

尚硅谷大数据技术之大数据概论

(作者：尚硅谷大数据研发部)

版本：V3.3

第1章 大数据概念

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。

大数据主要解决，海量数据的采集、存储和分析计算问题。

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

$$\begin{aligned}1\text{Byte} &= 8\text{bit} & 1\text{K} &= 1024\text{Byte} & 1\text{MB} &= 1024\text{K} \\1\text{G} &= 1024\text{M} & 1\text{T} &= 1024\text{G} & 1\text{P} &= 1024\text{T}\end{aligned}$$



第2章 大数据特点（4V）

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。



 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2025年，全球数据使用量将达到163ZB。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
2020年96秒，天猫交易额超过100亿**



让天下没有难学的技术

 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	20200710-9:10:10	面膜	2
1002	xiaozelaoshi	20200710-9:11:20	化妆品	3
1003	boduolaoshi	20200710-9:22:50	内衣	4
1004	sslaoshi	20200710-10:12:20	海狗人参丸	100

 网络日志

让天下没有难学的技术

 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。



太

第 3 章 大数据应用场景

 大数据应用场景

1、抖音：推荐的都是你喜欢的视频

我抖音里面的视频



ss抖音里面的视频



让天下没有难学的技术

2、电商站内广告推荐：给用户推荐可能喜欢的商品

The screenshot shows a user interface for an e-commerce website. At the top, there is a small photo of two people and the text "我选了一种药，又推荐了8种，太棒了，么么哒！" (I chose one medicine, and it recommended 8 more, too cool!). Below this, a green checkmark indicates "商品已成功加入购物车！" (Item successfully added to shopping cart). The main content area displays a product card for "罗博士 海狗人参丸100粒 男性保健品含淫羊藿非速效延时持久片" (Lobster and Deer Antler Pill 100 tablets, male health supplement containing Epimedium, non-immediate effect delay) at 1 piece for ¥98.00. To the right are buttons for "查看商品详情" (View product details) and "去购物车结算" (Go to shopping cart). Below the product card, a section titled "购买了该商品的用户还购买了" (Users who bought this product also bought) shows eight recommended items with their names, prices, and "加入购物车" (Add to cart) buttons. At the bottom, there is a navigation bar with numbers 1, 2, 3, 4.

3、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，纸尿布+啤酒。

The image illustrates a retail cross-sell strategy. In the center, large red text reads "漂亮媳妇" (Pretty Wife). On the left, there is a product image for "Suki's Baby Diapers" (舒比奇纸尿裤) with a "2包" (2 packages) label. An orange arrow points from this image towards the central text. On the right, there is another product image for "Yanjing Beer" (燕京啤酒) with a "纯生" (Purified) label. Another orange arrow points from this image towards the central text. At the bottom right, the slogan "让天下没有难学的技术" (Let the world learn difficult technology easily) is displayed.

 大数据应用场景

4、物流仓储：京东物流，上午下单下午送达、下午下单次日上午送达



让天下没有难学的技术

 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。

 大数据应用场景

8、人工智能 + 5G + 物联网 + 虚拟与现实



第 4 章 大数据发展前景

 大数据发展前景

- 1、党的十九大提出“**推动互联网、大数据、人工智能和实体经济深度融合**”。
- 2、2020年初，中央推出**34万亿**“新基建”投资计划

“新基建”投资规模拆分	
项目	2020年投资规模（亿元）
5G	3000
特高压	600
轨道交通	5000
充电桩	100
数据中心	1000
人工智能	350
工业互联网	100
合计	10150

让天下没有难学的技术



3、下一个风口

2020年是**5G**的元年，国家在大力铺设**5G**设备，**2021**年就是**5G**手机应用的开始，**也是大数据要爆发的1年**。**5G**带来的是每秒钟10g的数据，会给每家公司都带来海量的数据。那么传统的Java工具根本解决不了海量数据的存储。就更不用说海量数据的计算了。如果你对**5G**的感触不够深，可以回忆一下**3G**和**4G**的区别。**3G**时只能打电话、发短信，当时还觉得很好，觉得**3G**不错。但是**4G来了后**，大家很少打电话和发短信了，都改为**语音、视频、直播、网上购物**等生活方式，带火了淘宝、京东、美团、字节跳动等企业。没有跟上节奏的百度，有点摇摇欲坠。

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！

让天下没有难学的技术



4、人才紧缺、竞争压力小

有句话叫：“**选择大于努力**”选择一个好的方向，少奋斗十年。是否记得国家在2017年才开设大数据课程，当时是**北京大学、人民大学**等25所高校开设第一批大数据课程。今年才2021年。也就是今年才毕业，那么像Java、前端大学已经开设多少年了，包括培训班都加在一起，10多年，可想而知目前市场上，Java和前端的人才有多少。

大数据的人才目前除了培训机构培养的，没有真正的科班毕业，而且真正能培养好大数据人才的培训机构又有几个。**所以目前选择大数据是最佳选择。**

如果担心自己不是科班，其实也大可不必，因为大学真的学不了啥。只要是能考上本科，说明你不笨，那学大数据就没问题。

让天下没有难学的技术

 大数据发展前景

5、Boss直聘网站上的部分大数据工程师薪水如下

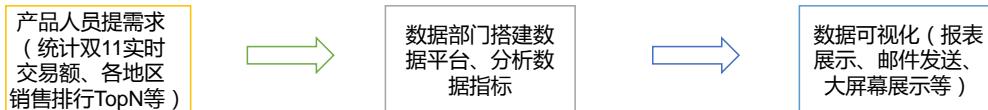


The screenshot shows five job listings from Boss直聘:

- 大数据flink开发 [北京 朝阳区 国贸]** 发布于11月25日
15-30K 1-3年 | 本科 袁女士 HR
Flink ETL 数据分析 Hive 实时计算
新瑞鹏宠物医疗集团 生活服务 不需要融资 10000人以上
- 大数据工程师 [北京 海淀区 知春路]** 发布于09月02日
25-50K 1-3年 | 本科 彭先生 内推
Hive 数据库开发 数据挖掘 SQL 数据仓库 今日头条 移动互联网 不需要融资 10000人以上
- 医美业务部_大数据开... [北京 海淀区 西北旺]** 发布于11月26日
25-35K 15薪 1-3年 | 本科 韩先生 高级前端工程师
数据仓库 数据分析 计算机基础理论 大数据开发工程师 百度 互联网 已上市 10000人以上
- 【社招】大数据开发... [北京 朝阳区 牡丹园]** 发布于10月14日
13-26K 1-3年 | 本科 张女士 人事专员
Spark Hadoop Hive 优化经验 大数据组件 中科院信工所 信息安全 不需要融资 1000-9999人
- 医美业务部_大数据开... [北京 海淀区 西北旺]** (重复项)

让天下没有难学的技术

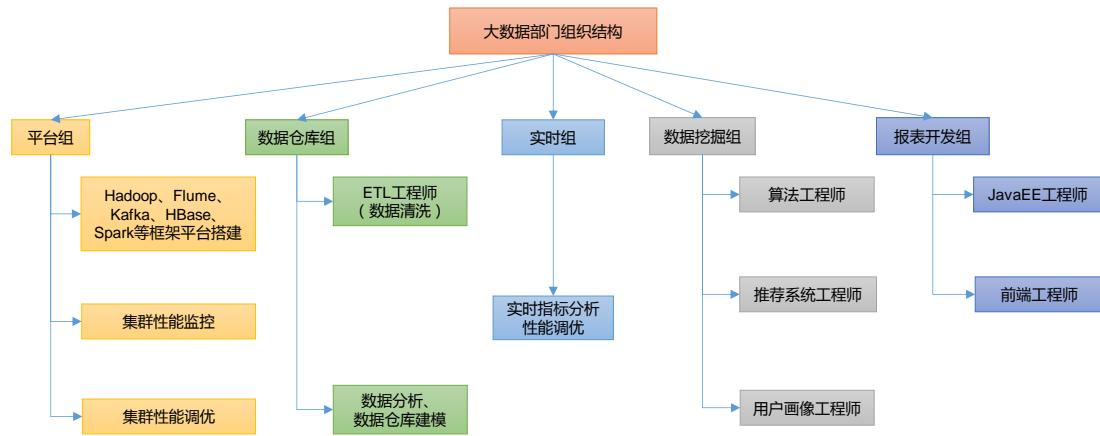
第 5 章 大数据部门间业务流程分析

 大数据部门间业务流程分析

让天下没有难学的技术

第 6 章 大数据部门内组织结构

大数据部门内组织结构



让天下没有难学的技术

尚硅谷大数据技术之 Hadoop (入门)

(作者: 尚硅谷大数据研发部)

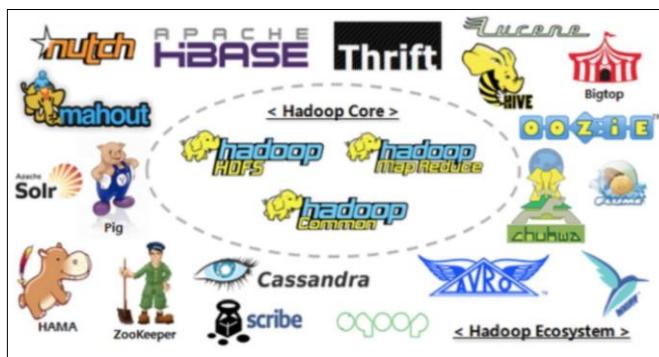
版本: V3.3

第 1 章 Hadoop 概述

1.1 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**。



让天下没有难学的技术

1.2 Hadoop 发展历史 (了解)



- 1) Hadoop创始人**Doug Cutting**，为了实现与Google类似全文搜索功能，他在**Lucene**框架基础上进行优化升级，查询引擎和索引引擎。



Hadoop创始人Doug Cutting

- 2) 2001年年底Lucene成为Apache基金会的一个子项目。
- 3) 对于海量数据的场景，Lucene框架面对与Google同样的困难，**存储海量数据困难，检索海量速度慢**。
- 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

让天下没有难学的技术

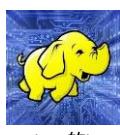


6) 2003-2004年，Google公开了部分GFS和MapReduce思想的细节，以此为基础Doug Cutting等人用了2年业余时间实现了DFS和MapReduce机制，使Nutch性能飙升。

7) 2005 年Hadoop 作为 Lucene的子项目 Nutch的一部分正式引入Apache基金会。

8) 2006 年 3 月份，Map-Reduce和Nutch Distributed File System (NDFS) 分别被纳入到 Hadoop 项目中，Hadoop就此正式诞生，标志着大数据时代来临。

9) 名字来源于Doug Cutting儿子的玩具大象



Hadoop的logo

让天下没有难学的技术

1.3 Hadoop 三大发行版本（了解）

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。2006

Cloudera 内部集成了很多大数据框架，对应产品 **CDH**。2008

Hortonworks 文档较好，对应产品 **HDP**。2011

Hortonworks 现在已经被 Cloudera 公司收购，推出新的品牌 **CDP**。

The screenshot shows the Cloudera website at cloudera.com/products/pricing.html. The main heading is "CDP私有云". Below it, a sub-headline reads: "在数据中心中获取云本地分析的速度，简便性和成本效益，或者将集群升级到最新的开源分析。或两者都做。你的选择。" Two pricing options are listed: "基本版" (Basic) and "加版" (Plus). The Basic version is \$10,000 / 节点+变量，并且有"年度订阅" (Annual Subscription) 选项。The Plus version is \$400 / CCU + \$25 / TB，并且有"年度订阅" (Annual Subscription) 选项。At the bottom, there is a red box highlighting the message: "Effective Jan 31, 2021, all Cloudera software requires a subscription and must be accessed via the payroll." A cookie consent banner is also visible.

1) Apache Hadoop

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

官网地址: <http://hadoop.apache.org>

下载地址: <https://hadoop.apache.org/releases.html>

2) Cloudera Hadoop

官网地址: <https://www.cloudera.com/downloads/cdh>

下载地址: https://docs.cloudera.com/documentation/enterprise/6/release-notes/topics/rg_cdh_6_download.html

(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。Cloudera 的标价为每年每个节点 **10000 美元**。

(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。

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) 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。

(4) 2018 年 Hortonworks 目前已经被 Cloudera 公司收购。

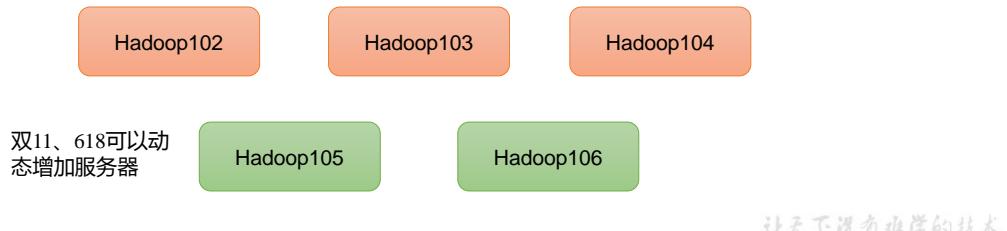
1.4 Hadoop 优势 (4 高)

Hadoop优势 (4高)

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。

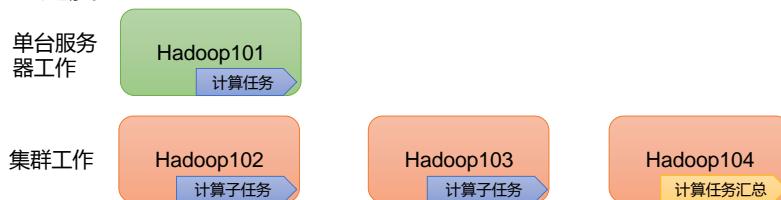


2) 高扩展性 : 在集群间分配任务数据 , 可方便的扩展数以千计的节点。

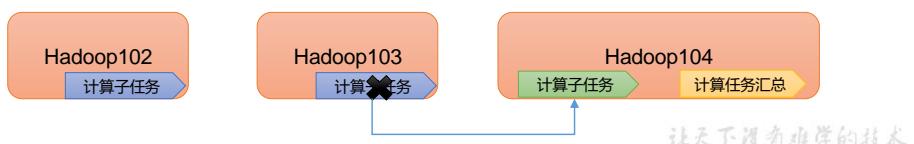


Hadoop优势 (4高)

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。



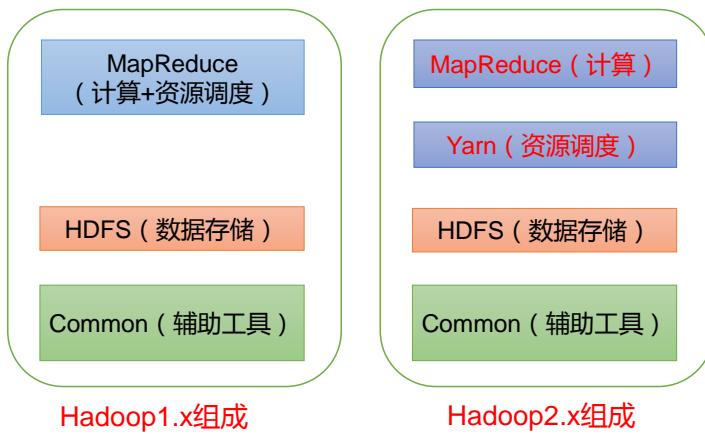
4) 高容错性 : 能够自动将失败的任务重新分配。



1.5 Hadoop 组成 (面试重点)



Hadoop1.x、2.x、3.x区别



在 Hadoop1.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。

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

Hadoop3.x 在组成上没有变化。

让天下没有难学的技术

1.5.1 HDFS 架构概述

Hadoop Distributed File System，简称 **HDFS**，是一个分布式文件系统。



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)：每隔一段时间对 NameNode 元数据备份。

让天下没有难学的技术

1.5.2 YARN 架构概述

Yet Another Resource Negotiator 简称 **YARN**，另一种资源协调者，是 Hadoop 的资源管理器。

 YARN架构概述

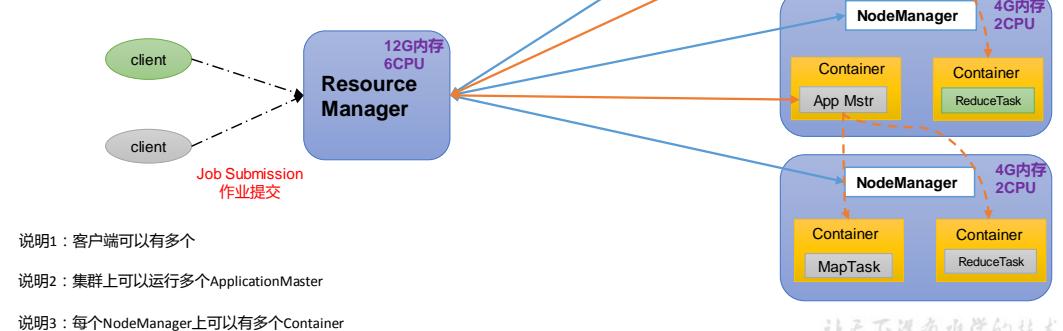
1) ResourceManager (RM) : 整个集群资源 (内存、CPU 等) 的老大

2) NodeManager (NM) : 单个节点服务器资源老大

3) ApplicationMaster (AM) : 单个任务运行的老大

4) Container : 容器, 相当一台独立的服务器, 里面封装了

任务运行所需要的资源, 如内存、CPU、磁盘、网络等。



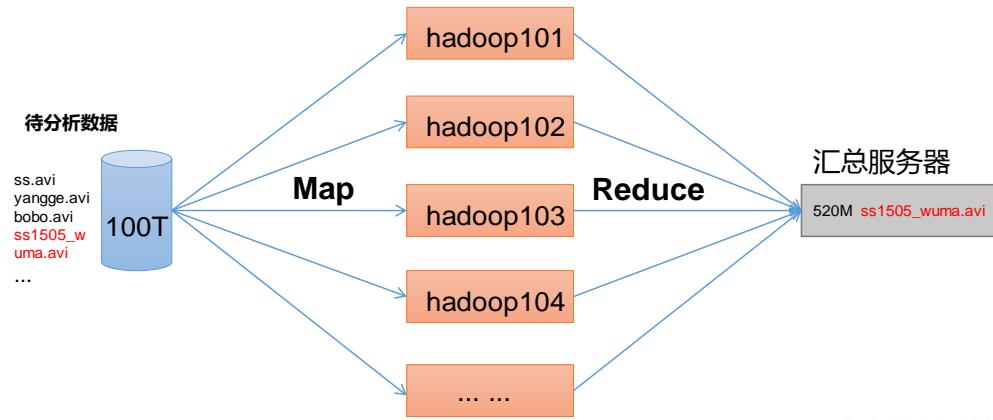
1.5.3 MapReduce 架构概述

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

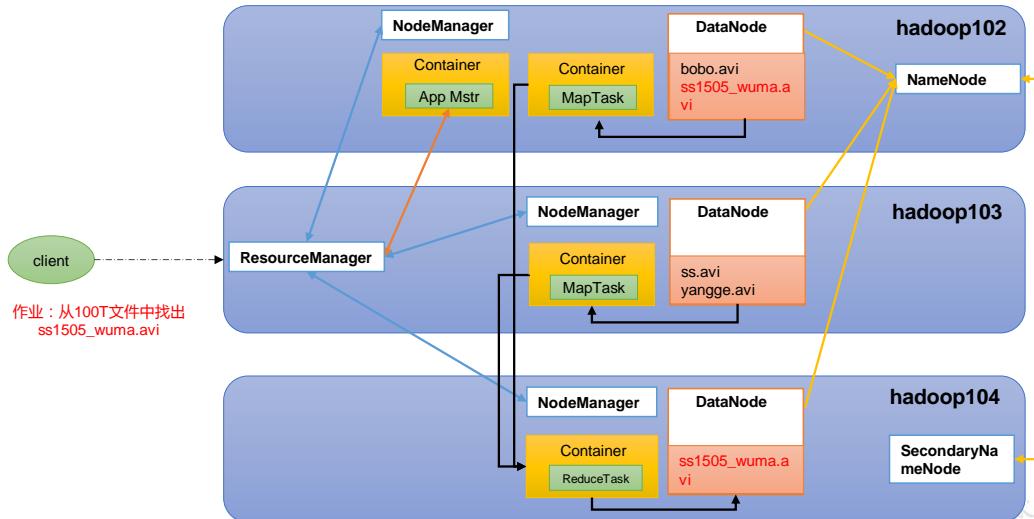
 MapReduce架构概述

任务需求: 找出宋宋老师2015年5月份的教学视频



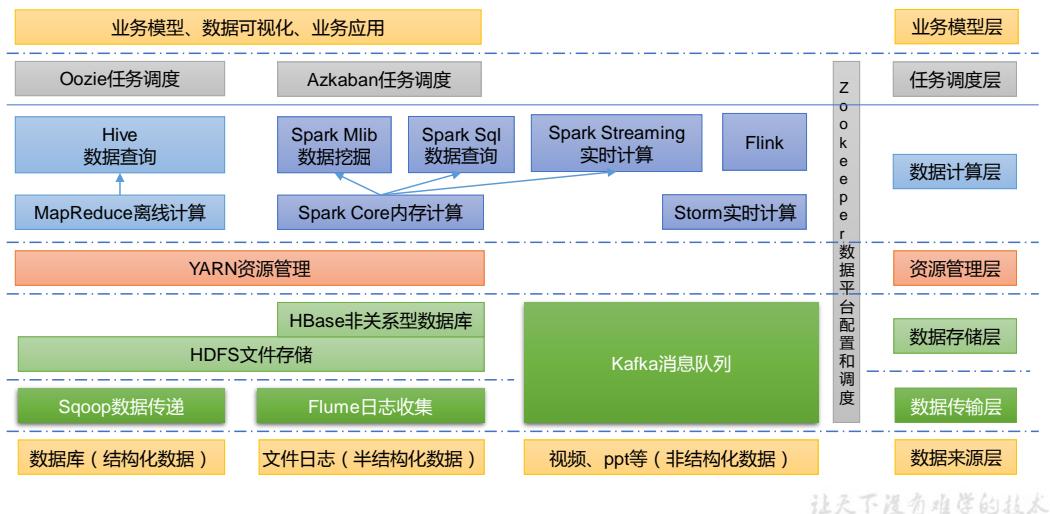
1.5.4 HDFS、YARN、MapReduce 三者关系

HDFS、YARN、MapReduce 三者关系



1.6 大数据技术生态体系

大数据技术生态体系

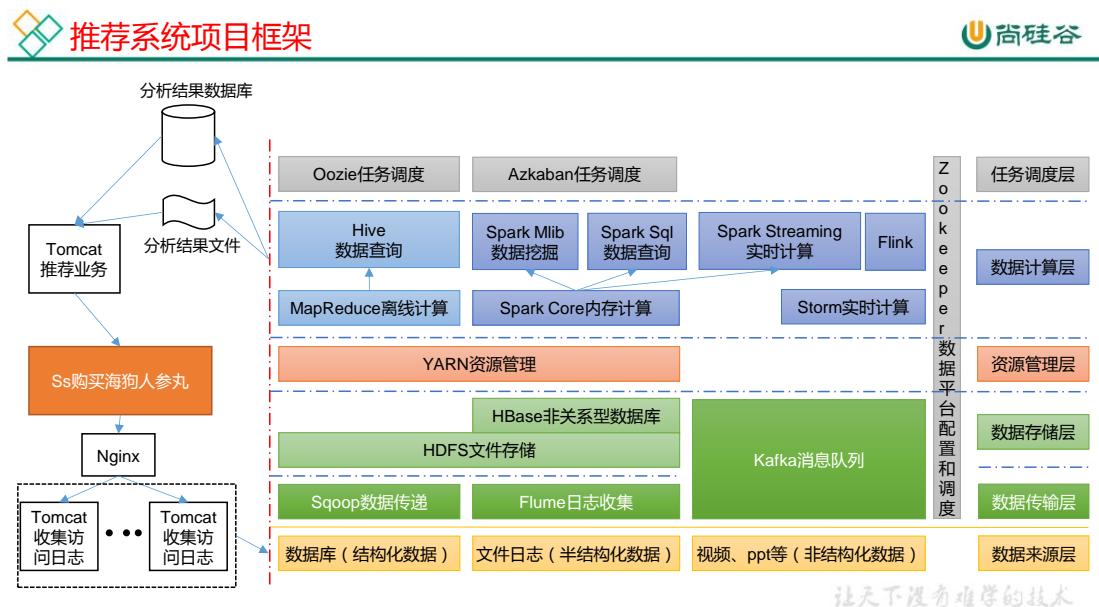


图中涉及的技术名词解释如下：

- 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；
- 3) **Kafka:** Kafka 是一种高吞吐量的分布式发布订阅消息系统；

- 4) 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 上存储的数据进行计算。
- 5) Flink: Flink 是当前最流行的开源大数据内存计算框架。用于实时计算的场景较多。
- 6) Oozie: Oozie 是一个管理 Hadoop 作业 (job) 的工作流程调度管理系统。
- 7) Hbase: HBase 是一个分布式的、面向列的开源数据库。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。
- 8) Hive: Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行。其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。
- 9) ZooKeeper: 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。

1.7 推荐系统框架图



第 2 章 Hadoop 运行环境搭建（开发重点）

2.1 模板虚拟机环境准备

- 0) 安装模板虚拟机，IP 地址 192.168.10.100、主机名称 hadoop100、内存 4G、硬盘 50G



尚硅谷大数据技术
之模板虚拟机环境准备

1) hadoop100 虚拟机配置要求如下（本文 Linux 系统全部以 CentOS-7.5-x86-1804 为例）

(1) 使用 yum 安装需要虚拟机可以正常上网，yum 安装前可以先测试下虚拟机联网情况

```
[root@hadoop100 ~]# ping www.baidu.com
PING www.baidu.com (14.215.177.39) 56(84) bytes of data.
64 bytes from 14.215.177.39 (14.215.177.39): icmp_seq=1
ttl=128 time=8.60 ms
64 bytes from 14.215.177.39 (14.215.177.39): icmp_seq=2
ttl=128 time=7.72 ms
```

(2) 安装 epel-release

注：Extra Packages for Enterprise Linux 是为“红帽系”的操作系统提供额外的软件包，适用于 RHEL、CentOS 和 Scientific Linux。相当于是一个软件仓库，大多数 rpm 包在官方 repository 中是找不到的）

```
[root@hadoop100 ~]# yum install -y epel-release
```

(3) 注意：如果 Linux 安装的是最小系统版，还需要安装如下工具；如果安装的是 Linux 桌面标准版，不需要执行如下操作

➤ net-tools：工具包集合，包含 ifconfig 等命令

```
[root@hadoop100 ~]# yum install -y net-tools
```

➤ vim：编辑器

```
[root@hadoop100 ~]# yum install -y vim
```

2) 关闭防火墙，关闭防火墙开机自启

```
[root@hadoop100 ~]# systemctl stop firewalld
[root@hadoop100 ~]# systemctl disable firewalld.service
```

注意：在企业开发时，通常单个服务器的防火墙时关闭的。公司整体对外会设置非常安全的防火墙

3) 创建 atguigu 用户，并修改 atguigu 用户的密码

```
[root@hadoop100 ~]# useradd atguigu
[root@hadoop100 ~]# passwd atguigu
```

4) 配置 atguigu 用户具有 root 权限，方便后期加 sudo 执行 root 权限的命令

```
[root@hadoop100 ~]# vim /etc/sudoers
```

修改/etc/sudoers 文件，在%wheel 这行下面添加一行，如下所示：

```
## Allow root to run any commands anywhere
root    ALL=(ALL)      ALL

## Allows people in group wheel to run all commands
%wheel  ALL=(ALL)      ALL
```

```
atguigu  ALL=(ALL)      NOPASSWD:ALL
```

注意: atguigu 这一行不要直接放到 root 行下面, 因为所有用户都属于 wheel 组, 你先配置了 atguigu 具有免密功能, 但是程序执行到%wheel 行时, 该功能又被覆盖回需要密码。所以 atguigu 要放到%wheel 这行下面。

5) 在/opt 目录下创建文件夹, 并修改所属主和所属组

(1) 在/opt 目录下创建 module、software 文件夹

```
[root@hadoop100 ~]# mkdir /opt/module
[root@hadoop100 ~]# mkdir /opt/software
```

(2) 修改 module、software 文件夹的所有者和所属组均为 atguigu 用户

```
[root@hadoop100 ~]# chown atguigu:atguigu /opt/module
[root@hadoop100 ~]# chown atguigu:atguigu /opt/software
```

(3) 查看 module、software 文件夹的所有者和所属组

```
[root@hadoop100 ~]# cd /opt/
[root@hadoop100 opt]# ll
总用量 12
drwxr-xr-x. 2 atguigu atguigu 4096 5月 28 17:18 module
drwxr-xr-x. 2 root     root    4096 9月  7 2017 rh
drwxr-xr-x. 2 atguigu atguigu 4096 5月 28 17:18 software
```

6) 卸载虚拟机自带的 JDK

注意: 如果你的虚拟机是最小化安装不需要执行这一步。

```
[root@hadoop100 ~]# rpm -qa | grep -i java | xargs -n1 rpm -e
--nodeps
```

- rpm -qa: 查询所安装的所有 rpm 软件包
- grep -i: 忽略大小写
- xargs -n1: 表示每次只传递一个参数
- rpm -e --nodeps: 强制卸载软件

7) 重启虚拟机

```
[root@hadoop100 ~]# reboot
```

2.2 克隆虚拟机

1) 利用模板机 hadoop100, 克隆三台虚拟机: hadoop102 hadoop103 hadoop104

注意: 克隆时, 要先关闭 hadoop100

2) 修改克隆机 IP, 以下以 hadoop102 举例说明

(1) 修改克隆虚拟机的静态 IP

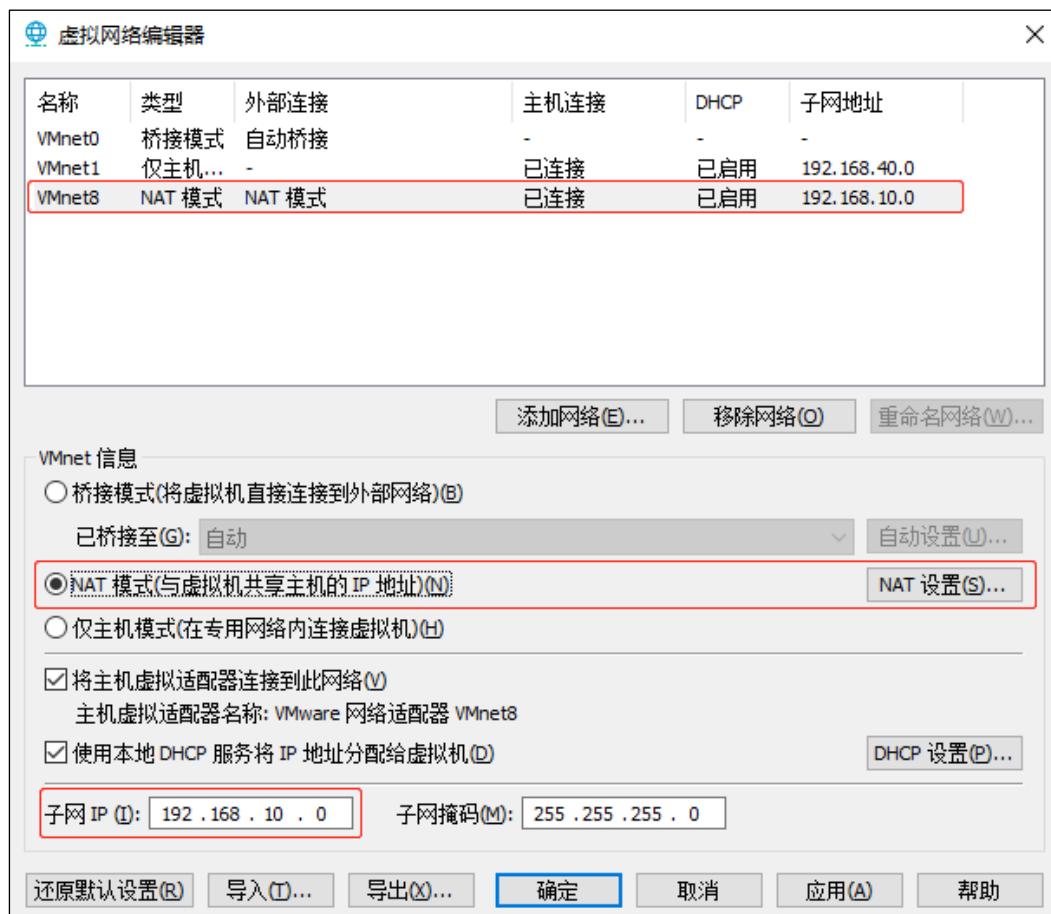
```
[root@hadoop100 ~]# vim /etc/sysconfig/network-scripts/ifcfg-
ens33
```

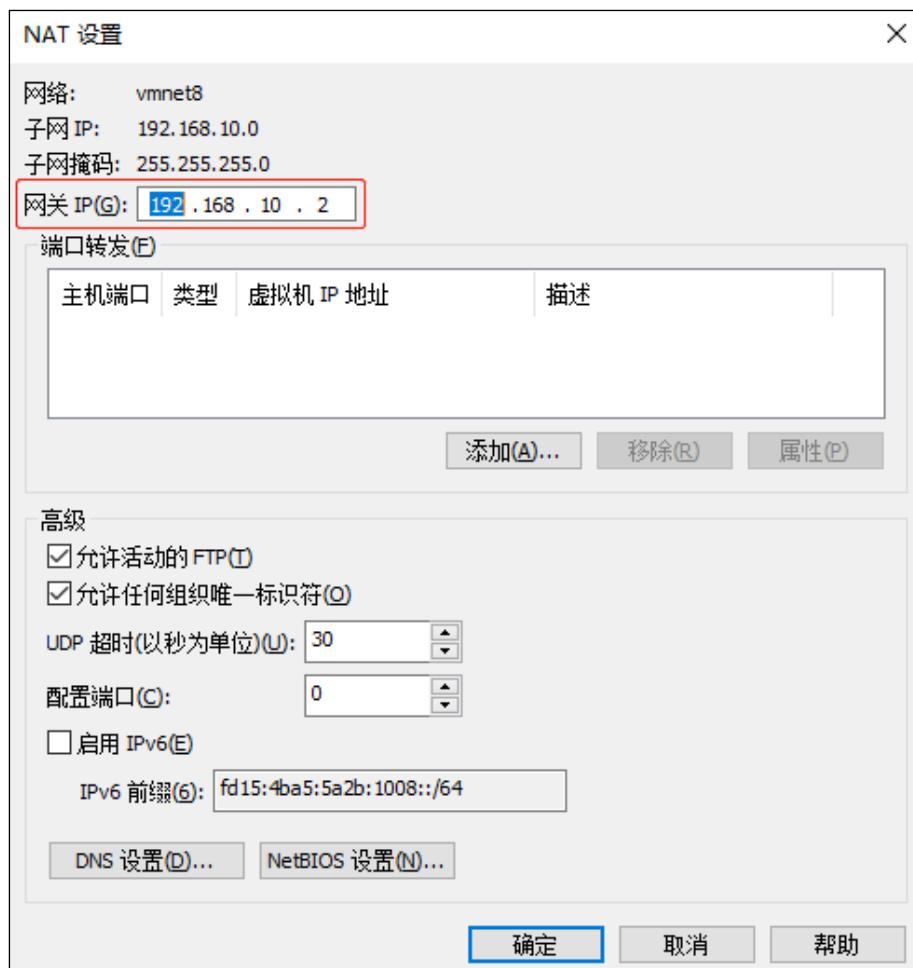
改成

```
DEVICE=ens33
TYPE=Ethernet
```

```
ONBOOT=yes
BOOTPROTO=static
NAME="ens33"
IPADDR=192.168.10.102
PREFIX=24
GATEWAY=192.168.10.2
DNS1=192.168.10.2
```

(2) 查看 Linux 虚拟机的虚拟网络编辑器, 编辑->虚拟网络编辑器->VMnet8





(3) 查看 Windows 系统适配器 VMware Network Adapter VMnet8 的 IP 地址



(4) 保证 Linux 系统 ifcfg-ens33 文件中 IP 地址、虚拟网络编辑器地址和 Windows 系统 VM8 网络 IP 地址相同。

3) 修改克隆机主机名，以下以 hadoop102 举例说明

(1) 修改主机名称

```
[root@hadoop100 ~]# vim /etc/hostname  
hadoop102
```

(2) 配置 Linux 克隆机主机名称映射 hosts 文件，打开/etc/hosts

```
[root@hadoop100 ~]# vim /etc/hosts
```

添加如下内容

```
192.168.10.100 hadoop100  
192.168.10.101 hadoop101  
192.168.10.102 hadoop102  
192.168.10.103 hadoop103  
192.168.10.104 hadoop104
```

```
192.168.10.105 hadoop105  
192.168.10.106 hadoop106  
192.168.10.107 hadoop107  
192.168.10.108 hadoop108
```

4) 重启克隆机 hadoop102

```
[root@hadoop100 ~]# reboot
```

5) 修改 windows 的主机映射文件 (hosts 文件)

(1) 如果操作系统是 window7, 可以直接修改

(a) 进入 C:\Windows\System32\drivers\etc 路径

(b) 打开 hosts 文件并添加如下内容, 然后保存

```
192.168.10.100 hadoop100  
192.168.10.101 hadoop101  
192.168.10.102 hadoop102  
192.168.10.103 hadoop103  
192.168.10.104 hadoop104  
192.168.10.105 hadoop105  
192.168.10.106 hadoop106  
192.168.10.107 hadoop107  
192.168.10.108 hadoop108
```

(2) 如果操作系统是 window10, 先拷贝出来, 修改保存以后, 再覆盖即可

(a) 进入 C:\Windows\System32\drivers\etc 路径

(b) 拷贝 hosts 文件到桌面

(c) 打开桌面 hosts 文件并添加如下内容

```
192.168.10.100 hadoop100  
192.168.10.101 hadoop101  
192.168.10.102 hadoop102  
192.168.10.103 hadoop103  
192.168.10.104 hadoop104  
192.168.10.105 hadoop105  
192.168.10.106 hadoop106  
192.168.10.107 hadoop107  
192.168.10.108 hadoop108
```

