1 Hadoop 是什么

- 1) Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构。
- 2) 主要解决，海量数据的存储和海量数据的分析计算问题。
- 3) 广义上来说，HADOOP 通常是指一个更广泛的概念——HADOOP 生态圈。



2.2 Hadoop 发展历史

- 1) Lucene 是 Doug Cutting 开创的开源软件，用 java 书写代码，实现与 Google 类似的全文搜索功能，它提供了全文检索引擎的架构，包括完整的查询引擎和索引引擎
- 2) 2001 年年底成为 Apache 基金会的一个子项目
- 3) 对于大数量的场景，Lucene 面对与 Google 同样的困难
- 4) 学习和模仿 Google 解决这些问题的办法：微型版 Nutch
- 5) 可以说 Google 是 hadoop 的思想之源(Google 在大数据方面的三篇论文)

GFS --->HDFS

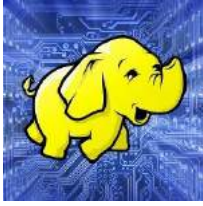
Map-Reduce --->MR

BigTable --->Hbase

- 6) 2003-2004 年, Google 公开了部分 GFS 和 Mapreduce 思想的细节, 以此为基础 Doug Cutting 等人用了 2 年业余时间实现了 DFS 和 Mapreduce 机制, 使 Nutch 性能飙升
- 7) 2005 年 Hadoop 作为 Lucene 的子项目 Nutch 的一部分正式引入 Apache 基金会。2006 年 3 月份, Map-Reduce 和 Nutch Distributed File System (NDFS) 分别被纳入称为 Hadoop 的项

目中

8) 名字来源于 Doug Cutting 儿子的玩具大象



9) Hadoop 就此诞生并迅速发展，标志这云计算时代来临

2.3 Hadoop 三大发行版本

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

Apache 版本最原始（最基础）的版本，对于入门学习最好。

Cloudera 在大型互联网企业中用的较多。

Hortonworks 文档较好。

1) Apache Hadoop

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

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

2) 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/>

(1) 2008 年成立的 Cloudera 是最早将 Hadoop 商用的公司，为合作伙伴提供 Hadoop 的商用解决方案，主要是包括支持、咨询服务、培训。

(2) 2009 年 Hadoop 的创始人 Doug Cutting 也加盟 Cloudera 公司。Cloudera 产品主要为 CDH, Cloudera Manager, Cloudera Support

(3) CDH 是 Cloudera 的 Hadoop 发行版，完全开源，比 Apache Hadoop 在兼容性，安全性，稳定性上有所增强。

(4) Cloudera Manager 是集群的软件分发及管理监控平台，可以在几个小时内部署好一个 Hadoop 集群，并对集群的节点及服务进行实时监控。Cloudera Support 即是对 Hadoop 的技术支持。

(5) Cloudera 的标价为每年每个节点 4000 美元。Cloudera 开发并贡献了可实时处理大数据的 Impala 项目。

3) Hortonworks Hadoop

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

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

(1) 2011 年成立的 Hortonworks 是雅虎与硅谷风投公司 Benchmark Capital 合资组建。

(2) 公司成立之初就吸纳了大约 25 名至 30 名专门研究 Hadoop 的雅虎工程师, 上述工程师均在 2005 年开始协助雅虎开发 Hadoop, 贡献了 Hadoop 80% 的代码。

(3) 雅虎工程副总裁、雅虎 Hadoop 开发团队负责人 Eric Baldeschwieler 出任 Hortonworks 的首席执行官。

(4) Hortonworks 的主打产品是 Hortonworks Data Platform (HDP), 也同样是 100% 开源的产品, HDP 除常见的项目外还包括了 Ambari, 一款开源的安装和管理系统。

(5) HCatalog, 一个元数据管理系统, HCatalog 现已集成到 Facebook 开源的 Hive 中。Hortonworks 的 Stinger 开创性的极大的优化了 Hive 项目。Hortonworks 为入门提供了一个非常好的, 易于使用的沙盒。

(6) Hortonworks 开发了很多增强特性并提交至核心主干, 这使得 Apache Hadoop 能够在包括 Window Server 和 Windows Azure 在内的 microsoft Windows 平台上本地运行。定价以集群为基础, 每 10 个节点每年为 12500 美元。

2.4 Hadoop 的优势

- 1) 高可靠性: Hadoop 底层维护多个数据副本, 所以即使 Hadoop 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, 可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, 以加快任务处理速度。
- 4) 高容错性: 能够自动将失败的任务重新分配。

2.5 Hadoop 组成

在 Hadoop 1.x 时代, Hadoop 中的 MapReduce 同时处理业务逻辑运算和资源的调度, 耦合性较大。

Hadoop1.x组成

Common（辅助工具）

MapReduce（资源调度+计算）

HDFS（数据存储）

1) Hadoop HDFS：一个高可靠、高吞吐量的分布式文件系统。

2) Hadoop MapReduce：一个分布式的资源调度和离线并行计算框架。

3) Hadoop Common：支持其他模块的工具模块（Configuration、RPC、序列化机制、日志操作）。

在 Hadoop2.x 时代，增加了 Yarn。Yarn 只负责资源的调度，MapReduce 只负责运算。

Hadoop2.x组成

MapReduce（计算）

Yarn（资源调度）

HDFS（数据存储）

C
o
m
m
o
n
（
辅
助
工
具
）

1) Hadoop HDFS：一个高可靠、高吞吐量的分布式文件系统。

2) Hadoop YARN：作业调度与集群资源管理的框架。

3) Hadoop MapReduce：一个分布式的离线并行计算框架。

4) Hadoop Common：支持其他模块的工具模块（Configuration、RPC、序列化机制、日志操作）。

2.5.1 HDFS 架构概述



HDFS架构概述

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。




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。



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。

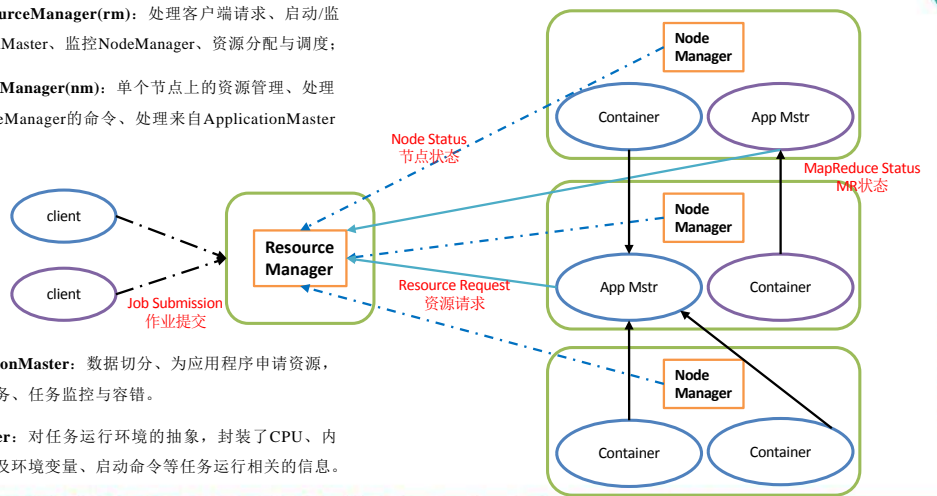
2.5.2 YARN 架构概述



YARN架构

1) Resource Manager(rm): 处理客户端请求、启动/监控ApplicationMaster、监控NodeManager、资源分配与调度;

2) NodeManager(nm): 单个节点上的资源管理、处理来自Resource Manager的命令、处理来自ApplicationMaster的命令;



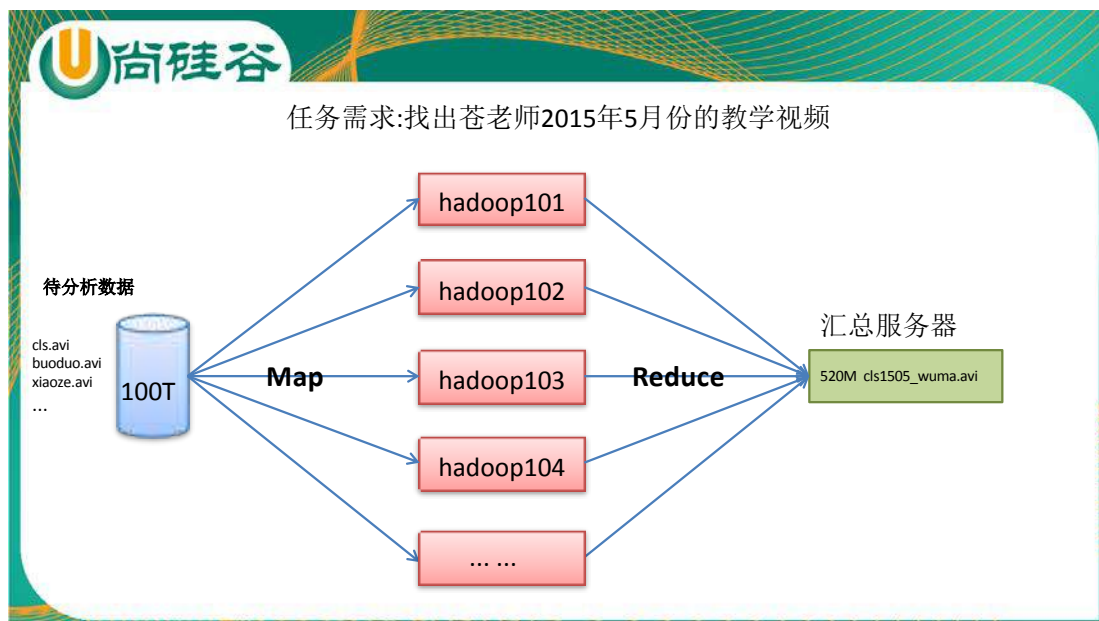
3) ApplicationMaster: 数据切分、为应用程序申请资源, 并分配给内部任务、任务监控与容错。

4) Container: 对任务运行环境的抽象, 封装了CPU、内存等多维资源以及环境变量、启动命令等任务运行相关的信息。

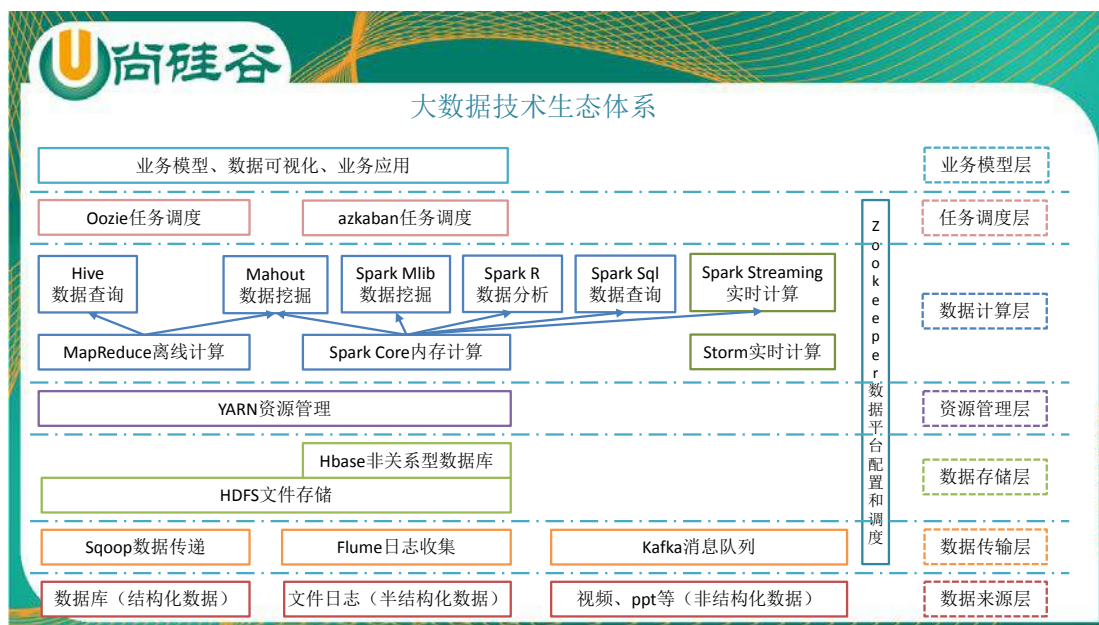
2.5.3 MapReduce 架构概述

MapReduce 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



2.6 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** 为分布式实时计算提供了一组通用原语，可被用于“流处理”之中，实时处理消息并更新数据库。这是管理队列及工作者集群的另一种方式。 **Storm** 也可被用于“连续计算” (continuous computation)，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。 **Oozie** 协调作业就是通过时间 (频率) 和有效数据触发当前的 **Oozie** 工作流程。

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，当前 **Mahout** 支持主要的 4 个用例：

推荐挖掘：搜集用户动作并以此给用户推荐可能喜欢的事物。

聚集：收集文件并进行相关文件分组。

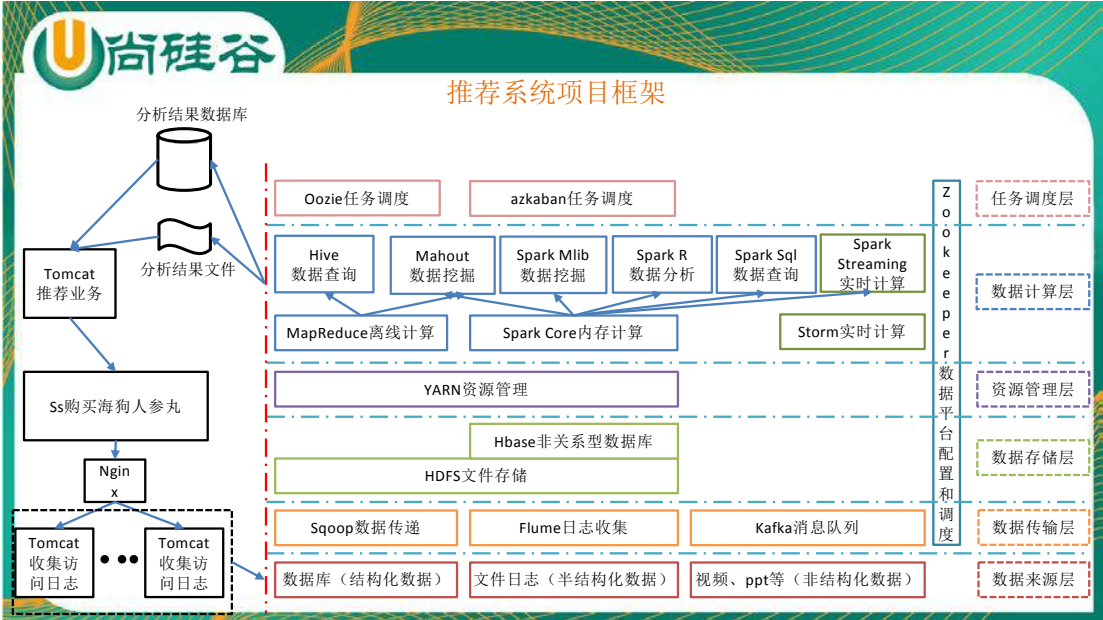
分类：从现有的分类文档中学习，寻找文档中的相似特征，并为无标签的文档进行正确的归类。

频繁项集挖掘：将一组项分组，并识别哪些个别项会经常一起出现。

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。

2.7 推荐系统框架图



三 Hadoop 运行环境搭建

3.1 虚拟机环境准备

- 1) 克隆虚拟机
- 2) 修改克隆虚拟机的静态 ip
- 3) 修改主机名
- 4) 关闭防火墙
- 5) 创建 atguigu 用户
- 6) 配置 atguigu 用户具有 root 权限

以上 6 条内容详见《尚硅谷大数据技术之 Linux》文档。

- 7) 在/opt 目录下创建文件夹

- (1) 在/opt 目录下创建 module、software 文件夹

```
[atguigu@hadoop101 opt]$ sudo mkdir module
```

```
[atguigu@hadoop101 opt]$ sudo mkdir software
```

- (2) 修改 module、software 文件夹的所有者

```
[atguigu@hadoop101 opt]$ sudo chown atguigu:atguigu module/ software/
```

```
[atguigu@hadoop101 opt]$ ll
```

```
总用量 8
```

```
drwxr-xr-x. 2 atguigu atguigu 4096 1 月 17 14:37 module
```

```
drwxr-xr-x. 2 atguigu atguigu 4096 1 月 17 14:38 software
```

3.2 安装 jdk

- 1) 卸载现有 jdk

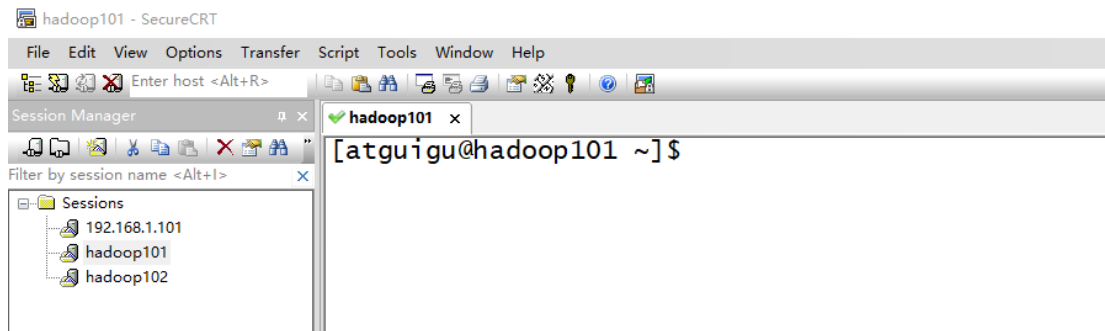
- (1) 查询是否安装 java 软件:

```
[atguigu@hadoop101 opt]$ rpm -qa | grep java
```

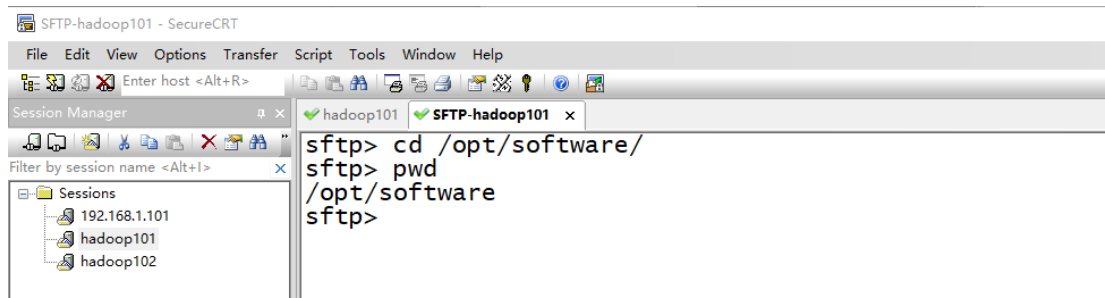
- (2) 如果安装的版本低于 1.7, 卸载该 jdk:

```
[atguigu@hadoop101 opt]$ sudo rpm -e 软件包
```

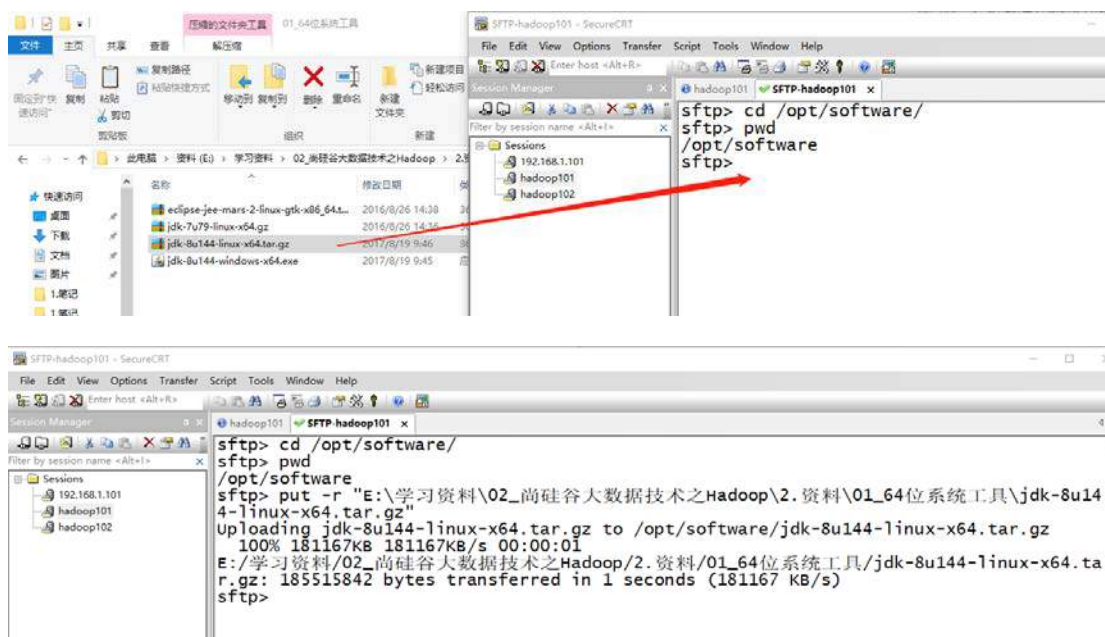
- 2) 用 SecureCRT 工具将 jdk 导入到 opt 目录下面的 software 文件夹下面



“alt+p”进入 sftp 模式



选择 jdk1.8 拖入



3) 在 linux 系统下的 opt 目录中查看软件包是否导入成功。

[atguigu@hadoop101 opt]\$ cd software/

[atguigu@hadoop101 software]\$ ls

hadoop-2.7.2.tar.gz jdk-8u144-linux-x64.tar.gz

4) 解压 jdk 到/opt/module 目录下

[atguigu@hadoop101 software]\$ tar -zxvf jdk-8u144-linux-x64.tar.gz -C /opt/module/

5) 配置 jdk 环境变量

(1) 先获取 jdk 路径:

```
[atgui@hadoop101 jdk1.8.0_144]$ pwd  
/opt/module/jdk1.8.0_144
```

(2) 打开/etc/profile 文件:

```
[atguigu@hadoop101 software]$ sudo vi /etc/profile
```

在 profile 文件末尾添加 jdk 路径:

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

(3) 保存后退出:

```
:wq
```

(4) 让修改后的文件生效:

```
[atguigu@hadoop101 jdk1.8.0_144]$ source /etc/profile
```

6) 测试 jdk 是否安装成功:

```
[atguigu@hadoop101 jdk1.8.0_144]# java -version  
java version "1.8.0_144"
```

注意: 重启 (如果 java -version 可以用就不用重启)

```
[atguigu@hadoop101 jdk1.8.0_144]$ sync
```

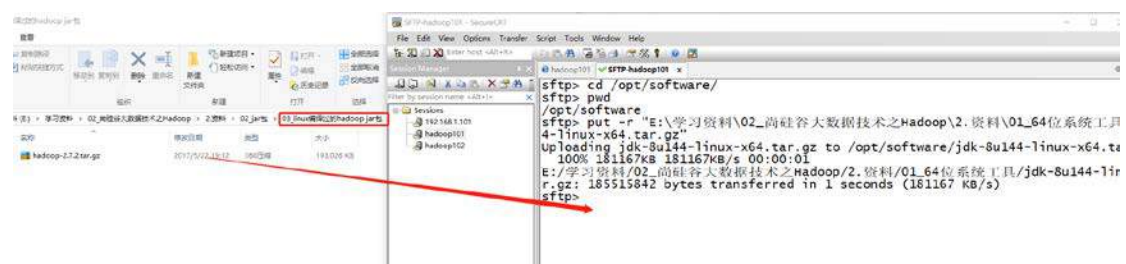
```
[atguigu@hadoop101 jdk1.8.0_144]$ sudo reboot
```

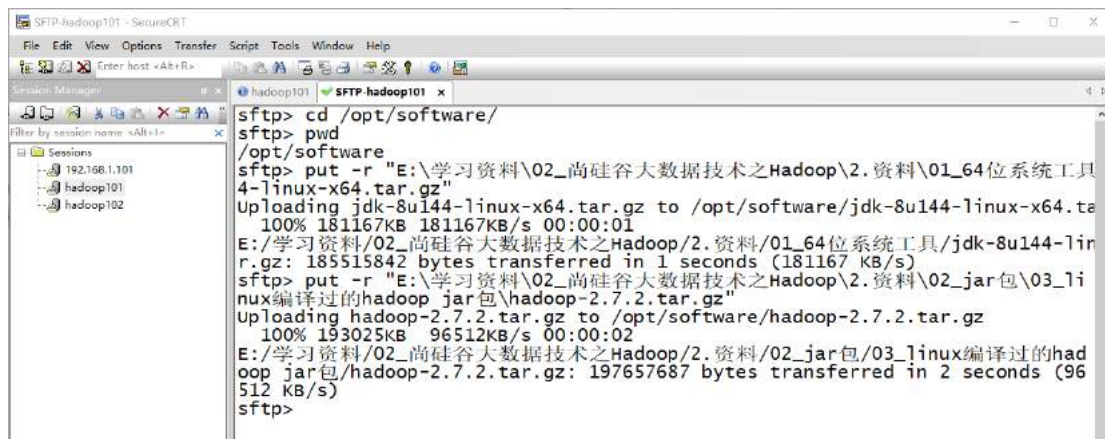
3.3 安装 Hadoop

0) Hadoop 下载地址:

<https://archive.apache.org/dist/hadoop/common/hadoop-2.7.2/>

1) 用 SecureCRT 工具将 hadoop-2.7.2.tar.gz 导入到 opt 目录下面的 software 文件夹下面
切换到 sftp 连接页面, 选择 Linux 下编译的 hadoop jar 包拖入





2) 进入到 Hadoop 安装包路径下:

```
[atguigu@hadoop101 ~]$ cd /opt/software/
```

3) 解压安装文件到/opt/module 下面

```
[atguigu@hadoop101 software]$ tar -zxvf hadoop-2.7.2.tar.gz -C /opt/module/
```

4) 查看是否解压成功

```
[atguigu@hadoop101 software]$ ls /opt/module/  
  
hadoop-2.7.2
```

5) 将 hadoop 添加到环境变量

(1) 获取 hadoop 安装路径:

```
[atguigu@hadoop101 hadoop-2.7.2]$ pwd  
  
/opt/module/hadoop-2.7.2
```

(2) 打开/etc/profile 文件:

```
[atguigu@hadoop101 hadoop-2.7.2]$ sudo vi /etc/profile
```

在 profile 文件末尾添加 jdk 路径: (shift+g)

```
##HADOOP_HOME
```

```
export HADOOP_HOME=/opt/module/hadoop-2.7.2
```

```
export PATH=$PATH:$HADOOP_HOME/bin
```

```
export PATH=$PATH:$HADOOP_HOME/sbin
```

(3) 保存后退出:

```
:wq
```

(4) 让修改后的文件生效:

```
[atguigu@hadoop101 hadoop-2.7.2]$ source /etc/profile
```

6) 测试是否安装成功

```
[atguigu@hadoop102 ~]$ hadoop version
```

```
Hadoop 2.7.2
```

7) 重启(如果 hadoop 命令不能用再重启):

```
[atguigu@hadoop101 hadoop-2.7.2]$ sync
```

```
[atguigu@hadoop101 hadoop-2.7.2]$ sudo reboot
```

四 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。

Hadoop 官方网站：<http://hadoop.apache.org/>

4.1 本地运行模式

4.1.1 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

```
[atguigu@hadoop101 hadoop-2.7.2]$ mkdir input
```

- 2) 将 hadoop 的 xml 配置文件复制到 input

```
[atguigu@hadoop101 hadoop-2.7.2]$ cp etc/hadoop/*.xml input
```

- 3) 执行 share 目录下的 mapreduce 程序

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hadoop jar
```

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

- 4) 查看输出结果

```
[atguigu@hadoop101 hadoop-2.7.2]$ cat output/*
```

4.1.2 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

```
[atguigu@hadoop101 hadoop-2.7.2]$ mkdir wcinput
```

- 2) 在 wcinput 文件下创建一个 wc.input 文件

```
[atguigu@hadoop101 hadoop-2.7.2]$ cd wcinput
```

```
[atguigu@hadoop101 wcinput]$ touch wc.input
```

- 3) 编辑 wc.input 文件

```
[atguigu@hadoop101 wcinput]$ vi wc.input
```

在文件中输入如下内容

```
hadoop yarn
```

```
hadoop mapreduce
```

```
atguigu
```

```
atguigu
```

```
保存退出:: wq
```

4) 回到 `hadoop` 目录 `/opt/module/hadoop-2.7.2`

5) 执行程序:

```
[atguigu@hadoop101 hadoop-2.7.2]$ hadoop jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount wcinput wcoutput
```

6) 查看结果:

```
[atguigu@hadoop101 hadoop-2.7.2]$ cat wcoutput/part-r-000000
atguigu 2
hadoop 2
mapreduce 1
yarn 1
```

4.2 伪分布式运行模式

4.2.1 启动 HDFS 并运行 MapReduce 程序

1) 分析:

- (1) 配置集群
- (2) 启动、测试集群增、删、查
- (3) 执行 `wordcount` 案例

2) 执行步骤

(1) 配置集群

(a) 配置: `hadoop-env.sh`

Linux 系统中获取 `jdk` 的安装路径:

```
[atguigu@hadoop101 ~]# echo $JAVA_HOME
/opt/module/jdk1.8.0_144
```

修改 `JAVA_HOME` 路径:

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

(b) 配置: `core-site.xml`

```
<!-- 指定 HDFS 中 NameNode 的地址 -->
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://hadoop101:9000</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>
```

(c) 配置: hdfs-site.xml

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

(2) 启动集群

(a) 格式化 NameNode (第一次启动时格式化, 以后就不要总格式化)

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs namenode -format
```

(b) 启动 NameNode

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/hadoop-daemon.sh start namenode
```

(c) 启动 DataNode

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/hadoop-daemon.sh start datanode
```

(3) 查看集群

(a) 查看是否启动成功

```
[atguigu@hadoop101 hadoop-2.7.2]$ jps
```

```
13586 NameNode
```

```
13668 DataNode
```

```
13786 Jps
```

(b) 查看产生的 log 日志

当前目录: /opt/module/hadoop-2.7.2/logs

```
[atguigu@hadoop101 logs]$ ls
```

```
hadoop-atguigu-datanode-hadoop.atguigu.com.log
```

```
hadoop-atguigu-datanode-hadoop.atguigu.com.out
```

```
hadoop-atguigu-namenode-hadoop.atguigu.com.log
```

```
hadoop-atguigu-namenode-hadoop.atguigu.com.out
```

```
SecurityAuth-root.audit
```

```
[atguigu@hadoop101 logs]# cat hadoop-atguigu-datanode-hadoop101.log
```

(c) web 端查看 HDFS 文件系统

<http://192.168.1.101:50070/dfshealth.html#tab-overview>

注意：如果不能查看，看如下帖子处理

<http://www.cnblogs.com/zlsich/p/6604189.html>

(4) 操作集群

(a) 在 hdfs 文件系统上**创建**一个 input 文件夹

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -mkdir -p /user/atguigu/input
```

(b) 将测试文件内容**上传**到文件系统上

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -put wcinput/wc.input  
/user/atguigu/input/
```

(c) **查看**上传的文件是否正确

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -ls /user/atguigu/input/
```

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -cat /user/atguigu/  
input/wc.input
```

(d) 运行 mapreduce 程序

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hadoop jar  
share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount /user/atguigu/input/  
/user/atguigu/output
```

(e) 查看输出结果

命令行查看：

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -cat /user/atguigu/output/*
```

浏览器查看

Browse Directory

/user/atguigu/output							Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	atguigu	supergroup	0 B	2017/12/1 上午11:05:18	1	128 MB	._SUCCESS
-rw-r--r--	atguigu	supergroup	38 B	2017/12/1 上午11:05:18	1	128 MB	part-r-00000

(f) 将测试文件内容**下载**到本地

```
[atguigu@hadoop101 hadoop-2.7.2]$ hadoop fs -get /user/atguigu/  
output/part-r-00000 ./wcoutput/
```

(g) **删除**输出结果

```
[atguigu@hadoop101 hadoop-2.7.2]$ hdfs dfs -rm -r /user/atguigu/output
```

4.2.2 YARN 上运行 MapReduce 程序

1) 分析:

- (1) 配置集群 yarn 上运行
- (2) 启动、测试集群增、删、查
- (3) 在 yarn 上执行 wordcount 案例

2) 执行步骤

(1) 配置集群

(a) 配置 yarn-env.sh

配置一下 JAVA_HOME

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

(b) 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>hadoop101</value>
</property>
```

(c) 配置: mapred-env.sh

配置一下 JAVA_HOME

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

(d) 配置: (对 mapred-site.xml.template 重新命名为) mapred-site.xml

```
[atguigu@hadoop101 hadoop]$ mv mapred-site.xml.template mapred-site.xml
```

```
[atguigu@hadoop101 hadoop]$ vi mapred-site.xml
```

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

(2) 启动集群

(a) 启动前必须保证 namenode 和 datanode 已经启动

(b) 启动 resourcemanager

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/yarn-daemon.sh start resourcemanager
```

(c) 启动 nodemanager

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/yarn-daemon.sh start nodemanager
```

(3) 集群操作

(a) yarn 的浏览器页面查看

<http://192.168.1.101:8088/cluster>

← ↻ 192.168.10.101:8088/cluster

🔍 ⭐ ⋮

Logged in as: dr.who

🏠

Cluster

About Nodes ApplicationsNEWNEW SAVING SUBMITTEDACCEPTEDRUNNINGFINISHEDFAILEDKILLED SchedulerTools

All Applications

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
0	0	0	0	0	0 B	8 GB	0 B	0	8	0	1	0	0	0	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
No data available in table										

Showing 0 to 0 of 0 entries

First Previous

Next Last

(b) 删除文件系统上的 output 文件

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -rm -R /user/atguigu/output
```

(c) 执行 mapreduce 程序

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount /user/atguigu/input /user/atguigu/output
```

(d) 查看运行结果

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -cat /user/atguigu/output/*
```

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
1	0	0	1	0	0 B	8 GB	0 B	0	8	0	1	0	0	0	0