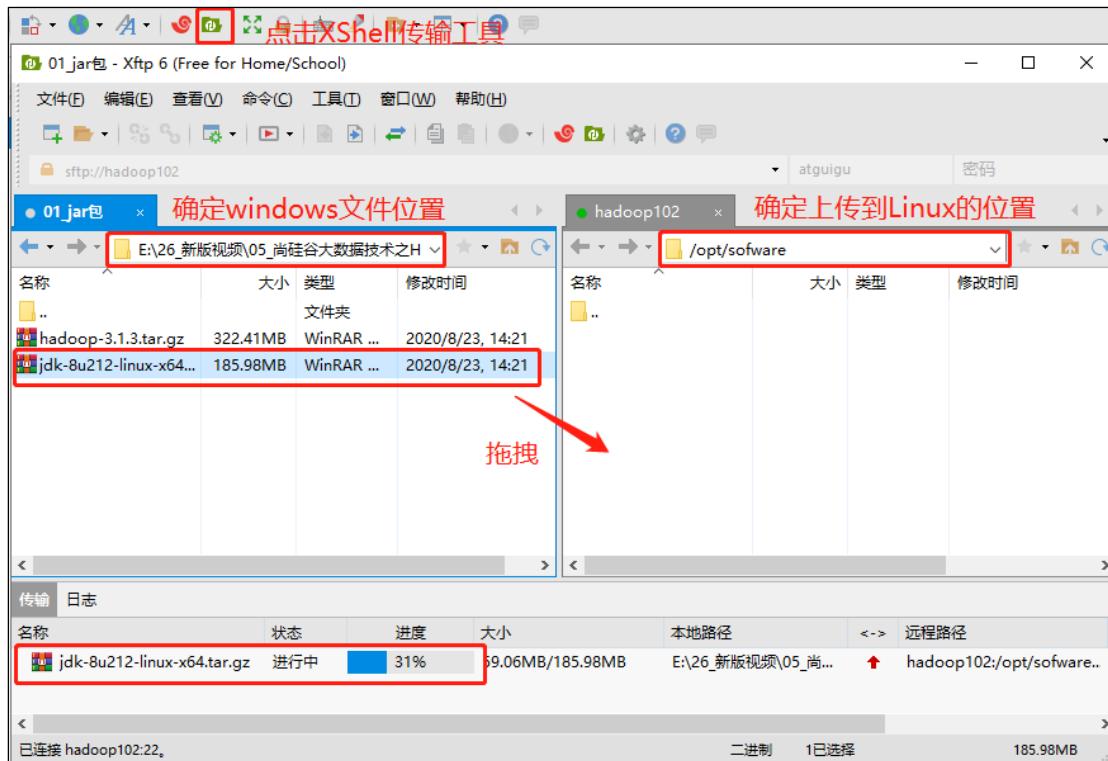
(d) 将桌面 hosts 文件覆盖 C:\Windows\System32\drivers\etc 路径 hosts 文件

2.3 在 hadoop102 安装 JDK

1) 卸载现有 JDK

注意: 安装 JDK 前, 一定确保提前删除了虚拟机自带的 JDK。详细步骤见问文档 3.1 节中卸载 JDK 步骤。

2) 用 XShell 传输工具将 JDK 导入到 opt 目录下面的 software 文件夹下面



3) 在 Linux 系统下的 opt 目录中查看软件包是否导入成功

```
[atguigu@hadoop102 ~]$ ls /opt/software/
```

看到如下结果：

```
jdk-8u212-linux-x64.tar.gz
```

4) 解压 JDK 到/opt/module 目录下

```
[atguigu@hadoop102 software]$ tar -zxvf jdk-8u212-linux-x64.tar.gz -C /opt/module/
```

5) 配置 JDK 环境变量

(1) 新建/etc/profile.d/my_env.sh 文件

```
[atguigu@hadoop102 ~]$ sudo vim /etc/profile.d/my_env.sh
```

添加如下内容

```
#JAVA_HOME
export JAVA_HOME=/opt/module/jdk1.8.0_212
export PATH=$PATH:$JAVA_HOME/bin
```

(2) 保存后退出

```
:wq
```

(3) source 一下/etc/profile 文件，让新的环境变量 PATH 生效

```
[atguigu@hadoop102 ~]$ source /etc/profile
```

6) 测试 JDK 是否安装成功

```
[atguigu@hadoop102 ~]$ java -version
```

如果能看到以下结果，则代表 Java 安装成功。

```
java version "1.8.0_212"
```

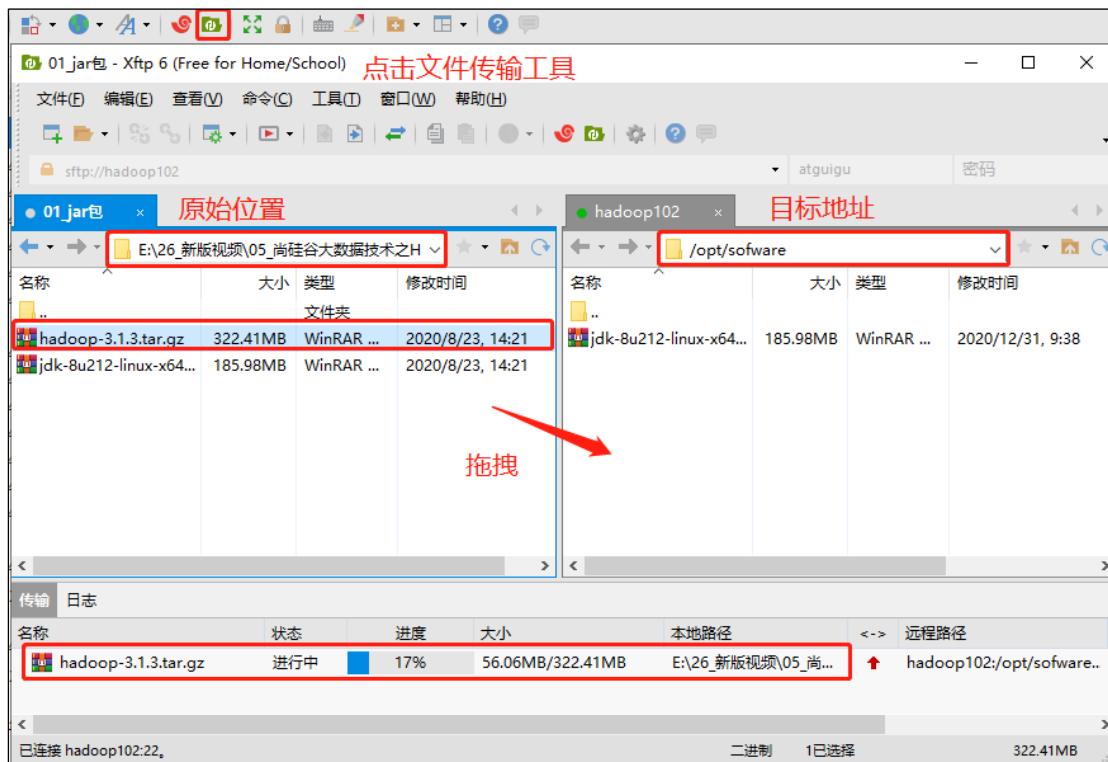
注意：重启（如果 `java -version` 可以用就不用重启）

```
[atguigu@hadoop102 ~]$ sudo reboot
```

2.4 在 hadoop102 安装 Hadoop

Hadoop 下载地址：<https://archive.apache.org/dist/hadoop/common/hadoop-3.1.3/>

- 用 XShell 文件传输工具将 `hadoop-3.1.3.tar.gz` 导入到 `opt` 目录下面的 `software` 文件夹下面



- 进入到 Hadoop 安装包路径下

```
[atguigu@hadoop102 ~]$ cd /opt/software/
```

- 解压安装文件到 `/opt/module` 下面

```
[atguigu@hadoop102 software]$ tar -zxvf hadoop-3.1.3.tar.gz -C /opt/module/
```

- 查看是否解压成功

```
[atguigu@hadoop102 software]$ ls /opt/module/
hadoop-3.1.3
```

- 将 Hadoop 添加到环境变量

(1) 获取 Hadoop 安装路径

```
[atguigu@hadoop102 hadoop-3.1.3]$ pwd
/opt/module/hadoop-3.1.3
```

(2) 打开 `/etc/profile.d/my_env.sh` 文件

```
[atguigu@hadoop102         .hadoop-3.1.3]$ sudo vim
/etc/profile.d/my_env.sh
```

- 在 my_env.sh 文件末尾添加如下内容: (shift+g)

```
#HADOOP_HOME
export HADOOP_HOME=/opt/module/hadoop-3.1.3
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
```

- 保存并退出: :wq

(3) 让修改后的文件生效

```
[atguigu@hadoop102 hadoop-3.1.3]$ source /etc/profile
```

6) 测试是否安装成功

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop version
Hadoop 3.1.3
```

7) 重启 (如果 Hadoop 命令不能用再重启虚拟机)

```
[atguigu@hadoop102 hadoop-3.1.3]$ sudo reboot
```

2.5 Hadoop 目录结构

1) 查看 Hadoop 目录结构

```
[atguigu@hadoop102 hadoop-3.1.3]$ ll
总用量 52
drwxr-xr-x. 2 atguigu atguigu 4096 5月 22 2017 bin
drwxr-xr-x. 3 atguigu atguigu 4096 5月 22 2017 etc
drwxr-xr-x. 2 atguigu atguigu 4096 5月 22 2017 include
drwxr-xr-x. 3 atguigu atguigu 4096 5月 22 2017 lib
drwxr-xr-x. 2 atguigu atguigu 4096 5月 22 2017 libexec
-rw-r--r--. 1 atguigu atguigu 15429 5月 22 2017 LICENSE.txt
-rw-r--r--. 1 atguigu atguigu 101 5月 22 2017 NOTICE.txt
-rw-r--r--. 1 atguigu atguigu 1366 5月 22 2017 README.txt
drwxr-xr-x. 2 atguigu atguigu 4096 5月 22 2017 sbin
drwxr-xr-x. 4 atguigu atguigu 4096 5月 22 2017 share
```

2) 重要目录

- (1) bin 目录: 存放对 Hadoop 相关服务 (hdfs, yarn, mapred) 进行操作的脚本
- (2) etc 目录: Hadoop 的配置文件目录, 存放 Hadoop 的配置文件
- (3) lib 目录: 存放 Hadoop 的本地库 (对数据进行压缩解压缩功能)
- (4) sbin 目录: 存放启动或停止 Hadoop 相关服务的脚本
- (5) share 目录: 存放 Hadoop 的依赖 jar 包、文档、和官方案例

第 3 章 Hadoop 运行模式

- 1) Hadoop 官方网站: <http://hadoop.apache.org/>
- 2) 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。

- **本地模式:** 单机运行, 只是用来演示一下官方案例。**生产环境不用。**

- 伪分布式模式：也是单机运行，但是具备 Hadoop 集群的所有功能，一台服务器模拟一个分布式的环境。**个别缺钱的公司用来测试，生产环境不用。**
- 完全分布式模式：多台服务器组成分布式环境。**生产环境使用。**

3.1 本地运行模式（官方 WordCount）

- 1) 创建在 hadoop-3.1.3 文件下面创建一个 wcinput 文件夹

```
[atguigu@hadoop102 hadoop-3.1.3]$ mkdir wcinput
```

- 2) 在 wcinput 文件下创建一个 word.txt 文件

```
[atguigu@hadoop102 hadoop-3.1.3]$ cd wcinput
```

- 3) 编辑 word.txt 文件

```
[atguigu@hadoop102 wcinput]$ vim word.txt
```

- 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-3.1.3

- 5) 执行程序

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar  
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar  
wordcount wcinput wcoutput
```

- 6) 查看结果

```
[atguigu@hadoop102 hadoop-3.1.3]$ cat wcoutput/part-r-00000
```

看到如下结果：

```
atguigu 2  
hadoop 2  
mapreduce 1  
yarn 1
```

3.2 完全分布式运行模式（开发重点）

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

3.2.1 虚拟机准备

详见 2.1、2.2 两节。

3.2.2 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。（from server1 to server2）

(2) 基本语法

scp	-r	\$pdir/\$fname	\$user@\$host:\$pdir/\$fname
命令	递归	要拷贝的文件路径/名称	目的地用户@主机:目的地路径/名称

(3) 案例实操

➤ **前提：**在 hadoop102、hadoop103、hadoop104 都已经创建好的 /opt/module、

/opt/software 两个目录，并且已经把这两个目录修改为 atguigu:atguigu

```
[atguigu@hadoop102 ~]$ sudo chown atguigu:atguigu -R /opt/module
```

(a) 在 hadoop102 上，将 hadoop102 中 /opt/module/jdk1.8.0_212 目录拷贝到 hadoop103 上。

```
[atguigu@hadoop102 ~]$ scp -r /opt/module/jdk1.8.0_212 atguigu@hadoop103:/opt/module
```

(b) 在 hadoop103 上，将 hadoop102 中 /opt/module/hadoop-3.1.3 目录拷贝到 hadoop103 上。

```
[atguigu@hadoop103 ~]$ scp -r atguigu@hadoop102:/opt/module/hadoop-3.1.3 /opt/module/
```

(c) 在 hadoop103 上操作，将 hadoop102 中 /opt/module 目录下所有目录拷贝到 hadoop104 上。

```
[atguigu@hadoop103 opt]$ scp -r atguigu@hadoop102:/opt/module/* atguigu@hadoop104:/opt/module
```

2) rsync 远程同步工具

rsync 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) 基本语法

rsync	-av	\$pdir/\$fname	\$user@\$host:\$pdir/\$fname
命令	选项参数	要拷贝的文件路径/名称	目的地用户@主机:目的地路径/名称
选项参数说明			

选项	功能
-a	归档拷贝
-v	显示复制过程

(2) 案例实操

(a) 删除 hadoop103 中 /opt/module/hadoop-3.1.3/wcinput

```
[atguigu@hadoop103 hadoop-3.1.3]$ rm -rf wcinput/
```

(b) 同步 hadoop102 中的 /opt/module/hadoop-3.1.3 到 hadoop103

```
[atguigu@hadoop102 module]$ rsync -av hadoop-3.1.3/
atguigu@hadoop103:/opt/module/hadoop-3.1.3/
```

3) xsync 集群分发脚本

(1) 需求：循环复制文件到所有节点的相同目录下

(2) 需求分析：

(a) rsync 命令原始拷贝：

```
rsync -av /opt/module atguigu@hadoop103:/opt/
```

(b) 期望脚本：

xsync 要同步的文件名称

(c) 期望脚本在任何路径都能使用（脚本放在声明了全局环境变量的路径）

```
[atguigu@hadoop102 ~]$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/atguigu/.local/bin:/home/atguigu/bin:/opt/module/jdk1.8.0_212/bin
```

(3) 脚本实现

(a) 在 /home/atguigu/bin 目录下创建 xsync 文件

```
[atguigu@hadoop102 opt]$ cd /home/atguigu
[atguigu@hadoop102 ~]$ mkdir bin
[atguigu@hadoop102 ~]$ cd bin
[atguigu@hadoop102 bin]$ vim xsync
```

在该文件中编写如下代码

```
#!/bin/bash

#1. 判断参数个数
if [ $# -lt 1 ]
then
    echo Not Enough Arguement!
    exit;
fi
```

```

#2. 遍历集群所有机器
for host in hadoop102 hadoop103 hadoop104
do
    echo ===== $host =====
#3. 遍历所有目录，挨个发送

for file in $@
do
    #4. 判断文件是否存在
    if [ -e $file ]
    then
        #5. 获取父目录
        pdir=$(cd -P $(dirname $file); pwd)

        #6. 获取当前文件的名称
        fname=$(basename $file)
        ssh $host "mkdir -p $pdir"
        rsync -av $pdir/$fname $host:$pdir
    else
        echo $file does not exists!
    fi
done
done

```

(b) 修改脚本 xsync 具有执行权限

```
[atguigu@hadoop102 bin]$ chmod +x xsync
```

(c) 测试脚本

```
[atguigu@hadoop102 ~]$ xsync /home/atguigu/bin
```

(d) 将脚本复制到/bin 中，以便全局调用

```
[atguigu@hadoop102 bin]$ sudo cp xsync /bin/
```

(e) 同步环境变量配置 (root 所有者)

```
[atguigu@hadoop102 ~]$ sudo ./bin/xsync
/etc/profile.d/my_env.sh
```

注意：如果用了 sudo，那么 xsync 一定要给它的路径补全。

让环境变量生效

```
[atguigu@hadoop103 bin]$ source /etc/profile
[atguigu@hadoop104 opt]$ source /etc/profile
```

3.2.3 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 ~]$ ssh hadoop103
```

➤ 如果出现如下内容

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

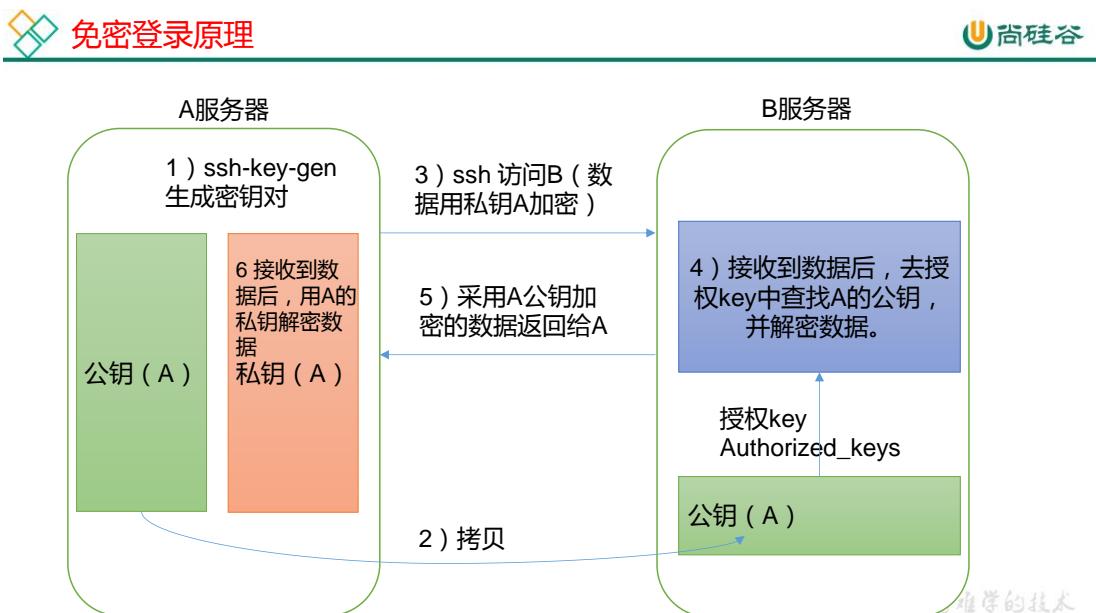
➤ 输入 yes，并回车

(3) 退回到 hadoop102

```
[atguigu@hadoop103 ~]$ exit
```

2) 无密钥配置

(1) 免密登录原理



(2) 生成公钥和私钥

```
[atguigu@hadoop102 .ssh]$ pwd  
/home/atguigu/.ssh
```

```
[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 hadoop102  
[atguigu@hadoop102 .ssh]$ ssh-copy-id hadoop103  
[atguigu@hadoop102 .ssh]$ ssh-copy-id hadoop104
```

注意：

还需要在 hadoop103 上采用 atguigu 账号配置一下无密登录到 hadoop102、hadoop103、

hadoop104 服务器上。

还需要在 hadoop104 上采用 atguigu 账号配置一下无密登录到 hadoop102、hadoop103、

hadoop104 服务器上。

还需要在 hadoop102 上采用 root 账号，配置一下无密登录到 hadoop102、hadoop103、

hadoop104；

3) .ssh 文件夹下 (~/.ssh) 的文件功能解释

known_hosts	记录 ssh 访问过计算机的公钥 (public key)
id_rsa	生成的私钥
id_rsa.pub	生成的公钥
authorized_keys	存放授权过的无密登录服务器公钥

3.2.4 集群配置

1) 集群部署规划

注意：

- NameNode 和 SecondaryNameNode 不要安装在同一台服务器
- ResourceManager 也很消耗内存，不要和 NameNode、SecondaryNameNode 配置在同一台机器上。

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

2) 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：

要获取的默认文件	文件存放在 Hadoop 的 jar 包中的位置
[core-default.xml]	hadoop-common-3.1.3.jar/core-default.xml
[hdfs-default.xml]	hadoop-hdfs-3.1.3.jar/hdfs-default.xml
[yarn-default.xml]	hadoop-yarn-common-3.1.3.jar/yarn-default.xml
[mapred-default.xml]	hadoop-mapreduce-client-core-3.1.3.jar/mapred-default.xml

(2) 自定义配置文件：

core-site.xml、**hdfs-site.xml**、**yarn-site.xml**、**mapred-site.xml** 四个配置文件存放在 \$HADOOP_HOME/etc/hadoop 这个路径上，用户可以根据项目需求重新进行修改配置。

3) 配置集群

(1) 核心配置文件

配置 core-site.xml

```
[atguigu@hadoop102 ~]$ cd $HADOOP_HOME/etc/hadoop
```

```
[atguigu@hadoop102 hadoop]$ vim core-site.xml
```

文件内容如下：

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

<configuration>
    <!-- 指定 NameNode 的地址 -->
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://hadoop102:8020</value>
    </property>

    <!-- 指定 hadoop 数据的存储目录 -->
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/opt/module/hadoop-3.1.3/data</value>
    </property>

    <!-- 配置 HDFS 网页登录使用的静态用户为 atguigu -->
    <property>
        <name>hadoop.http.staticuser.user</name>
        <value>atguigu</value>
    </property>
</configuration>
```

(2) HDFS 配置文件

配置 hdfs-site.xml

```
[atguigu@hadoop102 hadoop]$ vim hdfs-site.xml
```

文件内容如下：

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

<configuration>
    <!-- nn web 端访问地址-->
    <property>
        <name>dfs.namenode.http-address</name>
        <value>hadoop102:9870</value>
    </property>
    <!-- 2nn web 端访问地址-->
    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>hadoop104:9868</value>
    </property>
</configuration>
```

(3) YARN 配置文件

配置 yarn-site.xml

```
[atguigu@hadoop102 hadoop]$ vim yarn-site.xml
```

文件内容如下：

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

```

<configuration>
    <!-- 指定 MR 走 shuffle -->
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!-- 指定 ResourceManager 的地址-->
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>hadoop103</value>
    </property>

    <!-- 环境变量的继承 -->
    <property>
        <name>yarn.nodemanager.env-whitelist</name>
        <value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CONF_DIR,CLASSPATH_PREPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
    </property>
</configuration>

```

(4) MapReduce 配置文件

配置 mapred-site.xml

```
[atguigu@hadoop102 hadoop]$ vim mapred-site.xml
```

文件内容如下：

```

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

<configuration>
    <!-- 指定 MapReduce 程序运行在 Yarn 上 -->
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>

```

4) 在集群上分发配置好的 Hadoop 配置文件

```
[atguigu@hadoop102      hadoop]$      xsync      /opt/module/hadoop-3.1.3/etc/hadoop/
```

5) 去 103 和 104 上查看文件分发情况

```
[atguigu@hadoop103      ~]$      cat      /opt/module/hadoop-3.1.3/etc/hadoop/core-site.xml
[atguigu@hadoop104      ~]$      cat      /opt/module/hadoop-3.1.3/etc/hadoop/core-site.xml
```

3.2.5 群起集群

1) 配置 workers

```
[atguigu@hadoop102      hadoop]$      vim      /opt/module/hadoop-3.1.3/etc/hadoop/workers
```

在该文件中增加如下内容：

```
hadoop102  
hadoop103  
hadoop104
```

注意：该文件中添加的内容结尾不允许有空格，文件中不允许有空行。

同步所有节点配置文件

```
[atguigu@hadoop102 hadoop]$ xsync /opt/module/hadoop-3.1.3/etc
```

2) 启动集群

(1) **如果集群是第一次启动**，需要在 hadoop102 节点格式化 NameNode (注意：格式化 NameNode，会产生新的集群 id，导致 NameNode 和 DataNode 的集群 id 不一致，集群找不到已往数据。如果集群在运行过程中报错，需要重新格式化 NameNode 的话，一定要先停止 namenode 和 datanode 进程，并且要删除所有机器的 data 和 logs 目录，然后再进行格式化。)

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs namenode -format
```

(2) 启动 HDFS

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
```

(3) 在配置了 ResourceManager 的节点 (hadoop103) 启动 YARN

```
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

(4) Web 端查看 HDFS 的 NameNode

(a) 浏览器中输入：<http://hadoop102:9870>

(b) 查看 HDFS 上存储的数据信息

(5) Web 端查看 YARN 的 ResourceManager

(a) 浏览器中输入：<http://hadoop103:8088>

(b) 查看 YARN 上运行的 Job 信息

3) 集群基本测试

(1) 上传文件到集群

➤ 上传小文件

```
[atguigu@hadoop102 ~]$ hadoop fs -mkdir /input  
[atguigu@hadoop102 ~]$ hadoop fs -put $HADOOP_HOME/wcinput/word.txt /input
```

➤ 上传大文件

```
[atguigu@hadoop102 ~]$ hadoop fs -put /opt/software/jdk-8u212-linu-x64.tar.gz /
```

(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-3.1.3/data/dfs/data/current/192.168.10.102-1610603650062/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.tar.gz
[atguigu@hadoop102 subdir0]$ cat blk_1073741837>>tmp.tar.gz
[atguigu@hadoop102 subdir0]$ tar -zxvf tmp.tar.gz
```

(4) 下载

```
[atguigu@hadoop104 software]$ hadoop fs -get /jdk-8u212-linux-x64.tar.gz ./
```

(5) 执行 wordcount 程序

```
[atguigu@hadoop102          hadoop-3.1.3]$      hadoop      jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar
wordcount /input /output
```

3.2.6 配置历史服务器

为了查看程序的历史运行情况，需要配置一下历史服务器。具体配置步骤如下：

1) 配置 mapred-site.xml

```
[atguigu@hadoop102 hadoop]$ vim mapred-site.xml
```

在该文件里面增加如下配置。

```
<!-- 历史服务器端地址 -->
<property>
    <name>mapreduce.jobhistory.address</name>
    <value>hadoop102:10020</value>
</property>

<!-- 历史服务器 web 端地址 -->
<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>hadoop102:19888</value>
</property>
```

2) 分发配置

```
[atguigu@hadoop102          hadoop]$      xsync
$HADOOP_HOME/etc/hadoop/mapred-site.xml
```

3) 在 hadoop102 启动历史服务器

```
[atguigu@hadoop102 hadoop]$ mapred --daemon start historyserver
```

4) 查看历史服务器是否启动

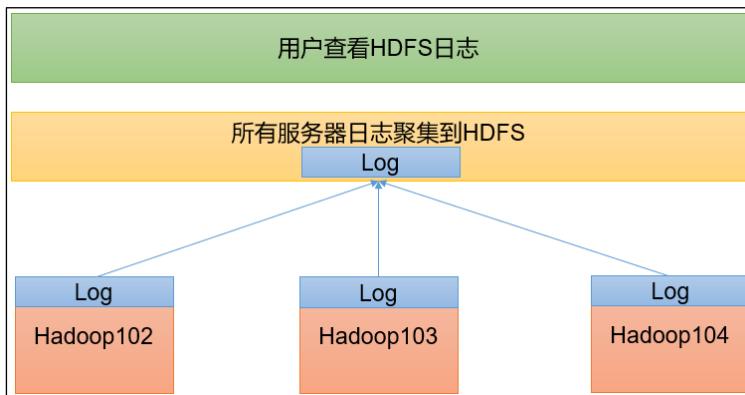
```
[atguigu@hadoop102 hadoop]$ jps
```

5) 查看 JobHistory

<http://hadoop102:19888/jobhistory>

3.2.7 配置日志的聚集

日志聚集概念：应用运行完成以后，将程序运行日志信息上传到 HDFS 系统上。



日志聚集功能好处：可以方便的查看到程序运行详情，方便开发调试。

注意：开启日志聚集功能，需要重新启动 NodeManager 、 ResourceManager 和 HistoryServer。

开启日志聚集功能具体步骤如下：

1) 配置 yarn-site.xml

```
[atguigu@hadoop102 hadoop]$ vim yarn-site.xml
```

在该文件里面增加如下配置。

```

<!-- 开启日志聚集功能 -->
<property>
    <name>yarn.log-aggregation-enable</name>
    <value>true</value>
</property>
<!-- 设置日志聚集服务器地址 -->
<property>
    <name>yarn.log.server.url</name>
    <value>http://hadoop102:19888/jobhistory/logs</value>
</property>
<!-- 设置日志保留时间为 7 天 -->
<property>
    <name>yarn.log-aggregation.retain-seconds</name>
    <value>604800</value>
</property>
  
```

2) 分发配置

```
[atguigu@hadoop102 hadoop]$ xsync $HADOOP_HOME/etc/hadoop/yarn-site.xml
```

3) 关闭 NodeManager 、 ResourceManager 和 HistoryServer

```
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/stop-yarn.sh
[atguigu@hadoop103 hadoop-3.1.3]$ mapred --daemon stop
```

```
historyserver
```

4) 启动 NodeManager、ResourceManager 和 HistoryServer

```
[atguigu@hadoop103 ~]$ start-yarn.sh
[atguigu@hadoop102 ~]$ mapred --daemon start historyserver
```

5) 删除 HDFS 上已经存在的输出文件

```
[atguigu@hadoop102 ~]$ hadoop fs -rm -r /output
```

6) 执行 WordCount 程序

```
[atguigu@hadoop102     .hadoop-3.1.3]$      hadoop      jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar
wordcount /input /output
```

7) 查看日志

(1) 历史服务器地址

<http://hadoop102:19888/jobhistory>

(2) 历史任务列表

Submit Time	Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed	Elapsed Time
2020.12.31 15:34:32 CST	2020.12.31 15:34:36 CST	2020.12.31 15:34:48 CST	job_1609399997152_0001	word count	atguigu	default	SUCCEEDED	1	1	1	1	00hrs, 00mins, 11sec
2020.12.31 15:22:04 CST	2020.12.31 15:22:11 CST	2020.12.31 15:22:21 CST	job_1609398468659_0001	word count	atguigu	default	SUCCEEDED	1	1	1	1	00hrs, 00mins, 10sec

(3) 查看任务运行日志

Attempt Number	Start Time	Node	Logs
1	Thu Dec 31 15:34:34 CST 2020	hadoop103:8042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

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

(4) 运行日志详情

```
← C ▲ 不安全 | hadoop102:19888/jobhistory/logs/hadoop103:38591/container_1609399997152_0001_01_000001/job_1609399997152_0001/atguigu

 hadoop

Log Type: directory.info
Log Upload Time: 星期四 十二月 31 15:34:56 +0800 2020
Log Length: 2320
ls -l:
total 36
-rw-r--r-- 1 atguigu atguigu 105 Dec 31 15:34 container_tokens
-rwxr--r-- 1 atguigu atguigu 770 Dec 31 15:34 default_container_executor.sh
-rw-r--r-- 1 atguigu atguigu 121 Dec 31 15:34 default_container_executor_session.sh
lrwxrwxrwx 1 atguigu atguigu 121 Dec 31 15:34 job.jar -> /opt/atguigu/hadoop-3.1.3/data/en-local-dir/usercache/atguigu/appcache/application_1609399997152_0001/filecache/11/job.jar
lrwxrwxrwx 1 atguigu atguigu 121 Dec 31 15:34 job.xml -> /opt/atguigu/hadoop-3.1.3/data/en-local-dir/usercache/atguigu/appcache/application_1609399997152_0001/filecache/13/job.xml
drwxr-xr-x 2 atguigu atguigu 4096 Dec 31 15:34 jobSubmitDir
-rwxr--r-- 1 atguigu atguigu 5216 Dec 31 15:34 launch_container.sh
drwxr-xr-x 2 atguigu atguigu 4096 Dec 31 15:34 tmp
find . -name depth 5 -ls:
2439862 4 -rwxr--r-- 1 atguigu atguigu 4096 Dec 31 15:34
2439877 4 -drwxr-xr-x 2 atguigu atguigu 4096 Dec 31 15:34 ./jobSubmitDir
2439844 4 -rwxr--r-- 1 atguigu atguigu 45 Dec 31 15:34 ./jobSubmitDir/job_splittemaininfo
2439951 4 -rwxr--r-- 1 atguigu atguigu 105 Dec 31 15:34 ./jobSplitDir/job_split
2439868 4 -rwxr--r-- 1 atguigu atguigu 92 Dec 31 15:34 ./launch_container.sh
2439869 4 -rwxr--r-- 1 atguigu atguigu 12 Dec 31 15:34 ./launch_container.shrc
2439895 4 -rwxr--r-- 1 atguigu atguigu 105 Dec 31 15:34 ./container_tokens
2439871 4 -rwxr--r-- 1 atguigu atguigu 778 Dec 31 15:34 ./default_container_executor.sh
2439867 8 -rwxr--r-- 1 atguigu atguigu 5216 Dec 31 15:34 ./launch_container.sh
2439870 4 -rwxr--r-- 1 atguigu atguigu 16 Dec 31 15:34 ./default_container_executor_session.sh
2439864 1 -rwxr--r-- 1 atguigu atguigu 723 Dec 31 15:34 ./default_container_executor_session.sh
2439854 180 -rwxr--r-- 1 atguigu atguigu 185396 Dec 31 15:34 ./job.xml
2439889 4 -rwxr--r-- 1 atguigu atguigu 1697 Dec 31 15:34 ./job.jar
2439849 312 -rwxr--r-- 1 atguigu atguigu 316382 Dec 31 15:34 ./job.jar/job.jar
2439864 4 -drwxr-xr-x 2 atguigu atguigu 4096 Dec 31 15:34 ./imp
2439872 4 -rwxr--r-- 1 atguigu atguigu 16 Dec 31 15:34 ./default_container_executor.shrc
broken symbolic((find . -name depth 5 -type f -izc):
-mdepth 5 -type f -izc):
```

3.2.8 集群启动/停止方式总结

1) 各个模块分开启/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

2) 各个服务组件逐一启动/停止

(1) 分别启动/停止 HDFS 组件

```
hdfs --daemon start/stop namenode/datanode/secondarynamenode
```

(2) 启动/停止 YARN

```
yarn --daemon start/stop resourcemanager/nodemanager
```

3.2.9 编写 Hadoop 集群常用脚本

1) Hadoop 集群启停脚本（包含 HDFS, Yarn, Historyserver）： myhadoop.sh

```
[atguigu@hadoop102 ~]$ cd /home/atguigu/bin  
[atguigu@hadoop102 bin]$ vim myhadoop.sh
```

➤ 输入如下内容

```
#!/bin/bash

if [ $# -lt 1 ]
then
    echo "No Args Input..."
    exit ;
fi

case $1 in
"start")
    echo " ===== 启动 hadoop 集群 ====="
    echo " ----- 启动 hdfs -----"
    ssh hadoop102 "/opt/module/hadoop-3.1.3/sbin/start-dfs.sh"
    echo " ----- 启动 yarn -----"
```

```

ssh hadoop103 "/opt/module/hadoop-3.1.3/sbin/start-yarn.sh"
echo " ----- 启动 historyserver -----"
ssh hadoop102 "/opt/module/hadoop-3.1.3/bin/mapred --daemon start
historyserver"
;;
"stop")
    echo " ===== 关闭 hadoop 集群 ====="

    echo " ----- 关闭 historyserver -----"
    ssh hadoop102 "/opt/module/hadoop-3.1.3/bin/mapred --daemon stop
historyserver"
    echo " ----- 关闭 yarn -----"
    ssh hadoop103 "/opt/module/hadoop-3.1.3/sbin/stop-yarn.sh"
    echo " ----- 关闭 hdfs -----"
    ssh hadoop102 "/opt/module/hadoop-3.1.3/sbin/stop-dfs.sh"
;;
*)
    echo "Input Args Error..."
;;
esac

```

➤ 保存后退出，然后赋予脚本执行权限

```
[atguigu@hadoop102 bin]$ chmod +x myhadoop.sh
```

2) 查看三台服务器 Java 进程脚本: jpsall

```

[atguigu@hadoop102 ~]$ cd /home/atguigu/bin
[atguigu@hadoop102 bin]$ vim jpsall

```

➤ 输入如下内容

```

#!/bin/bash

for host in hadoop102 hadoop103 hadoop104
do
    echo ===== $host =====
    ssh $host jps
done

```

➤ 保存后退出，然后赋予脚本执行权限

```
[atguigu@hadoop102 bin]$ chmod +x jpsall
```

3) 分发/home/atguigu/bin 目录，保证自定义脚本在三台机器上都可以使用

```
[atguigu@hadoop102 ~]$ xsync /home/atguigu/bin/
```

3.2.10 常用端口号说明

端口名称	Hadoop2.x	Hadoop3.x
NameNode 内部通信端口	8020 / 9000	8020 / 9000/9820
NameNode HTTP UI	50070	9870
MapReduce 查看执行任务端口	8088	8088
历史服务器通信端口	19888	19888

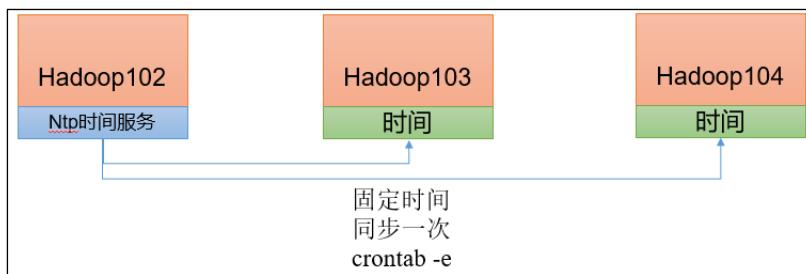
3.2.11 集群时间同步

如果服务器在公网环境（能连接外网），可以不采用集群时间同步，因为服务器会定期和公网时间进行校准；

如果服务器在内网环境，必须要配置集群时间同步，否则时间久了，会产生时间偏差，导致集群执行任务时间不同步。

1) 需求

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，生产环境根据任务对时间的准确程度要求周期同步。测试环境为了尽快看到效果，采用 1 分钟同步一次。



2) 时间服务器配置（必须 root 用户）

(1) 查看所有节点 ntpd 服务状态和开机自启动状态

```
[atguigu@hadoop102 ~]$ sudo systemctl status ntpd
[atguigu@hadoop102 ~]$ sudo systemctl start ntpd
[atguigu@hadoop102 ~]$ sudo systemctl is-enabled ntpd
```

(2) 修改 hadoop102 的 ntp.conf 配置文件

```
[atguigu@hadoop102 ~]$ sudo vim /etc/ntp.conf
```

修改内容如下

(a) 修改 1 (授权 192.168.10.0-192.168.10.255 网段上的所有机器可以从这台机器上查询和同步时间)

```
#restrict 192.168.10.0 mask 255.255.255.0 nomodify notrap
```

为 `restrict 192.168.10.0 mask 255.255.255.0 nomodify notrap`

(b) 修改 2 (集群在局域网中，不使用其他互联网上的时间)

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

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 (当该节点丢失网络连接, 依然可以采用本地时间作为时间服务器为集群中的其他节点提供时间同步)

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

(3) 修改 hadoop102 的/etc/sysconfig/ntp 服务

```
[atguigu@hadoop102 ~]$ sudo vim /etc/sysconfig/ntp
```

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 ntp 服务

```
[atguigu@hadoop102 ~]$ sudo systemctl start ntpd
```

(5) 设置 ntp 服务开机启动

```
[atguigu@hadoop102 ~]$ sudo systemctl enable ntpd
```

3) 其他机器配置 (必须 root 用户)

(1) 关闭所有节点上 ntp 服务和自启动

```
[atguigu@hadoop103 ~]$ sudo systemctl stop ntpd  
[atguigu@hadoop103 ~]$ sudo systemctl disable ntpd  
[atguigu@hadoop104 ~]$ sudo systemctl stop ntpd  
[atguigu@hadoop104 ~]$ sudo systemctl disable ntpd
```

(2) 在其他机器配置 1 分钟与时间服务器同步一次