Show 20 + entries

Search

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1489820373751_0001	root	word count	MAPREDUCE	default	Sat, 18 Mar 2017 07:15:25 GMT	Sat, 18 Mar 2017 07:15:42 GMT	FINISHED	SUCCEEDED		History

Showing 1 to 1 of 1 entries

FirstPreviousNextLast

4.2.3 配置历史服务器

1) 配置 mapred-site.xml

```
[atguigu@hadoop101 hadoop]$ vi mapred-site.xml
```

```
<property>
  <name>mapreduce.jobhistory.address</name>
```



```
<value>hadoop101:10020</value>
</property>
<property>
  <name>mapreduce.jobhistory.webapp.address</name>
  <value>hadoop101:19888</value>
</property>
```

2) 查看启动历史服务器文件目录:

```
[atguigu@hadoop101 hadoop-2.7.2]$ ls sbin/ | grep mr
mr-jobhistory-daemon.sh
```

3) 启动历史服务器

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/mr-jobhistory-daemon.sh start historyserver
```

4) 查看历史服务器是否启动

```
[atguigu@hadoop101 hadoop-2.7.2]$ jps
```

5) 查看 jobhistory

<http://192.168.1.101:19888/jobhistory>

4.2.4 配置日志的聚集

日志聚集概念: 应用运行完成以后, 将日志信息上传到 HDFS 系统上。

开启日志聚集功能步骤:

(1) 配置 yarn-site.xml

```
[atguigu@hadoop101 hadoop]$ vi 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>
```

(2) 关闭 nodemanager 、 resourcemanager 和 historymanager

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/yarn-daemon.sh stop resourcemanager
```

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/yarn-daemon.sh stop nodemanager
```

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/mr-jobhistory-daemon.sh stop historyserver
```

(3) 启动 nodemanager 、resourcemanager 和 historymanager

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/yarn-daemon.sh start resourcemanager
```

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/yarn-daemon.sh start nodemanager
```

```
[atguigu@hadoop101 hadoop-2.7.2]$ sbin/mr-jobhistory-daemon.sh start historyserver
```

(4) 删除 hdfs 上已经存在的 hdfs 文件

```
[atguigu@hadoop101 hadoop-2.7.2]$ bin/hdfs dfs -rm -R /user/atguigu/output
```

(5) 执行 wordcount 程序

```
[atguigu@hadoop101 hadoop-2.7.2]$ jar -xvf hadoop-mapreduce-examples-2.7.2.jar wordcount
/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar
/user/atguigu/input /user/atguigu/output
```

(6) 查看日志

<http://192.168.1.101:19888/jobhistory>

192.168.10.101:19888/jobhistory

JobHistory

Retired Jobs

Submit Time	Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2017.03.18 17:54:46 CST	2017.03.18 17:54:51 CST	2017.03.18 17:55:01 CST	job_1489830500161_0001	word count	root	default	SUCCEEDED	1	1	1	1
2017.03.18 17:20:31 CST	2017.03.18 17:20:39 CST	2017.03.18 17:20:50 CST	job_1489827711073_0001	word count	root	default	SUCCEEDED	1	1	1	1
2017.03.18 16:21:38 CST	2017.03.18 16:21:42 CST	2017.03.18 16:21:52 CST	job_1489820373751_0003	word count	root	default	SUCCEEDED	1	1	1	1
2017.03.18 15:29:57 CST	2017.03.18 15:30:02 CST	2017.03.18 15:30:12 CST	job_1489820373751_0002	word count	root	default	SUCCEEDED	1	1	1	1
2017.03.18 15:15:25 CST	2017.03.18 15:15:31 CST	2017.03.18 15:15:42 CST	job_1489820373751_0001	word count	root	default	SUCCEEDED	1	1	1	1

MapReduce Job job_1489830500161_0001

Job Overview

Job Name:	word count
User Name:	root
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Mar 18 17:54:46 CST 2017
Started:	Sat Mar 18 17:54:51 CST 2017
Finished:	Sat Mar 18 17:55:01 CST 2017
Elapsed:	9sec
Diagnostics:	
Average Map Time	2sec
Average Shuffle Time	2sec
Average Merge Time	0sec
Average Reduce Time	0sec

Attempt Number	Start Time	Node	Logs
1	Sat Mar 18 17:54:49 CST 2017	hadoop.atguigu.com#042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1



```
Log Type: stderr
Log Length: 222
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 Length: 312
Java HotSpot(TM) Server VM warning: You have loaded library /opt/module/hadoop-2.5.0/lib/native/libhadoop.so.1.0.0 which might have disabled stack guard. The VM will try to fix the stack guard now.
It's highly recommended that you fix the library with 'execstack -c libfile', or link it with '-z noexecstack'.

Log Type: syslog
Log Length: 34561
Showing 4096 bytes of 34561 total. Click here for the full log.
rr.JobHistoryEventHandler: Copying hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/root/.staging/job_1489830500161_0001/job_1489830500161_0001_1.jhist to hdfs://hadoop.atguigu.com:8020/tmp/hadoop
2017-03-18 17:55:02,058 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copied to done location: hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/history/
2017-03-18 17:55:02,060 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copying hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/root/.staging/job_1489830
2017-03-18 17:55:02,082 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copied to done location: hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/history/
2017-03-18 17:55:02,086 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/history/done_i
2017-03-18 17:55:02,088 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/history/done_i
2017-03-18 17:55:02,090 INFO [eventHandlingThread] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/history/done_i
2017-03-18 17:55:02,090 INFO [Thread-64] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Stopped JobHistoryEventHandler. super.stop()
2017-03-18 17:55:02,092 INFO [Thread-64] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Setting job diagnostics to
2017-03-18 17:55:02,093 INFO [Thread-64] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: History url is http://hadoop.atguigu.com:19888/jobhistory/job/job_1489830500161_0001
2017-03-18 17:55:02,106 INFO [Thread-64] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Waiting for application to be successfully unregistered
2017-03-18 17:55:03,112 INFO [Thread-64] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Final Stats: PendingReds:0 ScheduledMaps:0 ScheduledReds:0 AssignedMaps:0 AssignedReds:1 CompletedMap
2017-03-18 17:55:03,113 INFO [Thread-64] org.apache.hadoop.mapreduce.v2.app.MRAppMaster: Deleting staging directory hdfs://hadoop.atguigu.com:8020/tmp/hadoop-yarn/staging/root/.staging/job_148983050016
2017-03-18 17:55:03,118 INFO [Thread-64] org.apache.hadoop.ipc.Server: Stopping server on 56227
2017-03-18 17:55:03,120 INFO [IPC Server listener on 56227] org.apache.hadoop.ipc.Server: Stopping IPC Server listener on 56227
2017-03-18 17:55:03,122 INFO [TaskHeartbeatHandler PingChecker] org.apache.hadoop.mapreduce.v2.app.TaskHeartbeatHandler: TaskHeartbeatHandler thread interrupted
```

4.2.5 配置文件说明

Hadoop 配置文件分两类：默认配置文件和自定义配置文件，只有用户想修改某一默认配置值时，才需要修改自定义配置文件，更改相应属性值。

(1) 默认配置文件：存放在 hadoop 相应的 jar 包中

[core-default.xml]

hadoop-common-2.7.2.jar/ core-default.xml

[hdfs-default.xml]

hadoop-hdfs-2.7.2.jar/ hdfs-default.xml

[yarn-default.xml]

hadoop-yarn-common-2.7.2.jar/ yarn-default.xml

[mapred-default.xml]

hadoop-mapreduce-client-core-2.7.2.jar/ mapred-default.xml

(2) 自定义配置文件：存放在\$HADOOP_HOME/etc/hadoop

core-site.xml

hdfs-site.xml

yarn-site.xml

mapred-site.xml

4.3 完全分布式运行模式

分析：

1) 准备 3 台客户机（关闭防火墙、静态 ip、主机名称）

- 2) 安装 jdk
- 3) 配置环境变量
- 4) 安装 hadoop
- 5) 配置环境变量
- 6) 配置集群
- 7) 单点启动
- 8) 配置 ssh
- 9) 群起并测试集群

4.3.1 虚拟机准备

详见 3.1 章。

4.3.2 编写集群分发脚本 xsync

1) scp:secure copy 安全拷贝

(1) scp 定义:

scp 可以实现服务器与服务器之间的数据拷贝。

(2) 案例实操

(a) 将 hadoop101 中/opt/module 目录下的软件拷贝到 hadoop102 上。

```
[atguigu@hadoop101 /]$ scp -r /opt/module/* atguigu@hadoop102:/opt/module
```

(b) 将 hadoop101 服务器上的/opt/module 目录下的软件拷贝到 hadoop103 上。

```
[atguigu@hadoop103 opt]$ scp -r atguigu@hadoop101:/opt/module/*  
hadoop103:/opt/module
```

(c) 在 hadoop103 上操作将 hadoop101 中/opt/module 目录下的软件拷贝到 hadoop104 上。

```
[atguigu@hadoop103 opt]$ scp -r hadoop101:/opt/module/*  
hadoop104:/opt/module
```

2) rsync

rsync 远程同步工具，主要用于备份和镜像。具有速度快、避免复制相同内容和支持符号链接的优点。

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。

(1) 查看 rsync 使用说明

```
man rsync | more
```

(2) 基本语法

```
rsync -rvl      $pdir/$fname      $user@hadoop$host:$pdir
命令 命令参数 要拷贝的文件路径/名称 目的用户@主机:目的路径
```

(3) 选项说明

选项	功能
-r	递归
-v	显示复制过程
-l	拷贝符号连接

(4) 案例实操

把本机/opt/software 目录同步到 hadoop102 服务器的 root 用户下的/opt/目录

```
[atguigu@hadoop101 opt]$ rsync -rvl /opt/software/* hadoop102:/opt/software/
```

3) 脚本需求分析：循环复制文件到所有节点的相同目录下。

(1) 原始拷贝：

```
rsync -rvl      /opt/module      root@hadoop103:/opt/
```

(2) 期望脚本：

```
xsync 要同步的文件名称
```

(3) 在/home/atguigu/bin 这个目录下存放的脚本，atguigu 用户可以在系统任何地方直接执行。

4) 脚本实现：

(1) 在/home/atguigu 目录下创建 bin 目录，并在 bin 目录下 xsync 创建文件，文件内容如下：

```
[atguigu@hadoop102 ~]$ mkdir bin
[atguigu@hadoop102 ~]$ cd bin/
[atguigu@hadoop102 bin]$ touch xsync
[atguigu@hadoop102 bin]$ vi xsync
```

```
#!/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 获取当前用户名称
user=`whoami`

#5 循环
for((host=103; host<105; host++)); do
    echo ----- hadoop$host -----
    rsync -rvl $pdir/$fname $user@hadoop$host:$pdir
done
```

- (2) 修改脚本 xsync 具有执行权限
- ```
[atguigu@hadoop102 bin]$ chmod 777 xsync
```
- (3) 调用脚本形式: xsync 文件名称
- ```
[atguigu@hadoop102 bin]$ xsync /home/atguigu/bin
```

4.3.3 集群配置

1) 集群部署规划

	hadoop102	hadoop103	hadoop104
HDFS	NameNode		SecondaryNameNode
	DataNode	DataNode	DataNode
YARN		ResourceManager	
	NodeManager	NodeManager	NodeManager

2) 配置集群

- (1) 核心配置文件
- ```
core-site.xml
```
- ```
[atguigu@hadoop102 hadoop]$ vi core-site.xml
```

```
<!-- 指定 HDFS 中 NameNode 的地址 -->
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://hadoop102:9000</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>
```

(2) hdfs 配置文件

hadoop-env.sh

[atguigu@hadoop102 hadoop]\$ vi hadoop-env.sh

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

hdfs-site.xml

[atguigu@hadoop102 hadoop]\$ vi hdfs-site.xml

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

<property>
  <name>dfs.namenode.secondary.http-address</name>
  <value>hadoop104:50090</value>
</property>
```

(3) yarn 配置文件

yarn-env.sh

[atguigu@hadoop102 hadoop]\$ vi yarn-env.sh

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

yarn-site.xml

[atguigu@hadoop102 hadoop]\$ vi 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>hadoop103</value>
</property>
```

(4) mapreduce 配置文件

mapred-env.sh

[atguigu@hadoop102 hadoop]\$ vi mapred-env.sh

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

mapred-site.xml

[atguigu@hadoop102 hadoop]\$ cp mapred-site.xml.template mapred-site.xml

[atguigu@hadoop102 hadoop]\$ vi mapred-site.xml

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

3) 在集群上分发配置好的 Hadoop 配置文件

[atguigu@hadoop102 hadoop]\$ xsync /opt/module/hadoop-2.7.2/

4) 查看文件分发情况

[atguigu@hadoop103 hadoop]\$ cat /opt/module/hadoop-2.7.2/etc/hadoop/core-site.xml

4.3.4 集群单点启动

(0) 如果集群是第一次启动，需要格式化 NameNode

[atguigu@hadoop102 hadoop-2.7.2]\$ hadoop namenode -format

(1) 在 hadoop102 上启动 NameNode

[atguigu@hadoop102 hadoop-2.7.2]\$ hadoop-daemon.sh start namenode

[atguigu@hadoop102 hadoop-2.7.2]\$ jps

3461 NameNode

3531 Jps

(2) 在 hadoop102、hadoop103 以及 hadoop104 上分别启动 DataNode

[atguigu@hadoop102 hadoop-2.7.2]\$ hadoop-daemon.sh start datanode

[atguigu@hadoop102 hadoop-2.7.2]\$ jps

3461 NameNode

3608 Jps

3561 DataNode

[atguigu@hadoop103 hadoop-2.7.2]\$ hadoop-daemon.sh start datanode

[atguigu@hadoop103 hadoop-2.7.2]\$ jps

3190 DataNode

3279 Jps

[atguigu@hadoop104 hadoop-2.7.2]\$ hadoop-daemon.sh start datanode

[atguigu@hadoop104 hadoop-2.7.2]\$ jps

3237 Jps

3163 DataNode

4.3.5 SSH 无密登录配置

1) 配置 ssh

(1) 基本语法

ssh 另一台电脑的 ip 地址

(2) ssh 连接时出现 Host key verification failed 的解决方法

[atguigu@hadoop102 opt] \$ ssh 192.168.1.103

The authenticity of host '192.168.1.103 (192.168.1.103)' can't be established.

RSA key fingerprint is cf:1e:de:d7:d0:4c:2d:98:60:b4:fd:ae:b1:2d:ad:06.

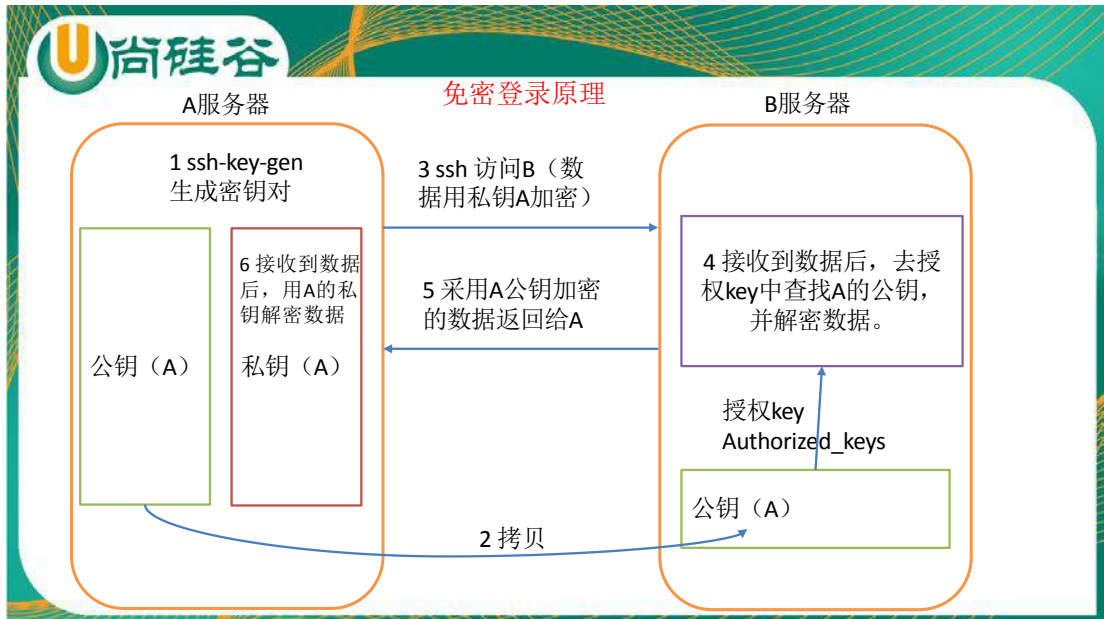
Are you sure you want to continue connecting (yes/no)?

Host key verification failed.

(3) 解决方案如下：直接输入 yes

2) 无密钥配置

(1) 免密登录原理



(2) 生成公钥和私钥:

```
[atguigu@hadoop102 .ssh]$ ssh-keygen -t rsa
```

然后敲 (三个回车), 就会生成两个文件 `id_rsa` (私钥)、`id_rsa.pub` (公钥)

(3) 将公钥拷贝到要免密登录的目标机器上

```
[atguigu@hadoop102 .ssh]$ ssh-copy-id hadoop103
```

```
[atguigu@hadoop102 .ssh]$ ssh-copy-id hadoop104
```

3) `.ssh` 文件夹下 (`~/ssh`) 的文件功能解释

(1) `known_hosts` : 记录 ssh 访问过计算机的公钥(public key)

(2) `id_rsa` : 生成的私钥

(3) `id_rsa.pub` : 生成的公钥

(4) `authorized_keys` : 存放授权过得无密登录服务器公钥

4.3.6 集群测试

1) 配置 slaves

```
/opt/module/hadoop-2.7.2/etc/hadoop/slaves
```

```
[atguigu@hadoop102 hadoop]$ vi slaves
```

```
hadoop102
hadoop103
hadoop104
```

2) 启动集群

(0) 如果集群是第一次启动，需要格式化 NameNode

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hdfs namenode -format
```

(1) 启动 HDFS:

```
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ jps
```

4166 NameNode

4482 Jps

4263 DataNode

```
[atguigu@hadoop103 hadoop-2.7.2]$ jps
```

3218 DataNode

3288 Jps

```
[atguigu@hadoop104 hadoop-2.7.2]$ jps
```

3221 DataNode

3283 SecondaryNameNode

3364 Jps

(2) 启动 yarn

```
[atguigu@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh
```

注意: NameNode 和 ResourceManger 如果不是同一台机器，不能在 NameNode 上启动 yarn，应该在 ResouceManager 所在的机器上启动 yarn。

(3) web 端查看 SecondaryNameNode

(a) 浏览器中输入: <http://hadoop104:50090/status.html>

(b) 查看 SecondaryNameNode 信息。

① hadoop104:50090/status.html

Hadoop Overview

Overview

Version	2.7.2
Compiled	2017-05-22T10:49Z by root from Unknown
NameNode Address	hadoop102:9000
Started	2017/12/11 上午6:01:48
Last Checkpoint	Never
Checkpoint Period	3600 seconds
Checkpoint Transactions	1000000

Checkpoint Image URI

- file:///opt/module/hadoop-2.7.2/data/tmp/dfs/namesecondary

Checkpoint Editlog URI

- file:///opt/module/hadoop-2.7.2/data/tmp/dfs/namesecondary

Hadoop, 2015.

3) 集群基本测试

(1) 上传文件到集群

上传小文件

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -mkdir -p /user/atguigu/input
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -put wcinput/wc.input /user/atguigu/input
```

上传大文件

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hadoop fs -put /opt/software/hadoop-2.7.2.tar.gz /user/atguigu/input
```

(2) 上传文件后查看文件存放在什么位置

查看 HDFS 文件存储路径

```
[atguigu@hadoop102 subdir0]$ pwd
/opt/module/hadoop-2.7.2/data/tmp/dfs/data/current/BP-938951106-192.168.10.107-1495462844069/current/finalized/subdir0/subdir0
```

查看 HDFS 在磁盘存储文件内容

```
[atguigu@hadoop102 subdir0]$ cat blk_1073741825
```

`hadoop yarn`

`hadoop mapreduce`

atguigu

atguigu

(3) 拼接

```
-rw-rw-r--. 1 atguigu atguigu 134217728 5 月 23 16:01 blk_1073741836
-rw-rw-r--. 1 atguigu atguigu 1048583 5 月 23 16:01 blk_1073741836_1012.meta
-rw-rw-r--. 1 atguigu atguigu 63439959 5 月 23 16:01 blk_1073741837
-rw-rw-r--. 1 atguigu atguigu 495635 5 月 23 16:01 blk_1073741837_1013.meta

[atguigu@hadoop102 subdir0]$ cat blk_1073741836>>tmp.file
[atguigu@hadoop102 subdir0]$ cat blk_1073741837>>tmp.file

[atguigu@hadoop102 subdir0]$ tar -zxvf tmp.file
```

(4) 下载

```
[atguigu@hadoop102 ~]$ cd /user/atguigu/input/hadoop-2.7.2
[hadoop102 ~]$ bin/hadoop fs -get /user/atguigu/input/hadoop-2.7.2.tar.gz ./
```

4) 性能测试集群

(1) 写海量数据

(2) 读海量数据

4.3.7 集群启动/停止方式

1) 各个服务组件逐一启动/停止

(1) 分别启动/停止 hdfs 组件

```
hadoop-daemon.sh start|stop namenode|datanode|secondarynamenode
```

(2) 启动/停止 yarn

```
yarn-daemon.sh start|stop resourcemanager|nodemanager
```

2) 各个模块分开启动/停止 (配置 ssh 是前提) 常用

(1) 整体启动/停止 hdfs

```
start-dfs.sh
```

```
stop-dfs.sh
```

(2) 整体启动/停止 yarn

```
start-yarn.sh
```

```
stop-yarn.sh
```


3) 全部启动/停止集群（不建议使用）

```
start-all.sh
```

```
stop-all.sh
```

4.3.8 集群时间同步

时间同步的方式：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。

配置时间同步实操：

1) 时间服务器配置（必须 root 用户）

(1) 检查 ntp 是否安装

```
[root@hadoop102 桌面]# rpm -qa|grep ntp  
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

```
[root@hadoop102 桌面]# vi /etc/ntp.conf
```

修改内容如下

a) 修改 1（设置本地网络上的主机不受限制。）

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

b) 修改 2（设置为不采用公共的服务器）

```
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 为  
#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
```

c) 添加 3（添加默认的一个内部时钟数据，使用它为局域网用户提供服务。）

server 127.127.1.0

fudge 127.127.1.0 stratum 10

(3) 修改/etc/sysconfig/ntpd 文件

```
[root@hadoop102 桌面]# vim /etc/sysconfig/ntpd
```

增加内容如下（让硬件时间与系统时间一起同步）

SYNC_HWCLOCK=yes

(4) 重新启动 ntpd

```
[root@hadoop102 桌面]# service ntpd status
```

ntpd 已停

```
[root@hadoop102 桌面]# service ntpd start
```

正在启动 ntpd:

[确定]

(5) 执行:

```
[root@hadoop102 桌面]# chkconfig ntpd on
```

2) 其他机器配置（必须 root 用户）

(1) 在其他机器配置 10 分钟与时间服务器同步一次

```
[root@hadoop103 hadoop-2.7.2]# crontab -e
```

编写脚本

```
*/10 * * * * /usr/sbin/ntpdate hadoop102
```

(2) 修改任意机器时间

```
[root@hadoop103 hadoop]# date -s "2017-9-11 11:11:11"
```

(3) 十分钟后查看机器是否与时间服务器同步

```
[root@hadoop103 hadoop]# date
```

五 Hadoop 源码编译

5.1 前期准备工作

1) CentOS 联网

配置 CentOS 能连接外网。Linux 虚拟机 ping www.baidu.com 是畅通的

注意：采用 **root** 角色编译，减少文件夹权限出现问题

2) jar 包准备(hadoop 源码、JDK8、maven、ant 、protobuf)

- (1) hadoop-2.7.2-src.tar.gz
- (2) jdk-8u144-linux-x64.tar.gz
- (3) apache-ant-1.9.9-bin.tar.gz
- (4) apache-maven-3.0.5-bin.tar.gz
- (5) protobuf-2.5.0.tar.gz

5.2 jar 包安装

0) 注意：所有操作必须在 **root** 用户下完成

1) JDK 解压、配置环境变量 JAVA_HOME 和 PATH，验证 java-version(如下都需要验证是否配置成功)

```
[root@hadoop101 software] # tar -zxf jdk-8u144-linux-x64.tar.gz -C /opt/module/
```

```
[root@hadoop101 software]# vi /etc/profile
```

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

```
[root@hadoop101 software]#source /etc/profile
```

验证命令：java -version

2) Maven 解压、配置 MAVEN_HOME 和 PATH。

```
[root@hadoop101 software]# tar -zxvf apache-maven-3.0.5-bin.tar.gz -C /opt/module/
```

```
[root@hadoop101 apache-maven-3.0.5]# vi conf/settings.xml
```

```
<mirrors>
  <!-- mirror
    | Specifies a repository mirror site to use instead of a given repository. The
    | repository that
    | this mirror serves has an ID that matches the mirrorOf element of this mirror. IDs
    | are used
    | for inheritance and direct lookup purposes, and must be unique across the set of
```

```

mirrors.
|
<mirror>
  <id>mirrorId</id>
  <mirrorOf>repositoryId</mirrorOf>
  <name>Human Readable Name for this Mirror.</name>
  <url>http://my.repository.com/repo/path</url>
</mirror>
-->
  <mirror>
    <id>nexus-aliyun</id>
    <mirrorOf>central</mirrorOf>
    <name>Nexus aliyun</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
  </mirror>
</mirrors>

```

[root@hadoop101 apache-maven-3.0.5]# vi /etc/profile

```

#MAVEN_HOME
export MAVEN_HOME=/opt/module/apache-maven-3.0.5
export PATH=$PATH:$MAVEN_HOME/bin

```

[root@hadoop101 software]#source /etc/profile

验证命令：mvn -version

3) ant 解压、配置 ANT_HOME 和 PATH。

[root@hadoop101 software]# tar -zxvf apache-ant-1.9.9-bin.tar.gz -C /opt/module/

[root@hadoop101 apache-ant-1.9.9]# vi /etc/profile

```

#ANT_HOME
export ANT_HOME=/opt/module/apache-ant-1.9.9
export PATH=$PATH:$ANT_HOME/bin

```

[root@hadoop101 software]#source /etc/profile

验证命令：ant -version

4) 安装 glibc-headers 和 g++ 命令如下：

[root@hadoop101 apache-ant-1.9.9]# yum install glibc-headers

[root@hadoop101 apache-ant-1.9.9]# yum install gcc-c++

5) 安装 make 和 cmake

```
[root@hadoop101 apache-ant-1.9.9]# yum install make
```

```
[root@hadoop101 apache-ant-1.9.9]# yum install cmake
```

6) 解压 protobuf , 进入到解压后 **protobuf** 主目录, /opt/module/protobuf-2.5.0

然后相继执行命令:

```
[root@hadoop101 software]# tar -zxvf protobuf-2.5.0.tar.gz -C /opt/module/
```

```
[root@hadoop101 opt]# cd /opt/module/protobuf-2.5.0/
```

```
[root@hadoop101 protobuf-2.5.0]# ./configure
```

```
[root@hadoop101 protobuf-2.5.0]# make
```

```
[root@hadoop101 protobuf-2.5.0]# make check
```

```
[root@hadoop101 protobuf-2.5.0]# make install
```

```
[root@hadoop101 protobuf-2.5.0]# ldconfig
```

```
[root@hadoop101 hadoop-dist]# vi /etc/profile
```

```
#LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/opt/module/protobuf-2.5.0
export PATH=$PATH:$LD_LIBRARY_PATH
```

```
[root@hadoop101 software]#source /etc/profile
```

验证命令: **protoc --version**

7) 安装 openssl 库

```
[root@hadoop101 software]#yum install openssl-devel
```

8) 安装 ncurses-devel 库:

```
[root@hadoop101 software]#yum install ncurses-devel
```

到此, 编译工具安装基本完成。

5.3 编译源码

1) 解压源码到/opt/目录

```
[root@hadoop101 software]# tar -zxvf hadoop-2.7.2-src.tar.gz -C /opt/
```

2) 进入到 hadoop 源码主目录

```
[root@hadoop101 hadoop-2.7.2-src]# pwd
```

```
/opt/hadoop-2.7.2-src
```

3) 通过 maven 执行编译命令

```
[root@hadoop101 hadoop-2.7.2-src]#mvn package -Pdist,native -DskipTests -Dtar
```

等待时间 30 分钟左右，最终成功是全部 SUCCESS。

```
[INFO] Apache Hadoop Common ..... SUCCESS [3:35.094s]
[INFO] Apache Hadoop NFS ..... SUCCESS [5.004s]
[INFO] Apache Hadoop KMS ..... SUCCESS [54.027s]
[INFO] Apache Hadoop Common Project ..... SUCCESS [0.022s]
[INFO] Apache Hadoop HDFS ..... SUCCESS [3:58.444s]
[INFO] Apache Hadoop HttpFS ..... SUCCESS [1:02.562s]
[INFO] Apache Hadoop HDFS BookKeeper Journal ..... SUCCESS [33.138s]
[INFO] Apache Hadoop HDFS-NFS ..... SUCCESS [3.993s]
[INFO] Apache Hadoop HDFS Project ..... SUCCESS [0.022s]
[INFO] hadoop-yarn ..... SUCCESS [0.037s]
[INFO] hadoop-yarn-api ..... SUCCESS [1:26.119s]
[INFO] hadoop-yarn-common ..... SUCCESS [1:20.025s]
[INFO] hadoop-yarn-server ..... SUCCESS [0.168s]
[INFO] hadoop-yarn-server-common ..... SUCCESS [9.107s]
[INFO] hadoop-yarn-server-nodemanager ..... SUCCESS [19.867s]
[INFO] hadoop-yarn-server-web-proxy ..... SUCCESS [3.397s]
[INFO] hadoop-yarn-server-applicationhistoryservice ..... SUCCESS [7.432s]
[INFO] hadoop-yarn-server-resourcemanager ..... SUCCESS [17.078s]
[INFO] hadoop-yarn-server-tests ..... SUCCESS [3.998s]
[INFO] hadoop-yarn-client ..... SUCCESS [5.962s]
[INFO] hadoop-yarn-server-sharedcachemanager ..... SUCCESS [2.803s]
[INFO] hadoop-yarn-applications ..... SUCCESS [0.024s]
[INFO] hadoop-yarn-applications-distributedshell ..... SUCCESS [1.841s]
[INFO] hadoop-yarn-applications-unmanaged-am-launcher .... SUCCESS [1.876s]
```

4) 成功的 64 位 hadoop 包在 /opt/hadoop-2.7.2-src/hadoop-dist/target 下。

```
[root@hadoop101 target]# pwd
```

```
/opt/hadoop-2.7.2-src/hadoop-dist/target
```

5.4 常见的问题及解决方案

1) MAVEN install 时候 JVM 内存溢出

处理方式: 在环境配置文件和 maven 的执行文件均可调整 MAVEN_OPTS 的 heap 大小。

(详情查阅 MAVEN 编译 JVM 调优问题, 如:

<http://outofmemory.cn/code-snippet/12652/maven-outofmemoryerror-method>)

2) 编译期间 maven 报错。可能网络阻塞问题导致依赖库下载不完整导致, 多次执行命令 (一次通过比较难) :

```
[root@hadoop101 hadoop-2.7.2-src]#mvn package -Pdist,native -DskipTests -Dtar
```

3) 报 ant、protobuf 等错误, 插件下载不完整或者插件版本问题, 最开始链接有较多特殊情况, 同时推荐

2.7.0 版本的问题汇总帖子 <http://www.tuicool.com/articles/IBn63qf>

六 常见错误及解决方案

- 1) 防火墙没关闭、或者没有启动 yarnx

INFO client.RMProxy: Connecting to ResourceManager at hadoop108/192.168.10.108:8032

- 2) 主机名称配置错误
- 3) ip 地址配置错误
- 4) ssh 没有配置好
- 5) root 用户和 atguigu 两个用户启动集群不统一
- 6) 配置文件修改不细心
- 7) 未编译源码

Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

17/05/22 15:38:58 INFO client.RMProxy: Connecting to ResourceManager at hadoop108/192.168.10.108:8032

- 8) datanode 不被 namenode 识别问题

Namenode 在 format 初始化的时候会形成两个标识, blockPoolId 和 clusterId。新的 datanode 加入时, 会获取这两个标识作为自己工作目录中的标识。

一旦 namenode 重新 format 后, namenode 的身份标识已变, 而 datanode 如果依然持有原来的 id, 就不会被 namenode 识别。

解决办法, 删除 datanode 节点中的数据后, 再次重新格式化 namenode。

- 9) 不识别主机名称

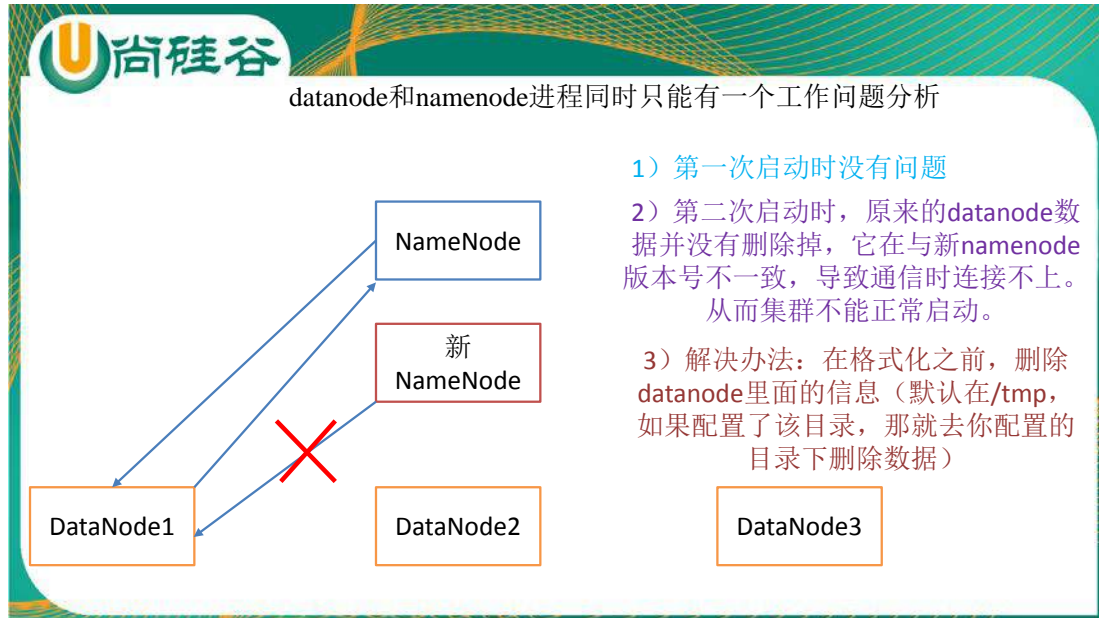
```
java.net.UnknownHostException: hadoop102: hadoop102
    at java.net.InetAddress.getLocalHost(InetAddress.java:1475)
    at
org.apache.hadoop.mapreduce.JobSubmitter.submitJobInternal(JobSubmitter.java:146)
    at org.apache.hadoop.mapreduce.Job$10.run(Job.java:1290)
    at org.apache.hadoop.mapreduce.Job$10.run(Job.java:1287)
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.Subject.doAs(Subject.java:415)
```

解决办法:

(1) 在/etc/hosts 文件中添加 192.168.1.102 hadoop102

(2) 主机名称不要起 hadoop hadoop000 等特殊名称

10) datanode 和 namenode 进程同时只能工作一个。



11) 执行命令不生效，粘贴 word 中命令时，遇到-和长-没区分开。导致命令失效

解决办法：尽量不要粘贴 word 中代码。

12) jps 发现进程已经没有，但是重新启动集群，提示进程已经开启。原因是在 linux 的根目录下/tmp 目录中存在启动的进程临时文件，将集群相关进程删除掉，再重新启动集群。

13) jps 不生效。

原因：全局变量 hadoop java 没有生效，需要 source /etc/profile 文件。

14) 8088 端口连接不上

[atguigu@hadoop102 桌面]\$ cat /etc/hosts

注释掉如下代码

```
#127.0.0.1    localhost localhost.localdomain localhost4 localhost4.localdomain4
```

```
#:::1        hadoop102
```

一 HDFS 概述

1.1 HDFS 产生背景

随着数据量越来越大，在一个操作系统管辖的范围内存不下了，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。

1.2 HDFS 概念

HDFS，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 的设计适合一次写入，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。

1.3 HDFS 优缺点

1.3.1 优点

1) 高容错性

- (1) 数据自动保存多个副本。它通过增加副本的形式，提高容错性；
- (2) 某一个副本丢失以后，它可以自动恢复。

2) 适合大数据处理

- (1) 数据规模：能够处理数据规模达到 GB、TB、甚至 PB 级别的数据；
- (2) 文件规模：能够处理百万规模以上的文件数量，数量相当之大。

3) 流式数据访问，它能保证数据的一致性。

4) 可构建在廉价机器上，通过多副本机制，提高可靠性。

1.3.2 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。

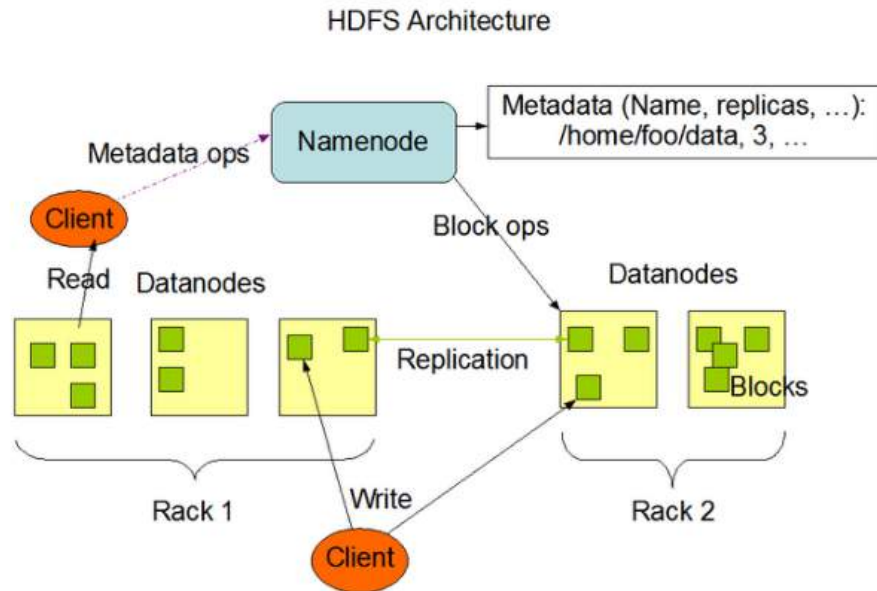
(1) 存储大量小文件的话，它会占用 NameNode 大量的内存来存储文件、目录和块信息。这样是不可取的，因为 NameNode 的内存总是有限的；

(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) 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。

1.4 HDFS 组成架构



HDFS 的架构图

这种架构主要由四个部分组成，分别为 HDFS Client、NameNode、DataNode 和 Secondary NameNode。下面我们分别介绍这四个组成部分。

1) 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，然后进行存储；

(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，比如启动或者关闭 HDFS；

(5) Client 可以通过一些命令来访问 HDFS；

2) NameNode：就是 Master，它是一个主管、管理者。

(1) 管理 HDFS 的名称空间；

(2) 管理数据块（Block）映射信息；

(3) 配置副本策略；

(4) 处理客户端读写请求。

3) 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。

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;

(2) 定期合并 Fsimage 和 Edits, 并推送给 NameNode;

(3) 在紧急情况下, 可辅助恢复 NameNode。

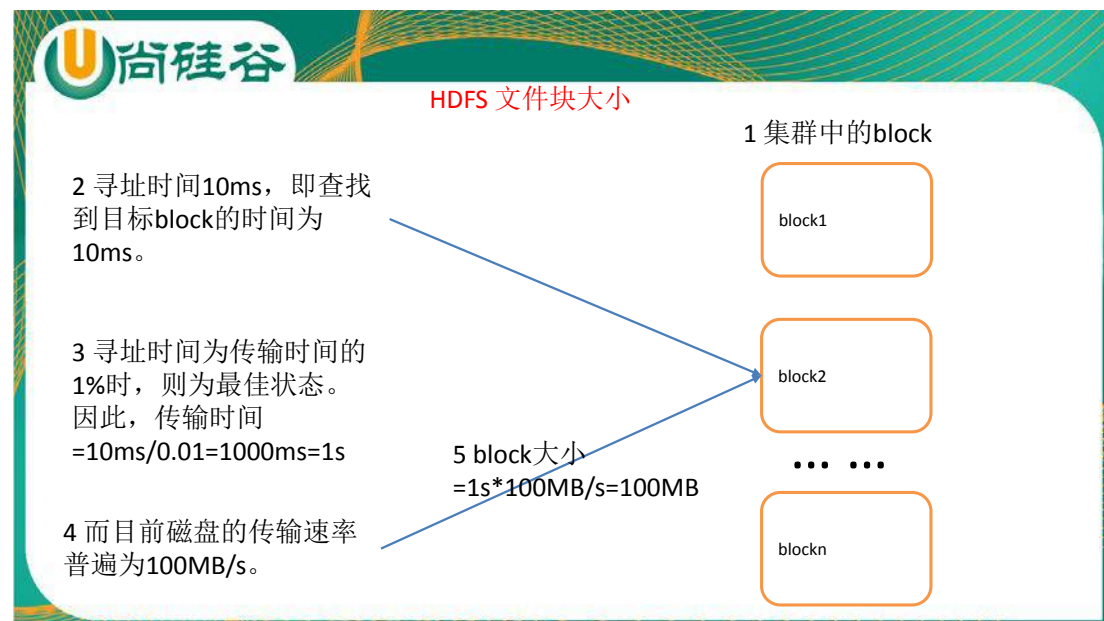
1.5 HDFS 文件块大小

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。

HDFS 的块比磁盘的块大, 其目的是为了最小化寻址开销。如果块设置得足够大, 从磁盘传输数据的时间会明显大于定位这个块开始位置所需的时间。因而, 传输一个由多个块组成的文件的时间取决于磁盘传输速率。

如果寻址时间约为 10ms, 而传输速率为 100MB/s, 为了使寻址时间仅占传输时间的 1%, 我们要将块大小设置约为 100MB。默认的块大小 128MB。

块的大小: $10\text{ms} \times 100 \times 100\text{M/s} = 100\text{M}$



二 HFDS 的 Shell 操作

1) 基本语法

`bin/hadoop fs` 具体命令

2) 命令大全

`[atguigu@hadoop102 hadoop-2.7.2]$ bin/hadoop fs`

```
[-appendToFile <localsrc> ... <dst>]
[-cat [-ignoreCrc] <src> ...]
[-checksum <src> ...]
[-chgrp [-R] GROUP PATH...]
[-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
[-chown [-R] [OWNER][:[GROUP]] PATH...]
[-copyFromLocal [-f] [-p] <localsrc> ... <dst>]
[-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
[-count [-q] <path> ...]
[-cp [-f] [-p] <src> ... <dst>]
[-createSnapshot <snapshotDir> [<snapshotName>]]
[-deleteSnapshot <snapshotDir> <snapshotName>]
[-df [-h] [<path> ...]]
[-du [-s] [-h] <path> ...]
[-expunge]
[-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
[-getfacl [-R] <path>]
[-getmerge [-nl] <src> <localdst>]
[-help [cmd ...]]
[-ls [-d] [-h] [-R] [<path> ...]]
[-mkdir [-p] <path> ...]
[-moveFromLocal <localsrc> ... <dst>]
[-moveToLocal <src> <localdst>]
[-mv <src> ... <dst>]
[-put [-f] [-p] <localsrc> ... <dst>]
[-renameSnapshot <snapshotDir> <oldName> <newName>]
[-rm [-f] [-r|-R] [-skipTrash] <src> ...]
[-rmdir [--ignore-fail-on-non-empty] <dir> ...]
[-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>][--set <acl_spec> <path>]]
[-setrep [-R] [-w] <rep> <path> ...]
[-stat [format] <path> ...]
[-tail [-f] <file>]
[-test [-defsz] <path>]
[-text [-ignoreCrc] <src> ...]
[-touchz <path> ...]
[-usage [cmd ...]]
```


3) 常用命令实操

(0) 启动 Hadoop 集群（方便后续的测试）

```
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh
```

```
[atguigu@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh
```

(1) -help: 输出这个命令参数

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -help rm
```

(2) -ls: 显示目录信息

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -ls /
```

(3) -mkdir: 在 hdfs 上创建目录

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -mkdir -p /sanguo/shuguo
```

(4) -moveFromLocal 从本地剪切粘贴到 hdfs

```
[atguigu@hadoop102 hadoop-2.7.2]$ touch kongming.txt
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -moveFromLocal ./kongming.txt  
/sanguo/shuguo
```

(5) --appendToFile : 追加一个文件到已经存在的文件末尾

```
[atguigu@hadoop102 hadoop-2.7.2]$ touch liubei.txt
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ vi liubei.txt
```

输入

san gu mao lu

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -appendToFile liubei.txt  
/sanguo/shuguo/kongming.txt
```

(6) -cat : 显示文件内容

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -cat /sanguo/shuguo/kongming.txt
```

(7) -tail: 显示一个文件的末尾

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -tail /sanguo/shuguo/kongming.txt
```

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

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -chmod 666  
/sanguo/shuguo/kongming.txt
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -chown atguigu:atguigu  
/sanguo/shuguo/kongming.txt
```

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

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -copyFromLocal README.txt /
```

(10) -copyToLocal: 从 hdfs 拷贝到本地

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -copyToLocal  
/sanguo/shuguo/kongming.txt ./
```

(11) -cp : 从 hdfs 的一个路径拷贝到 hdfs 的另一个路径

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -cp /sanguo/shuguo/kongming.txt  
/zhuge.txt
```

(12) -mv: 在 hdfs 目录中移动文件

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -mv /zhuge.txt /sanguo/shuguo/
```

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

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -get /sanguo/shuguo/kongming.txt ./
```

(14) -getmerge : 合并下载多个文件, 比如 hdfs 的目录 /aaa/下有多个文件:log.1,
log.2,log.3,...

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -getmerge  
/user/atguigu/test/* ./zaiyiqi.txt
```

(15) -put: 等同于 copyFromLocal

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -put ./zaiyiqi.txt /user/atguigu/test/
```

(16) -rm: 删除文件或文件夹

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -rm /user/atguigu/test/jinlian2.txt
```

(17) -rmdir: 删除空目录

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -mkdir /test
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -rmdir /test
```

(18) -du 统计文件夹的大小信息

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -du -s -h /user/atguigu/test
```

```
2.7 K /user/atguigu/test
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -du -h /user/atguigu/test
```

```
1.3 K /user/atguigu/test/README.txt
```

```
15 /user/atguigu/test/jinlian.txt
```

```
1.4 K /user/atguigu/test/zaiyiqi.txt
```

(19) -setrep: 设置 hdfs 中文件的副本数量

```
[atguigu@hadoop102 ~]$ hadoop fs -setrep 10 /sanguo/shuguo/kongming.txt
```

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	atguigu	supergroup	0 B	2018/1/23 下午4:11:10	10	128 MB	kongming.txt

这里设置的副本数只是记录在 NameNode 的元数据中，是否真的会有这么多副本，还得看 DataNode 的数量。因为目前只有 3 台设备，最多也就 3 个副本，只有节点数的增加到 10 台时，副本数才能达到 10。

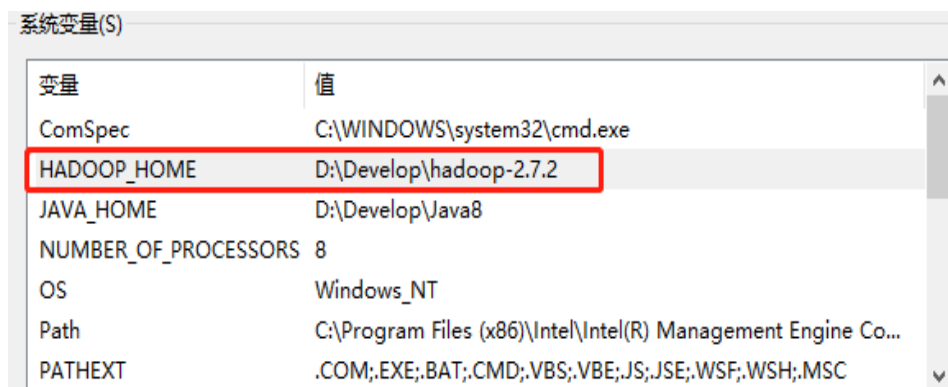
三 HDFS 客户端操作

3.1 HDFS 客户端环境准备

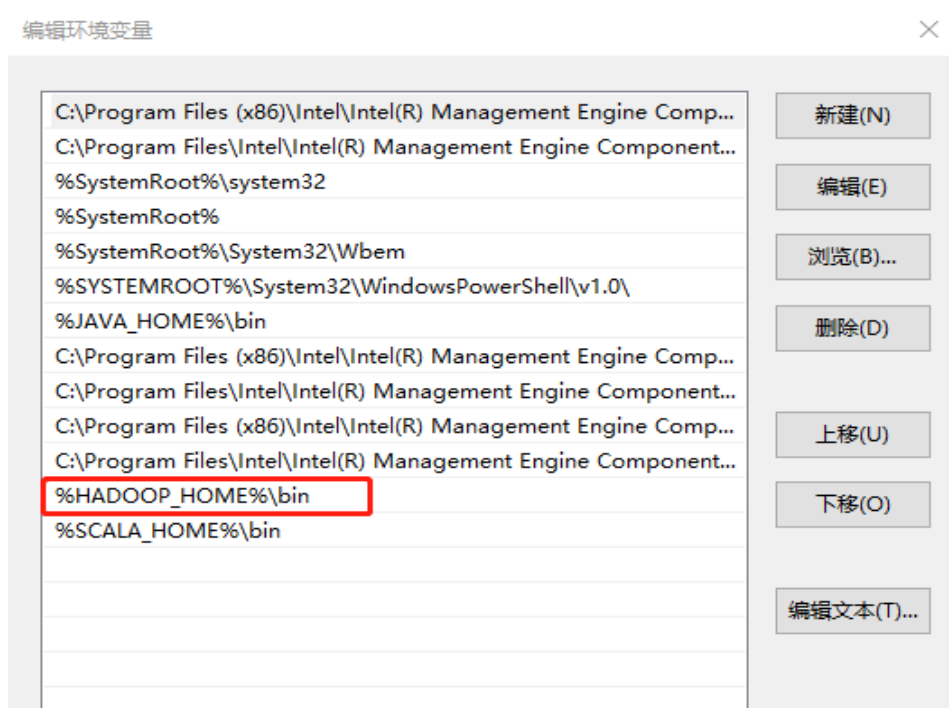
- 1) 根据自己电脑的操作系统拷贝对应的编译后的 hadoop jar 包到非中文路径（例如：D:\Develop\hadoop-2.7.2）。



- 2) 配置 HADOOP_HOME 环境变量



- 2) 配置 Path 环境变量



4) 创建一个 Maven 工程 HdfsClientDemo

5) 导入相应的依赖

```
<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>
```

6) 创建包名: com.atguigu.hdfs

7) 创建 HdfsClient 类

```
public class HdfsClient{
    @Test
    public void testMkdirs() throws IOException, InterruptedException,
        URISyntaxException{

        // 1 获取文件系统
        Configuration configuration = new Configuration();
        // 配置在集群上运行
        // configuration.set("fs.defaultFS", "hdfs://hadoop102:9000");
        // FileSystem fs = FileSystem.get(configuration);

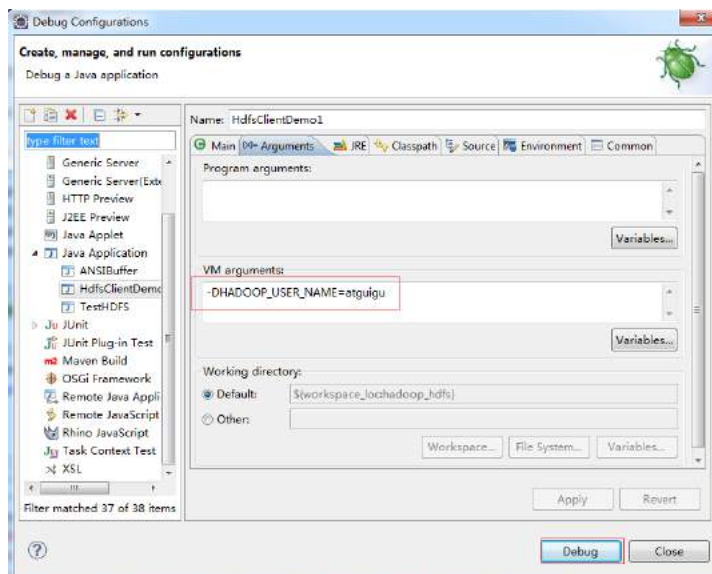
        Configuration configuration = new Configuration();
        FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,
            "atguigu");

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

        // 3 关闭资源
        fs.close();
    }
}
```

8) 执行程序

运行时需要配置用户名称



客户端去操作 hdfs 时，是有一个用户身份的。默认情况下，hdfs 客户端 api 会从 jvm 中获取一个参数来作为自己的用户身份：-DHADOOP_USER_NAME=atguigu，atguigu 为用户名称。

8) 注意：如果 eclipse 打印不出日志，在控制台上只显示

```
1.log4j:WARN No appenders could be found for logger (org.apache.hadoop.util.Shell).
2.log4j:WARN Please initialize the log4j system properly.
3.log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
```

需要在项目的 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
```

3.2 HDFS 的 API 操作

3.2.1 HDFS 文件上传（测试参数优先级）

1) 编写源代码

```
@Test
public void testCopyFromLocalFile() throws IOException, InterruptedException,
URISyntaxException {
    // 1 获取文件系统
    Configuration configuration = new Configuration();
```

```

        configuration.set("dfs.replication", "2");
        FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,
"atguigu");

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

        // 3 关闭资源
        fs.close();

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

```

2) 将 hdfs-site.xml 拷贝到项目的根目录下

```

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

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

```

3) 参数优先级

参数优先级排序：（1）客户端代码中设置的值 > （2）classpath 下的用户自定义配置文件 > （3）然后是服务器的默认配置

3.2.2 HDFS 文件下载

```

@Test
public void testCopyToLocalFile() throws IOException, InterruptedException,
URISyntaxException{
    // 1 获取文件系统
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,
"atguigu");

    // 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("/hello1.txt"), new Path("e:/hello1.txt"), true);
}

```

```
// 3 关闭资源
fs.close();

}
```

3.2.3 HDFS 文件夹删除

```
@Test
public void testDelete() throws IOException, InterruptedException,
URISyntaxException{
    // 1 获取文件系统
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,
"atguigu");

    // 2 执行删除
    fs.delete(new Path("/1108/"), true);

    // 3 关闭资源
    fs.close();
}
```

3.2.4 HDFS 文件名更改

```
@Test
public void testRename() throws IOException, InterruptedException,
URISyntaxException{
    // 1 获取文件系统
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,
"atguigu");

    // 2 修改文件名称
    fs.rename(new Path("/hello.txt"), new Path("/hello6.txt"));

    // 3 关闭资源
    fs.close();
}
```

3.2.5 HDFS 文件详情查看

查看文件名称、权限、长度、块信息

```
@Test
public void testListFiles() throws IOException, InterruptedException,
URISyntaxException{
    // 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.getLen());
            // 权限
            System.out.println(status.getPermission());
            // z 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-----");
        }
    }
}

```

3.2.6 HDFS 文件和文件夹判断

```

@Test
public void testListStatus() throws IOException, InterruptedException,
URISyntaxException{

    // 1 获取文件配置信息
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,

```

```

"atguigu");

    // 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();
}

```

3.3 HDFS 的 I/O 流操作

3.3.1 HDFS 文件上传

```

@Test
public void putFileToHDFS() throws IOException, InterruptedException,
URISyntaxException {
    // 1 获取文件系统
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:9000"), configuration,
"atguigu");

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

    // 3 获取输出流
    FSDataOutputStream fos = fs.create(new Path("/hello4.txt"));

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

    // 5 关闭资源
    IOUtils.closeStream(fis);
    IOUtils.closeStream(fos);
}

```

3.3.2 HDFS 文件下载

1) 需求：从 HDFS 上下载文件到本地 e 盘上。

2) 编写代码：

```
// 文件下载
@Test
public void getFileFromHDFS() throws IOException, InterruptedException,
URISyntaxException{
    // 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("/hello1.txt"));

    // 3 获取输出流
    FileOutputStream fos = new FileOutputStream(new File("e:/hello1.txt"));

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

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

3.3.3 定位文件读取

1) 下载第一块

```
@Test
public void readFileSeek1() throws IOException, InterruptedException,
URISyntaxException{
    // 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
    IOUtils.closeStream(fis);
    IOUtils.closeStream(fos);
}

```

2) 下载第二块

```

@Test
public void readFileSeek2() throws IOException, InterruptedException,
URISyntaxException{
    // 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
    IOUtils.copyBytes(fis, fos, configuration);

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

```

3) 合并文件

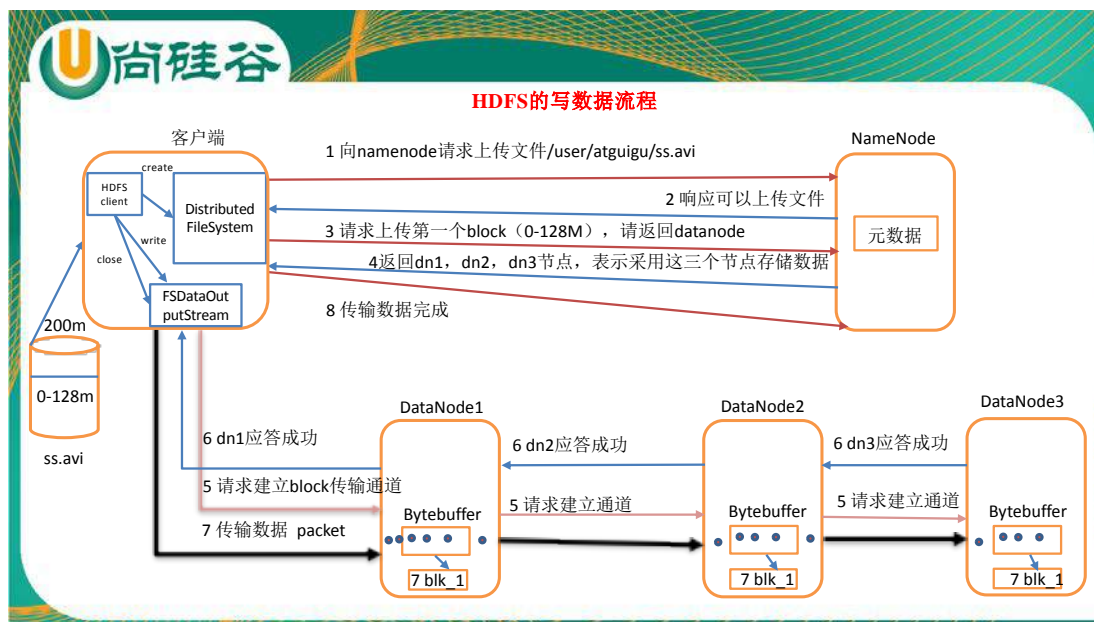
在 window 命令窗口中执行


```
type hadoop-2.7.2.tar.gz.part2 >> hadoop-2.7.2.tar.gz.part1
```

四 HDFS 的数据流

4.1 HDFS 写数据流程

4.1.1 剖析文件写入



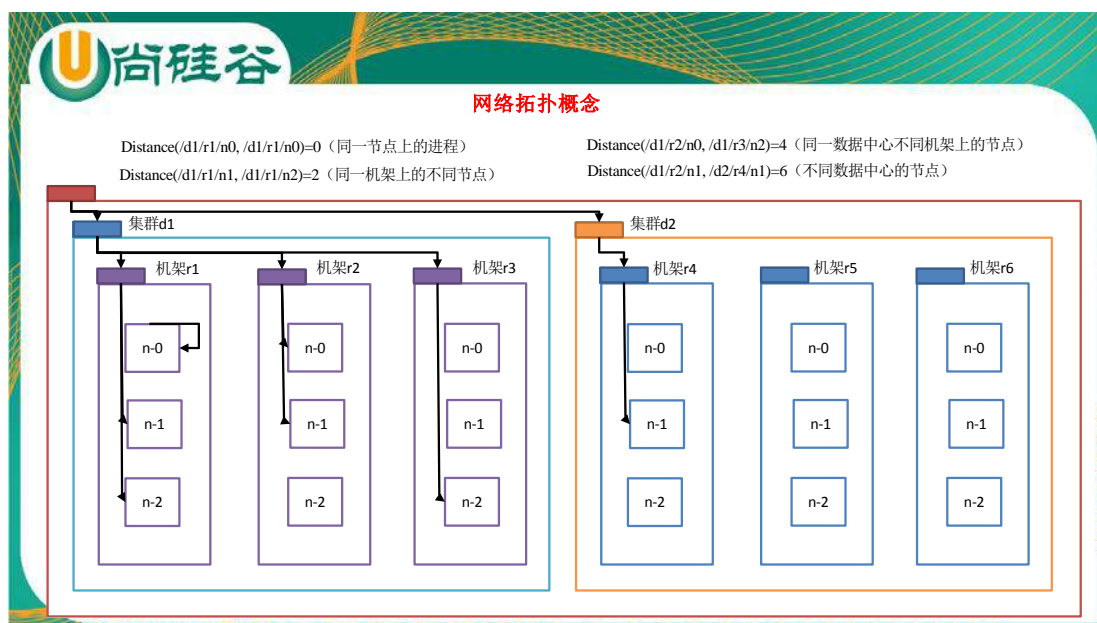
- 1) 客户端通过 Distributed FileSystem 模块向 NameNode 请求上传文件，NameNode 检查目标文件是否已存在，父目录是否存在。
- 2) NameNode 返回是否可以上传。
- 3) 客户端请求第一个 block 上传到哪几个 datanode 服务器上。
- 4) NameNode 返回 3 个 datanode 节点，分别为 dn1、dn2、dn3。
- 5) 客户端通过 FSDataOutputStream 模块请求 dn1 上传数据，dn1 收到请求会继续调用 dn2，然后 dn2 调用 dn3，将这个通信管道建立完成。
- 6) dn1、dn2、dn3 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。
- 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)。

4.1.2 网络拓扑概念

在本地网络中，两个节点被称为“彼此近邻”是什么意思？在海量数据处理中，其主要限

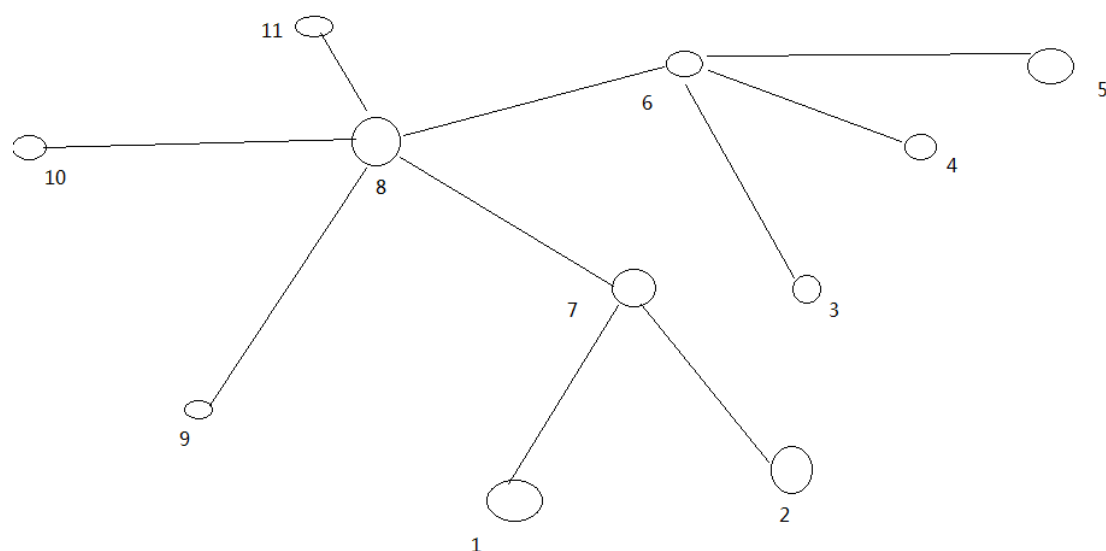
制因素是节点之间数据的传输速率——带宽很稀缺。这里的想法是将两个节点间的带宽作为距离的衡量标准。

节点距离：两个节点到达最近共同祖先的距离总和。



例如，假设有数据中心 d1 机架 r1 中的节点 n1。该节点可以表示为 $d1/r1/n1$ 。利用这种标记，这里给出四种距离描述。

大家算一算每两个节点之间的距离。



4.1.3 机架感知（副本节点选择）

1) 官方 ip 地址:

<http://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/RackAwareness.html>

http://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Data_Replication

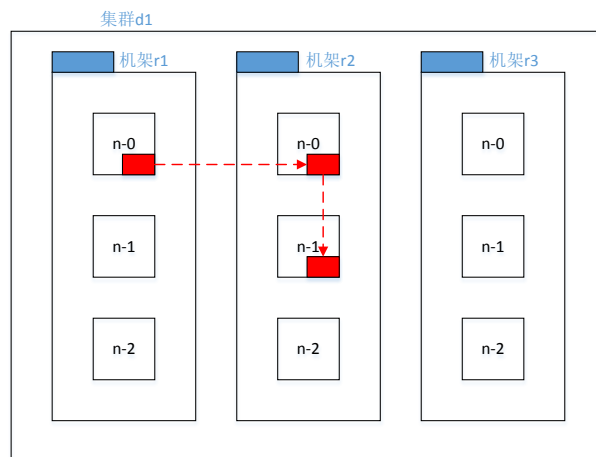
a_Replication

2) 低版本 Hadoop 副本节点选择

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不相同机架的随机节点上。

第三个副本和第二个副本位于相同机架，节点随机。

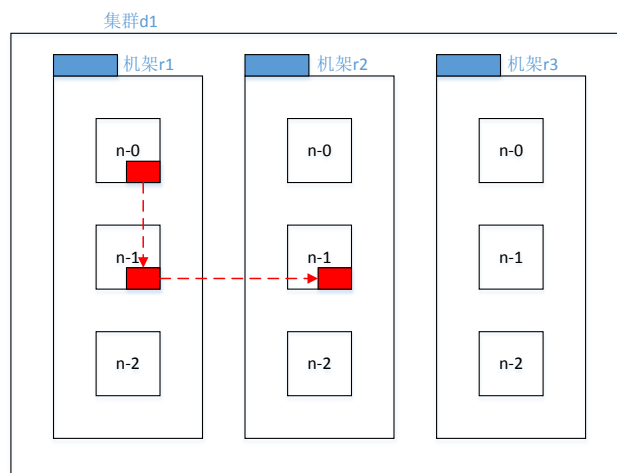


3) Hadoop2.7.2 副本节点选择

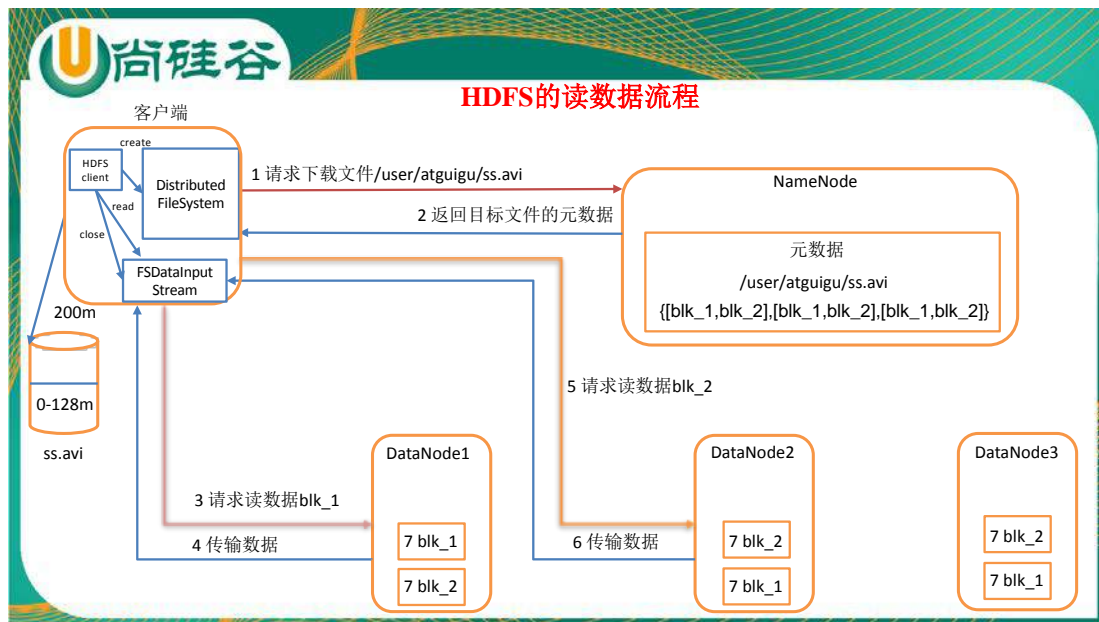
第一个副本在 **Client** 所处的节点上。如果客户端在集群外，随机选一个。

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。



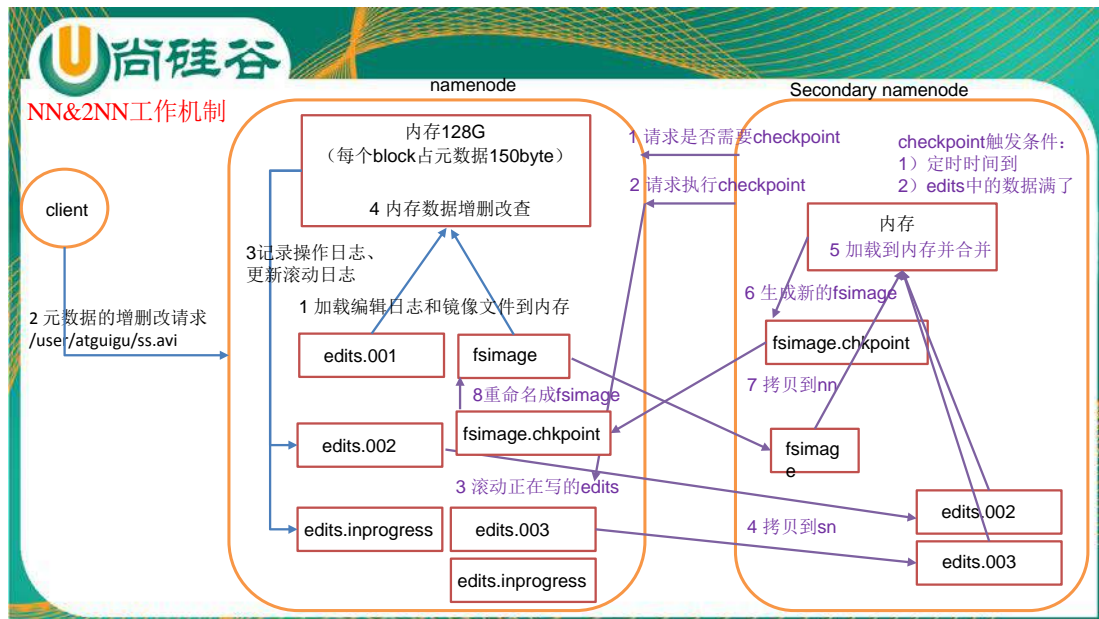
4.2 HDFS 读数据流程



- 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。

五 NameNode 和 SecondaryNameNode

5.1 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查。

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。
- (7) 拷贝 fsimage.chkpoint 到 NameNode。
- (8) NameNode 将 fsimage.chkpoint 重新命名成 fsimage。

5.2 Fimage 和 Edits 解析

1) 概念

namenode 被格式化之后, 将在/opt/module/hadoop-2.7.2/data/tmp/dfs/name/current 目录中产生如下文件