```
[atguigu@hadoop103 ~]$ sudo crontab -e
```

编写定时任务如下:

```
* /1 * * * * /usr/sbin/ntpdate hadoop102
```

(3) 修改任意机器时间

```
[atguigu@hadoop103 ~]$ sudo date -s "2021-9-11 11:11:11"
```

(4) 1 分钟后查看机器是否与时间服务器同步

```
[atguigu@hadoop103 ~]$ sudo date
```

第 4 章 常见错误及解决方案

1) 防火墙没关闭、或者没有启动 YARN

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) 不识别主机名称

```

java.net.UnknownHostException: hadoop102
        at
java.net.InetAddress.getLocalHost (InetAddress.java:1475)
        at
org.apache.hadoop.mapreduce.JobSubmitter.submitJobInternal (Job
Submitter.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)

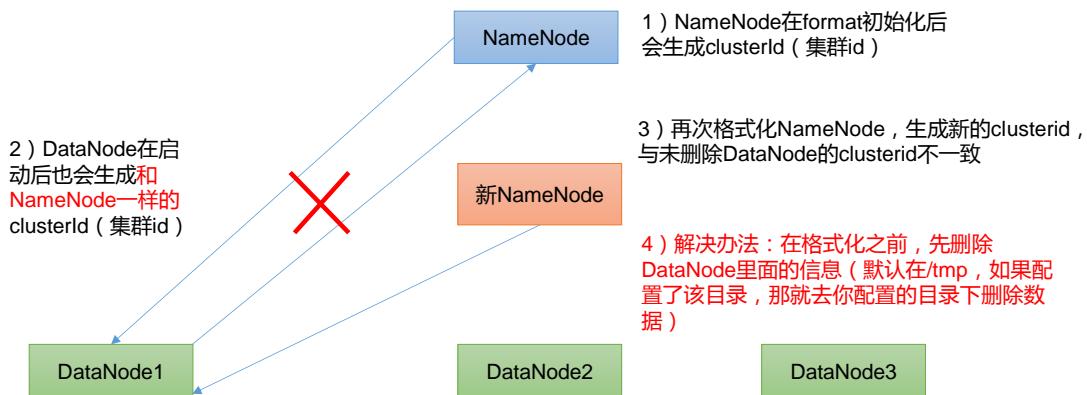
```

解决办法：

- (1) 在/etc/hosts 文件中添加 192.168.10.102 hadoop102
- (2) 主机名称不要起 hadoop hadoop000 等特殊名称
- 8) DataNode 和 NameNode 进程同时只能工作一个。



DataNode和NameNode进程同时只能有一个工作问题分析



让天下没有难学的技术

- 9) 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。

- 10) 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。

- 11) jps 不生效

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

- 12) 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
```

尚硅谷大数据技术之 Hadoop (HDFS)

(作者: 尚硅谷大数据研发部)

版本: V3.3

第 1 章 HDFS 概述

1.1 HDFS 产生背景及定义

1) HDFS 产生背景

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

2) HDFS 定义

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

HDFS 的使用场景：适合一次写入，多次读出的场景。一个文件经过创建、写入和关闭之后就不需要改变。

1.2 HDFS 优缺点

HDFS优点



1) 高容错性

- 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。



- 某一个副本丢失以后，它可以自动恢复。



2) 适合处理大数据

- 数据规模：能够处理数据规模达到GB、TB、甚至PB级别的数据；
- 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) 可构建在廉价机器上，通过多副本机制，提高可靠性。

让天下没有难学的技术

 HDFS缺点

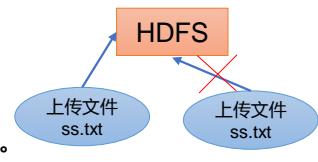
1) 不适合低延时数据访问 , 比如毫秒级的存储数据 , 是做不到的。

2) 无法高效的对大量小文件进行存储。

- 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 ;
- 小文件存储的寻址时间会超过读取时间 , 它违反了HDFS的设计目标。

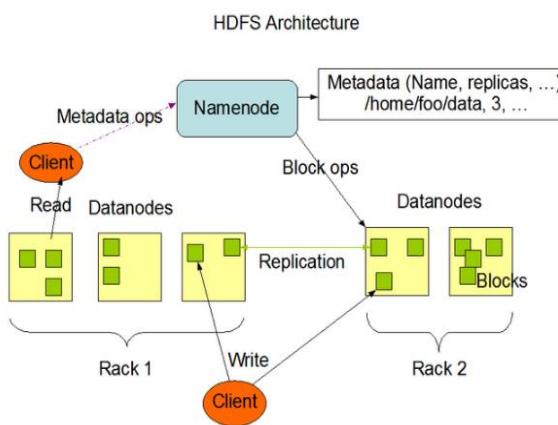
3) 不支持并发写入、文件随机修改。

- 一个文件只能有一个写 , 不允许多个线程同时写 ;
- 仅支持数据append (追加) , 不支持文件的随机修改。



让天下没有难学的技术

1.3 HDFS 组成架构

 HDFS组成架构


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

- (1) 管理HDFS的名称空间；
- (2) 配置副本策略；
- (3) 管理数据块 (Block) 映射信息；
- (4) 处理客户端读写请求。

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

- (1) 存储实际的数据块；
- (2) 执行数据块的读/写操作。

让天下没有难学的技术

 HDFS组成架构

3) Client : 就是客户端。

- (1) 文件切分。文件上传HDFS的时候 , Client将文件切分成一个一个的Block , 然后进行上传 ;
- (2) 与NameNode交互 , 获取文件的位置信息 ;
- (3) 与DataNode交互 , 读取或者写入数据 ;
- (4) Client提供一些命令来管理HDFS , 比如NameNode格式化 ;
- (5) Client可以通过一些命令来访问HDFS , 比如对HDFS增删查改操作 ;

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

- (1) 辅助NameNode , 分担其工作量 , 比如定期合并Fsimage和Edits , 并推送给NameNode ;
- (2) 在紧急情况下 , 可辅助恢复NameNode。

让天下没有难学的技术

1.4 HDFS 文件块大小 (面试重点)

 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/3.x版本中是128M , 1.x版本中是64M。

2) 如果寻址时间为10ms , 即查找到目标block的时间为10ms。

3) 寻址时间为传输时间的1%时 , 则为最佳状态。 (专家)
因此 , 传输时间
 $=10\text{ms}/0.01=1000\text{ms}=1\text{s}$

4) 而目前磁盘的传输速率普遍为100MB/s。

1) 集群中的block

block1

block2

blockn

$$\begin{aligned} \text{5 block大小} \\ =1\text{s} * 100\text{MB/s} = 100\text{MB} \end{aligned}$$

让天下没有难学的技术



思考：为什么块的大小不能设置太小，也不能设置太大？

(1) HDFS的块设置**太小**，会增加寻址时间，程序一直在找块的开始位置；

(2) 如果块设置的**太大**，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**。导致程序在处理这块数据时，会非常慢。

总结：HDFS块的大小设置主要取决于磁盘传输速率。

让天下没有难学的技术

第 2 章 HDFS 的 Shell 操作（开发重点）

2.1 基本语法

hadoop fs 具体命令 OR hdfs dfs 具体命令

两个是完全相同的。

2.2 命令大全

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hadoop fs  
[-appendToFile <localsrc> ... <dst>]  
    [-cat [-ignoreCrc] <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>]  
    [-df [-h] [<path> ...]]  
    [-du [-s] [-h] <path> ...]  
    [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]  
    [-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>]  
    [-rm [-f] [-r|-R] [-skipTrash] <src> ...]  
    [-rmdir [--ignore-fail-on-non-empty] <dir> ...]  
<acl_spec> <path>]]  
    [-setrep [-R] [-w] <rep> <path> ...]
```

```
[-stat [format] <path> ...]
[-tail [-f] <file>]
[-test -[defsz] <path>]
[-text [-ignoreCrc] <src> ...]
```

2.3 常用命令实操

2.3.1 准备工作

- 1) 启动 Hadoop 集群（方便后续的测试）

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

- 2) -help: 输出这个命令参数

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -help rm
```

- 3) 创建/sanguo 文件夹

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mkdir /sanguo
```

2.3.2 上传

- 1) -moveFromLocal: 从本地剪切粘贴到 HDFS

```
[atguigu@hadoop102 hadoop-3.1.3]$ vim shuguo.txt
输入:
shuguo

[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -moveFromLocal ./shuguo.txt
/sanguo
```

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

```
[atguigu@hadoop102 hadoop-3.1.3]$ vim weiguo.txt
输入:
weiguo

[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -copyFromLocal weiguo.txt
/sanguo
```

- 3) -put: 等同于 copyFromLocal, 生产环境更习惯用 put

```
[atguigu@hadoop102 hadoop-3.1.3]$ vim wuguo.txt
输入:
wuguo

[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -put ./wuguo.txt /sanguo
```

- 4) -appendToFile: 追加一个文件到已经存在的文件末尾

```
[atguigu@hadoop102 hadoop-3.1.3]$ vim liubei.txt
输入:
liubei

[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -appendToFile liubei.txt
/sanguo/shuguo.txt
```

2.3.3 下载

- 1) -copyToLocal: 从 HDFS 拷贝到本地

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -copyToLocal  
/sanguo/shuguo.txt .
```

2) -get: 等同于 copyToLocal, 生产环境更习惯用 get

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -get  
/sanguo/shuguo.txt ./shuguo2.txt
```

2.3.4 HDFS 直接操作

1) -ls: 显示目录信息

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls /sanguo
```

2) -cat: 显示文件内容

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -cat /sanguo/shuguo.txt
```

3) -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-3.1.3]$ hadoop fs -chmod 666  
/sanguo/shuguo.txt  
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -chown atguigu:atguigu  
/sanguo/shuguo.txt
```

4) -mkdir: 创建路径

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mkdir /jinguo
```

5) -cp: 从 HDFS 的一个路径拷贝到 HDFS 的另一个路径

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -cp /sanguo/shuguo.txt  
/jinguo
```

6) -mv: 在 HDFS 目录中移动文件

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mv /sanguo/wuguo.txt /jinguo  
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mv /sanguo/weiguo.txt  
/jinguo
```

7) -tail: 显示一个文件的末尾 1kb 的数据

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -tail /jinguo/shuguo.txt
```

8) -rm: 删除文件或文件夹

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -rm /sanguo/shuguo.txt
```

9) -rm -r: 递归删除目录及目录里面内容

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -rm -r /sanguo
```

10) -du 统计文件夹的大小信息

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -du -s -h /jinguo  
27 81 /jinguo  
  
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -du -h /jinguo  
14 42 /jinguo/shuguo.txt  
7 21 /jinguo/weiguo.txt  
6 18 /jinguo/wuguo.txt
```

说明: 27 表示文件大小; 81 表示 27*3 个副本; /jinguo 表示查看的目录

11) -setrep: 设置 HDFS 中文件的副本数量

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -setrep 10 /jinguo/shuguo.txt
```

□	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	⋮
□	-rw-r--r--	atguigu	supergroup	14 B	Jan 05 14:04	10	128 MB	shuguo.txt	☰

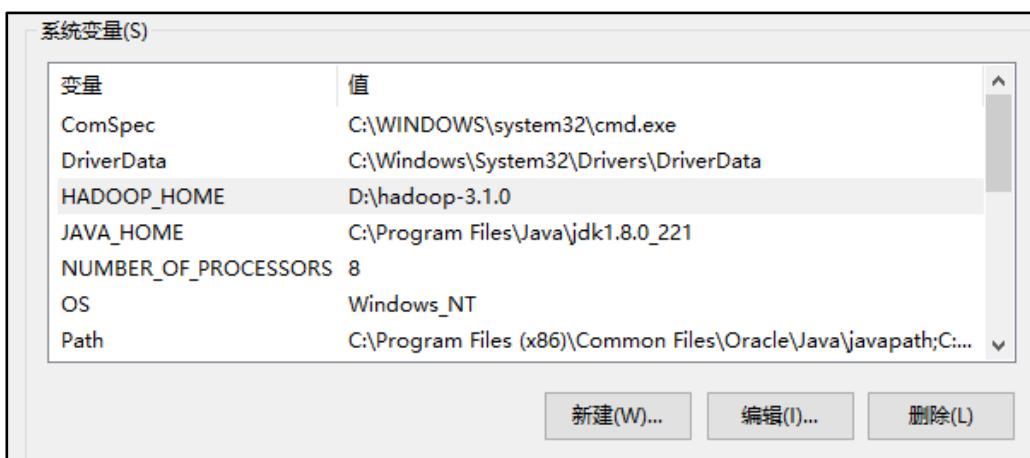
这里设置的副本数只是记录在 NameNode 的元数据中，是否真的会有这么多副本，还得看 DataNode 的数量。因为目前只有 3 台设备，最多也就 3 个副本，只有节点数的增加到 10 台时，副本数才能达到 10。

第 3 章 HDFS 的 API 操作

3.1 客户端环境准备

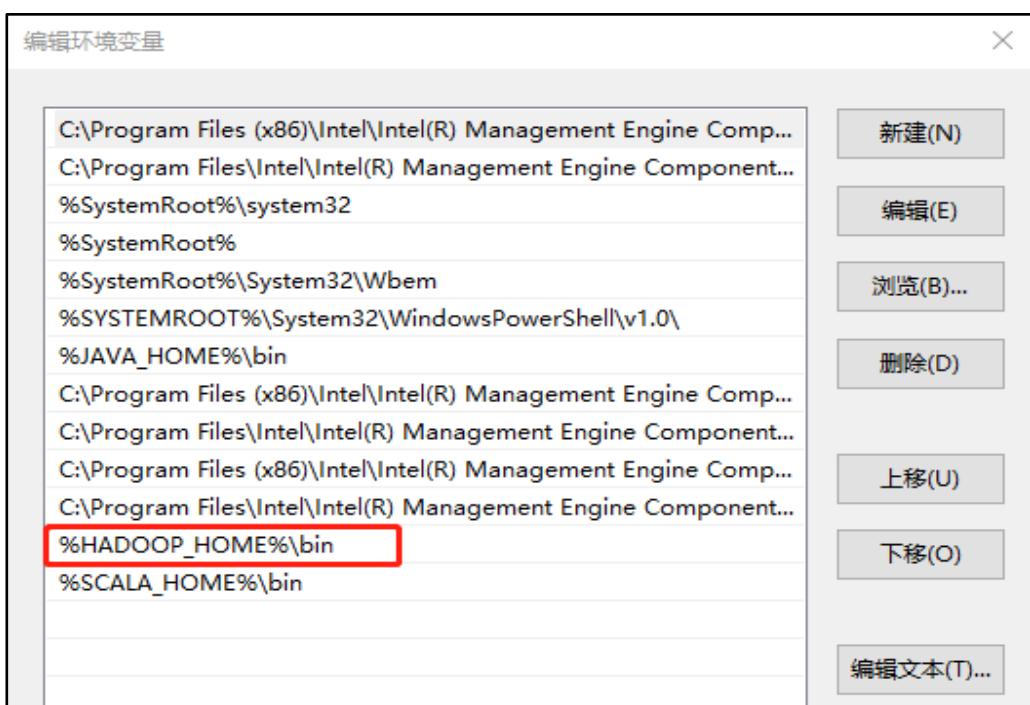
1) 找到资料包路径下的 Windows 依赖文件夹，拷贝 hadoop-3.1.0 到非中文路径（比如 d:\）。

2) 配置 HADOOP_HOME 环境变量



3) 配置 Path 环境变量。

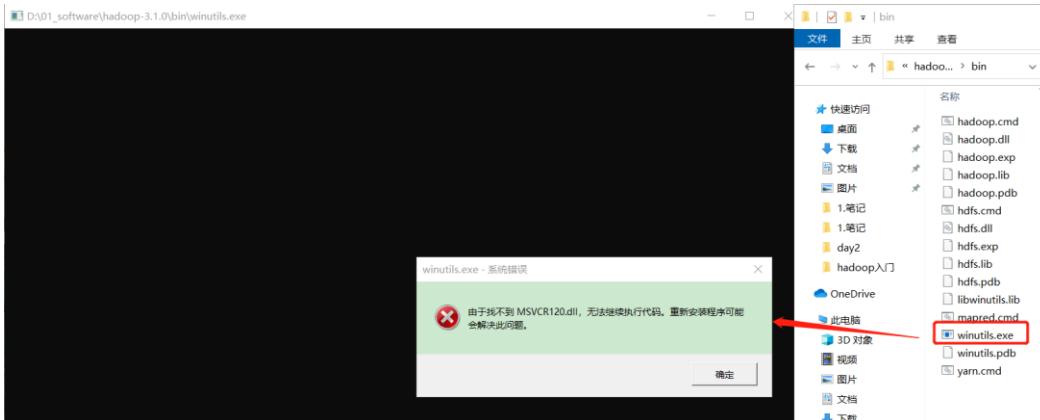
注意：如果环境变量不起作用，可以重启电脑试试。



验证 Hadoop 环境变量是否正常。双击 winutils.exe，如果报如下错误。说明缺少微软运

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

行库(正版系统往往有这个问题)。再资料包里面有对应的微软运行库安装包双击安装即可。



4) 在 IDEA 中创建一个 Maven 工程 HdfsClientDemo，并导入相应的依赖坐标+日志添加

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.30</version>
    </dependency>
</dependencies>
```

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
```

5) 创建包名： com.atguigu.hdfs

6) 创建 HdfsClient 类

```
public class HdfsClient {
    @Test
    public void testMkdirs() throws IOException, URISyntaxException,
    InterruptedException {
        // 1 获取文件系统
    }
}
```

```

        Configuration configuration = new Configuration();

        // FileSystem fs = FileSystem.get(new
URI("hdfs://hadoop102:8020"), configuration);
        FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:8020"),
configuration, "atguigu");

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

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

```

7) 执行程序

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 会从采用 Windows 默认用户访问 HDFS，会报权限异常错误。所以在访问 HDFS 时，一定要配置用户。

```

org.apache.hadoop.security.AccessControlException: Permission denied:
user=56576, access=WRITE,
inode="/xiyou/huaguoshan":atguigu:supergroup:drwxr-xr-x

```

3.2 HDFS 的 API 案例实操

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://hadoop102:8020"),
configuration, "atguigu");

    // 2 上传文件
    fs.copyFromLocalFile(new Path("d:/sunwukong.txt"), new
Path("/xiyou/huaguoshan"));

    // 3 关闭资源
    fs.close();
}

```

2) 将 hdfs-site.xml 拷贝到项目的 resources 资源目录下

```

<?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)然后是服务器的自定义配置(xxx-site.xml) >(4)服务器的默认配置(xxx-default.xml)

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://hadoop102:8020"),
        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("/xiyou/huaguoshan/sunwukong.txt"), new Path("d:/sunwukong2.txt"),
    true);

    // 3 关闭资源
    fs.close();
}
```

注意：如果执行上面代码，下载不了文件，有可能是你电脑的微软支持的运行库少，需要安装一下微软运行库。

3.2.3 HDFS 文件更名和移动

```
@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://hadoop102:8020"),
        configuration, "atguigu");

    // 2 修改文件名称
    fs.rename(new Path("/xiyou/huaguoshan/sunwukong.txt"), new
    Path("/xiyou/huaguoshan/meihouwang.txt"));

    // 3 关闭资源
    fs.close();
}
```

3.2.4 HDFS 删除文件和目录

```
@Test
public void testDelete() throws IOException, InterruptedException,
URIException{

    // 1 获取文件系统
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:8020"),
configuration, "atguigu");

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

    // 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,
URIException {

    // 1 获取文件系统
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:8020"),
configuration, "atguigu");

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

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

        System.out.println("======" + fileStatus.getPath() + "=====");
        System.out.println(fileStatus.getPermission());
        System.out.println(fileStatus.getOwner());
        System.out.println(fileStatus.getGroup());
        System.out.println(fileStatus.getLen());
        System.out.println(fileStatus.getModificationTime());
        System.out.println(fileStatus.getReplication());
        System.out.println(fileStatus.getBlockSize());
        System.out.println(fileStatus.getPath().getName());

        // 获取块信息
        BlockLocation[] blockLocations = fileStatus.getBlockLocations();
        System.out.println(Arrays.toString(blockLocations));
    }

    // 3 关闭资源
    fs.close();
}
```

3.2.6 HDFS 文件和文件夹判断

```

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

    // 1 获取文件配置信息
    Configuration configuration = new Configuration();
    FileSystem fs = FileSystem.get(new URI("hdfs://hadoop102:8020"),
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());
        }
    }

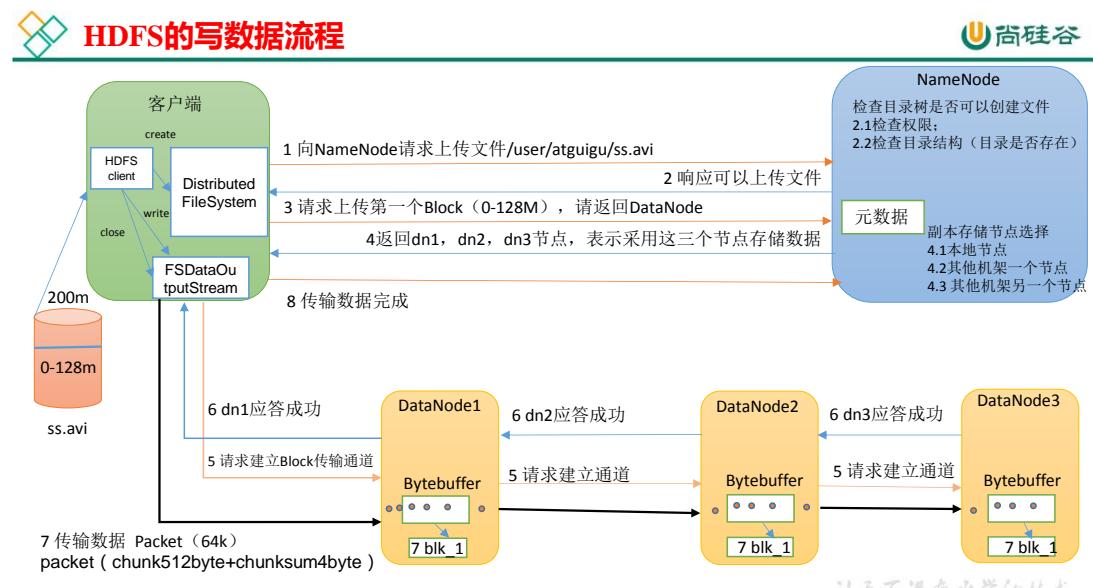
    // 3 关闭资源
    fs.close();
}

```

第 4 章 HDFS 的读写流程（面试重点）

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

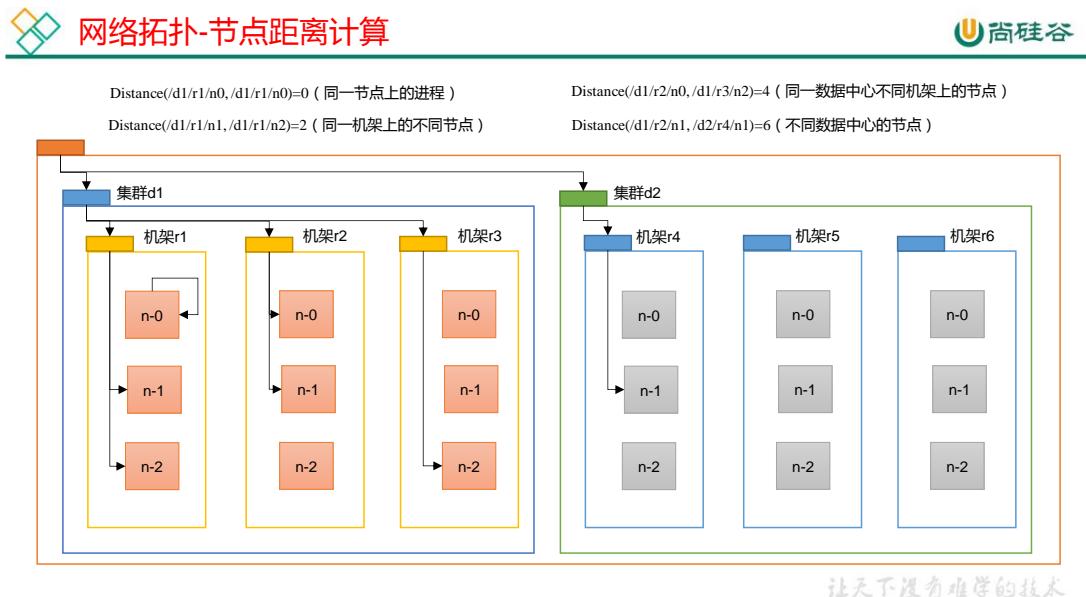
查目标文件是否已存在，父目录是否存在。

- (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-节点距离计算

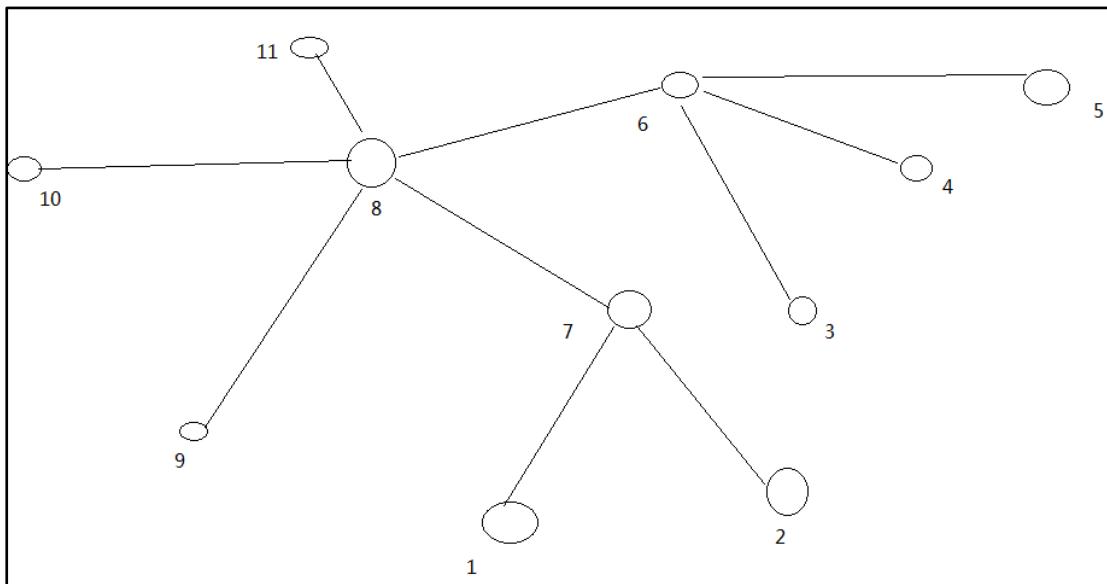
在 HDFS 写数据的过程中，NameNode 会选择距离待上传数据最近距离的 DataNode 接收数据。那么这个最近距离怎么计算呢？

节点距离：两个节点到达最近的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（副本存储节点选择）

1) 机架感知说明

(1) 官方说明

http://hadoop.apache.org/docs/r3.1.3/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Data_Replication

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on the local machine if the writer is on a datanode, otherwise on a random datanode, another replica on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure; this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.

(2) 源码说明

Crtl + n 查找 BlockPlacementPolicyDefault，在该类中查找 chooseTargetInOrder 方法。

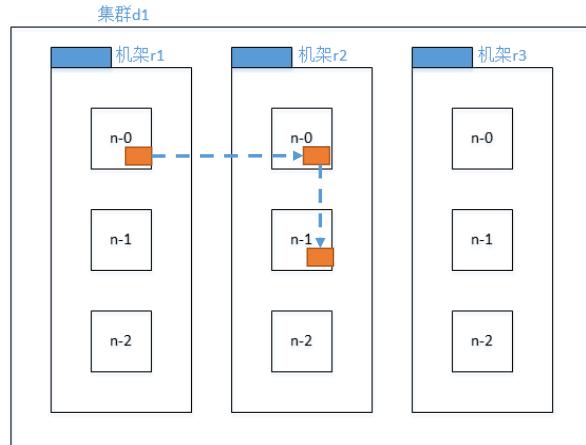
2) Hadoop3.1.3 副本节点选择

Hadoop3.1.3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。

第二个副本在另一个机架的随机
一个节点

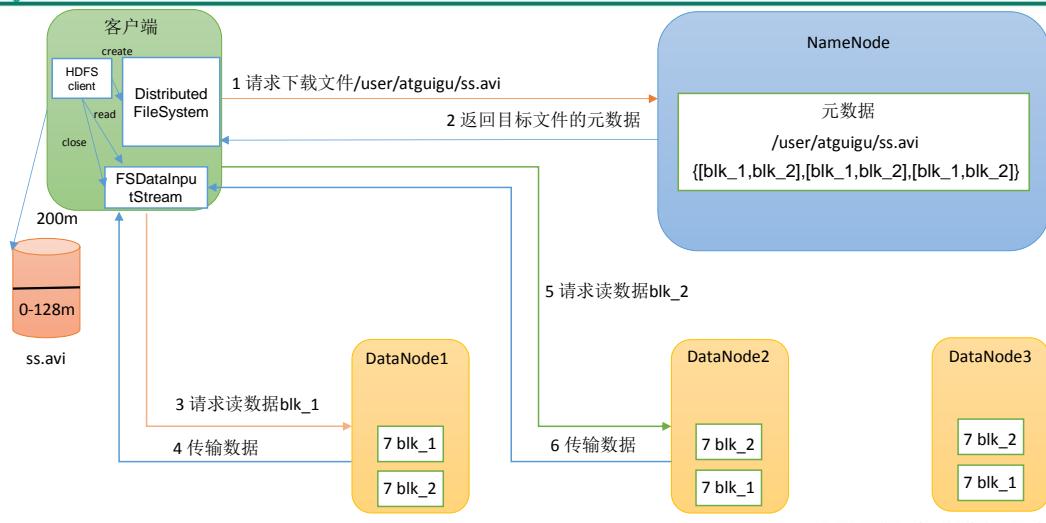
第三个副本在第二个副本所在机架的
随机节点



让天下没有难学的技术

4.2 HDFS 读数据流程

HDFS的读数据流程



让天下没有难学的技术

- (1) 客户端通过 DistributedFileSystem 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。

第 5 章 NameNode 和 SecondaryNameNode

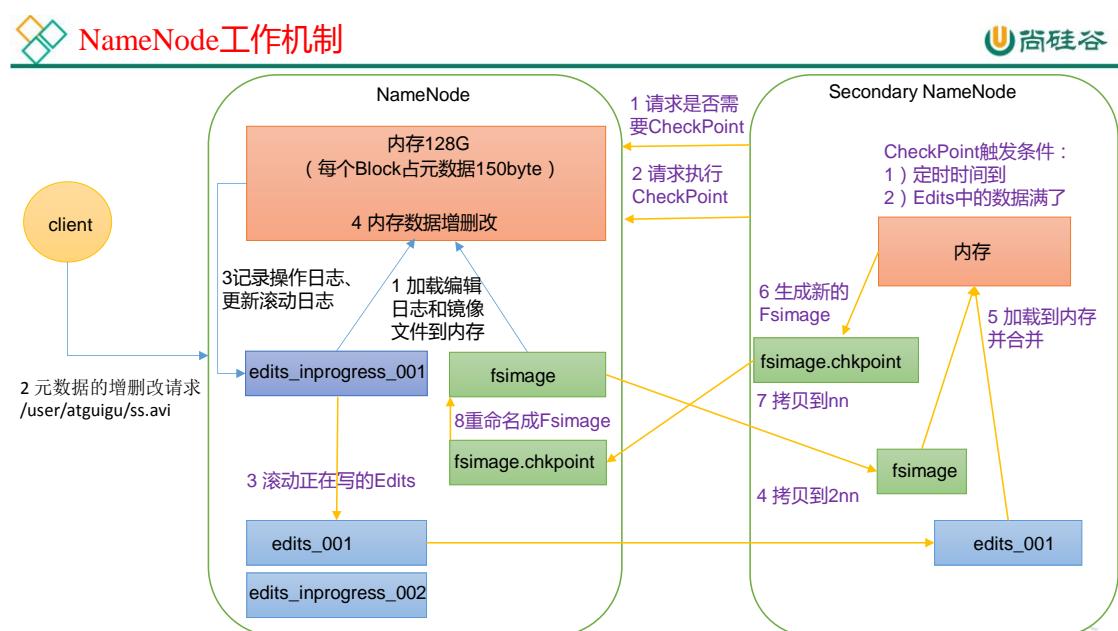
5.1 NN 和 2NN 工作机制

思考：NameNode 中的元数据是存储在哪里的？

首先，我们做个假设，如果存储在 NameNode 节点的磁盘中，因为经常需要进行随机访问，还有响应客户请求，必然是效率过低。因此，元数据需要存放在内存中。但如果只存在内存中，一旦断电，元数据丢失，整个集群就无法工作了。**因此产生在磁盘中备份元数据的 FsImage。**

这样又会带来新的问题，当在内存中的元数据更新时，如果同时更新 FsImage，就会导致效率过低，但如果不行，就会发生一致性问题，一旦 NameNode 节点断电，就会产生数据丢失。因此，引入 Edits 文件（只进行追加操作，效率很高）。每当元数据有更新或者添加元数据时，修改内存中的元数据并追加到 Edits 中。这样，一旦 NameNode 节点断电，可以通过 FsImage 和 Edits 的合并，合成元数据。

但是，如果长时间添加数据到 Edits 中，会导致该文件数据过大，效率降低，而且一旦断电，恢复元数据需要的时间过长。因此，需要定期进行 FsImage 和 Edits 的合并，如果这个操作由 NameNode 节点完成，又会效率过低。**因此，引入一个新的节点 SecondaryNamenode，专门用于 FsImage 和 Edits 的合并。**



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

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

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 在内存中对元数据进行增删改。

2) 第二阶段：Secondary NameNode 工作

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

5.2 Fsimage 和 Edits 解析

Fsimage和Edits概念



NameNode被格式化之后，将在/opt/module/hadoop-3.1.3/data/tmp/dfs/name/current目录中产生如下文件

```
fsimage_000000000000000000000000
fsimage_000000000000000000000000.md5
seen_txid
VERSION
```

(1) Fsimage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inode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，加载Edits里面的更新操作，保证内存中的元数据信息是最新的、同步的，可以看成NameNode启动的时候就将Fsimage和Edits文件进行了合并。

让天下没有难学的技术

1) 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-3.1.3/data/dfs/name/current  
  
[atguigu@hadoop102 current]$ hdfs oiv -p XML -i  
fsimage_0000000000000000025 -o /opt/module/hadoop-3.1.3/fsimage.xml  
  
[atguigu@hadoop102 current]$ cat /opt/module/hadoop-3.1.3/fsimage.xml
```

将显示的 xml 文件内容拷贝到 Idea 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>151272284477</mtime>  
  <permission>atguigu:supergroup:rwxr-xr-x</permission>  
  <nquota>-1</nquota>  
  <dquota>-1</dquota>  
</inode>  
<inode>  
  <id>16387</id>  
  <type>DIRECTORY</type>  
  <name>atguigu</name>  
  <mtime>1512790549080</mtime>  
  <permission>atguigu:supergroup:rwxr-xr-x</permission>  
  <nquota>-1</nquota>  
  <dquota>-1</dquota>  
</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--r--</permission>  
  <blocks>  
    <block>  
      <id>1073741825</id>  
      <genstamp>1001</genstamp>  
      <numBytes>59</numBytes>  
    </block>  
  </blocks>  
</inode >
```

思考：可以看出，Fsimage 中没有记录块所对应 DataNode，为什么？

在集群启动后，要求 DataNode 上报数据块信息，并间隔一段时间后再次上报。

2) 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-3.1.3/edits.xml

[atguigu@hadoop102 current]$ cat /opt/module/hadoop-3.1.3/edits.xml
```

将显示的 xml 文件内容拷贝到 Idea 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>192.168.10.102</CLIENT_MACHINE>
            <OVERWRITE>true</OVERWRITE>
            <PERMISSION_STATUS>
                <USERNAME>atguigu</USERNAME>
                <GROUPNAME>supergroup</GROUPNAME>
                <MODE>420</MODE>
            </PERMISSION_STATUS>
            <RPC_CLIENTID>908eaf4-9aec-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>
</EDITS>
```

```

        </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 >
```

思考：NameNode 如何确定下次开机启动的时候合并哪些 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>3600s</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>
```

```

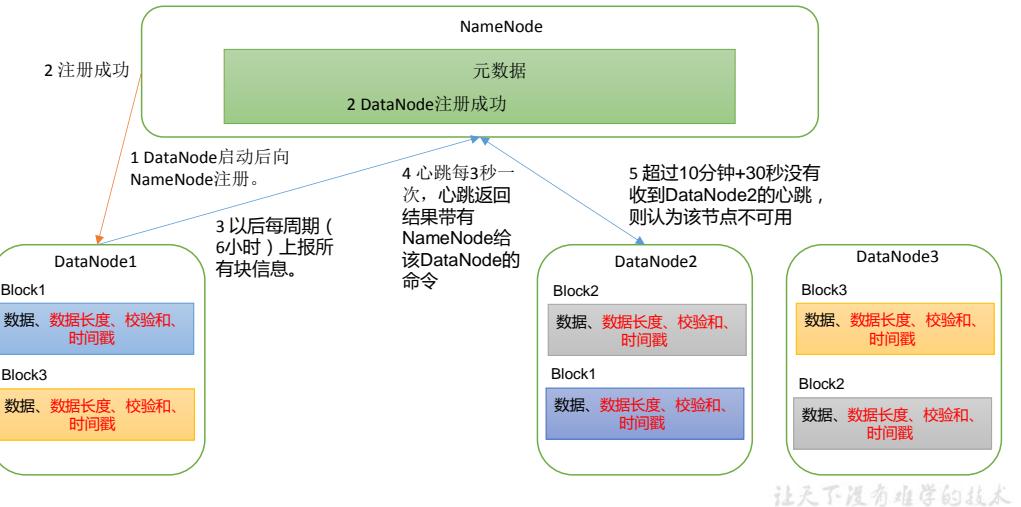
<name>dfs.namenode.checkpoint.txns</name>
<value>1000000</value>
<description>操作动作次数</description>
</property>

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

```

第 6 章 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（6 小时）的向 NameNode 上报所有的块信息。

DN 向 NN 汇报当前解读信息的时间间隔，默认 6 小时；

```

<property>
  <name>dfs.blockreport.intervalMsec</name>
  <value>21600000</value>
  <description>Determines block reporting interval in
milliseconds.</description>
</property>

```

DN 扫描自己节点块信息列表的时间，默认 6 小时

```

<property>
  <name>dfs.datanode.directoryscan.interval</name>
  <value>21600s</value>
  <description>Interval in seconds for Datanode to scan data
blocks</description>
</property>

```

```

directories and reconcile the difference between blocks in memory and on
the disk.

Support multiple time unit suffix(case insensitive), as described
in dfs.heartbeat.interval.

</description>
</property>

```

(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。

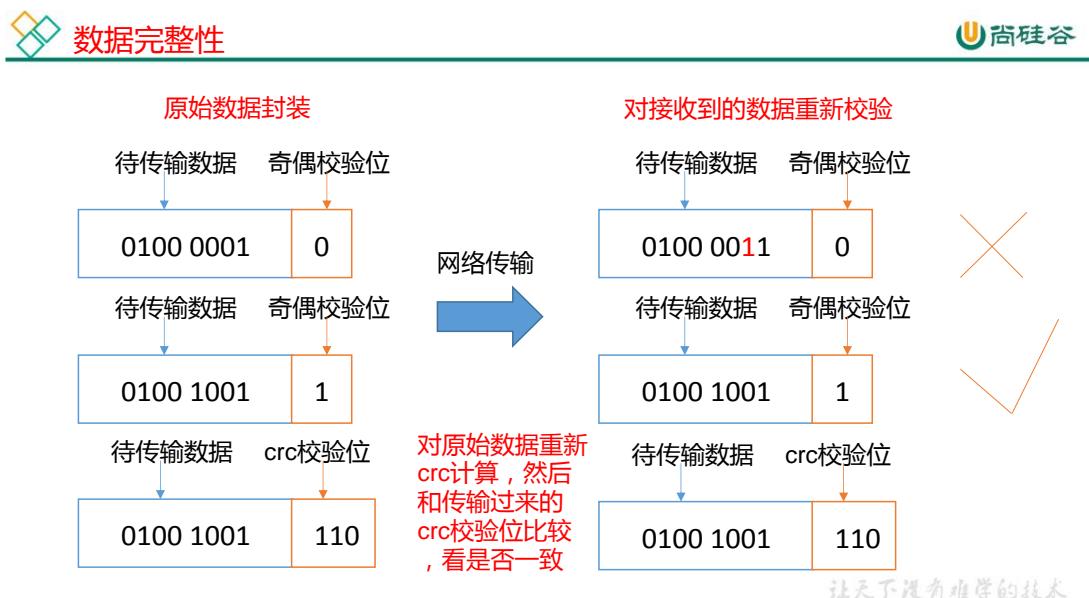
(4) 集群运行中可以安全加入和退出一些机器。

6.2 数据完整性

思考：如果电脑磁盘里面存储的数据是控制高铁信号灯的红灯信号(1)和绿灯信号(0)，但是存储该数据的磁盘坏了，一直显示是绿灯，是否很危险？同理 DataNode 节点上的数据损坏了，却没有发现，是否也很危险，那么如何解决呢？

如下是 DataNode 节点保证数据完整性的方法。

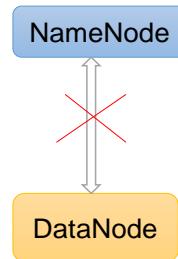
- (1) 当 DataNode 读取 Block 的时候，它会计算 CheckSum。
- (2) 如果计算后的 CheckSum，与 Block 创建时值不一样，说明 Block 已经损坏。
- (3) Client 读取其他 DataNode 上的 Block。
- (4) 常见的校验算法 crc (32) , md5 (128) , sha1 (160)
- (5) DataNode 在其文件创建后周期验证 CheckSum。



6.3 掉线时限参数设置

◆ DataNode掉线时限参数设置

1、DataNode进程死亡或者网络故障造成DataNode无法与NameNode通信



2、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。

3、HDFS默认的超时时长为10分钟+30秒。

4、如果定义超时时间为TimeOut，则超时时长的计算公式为：

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

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

让天下没有难学的技术

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>
```

尚硅谷大数据技术之 Hadoop (MapReduce)

(作者：尚硅谷大数据研发部)

版本：V3.3

第 1 章 MapReduce 概述

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

可以实现上千台服务器集群并发工作，提供数据处理能力。

1.2.2 缺点

1) 不擅长实时计算

MapReduce 无法像 MySQL 一样，在毫秒或者秒级内返回结果。

2) 不擅长流式计算

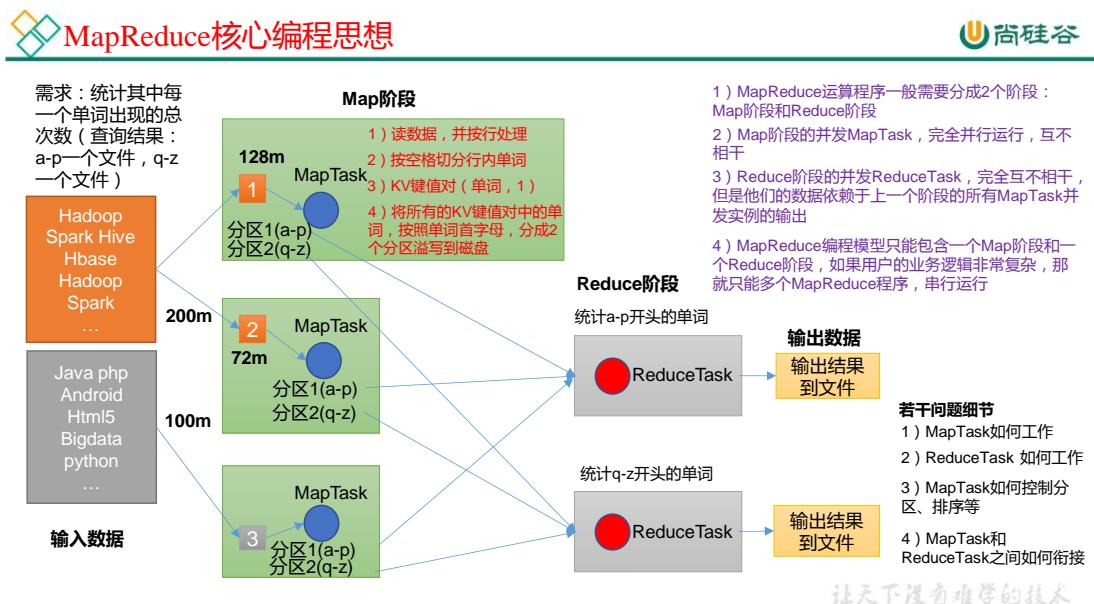
流式计算的输入数据是动态的，而 MapReduce 的输入数据集是静态的，不能动态变化。

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) 不擅长 DAG (有向无环图) 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 MapTask 并发实例，完全并行运行，互不相干。
- (3) 第二个阶段的 ReduceTask 并发实例互不相干，但是他们的数据依赖于上一个阶段的所有 MapTask 并发实例的输出。
- (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。

总结：分析 WordCount 数据流走向深入理解 MapReduce 核心思想。

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。

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- (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 官方 WordCount 源码

采用反编译工具反编译源码，发现 WordCount 案例有 Map 类、Reduce 类和驱动类。且数据的类型是 Hadoop 自身封装的序列化类型。

1.6 常用数据序列化类型

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
Null	NullWritable

1.7 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

让天下没有难学的技术

2. Reducer阶段

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

3. Driver阶段

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

让天下没有难学的技术

1.8 WordCount 案例实操

1.8.1 本地测试

1) 需求

在给定的文本文件中统计输出每一个单词出现的总次数

(1) 输入数据



hello.txt

(2) 期望输出数据

```
atguigu 2
banzhang 1
cls 2
hadoop 1
jiao 1
ss 2
xue 1
```

2) 需求分析

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)



1、输入数据

```
atguigu atguigu
ss ss
cls cls
jiao
banzhang
xue
hadoop
```

2、输出数据

```
atguigu 2
banzhang1
cls 2
hadoop 1
jiao 1
ss 2
xue 1
```

3、Mapper

```
// 3.1 将MapTask传给我们的文本  
内容先转换成String
```

```
atguigu atguigu
```

```
// 3.2 根据空格将这一行切分成单词
```

```
atguigu  
atguigu
```

```
// 3.3 将单词输出为<单词，1>
```

```
atguigu, 1  
atguigu, 1
```

4、Reducer

```
// 4.1 汇总各个key的个数
```

```
atguigu, 1  
atguigu, 1
```

```
// 4.2 输出该key的总次数
```

```
atguigu, 2
```

5、Driver

```
// 5.1 获取配置信息，获取job对象实例
```

```
// 5.2 指定本程序的jar包所在的本地路径
```

```
// 5.3 关联Mapper/Reducer业务类
```

```
// 5.4 指定Mapper输出数据的kv类型
```

```
// 5.5 指定最终输出的数据的kv类型
```

```
// 5.6 指定job的输入原始文件所在目录
```

```
// 5.7 指定job的输出结果所在目录
```

```
// 5.8 提交作业
```

3) 环境准备

(1) 创建 maven 工程， MapReduceDemo

(2) 在 pom.xml 文件中添加如下依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.30</version>
    </dependency>
</dependencies>
```

(2) 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) 创建包名： com.atguigu.mapreduce.wordcount

4) 编写程序

(1) 编写 Mapper 类

```
package com.atguigu.mapreduce.wordcount;
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;
    IntWritable v = new IntWritable();

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

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

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

(3) 编写 Driver 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 获取配置信息以及获取 job 对象
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 2 关联本 Driver 程序的 jar
        job.setJarByClass(WordCountDriver.class);

        // 3 关联 Mapper 和 Reducer 的 jar
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        // 4 设置 Mapper 输出的 kv 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

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

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

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

5) 本地测试

- (1) 需要首先配置好 HADOOP_HOME 变量以及 Windows 运行依赖
- (2) 在 IDEA/Eclipse 上运行程序

1.8.2 提交到集群测试

集群上测试

- (1) 用 maven 打 jar 包，需要添加的打包插件依赖

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
```

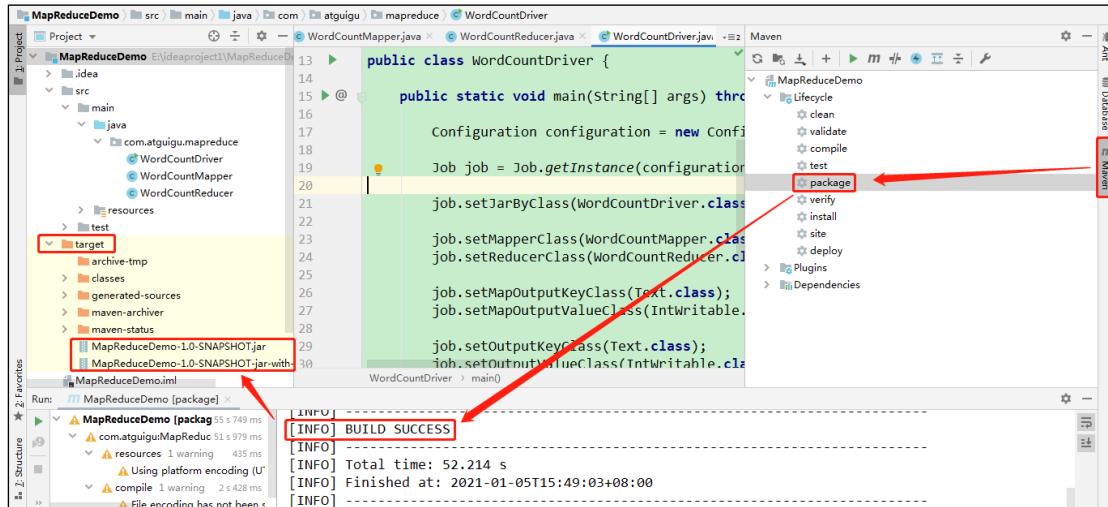
```

<version>3.6.1</version>
<configuration>
    <source>1.8</source>
    <target>1.8</target>
</configuration>
</plugin>
</plugins>
</build>

```

注意：如果工程上显示红叉。在项目上右键->maven->Reimport 刷新即可。

(2) 将程序打成 jar 包



(3) 修改不带依赖的 jar 包名称为 wc.jar，并拷贝该 jar 包到 Hadoop 集群的 /opt/module/hadoop-3.1.3 路径。

(4) 启动 Hadoop 集群

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

(5) 执行 WordCount 程序

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar wc.jar
com.atguigu.mapreduce.wordcount.WordCountDriver      /user/atguigu/input
/user/atguigu/output
```

第 2 章 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）以便于存储到磁盘（持久化）和网络传输。

反序列化就是将收到字节序列（或其他数据传输协议）或者是磁盘的持久化数据，转换成内存中的对象。

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）。

4) Hadoop 序列化特点：

- (1) 紧凑：高效使用存储空间。
- (2) 快速：读写数据的额外开销小。
- (3) 互操作：支持多语言的交互

2.2 自定义 bean 对象实现序列化接口（Writable）

在企业开发中往往常用的基本序列化类型不能满足所有需求，比如在 Hadoop 框架内部传递一个 bean 对象，那么该对象就需要实现序列化接口。

具体实现 bean 对象序列化步骤如下 7 步。

(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 过程要求对 key 必须能排序。详见后面排序案例。

```

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

```

2.3 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、总下行流量、总流量

(1) 输入数据



phone_data.txt

(2) 输入数据格式:

7	13560436666	120.196.100.99	1116	954	200
id	手机号码	网络 ip	上行流量	下行流量	网络状态码

(3) 期望输出数据格式

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

2) 需求分析



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、输入数据格式

7	13560436666	120.196.100.99	1116	954	200
Id	手机号码	网络ip	上行流量	下行流量	网络状态码

4、Map阶段

(1) 读取一行数据，切分字段

7	13560436666	120.196.100.99	1116	954	200
---	-------------	----------------	------	-----	-----

(2) 抽取手机号、上行流量、下行流量

13560436666	1116	954
手机号码	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);

(4) bean对象要想能够传输，必须实现序列化接口

3、期望输出数据格式

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

5、Reduce阶段

(1) 累加上行流量和下行流量得到总流量。

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

让天下没有难学的技术

3) 编写 MapReduce 程序

(1) 编写流量统计的 Bean 对象

```
package com.atguigu.mapreduce.writable;

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

//1 继承 Writable 接口
public class FlowBean implements Writable {

    private long upFlow; //上行流量
    private long downFlow; //下行流量
    private long sumFlow; //总流量

    //2 提供无参构造
    public FlowBean() {
    }

    //3 提供三个参数的 getter 和 setter 方法
    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;
}

public void setSumFlow() {
    this.sumFlow = this.upFlow + this.downFlow;
}

//4 实现序列化和反序列化方法,注意顺序一定要保持一致
@Override
public void write(DataOutput dataOutput) throws IOException {
    dataOutput.writeLong(upFlow);
    dataOutput.writeLong(downFlow);
    dataOutput.writeLong(sumFlow);
}

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

//5 重写 ToString
@Override
public String toString() {
    return upFlow + "\t" + downFlow + "\t" + sumFlow;
}
}
```

(2) 编写 Mapper 类

```
package com.atguigu.mapreduce.writable;

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

public class FlowMapper extends Mapper<LongWritable, Text, Text, FlowBean> {
    private Text outK = new Text();
    private FlowBean outV = new FlowBean();

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

        //1 获取一行数据,转成字符串
        String line = value.toString();

        //2 切割数据
        String[] split = line.split("\t");

        //3 抓取我们需要的数据:手机号,上行流量,下行流量
        String phone = split[1];
        String up = split[split.length - 3];
        String down = split[split.length - 2];

        //4 封装 outK outV
        outK.set(phone);
    }
}
```

```
        outV.setUpFlow(Long.parseLong(up));
        outV.setDownFlow(Long.parseLong(down));
        outV.setSumFlow();

        //5 写出 outK outV
        context.write(outK, outV);
    }
}
```

(3) 编写 Reducer 类

```
package com.atguigu.mapreduce.writable;

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

public class FlowReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
    private FlowBean outV = new FlowBean();
    @Override
    protected void reduce(Text key, Iterable<FlowBean> values, Context context) throws IOException, InterruptedException {

        long totalUp = 0;
        long totalDown = 0;

        //1 遍历 values, 将其中的上行流量, 下行流量分别累加
        for (FlowBean flowBean : values) {
            totalUp += flowBean.getUpFlow();
            totalDown += flowBean.getDownFlow();
        }

        //2 封装 outKV
        outVsetUpFlow(totalUp);
        outV.setDownFlow(totalDown);
        outV.setSumFlow();

        //3 写出 outK outV
        context.write(key, outV);
    }
}
```

(4) 编写 Driver 驱动类

```
package com.atguigu.mapreduce.writable;

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;
import java.io.IOException;

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

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

        //2 关联本 Driver 类
    }
}
```

```

        job.setJarByClass(FlowDriver.class);

        //3 关联 Mapper 和 Reducer
        job.setMapperClass(FlowMapper.class);
        job.setReducerClass(FlowReducer.class);

        //4 设置 Map 端输出 KV 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

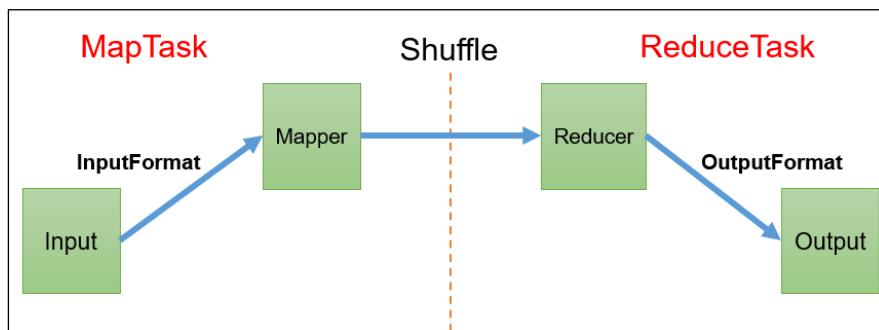
        //5 设置程序最终输出的 KV 类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        //6 设置程序的输入输出路径
        FileInputFormat.setInputPaths(job, new Path("D:\\\\inputflow"));
        FileOutputFormat.setOutputPath(job, new Path("D:\\\\flowoutput"));

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

```

第 3 章 MapReduce 框架原理



3.1 InputFormat 数据输入

3.1.1 切片与 MapTask 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。

思考：1G 的数据，启动 8 个 MapTask，可以提高集群的并发处理能力。那么 1K 的数据，也启动 8 个 MapTask，会提高集群性能吗？MapTask 并行任务是否越多越好呢？哪些因素影响了 MapTask 并行度？

2) MapTask 并行度决定机制

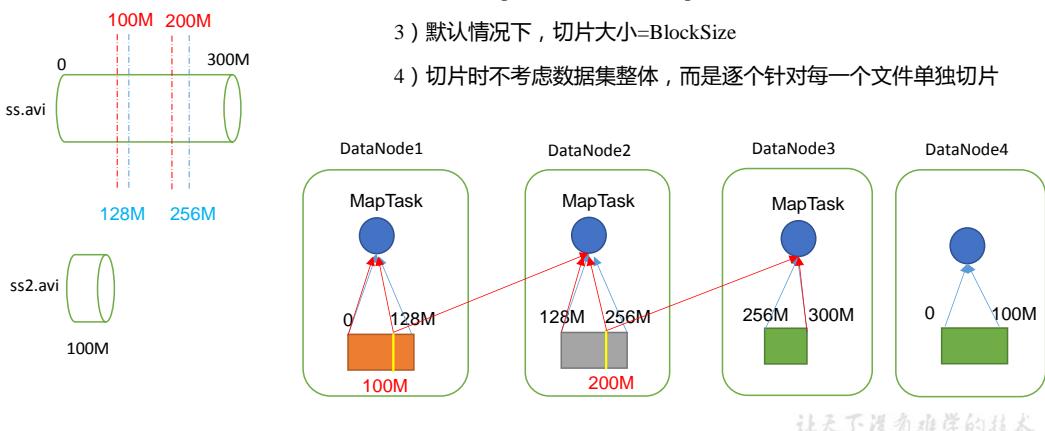
数据块：Block 是 HDFS 物理上把数据分成一块一块。**数据块是 HDFS 存储数据单位。**

数据切片：数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。**数据切片是 MapReduce 程序计算输入数据的单位**，一个切片会对应启动一个 MapTask。


数据切片与MapTask并行度决定机制

1、假设切片大小设置为100M

2、假设切片大小设置为128M



让天下没有难学的技术

3.1.2 Job 提交流程源码和切片源码详解

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) 判断是本地运行环境还是 yarn 集群运行环境
        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);

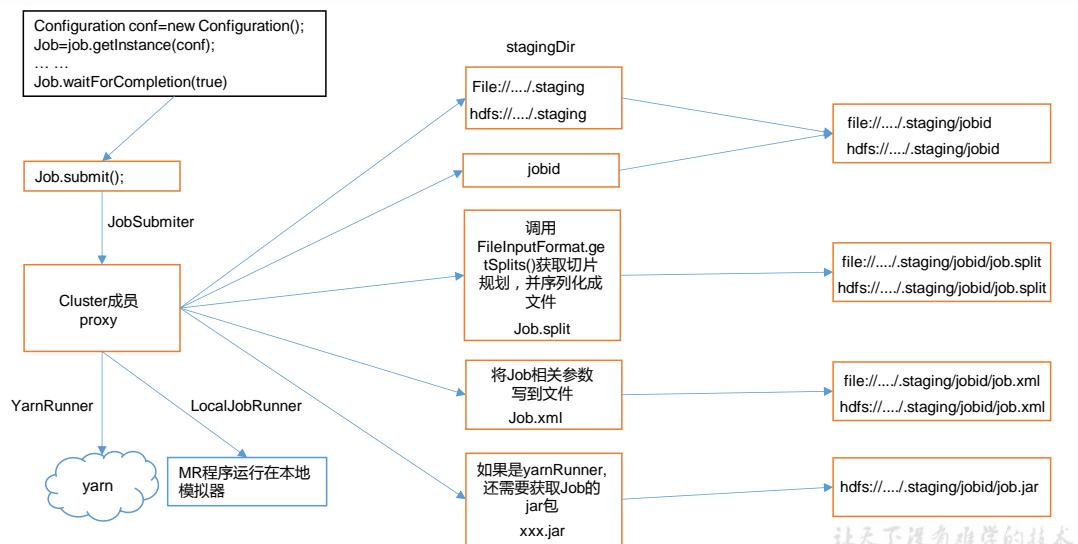
    // 5) 向 Stag 路径写 XML 配置文件
    writeConf(conf, submitJobFile);
    conf.writeXml(out);

    // 6) 提交 Job, 返回提交状态
    status = submitClient.submitJob(jobId, submitJobDir.toString(),

```

```
job.getCredentials() );
```

Job提交流程源码解析



让天下没有难学的技术

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

FileInputFormat切片源码解析

- (1) 程序先找到你数据存储的目录。
- (2) 开始遍历处理 (规划切片) 目录下的每一个文件
- (3) 遍历第一个文件ss.txt
 - a) 获取文件大小`fs.sizeOf(ss.txt)`
 - b) 计算切片大小
`computeSplitSize(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) **InputSplit只记录了切片的元数据信息**，比如起始位置、长度以及所在的节点列表等。
- (4) 提交切片规划文件到YARN上，YARN上的MrAppMaster就可以根据切片规划文件计算开启MapTask个数。

让天下没有难学的技术

3.1.3 FileInputFormat 切片机制



1、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

2、案例分析

(1) 输入数据有两个文件：

file1.txt	320M
file2.txt	10M

(2) 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

让天下没有难学的技术



(1) 源码中计算切片大小的公式

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

(2) 切片大小设置

maxsize (切片最大值)：参数如果调得比blockSize小，则会让切片变小，而且就等于配置的这个参数的值。
minsize (切片最小值)：参数调得比blockSize大，则可以让切片变得比blockSize还大。

(3) 获取切片信息API

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

让天下没有难学的技术

3.1.4 TextInputFormat

1) FileInputFormat 实现类

思考：在运行 MapReduce 程序时，输入的文件格式包括：基于行的日志文件、二进制格式文件、数据库表等。那么，针对不同的数据类型，MapReduce 是如何读取这些数据的呢？

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

2) TextInputFormat

TextInputFormat 是默认的 FileInputFormat 实现类。按行读取每条记录。键是存储该行在更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

整个文件中的起始字节偏移量， LongWritable 类型。值是这行的内容，不包括任何行终止符（换行符和回车符）， Text 类型。

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)  
(20, Intelligent learning engine)  
(49, Learning more convenient)  
(74, From the real demand for more close to the enterprise)
```

3.1.5 CombineTextInputFormat 切片机制

框架默认的 TextInputFormat 切片机制是对任务按文件规划切片，**不管文件多小，都会是一个单独的切片**，都会交给一个 MapTask，这样如果有大量小文件，**就会产生大量的 MapTask**，处理效率极其低下。

1) 应用场景：

CombineTextInputFormat 用于小文件过多的场景，它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个 MapTask 处理。

2) 虚拟存储切片最大值设置

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

注意：虚拟存储切片最大值设置最好根据实际的小文件大小情况来设置具体的值。

3) 切片机制

生成切片过程包括：虚拟存储过程和切片过程二部分。



CombineTextInputFormat切片机制



setMaxInputSplitSize值为4M

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

让天下没有难学的技术

(1) 虚拟存储过程:

将输入目录下所有文件大小，依次和设置的 `setMaxInputSplitSize` 值比较，如果不大于设置的最大值，逻辑上划分一个块。如果输入文件大于设置的最大值且大于两倍，那么以最大值切割一块；当剩余数据大小超过设置的最大值且不大于最大值 2 倍，此时将文件均分成 2 个虚拟存储块（防止出现太小切片）。

例如 `setMaxInputSplitSize` 值为 4M，输入文件大小为 8.02M，则先逻辑上分成一个 4M。剩余的大小为 4.02M，如果按照 4M 逻辑划分，就会出现 0.02M 的小的虚拟存储文件，所以将剩余的 4.02M 文件切分成 (2.01M 和 2.01M) 两个文件。

(2) 切片过程:

(a) 判断虚拟存储的文件大小是否大于 `setMaxInputSplitSize` 值，大于等于则单独形成一个切片。

(b) 如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。

(c) 测试举例：有 4 个小文件大小分别为 1.7M、5.1M、3.4M 以及 6.8M 这四个小文件，则虚拟存储之后形成 6 个文件块，大小分别为：

1.7M, (2.55M、2.55M), 3.4M 以及 (3.4M、3.4M)

最终会形成 3 个切片，大小分别为：

(1.7+2.55) M, (2.55+3.4) M, (3.4+3.4) M

3.1.6 CombineTextInputFormat 案例实操

1) 需求

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

将输入的大量小文件合并成一个切片统一处理。

(1) 输入数据

准备 4 个小文件



(2) 期望

期望一个切片处理 4 个文件

2) 实现过程

(1) 不做任何处理，运行 1.8 节的 WordCount 案例程序，观察切片个数为 4。

```
number of splits:4
```

(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 3。

(a) 驱动类中添加代码如下：

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

(b) 运行如果为 3 个切片。

```
number of splits:3
```

(3) 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。

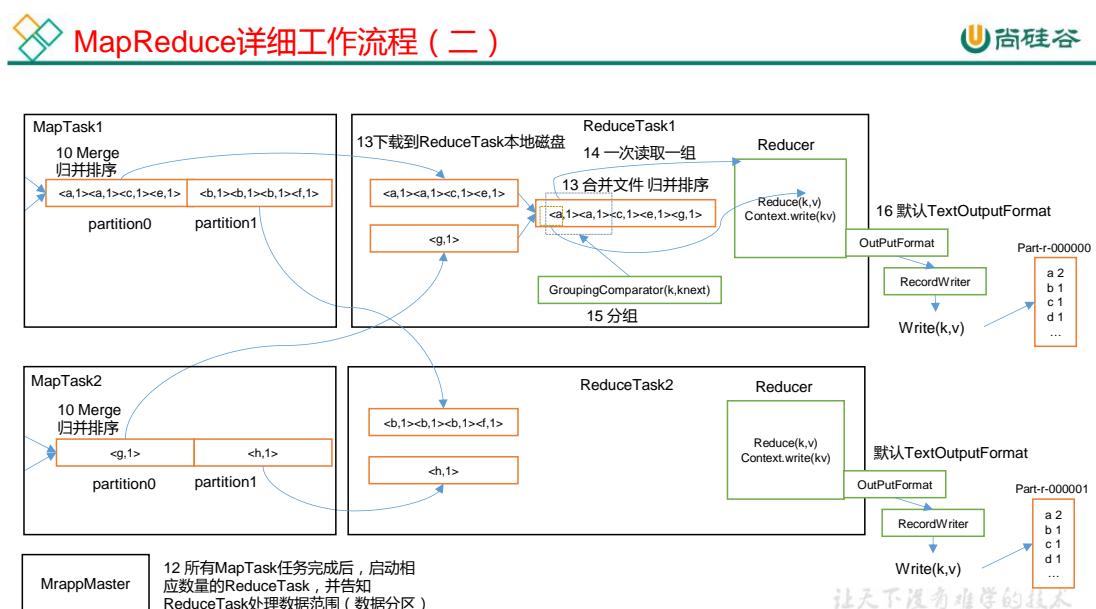
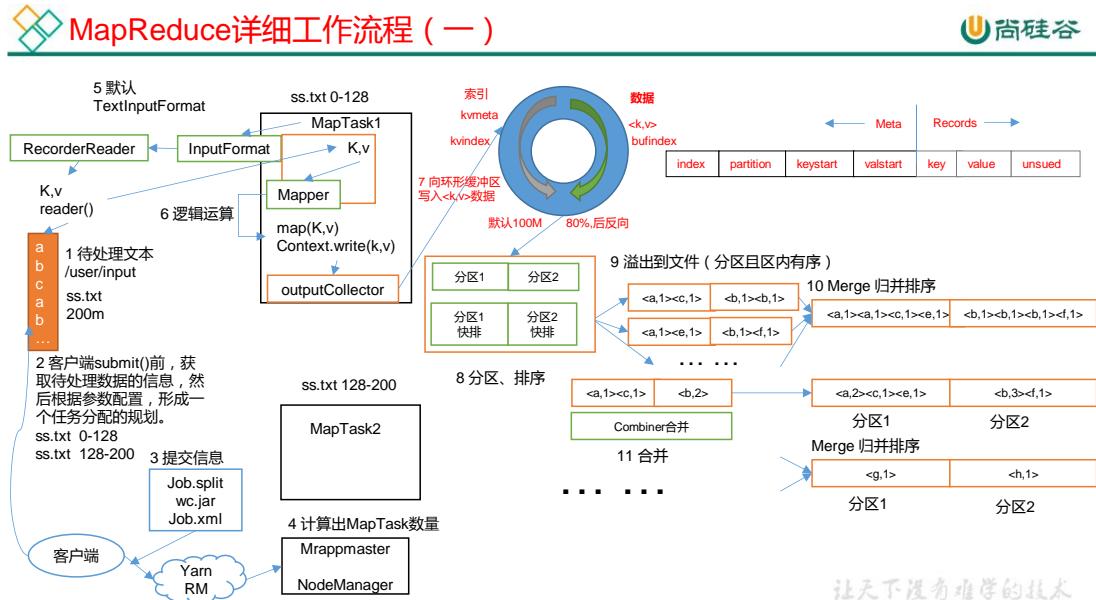
(a) 驱动中添加代码如下：

```
// 如果不设置 InputFormat，它默认用的是 TextInputFormat.class  
job.setInputFormatClass(CombineTextInputFormat.class);  
  
// 虚拟存储切片最大值设置 20m  
CombineTextInputFormat.setMaxInputSplitSize(job, 20971520);
```

(b) 运行如果为 1 个切片

```
number of splits:1
```

3.2 MapReduce 工作流程



上面的流程是整个 MapReduce 最全工作流程，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 会抓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)

注意:

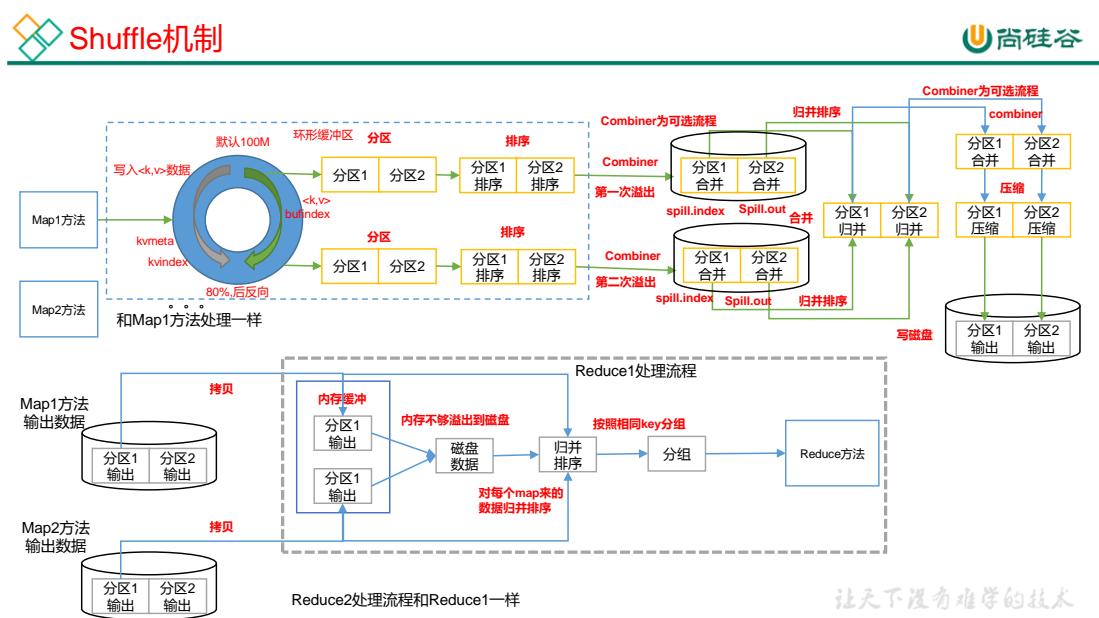
(1) Shuffle 中的缓冲区大小会影响到 MapReduce 程序的执行效率, 原则上说, 缓冲区越大, 磁盘 io 的次数越少, 执行速度就越快。

(2) 缓冲区的大小可以通过参数调整, 参数: mapreduce.task.io.sort.mb 默认 100M。

3.3 Shuffle 机制

3.3.1 Shuffle 机制

Map 方法之后, Reduce 方法之前的数据处理过程称之为 Shuffle。



3.3.2 Partition 分区



1、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）

2、默认Partitioner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。

让天下没有难学的技术



3、自定义Partitioner步骤

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

```
public class CustomPartitioner extends Partitioner<Text, FlowBean> {  
    @Override  
    public int getPartition(Text key, FlowBean value, int numPartitions) {  
        // 控制分区代码逻辑  
        ...  
        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ReduceTask

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

让天下没有难学的技术

 Partition 分区**4、分区总结**

- (1) 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；
- (2) 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；
- (3) 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；
- (4) 分区号必须从零开始，逐一累加。

5、案例分析

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

让天下没有难学的技术

3.3.3 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）

(1) 输入数据



phone_data.txt

(2) 期望输出数据

手机号 136、137、138、139 开头都分别放到一个独立的 4 个文件中，其他开头的放到一个文件中。

2) 需求分析

 Partition 分区案例分析

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、数据输入

13630577991	6960	690	文件1
13736230513	2481	24681	文件2
13846544121	264	0	文件3
13956435636	132	1512	文件4
13560439638	918	4938	文件5

3、期望数据输出

4、增加一个ProvincePartitioner分区

136	分区0
137	分区1
138	分区2
139	分区3
其他	分区4

5、Driver驱动类

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

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

让天下没有难学的技术

3) 在案例 2.3 的基础上，增加一个分区类

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

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

    @Override
    public int getPartition(Text text, FlowBean flowBean, int numPartitions)
    {
        //获取手机号前三位 prePhone
        String phone = text.toString();
        String prePhone = phone.substring(0, 3);

        //定义一个分区号变量 partition,根据 prePhone 设置分区号
        int partition;

        if("136".equals(prePhone)){
            partition = 0;
        }else if("137".equals(prePhone)){
            partition = 1;
        }else if("138".equals(prePhone)){
            partition = 2;
        }else if("139".equals(prePhone)){
            partition = 3;
        }else {
            partition = 4;
        }

        //最后返回分区号 partition
        return partition;
    }
}
```

4) 在驱动函数中增加自定义数据分区设置和 ReduceTask 设置

```
package com.atguigu.mapreduce.partition;
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;
import java.io.IOException;

public class FlowDriver {

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

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

        //2 关联本 Driver 类
        job.setJarByClass(FlowDriver.class);

        //3 关联 Mapper 和 Reducer
        job.setMapperClass(FlowMapper.class);
        job.setReducerClass(FlowReducer.class);

        //4 设置 Map 端输出数据的 KV 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

        //5 设置程序最终输出的 KV 类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        //8 指定自定义分区器
        job.setPartitionerClass(ProvincePartitioner.class);

        //9 同时指定相应数量的 ReduceTask
        job.setNumReduceTasks(5);

        //6 设置输入输出路径
        FileInputFormat.setInputPaths(job, new Path("D:\\inputflow"));
        FileOutputFormat.setOutputPath(job, new Path("D\\partitionout"));

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

3.3.4 WritableComparable 排序



排序是MapReduce框架中最重要的操作之一。

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

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**。

让天下没有难学的技术



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

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

让天下没有难学的技术

(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有序。

(2) 全排序

最终输出结果只有一个文件，且文件内部有序。实现方式是只设置一个ReduceTask。但该方法在处理大型文件时效率极低，因为一台机器处理所有文件，完全丧失了MapReduce所提供的并行架构。

(3) 辅助排序：(GroupingComparator分组)

在Reduce端对key进行分组。应用于：在接收的key为bean对象时，想让一个或几个字段相同（全部字段比较不相同）的key进入到同一个reduce方法时，可以采用分组排序。

(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。

让天下没有难学的技术

自定义排序 WritableComparable 原理分析

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

```
@Override
public int compareTo(FlowBean bean) {

    int result;

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

    return result;
}
```

3.3.5 WritableComparable 排序案例实操（全排序）**1) 需求**

根据案例 2.3 序列化案例产生的结果再次对总流量进行倒序排序。

(1) 输入数据

原始数据



phone_data.txt

第一次处理后的数据

part-r-00000

(2) 期望输出数据

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

2) 需求分析



WritableComparable排序案例分析 (全排序)



1、需求：根据手机的总流量进行倒序排序

2、输入数据

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

3、输出数据

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

4、FlowBean实现WritableComparable接口重写compareTo方法

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

6、Reducer类

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

5、Mapper类

```
context.write(bean, 手机号)
```

让天下没有难学的技术

3) 代码实现

(1) FlowBean 对象在需求 1 基础上增加了比较功能

```
package com.atguigu.mapreduce.writablecomparable;

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

public class FlowBean implements WritableComparable<FlowBean> {

    private long upFlow; //上行流量
    private long downFlow; //下行流量
    private long sumFlow; //总流量

    //提供无参构造
    public FlowBean() {
    }

    //生成三个属性的 getter 和 setter 方法
    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;
}

public void setSumFlow() {
    this.sumFlow = this.upFlow + this.downFlow;
}

//实现序列化和反序列化方法,注意顺序一定要一致
@Override
public void write(DataOutput out) throws IOException {
    out.writeLong(this.upFlow);
    out.writeLong(this.downFlow);
    out.writeLong(this.sumFlow);
}

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

//重写 ToString,最后要输出 FlowBean
@Override
public String toString() {
    return upFlow + "\t" + downFlow + "\t" + sumFlow;
}

@Override
public int compareTo(FlowBean o) {

    //按照总流量比较,倒序排列
    if(this.sumFlow > o.sumFlow) {
        return -1;
    }else if(this.sumFlow < o.sumFlow) {
        return 1;
    }else {
        return 0;
    }
}
}
```