```
edits_00000000000000000000
fsimage_00000000000000000000.md5
seen_txid
VERSION
```

(1) Fimage 文件: HDFS 文件系统元数据的一个永久性的检查点, 其中包含 HDFS 文件系统的所有目录和文件 idnode 的序列化信息。

(2) Edits 文件: 存放 HDFS 文件系统的所有更新操作的路径, 文件系统客户端执行的所有写操作首先会被记录到 edits 文件中。

(3) seen_txid 文件保存的是一个数字, 就是最后一个 edits_ 的数字

(4) 每次 NameNode 启动的时候都会将 fsimage 文件读入内存, 并从 00001 开始到 seen_txid 中记录的数字依次执行每个 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。

2) oiv 查看 fsimage 文件

(1) 查看 oiv 和 oev 命令

```
[atguigu@hadoop102 current]$ hdfs
```

```
oiv          apply the offline fsimage viewer to an fsimage
```

```
oev          apply the offline edits viewer to an edits file
```

(2) 基本语法

```
hdfs oiv -p 文件类型 -i 镜像文件 -o 转换后文件输出路径
```

(3) 案例实操

```
[atguigu@hadoop102 current]$ pwd
```

```
/opt/module/hadoop-2.7.2/data/tmp/dfs/name/current
```

```
[atguigu@hadoop102 current]$ hdfs oiv -p XML -i fsimage_00000000000000000025 -o
```

```
/opt/module/hadoop-2.7.2/fsimage.xml
```

```
[atguigu@hadoop102 current]$ cat /opt/module/hadoop-2.7.2/fsimage.xml
```

将显示的 xml 文件内容拷贝到 eclipse 中创建的 xml 文件中, 并格式化。部分显示

结果如下。

```
<inode>
  <id>16386</id>
  <type>DIRECTORY</type>
  <name>user</name>
  <mtime>1512722284477</mtime>
  <permission>atguigu:supergroup:rw-r-x</permission>
  <nsquota>-1</nsquota>
  <dsquota>-1</dsquota>
</inode>
<inode>
  <id>16387</id>
  <type>DIRECTORY</type>
  <name>atguigu</name>
  <mtime>1512790549080</mtime>
  <permission>atguigu:supergroup:rw-r-x</permission>
  <nsquota>-1</nsquota>
  <dsquota>-1</dsquota>
</inode>
<inode>
  <id>16389</id>
  <type>FILE</type>
  <name>wc.input</name>
  <replication>3</replication>
  <mtime>1512722322219</mtime>
  <atime>1512722321610</atime>
  <preferredBlockSize>134217728</preferredBlockSize>
  <permission>atguigu:supergroup:rw-r--</permission>
  <blocks>
    <block>
      <id>1073741825</id>
      <genstamp>1001</genstamp>
      <numBytes>59</numBytes>
    </block>
  </blocks>
</inode>
```

3) oev 查看 edits 文件

(1) 基本语法

hdfs oev -p 文件类型 -i 编辑日志 -o 转换后文件输出路径

(2) 案例实操

```
[atguigu@hadoop102 current]$ hdfs oev -p XML -i
```

```
edits_000000000000000012-000000000000000013 -o /opt/module/hadoop-2.7.2/edits.xml
```

```
[atguigu@hadoop102 current]$ cat /opt/module/hadoop-2.7.2/edits.xml
```

将显示的 xml 文件内容拷贝到 eclipse 中创建的 xml 文件中，并格式化。显示结果如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<EDITS>
  <EDITS_VERSION>-63</EDITS_VERSION>
  <RECORD>
    <OPCODE>OP_START_LOG_SEGMENT</OPCODE>
    <DATA>
      <TXID>129</TXID>
    </DATA>
  </RECORD>
  <RECORD>
    <OPCODE>OP_ADD</OPCODE>
    <DATA>
      <TXID>130</TXID>
      <LENGTH>0</LENGTH>
      <INODEID>16407</INODEID>
      <PATH>/hello7.txt</PATH>
      <REPLICATION>2</REPLICATION>
      <MTIME>1512943607866</MTIME>
      <ATIME>1512943607866</ATIME>
      <BLOCKSIZE>134217728</BLOCKSIZE>

      <CLIENT_NAME>DFSClient_NONMAPREDUCE_-1544295051_1</CLIENT_
NAME>
      <CLIENT_MACHINE>192.168.1.5</CLIENT_MACHINE>
      <OVERWRITE>true</OVERWRITE>
      <PERMISSION_STATUS>
        <USERNAME>atguigu</USERNAME>
        <GROUPNAME>supergroup</GROUPNAME>
        <MODE>420</MODE>
      </PERMISSION_STATUS>

      <RPC_CLIENTID>908eafd4-9aec-4288-96f1-e8011d181561</RPC_CLIENTID>
      <RPC_CALLID>0</RPC_CALLID>
    </DATA>
  </RECORD>
  <RECORD>
    <OPCODE>OP_ALLOCATE_BLOCK_ID</OPCODE>
    <DATA>
```

```
<TXID>131</TXID>
<BLOCK_ID>1073741839</BLOCK_ID>
</DATA>
</RECORD>
<RECORD>
  <OPCODE>OP_SET_GENSTAMP_V2</OPCODE>
  <DATA>
    <TXID>132</TXID>
    <GENSTAMPV2>1016</GENSTAMPV2>
  </DATA>
</RECORD>
<RECORD>
  <OPCODE>OP_ADD_BLOCK</OPCODE>
  <DATA>
    <TXID>133</TXID>
    <PATH>/hello7.txt</PATH>
    <BLOCK>
      <BLOCK_ID>1073741839</BLOCK_ID>
      <NUM_BYTES>0</NUM_BYTES>
      <GENSTAMP>1016</GENSTAMP>
    </BLOCK>
    <RPC_CLIENTID></RPC_CLIENTID>
    <RPC_CALLID>-2</RPC_CALLID>
  </DATA>
</RECORD>
<RECORD>
  <OPCODE>OP_CLOSE</OPCODE>
  <DATA>
    <TXID>134</TXID>
    <LENGTH>0</LENGTH>
    <INODEID>0</INODEID>
    <PATH>/hello7.txt</PATH>
    <REPLICATION>2</REPLICATION>
    <MTIME>1512943608761</MTIME>
    <ATIME>1512943607866</ATIME>
    <BLOCKSIZE>134217728</BLOCKSIZE>
    <CLIENT_NAME></CLIENT_NAME>
    <CLIENT_MACHINE></CLIENT_MACHINE>
    <OVERWRITE>>false</OVERWRITE>
    <BLOCK>
      <BLOCK_ID>1073741839</BLOCK_ID>
      <NUM_BYTES>25</NUM_BYTES>
      <GENSTAMP>1016</GENSTAMP>
    </BLOCK>
```

```

        <PERMISSION_STATUS>
            <USERNAME>atguigu</USERNAME>
            <GROUPNAME>supergroup</GROUPNAME>
            <MODE>420</MODE>
        </PERMISSION_STATUS>
    </DATA>
</RECORD>
</EDITS>

```

5.3 checkpoint 时间设置

(1) 通常情况下，SecondaryNameNode 每隔一小时执行一次。

[hdfs-default.xml]

```

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

```

(2) 一分钟检查一次操作次数，当操作次数达到 1 百万时，SecondaryNameNode 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>

```

5.4 NameNode 故障处理

NameNode 故障后，可以采用如下两种方法恢复数据。

方法一：将 SecondaryNameNode 中数据拷贝到 NameNode 存储数据的目录；

1) kill -9 namenode 进程

2) 删除 NameNode 存储的数据 (/opt/module/hadoop-2.7.2/data/tmp/dfs/name)

```

[atguigu@hadoop102 ~]$ cd /opt/module/hadoop-2.7.2/
[atguigu@hadoop102 ~]$ rm -rf /opt/module/hadoop-2.7.2/data/tmp/dfs/name/*

```

3) 拷贝 SecondaryNameNode 中数据到原 NameNode 存储数据目录

```
[atguigu@hadoop102 dfs]$ scp -r  
atguigu@hadoop104:/opt/module/hadoop-2.7.2/data/tmp/dfs/namespace/* ./name/
```

4) 重新启动 namenode

```
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/hadoop-daemon.sh start namenode
```

方法二：使用 **-importCheckpoint** 选项启动 **NameNode** 守护进程，从而将 **SecondaryNameNode** 中数据拷贝到 **NameNode** 目录中。

1) 修改 `hdfs-site.xml` 中的

```
<property>  
  <name>dfs.namenode.checkpoint.period</name>  
  <value>120</value>  
</property>  
  
<property>  
  <name>dfs.namenode.name.dir</name>  
  <value>/opt/module/hadoop-2.7.2/data/tmp/dfs/name</value>  
</property>
```

2) `kill -9 namenode` 进程

3) 删除 **NameNode** 存储的数据 (`/opt/module/hadoop-2.7.2/data/tmp/dfs/name`)

```
[atguigu@hadoop102 hadoop-2.7.2]$ rm -rf  
/opt/module/hadoop-2.7.2/data/tmp/dfs/name/*
```

4) 如果 **SecondaryNameNode** 不和 **NameNode** 在一个主机节点上，需要将 **SecondaryNameNode** 存储数据的目录拷贝到 **NameNode** 存储数据的同级目录，并删除 `in_use.lock` 文件。

```
[atguigu@hadoop102 dfs]$ scp -r  
atguigu@hadoop104:/opt/module/hadoop-2.7.2/data/tmp/dfs/namespace ./  
  
[atguigu@hadoop102 namespace]$ rm -rf in_use.lock  
  
[atguigu@hadoop102 dfs]$ pwd  
/opt/module/hadoop-2.7.2/data/tmp/dfs  
  
[atguigu@hadoop102 dfs]$ ls  
data  name  namespace
```

5) 导入检查点数据（等待一会 `ctrl+c` 结束掉）

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hdfs namenode -importCheckpoint
```

6) 启动 namenode

```
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/hadoop-daemon.sh start namenode
```

5.5 集群安全模式

1) 概述

NameNode 启动时，首先将映像文件（fsimage）载入内存，并执行编辑日志（edits）中的各项操作。一旦在内存中成功建立文件系统元数据的映像，则创建一个新的 fsimage 文件和一个空的编辑日志。此时，NameNode 开始监听 DataNode 请求。但是此刻，NameNode 运行在安全模式，即 NameNode 的文件系统对于客户端来说是只读的。

系统中的数据块的位置并不是由 NameNode 维护的，而是以块列表的形式存储在 DataNode 中。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）。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。

2) 基本语法

集群处于安全模式，不能执行重要操作（写操作）。集群启动完成后，自动退出安全模式。

- | | |
|--|-----------------|
| (1) <code>bin/hdfs dfsadmin -safemode get</code> | (功能描述：查看安全模式状态) |
| (2) <code>bin/hdfs dfsadmin -safemode enter</code> | (功能描述：进入安全模式状态) |
| (3) <code>bin/hdfs dfsadmin -safemode leave</code> | (功能描述：离开安全模式状态) |
| (4) <code>bin/hdfs dfsadmin -safemode wait</code> | (功能描述：等待安全模式状态) |

3) 案例

模拟等待安全模式

- (1) 先进入安全模式

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hdfs dfsadmin -safemode enter
```

- (2) 执行下面的脚本

编辑一个脚本

```
#!/bin/bash
```

```
bin/hdfs dfsadmin -safemode wait  
bin/hdfs dfs -put ~/hello.txt /root/hello.txt
```

(3) 再打开一个窗口，执行

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hdfs dfsadmin -safemode leave
```

5.6 NameNode 多目录配置

1) NameNode 的本地目录可以配置成多个，且每个目录存放内容相同，增加了可靠性。

2) 具体配置如下：

(1) 在 hdfs-site.xml 文件中增加如下内容

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

(2) 停止集群，删除 data 和 logs 中所有数据。

```
[atguigu@hadoop102 hadoop-2.7.2]$ rm -rf data/ logs/
```

```
[atguigu@hadoop103 hadoop-2.7.2]$ rm -rf data/ logs/
```

```
[atguigu@hadoop104 hadoop-2.7.2]$ rm -rf data/ logs/
```

(3) 格式化集群并启动。

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hdfs namenode -format
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh
```

(4) 查看结果

```
[atguigu@hadoop102 dfs]$ ll
```

总用量 12

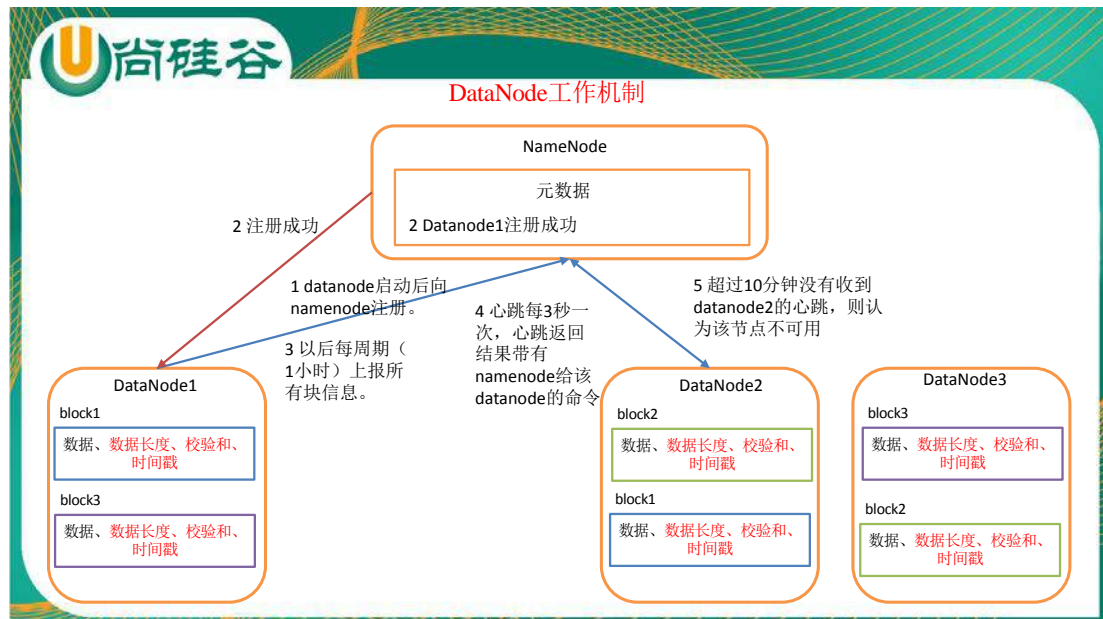
```
drwx-----. 3 atguigu atguigu 4096 12 月 11 08:03 data
```

```
drwxrwxr-x. 3 atguigu atguigu 4096 12 月 11 08:03 name1
```

```
drwxrwxr-x. 3 atguigu atguigu 4096 12 月 11 08:03 name2
```


六 DataNode

6.1 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。

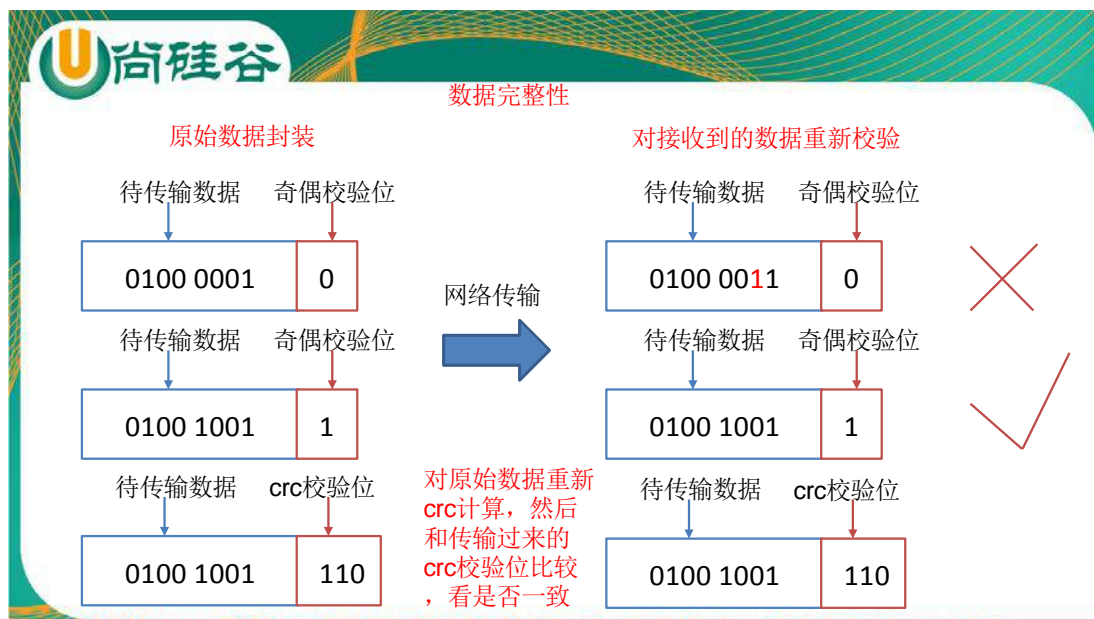
6.2 数据完整性

1) 当 DataNode 读取 block 的时候，它会计算 checksum。

2) 如果计算后的 checksum，与 block 创建时值不一样，说明 block 已经损坏。

3) client 读取其他 DataNode 上的 block。

4) datanode 在其文件创建后周期验证 checksum。



6.3 掉线时限参数设置

DataNode 进程死亡或者网络故障造成 DataNode 无法与 NameNode 通信，NameNode 不会立即把该节点判定为死亡，要经过一段时间，这段时间暂称作超时时长。HDFS 默认的超时时长为 10 分钟+30 秒。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。

需要注意的是 hdfs-site.xml 配置文件中的 heartbeat.recheck.interval 的单位为**毫秒**，dfs.heartbeat.interval 的单位为**秒**。

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

6.4 服役新数据节点

0) 需求：

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。

1) 环境准备

- (1) 克隆一台虚拟机
- (2) 修改 ip 地址和主机名称
- (3) 修改 xsync 文件，增加新增节点的 ssh 无密登录配置
- (4) 删除原来 HDFS 文件系统留存的文件

/opt/module/hadoop-2.7.2/data

2) 服役新节点具体步骤

- (1) 在 namenode 的 /opt/module/hadoop-2.7.2/etc/hadoop 目录下创建 dfs.hosts 文件

```
[atguigu@hadoop105 hadoop]$ pwd
```

```
/opt/module/hadoop-2.7.2/etc/hadoop
```

```
[atguigu@hadoop105 hadoop]$ touch dfs.hosts
```

```
[atguigu@hadoop105 hadoop]$ vi dfs.hosts
```

添加如下主机名称（包含新服役的节点）

hadoop102

hadoop103

hadoop104

hadoop105

- (2) 在 namenode 的 hdfs-site.xml 配置文件中增加 dfs.hosts 属性

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

- (3) 刷新 namenode

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfsadmin -refreshNodes
```

Refresh nodes successful

- (4) 更新 resourcemanager 节点

```
[atguigu@hadoop102 hadoop-2.7.2]$ yarn rmadmin -refreshNodes
```

```
17/06/24 14:17:11 INFO client.RMProxy: Connecting to ResourceManager at
hadoop103/192.168.1.103:8033
```

- (5) 在 NameNode 的 slaves 文件中增加新主机名称

增加 105

hadoop102

hadoop103

hadoop104

hadoop105

(6) 单独命令启动新的数据节点和节点管理器

```
[atguigu@hadoop105 hadoop-2.7.2]$ sbin/hadoop-daemon.sh start datanode
```

```
starting datanode, logging to
/opt/module/hadoop-2.7.2/logs/hadoop-atguigu-datanode-hadoop105.out
```

```
[atguigu@hadoop105 hadoop-2.7.2]$ sbin/yarn-daemon.sh start nodemanager
```

```
starting nodemanager, logging to
/opt/module/hadoop-2.7.2/logs/yarn-atguigu-nodemanager-hadoop105.out
```

(7) 在 web 浏览器上检查是否 ok

3) 如果数据不均衡, 可以用命令实现集群的再平衡

```
[atguigu@hadoop102 sbin]$ ./start-balancer.sh
```

```
starting balancer, logging to
/opt/module/hadoop-2.7.2/logs/hadoop-atguigu-balancer-hadoop102.out
```

```
Time Stamp          Iteration#  Bytes Already Moved  Bytes Left To Move
Bytes Being Moved
```

6.5 退役旧数据节点

1) 在 namenode 的 /opt/module/hadoop-2.7.2/etc/hadoop 目录下创建 dfs.hosts.exclude 文件

```
[atguigu@hadoop102 hadoop]$ pwd
```

```
/opt/module/hadoop-2.7.2/etc/hadoop
```

```
[atguigu@hadoop102 hadoop]$ touch dfs.hosts.exclude
```

```
[atguigu@hadoop102 hadoop]$ vi dfs.hosts.exclude
```

添加如下主机名称 (要退役的节点)

hadoop105

2) 在 namenode 的 hdfs-site.xml 配置文件中增加 dfs.hosts.exclude 属性

```
<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

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfsadmin -refreshNodes
```

Refresh nodes successful

```
[atguigu@hadoop102 hadoop-2.7.2]$ yarn rmadmin -refreshNodes
```

```
17/06/24 14:55:56 INFO client.RMProxy: Connecting to ResourceManager at
hadoop103/192.168.1.103:8033
```

4) 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。

hadoop105:50010 (192.168.1.105:50010)	0	Decommission In Progress	9.72 GB	190.11 MB	4.13 GB	5.4 GB	13	190.11 MB (1.91%)	0	2.7.2
--	---	--------------------------	---------	-----------	---------	--------	----	----------------------	---	-------

5) 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。

hadoop105:50010 (192.168.1.105:50010)	0	Decommissioned	9.72 GB	190.11 MB	4.13 GB	5.4 GB	13	190.11 MB (1.91%)	0	2.7.2
--	---	----------------	---------	-----------	---------	--------	----	----------------------	---	-------

```
[atguigu@hadoop105 hadoop-2.7.2]$ sbin/hadoop-daemon.sh stop datanode
```

stopping datanode

```
[atguigu@hadoop105 hadoop-2.7.2]$ sbin/yarn-daemon.sh stop nodemanager
```

stopping nodemanager

6) 从 include 文件中删除退役节点，再运行刷新节点的命令

(1) 从 namenode 的 dfs.hosts 文件中删除退役节点 **hadoop105**

hadoop102

hadoop103

hadoop104

(2) 刷新 namenode，刷新 resourcemanager

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfsadmin -refreshNodes
```

Refresh nodes successful

```
[atguigu@hadoop102 hadoop-2.7.2]$ yarn rmadmin -refreshNodes
```

```
17/06/24 14:55:56 INFO client.RMProxy: Connecting to ResourceManager at
```

hadoop103/192.168.1.103:8033

- 7) 从 namenode 的 slave 文件中删除退役节点 hadoop105

hadoop102

hadoop103

hadoop104

- 8) 如果数据不均衡，可以用命令实现集群的再平衡

[atguigu@hadoop102 hadoop-2.7.2]\$ sbin/start-balancer.sh

```
starting balancer, logging to
/opt/module/hadoop-2.7.2/logs/hadoop-atguigu-balancer-hadoop102.out
Time Stamp      Iteration#  Bytes Already Moved  Bytes Left To Move
Bytes Being Moved
```

6.6 Datanode 多目录配置

- 1) 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。

- 2) 具体配置如下：

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>
```

七 HDFS 2.X 新特性

7.1 集群间数据拷贝

1) scp 实现两个远程主机之间的文件复制

```
scp -r hello.txt root@hadoop103:/user/atguigu/hello.txt // 推 push
```

```
scp -r root@hadoop103:/user/atguigu/hello.txt hello.txt // 拉 pull
```

```
scp -r root@hadoop103:/user/atguigu/hello.txt root@hadoop104:/user/atguigu //是通过本地主机中转实现两个远程主机的文件复制；如果在两个远程主机之间 ssh 没有配置的情况下可以使用该方式。
```

2) 采用 discp 命令实现两个 hadoop 集群之间的递归数据复制

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hadoop distcp  
hdfs://hadoop102:9000/user/atguigu/hello.txt hdfs://hadoop103:9000/user/atguigu/hello.txt
```

7.2 Hadoop 存档

1) hdfs 存储小文件弊端

每个文件均按块存储，每个块的元数据存储存储在 NameNode 的内存中，因此 hadoop 存储小文件会非常低效。因为大量的小文件会耗尽 NameNode 中的大部分内存。但注意，存储小文件所需要的磁盘容量和存储这些文件原始内容所需要的磁盘空间相比也不会增多。例如，一个 1MB 的文件以大小为 128MB 的块存储，使用的是 1MB 的磁盘空间，而不是 128MB。

2) 解决存储小文件办法之一

Hadoop 存档文件或 HAR 文件，是一个更高效的文件存档工具，它将文件存入 HDFS 块，在减少 NameNode 内存使用的同时，允许对文件进行透明的访问。具体说来，Hadoop 存档文件对内还是一个一个独立文件，对 NameNode 而言却是一个整体，减少了 NameNode 的内存。

3) 案例实操

(1) 需要启动 yarn 进程

```
[atguigu@hadoop102 hadoop-2.7.2]$ start-yarn.sh
```

(2) 归档文件

把/user/atguigu 目录里面的所有文件归档成一个叫 myhar.har 的归档文件，并把归档后文件存储到/user/my 路径下。


```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hadoop archive -archiveName myhar.har -p /user/atguigu /user/my
```

(3) 查看归档

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -lsr /user/my/myhar.har
```

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -lsr har:///myhar.har
```

(4) 解归档文件

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -cp har:/// user/my/myhar.har /* /user/atguigu
```

7.3 快照管理

快照相当于对目录做一个备份。并不会立即复制所有文件，而是指向同一个文件。当写入发生时，才会产生新文件。

1) 基本语法

(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)

2) 案例实操

(1) 开启/禁用指定目录的快照功能

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfsadmin -allowSnapshot /user/atguigu/data
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfsadmin -disallowSnapshot /user/atguigu/data
```

(2) 对目录创建快照

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfs -createSnapshot /user/atguigu/data
```

通过 web 访问 `hdfs://hadoop102:50070/user/atguigu/data/.snapshot/s.....//` 快照和源文

件使用相同数据块

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfs -lsr /user/atguigu/data/.snapshot/
```

(3) 指定名称创建快照

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfs -createSnapshot /user/atguigu/data  
miao170508
```

(4) 重命名快照

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfs -renameSnapshot /user/atguigu/data/  
miao170508 atguigu170508
```

(5) 列出当前用户所有可快照目录

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs lsSnapshottableDir
```

(6) 比较两个快照目录的不同之处

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs snapshotDiff  
/user/atguigu/data/ . .snapshot/atguigu170508
```

(7) 恢复快照

```
[atguigu@hadoop102 hadoop-2.7.2]$ hdfs dfs -cp  
/user/atguigu/input/.snapshot/s20170708-134303.027 /user
```

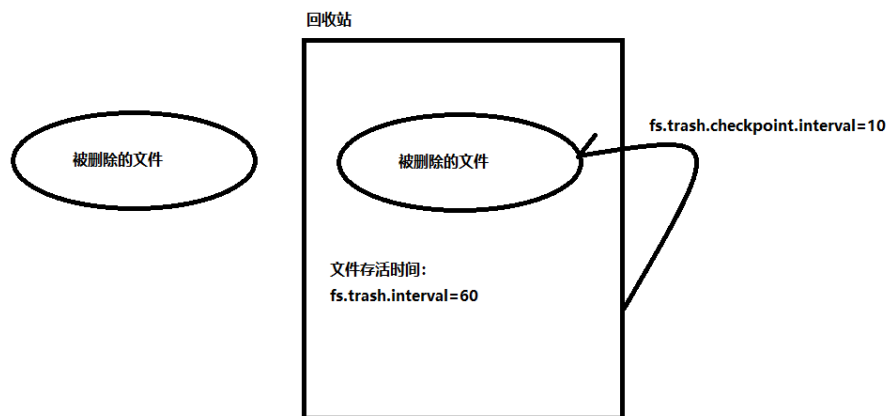
7.4 回收站

1) 默认回收站

默认值 `fs.trash.interval=0`，0 表示禁用回收站，可以设置删除文件的存活时间。

默认值 `fs.trash.checkpoint.interval=0`，检查回收站的间隔时间。**如果该值为 0，则该值设置和 `fs.trash.interval` 的参数值相等。**

要求 `fs.trash.checkpoint.interval<=fs.trash.interval`。



2) 启用回收站

修改 core-site.xml，配置垃圾回收时间为 1 分钟。

```
<property>
  <name>fs.trash.interval</name>
  <value>1</value>
</property>
```

3) 查看回收站

回收站在集群中的；路径：/user/atguigu/.Trash/....

4) 修改访问垃圾回收站用户名称

进入垃圾回收站用户名称，默认是 dr.who，修改为 atguigu 用户

[core-site.xml]

```
<property>
  <name>hadoop.http.staticuser.user</name>
  <value>atguigu</value>
</property>
```

5) 通过程序删除的文件不会经过回收站，需要调用 moveToTrash()才进入回收站

```
Trash trash = New Trash(conf);
```

```
trash.moveToTrash(path);
```

6) 恢复回收站数据

```
[atguigu@hadoop102 ~]$ hadoop-2.7.2$ hadoop fs -mv
/user/atguigu/.Trash/Current/user/atguigu/input /user/atguigu/input
```

7) 清空回收站

```
[atguigu@hadoop102 ~]$ hadoop-2.7.2$ hadoop fs -expunge
```

八 HDFS HA 高可用

8.1 HA 概述

- 1) 所谓 HA (high available), 即高可用 (7*24 小时不中断服务)。
- 2) 实现高可用最关键的策略是消除单点故障。HA 严格来说应该分成各个组件的 HA 机制: HDFS 的 HA 和 YARN 的 HA。

- 3) Hadoop2.0 之前, 在 HDFS 集群中 NameNode 存在单点故障 (SPOF)。

- 4) NameNode 主要在以下两个方面影响 HDFS 集群

NameNode 机器发生意外, 如宕机, 集群将无法使用, 直到管理员重启

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

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。

8.2 HDFS-HA 工作机制

- 1) 通过双 namenode 消除单点故障

8.2.1 HDFS-HA 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

8.2.2 HDFS-HA 自动故障转移工作机制

前面学习了使用命令 `hdfs haadmin -failover` 手动进行故障转移，在该模式下，即使现役 NameNode 已经失效，系统也不会自动从现役 NameNode 转移到待机 NameNode，下面学习如何配置部署 HA 自动进行故障转移。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。

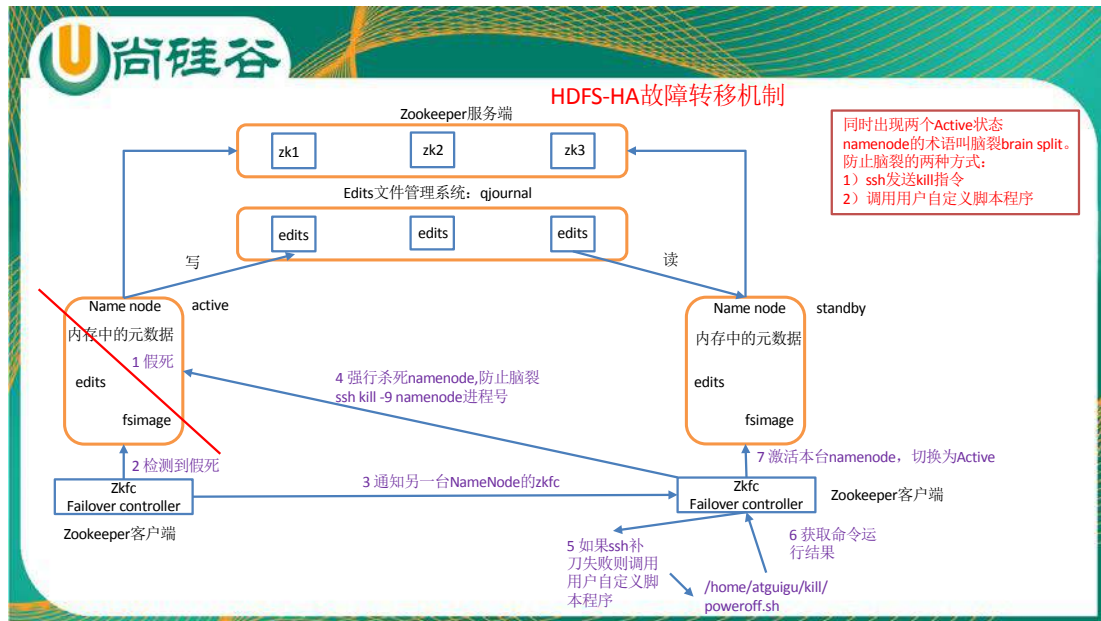
2) 现役 NameNode 选择：ZooKeeper 提供了一个简单的机制用于唯一的选择一个节点为 active 状态。如果目前现役 NameNode 崩溃，另一个节点可能从 ZooKeeper 获得特殊的排外锁以表明它应该成为现役 NameNode。

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，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，如果会话终止，锁节点将自动删除。

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。



8.3 HDFS-HA 集群配置

8.3.1 环境准备

- 1) 修改 IP
- 2) 修改主机名及主机名和 IP 地址的映射
- 3) 关闭防火墙
- 4) ssh 免密登录
- 5) 安装 JDK，配置环境变量等

8.3.2 规划集群

hadoop102	hadoop103	hadoop104
NameNode	NameNode	
JournalNode	JournalNode	JournalNode
DataNode	DataNode	DataNode
ZK	ZK	ZK
	ResourceManager	
NodeManager	NodeManager	NodeManager

8.3.3 配置 Zookeeper 集群

0) 集群规划

在 hadoop102、hadoop103 和 hadoop104 三个节点上部署 Zookeeper。

1) 解压安装

(1) 解压 zookeeper 安装包到/opt/module/目录下

```
[atguigu@hadoop102 software]$ tar -zxvf zookeeper-3.4.10.tar.gz -C /opt/module/
```

(2) 在/opt/module/zookeeper-3.4.10/这个目录下创建 zkData

```
mkdir -p zkData
```

(3) 重命名/opt/module/zookeeper-3.4.10/conf 这个目录下的 zoo_sample.cfg 为 zoo.cfg

```
mv zoo_sample.cfg zoo.cfg
```

2) 配置 zoo.cfg 文件

(1) 具体配置

```
dataDir=/opt/module/zookeeper-3.4.10/zkData
```

增加如下配置

```
#####cluster#####
```

```
server.2=hadoop102:2888:3888
```

```
server.3=hadoop103:2888:3888
```

```
server.4=hadoop104:2888:3888
```

(2) 配置参数解读

Server.A=B:C:D。

A 是一个数字，表示这个是第几号服务器；

B 是这个服务器的 ip 地址；

C 是这个服务器与集群中的 Leader 服务器交换信息的端口；

D 是万一集群中的 Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的 Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

集群模式下配置一个文件 myid，这个文件在 dataDir 目录下，这个文件里面有一个数据就是 A 的值，Zookeeper 启动时读取此文件，拿到里面的数据与 zoo.cfg 里面的配置信息比较从而判断到底是哪个 server。

3) 集群操作

(1) 在/opt/module/zookeeper-3.4.10/zkData 目录下创建一个 myid 的文件

```
touch myid
```

添加 myid 文件，注意一定要在 linux 里面创建，在 notepad++ 里面很可能乱码

(2) 编辑 myid 文件

```
vi myid
```

在文件中添加与 server 对应的编号：如 2

(3) 拷贝配置好的 zookeeper 到其他机器上

```
scp -r zookeeper-3.4.10/ root@hadoop103.atguigu.com:/opt/app/
```

```
scp -r zookeeper-3.4.10/ root@hadoop104.atguigu.com:/opt/app/
```

并分别修改 myid 文件中内容为 3、4

(4) 分别启动 zookeeper

```
[root@hadoop102 zookeeper-3.4.10]# bin/zkServer.sh start
```

```
[root@hadoop103 zookeeper-3.4.10]# bin/zkServer.sh start
```

```
[root@hadoop104 zookeeper-3.4.10]# bin/zkServer.sh start
```

(5) 查看状态

```
[root@hadoop102 zookeeper-3.4.10]# bin/zkServer.sh status
```

JMX enabled by default

Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg

Mode: **follower**

```
[root@hadoop103 zookeeper-3.4.10]# bin/zkServer.sh status
```

JMX enabled by default

Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg

Mode: **leader**

```
[root@hadoop104 zookeeper-3.4.5]# bin/zkServer.sh status
```

JMX enabled by default

Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg

Mode: **follower**

8.3.4 配置 HDFS-HA 集群

1) 官方地址: <http://hadoop.apache.org/>

2) 在 opt 目录下创建一个 ha 文件夹

```
mkdir ha
```

3) 将/opt/app/下的 hadoop-2.7.2 拷贝到/opt/ha 目录下


```
cp -r hadoop-2.7.2/ /opt/ha/
```

4) 配置 hadoop-env.sh

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

5) 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>
```

6) 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>
</configuration>
```

```
</property>

<!-- nn1 的 http 通信地址 -->
<property>
  <name>dfs.namenode.http-address.mycluster.nn1</name>
  <value>hadoop102:50070</value>
</property>

<!-- nn2 的 http 通信地址 -->
<property>
  <name>dfs.namenode.http-address.mycluster.nn2</name>
  <value>hadoop103:50070</value>
</property>

<!-- 指定 NameNode 元数据在 JournalNode 上的存放位置 -->
<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 无密钥登录-->
<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-->
<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>
```

7) 拷贝配置好的 hadoop 环境到其他节点

8.3.5 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[nn1]上, 对其进行格式化, 并启动:

```
bin/hdfs namenode -format
```

```
sbin/hadoop-daemon.sh start namenode
```

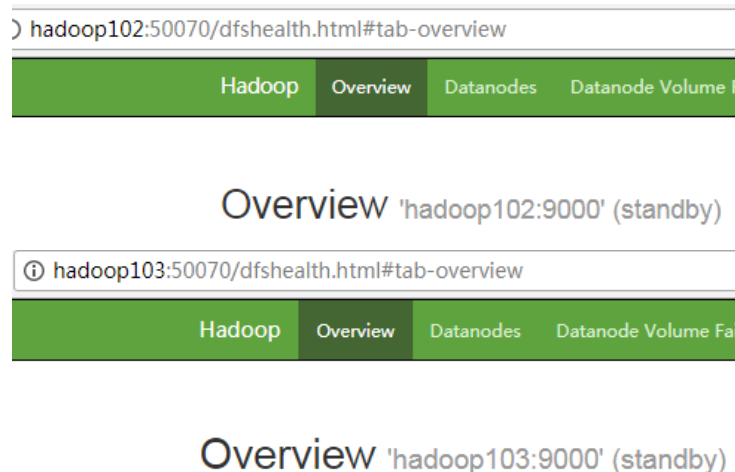
3) 在[nn2]上, 同步 nn1 的元数据信息:

```
bin/hdfs namenode -bootstrapStandby
```

4) 启动[nn2]:

```
sbin/hadoop-daemon.sh start namenode
```

5) 查看 web 页面显示



6) 在[nn1]上, 启动所有 datanode

```
sbin/hadoop-daemons.sh start datanode
```

7) 将[nn1]切换为 Active

```
bin/hdfs haadmin -transitionToActive nn1
```

8) 查看是否 Active

```
bin/hdfs haadmin -getServiceState nn1
```

8.3.6 配置 HDFS-HA 自动故障转移

1) 具体配置

(1) 在 hdfs-site.xml 中增加

```
<property>
  <name>dfs.ha.automatic-failover.enabled</name>
  <value>true</value>
</property>
```

(2) 在 core-site.xml 文件中增加

```
<property>
  <name>ha.zookeeper.quorum</name>
  <value>hadoop102:2181,hadoop103:2181,hadoop104:2181</value>
</property>
```

2) 启动

(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
```

3) 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
```

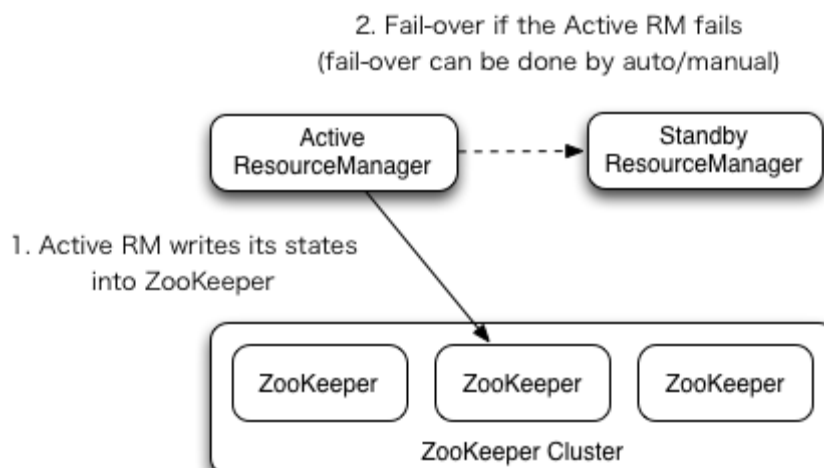
8.4 YARN-HA 配置

8.4.1 YARN-HA 工作机制

1) 官方文档:

<http://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html>

2) YARN-HA 工作机制



8.4.2 配置 YARN-HA 集群

0) 环境准备

- (1) 修改 IP
- (2) 修改主机名及主机名和 IP 地址的映射
- (3) 关闭防火墙
- (4) ssh 免密登录
- (5) 安装 JDK，配置环境变量等
- (6) 配置 Zookeeper 集群

1) 规划集群

hadoop102	hadoop103	hadoop104
NameNode	NameNode	
JournalNode	JournalNode	JournalNode
DataNode	DataNode	DataNode
ZK	ZK	ZK
ResourceManager	ResourceManager	

NodeManager

NodeManager

NodeManager

2) 具体配置

(1) 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<
    /value>
</property>

</configuration>
```

(2) 同步更新其他节点的配置信息

3) 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[nn1]上, 对其进行格式化, 并启动:

```
bin/hdfs namenode -format
```

```
sbin/hadoop-daemon.sh start namenode
```

(3) 在[nn2]上, 同步 nn1 的元数据信息:

```
bin/hdfs namenode -bootstrapStandby
```

(4) 启动[nn2]:

```
sbin/hadoop-daemon.sh start namenode
```

(5) 启动所有 datanode

```
sbin/hadoop-daemons.sh start datanode
```

(6) 将[nn1]切换为 Active

```
bin/hdfs haadmin -transitionToActive nn1
```

4) 启动 yarn

(1) 在 hadoop102 中执行:

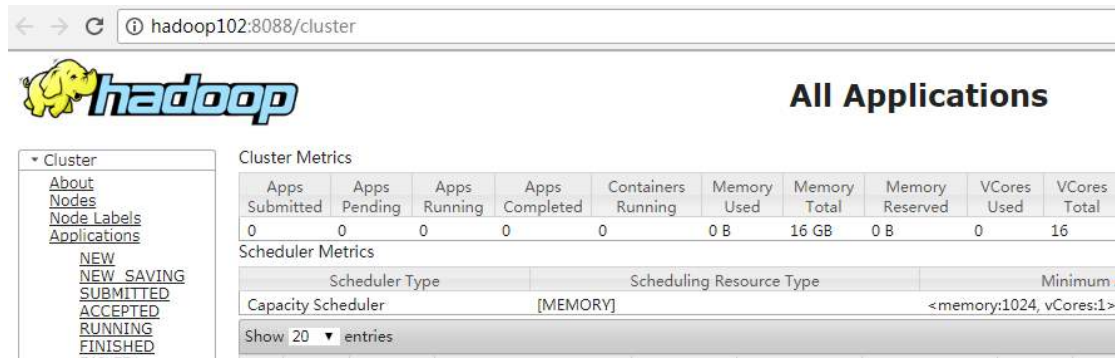
```
sbin/start-yarn.sh
```

(2) 在 hadoop103 中执行:

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

(3) 查看服务状态

```
bin/yarn rmadmin -getServiceState rml
```



The screenshot shows the Hadoop web interface at `hadoop102:8088/cluster`. It displays the Hadoop logo and a sidebar with navigation links like 'Cluster', 'About', 'Nodes', 'Node Labels', 'Applications', 'NEW', 'NEW SAVING', 'SUBMITTED', 'ACCEPTED', 'RUNNING', 'FINISHED', and 'FAILED'. The main content area is titled 'All Applications' and shows 'Cluster Metrics' and 'Scheduler Metrics'.

Cluster Metrics									
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total
0	0	0	0	0	0 B	16 GB	0 B	0	16

Scheduler Metrics		
Scheduler Type	Scheduling Resource Type	Minimum
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>

Below the scheduler metrics, there is a 'Show 20 entries' dropdown.

8.5 HDFS Federation 架构设计

1) 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。

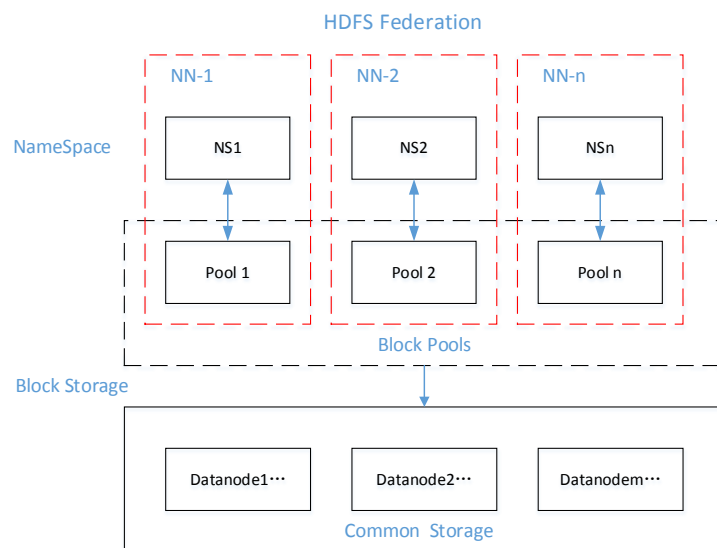
(3) 性能的瓶颈

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。

2) HDFS Federation 架构设计

能不能有多个 NameNode

NameNode	NameNode	NameNode
元数据	元数据	元数据
Log	machine	电商数据/话单数据



3) HDFS Federation 应用思考

不同应用可以使用不同 NameNode 进行数据管理

图片业务、爬虫业务、日志审计业务

Hadoop 生态系统中，不同的框架使用不同的 namenode 进行管理 namespace。（隔离性）

一 MapReduce 入门

1.1 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。

1.2 MapReduce 优缺点

1.2.1 优点

1) **MapReduce 易于编程**。它简单的实现一些接口，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的 PC 机器上运行。也就是说你写一个分布式程序，跟写一个简单的串行程序是一模一样的。就是因为这个特点使得 MapReduce 编程变得非常流行。

2) **良好的扩展性**。当你的计算资源不能得到满足的时候，你可以通过简单的增加机器来扩展它的计算能力。

3) **高容错性**。MapReduce 设计的初衷就是使程序能够部署在廉价的 PC 机器上，这就要求它具有很高的容错性。比如其中一台机器挂了，它可以把上面的计算任务转移到另外一个节点上运行，不至于这个任务运行失败，而且这个过程不需要人工参与，而完全是由 Hadoop 内部完成的。

4) **适合 PB 级以上海量数据的离线处理**。这里加红字体离线处理，说明它适合离线处理而不适合在线处理。比如像毫秒级别的返回一个结果，MapReduce 很难做到。

1.2.2 缺点

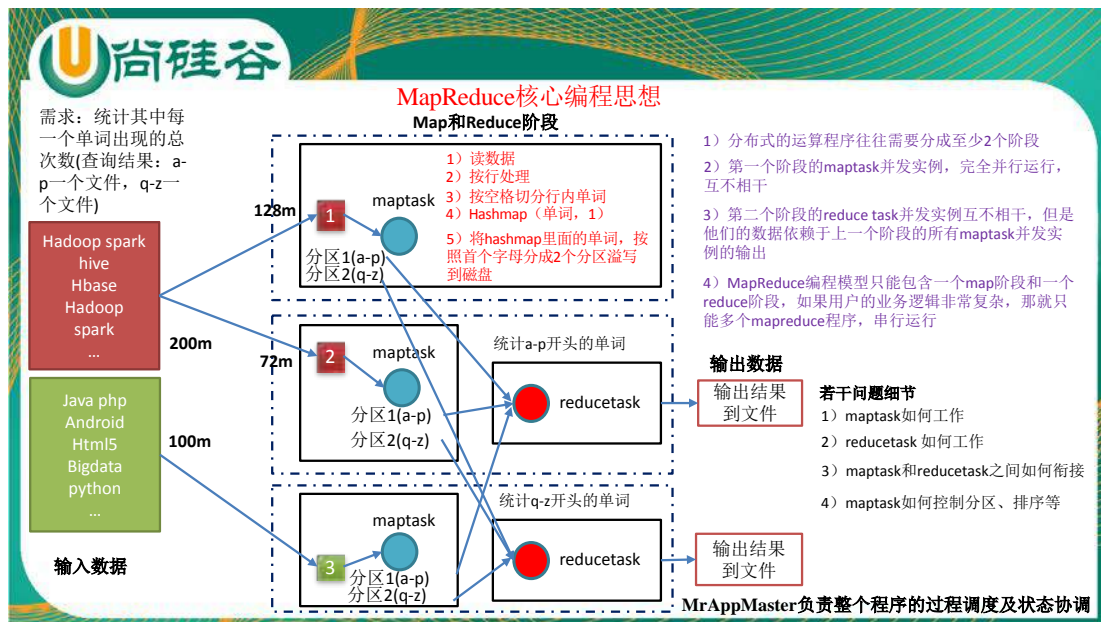
MapReduce 不擅长做实时计算、流式计算、DAG（有向图）计算。

1) **实时计算**。MapReduce 无法像 Mysql 一样，在毫秒或者秒级内返回结果。

2) **流式计算**。流式计算的输入数据是动态的，而 MapReduce 的输入数据集是静态的，不能动态变化。这是因为 MapReduce 自身的设计特点决定了数据源必须是静态的。

3) **DAG（有向图）计算**。多个应用程序存在依赖关系，后一个应用程序的输入为前一个的输出。在这种情况下，MapReduce 并不是不能做，而是使用后，每个 MapReduce 作业的输出结果都会写入到磁盘，会造成大量的磁盘 IO，导致性能非常的低下。

1.3 MapReduce 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。
- 2) 第一个阶段的 m1task 并发实例，完全并行运行，互不相干。
- 3) 第二个阶段的 reduce task 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 m1task 并发实例的输出。
- 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。

1.4 MapReduce 进程

一个完整的 mapreduce 程序在分布式运行时有三类实例进程：

- 1) MrAppMaster：负责整个程序的过程调度及状态协调。
- 2) MapTask：负责 map 阶段的整个数据处理流程。
- 3) ReduceTask：负责 reduce 阶段的整个数据处理流程。

1.5 MapReduce 编程规范

用户编写的程序分成三个部分：Mapper、Reducer 和 Driver。

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

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

1.6 WordCount 案例实操

1) 需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

2) 数据准备：



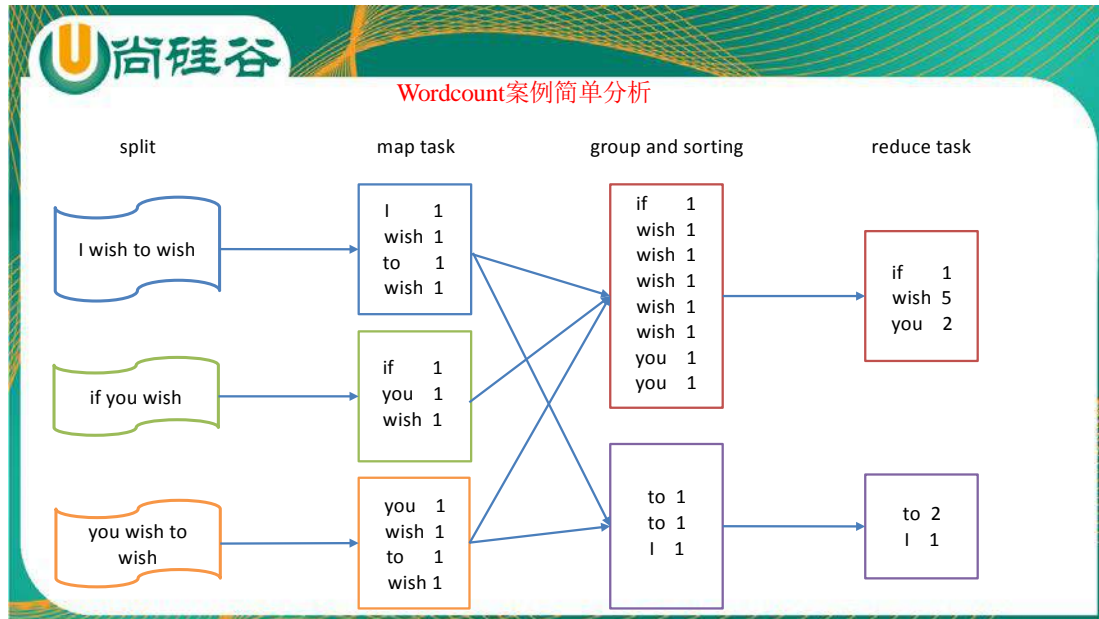
hello.txt

3) 分析

按照 mapreduce 编程规范，分别编写 Mapper，Reducer，Driver。

需求：统计一堆文件中单词出现的个数（WordCount案例）

Mapper	Reducer	Driver
// 1 将maptask传给我们的文本内容先转换成String	// 1 汇总各个key的个数	// 1 获取配置信息，获取job对象实例
		// 2 指定本程序的jar包所在的本地路径
		// 3 关联mapper/Reducer业务类
// 2 根据空格将这一行切分成单词	// 2输出该key的总次数	// 4 指定mapper输出数据的kv类型
		// 5 指定最终输出的数据的kv类型
		// 6 指定job的输入原始文件所在目录
// 3 将单词输出为<单词，1>		// 7 提交



4) 编写程序

(1) 编写 mapper 类

```
package com.atguigu.mapreduce;
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;

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);
    }
}
}
```

(2) 编写 reducer 类

```
package com.atguigu.mapreduce.wordcount;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

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

    int sum;
    LongWritable v = new LongWritable();

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

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

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

(3) 编写驱动类

```
package com.atguigu.mapreduce.wordcount;
import java.io.IOException;
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 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 Reduce 输出
    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
    boolean result = job.waitForCompletion(true);

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

```

5) 本地测试

(1) 在 windows 环境上配置 HADOOP_HOME 环境变量

(2) 在 eclipse 上运行程序

(3) 注意：如果 eclipse 打印不出日志，在控制台上只显示

```

1.log4j:WARN No appenders could be found for logger (org.apache.hadoop.util.Shell).
2.log4j:WARN Please initialize the log4j system properly.
3.log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

```

需要在项目的 src 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) 集群上测试

(1) 在依赖中添加

```
<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.WordCountReduce</mainClass>
          </manifest>
        </archive>
      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```


(2) 将程序打成 jar 包，然后拷贝到 hadoop 集群中

(3) 启动 hadoop 集群

(4) 执行 wordcount 程序

```
[atguigu@hadoop102 ~]$ hadoop jar wc.jar  
com.atguigu.wordcount.WordcountDriver /user/atguigu/input /user/atguigu/output1
```

二 Hadoop 序列化

2.1 序列化概述

1) 什么是序列化

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（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）或者是硬盘的持久化数据，转换成内存中的对象。

2) 为什么要序列化

一般来说，“活的”对象只生存在内存里，关机断电就没有了。而且“活的”对象只能由本地的进程使用，不能被发送到网络上的另外一台计算机。然而序列化可以存储“活的”对象，可以将“活的”对象发送到远程计算机。

3) 为什么不用 Java 的序列化

Java 的序列化是一个重量级序列化框架（`Serializable`），一个对象被序列化后，会附带很多额外的信息（各种校验信息，`header`，继承体系等），不便于在网络中高效传输。所以，hadoop 自己开发了一套序列化机制（`Writable`），特点如下：

（1）紧凑：紧凑的格式能让我们充分利用网络带宽，而带宽是数据中心最稀缺的资源

（2）快速：进程通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是基本的；

（3）可扩展：协议为了满足新的需求变化，所以控制客户端和服务端过程中，需要直接引进相应的协议，这些是新协议，原序列化方式能支持新的协议报文；

（4）互操作：能支持不同语言写的客户端和服务端进行交互；

2.2 常用数据序列化类型

常用的数据类型对应的 hadoop 数据序列化类型

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

2.3 自定义 bean 对象实现序列化接口（Writable）

1) 自定义 bean 对象要想序列化传输，必须实现序列化接口，需要注意以下 7 项。

(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

```
public FlowBean() {
    super();
}
```

(3) 重写序列化方法

```
@Override
public void write(DataOutput out) throws IOException {
    out.writeLong(upFlow);
    out.writeLong(downFlow);
    out.writeLong(sumFlow);
}
```

(4) 重写反序列化方法

```
@Override
public void readFields(DataInput in) throws IOException {
    upFlow = in.readLong();
    downFlow = in.readLong();
    sumFlow = in.readLong();
}
```

(5) 注意反序列化的顺序和序列化的顺序完全一致

(6) 要想把结果显示在文件中，需要重写 toString()，可用“\t”分开，方便后续用。

(7) 如果需要将自定义的 bean 放在 key 中传输，则还需要实现 comparable 接口，因为 mapreduce 框中的 shuffle 过程一定会对 key 进行排序。

```
@Override
public int compareTo(FlowBean o) {
    // 倒序排列，从大到小
    return this.sumFlow > o.getSumFlow() ? -1 : 1;
}
```

2.4 序列化案例实操

1) 需求：

统计每一个手机号耗费的总上行流量、下行流量、总流量

2) 数据准备



phone_data.txt

输入数据格式:

1363157993055	13560436666	C4-17-FE-BA-DE-D9:CMCC	120.196.100.99	18	15	1116	954	200
手机号码						上行流量 下行流量		

输出数据格式

1356-0436666	1116	954	2070
手机号码	上行流量	下行流量	总流量

3) 分析

基本思路:

Map 阶段:

- (1) 读取一行数据, 切分字段
- (2) 抽取手机号、上行流量、下行流量
- (3) 以手机号为 key, bean 对象为 value 输出, 即 `context.write(手机号,bean);`

Reduce 阶段:

- (1) 累加上行流量和下行流量得到总流量。
- (2) 实现自定义的 bean 来封装流量信息, 并将 bean 作为 map 输出的 key 来传输
- (3) MR 程序在处理数据的过程中会对数据排序(map 输出的 kv 对传输到 reduce 之前, 会排序), 排序的依据是 map 输出的 key

所以, 我们如果要想实现自己需要的排序规则, 则可以考虑将排序因素放到 key 中, 让 key 实现接口: **WritableComparable**。

然后重写 key 的 **compareTo** 方法。

4) 编写 mapreduce 程序

- (1) 编写流量统计的 bean 对象

```
package com.atguigu.mapreduce.flowsum;
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{
```

```
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
@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，方便后续打印到文本
@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;
    }
}

```

(2) 编写 mapper

```

package com.atguigu.mapreduce.flowsum;
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
        long upFlow = Long.parseLong(fields[fields.length - 3]);
        long downFlow = Long.parseLong(fields[fields.length - 2]);

        v.set(downFlow, upFlow);

        // 4 写出
        context.write(new Text(phoneNum), new FlowBean(upFlow, downFlow));
    }
}

```

(3) 编写 reducer

```

package com.atguigu.mapreduce.flowsum;
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
        for (FlowBean flowBean : values) {
            sum_upFlow += flowBean.getSumFlow();
            sum_downFlow += flowBean.getDownFlow();
        }

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

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

```

(4) 编写驱动

```

package com.atguigu.mapreduce.flowsum;
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 FlowsunDriver {

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

        // 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(FlowsunDriver.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);

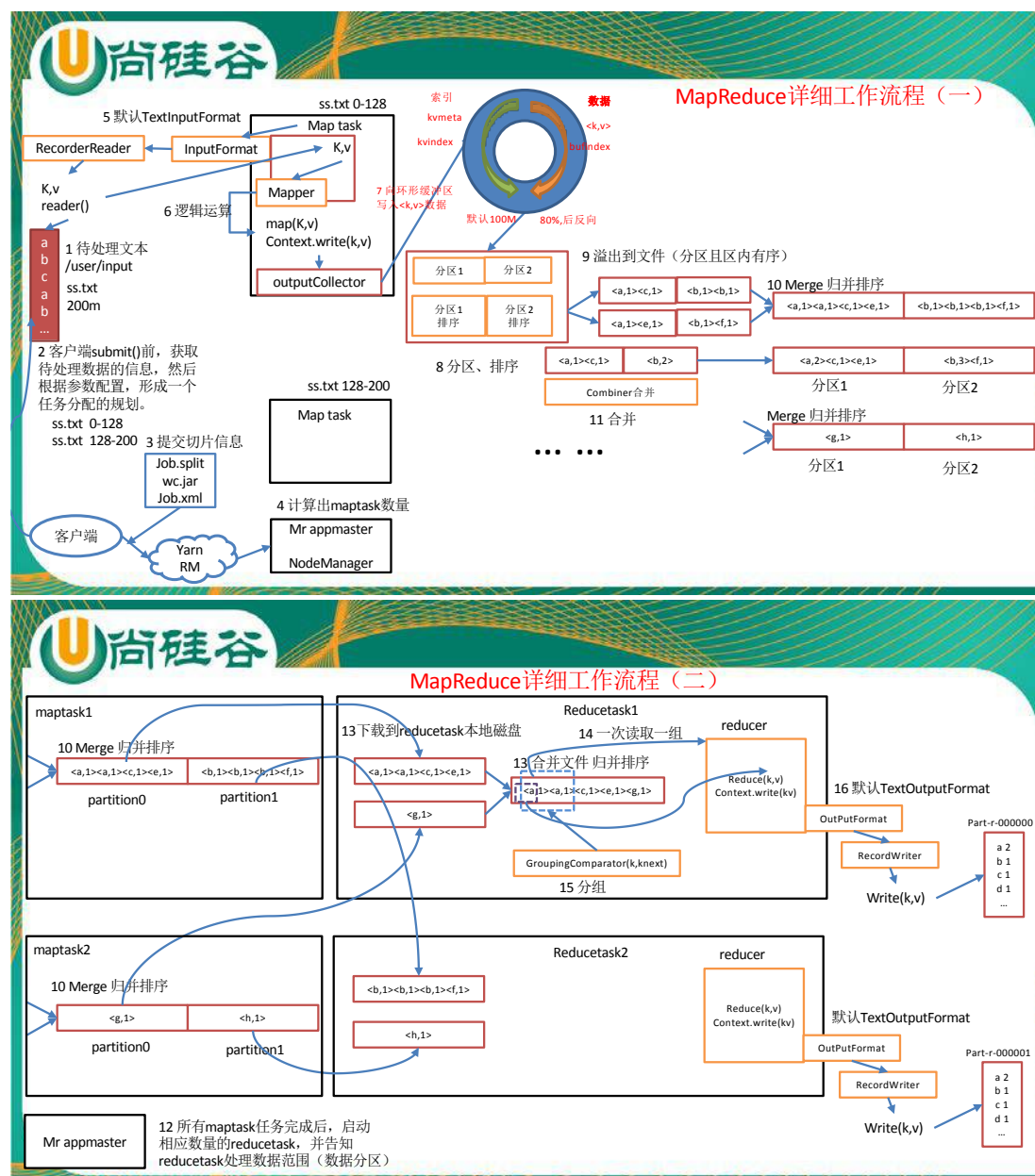
        // 5 指定 job 的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 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);
    }
}
```