(2) 编写 Mapper 类

```
package com.atguigu.mapreduce.writablecomparable;

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

```

import java.io.IOException;

public class FlowMapper extends Mapper<LongWritable, Text, FlowBean, Text>
{
    private FlowBean outK = new FlowBean();
    private Text outV = new Text();

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

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

        //2 按照"\t",切割数据
        String[] split = line.split("\t");

        //3 封装 outK outV
        outK.setUpFlow(Long.parseLong(split[1]));
        outK.setDownFlow(Long.parseLong(split[2]));
        outK.setSumFlow();
        outV.set(split[0]);

        //4 写出 outK outV
        context.write(outK,outV);
    }
}

```

(3) 编写 Reducer 类

```

package com.atguigu.mapreduce.writablecomparable;

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

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

        //遍历 values 集合,循环写出,避免总流量相同的情况
        for (Text value : values) {
            //调换 KV 位置,反向写出
            context.write(value,key);
        }
    }
}

```

(4) 编写 Driver 类

```

package com.atguigu.mapreduce.writablecomparable;

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;
import java.io.IOException;

public class FlowDriver {

```

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

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

    //2 关联本 Driver 类
    job.setJarByClass(FlowDriver.class);

    //3 关联 Mapper 和 Reducer
    job.setMapperClass(FlowMapper.class);
    job.setReducerClass(FlowReducer.class);

    //4 设置 Map 端输出数据的 KV 类型
    job.setMapOutputKeyClass(FlowBean.class);
    job.setMapOutputValueClass(Text.class);

    //5 设置程序最终输出的 KV 类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FlowBean.class);

    //6 设置输入输出路径
    FileInputFormat.setInputPaths(job, new Path("D:\\\\inputflow2"));
    FileOutputFormat.setOutputPath(job, new Path("D:\\\\comparout"));

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

3.3.6 WritableComparable 排序案例实操（区内排序）

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) 需求分析

基于前一个需求，增加自定义分区类，分区按照省份手机号设置。

 分区内排序案例分析

1、数据输入

```

13509468723 7335    110349    117684
13975057813 11058   48243     59301
13568436656 3597   25635     29232
13736230513 2481   24681     27162
18390173782 9531   2412      11943
13630577991 6960   690       7650
15043685818 3659   3538     7197
13992314666 3008   3720      6728
15910133277 3156   2936      6092
13560439638 918    4938      5856
84188413 4116     1432      5548
13682846555 1938   2910      4848
18271575951 1527   2106      3633
15959002129 1938   180       2118
13590439668 1116   954       2070
13956435636 132    1512      1644
13470253144 180    180       360
13846544121 264    0         264
13966251146 240    0         240
13768778790 120    120      240
13729199489 240    0         240
...
...

```

2、期望数据输出

	part-r-00000	13630577991 6960	690	7650
	13682846555 1938	2910	2910	4848
	13736230513 2481	24681	24681	27162
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13975057813 11058	48243	48243	59301
	13992314666 3008	3720	3720	6728
	13956435636 132	1512	1512	1644
	13966251146 240	0	0	240
	13509468723 7335	110349	110349	117684
	13568436656 3597	25635	25635	29232
	18390173782 9531	2412	2412	11943
	15043685818 3659	3538	3538	7197
	15910133277 3156	2936	2936	6092
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120	120	240
	13729199489 240	0	0	240
	13846544121 264	0	0	264
	13966251146 240	0	0	240
	13768778790 120	120		

```
// 设置对应的 ReduceTask 的个数
job.setNumReduceTasks(5);
```

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

让天下没有难学的技术

(6) 自定义 Combiner 实现步骤

(a) 自定义一个 Combiner 继承 Reducer，重写 Reduce 方法

```
public class WordCountCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {

    private IntWritable outV = new IntWritable();

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

        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }

        outV.set(sum);

        context.write(key, outV);
    }
}
```

(b) 在 Job 驱动类中设置：

```
job.setCombinerClass(WordCountCombiner.class);
```

3.3.8 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。

(1) 数据输入



hello.txt

(2) 期望输出数据

期望: Combine 输入数据多, 输出时经过合并, 输出数据降低。

2) 需求分析



需求: 对每一个MapTask的输出局部汇总 (Combiner)



1、数据输入

banzhang ni hao	<banzhang,4>
xihuan hadoop	<ni,2>
banzhang	<hao,2>
banzhang ni hao	<xihuan,2>
xihuan hadoop	<Hadoop,2>
banzhang	

2、期望输出

Map-Reduce Framework
 Map input records=4
 Map output records=12
 Map output bytes=126
 Map output materialized bytes=66
 Input split bytes=99
 Combine input records=12
 Combine output records=5 使用后
 Reduce input groups=5
 Reduce shuffle bytes=66

方案一

- 1) 增加一个WordcountCombiner类继承Reducer
- 2) 在WordcountCombiner中

- (1) 统计单词汇总
- (2) 将统计结果输出

方案二

- 1) 将 WordcountReducer作为Combiner在 WordcountDriver驱动类中指定

```
job.setCombinerClass(WordcountReducer.class);
```

让天下没有难学的技术

3) 案例实操-方案一

(1) 增加一个 WordCountCombiner 类继承 Reducer

```
package com.atguigu.mapreduce.combiner;

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

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

    private IntWritable outV = new IntWritable();

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

        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }

        //封装 outKV
        outV.set(sum);
    }
}
```

```

        //写出 outKV
        context.write(key,outV);
    }
}

```

(2) 在 WordcountDriver 驱动类中指定 Combiner

```

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

```

4) 案例实操-方案二

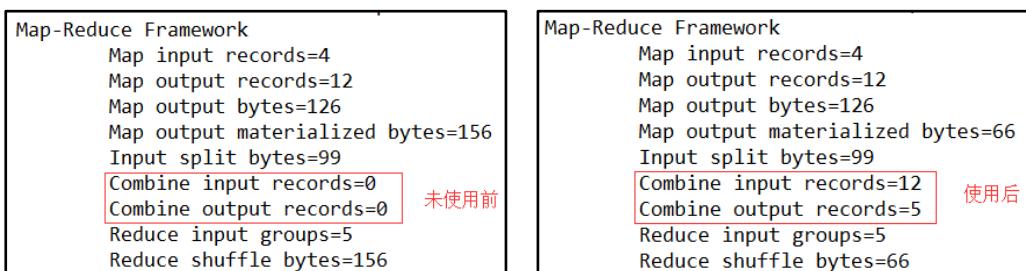
(1) 将 WordcountReducer 作为 Combiner 在 WordcountDriver 驱动类中指定

```

// 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，如下图所示



3.4 OutputFormat 数据输出

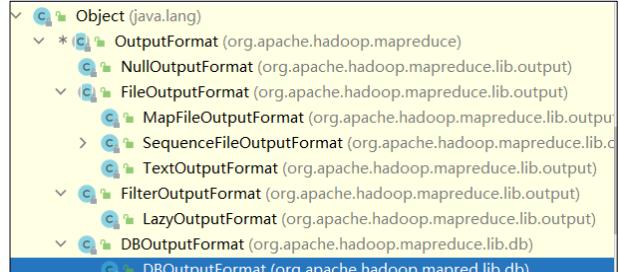
3.4.1 OutputFormat 接口实现类

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. OutputFormat实现类



2. 默认输出格式TextOutputFormat

3. 自定义OutputFormat

3.1 应用场景：

例如：输出数据到MySQL/HBase/Elasticsearch等存储框架中。

3.2 自定义OutputFormat步骤

- 自定义一个类继承FileOutputFormat。
- 改写RecordWriter，具体改写输出数据的方法write()。

让天下没有难学的技术

3.4.2 自定义 OutputFormat 案例实操

1) 需求

过滤输入的 log 日志，包含 atguigu 的网站输出到 e:/atguigu.log，不包含 atguigu 的网站

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

输出到 e:/other.log。

(1) 输入数据



log.txt

(2) 期望输出数据



atguigu.log other.log

2) 需求分析

自定义OutputFormat案例分析

1. **需求**：过滤输入的log日志，包含atguigu的网站输出到e:/atguigu.log，不包含atguigu的网站输出到e:/other.log

2. 输入数据

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

3. 输出数据

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

4. 自定义一个OutputFormat类

(1) 创建一个类LogRecordWriter继承RecordWriter

- (a) 创建两个文件的输出流：atguiguOut、otherOut
- (b) 如果输入数据包含atguigu，输出到atguiguOut流
如果不包含atguigu，输出到otherOut流

5. 驱动类Driver

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

让天下没有难学的技术

3) 案例实操

(1) 编写 LogMapper 类

```
package com.atguigu.mapreduce.outputformat;

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

import java.io.IOException;

public class LogMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
    @Override
    protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
        //不做任何处理，直接写出一行 log 数据
        context.write(value, NullWritable.get());
    }
}
```

(2) 编写 LogReducer 类

```
package com.atguigu.mapreduce.outputformat;

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

import java.io.IOException;

public class LogReducer extends Reducer<Text, NullWritable, Text, NullWritable> {
    @Override
    protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {
        // 防止有相同的数据,迭代写出
        for (NullWritable value : values) {
            context.write(key, NullWritable.get());
        }
    }
}
```

(3) 自定义一个 LogOutputFormat 类

```
package com.atguigu.mapreduce.outputformat;

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;

import java.io.IOException;

public class LogOutputFormat extends FileOutputFormat<Text, NullWritable> {
    @Override
    public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext job) throws IOException, InterruptedException {
        // 创建一个自定义的 RecordWriter 返回
        LogRecordWriter logRecordWriter = new LogRecordWriter(job);
        return logRecordWriter;
    }
}
```

(4) 编写 LogRecordWriter 类

```
package com.atguigu.mapreduce.outputformat;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
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 java.io.IOException;

public class LogRecordWriter extends RecordWriter<Text, NullWritable> {
    private FSDataOutputStream atguiguOut;
```

```

private FSDataOutputStream otherOut;

public LogRecordWriter(TaskAttemptContext job) {
    try {
        //获取文件系统对象
        FileSystem fs = FileSystem.get(job.getConfiguration());
        //用文件系统对象创建两个输出流对应不同的目录
        atguiguOut = fs.create(new Path("d:/hadoop/atguigu.log"));
        otherOut = fs.create(new Path("d:/hadoop/other.log"));
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Override
public void write(Text key, NullWritable value) throws IOException,
InterruptedException {
    String log = key.toString();
    //根据一行的 log 数据是否包含 atguigu, 判断两条输出流输出的内容
    if (log.contains("atguigu")) {
        atguiguOut.writeBytes(log + "\n");
    } else {
        otherOut.writeBytes(log + "\n");
    }
}

@Override
public void close(TaskAttemptContext context) throws IOException,
InterruptedException {
    //关闭
    IOUtils.closeStream(atguiguOut);
    IOUtils.closeStream(otherOut);
}
}

```

(5) 编写 LogDriver 类

```

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;

import java.io.IOException;

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

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

        job.setJarByClass(LogDriver.class);
        job.setMapperClass(LogMapper.class);
        job.setReducerClass(LogReducer.class);

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

```

```

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

//设置自定义的 outputformat
job.setOutputFormatClass(LogOutputFormat.class);

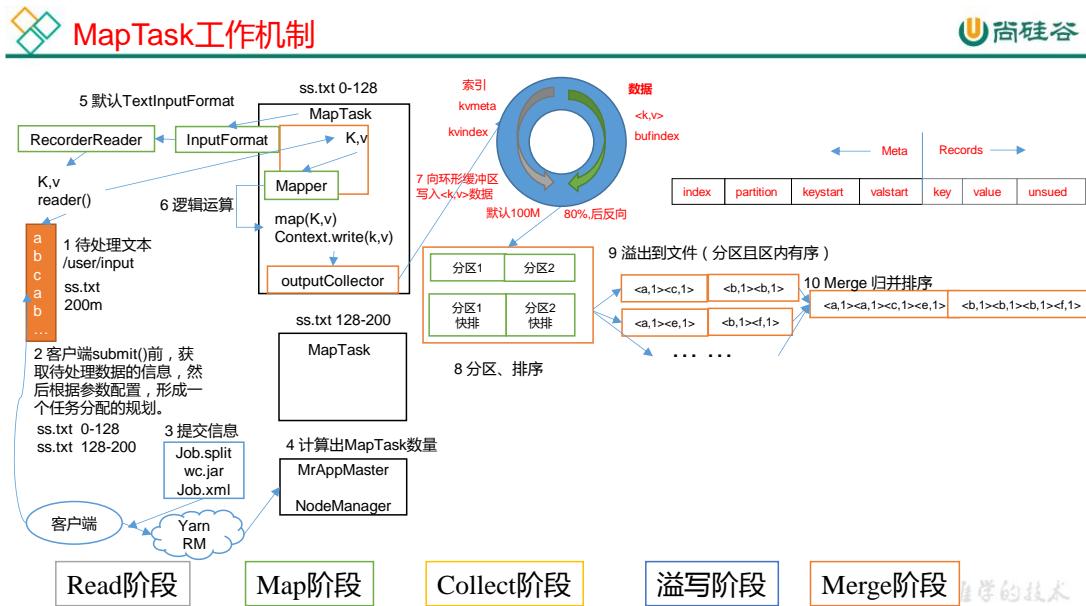
FileInputFormat.setInputPaths(job, new Path("D:\\input"));
//虽然我们自定义了 outputformat，但是因为我们的 outputformat 继承自
fileoutputformat
//而 fileoutputformat 要输出一个 SUCCESS 文件，所以在这还得指定一个输出目录
FileOutputFormat.setOutputPath(job, new Path("D:\\logoutput"));

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

```

3.5 MapReduce 内核源码解析

3.5.1 MapTask 工作机制



(1) Read 阶段: MapTask 通过 InputFormat 获得的 RecordReader, 从输入 InputSplit 中解析出一个个 key/value。

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

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

(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，且同一分区内的所有数据按照 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。

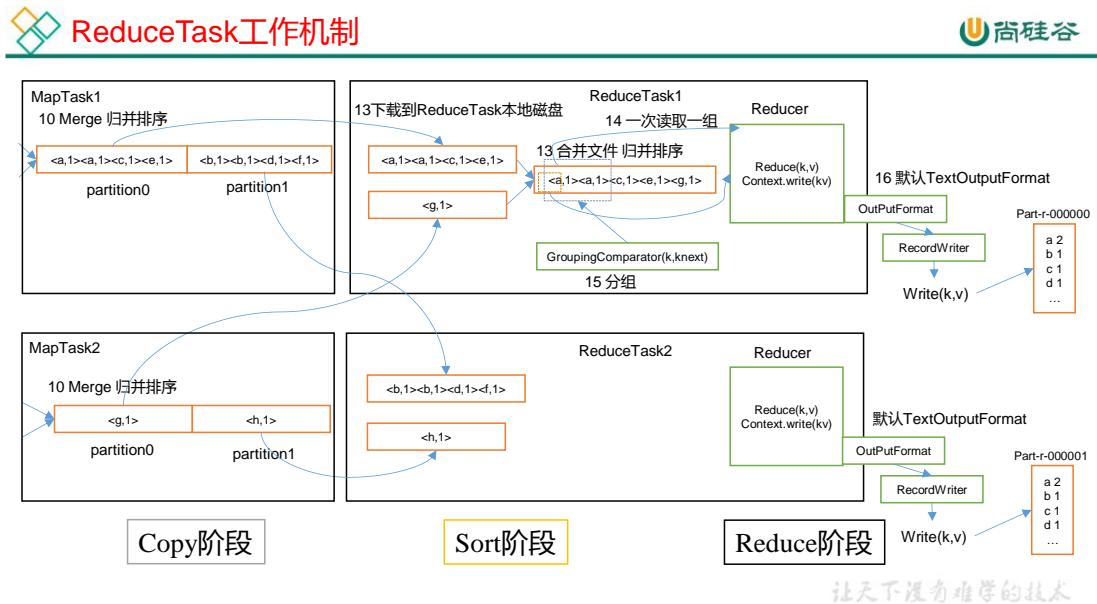
（5）Merge 阶段：当所有数据处理完成后，MapTask 对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

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 mapreduce.task.io.sort.factor（默认 10）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

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

3.5.2 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) Sort 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。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。

(3) Reduce 阶段: reduce() 函数将计算结果写到 HDFS 上。

3.5.3 ReduceTask 并行度决定机制

回顾: MapTask 并行度由切片个数决定，切片个数由输入文件和切片规则决定。

思考: ReduceTask 并行度由谁决定？

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) 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 改变 ReduceTask (数据量为 1GB)

MapTask =16										
ReduceTask	1	5	10	15	16	20	25	30	45	60
总时间	892	146	110	92	88	100	128	101	145	104

3) 注意事项



- (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 , 就只能有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。

让天下没有难学的技术

3.5.4 MapTask & ReduceTask 源码解析

1) MapTask 源码解析流程

```
===== MapTask =====
context.write(k, NullWritable.get()); //自定义的 map 方法的写出, 进入
output.write(key, value);
//MapTask727 行, 收集方法, 进入两次
collector.collect(key, value, partitioner.getPartition(key, value, partitions));
HashPartitioner(); //默认分区器
collect() //MapTask1082 行 map 端所有的 kv 全部写出后会走下面的 close 方法
close() //MapTask732 行
collector.flush() // 溢出刷写方法, MapTask735 行, 提前打个断点, 进入
sortAndSpill() //溢写排序, MapTask1505 行, 进入
sorter.sort() QuickSort //溢写排序方法, MapTask1625 行, 进入
mergeParts(); //合并文件, MapTask1527 行, 进入
file.out
file.out.index
collector.close(); //MapTask739 行, 收集器关闭, 即将进入 ReduceTask
```

2) ReduceTask 源码解析流程

```
===== ReduceTask =====
if (isMapOrReduce()) //reduceTask324 行, 提前打断点
```

```

initialize() // reduceTask333 行,进入
init(shuffleContext); // reduceTask375 行,走到这需要先给下面的打断点
    totalMaps = job.getNumMapTasks(); // ShuffleSchedulerImpl 第 120 行, 提前打断点
    merger = createMergeManager(context); // 合并方法, Shuffle 第 80 行
        // MergeManagerImpl 第 232 235 行, 提前打断点
        this.inMemoryMerger = createInMemoryMerger(); // 内存合并
        this.onDiskMerger = new OnDiskMerger(this); // 磁盘合并
    rIter = shuffleConsumerPlugin.run();
    eventFetcher.start(); // 开始抓取数据, Shuffle 第 107 行, 提前打断点
    eventFetcher.shutDown(); // 抓取结束, Shuffle 第 141 行, 提前打断点
    copyPhase.complete(); // copy 阶段完成, Shuffle 第 151 行
    taskStatus.setPhase(TaskStatus.Phase.SORT); // 开始排序阶段, Shuffle 第 152 行
    sortPhase.complete(); // 排序阶段完成, 即将进入 reduce 阶段 reduceTask382 行
reduce(); // reduce 阶段调用的就是我们自定义的 reduce 方法, 会被调用多次
cleanup(context); // reduce 完成之前, 会最后调用一次 Reducer 里面的 cleanup 方法

```

3.6 Join 应用

3.6.1 Reduce Join

Map 端的主要工作：为来自不同表或文件的 key/value 对，打标签以区别不同来源的记录。然后用连接字段作为 key，其余部分和新加的标志作为 value，最后进行输出。

Reduce 端的主要工作：在 Reduce 端以连接字段作为 key 的分组已经完成，我们只需要在每一个分组当中将那些来源于不同文件的记录（在 Map 阶段已经打标志）分开，最后进行合并就 ok 了。

3.6.2 Reduce Join 案例实操

1) 需求



表 4-4 订单数据表 t_order

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



表 4-5 商品信息表 t_product

pid	pname
01	小米

02	华为
03	格力

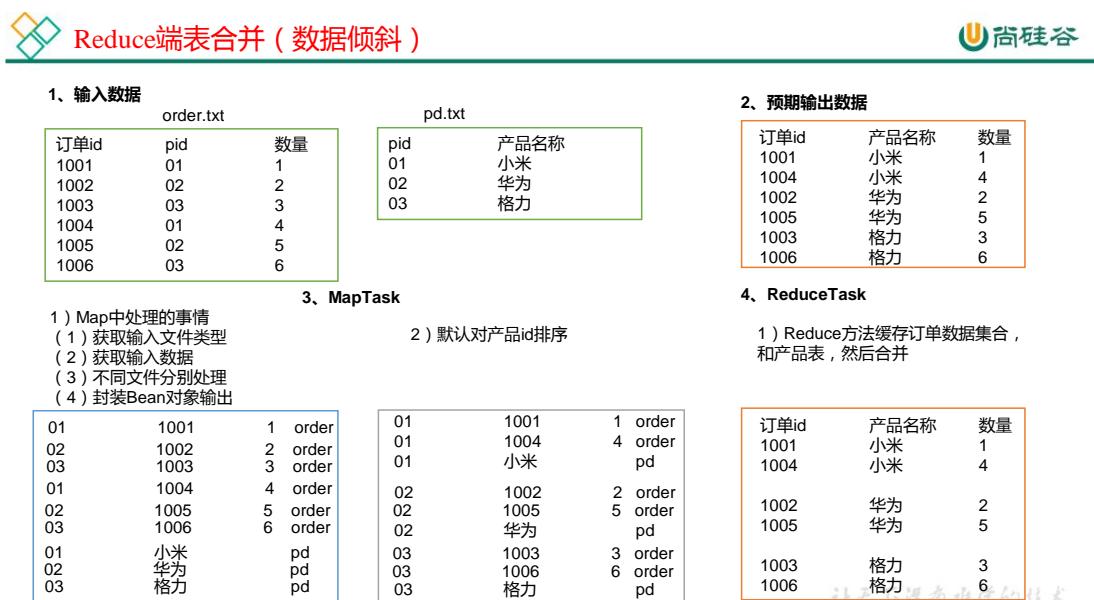
将商品信息表中数据根据商品 pid 合并到订单数据表中。

表 4-6 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

2) 需求分析

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 ReduceTask，在 Reduce 中进行数据的串联。



3) 代码实现

(1) 创建商品和订单合并后的 TableBean 类

```
package com.atguigu.mapreduce.reducejoin;

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

public class TableBean implements Writable {
    private String id; //订单 id
    private String pid; //产品 id
    private int amount; //产品数量
}
```

```
private String pname; //产品名称
private String flag; //判断是 order 表还是 pd 表的标志字段

public TableBean() {
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public String getPid() {
    return pid;
}

public void setPid(String pid) {
    this.pid = pid;
}

public int getAmount() {
    return amount;
}

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

public String getPname() {
    return pname;
}

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

public String getFlag() {
    return flag;
}

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

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

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

@Override
public void readFields(DataInput in) throws IOException {
```

```
    this.id = in.readUTF();
    this.pid = in.readUTF();
    this.amount = in.readInt();
    this.pname = in.readUTF();
    this.flag = in.readUTF();
}
}
```

(2) 编写 TableMapper 类

```
package com.atguigu.mapreduce.reducejoin;

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

import java.io.IOException;

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

    private String filename;
    private Text outK = new Text();
    private TableBean outV = new TableBean();

    @Override
    protected void setup(Context context) throws IOException,
    InterruptedException {
        //获取对应文件名称
        InputSplit split = context.getInputSplit();
        FileSplit fileSplit = (FileSplit) split;
        filename = fileSplit.getPath().getName();
    }

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

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

        //判断是哪个文件,然后针对文件进行不同的操作
        if(filename.contains("order")){
            //订单表的处理
            String[] split = line.split("\t");
            //封装 outK
            outK.set(split[1]);
            //封装 outV
            outV.setId(split[0]);
            outV.setPid(split[1]);
            outV.setAmount(Integer.parseInt(split[2]));
            outV.setPname("");
            outV.setFlag("order");
        }else{
            //商品表的处理
            String[] split = line.split("\t");
            //封装 outK
            outK.set(split[0]);
            //封装 outV
            outV.setId("");
            outV.setPid(split[0]);
            outV.setAmount(0);
            outV.setPname(split[1]);
        }
    }
}
```

```
        outV.setFlag("pd");
    }

    //写出 KV
    context.write(outK,outV);
}
}
```

(3) 编写 TableReducer 类

```
package com.atguigu.mapreduce.reducejoin;

import org.apache.commons.beanutils.BeanUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;

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

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

        ArrayList<TableBean> orderBeans = new ArrayList<>();
        TableBean pdBean = new TableBean();

        for (TableBean value : values) {

            //判断数据来自哪个表
            if("order".equals(value.getFlag())){ //订单表

                //创建一个临时 TableBean 对象接收 value
                TableBean tmpOrderBean = new TableBean();

                try {
                    BeanUtils.copyProperties(tmpOrderBean,value);
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                }

                //将临时 TableBean 对象添加到集合 orderBeans
                orderBeans.add(tmpOrderBean);
            }else { //商品表
                try {
                    BeanUtils.copyProperties(pdBean,value);
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                }
            }
        }

        //遍历集合 orderBeans, 替换掉每个 orderBean 的 pid 为 pname, 然后写出
        for (TableBean orderBean : orderBeans) {
    }
```



```
        orderBean.setPname(pdBean.getPname());  
  
        //写出修改后的 orderBean 对象  
        context.write(orderBean, NullWritable.get());  
    }  
}  
}
```

(4) 编写 TableDriver 类

```
package com.atguigu.mapreduce.reducejoin;

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;

import java.io.IOException;

public class TableDriver {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        Job job = Job.getInstance(new Configuration());

        job.setJarByClass(TableDriver.class);
        job.setMapperClass(TableMapper.class);
        job.setReducerClass(TableReducer.class);

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

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

        FileInputFormat.setInputPaths(job, new Path("D:\\\\input"));
        FileOutputFormat.setOutputPath(job, new Path("D:\\\\output"));

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

4) 测试

运行程序查看结果

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

5) 总结

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。

3.6.3 Map Join

1) 使用场景

Map Join 适用于一张表十分小、一张表很大的场景。

2) 优点

思考：在 Reduce 端处理过多的表，非常容易产生数据倾斜。怎么办？

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 Driver 驱动类中加载缓存。

```
//缓存普通文件到 Task 运行节点。
job.addCacheFile(new URI("file:///e:/cache/pd.txt"));
//如果是集群运行，需要设置 HDFS 路径
job.addCacheFile(new URI("hdfs://hadoop102:8020/cache/pd.txt"));
```

3.6.4 Map Join 案例实操

1) 需求

表 订单数据表 t_order



order.txt

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

表 商品信息表 t_product



pd.txt

pid	pname
01	小米
02	华为
03	格力

将商品信息表中数据根据商品 pid 合并到订单数据表中。

表 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

2) 需求分析

MapJoin 适用于关联表中有小表的情形。

Map端表合并案例分析 (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pid
// 4 缓存数据到集合 <pid, pname> 01,小米 02,华为 03,格力	// 4 获取订单id和 商品名称
// 5 关流	// 5 拼接
	// 6 写出

让天下没有难学的技术

3) 实现代码

(1) 先在 MapJoinDriver 驱动类中添加缓存文件

```
package com.atguigu.mapreduce.mapjoin;

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;

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

public class MapJoinDriver {

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

        // 1 获取 job 信息
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        // 2 设置加载 jar 包路径
        job.setJarByClass(MapJoinDriver.class);
        // 3 关联 mapper
        job.setMapperClass(MapJoinMapper.class);
        // 4 设置 Map 输出 KV 类型
    }
}
```

```

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(NullWritable.class);
        // 5 设置最终输出 KV 类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        // 加载缓存数据
        job.addCacheFile(new URI("file:///D:/input/tablecache/pd.txt"));
        // Map 端 Join 的逻辑不需要 Reduce 阶段, 设置 reduceTask 数量为 0
        job.setNumReduceTasks(0);

        // 6 设置输入输出路径
        FileInputFormat.setInputPaths(job, new Path("D:\\input"));
        FileOutputFormat.setOutputPath(job, new Path("D:\\output"));
        // 7 提交
        boolean b = job.waitForCompletion(true);
        System.exit(b ? 0 : 1);
    }
}

```

(2) 在 MapJoinMapper 类中的 setup 方法中读取缓存文件

```

package com.atguigu.mapreduce.mapjoin;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;
import java.util.HashMap;
import java.util.Map;

public class MapJoinMapper extends Mapper<LongWritable, Text, Text, NullWritable> {

    private Map<String, String> pdMap = new HashMap<>();
    private Text text = new Text();

    //任务开始前将 pd 数据缓存进 pdMap
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        //通过缓存文件得到小表数据 pd.txt
        URI[] cacheFiles = context.getCacheFiles();
        Path path = new Path(cacheFiles[0]);

        //获取文件系统对象，并开流
        FileSystem fs = FileSystem.get(context.getConfiguration());
        FSDataInputStream fis = fs.open(path);

        //通过包装流转换为 reader，方便按行读取
        BufferedReader reader = new BufferedReader(new InputStreamReader(fis, "UTF-8"));

```

```

    //逐行读取，按行处理
    String line;
    while (StringUtils.isNotEmpty(line = reader.readLine())) {
        //切割一行
        //01 小米
        String[] split = line.split("\t");
        pdMap.put(split[0], split[1]);
    }

    //关流
    IOUtils.closeStream(reader);
}

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

    //读取大表数据
    //1001 01 1
    String[] fields = value.toString().split("\t");

    //通过大表每行数据的 pid,去 pdMap 里面取出 pname
    String pname = pdMap.get(fields[1]);

    //将大表每行数据的 pid 替换为 pname
    text.set(fields[0] + "\t" + pname + "\t" + fields[2]);

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

```

3.7 数据清洗 (ETL)

“ETL，是英文 Extract-Transform-Load 的缩写，用来描述将数据从来源端经过抽取（Extract）、转换（Transform）、加载（Load）至目的端的过程。ETL 一词较常用在数据仓库，但其对象并不限于数据仓库

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。

1) 需求

去除日志中字段个数小于等于 11 的日志。

(1) 输入数据



(2) 期望输出数据

每行字段长度都大于 11。

2) 需求分析

需要在 Map 阶段对输入的数据根据规则进行过滤清洗。

3) 实现代码

(1) 编写 WebLogMapper 类

```
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 WebLogMapper extends Mapper<LongWritable, Text, Text, NullWritable> {

    @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 日志合法就直接写出
        context.write(value, NullWritable.get());
    }

    // 2 封装解析日志的方法
    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) {
            return true;
        } else {
            return false;
        }
    }
}
```

(2) 编写 WebLogDriver 类

```
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 WebLogDriver {  
    public static void main(String[] args) throws Exception {  
  
        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  
        args = new String[] { "D:/input/inputlog", "D:/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(WebLogMapper.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  
        boolean b = job.waitForCompletion(true);  
        System.exit(b ? 0 : 1);  
    }  
}
```

3.8 MapReduce 开发总结

1) 输入数据接口: InputFormat

- (1) 默认使用的实现类是: TextInputFormat
- (2) 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。
- (3) CombineTextInputFormat 可以把多个小文件合并成一个切片处理, 提高处理效率。

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

- (1) 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
- (2) 如果业务上有特别的需求, 可以自定义分区。

4) Comparable 排序

- (1) 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。
- (2) 部分排序：对最终输出的每一个文件进行内部排序。
- (3) 全排序：对所有数据进行排序，通常只有一个 Reduce。
- (4) 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) 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()

7) 输出数据接口：OutputFormat

(1) 默认实现类是 TextOutputFormat，功能逻辑是：将每一个 KV 对，向目标文本文件输出一行。

(2) 用户还可以自定义 OutputFormat。

第 4 章 Hadoop 数据压缩

4.1 概述

1) 压缩的好处和坏处

压缩的优点：以减少磁盘 IO、减少磁盘存储空间。

压缩的缺点：增加 CPU 开销。

2) 压缩原则

(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

1) 压缩算法对比介绍

压缩格式	Hadoop 自带？	算法	文件扩展名	是否可切片	换成压缩格式后，原来的程序是否需要修改
DEFLATE	是，直接使用	DEFLATE	.deflate	否	和文本处理一样，不需要修改

Gzip	是, 直接使用	DEFLATE	.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	是, 直接使用	Snappy	.snappy	否	和文本处理一样, 不需要修改

2) 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/>

Snappy is a compression/decompression library. It **does not aim for maximum compression**, or compatibility with any other compression library; instead, it **aims for very high speeds** and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. 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

压缩方式选择时重点考虑: **压缩/解压缩速度、压缩率(压缩后存储大小)、压缩后是否可以支持切片。**

4.3.1 Gzip 压缩

优点: 压缩率比较高;

缺点: 不支持 Split; 压缩/解压速度一般;

4.3.2 Bzip2 压缩

优点: 压缩率高; 支持 Split;

缺点: 压缩/解压速度慢。

4.3.3 Lzo 压缩

优点: 压缩/解压速度比较快; 支持 Split;

缺点: 压缩率一般; 想支持切片需要额外创建索引。

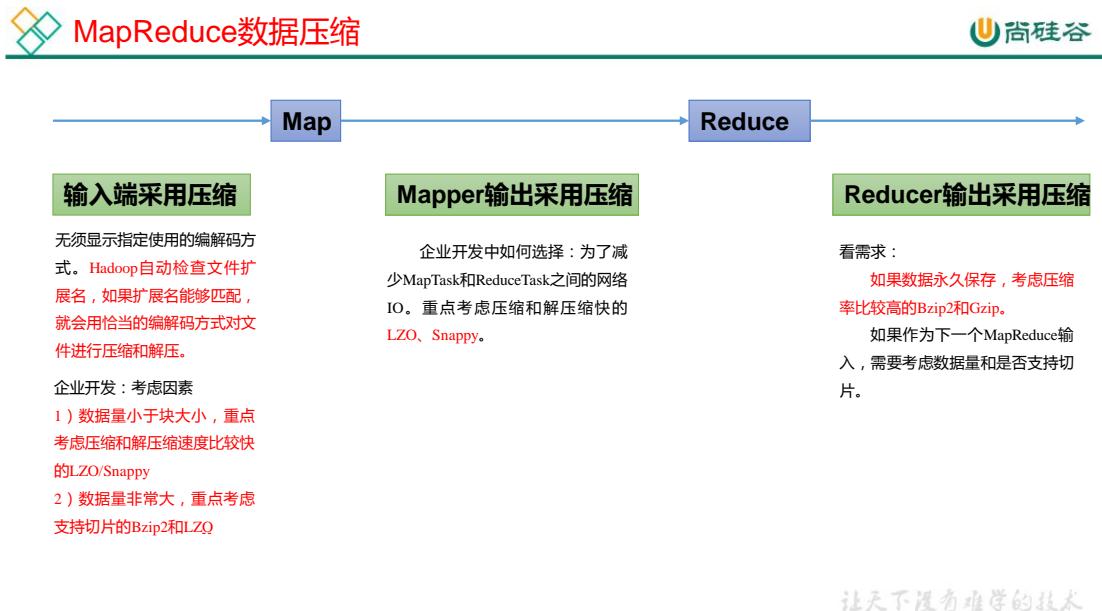
4.3.4 Snappy 压缩

优点：压缩和解压缩速度快；

缺点：不支持 Split；压缩率一般；

4.3.5 压缩位置选择

压缩可以在 MapReduce 作用的任意阶段启用。



让天下没有难学的技术

4.4 压缩参数配置

1) 为了支持多种压缩/解压缩算法，Hadoop 引入了编码/解码器

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

2) 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
io.compression.codecs (在 core-site.xml 中配置)	无，这个需要在命令行输入 hadoop checknative 查看	输入压缩	Hadoop 使用文件扩展名判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.codec (在 mapred-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec	mapper 输出	企业多使用 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

4.5 压缩实操案例

4.5.1 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 conf = new Configuration();

        // 开启 map 端输出压缩
        conf.setBoolean("mapreduce.map.output.compress", true);

        // 设置 map 端输出压缩方式
        conf.setClass("mapreduce.map.output.compress.codec",
        BZip2Codec.class, CompressionCodec.class);
    }
}
```

```

        Job job = Job.getInstance(conf);

        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 ? 0 : 1);
    }
}

```

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>{

    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 循环写出
        for(String word:words){
            k.set(word);
            context.write(k, v);
        }
    }
}

```

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>{

    IntWritable v = new IntWritable();

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

        int sum = 0;

        // 1 汇总
        for(IntWritable value:values){
            sum += value.get();
        }

        v.set(sum);

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

4.5.2 Reduce 输出端采用压缩

基于 WordCount 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 conf = new Configuration();

        Job job = Job.getInstance(conf);

        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?0:1);
}
```

2) Mapper 和 Reducer 保持不变 (详见 4.5.1)

第 5 章 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在 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。

方案二：创建如下包名，并将 NativeIO.java 拷贝到该包名下



10) 自定义 Outputformat 时，注意在 RecordWriter 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();  
    }  
}
```

尚硅谷大数据技术之 Hadoop (Yarn)

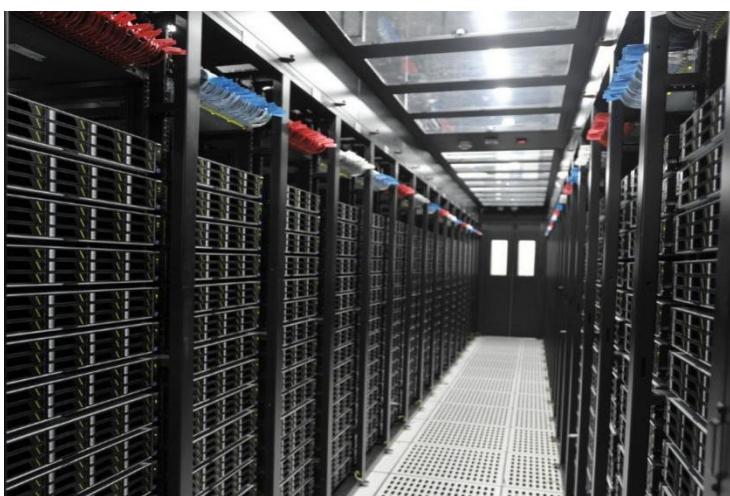
(作者：尚硅谷大数据研发部)

版本：V3.3

第 1 章 Yarn 资源调度器

思考：

- 1) 如何管理集群资源？
- 2) 如何给任务合理分配资源？

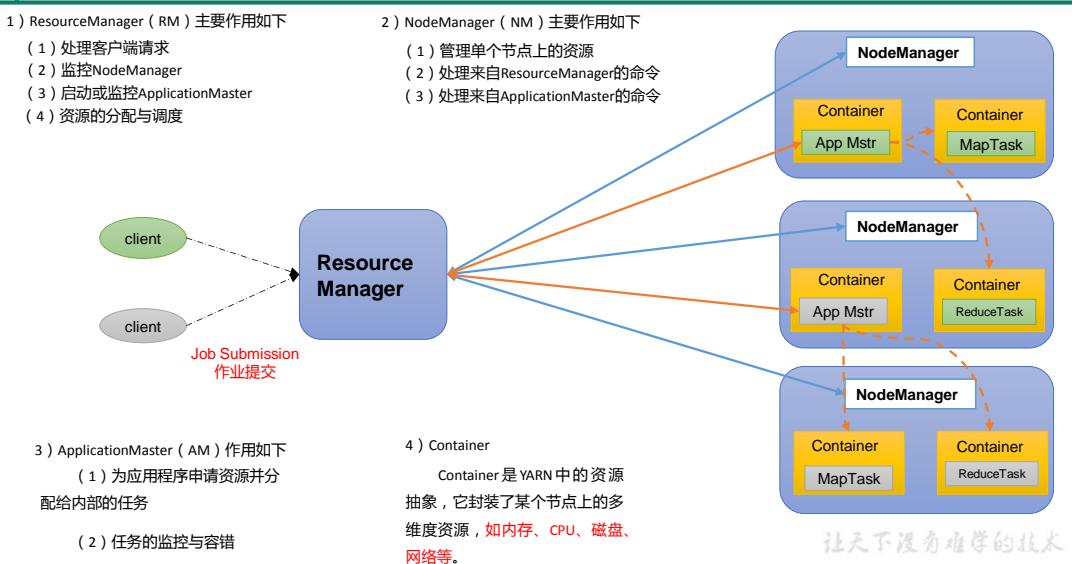


Yarn 是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而 MapReduce 等运算程序则相当于运行于操作系统之上的应用程序。

1.1 Yarn 基础架构

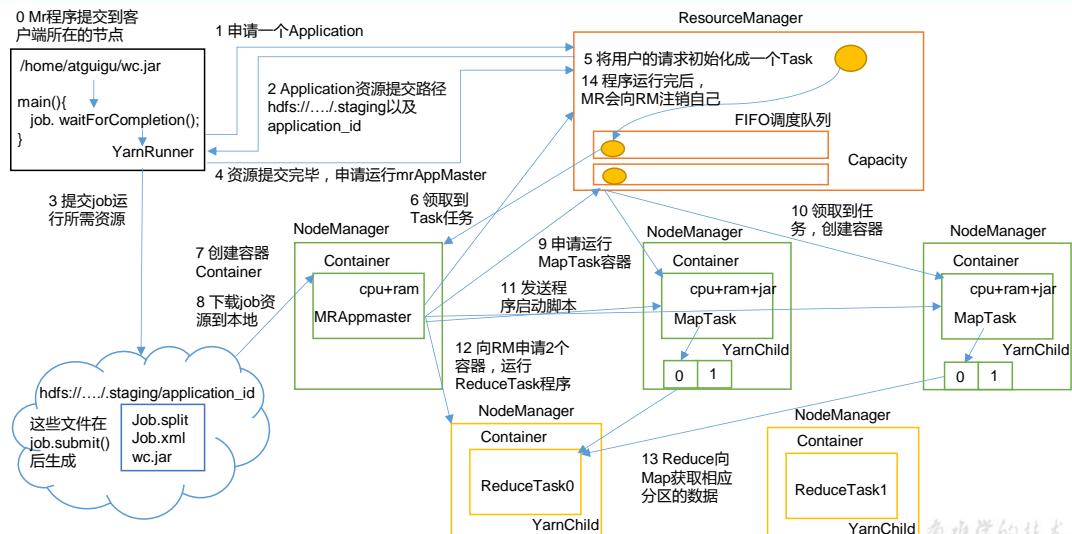
YARN 主要由 ResourceManager、NodeManager、ApplicationMaster 和 Container 等组件构成。

YARN基础架构



1.2 Yarn 工作机制

YARN工作机制

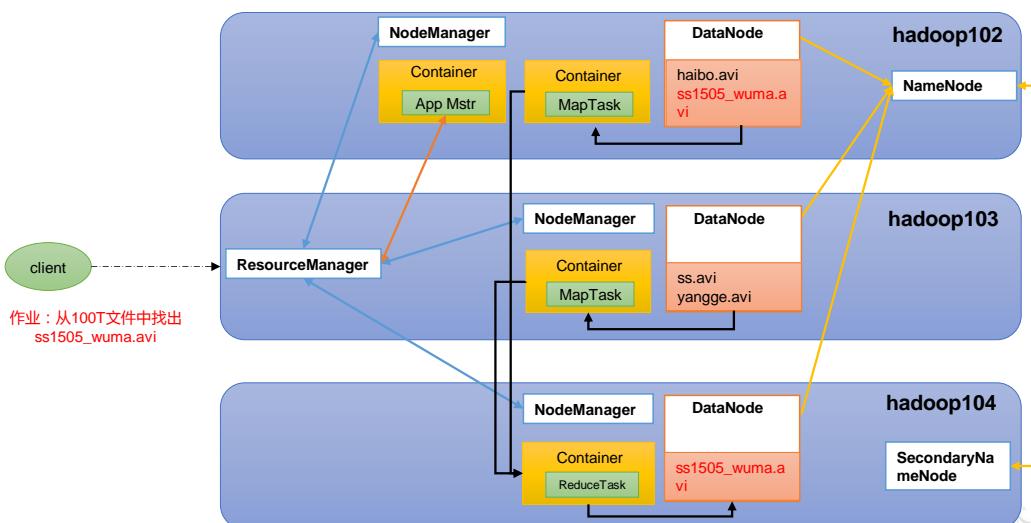


- (1) MR 程序提交到客户端所在的节点。
- (2) YarnRunner 向 ResourceManager 申请一个 Application。
- (3) RM 将该应用程序的资源路径返回给 YarnRunner。
- (4) 该程序将运行所需资源提交到 HDFS 上。
- (5) 程序资源提交完毕后，申请运行 mrAppMaster。
- (6) RM 将用户的请求初始化成一个 Task。
- (7) 其中一个 NodeManager 领取到 Task 任务。
- (8) 该 NodeManager 创建容器 Container，并产生 MRAppmaster。

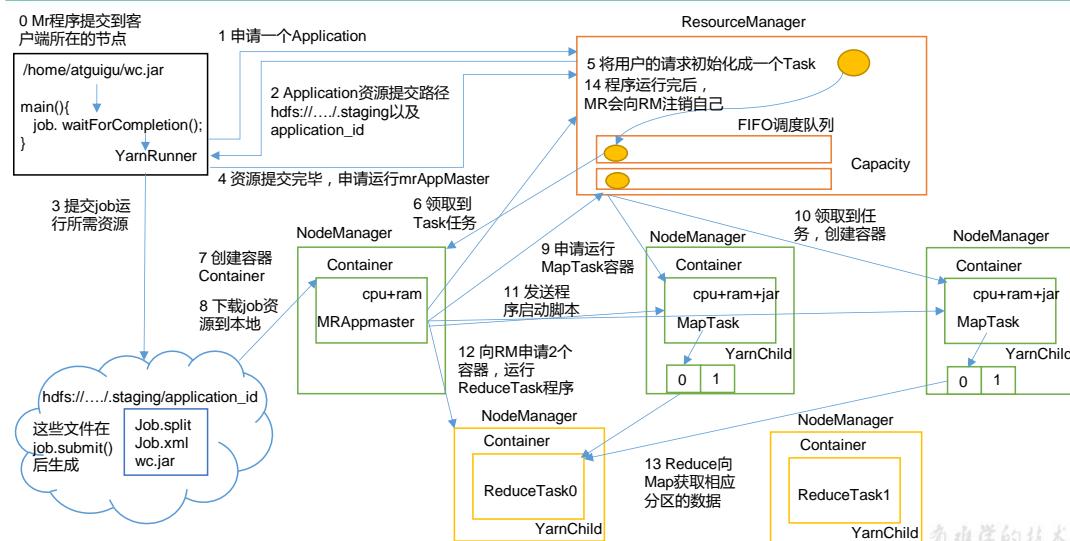
- (9) Container 从 HDFS 上拷贝资源到本地。
- (10) MRAppmaster 向 RM 申请运行 MapTask 资源。
- (11) RM 将运行 MapTask 任务分配给另外两个 NodeManager, 另两个 NodeManager 分别领取任务并创建容器。
- (12) MR 向两个接收到任务的 NodeManager 发送程序启动脚本, 这两个 NodeManager 分别启动 MapTask, MapTask 对数据分区排序。
- (13) MrAppMaster 等待所有 MapTask 运行完毕后, 向 RM 申请容器, 运行 ReduceTask。
- (14) ReduceTask 向 MapTask 获取相应分区的数据。
- (15) 程序运行完毕后, MR 会向 RM 申请注销自己。

1.3 作业提交全过程

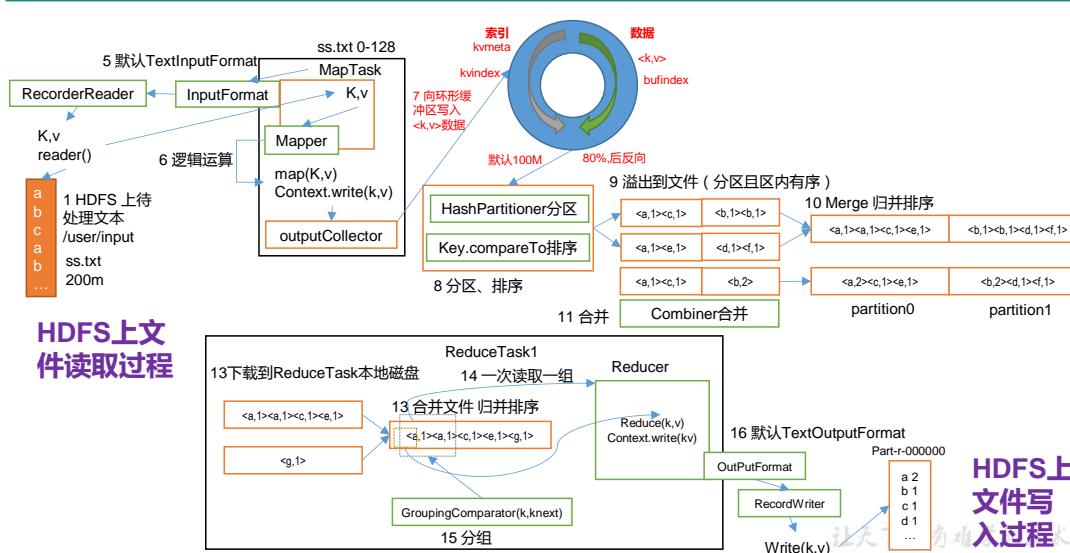
HDFS、YARN、MapReduce三者关系



作业提交过程之 YARN



作业提交过程之HDFS & MapReduce



作业提交全过程详解

(1) 作业提交

第 1 步：Client 调用 job.waitForCompletion 方法，向整个集群提交 MapReduce 作业。

第 2 步：Client 向 RM 申请一个作业 id。

第 3 步：RM 给 Client 返回该 job 资源的提交路径和作业 id。

第 4 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 5 步：Client 提交完资源后，向 RM 申请运行 MrAppMaster。

(2) 作业初始化

第 6 步：当 RM 收到 Client 的请求后，将该 job 添加到容量调度器中。

第 7 步：某一个空闲的 NM 领取到该 Job。

第 8 步：该 NM 创建 Container，并产生 MRAppmaster。

第 9 步：下载 Client 提交的资源到本地。

(3) 任务分配

第 10 步：MrAppMaster 向 RM 申请运行多个 MapTask 任务资源。

第 11 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 12 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 13 步：MrAppMaster 等待所有 MapTask 运行完毕后，向 RM 申请容器，运行 ReduceTask。

第 14 步：ReduceTask 向 MapTask 获取相应分区的数据。

第 15 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 秒都会通过调用 waitForCompletion() 来检查作业是否完成。时间间隔可以通过 mapreduce.client.completion.pollinterval 来设置。作业完成之后，应用管理器和 Container 会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查。

1.4 Yarn 调度器和调度算法

目前，Hadoop 作业调度器主要有三种：FIFO、容量（Capacity Scheduler）和公平（Fair Scheduler）。Apache Hadoop3.1.3 默认的资源调度器是 Capacity Scheduler。

CDH 框架默认调度器是 Fair 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.capaci
```

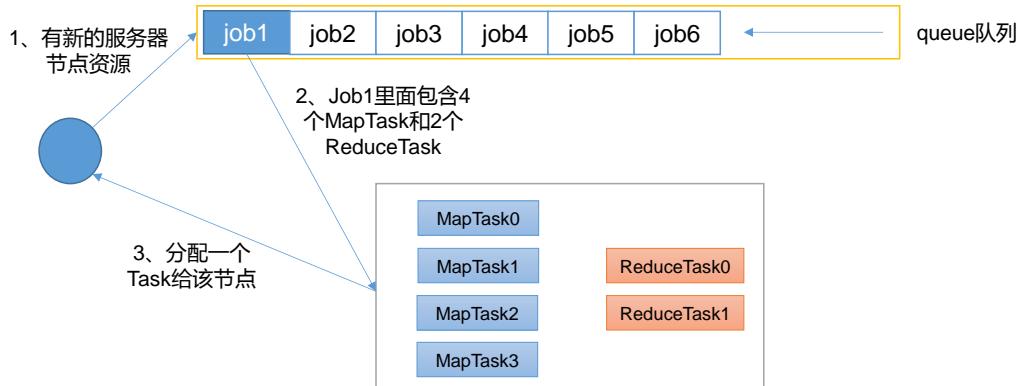
```
ty.CapacityScheduler</value>
</property>
```

1.4.1 先进先出调度器 (FIFO)

FIFO 调度器 (First In First Out)：单队列，根据提交作业的先后顺序，先来先服务。



按照到达时间排序，先到先服务



让天下没有难学的技术

优点：简单易懂；

缺点：不支持多队列，生产环境很少使用；

1.4.2 容量调度器 (Capacity Scheduler)

Capacity Scheduler 是 Yahoo 开发的多用户调度器。



- 1、多队列：每个队列可配置一定的资源量，每个队列采用FIFO调度策略。
- 2、容量保证：管理员可为每个队列设置资源最低保证和资源使用上限
- 3、灵活性：如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列借调的资源会归还给该队列。
- 4、多租户：
支持多用户共享集群和多应用程序同时运行。
为了防止同一个用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源量进行限定。

容量调度器资源分配算法

```
root
|---queueA 20%
|---queueB 50%
|---queueC 30%
  |---ss 50%
  |---cls 50%
```

1) 队列资源分配

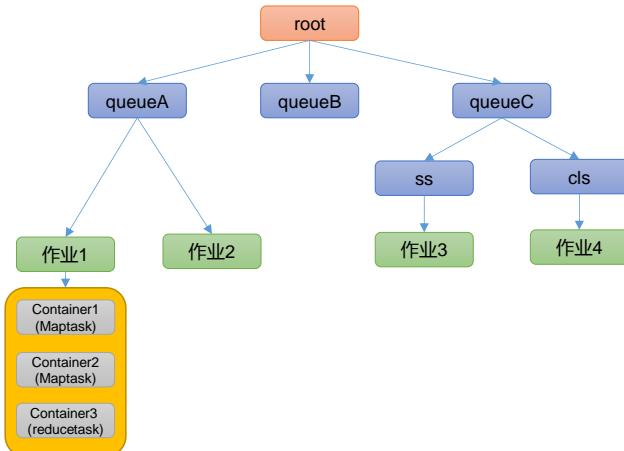
从root开始，使用深度优先算法，优先选择资源占用率最低的队列分配资源。

2) 作业资源分配

默认按照提交作业的优先级和提交时间顺序分配资源。

3) 容器资源分配

按照容器的优先级分配资源；
如果优先级相同，按照数据本地性原则：
(1) 任务和数据在同一节点
(2) 任务和数据在同一机架
(3) 任务和数据不在同一节点也不在同一机架



让天下没有难学的技术

1.4.3 公平调度器 (Fair Scheduler)

Fair Scheduler 是 Facebook 开发的多用户调度器。

公平调度器特点

```
root
|---queueA 20%
|---queueB 50%
|---queueC 30%
  |---ss 50%
  |---cls 50%
```

同队列所有任务共享资源，在时间尺度上获得公平的资源



20% 资源，4个task，每个5% queueA



50% 资源，5个task，每个10% queueB



30% 资源，5个task，每个6% queueC

1) 与容量调度器相同点

- (1) 多队列：支持多队列多作业
- (2) 容量保证：管理员可为每个队列设置资源最低保证和资源使用上线
- (3) 灵活性：如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列借调的资源会归还给该队列。
- (4) 多租户：支持多用户共享集群和多应用程序同时运行；为了防止同一个用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源量进行限定。

2) 与容量调度器不同点

- (1) 核心调度策略不同

容量调度器：优先选择资源利用率低的队列
公平调度器：优先选择对资源的缺额比例大的

- (2) 每个队列可以单独设置资源分配方式

容量调度器：FIFO、DRF

公平调度器：FIFO、FAIR、DRF

公平调度器——缺额

理想：



现实：



- 公平调度器设计目标是：在时间尺度上，所有作业获得公平的资源。某一时刻一个作业应获资源和实际获取资源的差距叫“**缺额**”
- 调度器会**优先为缺额大的作业分配资源**

让天下没有难学的技术

公平调度器队列资源分配方式

1) FIFO策略

公平调度器每个队列资源分配策略如果选择FIFO的话，此时公平调度器相当于上面讲过的容量调度器。

2) Fair策略

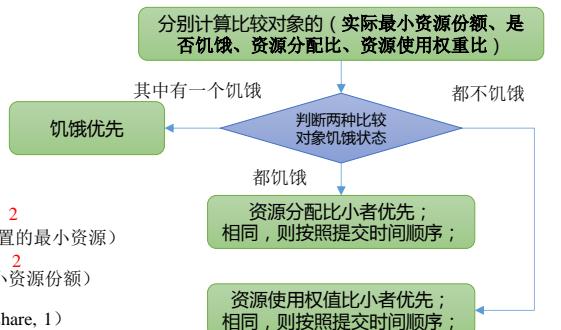
Fair 策略（默认）是一种基于最大最小公平算法实现的资源多路复用方式，默认情况下，每个队列内部采用该方式分配资源。这意味着，如果一个队列中有两个应用程序同时运行，则每个应用程序可得到1/2的资源；如果三个应用程序同时运行，则每个应用程序可得到1/3的资源。

具体资源分配流程和容量调度器一致：

- (1) 选择队列
- (2) 选择作业
- (3) 选择容器

以上三步，每一步都是按照公平策略分配资源

- 实际最小资源份额： $mindshare = \text{Min}(\text{资源需求量}, \text{配置的最小资源})$
- 是否饥饿： $isNeedy = \text{资源使用量} < mindshare$ (实际最小资源份额)
- 资源分配比： $minShareRatio = \text{资源使用量} / \text{Max}(mindshare, 1)$
- 资源使用权重比： $useToWeightRatio = \text{资源使用量} / \text{权重}$



让天下没有难学的技术

公平调度器资源分配算法

```

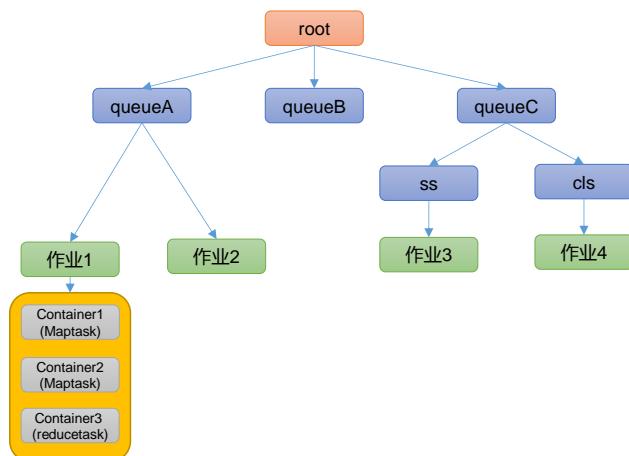
root
|---queueA 20%
|---queueB 50%
|---queueC 30%
    |---ss 50%
    |---cls 50%
  
```

(1) 队列资源分配

需求：集群总资源100，有三个队列，对资源的需求分别是：
queueA -> 20, queueB -> 50, queueC -> 30

第一次算： $100 / 3 = 33.33$
queueA : 分33.33 -> 多13.33
queueB : 分33.33 -> 少16.67
queueC : 分33.33 -> 多3.33

第二次算： $(13.33 + 3.33) / 1 = 16.66$
queueA : 分20
queueB : 分33.33 + 16.66 = 50
queueC : 分30



让天下没有难学的技术

公平调度器队列资源分配方式

(2) 作业资源分配

(a) 不加权（关注点是Job的个数）：

需求：有一条队列总资源12个，有4个job，对资源的需求分别是：
job1->1, job2->2, job3->6, job4->5

第一次算： $12 / 4 = 3$
job1: 分3 -> 多2个
job2: 分3 -> 多1个
job3: 分3 -> 差3个
job4: 分3 -> 差2个

第二次算： $3 / 2 = 1.5$
job1: 分1
job2: 分2
job3: 分3 -> 差3个 -> 分1.5 -> 最终: 4.5
job4: 分3 -> 差2个 -> 分1.5 -> 最终: 4.5

第n次算：一直算到没有空闲资源

(b) 加权（关注点是Job的权重）：

需求：有一条队列总资源16，有4个job

对资源的需求分别是：

job1->4 job2->2 job3->10 job4->4

每个job的权重为：

job1->5 job2->8 job3->1 job4->2

第一次算： $16 / (5+8+1+2) = 1$

job1: 分5 -> 多1
job2: 分8 -> 多6
job3: 分1 -> 少9
job4: 分2 -> 少2

第二次算： $7 / (1+2) = 7/3$

job1: 分4
job2: 分2
job3: 分1 -> 分 $7/3 (2.33)$ -> 少6.67
job4: 分2 -> 分 $14/3 (4.66)$ -> 多2.66

第三次算： $2.66/1=2.66$

job1: 分4
job2: 分2
job3: 分1 -> 分 $2.66/1$ -> 分2.66
job4: 分4

第n次算：一直算到没有空闲资源

 公平调度器队列资源分配方式

3) DRF策略

DRF (Dominant Resource Fairness)，我们之前说的资源，都是单一标准，例如只考虑内存（也是Yarn默认的情况）。但是很多时候我们资源有很多种，例如内存，CPU，网络带宽等，这样我们很难衡量两个应用应该分配的资源比例。

那么在YARN中，我们用DRF来决定如何调度：

假设集群一共有100 CPU和10T 内存，而应用A需要（2 CPU, 300GB），应用B需要（6 CPU, 100GB）。则两个应用分别需要A（2%CPU, 3%内存）和B（6%CPU, 1%内存）的资源，这就意味着A是内存主导的，B是CPU主导的，针对这种情况，我们可以选择DRF策略对不同应用进行不同资源（CPU和内存）的一个不同比例的限制。

让天下没有难学的技术

1.5 Yarn 常用命令

Yarn 状态的查询，除了可以在 hadoop103:8088 页面查看外，还可以通过命令操作。常见的命令操作如下所示：

需求：执行 WordCount 案例，并用 Yarn 命令查看任务运行情况。

```
[atguigu@hadoop102 hadoop-3.1.3]$ myhadoop.sh start
[atguigu@hadoop102          hadoop-3.1.3]$ hadoop          jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar      wordcount
/input /output
```

1.5.1 yarn application 查看任务

(1) 列出所有 Application:

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn application -list
2021-02-06 10:21:19,238 INFO client.RMProxy: Connecting to ResourceManager
at hadoop103/192.168.10.103:8032
Total number of applications (application-types: [], states: [SUBMITTED,
ACCEPTED, RUNNING] and tags: []):0
      Application-Id      Application-Name      Application-Type
User        Queue        State        Final-State        Progress
                           Tracking-URL
```

(2) 根据 Application 状态过滤: yarn application -list -appStates (所有状态: ALL、NEW、
NEW_SAVING、SUBMITTED、ACCEPTED、RUNNING、FINISHED、FAILED、KILLED)

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn application -list -appStates
FINISHED
2021-02-06 10:22:20,029 INFO client.RMProxy: Connecting to ResourceManager
at hadoop103/192.168.10.103:8032
Total number of applications (application-types: [], states: [FINISHED]
and tags: []):1
      Application-Id      Application-Name      Application-Type
User        Queue        State        Final-State        Progress
                           Tracking-URL
application_1612577921195_0001      word count      MAPREDUCE
```

```
atguigu default FINISHED SUCCEEDED 100%
http://hadoop102:19888/jobhistory/job/job_1612577921195_0001
```

(3) Kill 掉 Application:

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn application -kill
application_1612577921195_0001
2021-02-06 10:23:48,530 INFO client.RMProxy: Connecting to ResourceManager
at hadoop103/192.168.10.103:8032
Application application_1612577921195_0001 has already finished
```

1.5.2 yarn logs 查看日志

(1) 查询 Application 日志: yarn logs -applicationId <ApplicationId>

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn logs -applicationId
application_1612577921195_0001
```

(2) 查询 Container 日志: yarn logs -applicationId <ApplicationId> -containerId <ContainerId>

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn logs -applicationId
application_1612577921195_0001 -containerId
container_1612577921195_0001_01_000001
```

1.5.3 yarn applicationattempt 查看尝试运行的任务

(1) 列出所有 Application 尝试的列表: yarn applicationattempt -list <ApplicationId>

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn applicationattempt -list
application_1612577921195_0001
2021-02-06 10:26:54,195 INFO client.RMProxy: Connecting to ResourceManager
at hadoop103/192.168.10.103:8032
Total number of application attempts :1
ApplicationAttempt-Id State AM-
Container-Id Tracking-URL
appattempt_1612577921195_0001_000001 FINISHED
    container_1612577921195_0001_01_000001
    http://hadoop103:8088/proxy/application_1612577921195_0001/
```

(2) 打印 ApplicationAttempt 状态: yarn applicationattempt -status <ApplicationAttemptId>

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn applicationattempt -status
appattempt_1612577921195_0001_000001
2021-02-06 10:27:55,896 INFO client.RMProxy: Connecting to ResourceManager
at hadoop103/192.168.10.103:8032
Application Attempt Report :
    ApplicationAttempt-Id : appattempt_1612577921195_0001_000001
    State : FINISHED
    AMContainer : container_1612577921195_0001_01_000001
    Tracking-URL :
    http://hadoop103:8088/proxy/application_1612577921195_0001/
    RPC Port : 34756
    AM Host : hadoop104
    Diagnostics :
```

1.5.4 yarn container 查看容器

(1) 列出所有 Container: yarn container -list <ApplicationAttemptId>

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn container -list
appattempt_1612577921195_0001_000001
2021-02-06 10:28:41,396 INFO client.RMProxy: Connecting to ResourceManager
at hadoop103/192.168.10.103:8032
Total number of containers :0
Container-Id Start Time Finish Time
```

State	Host	Node	Http Address
(2) 打印 Container 状态: yarn container -status <ContainerId>			
[atguigu@hadoop102 hadoop-3.1.3]\$ yarn container -status container_1612577921195_0001_01_000001			
2021-02-06 10:29:58,554 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032			
Container with id 'container_1612577921195_0001_01_000001' doesn't exist in RM or Timeline Server.			

注: 只有在任务跑的途中才能看到 container 的状态

1.5.5 yarn node 查看节点状态

列出所有节点: yarn node -list -all

Node-Id	Node-State	Node-Http-Address	Number-of-Running-Containers
hadoop103:38168	RUNNING	hadoop103:8042	0
hadoop102:42012	RUNNING	hadoop102:8042	0
hadoop104:39702	RUNNING	hadoop104:8042	0

1.5.6 yarn rmadmin 更新配置

加载队列配置: yarn rmadmin -refreshQueues

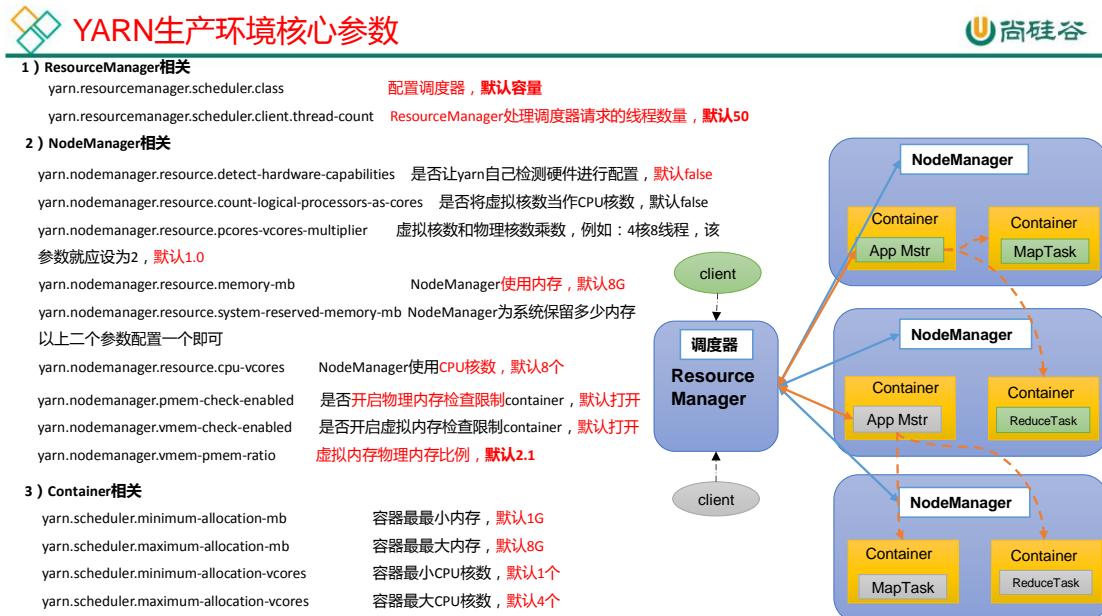
[atguigu@hadoop102 hadoop-3.1.3]\$ yarn rmadmin -refreshQueues
2021-02-06 10:32:03,331 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8033

1.5.7 yarn queue 查看队列

打印队列信息: yarn queue -status <QueueName>

[atguigu@hadoop102 hadoop-3.1.3]\$ yarn queue -status default
2021-02-06 10:32:33,403 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
Queue Information :
Queue Name : default
State : RUNNING
Capacity : 100.0%
Current Capacity : .0%
Maximum Capacity : 100.0%
Default Node Label expression : <DEFAULT_PARTITION>
Accessible Node Labels : *
Preemption : disabled
Intra-queue Preemption : disabled

1.6 Yarn 生产环境核心参数



第 2 章 Yarn 案例实操

注：调整下列参数之前尽量拍摄 Linux 快照，否则后续的案例，还需要重写准备集群。

2.1 Yarn 生产环境核心参数配置案例

1) 需求：从 1G 数据中，统计每个单词出现次数。服务器 3 台，每台配置 4G 内存，4 核 CPU，4 线程。

2) 需求分析：

$1G / 128m = 8$ 个 MapTask； 1 个 ReduceTask； 1 个 mrAppMaster

平均每个节点运行 $10 / 3 \approx 3$ 个任务 ($4 / 3 = 3$)

3) 修改 yarn-site.xml 配置参数如下：

```
<!-- 选择调度器， 默认容量 -->
<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>

<!-- ResourceManager 处理调度器请求的线程数量， 默认 50；如果提交的任务数大于 50，可以增加该值，但是不能超过 3 台 * 4 线程 = 12 线程（去除其他应用实际不能超过 8） -->
<property>
    <description>Number of threads to handle scheduler interface.</description>
    <name>yarn.resourcemanager.scheduler.client.thread-count</name>
    <value>8</value>
</property>
```

```
<!-- 是否让 yarn 自动检测硬件进行配置，默认是 false，如果该节点有很多其他应用程序，建议手动配置。如果该节点没有其他应用程序，可以采用自动 -->
<property>
    <description>Enable auto-detection of node capabilities such as memory and CPU.
    </description>
    <name>yarn.nodemanager.resource.detect-hardware-capabilities</name>
    <value>false</value>
</property>

<!-- 是否将虚拟核数当作 CPU 核数，默认是 false，采用物理 CPU 核数 -->
<property>
    <description>Flag to determine if logical processors(such as hyperthreads) should be counted as cores. Only applicable on Linux when yarn.nodemanager.resource.cpu-vcores is set to -1 and yarn.nodemanager.resource.detect-hardware-capabilities is true.
    </description>
    <name>yarn.nodemanager.resource.count-logical-processors-as-cores</name>
    <value>false</value>
</property>

<!-- 虚拟核数和物理核数乘数，默认是 1.0 -->
<property>
    <description>Multiplier to determine how to convert physical cores to vcores. This value is used if yarn.nodemanager.resource.cpu-vcores is set to -1(which implies auto-calculate vcores) and yarn.nodemanager.resource.detect-hardware-capabilities is set to true. The number of vcores will be calculated as number of CPUs * multiplier.
    </description>
    <name>yarn.nodemanager.resource.pcores-vcores-multiplier</name>
    <value>1.0</value>
</property>

<!-- NodeManager 使用内存数，默认 8G，修改为 4G 内存 -->
<property>
    <description>Amount of physical memory, in MB, that can be allocated for containers. If set to -1 and yarn.nodemanager.resource.detect-hardware-capabilities is true, it is automatically calculated(in case of Windows and Linux). In other cases, the default is 8192MB.
    </description>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>4096</value>
</property>

<!-- nodemanager 的 CPU 核数，不按照硬件环境自动设定时默认是 8 个，修改为 4 个 -->
<property>
    <description>Number of vcores that can be allocated for containers. This is used by the RM scheduler when allocating resources for containers. This is not used to limit the number of CPUs used by YARN containers. If it is set to -1 and yarn.nodemanager.resource.detect-hardware-capabilities is true, it is automatically determined from the hardware in case of Windows and Linux. In other cases, number of vcores is 8 by default.</description>
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>4</value>
</property>

<!-- 容器最小内存，默认 1G -->
<property>
    <description>The minimum allocation for every container request at the
```

```
RM in MBs. Memory requests lower than this will be set to the value of
this property. Additionally, a node manager that is configured to have
less memory than this value will be shut down by the resource manager.
</description>
<name>yarn.scheduler.minimum-allocation-mb</name>
<value>1024</value>
</property>

<!-- 容器最大内存， 默认 8G， 修改为 2G --&gt;
&lt;property&gt;
    &lt;description&gt;The maximum allocation for every container request at the
RM in MBs. Memory requests higher than this will throw an
InvalidResourceRequestException.
&lt;/description&gt;
&lt;name&gt;yarn.scheduler.maximum-allocation-mb&lt;/name&gt;
&lt;value&gt;2048&lt;/value&gt;
&lt;/property&gt;

<!-- 容器最小 CPU 核数， 默认 1 个 --&gt;
&lt;property&gt;
    &lt;description&gt;The minimum allocation for every container request at the
RM in terms of virtual CPU cores. Requests lower than this will be set to
the value of this property. Additionally, a node manager that is configured
to have fewer virtual cores than this value will be shut down by the
resource manager.
&lt;/description&gt;
&lt;name&gt;yarn.scheduler.minimum-allocation-vcores&lt;/name&gt;
&lt;value&gt;1&lt;/value&gt;
&lt;/property&gt;

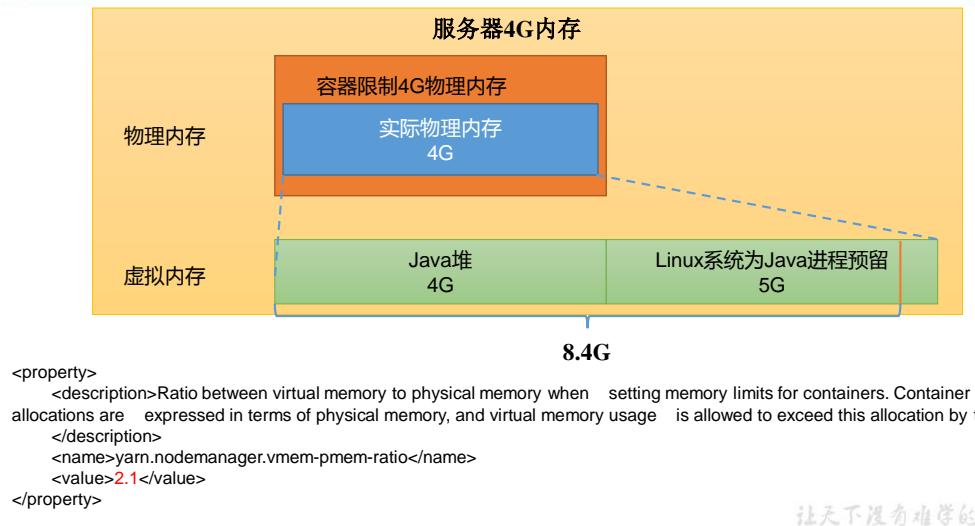
<!-- 容器最大 CPU 核数， 默认 4 个， 修改为 2 个 --&gt;
&lt;property&gt;
    &lt;description&gt;The maximum allocation for every container request at the
RM in terms of virtual CPU cores. Requests higher than this will throw an
InvalidResourceRequestException.&lt;/description&gt;
&lt;name&gt;yarn.scheduler.maximum-allocation-vcores&lt;/name&gt;
&lt;value&gt;2&lt;/value&gt;
&lt;/property&gt;

<!-- 虚拟内存检查， 默认打开， 修改为关闭 --&gt;
&lt;property&gt;
    &lt;description&gt;Whether virtual memory limits will be enforced for
containers.&lt;/description&gt;
&lt;name&gt;yarn.nodemanager.vmem-check-enabled&lt;/name&gt;
&lt;value&gt;false&lt;/value&gt;
&lt;/property&gt;

<!-- 虚拟内存和物理内存设置比例， 默认 2.1 --&gt;
&lt;property&gt;
    &lt;description&gt;Ratio between virtual memory to physical memory when
setting memory limits for containers. Container allocations are
expressed in terms of physical memory, and virtual memory usage is
allowed to exceed this allocation by this ratio.
&lt;/description&gt;
&lt;name&gt;yarn.nodemanager.vmem-pmem-ratio&lt;/name&gt;
&lt;value&gt;2.1&lt;/value&gt;
&lt;/property&gt;</pre>
```

关闭虚拟内存检查原因

尚硅谷



4) 分发配置。

注意：如果集群的硬件资源不一致，要每个 NodeManager 单独配置

5) 重启集群

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/stop-yarn.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

6) 执行 WordCount 程序

```
[atguigu@hadoop102          hadoop-3.1.3]$          hadoop          jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar      wordcount
/input /output
```

7) 观察 Yarn 任务执行页面

<http://hadoop103:8088/cluster/apps>

2.2 容量调度器多队列提交案例

1) 在生产环境怎么创建队列？

- (1) 调度器默认就 1 个 default 队列，不能满足生产要求。
- (2) 按照框架：hive /spark/ flink 每个框架的任务放入指定的队列（企业用的不是特别多）
- (3) 按照业务模块：登录注册、购物车、下单、业务部门 1、业务部门 2

2) 创建多队列的好处？

- (1) 因为担心员工不小心，写递归死循环代码，把所有资源全部耗尽。
- (2) 实现任务的降级使用，特殊时期保证重要的任务队列资源充足。11.11 6.18

业务部门 1 (重要) => 业务部门 2 (比较重要) => 下单 (一般) => 购物车 (一般) =>
登录注册 (次要)