三 MapReduce 框架原理

3.1 MapReduce 工作流程

1) 流程示意图



2) 流程详解

上面的流程是整个 mapreduce 最全流程，但是 shuffle 过程只是从第 7 步开始到第 16 步结束，具体 shuffle 过程详解，如下：

- 1) maptask 收集我们的 map() 方法输出的 kv 对，放到内存缓冲区中
- 2) 从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件
- 3) 多个溢出文件会被合并成大的溢出文件

4) 在溢出过程中, 及合并的过程中, 都要调用 `partitioner` 进行分区和针对 `key` 进行排序

5) `reducetask` 根据自己的分区号, 去各个 `maptask` 机器上取相应的结果分区数据

6) `reducetask` 会取到同一个分区的来自不同 `maptask` 的结果文件, `reducetask` 会将这些文件再进行合并 (归并排序)

7) 合并成大文件后, `shuffle` 的过程也就结束了, 后面进入 `reducetask` 的逻辑运算过程 (从文件中取出一个一个的键值对 `group`, 调用用户自定义的 `reduce()` 方法)

3) 注意

`Shuffle` 中的缓冲区大小会影响到 `mapreduce` 程序的执行效率, 原则上说, 缓冲区越大, 磁盘 `io` 的次数越少, 执行速度就越快。

缓冲区的大小可以通过参数调整, 参数: `io.sort.mb` 默认 100M。

3.2 InputFormat 数据输入

3.2.1 Job 提交流程和切片源码详解

1) job 提交流程源码详解

```
waitForCompletion()

submit();

// 1 建立连接

connect();

// 1) 创建提交 job 的代理

new Cluster(getConfiguration());

// (1) 判断是本地 yarn 还是远程

initialize(jobTrackAddr, conf);

// 2 提交 job

submitter.submitJobInternal(Job.this, cluster)

// 1) 创建给集群提交数据的 Stag 路径

Path jobStagingArea = JobSubmissionFiles.getStagingDir(cluster, conf);

// 2) 获取 jobid , 并创建 job 路径

JobID jobId = submitClient.getNewJobID();

// 3) 拷贝 jar 包到集群
```

```

copyAndConfigureFiles(job, submitJobDir);

rUploader.uploadFiles(job, jobSubmitDir);

// 4) 计算切片, 生成切片规划文件

writeSplits(job, submitJobDir);

maps = writeNewSplits(job, jobSubmitDir);

input.getSplits(job);

// 5) 向 Stag 路径写 xml 配置文件

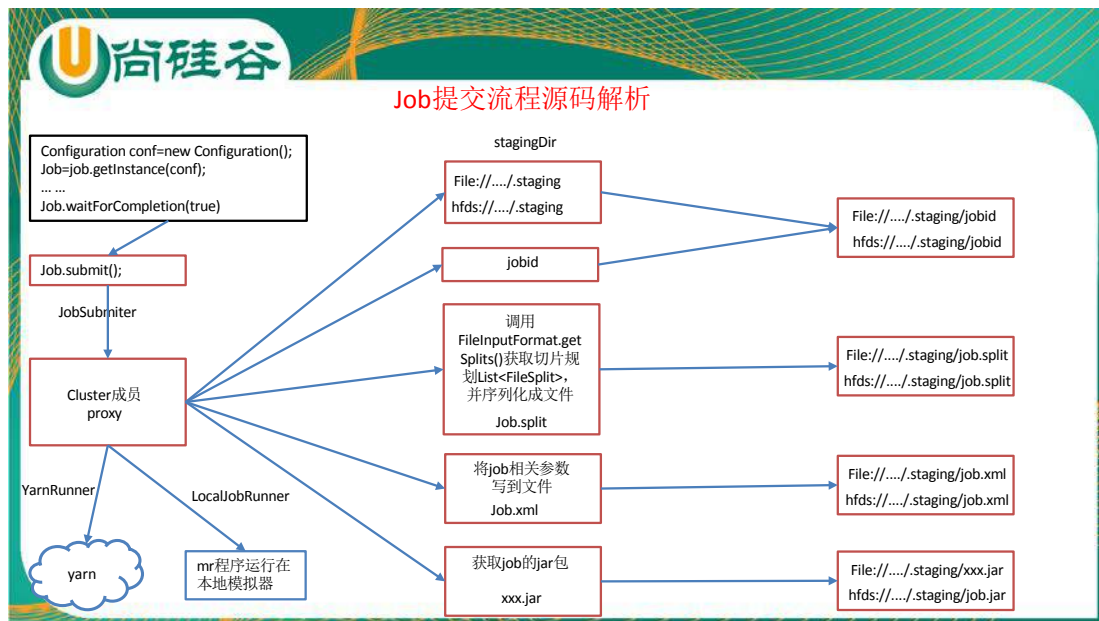
writeConf(conf, submitJobFile);

conf.writeXml(out);

// 6) 提交 job, 返回提交状态

status = submitClient.submitJob(jobId, submitJobDir.toString(),
job.getCredentials());

```



2) FileInputFormat 源码解析(input.getSplits(job))

- (1) 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

`computeSliteSize(Math.max(minSize,Math.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) 数据切片只是在逻辑上对输入数据进行分片，并不会再磁盘上将其切分成分片进行存储。InputSplit 只记录了分片的元数据信息，比如起始位置、长度以及所在的节点列表等

h) 注意：block 是 HDFS 物理上存储的数据，切片是对数据逻辑上的划分

(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。

3.2.2 FileInputFormat 切片机制

1) FileInputFormat 中默认切片机制：

(1) 简单地按照文件的内容长度进行切片

(2) 切片大小，默认等于 block 大小

(3) 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

比如待处理数据有两个文件：

file1.txt	320M
file2.txt	10M

经过 FileInputFormat 的切片机制运算后，形成的切片信息如下：

file1.txt.split1--	0~128
file1.txt.split2--	128~256
file1.txt.split3--	256~320
file2.txt.split1--	0~10M

2) FileInputFormat 切片大小的参数配置

通过分析源码，在 FileInputFormat 中，计算切片大小的逻辑： $\text{Math.max}(\text{minSize}, \text{Math.min}(\text{maxSize}, \text{blockSize}))$;

切片主要由这几个值来运算决定

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。

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

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

3.2.3 CombineTextInputFormat 切片机制

关于大量小文件的优化策略

1) 默认情况下 TextInputFormat 对任务的切片机制是按文件规划切片，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。

2) 优化策略

(1) 最好的办法，在数据处理系统的最前端（预处理/采集），将小文件先合并成大文件，再上传到 HDFS 做后续分析。

(2) 补救措施：如果已经是大量小文件在 HDFS 中了，可以使用另一种 InputFormat 来做切片（CombineTextInputFormat），它的切片逻辑跟 TextFileInputFormat 不同：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。

(3) 优先满足最小切片大小，不超过最大切片大小

```
CombineTextInputFormat.setMaxInputSplitSize(job, 4194304); // 4m
```

```
CombineTextInputFormat.setMinInputSplitSize(job, 2097152); // 2m
```

举例：0.5m+1m+0.3m+5m=2m + 4.8m=2m + 4m + 0.8m

3) 具体实现步骤

```
// 如果不设置 InputFormat,它默认用的是 TextInputFormat.class
job.setInputFormatClass(CombineTextInputFormat.class)
CombineTextInputFormat.setMaxInputSplitSize(job, 4194304); // 4m
CombineTextInputFormat.setMinInputSplitSize(job, 2097152); // 2m
```

3.2.4 CombineTextInputFormat 案例实操

1) 需求：将输入的大量小文件合并成一个切片统一处理。

2) 输入数据: 准备 5 个小文件

3) 实现过程

(1) 不做任何处理, 运行需求 1 中的 wordcount 程序, 观察切片个数为 5

```
2017-05-31 10:15:48,759 INFO [org.apache.hadoop.mapreduce.lib.input.FileInputFormat] - Total input p
2017-05-31 10:15:48,787 INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:5
```

(2) 在 WordcountDriver 中增加如下代码, 运行程序, 并观察运行的切片个数为 1

```
// 如果不设置 InputFormat, 它默认用的是 TextInputFormat.class
job.setInputFormatClass(CombineTextInputFormat.class);
CombineTextInputFormat.setMaxInputSplitSize(job, 4194304); // 4m
CombineTextInputFormat.setMinInputSplitSize(job, 2097152); // 2m
```

```
2017-05-31 10:37:17,595 INFO [org.apache.hadoop.mapreduce.lib.input.CombineFileInputFormat] - DEBI
2017-05-31 10:37:17,611 INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:1
```

3.2.5 InputFormat 接口实现类

MapReduce 任务的输入文件一般是存储在 HDFS 里面。输入的文件格式包括: 基于行的日志文件、二进制格式文件等。这些文件一般会很大, 达到数十 GB, 甚至更大。那么 MapReduce 是如何读取这些数据呢? 下面我们首先学习 InputFormat 接口。

InputFormat 常见的接口实现类包括: TextInputFormat、KeyValueTextInputFormat、NLineInputFormat、CombineTextInputFormat 和自定义 InputFormat 等。

1) TextInputFormat

TextInputFormat 是默认的 InputFormat。每条记录是一行输入。键是 LongWritable 类型, 存储该行在整个文件中的字节偏移量。值是这行的内容, 不包括任何行终止符(换行符和回车符)。

以下是一个示例, 比如, 一个分片包含了如下 4 条文本记录。

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

每条记录表示为以下键/值对:

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

很明显，键并不是行号。一般情况下，很难取得行号，因为文件按字节而不是按行切分为分片。

2) 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, " ");`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。

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

每条记录表示为以下键/值对：

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

此时的键是每行排在制表符之前的 `Text` 序列。

3) NLineInputFormat

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。

以下是一个示例，仍然以上面的 4 行输入为例。

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

例如，如果 `N` 是 2，则每个输入分片包含两行。开启 2 个 `maptask`。

```
(0,Rich learning form)
```

(19,Intelligent learning engine)

另一个 mapper 则收到后两行:

(47,Learning more convenient)

(72,From the real demand for more close to the enterprise)

这里的键和值与 TextInputFormat 生成的一样。

3.2.6 KeyValueTextInputFormat 使用案例

1) 需求: 统计输入文件中每一行的第一个单词相同的行数。

2) 输入文件:

banzhang ni hao

xihuan hadoop banzhang dc

banzhang ni hao

xihuan hadoop banzhang dc

3) 输出

banzhang 2

xihuan 2

4) 代码实现

(1) 编写 mapper

```
package com.atguigu.mapreduce.KeyValueTextInputFormat;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class KVTextMapper extends Mapper<Text, Text, Text, LongWritable>{

    final Text k = new Text();
    final LongWritable v = new LongWritable();

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

        // banzhang ni hao
        // 1 设置 key 和 value
        // banzhang
        k.set(key);
```



```

        // 设置 key 的个数
        v.set(1);

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

```

(2) 编写 reducer

```

package com.atguigu.mapreduce.KeyValueTextInputFormat;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class KVTextReducer extends Reducer<Text, LongWritable, Text, LongWritable>{

    LongWritable v = new LongWritable();

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

        long count = 0L;
        // 1 汇总统计
        for (LongWritable value : values) {
            count += value.get();
        }

        v.set(count);

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

```

(3) 编写 Driver

```

package com.atguigu.mapreduce.keyvaleTextInputFormat;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

```

```

import org.apache.hadoop.mapreduce.lib.input.KeyValueLineRecordReader;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MyDriver {

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

        Configuration conf = new Configuration();
        // 设置切割符
        conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, " ");
        // 获取 job 对象
        Job job = Job.getInstance(conf);

        // 设置 jar 包位置，关联 mapper 和 reducer
        job.setJarByClass(MyDriver.class);
        job.setMapperClass(MyMapper.class);
        job.setOutputValueClass(LongWritable.class);

        // 设置 map 输出 kv 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);

        // 设置最终输出 kv 类型
        job.setReducerClass(MyReducer.class);
        job.setOutputKeyClass(Text.class);

        // 设置输入输出数据路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));

        // 设置输入格式
        job.setInputFormatClass(KeyValueTextInputFormat.class);

        // 设置输出数据路径
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 提交 job
        job.waitForCompletion(true);
    }
}

```

3.2.7 NLineInputFormat 使用案例

1) 需求：根据每个输入文件的行数来规定输出多少个切片。例如每三行放入一个切片中。

2) 输入数据:

```
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dcbanzhang ni hao
xihuan hadoop banzhang dc
```

3) 输出结果:

Number of splits:4

4) 代码实现:

(1) 编写 mapper

```
package com.atguigu.mapreduce.nline;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class NLineMapper extends Mapper<LongWritable, Text, Text, LongWritable>{

    private Text k = new Text();
    private LongWritable v = new LongWritable(1);

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

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

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

        // 3 循环写出
        for (int i = 0; i < splited.length; i++) {

            k.set(splited[i]);
```

```
        context.write(k, v);
    }
}
}
```

(2) 编写 Reducer

```
package com.atguigu.mapreduce.nline;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class NLineReducer extends Reducer<Text, LongWritable, Text, LongWritable>{

    LongWritable v = new LongWritable();

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

        long count = 0L;
        // 1 汇总
        for (LongWritable value : values) {
            count += value.get();
        }

        v.set(count);

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

(3) 编写 driver

```
package com.atguigu.mapreduce.nline;
import java.io.IOException;
import java.net.URISyntaxException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.NLineInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```

public class NLineDriver {

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

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

        // 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);

        // 设置 jar 包位置，关联 mapper 和 reducer
        job.setJarByClass(NLineDriver.class);
        job.setMapperClass(NLineMapper.class);
        job.setReducerClass(NLineReducer.class);

        // 设置 map 输出 kv 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);

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

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

        // 提交 job
        job.waitForCompletion(true);
    }
}

```

5) 结果查看

(1) 输入数据

banzhang ni hao

xihuan hadoop banzhang dc

banzhang ni hao

xihuan hadoop banzhang dc

banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dcbanzhang ni hao
xihuan hadoop banzhang dc

(2) 输出结果的切片数:

```
WARN [org.apache.hadoop.mapreduce.JobResourceUploader] - Hadoop command-line  
WARN [org.apache.hadoop.mapreduce.JobResourceUploader] - No job jar file set  
INFO [org.apache.hadoop.mapreduce.lib.input.FileInputFormat] - Total input p  
INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:4  
INFO [org.apache.hadoop.mapreduce.JobSubmitter] - Submitting tokens for job:  
INFO [org.apache.hadoop.mapreduce.Job] - The url to track the job: http://lo  
INFO [org.apache.hadoop.mapreduce.Job] - Running job: job_local998538859_000  
INFO [org.apache.hadoop.mapred.LocalJobRunner] - OutputCommitter set in conf  
INFO [org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter] - File Out
```

3.2.8 自定义 InputFormat

1) 概述

- (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。

3.2.9 自定义 InputFormat 案例实操

1) 需求

无论 hdfs 还是 mapreduce，对于小文件都有损效率，实践中，又难免面临处理大量小文件的场景，此时，就需要有相应解决方案。将多个小文件合并成一个文件 SequenceFile，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。

2) 输入数据



one.txt



two.txt



three.txt

最终预期文件格式:

part-r-00000

3) 分析

小文件的优化无非以下几种方式:

- (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

(2) 在业务处理之前，在 HDFS 上使用 `mapreduce` 程序对小文件进行合并

(3) 在 `mapreduce` 处理时，可采用 `CombineTextInputFormat` 提高效率

4) 具体实现

本节采用自定义 `InputFormat` 的方式，处理输入小文件的问题。

(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

5) 程序实现：

(1) 自定义 `InputFormat`

```
package com.atguigu.mapreduce.inputformat;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
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;

// 定义类继承 FileInputFormat
public class WholeFileInputformat extends FileInputFormat<NullWritable, BytesWritable>{

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

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

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

        return recordReader;
    }
}
```

(2) 自定义 RecordReader

```
package com.atguigu.mapreduce.inputformat;
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.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class WholeRecordReader extends RecordReader<NullWritable, BytesWritable>{

    private Configuration configuration;
    private FileSplit split;

    private boolean processed = false;
    private BytesWritable value = new BytesWritable();

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context) throws IOException,
    InterruptedException {

        this.split = (FileSplit)split;
        configuration = context.getConfiguration();
    }

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

        if (!processed) {
            // 1 定义缓存区
            byte[] contents = new byte[(int)split.getLength()];

            FileSystem fs = null;
            FSDataInputStream fis = null;

            try {
                // 2 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
        value.set(contents, 0, contents.length);
    } catch (Exception e) {

    } finally {
        IOUtils.closeStream(fis);
    }

    processed = true;

    return true;
}

return false;
}

@Override
public NullWritable getCurrentKey() throws IOException, InterruptedException {
    return NullWritable.get();
}

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

@Override
public float getProgress() throws IOException, InterruptedException {
    return processed? 1:0;
}

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

```

(3) SequenceFileMapper 处理流程

```

package com.atguigu.mapreduce.inputformat;
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<NullWritable, BytesWritable, Text,
BytesWritable>{

    Text k = new Text();

    @Override
    protected void setup(Mapper<NullWritable, BytesWritable, Text, BytesWritable>.Context
context)
        throws IOException, InterruptedException {
        // 1 获取文件切片信息
        FileSplit inputSplit = (FileSplit) context.getInputSplit();
        // 2 获取切片名称
        String name = inputSplit.getPath().toString();
        // 3 设置 key 的输出
        k.set(name);
    }

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

        context.write(k, value);
    }
}

```

(4) SequenceFileReducer 处理流程

```

package com.atguigu.mapreduce.inputformat;
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());
}
}
```

(5) SequenceFileDriver 处理流程

```
package com.atguigu.mapreduce.inputformat;
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 {

        args = new String[] { "e:/input/inputinputformat", "e:/output1" };
        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);
        job.setJarByClass(SequenceFileDriver.class);
        job.setMapperClass(SequenceFileMapper.class);
        job.setReducerClass(SequenceFileReducer.class);

        // 设置输入的 inputFormat
        job.setInputFormatClass(WholeFileInputformat.class);
        // 设置输出的 outputFormat
        job.setOutputFormatClass(SequenceFileOutputFormat.class);

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

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

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
    }
}
```

```
boolean result = job.waitForCompletion(true);

System.exit(result ? 0 : 1);

}

}
```

3.3 MapTask 工作机制

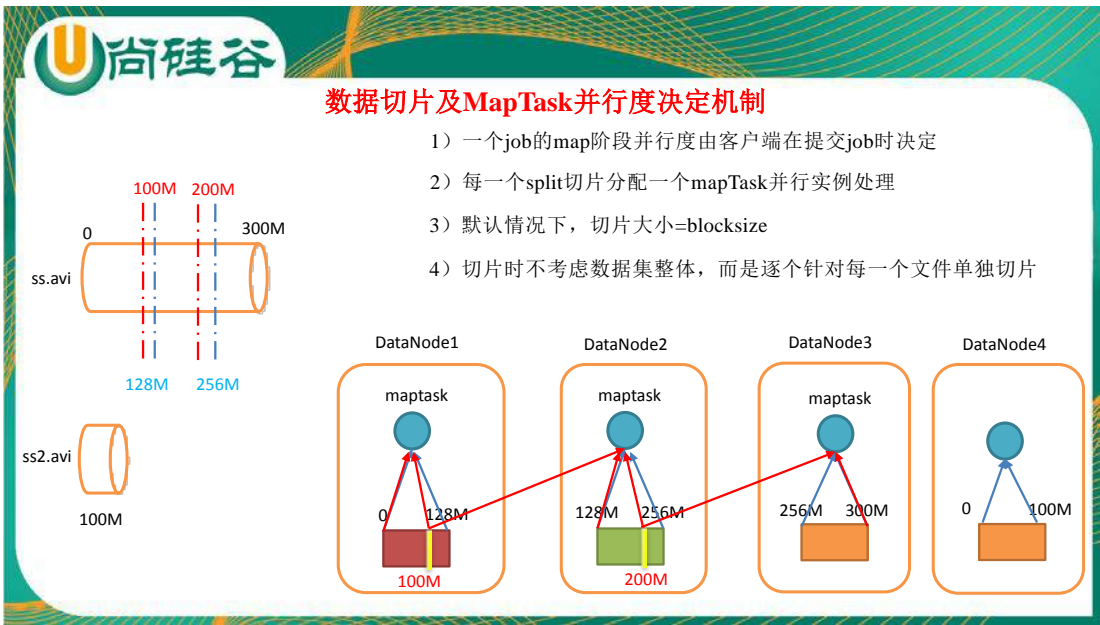
3.3.1 并行度决定机制

1) 问题引出

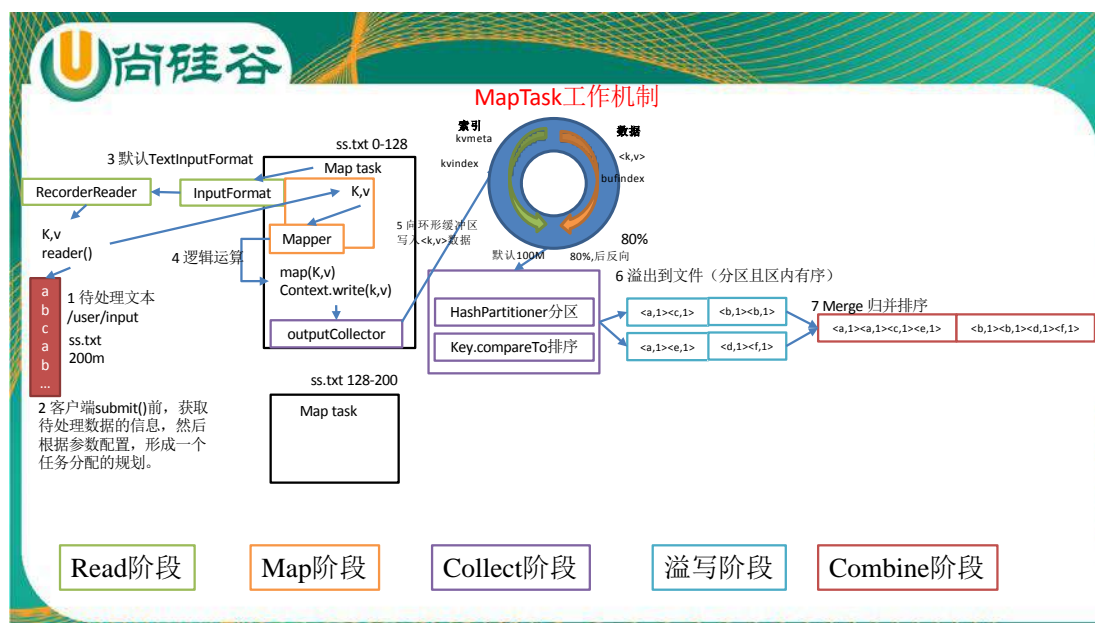
maptask 的并行度决定 map 阶段的任务处理并发度，进而影响到整个 job 的处理速度。
那么，mapTask 并行任务是否越多越好呢？

2) MapTask 并行度决定机制

一个 job 的 map 阶段 MapTask 并行度（个数），由客户端提交 job 时的切片个数决定。



3.3.2 MapTask 工作机制



(1) Read 阶段：Map Task 通过用户编写的 `RecordReader`，从输入 `InputSplit` 中解析出一个一个 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。

(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，且同一分区内所有数据按照 `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`。

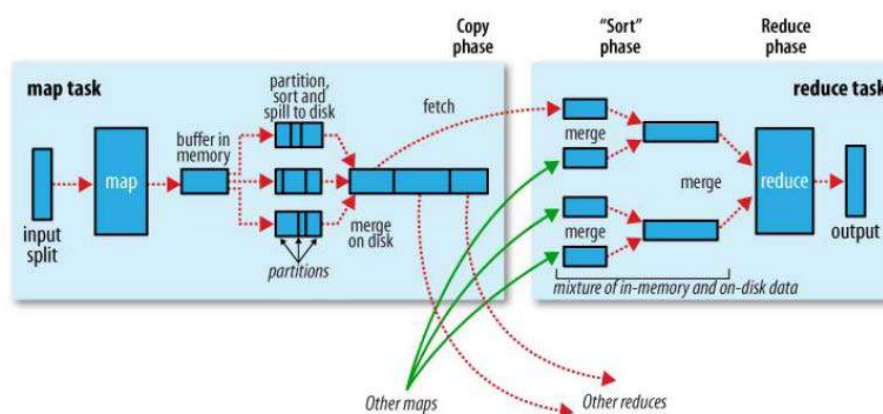
在进行文件合并过程中，MapTask 以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式。每轮合并 `io.sort.factor`（默认 100）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

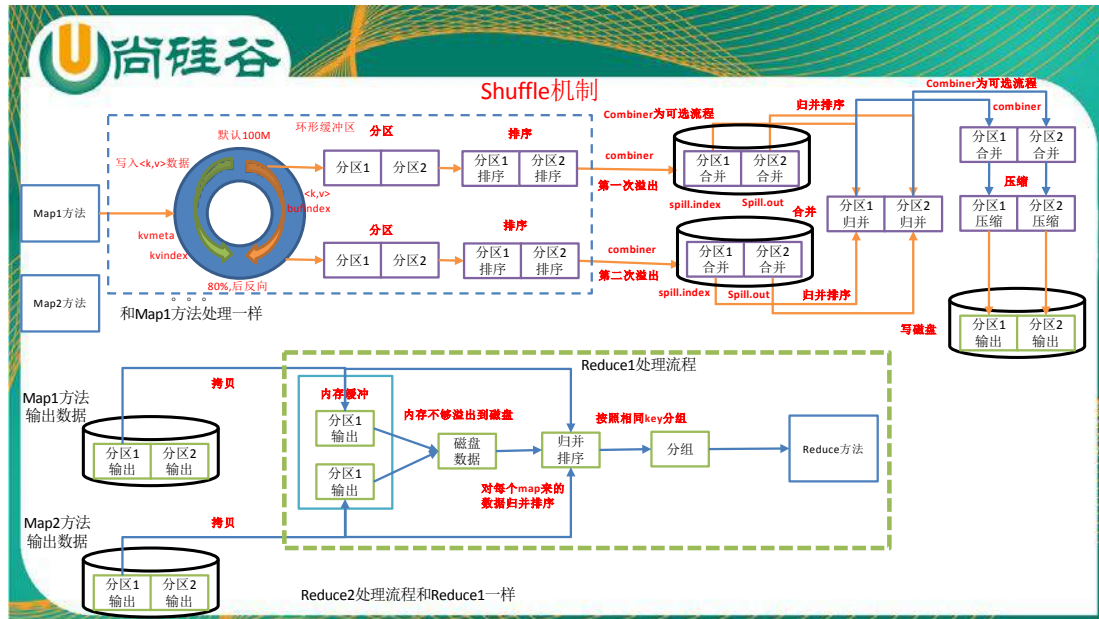
让每个 MapTask 最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

3.4 Shuffle 机制

3.4.1 Shuffle 机制

Mapreduce 确保每个 reducer 的输入都是按键排序的。系统执行排序的过程（即将 map 输出作为输入传给 reducer）称为 shuffle。





3.4.2 Partition 分区

0) 问题引出：要求将统计结果按照条件输出到不同文件中（分区）。比如：将统计结果按照手机归属地不同省份输出到不同文件中（分区）

1) 默认 partition 分区

```
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;
    }
}
```

默认分区是根据 key 的 hashCode 对 reduceTasks 个数取模得到的。用户没法控制哪个 key 存储到哪个分区。

2) 自定义 Partitioner 步骤

(1) 自定义类继承 Partitioner，重写 getPartition()方法

```
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;
}
}

```

(2) 在 job 驱动中，设置自定义 partitioner:

```
job.setPartitionerClass(CustomPartitioner.class);
```

(3) 自定义 partition 后，要根据自定义 partitioner 的逻辑设置相应数量的 reduce task

```
job.setNumReduceTasks(5);
```

3) 注意:

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;

例如：假设自定义分区数为 5，则

(1) job.setNumReduceTasks(1);会正常运行，只不过会产生一个输出文件

(2) job.setNumReduceTasks(2);会报错

(3) job.setNumReduceTasks(6);大于 5，程序会正常运行，会产生空文件

3.4.3 Partition 分区案例实操

1) 需求：将统计结果按照手机归属地不同省份输出到不同文件中（分区）

2) 数据准备



phone_data.txt

3) 分析

(1) Mapreduce 中会将 map 输出的 kv 对，按照相同 key 分组，然后分发给不同的

reducetask。默认的分发规则为：根据 key 的 hashCode%reducetask 数来分发

(2) 如果要按照我们自己的需求进行分组，则需要改写数据分发(分组)组件 Partitioner
自定义一个 CustomPartitioner 继承抽象类：Partitioner

(3) 在 job 驱动中，设置自定义 partitioner：job.setPartitionerClass(CustomPartitioner.class)

4) 在案例 2.4 的基础上，增加一个分区类

```
package com.atguigu.mapreduce.flowsum;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

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;
    }
}
```

2) 在驱动函数中增加自定义数据分区设置和 reduce task 设置

```
package com.atguigu.mapreduce.flowsum;
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 FlowsunDriver {

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

        // 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(FlowsunDriver.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);

        // 8 指定自定义数据分区
        job.setPartitionerClass(ProvincePartitioner.class);
        // 9 同时指定相应数量的 reduce task
        job.setNumReduceTasks(5);

        // 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);
    }
}

```

3.4.4 WritableComparable 排序

排序是 MapReduce 框架中最重要的操作之一。Map Task 和 Reduce Task 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。**默认排序是按照字典顺序排序，且实现该排序的方法是快速排序。**

对于 Map Task，它会将处理的结果暂时放到一个缓冲区中，当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排序，并将这些有序数据写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行一次合并，以将这些文件合并成一个大的有序文件。

对于 Reduce Task，它从每个 Map Task 上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放到内存中。如果磁盘上文件数目达到一定阈值，则进行一次合并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写到磁盘上。当所有数据拷贝完毕后，Reduce Task 统一对内存和磁盘上的所有数据进行一次合并。

每个阶段的默认排序

1) 排序的分类:

(1) 部分排序:

MapReduce 根据输入记录的键对数据集排序。保证输出的每个文件内部排序。

(2) 全排序:

如何用 Hadoop 产生一个全局排序的文件？最简单的方法是使用一个分区。但该方法在处理大型文件时效率极低，因为一台机器必须处理所有输出文件，从而完全丧失了 MapReduce 所提供的并行架构。

替代方案：首先创建一系列排好序的文件；其次，串联这些文件；最后，生成一个全局排序的文件。主要思路是使用一个分区来描述输出的全局排序。例如：可以为上述文件创建 3 个分区，在第一分区中，记录的单词首字母 a-g，第二分区记录单词首字母 h-n，第三分区记录单词首字母 o-z。

(3) 辅助排序：(GroupingComparator 分组)

Mapreduce 框架在记录到达 reducer 之前按键对记录排序，但键所对应的值并没有被排序。甚至在不同的执行轮次中，这些值的排序也不固定，因为它们来自不同的 map 任务且这些 map 任务在不同轮次中完成时间各不相同。一般来说，大多数 MapReduce 程序会避免让 reduce 函数依赖于值的排序。但是，有时也需要通过特定的方法对键进行排序和分组等以实现值的排序。

(4) 二次排序:

在自定义排序过程中，如果 compareTo 中的判断条件为两个即为二次排序。

2) 自定义排序 WritableComparable

(1) 原理分析

bean 对象实现 **WritableComparable** 接口重写 **compareTo** 方法，就可以实现排序

```
@Override
public int compareTo(FlowBean o) {
    // 倒序排列，从大到小
    return this.sumFlow > o.getSumFlow() ? -1 : 1;
}
```

3.4.5 WritableComparable 排序案例实操

案例一

1) 需求

根据案例 2.4 产生的结果再次对总流量进行排序。

2) 数据准备



phone_data.txt

3) 分析

(1) 把程序分两步走，第一步正常统计总流量，第二步再把结果进行排序

(2) context.write(总流量, 手机号)

(3) FlowBean 实现 WritableComparable 接口重写 compareTo 方法

```
@Override
public int compareTo(FlowBean o) {
    // 倒序排列，从大到小
    return this.sumFlow > o.getSumFlow() ? -1 : 1;
}
```

4) 代码实现

(1) FlowBean 对象在在需求 1 基础上增加了比较功能

```
package com.atguigu.mapreduce.sort;
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 o) {
    // 倒序排列，从大到小
    return this.sumFlow > o.getSumFlow() ? -1 : 1;
}
}

```

(2) 编写 mapper

```

package com.atguigu.mapreduce.sort;
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);
}
}

```

(3) 编写 reducer

```

package com.atguigu.mapreduce.sort;
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);
        }
    }
}

```

(4) 编写 driver

```
package com.atguigu.mapreduce.sort;
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 {

        // 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(FlowCountSortDriver.class);

        // 2 指定本业务 job 要使用的 mapper/Reducer 业务类
        job.setMapperClass(FlowCountSortMapper.class);
        job.setReducerClass(FlowCountSortReducer.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);
    }
}
```


案例二

1) 需求

要求每个省份手机号输出的文件中按照总流量内部排序。

2) 分析:

基于前一个需求，增加自定义分区类即可。

3) 案例实操

(1) 增加自定义分区类

```
package com.atguigu.mapreduce.sort;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<FlowBean, Text> {

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

        // 1 获取手机号码前三位
        String preNum = value.toString().substring(0, 3);

        int partition = 4;

        // 2 根据手机号归属地设置分区
        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;
    }
}
```

(2) 在驱动类中添加分区类

```
// 加载自定义分区类

job.setPartitionerClass(FlowSortPartitioner.class);
```

```
// 设置 Reducetask 个数  
job.setNumReduceTasks(5);
```

3.4.6 Combiner 合并

1) combiner 是 MR 程序中 Mapper 和 Reducer 之外的一种组件。

2) combiner 组件的父类就是 Reducer。

3) combiner 和 reducer 的区别在于运行的位置：

Combiner 是在每一个 maptask 所在的节点运行；

Reducer 是接收全局所有 Mapper 的输出结果；

4) combiner 的意义就是对每一个 maptask 的输出进行局部汇总，以减小网络传输量。

5) combiner 能够应用的前提是不能影响最终的业务逻辑，而且，combiner 的输出 kv 应该跟 reducer 的输入 kv 类型要对应起来。

Mapper

3 5 7 ->(3+5+7)/3=5

2 6 ->(2+6)/2=4

Reducer

(3+5+7+2+6)/5=23/5 不等于 (5+4)/2=9/2

6) 自定义 Combiner 实现步骤：

(1) 自定义一个 combiner 继承 Reducer，重写 reduce 方法


```
public class WordcountCombiner extends Reducer<Text, IntWritable, Text, IntWritable>{  
    @Override  
    protected void reduce(Text key, Iterable<IntWritable> values,  
        Context context) throws IOException, InterruptedException {  
        // 1 汇总操作  
        int count = 0;  
        for(IntWritable v : values){  
            count = v.get();  
        }  
        // 2 写出  
        context.write(key, new IntWritable(count));  
    }  
}
```

(2) 在 job 驱动类中设置：

```
job.setCombinerClass(WordcountCombiner.class);
```

3.4.7 Combiner 合并案例实操

1) 需求：统计过程中对每一个 maptask 的输出进行局部汇总，以减小网络传输量即采用 Combiner 功能。



3.1.3 需求3：对每一个maptask的输出局部汇总（Combiner）

方案一	方案二
1) 增加一个WordcountCombiner类继承Reducer	1) 将WordcountReducer作为combiner在WordcountDriver驱动类中指定
2) 在WordcountCombiner中	<code>job.setCombinerClass(WordcountReducer.class);</code>
(1) 统计单词汇总	
(2) 将统计结果输出	

2) 数据准备：



hello.txt

方案一

1) 增加一个 WordcountCombiner 类继承 Reducer

```
package com.atguigu.mr.combiner;
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>{

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
        // 1 汇总
        int count = 0;
        for(IntWritable v : values){
            count += v.get();
        }
        // 2 写出
```

```
        context.write(key, new IntWritable(count));
    }
}
```

2) 在 WordcountDriver 驱动类中指定 combiner

```
// 9 指定需要使用 combiner，以及用哪个类作为 combiner 的逻辑
job.setCombinerClass(WordcountCombiner.class);
```

方案二

1) 将 WordcountReducer 作为 combiner 在 WordcountDriver 驱动类中指定

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

运行程序

```
Map-Reduce Framework
  Map input records=5
  Map output records=8
  Map output bytes=72
  Map output materialized bytes=94
  Input split bytes=199
  Combine input records=0
  Combine output records=0
  Reduce input groups=6
  Reduce shuffle bytes=94
```

未使用前

```
Map-Reduce Framework
  Map input records=5
  Map output records=8
  Map output bytes=72
  Map output materialized bytes=72
  Input split bytes=199
  Combine input records=8
  Combine output records=6
  Reduce input groups=6
  Reduce shuffle bytes=72
```

使用后

3.4.8 GroupingComparator 分组（辅助排序）

对 reduce 阶段的数据根据某一个或几个字段进行分组。

3.4.9 GroupingComparator 分组案例实操

1) 需求

有如下订单数据

订单 id	商品 id	成交金额
0000001	Pdt_01	222.8
0000001	Pdt_06	25.8
0000002	Pdt_03	522.8
0000002	Pdt_04	122.4
0000002	Pdt_05	722.4

0000003	Pdt_01	222.8
0000003	Pdt_02	33.8

现在要求出每一个订单中最贵的商品。

2) 输入数据



GroupingComparator.txt

输出数据预期:



part-r-00000.txt



part-r-00001.txt



part-r-00002.txt

3) 分析

(1) 利用“订单 id 和成交金额”作为 key，可以将 map 阶段读取到的所有订单数据按照 id 分区，按照金额排序，发送到 reduce。

(2) 在 reduce 端利用 groupingcomparator 将订单 id 相同的 kv 聚合成组，然后取第一个即是最大值。

求每个订单中最贵的商品 (GroupingComparator)

输入数据

0000001	Pdt_01	222.8
0000002	Pdt_06	722.4
0000001	Pdt_05	25.8
0000003	Pdt_01	222.8
0000003	Pdt_01	33.8
0000002	Pdt_03	522.8
0000002	Pdt_04	122.4

maptask

排序分区

bean1, nullwritable	0000001	222.8
bean3, nullwritable	0000001	25.8
bean2, nullwritable	0000002	722.4
bean6, nullwritable	0000002	522.8
bean7, nullwritable	0000002	122.4
bean4, nullwritable	0000003	122.4
bean5, nullwritable	0000003	33.8

reduce方法只把一组key的第一个写出去

0000001	222.8
0000002	722.4
0000002	522.8
0000002	122.4
0000003	122.4
0000003	33.8

reducetask

预期输出数据

0000001	222.8
0000002	722.4
0000003	222.8

1) map中处理的事情

- (1) 获取一行
- (2) 切割出每个字段
- (3) 一行封装成bean对象

bean1, nullwritable 0000001 222.8
bean2, nullwritable 0000002 722.4
bean3, nullwritable 0000001 25.8
bean4, nullwritable 0000003 222.8
bean5, nullwritable 0000003 33.8
bean6, nullwritable 0000002 522.8
bean7, nullwritable 0000002 122.4

2) 排序分区

3) reduce方法只把一组key的第一个写出去

4) 代码实现

(1) 定义订单信息 OrderBean

```
package com.atguigu.mapreduce.order;
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;
}
}

```

(2) 编写 OrderSortMapper

```

package com.atguigu.mapreduce.order;
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());
    }
}

```

(3) 编写 OrderSortPartitioner

```

package com.atguigu.mapreduce.order;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Partitioner;

public class OrderPartitioner extends Partitioner<OrderBean, NullWritable> {

    @Override
    public int getPartition(OrderBean key, NullWritable value, int numReduceTasks) {

        return (key.getOrder_id() & Integer.MAX_VALUE) % numReduceTasks;
    }
}

```

(4) 编写 OrderSortGroupingComparator

```

package com.atguigu.mapreduce.order;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class OrderGroupingComparator extends WritableComparator {

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

    @SuppressWarnings("rawtypes")
    @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;
    }
}

```

(5) 编写 OrderSortReducer

```

package com.atguigu.mapreduce.order;
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());
    }
}

```

(6) 编写 OrderSortDriver

```

package com.atguigu.mapreduce.order;
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 {

        // 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]));

// 10 设置 reduce 端的分组
job.setGroupingComparatorClass(OrderGroupingComparator.class);

// 7 设置分区
job.setPartitionerClass(OrderPartitioner.class);

// 8 设置 reduce 个数
job.setNumReduceTasks(3);

// 9 提交
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}

```

3.5 ReduceTask 工作机制

1) 设置 ReduceTask 并行度（个数）

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);

```

2) 注意

- (1) reducetask=0，表示没有 reduce 阶段，输出文件个数和 map 个数一致。
- (2) reducetask 默认值就是 1，所以输出文件个数为一个。

(3) 如果数据分布不均匀，就有可能在 reduce 阶段产生数据倾斜

(4) `reducetask` 数量并不是任意设置，还要考虑业务逻辑需求，有些情况下，需要计算全局汇总结果，就只能是 1 个 `reducetask`。

(5) 具体多少个 `reducetask`，需要根据集群性能而定。

(6) 如果分区数不是 1，但是 `reducetask` 为 1，是否执行分区过程。答案是：不执行分区过程。因为在 `maptask` 的源码中，执行分区的前提是先判断 `reduceNum` 个数是否大于 1。不大于 1 肯定不执行。

3) 实验：测试 `reducetask` 多少合适。

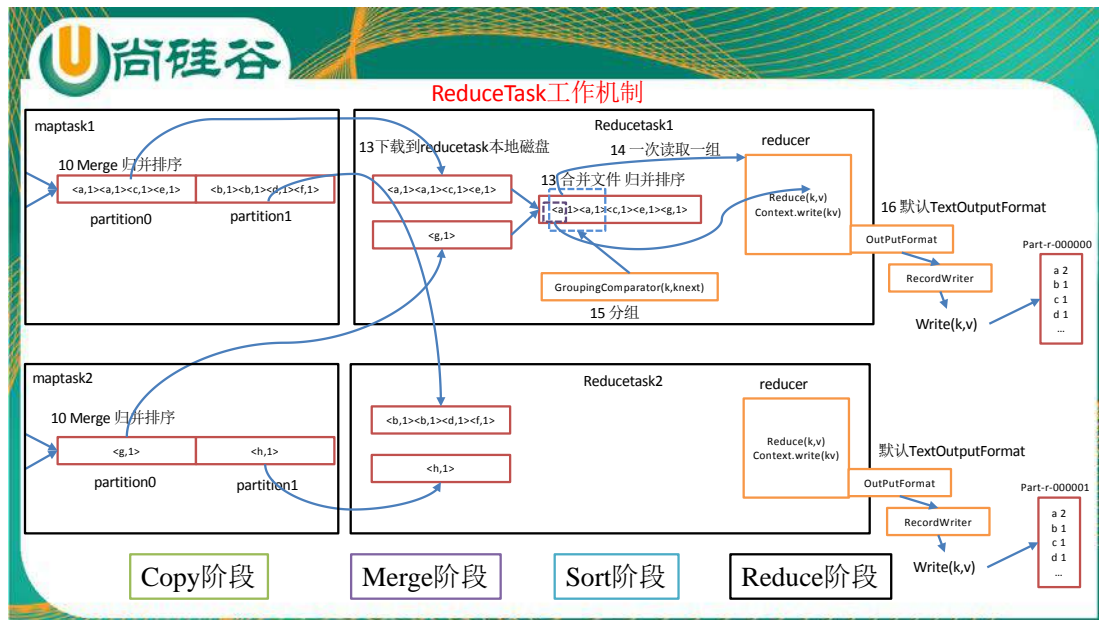
(1) 实验环境：1 个 master 节点，16 个 slave 节点：CPU:8GHZ，内存: 2G

(2) 实验结论：

表 1 改变 reduce task （数据量为 1GB）

Map task =16										
Reduce task	1	5	10	15	16	20	25	30	45	60
总时间	892	146	110	92	88	100	128	101	145	104

4) ReduceTask 工作机制



(1) Copy 阶段：ReduceTask 从各个 MapTask 上远程拷贝一片数据，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。

(4) Reduce 阶段：`reduce()` 函数将计算结果写到 HDFS 上。

3.6 OutputFormat 数据输出

3.6.1 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。下面我们介绍几种常见的 OutputFormat 实现类。

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

SequenceFileOutputFormat 将它的输出写为一个顺序文件。如果输出需要作为后续 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

根据用户需求，自定义实现输出。

3.6.2 自定义 OutputFormat

为了实现控制最终文件的输出路径，可以自定义 OutputFormat。

要在一个 mapreduce 程序中根据数据的不同输出两类结果到不同目录，这类灵活的输出需求可以通过自定义 outputformat 来实现。

1) 自定义 OutputFormat 步骤

(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()`。

3.6.3 自定义 OutputFormat 案例实操

1) 需求

过滤输入的 log 日志中是否包含 atguigu

(1) 包含 atguigu 的网站输出到 `e:/atguigu.log`

(2) 不包含 atguigu 的网站输出到 `e:/other.log`

2) 输入数据



log.txt

输出预期:



atguigu.log



other.log

3) 具体程序:

(1) 自定义一个 outputformat

```
package com.atguigu.mapreduce.outputformat;
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);
    }
}
```

(2) 具体的写数据 RecordWriter

```
package com.atguigu.mapreduce.outputformat;
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;
```

```

FSDDataOutputStream 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
    if (atguiguOut != null) {
        atguiguOut.close();
    }

    if (otherOut != null) {
        otherOut.close();
    }
}

```

```
}  
}
```

(3) 编写 FilterMapper

```
package com.atguigu.mapreduce.outputformat;  
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>{  
  
    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();  
  
        k.set(line);  
  
        // 3 写出  
        context.write(k, NullWritable.get());  
    }  
}
```

(4) 编写 FilterReducer

```
package com.atguigu.mapreduce.outputformat;  
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> {  
  
    @Override  
    protected void reduce(Text key, Iterable<NullWritable> values, Context context)  
        throws IOException, InterruptedException {  
  
        String k = key.toString();  
        k = k + "\r\n";  
  
        context.write(new Text(k), NullWritable.get());  
    }  
}
```

```
}  
}
```

(5) 编写 FilterDriver

```
package com.atguigu.mapreduce.outputformat;  
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 {  
  
        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);  
    }  
}
```



```
}  
}
```

3.7 Join 多种应用

3.7.1 Reduce join

1) 原理:

Map 端的主要工作: 为来自不同表(文件)的 key/value 对打标签以区别不同来源的记录。
然后用连接字段作为 key, 其余部分和新加的标志作为 value, 最后进行输出。

Reduce 端的主要工作: 在 reduce 端以连接字段作为 key 的分组已经完成, 我们只需要在每一个分组当中将那些来源于不同文件的记录(在 map 阶段已经打标志)分开, 最后进行合并就 ok 了。

2) 该方法的缺点

这种方式的缺点很明显就是会造成 map 和 reduce 端也就是 **shuffle 阶段出现大量的数据传输, 效率很低。**

3.7.2 Reduce join 案例实操

1) 需求:

订单数据表 t_order:

id	pid	amount
1001	01	1
1002	02	2
1003	03	3



order.txt

商品信息表 t_product

pid	pname
01	小米
02	华为
03	格力



pd.txt

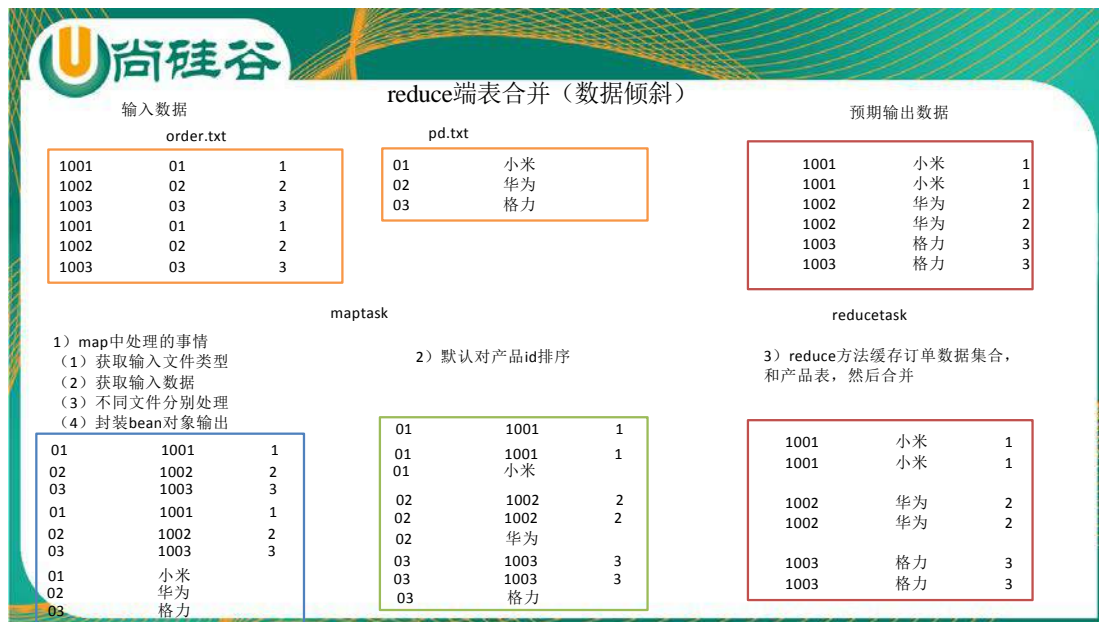
将商品信息表中数据根据商品 pid 合并到订单数据表中。

最终数据形式:

id	pname	amount
----	-------	--------

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

通过将关联条件作为 map 输出的 key, 将两表满足 join 条件的数据并携带数据所来源的文件信息, 发往同一个 reduce task, 在 reduce 中进行数据的串联。



1) 创建商品和订合并后的 bean 类

```
package com.atguigu.mapreduce.table;
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 setName(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" ;
    }
}

```

2) 编写 TableMapper 程序

```

package com.atguigu.mapreduce.table;
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>{
    TableBean bean = new TableBean();
    Text k = new Text();

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

        // 1 获取输入文件类型
    }
}

```

```

FileSplit split = (FileSplit) context.getInputSplit();
String name = split.getPath().getName();

// 2 获取输入数据
String line = value.toString();

// 3 不同文件分别处理
if (name.startsWith("order")) { // 订单表处理
    // 3.1 切割
    String[] fields = line.split("\t");

    // 3.2 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("0");

    k.set(fields[1]);
} else { // 产品表处理
    // 3.3 切割
    String[] fields = line.split("\t");

    // 3.4 封装 bean 对象
    bean.setP_id(fields[0]);
    bean.setPname(fields[1]);
    bean.setFlag("1");
    bean.setAmount(0);
    bean.setOrder_id("");

    k.set(fields[0]);
}
// 4 写出
context.write(k, bean);
}
}

```

3) 编写 TableReducer 程序

```

package com.atguigu.mapreduce.table;
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 ("0".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());
        }
    }
}

```

4) 编写 TableDriver 程序

```

package com.atguigu.mapreduce.table;
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 {
        // 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);
    }
}

```

3) 运行程序查看结果

1001	小米	1
1001	小米	1

1002	华为	2
1002	华为	2
1003	格力	3
1003	格力	3

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：map 端实现数据合并

3.7.3 Map join

1) 使用场景：一张表十分小、一张表很大。

2) 解决方案

在 map 端缓存多张表，提前处理业务逻辑，这样增加 map 端业务，减少 reduce 端数据的压力，尽可能的减少数据倾斜。

3) 具体办法：采用 distributedcache

(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。

job.addCacheFile(new URI("file:/e:/mapjoincache/pd.txt")); // 缓存普通文件到 task 运行节点



map端表合并（Distributedcache）

1) DistributedCacheDriver 缓存文件

```
// 1 加载缓存数据
job.addCacheFile(new URI("file:///e:/cache/pd.txt"));

// 2 map端join的逻辑不需要reduce阶段，设置reducetask数量为0
job.setNumReduceTasks(0);
```

2) 读取缓存的文件数据

setup()方法中	map方法中
// 1 获取缓存的文件	// 1 获取一行
// 2 循环读取缓存文件一行	// 2 截取
// 3 切割	// 3 获取订单id
// 4 缓存数据到集合	// 4 获取商品名称
// 5 关流	// 5 拼接
	// 6 写出

3.7.4 Map join 案例实操

1) 分析

适用于关联表中有小表的情形；

可以将小表分发到所有的 map 节点，这样，map 节点就可以在本地对自己所读到的大表数据进行合并并输出最终结果，可以大大提高合并操作的并发度，加快处理速度。

2) 实现代码:

(1) 先在驱动模块中添加缓存文件

```
package test;
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 {
        // 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:/tableinput/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);
    }
}

```

(2) 读取缓存的文件数据

```

package test;
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
        BufferedReader reader = new BufferedReader(new InputStreamReader(new
FileInputStream("pd.txt"), "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());
}
}

```

3.8 计数器应用

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。

1) API

- (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", "countera").increment(1);

```

组名和计数器名称随便起，但最好有意义。

- (3) 计数结果在程序运行后的控制台上查看。

3.9 数据清洗（ETL）

1) 概述

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。

3.10 数据清洗案例实操

3.10.1 简单解析版

1) 需求：

去除日志中字段长度小于等于 11 的日志。

2) 输入数据



web.log

3) 实现代码：

（1）编写 LogMapper

```
package com.atguigu.mapreduce.weblog;
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;
    }
}
}

```

(2) 编写 LogDriver

```

package com.atguigu.mapreduce.weblog;
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 {

        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);
}
}

```

3.10.2 复杂解析版

1) 需求:

对 web 访问日志中的各字段识别切分

去除日志中不合法的记录

根据统计需求, 生成各类访问请求过滤数据

2) 输入数据



web.log

3) 实现代码:

(1) 定义一个 bean, 用来记录日志数据中的各数据字段

```

package com.atguigu.mapreduce.log;

public class LogBean {
    private String remote_addr; // 记录客户端的 ip 地址
    private String remote_user; // 记录客户端用户名, 忽略属性 "-"
    private String time_local; // 记录访问时间与时区
}

```

```
private String request;// 记录请求的 url 与 http 协议
private String status;// 记录请求状态; 成功是 200
private String body_bytes_sent;// 记录发送给客户端文件主体内容大小
private String http_referer;// 用来记录从那个页面链接访问过来的
private String http_user_agent;// 记录客户浏览器的相关信息

private boolean valid = true;// 判断数据是否合法

public String getRemote_addr() {
    return remote_addr;
}

public void setRemote_addr(String remote_addr) {
    this.remote_addr = remote_addr;
}

public String getRemote_user() {
    return remote_user;
}

public void setRemote_user(String remote_user) {
    this.remote_user = remote_user;
}

public String getTime_local() {
    return time_local;
}

public void setTime_local(String time_local) {
    this.time_local = time_local;
}

public String getRequest() {
    return request;
}

public void setRequest(String request) {
    this.request = request;
}

public String getStatus() {
    return status;
}
```

```
public void setStatus(String status) {
    this.status = status;
}

public String getBody_bytes_sent() {
    return body_bytes_sent;
}

public void setBody_bytes_sent(String body_bytes_sent) {
    this.body_bytes_sent = body_bytes_sent;
}

public String getHttp_referer() {
    return http_referer;
}

public void setHttp_referer(String http_referer) {
    this.http_referer = http_referer;
}

public String getHttp_user_agent() {
    return http_user_agent;
}

public void setHttp_user_agent(String http_user_agent) {
    this.http_user_agent = http_user_agent;
}

public boolean isValid() {
    return valid;
}

public void setValid(boolean valid) {
    this.valid = valid;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(this.valid);
    sb.append("\001").append(this.remote_addr);
    sb.append("\001").append(this.remote_user);
    sb.append("\001").append(this.time_local);
    sb.append("\001").append(this.request);
}
```



```

        sb.append("\001").append(this.status);
        sb.append("\001").append(this.body_bytes_sent);
        sb.append("\001").append(this.http_referer);
        sb.append("\001").append(this.http_user_agent);

        return sb.toString();
    }
}

```

(2) 编写 LogMapper 程序

```

package com.atguigu.mapreduce.log;
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
        String line = value.toString();

        // 2 解析日志是否合法
        LogBean bean = pressLog(line);

        if (!bean.isValid()) {
            return;
        }

        k.set(bean.toString());

        // 3 输出
        context.write(k, NullWritable.get());
    }

    // 解析日志
    private LogBean pressLog(String line) {
        LogBean logBean = new LogBean();

        // 1 截取
        String[] fields = line.split(" ");
    }
}

```

```

        if (fields.length > 11) {
            // 2 封装数据
            logBean.setRemote_addr(fields[0]);
            logBean.setRemote_user(fields[1]);
            logBean.setTime_local(fields[3].substring(1));
            logBean.setRequest(fields[6]);
            logBean.setStatus(fields[8]);
            logBean.setBody_bytes_sent(fields[9]);
            logBean.setHttp_referer(fields[10]);

            if (fields.length > 12) {
                logBean.setHttp_user_agent(fields[11] + " " + fields[12]);
            } else {
                logBean.setHttp_user_agent(fields[11]);
            }

            // 大于 400, HTTP 错误
            if (Integer.parseInt(logBean.getStatus()) >= 400) {
                logBean.setValid(false);
            }
        } else {
            logBean.setValid(false);
        }

        return logBean;
    }
}

```

(3) 编写 LogDriver 程序

```

package com.atguigu.mapreduce.log;
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 {
        // 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);

// 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);
}
```

3.11 MapReduce 开发总结

在编写 mapreduce 程序时，需要考虑的几个方面：

1) 输入数据接口：InputFormat

默认使用的实现类是：TextInputFormat

TextInputFormat 的功能逻辑是：一次读一行文本，然后将该行的起始偏移量作为 key，行内容作为 value 返回。

KeyValueTextInputFormat 每一行均为一条记录，被分隔符分割为 key，value。默认分隔符是 tab (\t)。

NLineInputFormat 按照指定的行数 N 来划分切片。

CombineTextInputFormat 可以把多个小文件合并成一个切片处理，提高处理效率。

用户还可以自定义 InputFormat。

2) 逻辑处理接口：Mapper

用户根据业务需求实现其中三个方法：map() setup() cleanup()

3) Partitioner 分区

有默认实现 HashPartitioner，逻辑是根据 key 的哈希值和 numReduces 来返回一个分区号；key.hashCode() & Integer.MAXVALUE % numReduces

如果业务上有特别的需求，可以自定义分区。

4) Comparable 排序

当我们用自定义的对象作为 key 来输出时，就必须要实现 WritableComparable 接口，重写其中的 compareTo()方法。

部分排序：对最终输出的每一个文件进行内部排序。

全排序：对所有数据进行排序，通常只有一个 Reduce。

二次排序：排序的条件有两个。

5) Combiner 合并

Combiner 合并可以提高程序执行效率，减少 io 传输。但是使用时必须不能影响原有的业务处理结果。

6) reduce 端分组：GroupingComparator

reduceTask 拿到输入数据（一个 partition 的所有数据）后，首先需要对数据进行分组，其分组的默认原则是 key 相同，然后对每一组 kv 数据调用一次 reduce()方法，并且将这一组 kv 中的第一个 kv 的 key 作为参数传给 reduce 的 key，将这一组数据的 value 的迭代器传给 reduce()的 values 参数。

利用上述这个机制，我们可以实现一个高效的分组取最大值的逻辑。

自定义一个 bean 对象用来封装我们的数据，然后改写其 compareTo 方法产生倒序排序的效果。然后自定义一个 GroupingComparator，将 bean 对象的分组逻辑改成按照我们的业务分组 id 来分组（比如订单号）。这样，我们要取的最大值就是 reduce()方法中传进来 key。

7) 逻辑处理接口：Reducer

用户根据业务需求实现其中三个方法：reduce() setup() cleanup ()

8) 输出数据接口：OutputFormat

默认实现类是 TextOutputFormat，功能逻辑是：将每一个 KV 对向目标文本文件中输出为一行。

SequenceFileOutputFormat 将它的输出写为一个顺序文件。如果输出需要作为后续 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。用户还可以自定义 OutputFormat。

四 Hadoop 数据压缩

4.1 概述

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 Hadoop 下，尤其是数据规模很大和工作负载密集的情况下，使用数据压缩显得非常重要。在这种情况下，I/O 操作和网络数据传输要花大量的时间。还有，Shuffle 与 Merge 过程同样也面临着巨大的 I/O 压力。

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。不过，尽管压缩与解压操作的 CPU 开销不高，其性能的提升和资源的节省并非没有代价。

如果磁盘 I/O 和网络带宽影响了 MapReduce 作业性能，在任意 MapReduce 阶段启用压缩都可以改善端到端处理时间并减少 I/O 和网络流量。

压缩 Mapreduce 的一种优化策略：通过压缩编码对 Mapper 或者 Reducer 的输出进行压缩，以减少磁盘 IO，提高 MR 程序运行速度（但相应增加了 cpu 运算负担）。

注意：压缩特性运用得当能提高性能，但运用不当也可能降低性能。

基本原则：

- （1）运算密集型的 job，少用压缩
- （2）IO 密集型的 job，多用压缩

4.2 MR 支持的压缩编码

压缩格式	hadoop 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
DEFAULT	是，直接使用	DEFAULT	.deflate	否	和文本处理一样，不需要修改
Gzip	是，直接使用	DEFAULT	.gz	否	和文本处理一样，不需要修改
bzip2	是，直接使用	bzip2	.bz2	是	和文本处理一样，不需要修改
LZO	否，需要安装	LZO	.lzo	是	需要建索引，还需要指定输入格式
Snappy	否，需要安装	Snappy	.snappy	否	和文本处理一样，不需要修改

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

压缩性能的比较

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

<http://google.github.io/snappy/>

On a single core of a Core i7 processor in 64-bit mode, Snappy **compresses** at about **250 MB/sec** or more and **decompresses** at about **500 MB/sec** or more.

4.3 压缩方式选择

4.3.1 Gzip 压缩

优点：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。

缺点：不支持 split。

应用场景：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。例如说一天或者一个小时的日志压缩成一个 gzip 文件，运行 mapreduce 程序的时候通过多个 gzip 文件达到并发。hive 程序，streaming 程序，和 java 写的 mapreduce 程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

4.3.2 Bzip2 压缩

优点：支持 split；具有很高的压缩率，比 gzip 压缩率都高；hadoop 本身支持，但不支持 native；在 linux 系统下自带 bzip2 命令，使用方便。

缺点：压缩/解压速度慢；不支持 native。

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，可以作为 mapreduce 作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持 split，而且兼容之前的应用程序（即应用程序不需要修改）的情况。

4.3.3 Lzo 压缩

优点：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 lzop 命令，使用方便。

缺点：压缩率比 `gzip` 要低一些；`hadoop` 本身不支持，需要安装；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）。

应用场景：一个很大的文本文件，压缩之后还大于 `200M` 以上的可以考虑，而且单个文件越大，`lzo` 优点越越明显。

4.3.4 Snappy 压缩

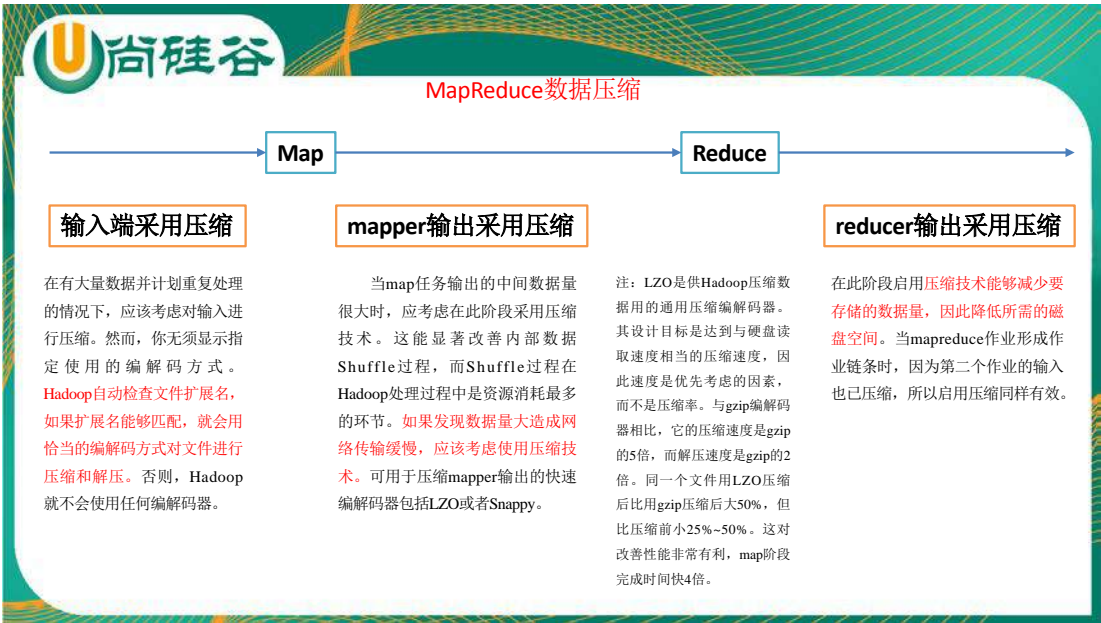
优点：高速压缩速度和合理的压缩率。

缺点：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。

4.4 压缩位置选择

压缩可以在 `MapReduce` 作用的任意阶段启用。



4.5 压缩参数配置

要在 `Hadoop` 中启用压缩，可以配置如下参数：

参数	默认值	阶段	建议
<code>io.compression.codecs</code> (在 <code>core-site.xml</code> 中配置)	<code>org.apache.hadoop.io.compress.DefaultCodec,</code> <code>org.apache.hadoop.io.compress.GzipCodec,</code>	输入压缩	Hadoop 使用文件扩展名判

	org.apache.hadoop.io.compress.BZip2Codec		断是否支持某种编解码器
mapreduce.map.output.compress(在 mapred-site.xml 中配置)	false	mapper 输出	这个参数设为 true 启用压缩
mapreduce.map.output.compress.codec (在 mapred-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec	mapper 输出	使用 LZO 或 snappy 编解码器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress (在 mapred-site.xml 中配置)	false	reducer 输出	这个参数设为 true 启用压缩
mapreduce.output.fileoutputformat.compress.codec(在 mapred-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec	reducer 输出	使用标准工具或者编解码器, 如 gzip 和 bzip2
mapreduce.output.fileoutputformat.compress.type (在 mapred-site.xml 中配置)	RECORD	reducer 输出	SequenceFile 输出使用的压缩类型 : NONE 和 BLOCK

4.6 压缩实操案例

4.6.1 数据流的压缩和解压缩

CompressionCodec 有两个方法可以用于轻松地压缩或解压缩数据。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(OutputStream out)` 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。

测试一下如下压缩方式：

DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec

```
package com.atguigu.mapreduce.compress;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.CompressionCodecFactory;
import org.apache.hadoop.io.compress.CompressionInputStream;
import org.apache.hadoop.io.compress.CompressionOutputStream;
import org.apache.hadoop.util.ReflectionUtils;

public class TestCompress {

    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
    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
    fis.close();
    cos.close();
    fos.close();
}

// 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();
}
}

```

4.6.2 Map 输出端采用压缩

即使你的 MapReduce 的输入输出文件都是未压缩的文件，你仍然可以对 map 任务的中

间结果输出做压缩，因为它要写在硬盘并且通过网络传输到 `reduce` 节点，对其压缩可以提高很多性能，这些工作只要设置两个属性即可，我们来看下代码怎么设置：

1) 给大家提供的 `hadoop` 源码支持的压缩格式有：`BZip2Codec`、`DefaultCodec`

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
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.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {

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

        Configuration configuration = new Configuration();

        // 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);

        Job job = Job.getInstance(configuration);

        job.setJarByClass(WordCountDriver.class);

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.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]));

        boolean result = job.waitForCompletion(true);

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

```

2) Mapper 保持不变

```

package com.atguigu.mapreduce.compress;
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;

public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    @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 循环写出
        for(String word:words){
            context.write(new Text(word), new IntWritable(1));
        }
    }
}

```

3) Reducer 保持不变

```

package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

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

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