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

2.2.1 需求

需求 1: default 队列占总内存的 40%，最大资源容量占总资源 60%，hive 队列占总内存的 60%，最大资源容量占总资源 80%。

需求 2: 配置队列优先级

2.2.2 配置多队列的容量调度器

1) 在 capacity-scheduler.xml 中配置如下:

(1) 修改如下配置

```
<!-- 指定多队列，增加 hive 队列 -->
<property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>default,hive</value>
    <description>
        The queues at the this level (root is the root queue).
    </description>
</property>

<!-- 降低 default 队列资源额定容量为 40%，默认 100% -->
<property>
    <name>yarn.scheduler.capacity.root.default.capacity</name>
    <value>40</value>
</property>

<!-- 降低 default 队列资源最大容量为 60%，默认 100% -->
<property>
    <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>
    <value>60</value>
</property>
```

(2) 为新加队列添加必要属性:

```
<!-- 指定 hive 队列的资源额定容量 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.capacity</name>
    <value>60</value>
</property>

<!-- 用户最多可以使用队列多少资源，1 表示 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.user-limit-factor</name>
    <value>1</value>
</property>

<!-- 指定 hive 队列的资源最大容量 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.maximum-capacity</name>
    <value>80</value>
</property>

<!-- 启动 hive 队列 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.state</name>
    <value>RUNNING</value>
</property>
```

```

<!-- 哪些用户有权向队列提交作业 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.acl_submit_applications</name>
    <value>*</value>
</property>

<!-- 哪些用户有权操作队列，管理员权限（查看/杀死） -->
<property>
    <name>yarn.scheduler.capacity.root.hive.acl_administer_queue</name>
    <value>*</value>
</property>

<!-- 哪些用户有权配置提交任务优先级 -->
<property>

<name>yarn.scheduler.capacity.root.hive.acl_application_max_priority</name>
    <value>*</value>
</property>

<!-- 任务的超时时间设置: yarn application -appId appId -updateLifetime Timeout
参考资料： https://blog.cloudera.com/enforcing-application-lifetime-slas-yarn/ -->

<!-- 如果 application 指定了超时时间，则提交到该队列的 application 能够指定的最大超时
时间不能超过该值。
-->
<property>
    <name>yarn.scheduler.capacity.root.hive.maximum-application-lifetime</name>
    <value>-1</value>
</property>

<!-- 如果 application 没指定超时时间，则用 default-application-lifetime 作为默认
值 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.default-application-lifetime</name>
    <value>-1</value>
</property>

```

2) 分发配置文件

3) 重启 Yarn 或者执行 `yarn rmadmin -refreshQueues` 刷新队列，就可以看到两条队列：

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn rmadmin -refreshQueues
```

The screenshot shows the Hadoop Cluster Management UI. On the left, there's a sidebar with a navigation tree. Under 'Cluster', the 'Scheduler' link is highlighted with a red box. The main area displays 'Cluster Metrics' and 'Scheduler Metrics' tables. Below that is the 'Application Queues' section, which lists 'root', 'default', and 'hive' queues. The 'root' queue is also highlighted with a red box.

	Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total
	0	0	0	0	0	0 B	24 GB

Scheduler Type	Scheduling Resource Type
Capacity Scheduler	[MEMORY]

Application Queues	
Legend:	Capacity Used Used (over capacity) Max Capacity
- root	
↳ default	
↳ hive	

2.2.3 向 Hive 队列提交任务

1) hadoop jar 的方式

```
[atguigu@hadoop102      hadoop-3.1.3]$      hadoop      jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar  wordcount -D
mapreduce.job.queuename=hive /input /output
```

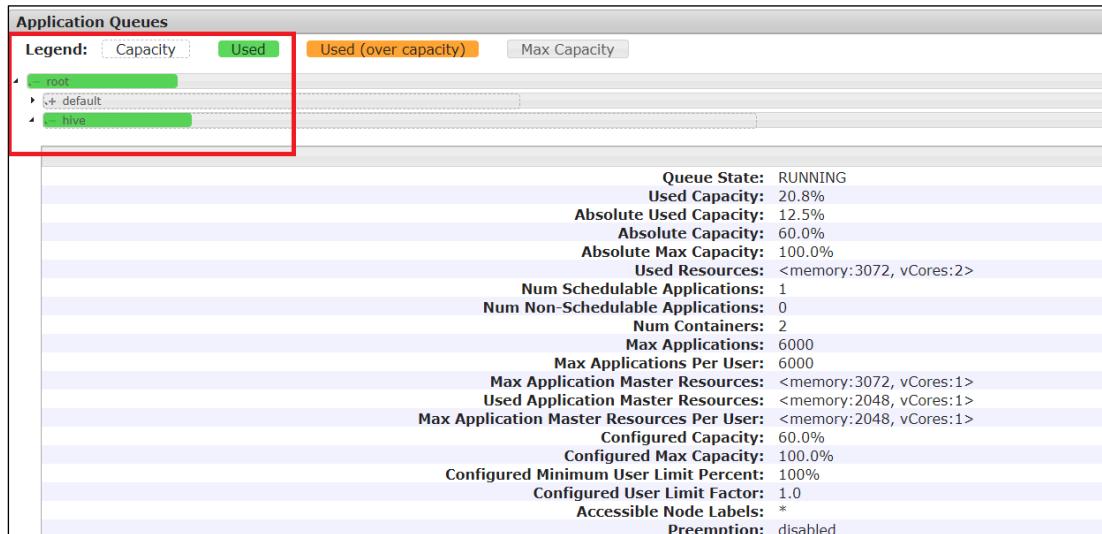
注: -D 表示运行时改变参数值

2) 打 jar 包的方式

默认的任务提交都是提交到 default 队列的。如果希望向其他队列提交任务，需要在 Driver 中声明：

```
public class WcDriver {
    public static void main(String[] args) throws IOException,
    ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        conf.set("mapreduce.job.queuename", "hive");
        //1. 获取一个 Job 实例
        Job job = Job.getInstance(conf);
        . . . .
        //6. 提交 Job
        boolean b = job.waitForCompletion(true);
        System.exit(b ? 0 : 1);
    }
}
```

这样，这个任务在集群提交时，就会提交到 hive 队列：



2.2.4 任务优先级

容量调度器，支持任务优先级的配置，在资源紧张时，优先级高的任务将优先获取资源。

默认情况，Yarn 将所有任务的优先级限制为 0，若想使用任务的优先级功能，须开放该限制。

1) 修改 yarn-site.xml 文件，增加以下参数

```
<property>
  <name>yarn.cluster.max-application-priority</name>
  <value>5</value>
</property>
```

2) 分发配置，并重启 Yarn

```
[atguigu@hadoop102 hadoop]$ xsync yarn-site.xml
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/stop-yarn.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

3) 模拟资源紧张环境，可连续提交以下任务，直到新提交的任务申请不到资源为止。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar pi 5
20000000
```

Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
0	Wed Jan 20 17:22:46 +0800 2021	N/A	N/A	ACCEPTED	UNDEFINED	0	0	0	0	0.0	0.0	0.0	0.0	ApplicationMaster	0
															已无空余资源
0	Wed Jan 20 17:21:22 +0800 2021	Wed Jan 20 17:21:23 +0800 2021	N/A	RUNNING	UNDEFINED	1	1	1536	0	0	25.0	12.5	0.0	ApplicationMaster	0
0	Wed Jan 20 17:20:32 +0800 2021	Wed Jan 20 17:20:32 +0800 2021	N/A	RUNNING	UNDEFINED	5	5	5632	0	0	91.7	45.8	0.0	ApplicationMaster	0

4) 再次重新提交优先级高的任务

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar pi -D
mapreduce.job.priority=5 5 2000000
```

Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
5	Wed Jan 20 17:23:59 +0800 2021	Wed Jan 20 17:23:59 +0800 2021	N/A	RUNNING	UNDEFINED	3	3	3584	0	0	58.3	29.2	0.0	ApplicationMaster	0
															高优先级任务，优先获取资源
0	Wed Jan 20 17:22:46 +0800 2021	N/A	N/A	ACCEPTED	UNDEFINED	0	0	0	0	0	0.0	0.0	0.0	ApplicationMaster	0
0	Wed Jan 20 17:21:22 +0800 2021	Wed Jan 20 17:21:23 +0800 2021	N/A	RUNNING	UNDEFINED	1	1	1536	0	0	25.0	12.5	0.0	ApplicationMaster	0
0	Wed Jan 20 17:20:32 +0800 2021	Wed Jan 20 17:20:32 +0800 2021	N/A	RUNNING	UNDEFINED	1	1	1536	0	0	25.0	12.5	0.0	ApplicationMaster	0

5) 也可以通过以下命令修改正在执行的任务的优先级。

yarn application -appID <ApplicationID> -updatePriority 优先级

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn application -appID
application_1611133087930_0009 -updatePriority 5
```

2.3 公平调度器案例

2.3.1 需求

创建两个队列，分别是 test 和 atguigu（以用户所属组命名）。期望实现以下效果：若用户提交任务时指定队列，则任务提交到指定队列运行；若未指定队列，test 用户提交的任务到 root.group.test 队列运行，atguigu 提交的任务到 root.group.atguigu 队列运行（注：group 为

用户所属组）。

公平调度器的配置涉及到两个文件，一个是 yarn-site.xml，另一个是公平调度器队列分配文件 fair-scheduler.xml（文件名可自定义）。

(1) 配置文件参考资料：

<https://hadoop.apache.org/docs/r3.1.3/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>

(2) 任务队列放置规则参考资料：

<https://blog.cloudera.com/untangling-apache-hadoop-yarn-part-4-fair-scheduler-queue-basics/>

2.3.2 配置多队列的公平调度器

1) 修改 yarn-site.xml 文件，加入以下参数

```
<property>
    <name>yarn.resourcemanager.scheduler.class</name>
    <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler</value>
    <description>配置使用公平调度器</description>
</property>

<property>
    <name>yarn.scheduler.fair.allocation.file</name>
    <value>/opt/module/hadoop-3.1.3/etc/hadoop/fair-scheduler.xml</value>
    <description>指明公平调度器队列分配配置文件</description>
</property>

<property>
    <name>yarn.scheduler.fair.preemption</name>
    <value>false</value>
    <description>禁止队列间资源抢占</description>
</property>
```

2) 配置 fair-scheduler.xml

```
<?xml version="1.0"?>
<allocations>
    <!-- 单个队列中 Application Master 占用资源的最大比例, 取值 0-1 , 企业一般配置 0.1 -->
    <queueMaxAMShareDefault>0.5</queueMaxAMShareDefault>
    <!-- 单个队列最大资源的默认值 test atguigu default -->
    <queueMaxResourcesDefault>4096mb,4vcores</queueMaxResourcesDefault>

    <!-- 增加一个队列 test -->
    <queue name="test">
        <!-- 队列最小资源 -->
        <minResources>2048mb,2vcores</minResources>
        <!-- 队列最大资源 -->
        <maxResources>4096mb,4vcores</maxResources>
        <!-- 队列中最多同时运行的应用数, 默认 50, 根据线程数配置 -->
        <maxRunningApps>4</maxRunningApps>
        <!-- 队列中 Application Master 占用资源的最大比例 -->
        <maxAMShare>0.5</maxAMShare>
        <!-- 该队列资源权重, 默认值为 1.0 -->
    </queue>
</allocations>
```

```

<weight>1.0</weight>
<!-- 队列内部的资源分配策略 -->
<schedulingPolicy>fair</schedulingPolicy>
</queue>
<!-- 增加一个队列 atguigu -->
<queue name="atguigu" type="parent">
    <!-- 队列最小资源 -->
    <minResources>2048mb, 2vcores</minResources>
    <!-- 队列最大资源 -->
    <maxResources>4096mb, 4vcores</maxResources>
    <!-- 队列中最多同时运行的应用数, 默认 50, 根据线程数配置 -->
    <maxRunningApps>4</maxRunningApps>
    <!-- 队列中 Application Master 占用资源的最大比例 -->
    <maxAMShare>0.5</maxAMShare>
    <!-- 该队列资源权重,默认值为 1.0 -->
    <weight>1.0</weight>
    <!-- 队列内部的资源分配策略 -->
    <schedulingPolicy>fair</schedulingPolicy>
</queue>

<!-- 任务队列分配策略, 可配置多层规则, 从第一个规则开始匹配, 直到匹配成功 -->
<queuePlacementPolicy>
    <!-- 提交任务时指定队列, 如未指定提交队列, 则继续匹配下一个规则; false 表示: 如果指定队列不存在, 不允许自动创建-->
    <rule name="specified" create="false"/>
    <!-- 提交到 root.group.username 队列, 若 root.group 不存在, 不允许自动创建; 若 root.group.user 不存在, 允许自动创建 -->
    <rule name="nestedUserQueue" create="true">
        <rule name="primaryGroup" create="false"/>
    </rule>
    <!-- 最后一个规则必须为 reject 或者 default。Reject 表示拒绝创建提交失败, default 表示把任务提交到 default 队列 -->
    <rule name="reject" />
</queuePlacementPolicy>
</allocations>

```

3) 分发配置并重启 Yarn

```

[atguigu@hadoop102 hadoop]$ xsync yarn-site.xml
[atguigu@hadoop102 hadoop]$ xsync fair-scheduler.xml

[atguigu@hadoop103 hadoop-3.1.3]$ sbin/stop-yarn.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh

```

2.3.3 测试提交任务

1) 提交任务时指定队列, 按照配置规则, 任务会到指定的 root.test 队列

```

[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar pi -
Dmapreduce.job.queuename=root.test 1 1

```

application_1611023120675_0002	atguigu	QuasiMonteCarlo	MAPREDUCE	root.test
--------------------------------	---------	-----------------	-----------	-----------

2) 提交任务时不指定队列, 按照配置规则, 任务会到 root.atguigu.atguigu 队列

```

[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar pi 1 1

```

application_1611023120675_0003	atguigu	QuasiMonteCarlo	MAPREDUCE	root.atguigu.atguigu
--------------------------------	---------	-----------------	-----------	----------------------

2.4 Yarn 的 Tool 接口案例

0) 回顾:

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar wc.jar  
com.atguigu.mapreduce.wordcount2.WordCountDriver /input  
/output1
```

期望可以动态传参，结果报错，误认为是第一个输入参数。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar wc.jar  
com.atguigu.mapreduce.wordcount2.WordCountDriver -  
Dmapreduce.job.queuename=root.test /input /output1
```

1) 需求：自己写的程序也可以动态修改参数。编写 Yarn 的 Tool 接口。

2) 具体步骤：

(1) 新建 Maven 项目 YarnDemo，pom 如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
http://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
  
    <groupId>com.atguigu.hadoop</groupId>  
    <artifactId>yarn_tool_test</artifactId>  
    <version>1.0-SNAPSHOT</version>  
  
    <dependencies>  
        <dependency>  
            <groupId>org.apache.hadoop</groupId>  
            <artifactId>hadoop-client</artifactId>  
            <version>3.1.3</version>  
        </dependency>  
    </dependencies>  
</project>
```

(2) 新建 com.atguigu.yarn 报名

(3) 创建类 WordCount 并实现 Tool 接口：

```
package com.atguigu.yarn;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.Mapper;  
import org.apache.hadoop.mapreduce.Reducer;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import  
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.util.Tool;  
  
import java.io.IOException;
```

```
public class WordCount implements Tool {  
  
    private Configuration conf;  
  
    @Override  
    public int run(String[] args) throws Exception {  
  
        Job job = Job.getInstance(conf);  
  
        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]));  
  
        return job.waitForCompletion(true) ? 0 : 1;  
    }  
  
    @Override  
    public void setConf(Configuration conf) {  
        this.conf = conf;  
    }  
  
    @Override  
    public Configuration getConf() {  
        return conf;  
    }  
  
    public static class WordCountMapper extends  
Mapper<LongWritable, Text, Text, IntWritable> {  
  
        private Text outK = new Text();  
        private IntWritable outV = new IntWritable(1);  
  
        @Override  
        protected void map(LongWritable key, Text value,  
Context context) throws IOException, InterruptedException {  
  
            String line = value.toString();  
            String[] words = line.split(" ");  
  
            for (String word : words) {  
                outK.set(word);  
  
                context.write(outK, outV);  
            }  
        }  
    }  
  
    public static class WordCountReducer extends Reducer<Text,
```

```

IntWritable, Text, IntWritable> {
    private IntWritable outV = new IntWritable();

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

        int sum = 0;

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

        context.write(key, outV);
    }
}
}

```

(4) 新建 WordCountDriver

```

package com.atguigu.yarn;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import java.util.Arrays;

public class WordCountDriver {

    private static Tool tool;

    public static void main(String[] args) throws Exception {
        // 1. 创建配置文件
        Configuration conf = new Configuration();

        // 2. 判断是否有 tool 接口
        switch (args[0]){
            case "wordcount":
                tool = new WordCount();
                break;
            default:
                throw new RuntimeException(" No such tool: "+args[0] );
        }
        // 3. 用 Tool 执行程序
        // Arrays.copyOfRange 将老数组的元素放到新数组里面
        int run = ToolRunner.run(conf, tool,
        Arrays.copyOfRange(args, 1, args.length));

        System.exit(run);
    }
}

```

- 3) 在 HDFS 上准备输入文件，假设为/input 目录，向集群提交该 Jar 包

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn jar YarnDemo.jar
com.atguigu.yarn.WordCountDriver wordcount /input /output
```

注意此时提交的 3 个参数，第一个用于生成特定的 Tool，第二个和第三个为输入输出目录。此时如果我们希望加入设置参数，可以在 wordcount 后面添加参数，例如：

```
[atguigu@hadoop102 hadoop-3.1.3]$ yarn jar YarnDemo.jar  
com.atguigu.yarn.WordCountDriver wordcount -  
Dmapreduce.job.queuename=root.test /input /output1
```

- 4) 注：以上操作全部做完过后，快照回去或者手动将配置文件修改成之前的状态，因为本身资源就不够，分成了这么多，不方便以后测试。

尚硅谷大数据技术之 Hadoop (生产调优手册)

(作者: 尚硅谷大数据研发部)

版本: V3.3

第 1 章 HDFS—核心参数

1.1 NameNode 内存生产配置

1) NameNode 内存计算

每个文件块大概占用 150byte, 一台服务器 128G 内存为例, 能存储多少文件块呢?

$$128 * 1024 * 1024 * 1024 / 150\text{Byte} \approx 9.1 \text{ 亿}$$

G MB KB Byte

2) Hadoop2.x 系列, 配置 NameNode 内存

NameNode 内存默认 2000m, 如果服务器内存 4G, NameNode 内存可以配置 3g。在 hadoop-env.sh 文件中配置如下。

HADOOP_NAMENODE_OPTS=-Xmx3072m

3) Hadoop3.x 系列, 配置 NameNode 内存

(1) hadoop-env.sh 中描述 Hadoop 的内存是动态分配的

```
# The maximum amount of heap to use (Java -Xmx). If no unit
# is provided, it will be converted to MB. Daemons will
# prefer any Xmx setting in their respective _OPT variable.
# There is no default; the JVM will autoscale based upon machine
# memory size.
# export HADOOP_HEAPSIZE_MAX=

# The minimum amount of heap to use (Java -Xms). If no unit
# is provided, it will be converted to MB. Daemons will
# prefer any Xms setting in their respective _OPT variable.
# There is no default; the JVM will autoscale based upon machine
# memory size.
# export HADOOP_HEAPSIZE_MIN=
HADOOP_NAMENODE_OPTS=-Xmx102400m
```

(2) 查看 NameNode 占用内存

```
[atguigu@hadoop102 ~]$ jps
3088 NodeManager
2611 NameNode
3271 JobHistoryServer
2744 DataNode
```

```
3579 Jps
[atguigu@hadoop102 ~]$ jmap -heap 2611
Heap Configuration:
MaxHeapSize = 1031798784 (984.0MB)
```

(3) 查看 DataNode 占用内存

```
[atguigu@hadoop102 ~]$ jmap -heap 2744
Heap Configuration:
MaxHeapSize = 1031798784 (984.0MB)
```

查看发现 hadoop102 上的 NameNode 和 DataNode 占用内存都是自动分配的，且相等。

不是很合理。

经验参考：

https://docs.cloudera.com/documentation/enterprise/6/release-notes/topics/rg_hardware_requirements.html#concept_fzz_dq4_gbb

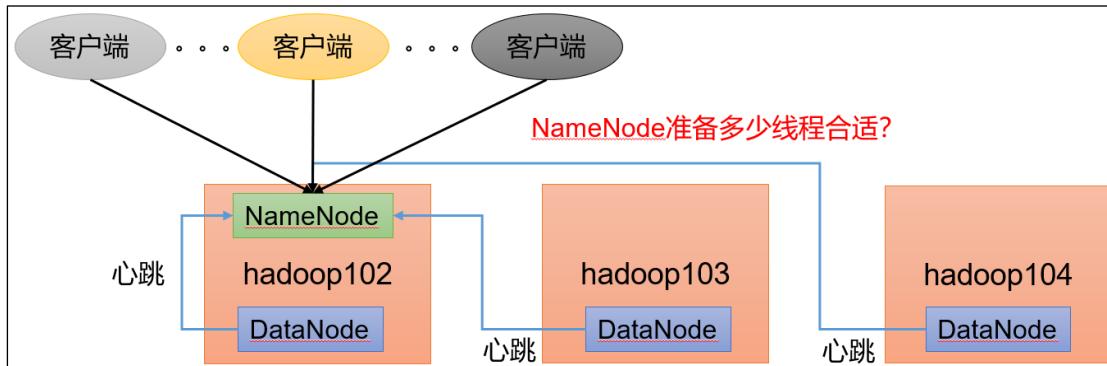
NameNode namenode最小值 1G, 每增加1000000 个block, 增加1G内存 <ul style="list-style-type: none"> Minimum: 1 GB (for proof-of-concept deployments) Add an additional 1 GB for each additional 1,000,000 blocks Snapshots and encryption can increase the required heap memory. <p>See Sizing NameNode Heap Memory</p> <p>Set this value using the Java Heap Size of NameNode in Bytes HDFS configuration property.</p>	DataNode datanode最小值 4G, block数, 或者副本数升高, 都应该调大 datanode的值。 一个datanode上的 副本总数低于4000000, 调为4G, 超过4000000, 每增加1000000, 增加 1G <p>Minimum: 4 GB</p> <p>Increase the memory for higher replica counts or a higher number of blocks per DataNode. When increasing the memory, Cloudera recommends an additional 1 GB of memory for every 1 million replicas above 4 million on the DataNodes. For example, 5 million replicas require 5 GB of memory.</p> <p>Set this value using the Java Heap Size of DataNode in Bytes HDFS configuration property.</p>
---	--

具体修改：hadoop-env.sh

```
export HDFS_NAMENODE_OPTS="-Dhadoop.security.logger=INFO,RFAS -Xmx1024m"

export HDFS_DATANODE_OPTS="-Dhadoop.security.logger=ERROR,RFAS -Xmx1024m"
```

1.2 NameNode 心跳并发配置



1) hdfs-site.xml

The number of Namenode RPC server threads that listen to requests from clients. If `dfs.namenode.servicerpc-address` is not configured then Namenode RPC server threads listen to requests from all nodes.

NameNode 有一个工作线程池，用来处理不同 **DataNode** 的并发心跳以及客户端并发的元数据操作。

对于大集群或者有大量客户端的集群来说，通常需要增大该参数。默认值是 10。

```
<property>
    <name>dfs.namenode.handler.count</name>
    <value>21</value>
</property>
```

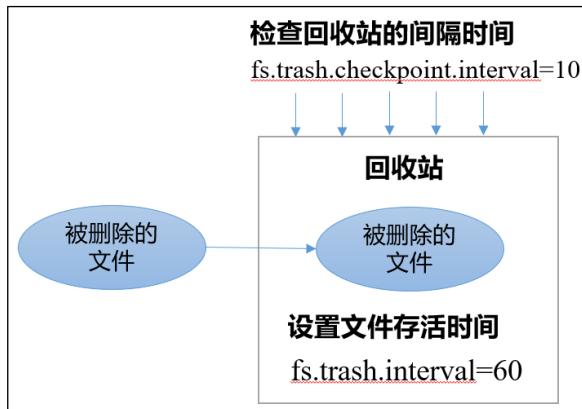
企业经验： $\text{dfs.namenode.handler.count} = 20 \times \log_e^{\text{Cluster Size}}$ ，比如集群规模（**DataNode** 台数）为 3 台时，此参数设置为 21。可通过简单的 python 代码计算该值，代码如下。

```
[atguigu@hadoop102 ~]$ sudo yum install -y python
[atguigu@hadoop102 ~]$ python
Python 2.7.5 (default, Apr 11 2018, 07:36:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import math
>>> print int(20*math.log(3))
21
>>> quit()
```

1.3 开启回收站配置

开启回收站功能，可以将删除的文件在不超时的情况下，恢复原数据，起到防止误删除、备份等作用。

1) 回收站工作机制



2) 开启回收站功能参数说明

- (1) 默认值 `fs.trash.interval = 0`, 0 表示禁用回收站; 其他值表示设置文件的存活时间。
- (2) 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。
- (3) 要求 `fs.trash.checkpoint.interval <= fs.trash.interval`。

3) 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>
```

4) 查看回收站

回收站目录在 HDFS 集群中的路径: `/user/atguigu/.Trash/...`

5) 注意: 通过网页上直接删除的文件也不会走回收站。

6) 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);
```

7) 只有在命令行利用 `hadoop fs -rm` 命令删除的文件才会走回收站。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -rm -r
/usr/atguigu/input
2021-07-14 16:13:42,643 INFO fs.TrashPolicyDefault: Moved:
'hdfs://hadoop102:9820/user/atguigu/input' to trash at:
hdfs://hadoop102:9820/user/atguigu/.Trash/Current/user/atguigu
/input
```

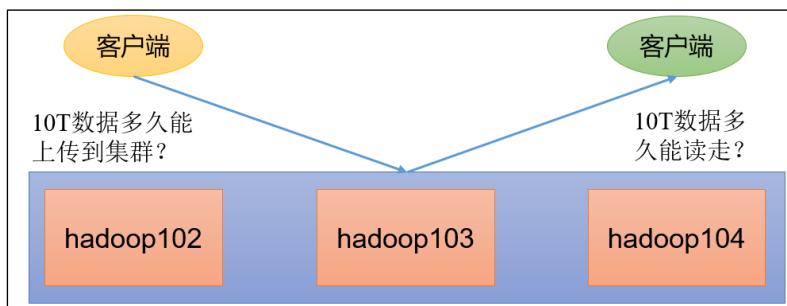
8) 恢复回收站数据

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mv
/usr/atguigu/.Trash/Current/user/atguigu/input
/usr/atguigu/input
```

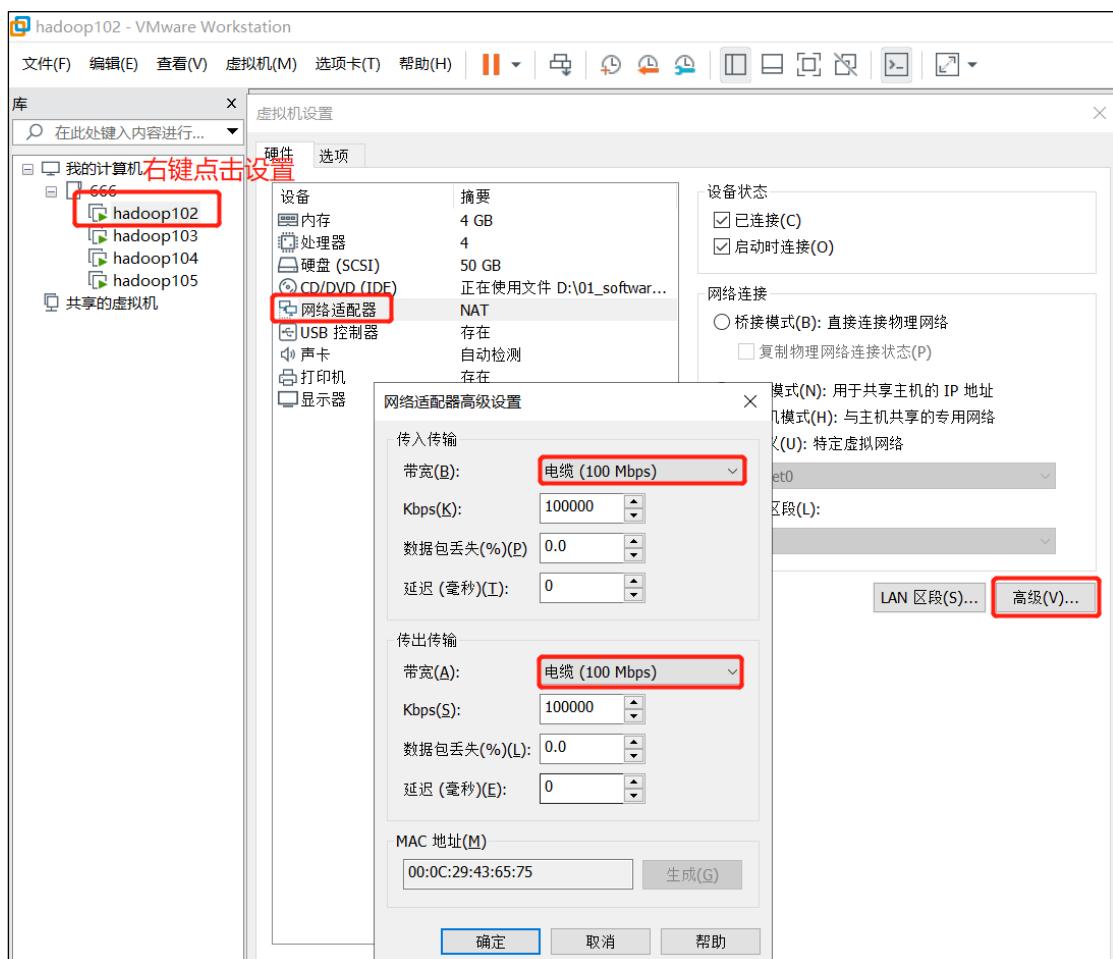
第 2 章 HDFS—集群压测

在企业中非常关心每天从 Java 后台拉取过来的数据，需要多久能上传到集群？消费者关心多久能从 HDFS 上拉取需要的数据？

为了搞清楚 HDFS 的读写性能，生产环境上非常需要对集群进行压测。



HDFS 的读写性能主要受**网络和磁盘**影响比较大。为了方便测试，将 hadoop102、hadoop103、hadoop104 虚拟机网络都设置为 100mbps。



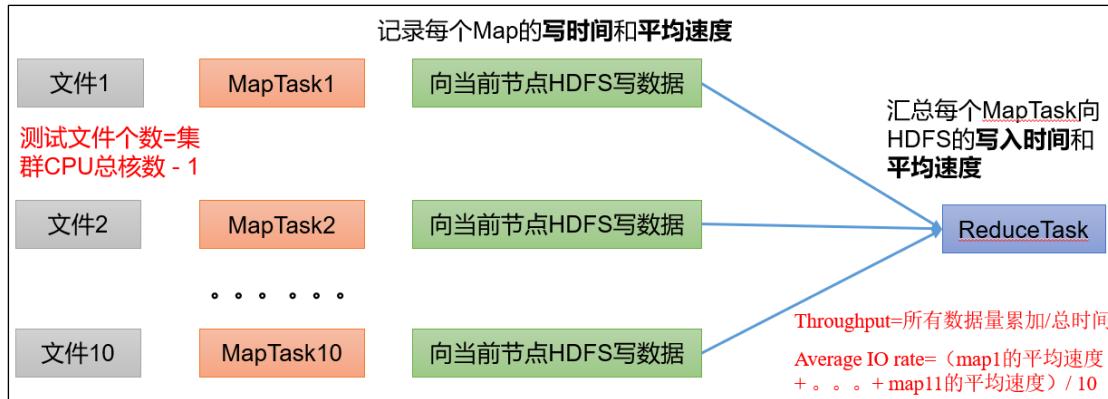
100Mbps 单位是 bit；10M/s 单位是 byte；1byte=8bit， $100\text{Mbps}/8=12.5\text{M/s}$ 。

测试网速：来到 hadoop102 的 /opt/module 目录，创建一个

```
[atguigu@hadoop102 software]$ python -m SimpleHTTPServer
```

2.1 测试 HDFS 写性能

0) 写测试底层原理



1) 测试内容：向 HDFS 集群写 10 个 128M 的文件

```
[atguigu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-3.1.3-tests.jar TestDFSIO -write -nrFiles 10 -fileSize 128MB

2021-02-09 10:43:16,853 INFO fs.TestDFSIO: ----- TestDFSIO ----- : write
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Date & time: Tue Feb 09 10:43:16 CST 2021
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Number of files: 10
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Total MBytes processed: 1280
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Throughput mb/sec: 1.61
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Average IO rate mb/sec: 1.9
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: IO rate std deviation: 0.76
2021-02-09 10:43:16,854 INFO fs.TestDFSIO: Test exec time sec: 133.05
2021-02-09 10:43:16,854 INFO fs.TestDFSIO:
```

注意: nrFilesn 为生成 mapTask 的数量, 生产环境一般可通过 hadoop103:8088 查看 CPU

核数, 设置为 (CPU 核数 - 1)

- Number of files: 生成 mapTask 数量, 一般是集群中 (CPU 核数-1), 我们测试虚拟机就按照实际的物理内存-1 分配即可
- Total MBytes processed: 单个 map 处理的文件大小
- Throughput mb/sec: 单个 mapTak 的吞吐量
计算方式: 处理的总文件大小/每一个 mapTask 写数据的时间累加
集群整体吞吐量: 生成 mapTask 数量*单个 mapTak 的吞吐量
- Average IO rate mb/sec: 平均 mapTak 的吞吐量
计算方式: 每个 mapTask 处理文件大小/每一个 mapTask 写数据的时间

全部相加除以 task 数量

- IO rate std deviation: 方差、反映各个 mapTask 处理的差值，越小越均衡

2) 注意: 如果测试过程中, 出现异常

- (1) 可以在 yarn-site.xml 中设置虚拟内存检测为 false

```
<!--是否启动一个线程检查每个任务正使用的虚拟内存量, 如果任务超出分配值, 则直接将其杀掉, 默认是 true -->
<property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>
```

- (2) 分发配置并重启 Yarn 集群

3) 测试结果分析

- (1) 由于副本 1 就在本地, 所以该副本不参与测试



一共参与测试的文件: 10 个文件 * 2 个副本 = 20 个

压测后的速度: 1.61

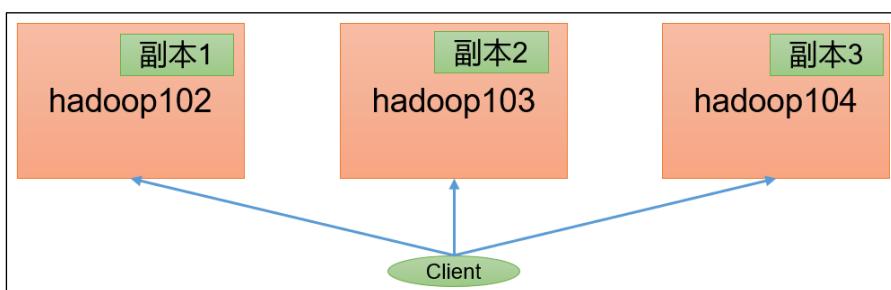
实测速度: 1.61M/s * 20 个文件 ≈ 32M/s

三台服务器的带宽: 12.5 + 12.5 + 12.5 ≈ 30m/s

所有网络资源都已经用满。

如果实测速度远远小于网络, 并且实测速度不能满足工作需求, 可以考虑采用固态硬盘或者增加磁盘个数。

- (2) 如果客户端不在集群节点, 那就三个副本都参与计算



2.2 测试 HDFS 读性能

- 1) 测试内容: 读取 HDFS 集群 10 个 128M 的文件

```
[atguigu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
```

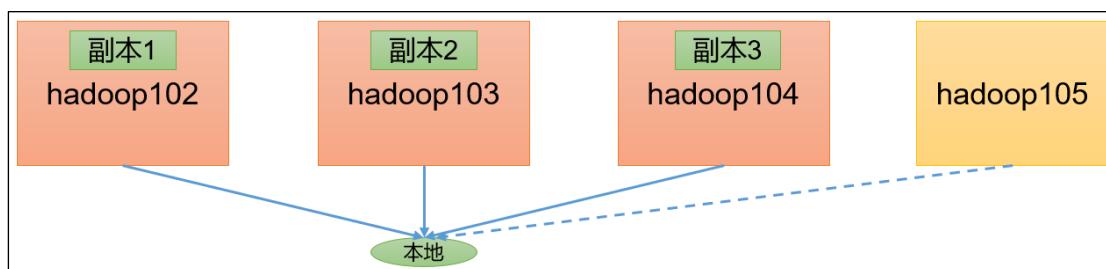
```
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-client-
jobclient-3.1.3-tests.jar TestDFSIO -read -nrFiles 10 -fileSize
128MB

2021-02-09 11:34:15,847 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: Date & time: Tue Feb
09 11:34:15 CST 2021
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: Number of files: 10
2021-02-09 11:34:15,847 INFO fs.TestDFSIO: Total MBytes processed: 1280
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: Throughput mb/sec: 200.28
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: Average IO rate mb/sec: 266.74
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: IO rate std deviation: 143.12
2021-02-09 11:34:15,848 INFO fs.TestDFSIO: Test exec time sec: 20.83
```

2) 删除测试生成数据

```
[atguigu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-
3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-client-
jobclient-3.1.3-tests.jar TestDFSIO -clean
```

3) 测试结果分析：为什么读取文件速度大于网络带宽？由于目前只有三台服务器，且有三个副本，数据读取就近原则，相当于都是读取的本地磁盘数据，没有走网络。



第3章 HDFS—多目录

3.1 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 文件中添加如下内容

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

注意：因为每台服务器节点的磁盘情况不同，所以这个配置配完之后，可以选择不分发

(2) 停止集群，删除三台节点的 data 和 logs 中所有数据。

```
[atguigu@hadoop102 hadoop-3.1.3]$ rm -rf data/ logs/
[atguigu@hadoop103 hadoop-3.1.3]$ rm -rf data/ logs/
[atguigu@hadoop104 hadoop-3.1.3]$ rm -rf data/ logs/
```

(3) 格式化集群并启动。

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hdfs namenode -format
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
```

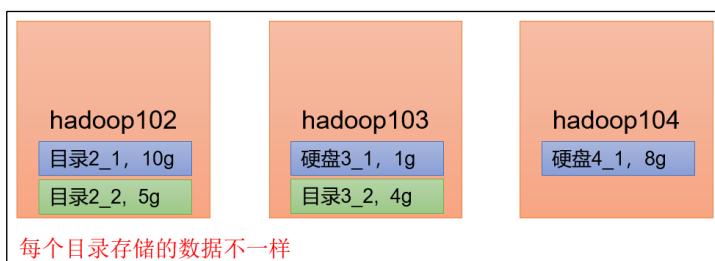
3) 查看结果

```
[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
```