```

```
        int count = 0;
        // 1 汇总
        for(IntWritable value:values){
            count += value.get();
        }

        // 2 输出
        context.write(key, new IntWritable(count));
    }
}
```

4.6.3 Reduce 输出端采用压缩

基于 workcount 案例处理

1) 修改驱动

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
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.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.DefaultCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.io.compress.Lz4Codec;
import org.apache.hadoop.io.compress.SnappyCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {

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

        Configuration configuration = new Configuration();

        Job job = Job.getInstance(configuration);

        job.setJarByClass(WordCountDriver.class);

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.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]));

// 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);
// FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
// FileOutputFormat.setOutputCompressorClass(job, DefaultCodec.class);

boolean result = job.waitForCompletion(true);

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

2) Mapper 和 Reducer 保持不变（详见 4.6.2）

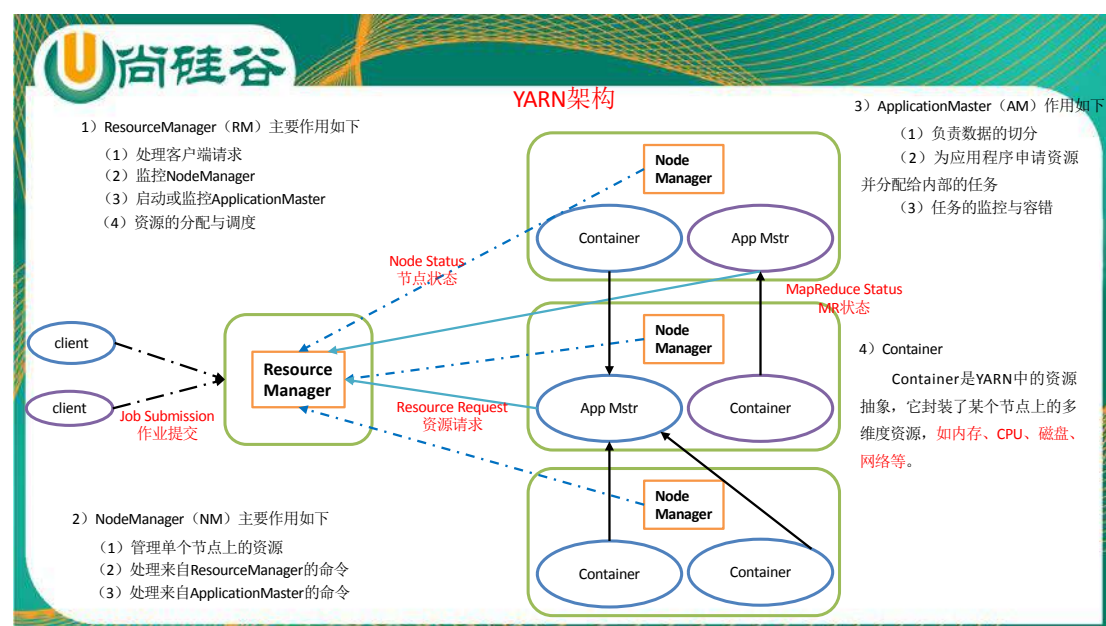
五 Yarn

5.1 Yarn 概述

Yarn 是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而 **MapReduce** 等运算程序则相当于运行于操作系统之上的应用程序。

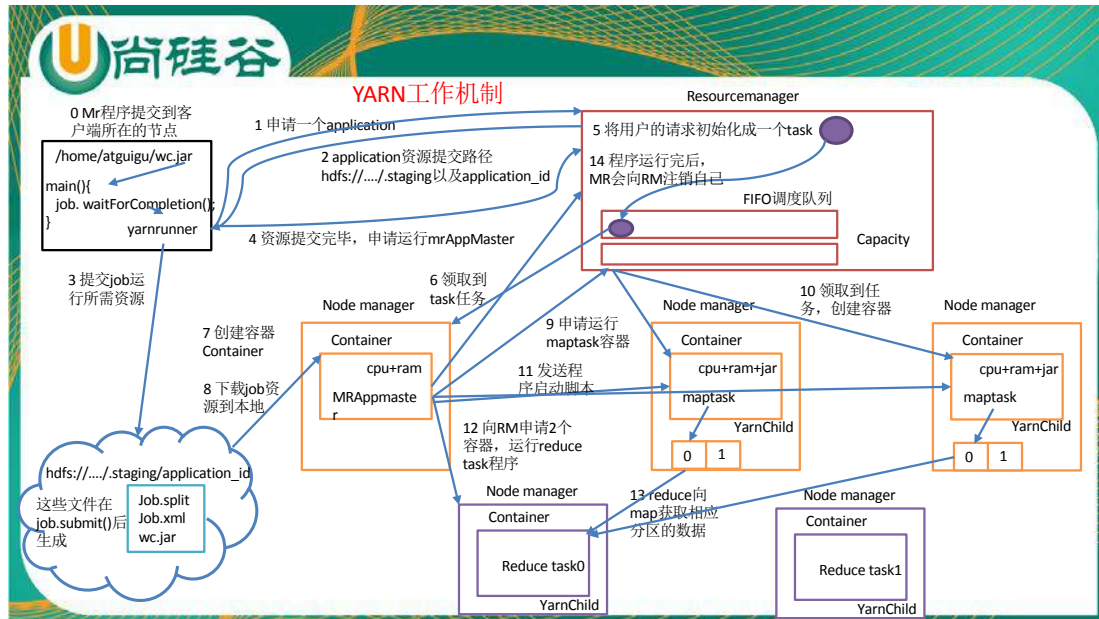
5.2 Yarn 基本架构

YARN 主要由 ResourceManager、NodeManager、ApplicationMaster 和 Container 等组件构成。



5.3 Yarn 工作机制

1) Yarn 运行机制

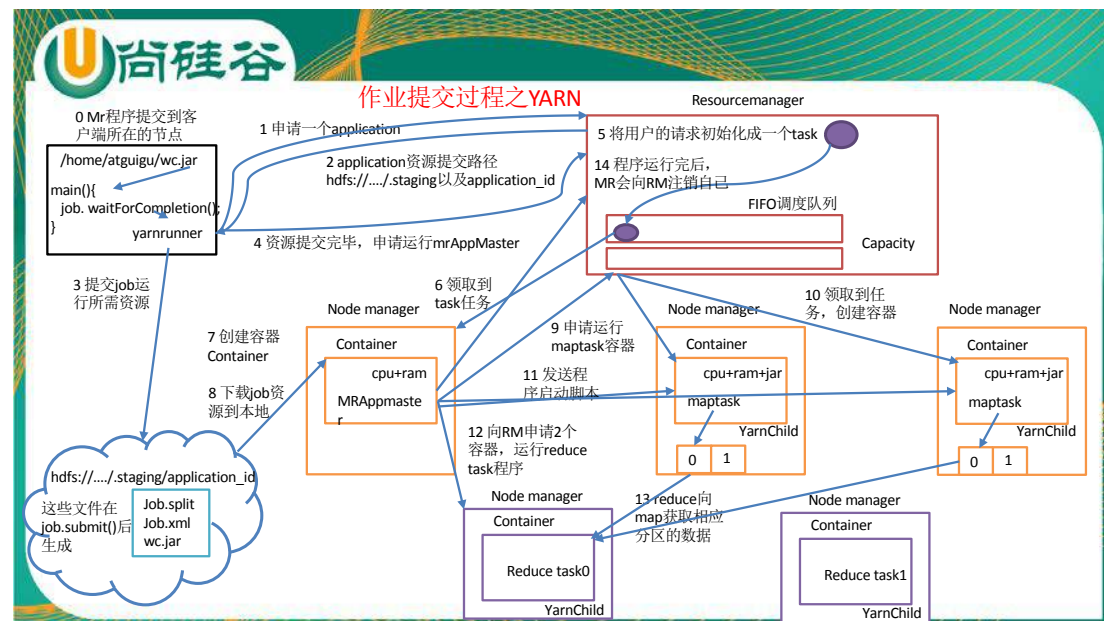


2) 工作机制详解

- (0) Mr 程序提交到客户端所在的节点。
- (1) Yarnrunner 向 Resourcemanager 申请一个 Application。
- (2) rm 将该应用程序的资源路径返回给 yarnrunner。
- (3) 该程序将运行所需资源提交到 HDFS 上。
- (4) 程序资源提交完毕后，申请运行 `mrAppMaster`。
- (5) RM 将用户的请求初始化成一个 task。
- (6) 其中一个 NodeManager 领取到 task 任务。
- (7) 该 NodeManager 创建容器 `Container`，并产生 `MRAppmaster`。
- (8) `Container` 从 HDFS 上拷贝资源到本地。
- (9) `MRAppmaster` 向 RM 申请运行 `maptask` 资源。
- (10) 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。
- (11) 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。
- (12) `MRAppMaster` 等待所有 `maptask` 运行完毕后，向 RM 申请容器，运行 `reduce task`。
- (13) `reduce task` 向 `maptask` 获取相应分区的数据。
- (14) 程序运行完毕后，MR 会向 RM 申请注销自己。

5.4 作业提交全过程

1) 作业提交过程之 YARN



作业提交全过程详解

(1) 作业提交

第 0 步: client 调用 `job.waitForCompletion` 方法, 向整个集群提交 MapReduce 作业。

第 1 步: client 向 RM 申请一个作业 id。

第 2 步: RM 给 client 返回该 job 资源的提交路径和作业 id。

第 3 步: client 提交 jar 包、切片信息和配置文件到指定的资源提交路径。

第 4 步: client 提交完资源后, 向 RM 申请运行 MrAppMaster。

(2) 作业初始化

第 5 步: 当 RM 收到 client 的请求后, 将该 job 添加到容量调度器中。

第 6 步: 某一个空闲的 NM 领取到该 job。

第 7 步: 该 NM 创建 Container, 并产生 MRAppmaster。

第 8 步: 下载 client 提交的资源到本地。

(3) 任务分配

第 9 步: MrAppMaster 向 RM 申请运行多个 maptask 任务资源。

第 10 步: 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。

(4) 任务运行

第 11 步: 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。

第 12 步: MrAppMaster 等待所有 maptask 运行完毕后, 向 RM 申请容器, 运行 reduce task。

第 13 步: reduce task 向 maptask 获取相应分区的数据。

第 14 步: 程序运行完毕后, MR 会向 RM 申请注销自己。

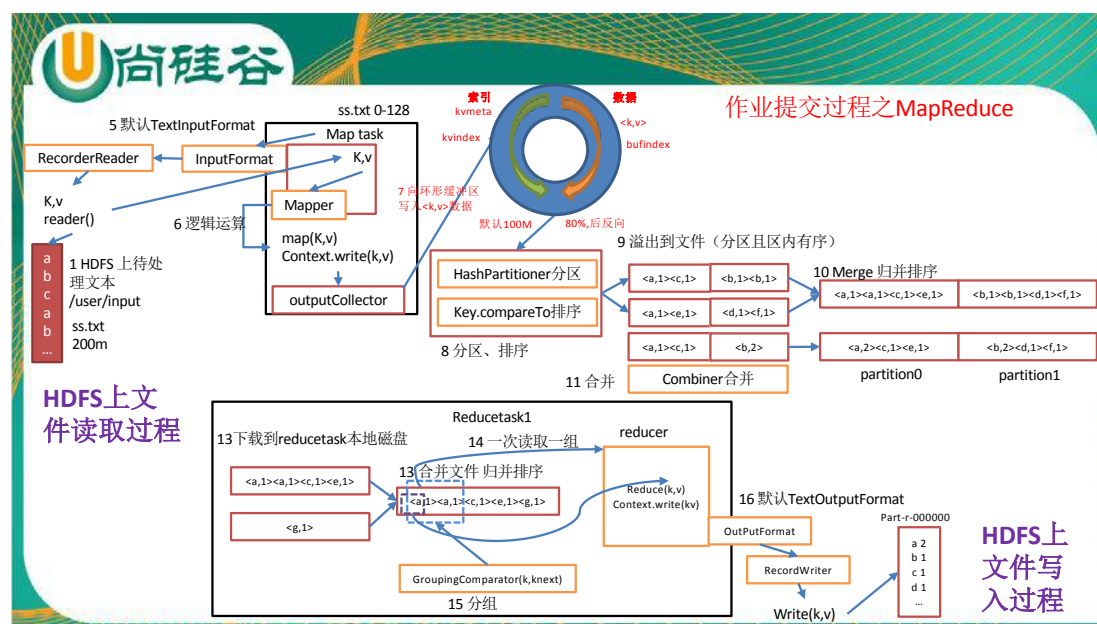
(5) 进度和状态更新

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。

(6) 作业完成

除了向应用管理器请求作业进度外, 客户端每 5 分钟都会通过调用 `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。

2) 作业提交过程之 MapReduce



5.5 资源调度器

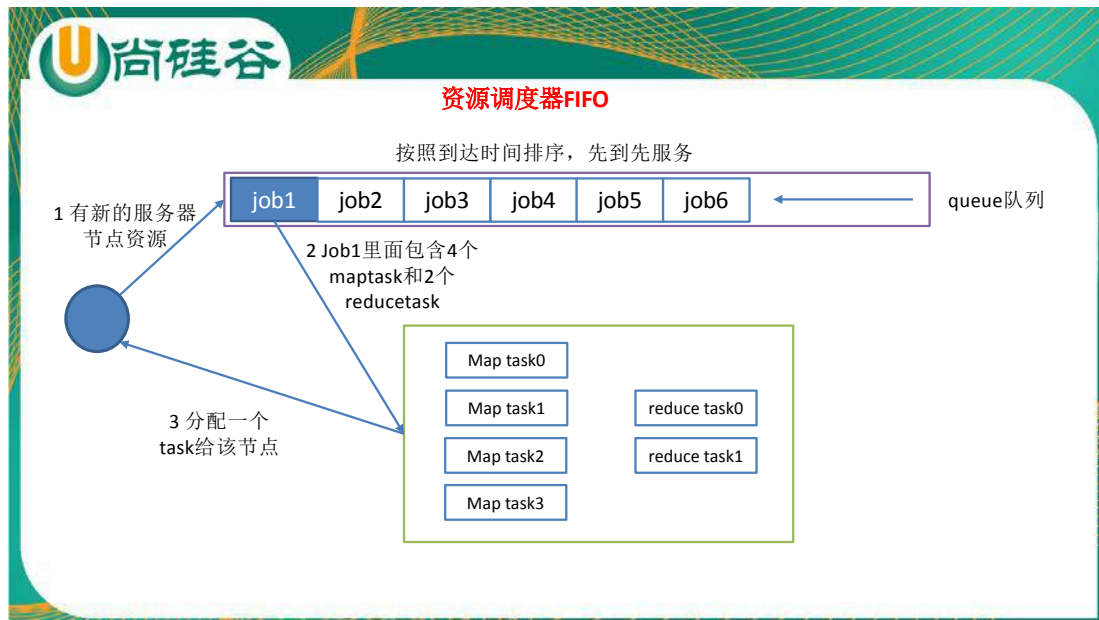
目前, Hadoop 作业调度器主要有三种: FIFO、Capacity Scheduler 和 Fair Scheduler。Hadoop2.7.2 默认的资源调度器是 Capacity Scheduler。

具体设置详见: yarn-default.xml 文件

```
<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>
```

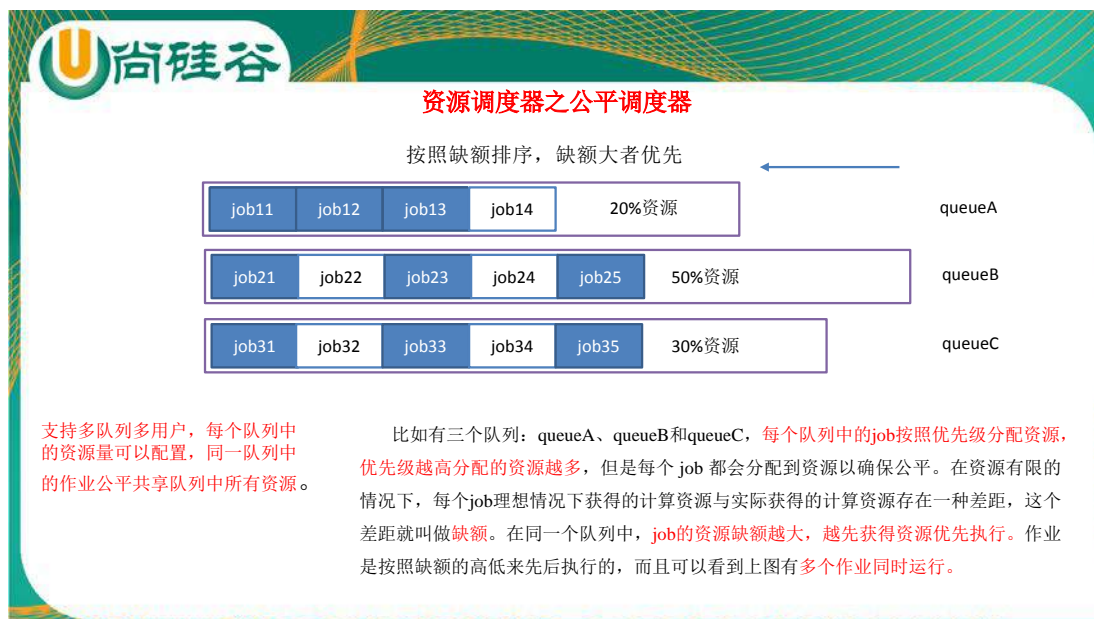
1) 先进先出调度器 (FIFO)



2) 容量调度器 (Capacity Scheduler)



3) 公平调度器 (Fair Scheduler)



5.6 任务的推测执行

1) 作业完成时间取决于最慢的任务完成时间

一个作业由若干个 Map 任务和 Reduce 任务构成。因硬件老化、软件 Bug 等，某些任务可能运行非常慢。

典型案例：系统中有 99% 的 Map 任务都完成了，只有少数几个 Map 老是进度很慢，完不成，怎么办？

2) 推测执行机制：

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。

3) 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%)
- (3) 开启推测执行参数设置。Hadoop2.7.2 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>
```

4) 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;
- (2) 特殊任务, 比如任务向数据库中写数据。

5) 算法原理



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:

$$\text{estimateEndTime} = \text{estimatedRunTime} + \text{taskStartTime}$$
$$\text{推测执行完时刻 } 60 = \text{推测运行时间 (60s)} + \text{任务启动时刻 (0)}$$
$$\text{estimatedRunTime} = (\text{currentTimestamp} - \text{taskStartTime}) / \text{progress}$$
$$\text{推测运行时间 (60s)} = (\text{当前时刻 (6)} - \text{任务启动时刻 (0)}) / \text{任务运行比例 (10\%)}$$
$$\text{estimateEndTime`} = \text{currentTimestamp} + \text{averageRunTime}$$
$$\text{备份任务推测完成时刻 (16)} = \text{当前时刻 (6)} + \text{运行完成任务的平均时间 (10s)}$$

- 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。

六 Hadoop 企业优化

6.1 MapReduce 跑的慢的原因

Mapreduce 程序效率的瓶颈在于两点：

1) 计算机性能

CPU、内存、磁盘健康、网络

2) I/O 操作优化

- (1) 数据倾斜
- (2) map 和 reduce 数设置不合理
- (3) map 运行时间太长，导致 reduce 等待过久
- (4) 小文件过多
- (5) 大量的不可分块的超大文件
- (6) spill 次数过多
- (7) merge 次数过多等。

6.2 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。

6.2.1 数据输入

(1) 合并小文件：在执行 mr 任务前将小文件进行合并，大量的小文件会产生大量的 map 任务，增大 map 任务装载次数，而任务的装载比较耗时，从而导致 mr 运行较慢。

(2) 采用 CombineTextInputFormat 来作为输入，解决输入端大量小文件场景。

6.2.2 Map 阶段

1) 减少溢写 (spill) 次数：通过调整 io.sort.mb 及 sort.spill.percent 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。

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。

6.2.3 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, 中间多个一个写磁盘->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: `mapred.job.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。

6.2.4 I/O 传输

1) 采用数据压缩的方式, 减少网络 IO 的时间。安装 Snappy 和 LZ4 压缩编码器。

2) 使用 `SequenceFile` 二进制文件。

6.2.5 数据倾斜问题

1) 数据倾斜现象

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

数据大小倾斜——部分记录的大小远远大于平均值。

2) 如何收集倾斜数据

在 reduce 方法中加入记录 map 输出键的详细情况的功能。

```
public static final String MAX_VALUES = "skew.maxvalues";
private int maxThreshold;

@Override
public void configure(JobConf job) {
    maxThreshold = job.getInt(MAX_VALUES, 100);
}

@Override
public void reduce(Text key, Iterator<Text> values,
```



```
OutputCollector<Text, Text> output,
Reporter reporter) throws IOException {
    int i = 0;
    while (values.hasNext()) {
        values.next();
        i++;
    }

    if (++i > maxValThreshold) {
        log.info("Received " + i + " values for key " + key);
    }
}
```

3) 减少数据倾斜的方法

方法 1: 抽样和范围分区

可以通过对原始数据进行抽样得到的结果集来预设分区边界值。

方法 2: 自定义分区

基于输出键的背景知识进行自定义分区。例如，如果 map 输出键的单词来源于一本书。

且其中某几个专业词汇较多。那么就可以自定义分区将这这些专业词汇发送给固定的一部分 reduce 实例。而将其他的都发送给剩余的 reduce 实例。

方法 3: Combine

使用 Combine 可以大量地减小数据倾斜。在可能的情况下，combine 的目的就是聚合并精简数据。

方法 4: 采用 Map Join，尽量避免 Reduce Join。

6.2.6 常用的调优参数

1) 资源相关参数

(1) 以下参数是在用户自己的 mr 应用程序中配置就可以生效 (mapred-default.xml)

配置参数	参数说明
mapreduce.map.memory.mb	一个 Map Task 可使用的资源上限 (单位:MB)，默认为 1024。如果 Map Task 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 Reduce Task 可使用的资源上限 (单位:MB)，默认为 1024。如果 Reduce Task 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 Map task 可使用的最多 cpu core 数目，默认值: 1

mapreduce.reduce.cpu.vcores	每个 Reduce task 可使用的最多 cpu core 数目，默认值: 1
mapreduce.reduce.shuffle.parallelcopies	每个 reduce 去 map 中拿数据的并行数。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
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

(2) 应该在 yarn 启动之前就配置在服务器的配置文件中才能生效 (yarn-default.xml)

配置参数	参数说明
yarn.scheduler.minimum-allocation-mb 1024	给应用程序 container 分配的最小内存
yarn.scheduler.maximum-allocation-mb 8192	给应用程序 container 分配的最大内存
yarn.scheduler.minimum-allocation-vcores 1	每个 container 申请的最小 CPU 核数
yarn.scheduler.maximum-allocation-vcores 32	每个 container 申请的最大 CPU 核数
yarn.nodemanager.resource.memory-mb 8192	给 containers 分配的最大物理内存

(3) shuffle 性能优化的关键参数，应在 yarn 启动之前就配置好 (mapred-default.xml)

配置参数	参数说明
mapreduce.task.io.sort.mb 100	shuffle 的环形缓冲区大小，默认 100m
mapreduce.map.sort.spill.percent 0.8	环形缓冲区溢出的阈值，默认 80%

2) 容错相关参数(mapreduce 性能优化)

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，也许永远会卡主，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.”。
--	---

6.3 HDFS 小文件优化方法

6.3.1 HDFS 小文件弊端

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时, 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是索引速度变慢。

6.3.2 解决方案

1) Hadoop Archive:

是一个高效地将小文件放入 HDFS 块中的文件存档工具, 它能够将多个小文件打包成一个 HAR 文件, 这样就减少了 namenode 的内存使用。

2) Sequence file:

sequence file 由一系列的二进制 key/value 组成, 如果 key 为文件名, value 为文件内容, 则可以将大批小文件合并成一个大文件。

3) CombineFileInputFormat:

CombineFileInputFormat 是一种新的 inputformat, 用于将多个文件合并成一个单独的 split, 另外, 它会考虑数据的存储位置。

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。

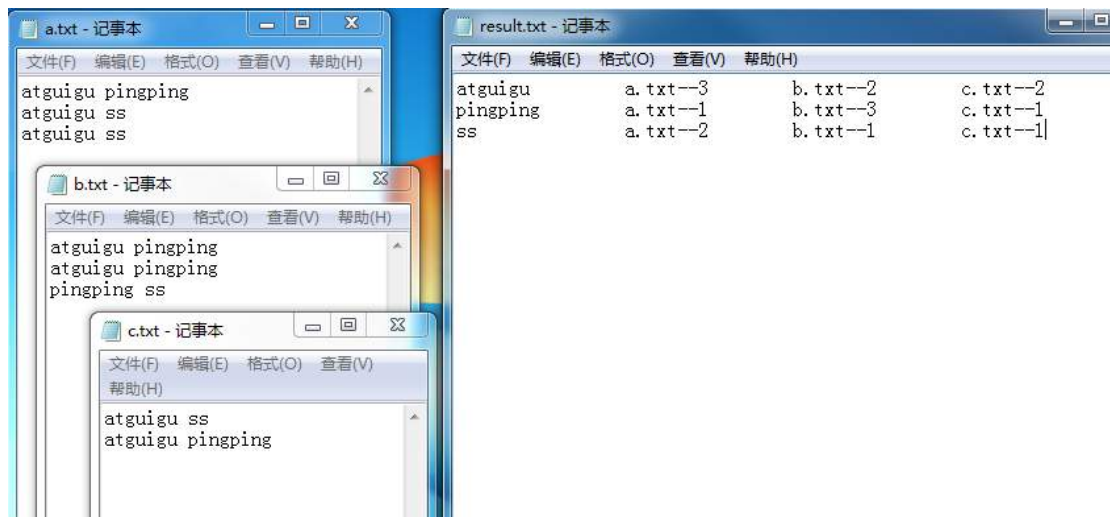
JVM 重用理解: 一个 map 运行一个 jvm, 重用的话, 在一个 map 在 jvm 上运行完毕后, jvm 继续运行其他 map。

具体设置: mapreduce.job.jvm.numtasks 值在 10-20 之间。

第 7 章 MapReduce 扩展案例

7.1 倒排索引案例（多 job 串联）

1) 需求：有大量的文本（文档、网页），需要建立搜索索引



(1) 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
```

(2) 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
```

2) 第一次处理

(1) 第一次处理，编写 OneIndexMapper

```
package com.atguigu.mapreduce.index;
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);
        }
    }
}

```

(2) 第一次处理，编写 OneIndexReducer

```

package com.atguigu.mapreduce.index;
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>{

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

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

        // 2 写出
        context.write(key, new IntWritable(count));
    }
}

```

(3) 第一次处理，编写 OneIndexDriver

```

package com.atguigu.mapreduce.index;
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 {

        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);
    }
}

```

(4) 查看第一次输出结果

```

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

```

3) 第二次处理

(1) 第二次处理，编写 TwoIndexMapper

```

package com.atguigu.mapreduce.index;
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);
    }
}
```

(2) 第二次处理，编写 TwoIndexReducer

```
package com.atguigu.mapreduce.index;
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> {

    @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");
        }
        // 2 写出
        context.write(key, new Text(sb.toString()));
    }
}
```

(3) 第二次处理，编写 TwoIndexDriver

```
package com.atguigu.mapreduce.index;
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 {

        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);

FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

```

(4) 第二次查看最终结果

```

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

```

7.2 找博客共同好友案例

1) 需求:

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)



friends.txt

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?

2) 需求分析:

先求出 A、B、C、....等是谁的好友

第一次输出结果

```

A  I,K,C,B,G,F,H,O,D,
B  A,F,J,E,
C  A,E,B,H,F,G,K,

```


D	G,C,K,A,L,F,E,H,
E	G,M,L,H,A,F,B,D,
F	L,M,D,C,G,A,
G	M,
H	O,
I	O,C,
J	O,
K	B,
L	D,E,
M	E,F,
O	A,H,I,J,F,

第二次输出结果

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

3) 代码实现:

(1) 第一次 Mapper

```
package com.atguigu.mapreduce.friends;
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));
        }
    }
}
```

(2) 第一次 Reducer

```
package com.atguigu.mapreduce.friends;
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()));
}
}
```

(3) 第一次 Driver

```
package com.atguigu.mapreduce.friends;
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);
    }
}

```

(4) 第二次 Mapper

```

package com.atguigu.mapreduce.friends;
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));
            }
        }
    }
}

```

(5) 第二次 Reducer

```

package com.atguigu.mapreduce.friends;
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()));
    }
}

```

(6) 第二次 Driver

```

package com.atguigu.mapreduce.friends;
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);
}
}

```

7.3 Top10 案例

- 1) 需求：对需求 2.4 输出结果进行加工，输出流量使用量在前 10 的用户信息
- 2) 实现代码

(1) 编写 JavaBean 类

```

package com.atguigu.mr;

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 sumFlow; // 总流量
    private String phoneNum; // 手机号

    // 空参构造
    public FlowBean() {
        super();
    }

    public Long getSumFlow() {
        return sumFlow;
    }

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

    public String getPhoneNum() {
        return phoneNum;
    }

    public void setPhoneNum(String phoneNum) {
        this.phoneNum = phoneNum;
    }

    @Override

```

```

        public String toString() {
            return sumFlow + "\t" + phoneNum;
        }

        @Override
        public void write(DataOutput out) throws IOException {
            // 序列化
            out.writeLong(sumFlow);
            out.writeUTF(phoneNum);
        }

        @Override
        public void readFields(DataInput in) throws IOException {
            // 反序列化
            this.sumFlow = in.readLong();
            this.phoneNum = in.readUTF();
        }

        @Override
        public int compareTo(FlowBean o) {
            int result;

            result = this.sumFlow.compareTo(o.sumFlow);
            if (result == 0) {
                result = this.phoneNum.compareTo(o.phoneNum);
            }
            return result;
        }
    }
}

```

(2) 编写 TopTenMapper 类

```

package com.atguigu.mr;

import java.io.IOException;
import java.util.TreeMap;

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 TopTenMapper extends Mapper<LongWritable, Text, NullWritable, 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>();

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

```



```

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

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

        long sumFlow = Long.parseLong(fields[3]);

        bean.setSumFlow(sumFlow);
        bean.setPhoneNum(fields[0]);

        // 3.向 TreeMap 中添加数据
        flowMap.put(bean, new Text(value));

        // 4.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());
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
        // 输出
        for (Text t : flowMap.values()) {
            context.write(NullWritable.get(), t);
        }
    }
}

```

(3) 编写 TopTenReducer 类

```

package com.atguigu.mr;

import java.io.IOException;
import java.util.TreeMap;

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

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

    @Override

```

```

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

    // 1.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>();

    for (Text value : values) {
        FlowBean bean = new FlowBean();
        bean.setPhoneNum(value.toString().split("\t")[0]);
        bean.setSumFlow(Long.parseLong(value.toString().split("\t")[3]));
        flowMap.put(bean, new Text(value));

        // 2.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());
        }
    }

    // 3.输出
    for (Text t : flowMap.descendingMap().values()) {
        context.write(NullWritable.get(), t);
    }
}

```

（4）编写驱动类

```

package com.atguigu.mr;

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

        // 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(TopTenDriver.class);

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

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

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

八 常见错误及解决方案

1) 导包容易出错。尤其 Text 和 CombineTextInputFormat。

2) Mapper 中第一个输入的参数必须是 LongWritable 或者 NullWritable, 不可以是 IntWritable。
报的错误是类型转换异常。

3) java.lang.Exception: java.io.IOException: Illegal partition for 13926435656 (4), 说明 partition 和 reducetask 个数没对上, 调整 reducetask 个数。

4) 如果分区数不是 1, 但是 reducetask 为 1, 是否执行分区过程。答案是: 不执行分区过程。
因为在 maptask 的源码中, 执行分区的前提是先判断 reduceNum 个数是否大于 1。不大于 1 肯定不执行。

5) 在 Windows 环境编译的 jar 包导入到 linux 环境中运行,

```
hadoop jar wc.jar com.atguigu.mapreduce.wordcount.WordCountDriver /user/atguigu/  
/user/atguigu/output
```

报如下错误:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError:  
com.atguigu.mapreduce.wordcount.WordCountDriver : Unsupported major.minor version 52.0
```

原因是 Windows 环境用的 jdk1.7, linux 环境用的 jdk1.8。

解决方案: 统一 jdk 版本。

6) 缓存 pd.txt 小文件案例中, 报找不到 pd.txt 文件

原因: 大部分为路径书写错误。还有就是要检查 pd.txt.txt 的问题。还有个别电脑写相对路径找不到 pd.txt, 可以修改为绝对路径。

7) 报类型转换异常。

通常都是在驱动函数中设置 map 输出和最终输出时编写错误。

Map 输出的 key 如果没有排序, 也会报类型转换异常。

8) 集群中运行 wc.jar 时出现了无法获得输入文件。

原因: wordcount 案例的输入文件不能放用 hdfs 集群的根目录。

9) 出现了如下相关异常

```
Exception in thread "main" java.lang.UnsatisfiedLinkError:  
org.apache.hadoop.io.nativeio.NativeIO$Windows.access0(Ljava/lang/String;I)Z
```

```
at org.apache.hadoop.io.nativeio.NativeIO$Windows.access0(Native Method)
```

```
at org.apache.hadoop.io.nativeio.NativeIO$Windows.access(NativeIO.java:609)
```

```
at org.apache.hadoop.fs.FileUtil.canRead(FileUtil.java:977)
```

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

```
at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:356)
```

```
at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:371)
```

```
at org.apache.hadoop.util.Shell.<clinit>(Shell.java:364)
```

解决方案：拷贝 `hadoop.dll` 文件到 windows 目录 `C:\Windows\System32`。个别同学电脑还需要修改 `hadoop` 源码。

10) 自定义 `outputformat` 时，注意在 `recordWirter` 中的 `close` 方法必须关闭流资源。否则输出的文件内容中数据为空。

```
@Override
public void close(TaskAttemptContext context) throws IOException,
InterruptedException {
    if (atguigufos != null) {
        atguigufos.close();
    }
    if (otherfos != null) {
        otherfos.close();
    }
}
```