检查 name1 和 name2 里面的内容，发现一模一样。

3.2 DataNode 多目录配置

1) DataNode 可以配置成多个目录，**每个目录存储的数据不一样**（数据不是副本）



2) 具体配置如下

在 hdfs-site.xml 文件中添加如下内容

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

3) 查看结果

```
[atguigu@hadoop102 dfs]$ ll
总用量 12
drwx----- 3 atguigu atguigu 4096 4月 4 14:22 data1
drwx----- 3 atguigu atguigu 4096 4月 4 14:22 data2
drwxrwxr-x 3 atguigu atguigu 4096 12月 11 08:03 name1
drwxrwxr-x 3 atguigu atguigu 4096 12月 11 08:03 name2
```

4) 向集群上传一个文件，再次观察两个文件夹里面的内容发现不一致(一个有数一个没有)

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -put
wcinput/word.txt /
```

3.3 集群数据均衡之磁盘间数据均衡

生产环境，由于硬盘空间不足，往往需要增加一块硬盘。刚加载的硬盘没有数据时，可以执行磁盘数据均衡命令。（Hadoop3.x 新特性）



(1) 生成均衡计划（我们只有一块磁盘，不会生成计划）

```
hdfs diskbalancer -plan hadoop103
```

(2) 执行均衡计划

```
hdfs diskbalancer -execute hadoop103.plan.json
```

(3) 查看当前均衡任务的执行情况

```
hdfs diskbalancer -query hadoop103
```

(4) 取消均衡任务

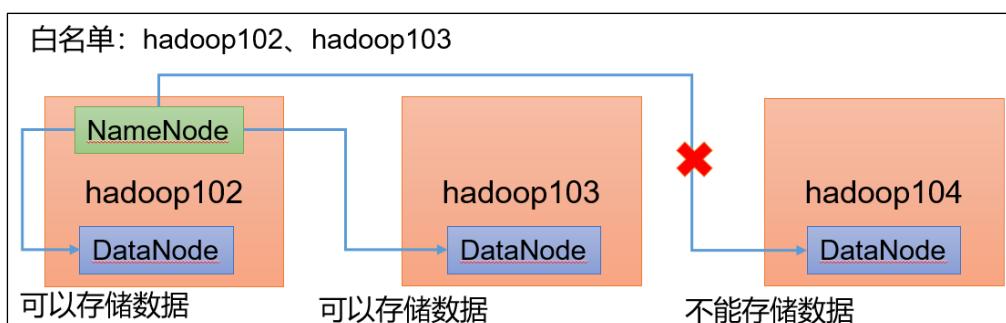
```
hdfs diskbalancer -cancel hadoop103.plan.json
```

第4章 HDFS—集群扩容及缩容

4.1 添加白名单

白名单：表示在白名单的主机 IP 地址可以，用来存储数据。

企业中：配置白名单，可以尽量防止黑客恶意访问攻击。



配置白名单步骤如下：

1) 在 NameNode 节点的 /opt/module/hadoop-3.1.3/etc/hadoop 目录下分别创建 whitelist 和 blacklist 文件

(1) 创建白名单

```
[atguigu@hadoop102 hadoop]$ vim whitelist
```

在 whitelist 中添加如下主机名称，假如集群正常工作的节点为 102 103

```
hadoop102
hadoop103
```

(2) 创建黑名单

```
[atguigu@hadoop102 hadoop]$ touch blacklist
```

保持空的就可以

2) 在 hdfs-site.xml 配置文件中增加 dfs.hosts 配置参数

```
<!-- 白名单 -->
<property>
    <name>dfs.hosts</name>
    <value>/opt/module/hadoop-3.1.3/etc/hadoop/whitelist</value>
</property>

<!-- 黑名单 -->
<property>
    <name>dfs.hosts.exclude</name>
    <value>/opt/module/hadoop-3.1.3/etc/hadoop/blacklist</value>
</property>
```

3) 分发配置文件 whitelist, hdfs-site.xml

```
[atguigu@hadoop104 hadoop]$ xsync hdfs-site.xml whitelist
```

4) 第一次添加白名单必须重启集群，不是第一次，只需要刷新 NameNode 节点即可

```
[atguigu@hadoop102 hadoop-3.1.3]$ myhadoop.sh stop
[atguigu@hadoop102 hadoop-3.1.3]$ myhadoop.sh start
```

5) 在 web 浏览器上查看 DN, http://hadoop102:9870/dfshealth.html#tab-datanode

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓hadoop102:9866 (192.168.10.102:9866)	http://hadoop102:9864	0s	2m	89.95 GB	3	61.53 KB (0%)	3.1.3
✓hadoop103:9866 (192.168.10.103:9866)	http://hadoop103:9864	2s	2m	89.95 GB	3	61.53 KB (0%)	3.1.3

Showing 1 to 2 of 2 entries

6) 在 hadoop104 上执行上传数据数据失败

```
[atguigu@hadoop104 hadoop-3.1.3]$ hadoop fs -put NOTICE.txt /
```

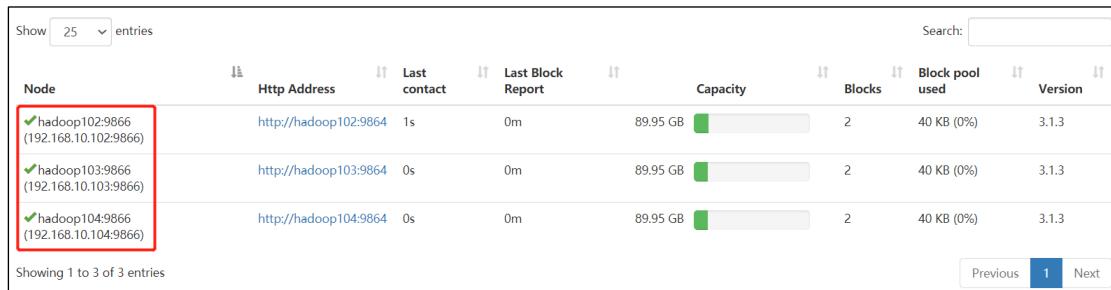
7) 二次修改白名单，增加 hadoop104

```
[atguigu@hadoop102 hadoop]$ vim whitelist
修改为如下内容
hadoop102
hadoop103
hadoop104
```

8) 刷新 NameNode

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
```

9) 在 web 浏览器上查看 DN, http://hadoop102:9870/dfshealth.html#tab-datanode



Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓ hadoop102:9866 (192.168.10.102:9866)	http://hadoop102:9866	1s	0m	89.95 GB	2	40 KB (0%)	3.1.3
✓ hadoop103:9866 (192.168.10.103:9866)	http://hadoop103:9866	0s	0m	89.95 GB	2	40 KB (0%)	3.1.3
✓ hadoop104:9866 (192.168.10.104:9866)	http://hadoop104:9866	0s	0m	89.95 GB	2	40 KB (0%)	3.1.3

4.2 服役新服务器

1) 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。

2) 环境准备

(1) 在 hadoop100 主机上再克隆一台 hadoop105 主机

(2) 修改 IP 地址和主机名称

```
[root@hadoop105 ~]# vim /etc/sysconfig/network-scripts/ifcfg-ens33
[root@hadoop105 ~]# vim /etc/hostname
```

(3) 拷贝 hadoop102 的 /opt/module 目录和 /etc/profile.d/my_env.sh 到 hadoop105

```
[atguigu@hadoop102      opt]$      scp      -r      module/*
atguigu@hadoop105:/opt/module/

[atguigu@hadoop102      opt]$  sudo  scp  /etc/profile.d/my_env.sh
root@hadoop105:/etc/profile.d/my_env.sh

[atguigu@hadoop105  hadoop-3.1.3]$ source /etc/profile
```

(4) 删除 hadoop105 上 Hadoop 的历史数据，data 和 log 数据

```
[atguigu@hadoop105 hadoop-3.1.3]$ rm -rf data/ logs/
```

(5) 配置 hadoop102 和 hadoop103 到 hadoop105 的 ssh 无密登录

```
[atguigu@hadoop102 .ssh]$ ssh-copy-id hadoop105
[atguigu@hadoop103 .ssh]$ ssh-copy-id hadoop105
```

3) 服役新节点具体步骤

(1) 直接启动 DataNode，即可关联到集群

```
[atguigu@hadoop105 hadoop-3.1.3]$ hdfs --daemon start datanode
[atguigu@hadoop105      hadoop-3.1.3]$      yarn      --daemon      start
nodemanager
```

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓ hadoop102:9866 (192.168.10.102:9866)	http://hadoop102:9864	2s	1m	89.95 GB	3	68 KB (0%)	3.1.3
✓ hadoop103:9866 (192.168.10.103:9866)	http://hadoop103:9864	0s	33m	89.95 GB	3	68 KB (0%)	3.1.3
✓ hadoop104:9866 (192.168.10.104:9866)	http://hadoop104:9864	0s	23m	89.95 GB	3	68 KB (0%)	3.1.3
✓ hadoop105:9866 (192.168.10.105:9866)	http://hadoop105:9864	2s	0m	89.95 GB	0	8 KB (0%)	3.1.3

Showing 1 to 4 of 4 entries

Previous 1 Next

4) 在白名单中增加新服役的服务器

(1) 在白名单 whitelist 中增加 hadoop104、hadoop105，并重启集群

```
[atguigu@hadoop102 hadoop]$ vim whitelist
修改为如下内容
hadoop102
hadoop103
hadoop104
hadoop105
```

(2) 分发

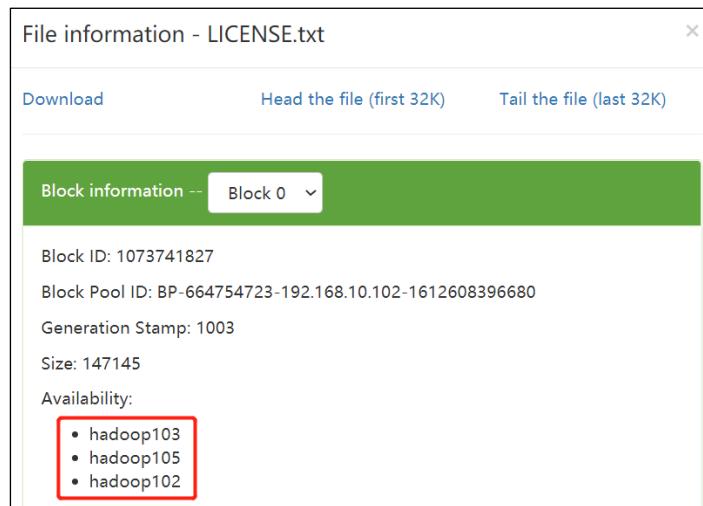
```
[atguigu@hadoop102 hadoop]$ xsync whitelist
```

(3) 刷新 NameNode

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
```

5) 在 hadoop105 上上传文件

```
[atguigu@hadoop105      hadoop-3.1.3]$      hadoop      fs      -put
/opt/module/hadoop-3.1.3/LICENSE.txt /
```



思考：如果数据不均衡（hadoop105 数据少，其他节点数据多），怎么处理？

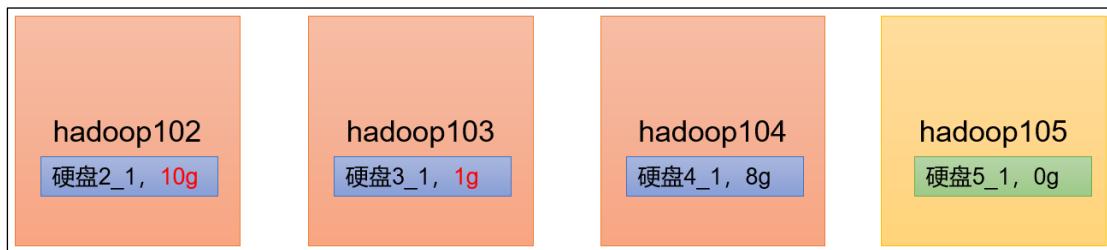
4.3 服务器间数据均衡

1) 企业经验：

在企业开发中，如果经常在 hadoop102 和 hadoop104 上提交任务，且副本数为 2，由于更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

数据本地性原则，就会导致 hadoop102 和 hadoop104 数据过多，hadoop103 存储的数据量小。

另一种情况，就是新服役的服务器数据量比较少，需要执行集群均衡命令。



2) 开启数据均衡命令：

```
[atguigu@hadoop105 hadoop-3.1.3]$ sbin/start-balancer.sh -threshold 10
```

对于参数 10，代表的是集群中各个节点的磁盘空间利用率相差不超过 10%，可根据实际情况进行调整。

3) 停止数据均衡命令：

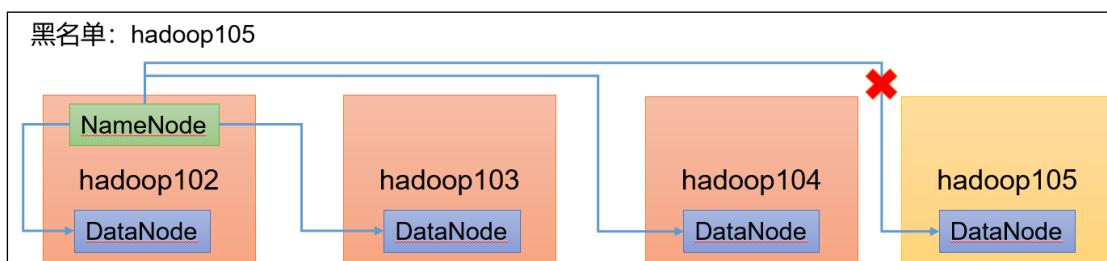
```
[atguigu@hadoop105 hadoop-3.1.3]$ sbin/stop-balancer.sh
```

注意：由于 HDFS 需要启动单独的 Rebalance Server 来执行 Rebalance 操作，所以尽量不要在 NameNode 上执行 start-balancer.sh，而是找一台比较空闲的机器。

4.4 黑名单退役服务器

黑名单：表示在黑名单的主机 IP 地址不可以，用来存储数据。

企业中：配置黑名单，用来退役服务器。



黑名单配置步骤如下：

1) 编辑 /opt/module/hadoop-3.1.3/etc/hadoop 目录下的 blacklist 文件

```
[atguigu@hadoop102 hadoop] vim blacklist
```

添加如下主机名称（要退役的节点）

```
hadoop105
```

注意：如果白名单中没有配置，需要在 hdfs-site.xml 配置文件中增加 dfs.hosts 配置参数

```
<!-- 黑名单 -->
<property>
    <name>dfs.hosts.exclude</name>
    <value>/opt/module/hadoop-3.1.3/etc/hadoop/blacklist</value>
</property>
```

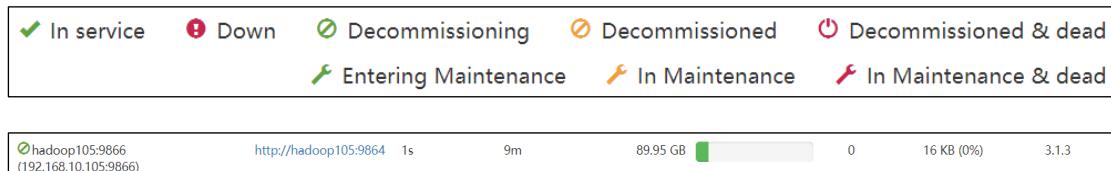
2) 分发配置文件 **blacklist, hdfs-site.xml**

```
[atguigu@hadoop104 hadoop]$ xsync hdfs-site.xml blacklist
```

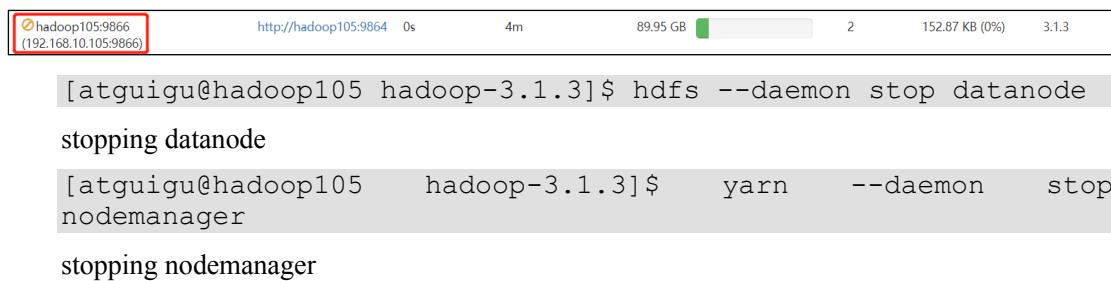
3) 第一次添加黑名单必须重启集群，不是第一次，只需要刷新 NameNode 节点即可

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
```

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



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



6) 如果数据不均衡，可以用命令实现集群的再平衡

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-balancer.sh -threshold 10
```

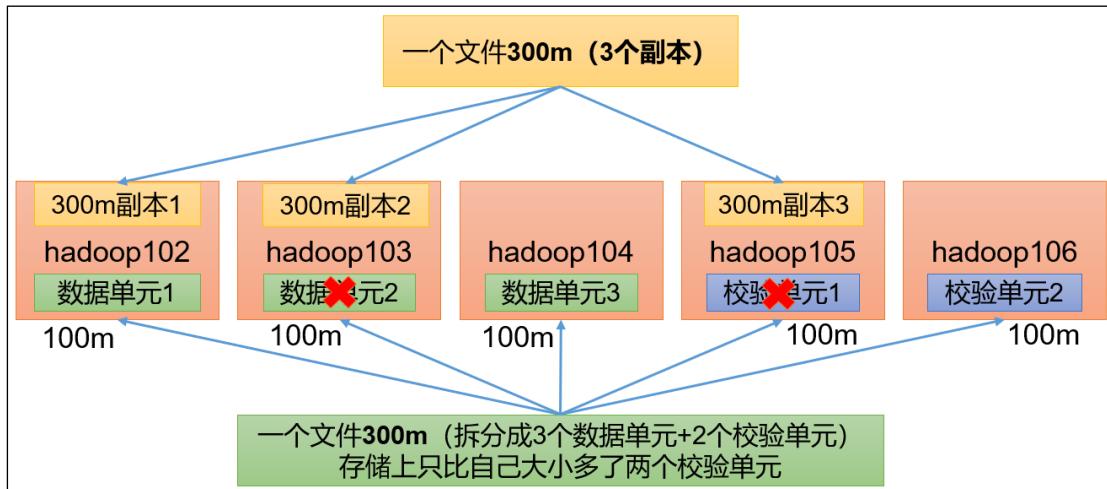
第 5 章 HDFS—存储优化

注：演示纠删码和异构存储需要一共 5 台虚拟机。尽量拿另外一套集群。提前准备 5 台服务器的集群。

5.1 纠删码

5.1.1 纠删码原理

HDFS 默认情况下，一个文件有 3 个副本，这样提高了数据的可靠性，但也带来了 2 倍的冗余开销。Hadoop3.x 引入了纠删码，采用计算的方式，可以节省约 50% 左右的存储空间。



1) 纠删码操作相关的命令

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs ec
Usage: bin/hdfs ec [COMMAND]
      [-listPolicies]
      [-addPolicies -policyFile <file>]
      [-getPolicy -path <path>]
      [-removePolicy -policy <policy>]
      [-setPolicy -path <path> [-policy <policy>] [-replicate]]
      [-unsetPolicy -path <path>]
      [-listCodecs]
      [-enablePolicy -policy <policy>]
      [-disablePolicy -policy <policy>]
      [-help <command-name>].
```

2) 查看当前支持的纠删码策略

```
[atguigu@hadoop102 hadoop-3.1.3] hdfs ec -listPolicies

Erasure Coding Policies:
ErasureCodingPolicy=[Name=RS-10-4-1024k, Schema=[ECSchema=[Codec=rs,
numDataUnits=10, numParityUnits=4]], CellSize=1048576, Id=5],
State=DISABLED

ErasureCodingPolicy=[Name=RS-3-2-1024k, Schema=[ECSchema=[Codec=rs,
numDataUnits=3, numParityUnits=2]], CellSize=1048576, Id=2],
State=DISABLED

ErasureCodingPolicy=[Name=RS-6-3-1024k, Schema=[ECSchema=[Codec=rs,
numDataUnits=6, numParityUnits=3]], CellSize=1048576, Id=1],
State=ENABLED

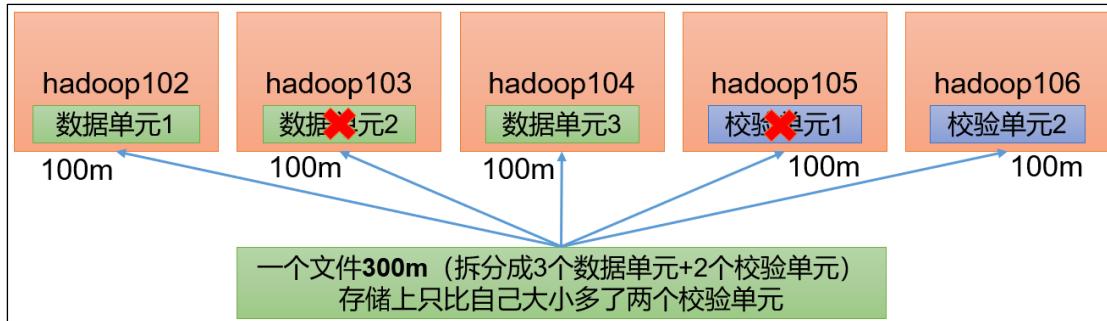
ErasureCodingPolicy=[Name=RS-LEGACY-6-3-1024k,
Schema=[ECSchema=[Codec=rs-legacy, numDataUnits=6, numParityUnits=3]],
CellSize=1048576, Id=3], State=DISABLED

ErasureCodingPolicy=[Name=XOR-2-1-1024k, Schema=[ECSchema=[Codec=xor,
numDataUnits=2, numParityUnits=1]], CellSize=1048576, Id=4],
State=DISABLED
```

3) 纠删码策略解释:

RS-3-2-1024k: 使用 RS 编码, 每 3 个数据单元, 生成 2 个校验单元, 共 5 个单元, 也就是说: 这 5 个单元中, 只要有任意的 3 个单元存在 (不管是数据单元还是校验单元, 只要

总数=3），就可以得到原始数据。每个单元的大小是 $1024k=1024*1024=1048576$ 。



RS-10-4-1024k: 使用 RS 编码，每 10 个数据单元（cell），生成 4 个校验单元，共 14 个单元，也就是说：这 14 个单元中，只要有任意的 10 个单元存在（不管是数据单元还是校验单元，只要总数=10），就可以得到原始数据。每个单元的大小是 $1024k=1024*1024=1048576$ 。

RS-6-3-1024k: 使用 RS 编码，每 6 个数据单元，生成 3 个校验单元，共 9 个单元，也就是说：这 9 个单元中，只要有任意的 6 个单元存在（不管是数据单元还是校验单元，只要总数=6），就可以得到原始数据。每个单元的大小是 $1024k=1024*1024=1048576$ 。

RS-LEGACY-6-3-1024k: 策略和上面的 RS-6-3-1024k 一样，只是编码的算法用的是 rs-legacy。

XOR-2-1-1024k: 使用 XOR 编码（速度比 RS 编码快），每 2 个数据单元，生成 1 个校验单元，共 3 个单元，也就是说：这 3 个单元中，只要有任意的 2 个单元存在（不管是数据单元还是校验单元，只要总数 = 2），就可以得到原始数据。每个单元的大小是 $1024k=1024*1024=1048576$ 。

5.1.2 纠删码案例实操



纠删码策略是给具体一个路径设置。所有往此路径下存储的文件，都会执行此策略。

默认只开启对 RS-6-3-1024k 策略的支持，如要使用别的策略需要提前启用。

1) 需求：将 /input 目录设置为 RS-3-2-1024k 策略

2) 具体步骤

(1) 开启对 RS-3-2-1024k 策略的支持

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs ec -enablePolicy -
```

```
policy RS-3-2-1024k
Erasure coding policy RS-3-2-1024k is enabled
```

(2) 在 HDFS 创建目录，并设置 RS-3-2-1024k 策略

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs dfs -mkdir /input
```

```
[atguigu@hadoop202 hadoop-3.1.3]$ hdfs ec -setPolicy -path
/input -policy RS-3-2-1024k
```

(3) 上传文件，并查看文件编码后的存储情况

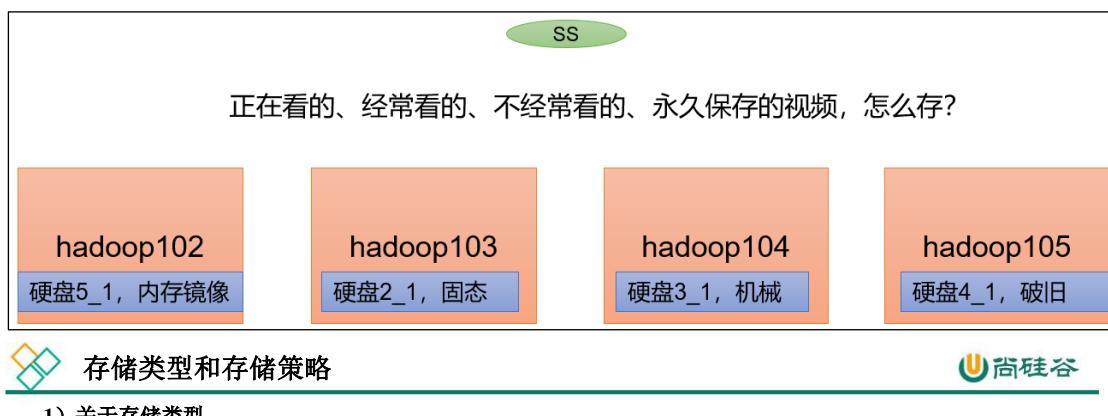
```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs dfs -put web.log /input
```

注：你所上传的文件需要大于 2M 才能看出效果。（低于 2M，只有一个数据单元和两个校验单元）

(4) 查看存储路径的数据单元和校验单元，并作破坏实验

5.2 异构存储（冷热数据分离）

异构存储主要解决，不同的数据，存储在不同类型的硬盘中，达到最佳性能的问题。



1) 关于存储类型

RAM_DISK: (内存镜像文件系统)

SSD: (SSD固态硬盘)

DISK: (普通磁盘，在HDFS中，如果没有主动声明数据目录存储类型默认都是DISK)

ARCHIVE: (没有特指哪种存储介质，主要的指的是计算能力比较弱而存储密度比较高的存储介质，用来解决数据量的容量扩增的问题，一般用于归档)

2) 关于存储策略 说明：从Lazy_Persist到Cold，分别代表了设备的访问速度从快到慢

策略ID	策略名称	副本分布	
15	Lazy_Persist	RAM_DISK:1 , DISK:n-1	一个副本保存在内存RAM_DISK中，其余副本保存在磁盘中。
12	All_SSD	SSD:n	所有副本都保存在SSD中。
10	One_SSD	SSD:1 , DISK:n-1	一个副本保存在SSD中，其余副本保存在磁盘中。
7	Hot(default)	DISK:n	Hot: 所有副本保存在磁盘中，这也是默认的存储策略。
5	Warm	DISK:1 , ARCHIVE:n-1	一个副本保存在磁盘上，其余副本保存在归档存储上。
2	Cold	ARCHIVE:n	所有副本都保存在归档存储上。

让天下没有难学的技术

5.2.1 异构存储 Shell 操作

(1) 查看当前有哪些存储策略可以用

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -
```

listPolicies

(2) 为指定路径 (数据存储目录) 设置指定的存储策略

```
hdfs storagepolicies -setStoragePolicy -path xxx -policy xxx
```

(3) 获取指定路径 (数据存储目录或文件) 的存储策略

```
hdfs storagepolicies -getStoragePolicy -path xxx
```

(4) 取消存储策略; 执行改命令之后该目录或者文件, 以其上级的目录为准, 如果是根目录, 那么就是 HOT

```
hdfs storagepolicies -unsetStoragePolicy -path xxx
```

(5) 查看文件块的分布

```
bin/hdfs fsck xxx -files -blocks -locations
```

(6) 查看集群节点

```
hadoop dfsadmin -report
```

5.2.2 测试环境准备

1) 测试环境描述

服务器规模: 5 台

集群配置: 副本数为 2, 创建好带有存储类型的目录 (提前创建)

集群规划:

节点	存储类型分配
hadoop102	RAM_DISK, SSD
hadoop103	SSD, DISK
hadoop104	DISK, RAM_DISK
hadoop105	ARCHIVE
hadoop106	ARCHIVE

2) 配置文件信息

(1) 为 hadoop102 节点的 hdfs-site.xml 添加如下信息

```
<property>
    <name>dfs.replication</name>
    <value>2</value>
</property>
<property>
    <name>dfs.storage.policy.enabled</name>
    <value>true</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>[SSD]file:///opt/module/hadoop-
3.1.3/hdfsdata/ssd, [RAM_DISK]file:///opt/module/hadoop-
3.1.3/hdfsdata/ram_disk</value>
</property>
```

(2) 为 hadoop103 节点的 hdfs-site.xml 添加如下信息

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

```
<value>2</value>
</property>
<property>
    <name>dfs.storage.policy.enabled</name>
    <value>true</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>[SSD]file:///opt/module/hadoop-
3.1.3/hdfsdata/ssd, [DISK]file:///opt/module/hadoop-
3.1.3/hdfsdata/disk</value>
</property>
```

(3) 为 hadoop104 节点的 hdfs-site.xml 添加如下信息

```
<property>
    <name>dfs.replication</name>
    <value>2</value>
</property>
<property>
    <name>dfs.storage.policy.enabled</name>
    <value>true</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>[RAM_DISK]file:///opt/module/hdfsdata/ram_disk, [DISK]file:///o
pt/module/hadoop-3.1.3/hdfsdata/disk</value>
</property>
```

(4) 为 hadoop105 节点的 hdfs-site.xml 添加如下信息

```
<property>
    <name>dfs.replication</name>
    <value>2</value>
</property>
<property>
    <name>dfs.storage.policy.enabled</name>
    <value>true</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>[ARCHIVE]file:///opt/module/hadoop-
3.1.3/hdfsdata/archive</value>
</property>
```

(5) 为 hadoop106 节点的 hdfs-site.xml 添加如下信息

```
<property>
    <name>dfs.replication</name>
    <value>2</value>
</property>
<property>
    <name>dfs.storage.policy.enabled</name>
    <value>true</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>[ARCHIVE]file:///opt/module/hadoop-
3.1.3/hdfsdata/archive</value>
</property>
```

3) 数据准备

(1) 启动集群

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs namenode -format  
[atguigu@hadoop102 hadoop-3.1.3]$ myhadoop.sh start
```

(1) 并在 HDFS 上创建文件目录

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mkdir /hdfsdata
```

(2) 并将文件资料上传

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -put /opt/module/hadoop-  
3.1.3/NOTICE.txt /hdfsdata
```

5.2.3 HOT 存储策略案例

(1) 最开始我们未设置存储策略的情况下，我们获取该目录的存储策略

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -getStoragePolicy  
-path /hdfsdata
```

(2) 我们查看上传的文件块分布

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -  
locations  
  
[DatanodeInfoWithStorage[192.168.10.104:9866, DS-0b133854-7f9e-48df-939b-  
5ca6482c5afb, DISK], DatanodeInfoWithStorage[192.168.10.103:9866, DS-  
ca1bd3b9-d9a5-4101-9f92-3da5f1baa28b, DISK]]
```

未设置存储策略，所有文件块都存储在 DISK 下。所以，**默认存储策略为 HOT**。

5.2.4 WARM 存储策略测试

(1) 接下来我们为数据降温

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy  
-path /hdfsdata -policy WARM
```

(2) 再次查看文件块分布，我们可以看到文件块依然放在原处。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -  
locations
```

(3) 我们需要让他 HDFS 按照存储策略自行移动文件块

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(4) 再次查看文件块分布，

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -  
locations  
  
[DatanodeInfoWithStorage[192.168.10.105:9866, DS-d46d08e1-80c6-4fc-a2-  
4a3dd7ec7459, ARCHIVE], DatanodeInfoWithStorage[192.168.10.103:9866, DS-  
ca1bd3b9-d9a5-4101-9f92-3da5f1baa28b, DISK]]
```

文件块一半在 DISK，一半在 ARCHIVE，符合我们设置的 WARM 策略

5.2.5 COLD 策略测试

(1) 我们继续将数据降温为 cold

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy  
-path /hdfsdata -policy COLD
```

注意：当我们设置目录为 COLD 并且我们未配置 ARCHIVE 存储目录的情况下，不可以向该目录直接上传文件，会报出异常。

(2) 手动转移

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 检查文件块的分布

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hdfs fsck /hdfsdata -files -blocks -locations
```

```
[DatanodeInfoWithStorage[192.168.10.105:9866, DS-d46d08e1-80c6-4fca-b0a2-4a3dd7ec7459, ARCHIVE], DatanodeInfoWithStorage[192.168.10.106:9866, DS-827b3f8b-84d7-47c6-8a14-0166096f919d, ARCHIVE]]
```

所有文件块都在 ARCHIVE，符合 COLD 存储策略。

5.2.6 ONE_SSD 策略测试

(1) 接下来我们将存储策略从默认的 HOT 更改为 One_SSD

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy -path /hdfsdata -policy One_SSD
```

(2) 手动转移文件块

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 转移完成后，我们查看文件块分布，

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hdfs fsck /hdfsdata -files -blocks -locations
```

```
[DatanodeInfoWithStorage[192.168.10.104:9866, DS-0b133854-7f9e-48df-939b-5ca6482c5afb, DISK], DatanodeInfoWithStorage[192.168.10.103:9866, DS-2481a204-59dd-46c0-9f87-ec4647ad429a, SSD]]
```

文件块分布为一半在 SSD，一半在 DISK，符合 One_SSD 存储策略。

5.2.7 ALL_SSD 策略测试

(1) 接下来，我们再将存储策略更改为 All_SSD

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy -path /hdfsdata -policy All_SSD
```

(2) 手动转移文件块

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 查看文件块分布，我们可以看到，

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hdfs fsck /hdfsdata -files -blocks -locations
```

```
[DatanodeInfoWithStorage[192.168.10.102:9866, DS-c997cfb4-16dc-4e69-a0c4-9411a1b0c1eb, SSD], DatanodeInfoWithStorage[192.168.10.103:9866, DS-2481a204-59dd-46c0-9f87-ec4647ad429a, SSD]]
```

所有的文件块都存储在 SSD，符合 All_SSD 存储策略。

5.2.8 LAZY_PERSIST 策略测试

(1) 继续改变策略，将存储策略改为 lazy_persist

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs storagepolicies -setStoragePolicy -path /hdfsdata -policy lazy_persist
```

(2) 手动转移文件块

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs mover /hdfsdata
```

(3) 查看文件块分布

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs fsck /hdfsdata -files -blocks -locations
```

```
[DatanodeInfoWithStorage[192.168.10.104:9866, DS-0b133854-7f9e-48df-939b-5ca6482c5afb, DISK], DatanodeInfoWithStorage[192.168.10.103:9866, DS-ca1bd3b9-d9a5-4101-9f92-3da5f1baa28b, DISK]]
```

这里我们发现所有的文件块都是存储在 DISK，按照理论一个副本存储在 RAM_DISK，其他副本存储在 DISK 中，这是因为，我们还需要配置“dfs.datanode.max.locked.memory”，“dfs.block.size”参数。

那么出现存储策略为 LAZY_PERSIST 时，文件块副本都存储在 DISK 上的原因有如下两点：

(1) 当客户端所在的 DataNode 节点没有 RAM_DISK 时，则会写入客户端所在的 DataNode 节点的 DISK 磁盘，其余副本会写入其他节点的 DISK 磁盘。

(2) 当客户端所在的 DataNode 有 RAM_DISK，但“dfs.datanode.max.locked.memory”参数值未设置或者设置过小（小于“dfs.block.size”参数值）时，则会写入客户端所在的 DataNode 节点的 DISK 磁盘，其余副本会写入其他节点的 DISK 磁盘。

但是由于虚拟机的“max locked memory”为 64KB，所以，如果参数配置过大，还会报出错误：

```
ERROR org.apache.hadoop.hdfs.server.datanode.DataNode: Exception in
secureMain
java.lang.RuntimeException: Cannot start datanode because the configured
max locked memory size (dfs.datanode.max.locked.memory) of 209715200
bytes is more than the datanode's available RLIMIT_MEMLOCK ulimit of
65536 bytes.
```

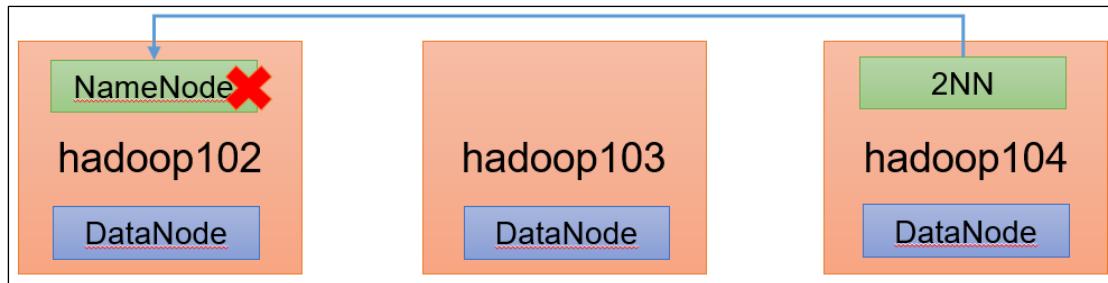
我们可以通过该命令查询此参数的内存

```
[atguigu@hadoop102 hadoop-3.1.3]$ ulimit -a
max locked memory      (kbytes, -1)  64
```

第 6 章 HDFS—故障排除

注意：采用三台服务器即可，恢复到 Yarn 开始的服务器快照。

6.1 NameNode 故障处理



1) 需求:

NameNode 进程挂了并且存储的数据也丢失了，如何恢复 NameNode

2) 故障模拟

- (1) kill -9 NameNode 进程

```
[atguigu@hadoop102 current]$ kill -9 19886
```

- (2) 删除 NameNode 存储的数据 (/opt/module/hadoop-3.1.3/data/tmp/dfs/name)

```
[atguigu@hadoop102 hadoop-3.1.3]$ rm -rf /opt/module/hadoop-3.1.3/data/dfs/name/*
```

3) 问题解决

- (1) 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-3.1.3/data/dfs/namesecondary/* ./name/
```

- (2) 重新启动 NameNode

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs --daemon start namenode
```

- (3) 向集群上传一个文件

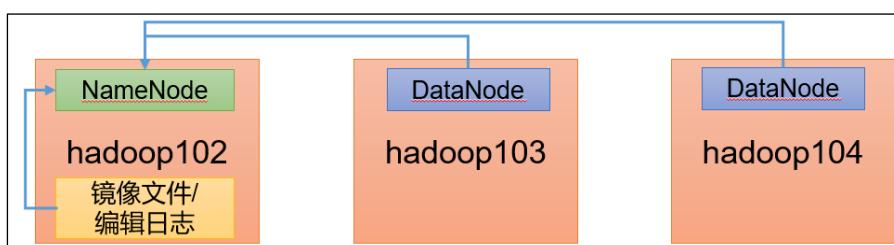
6.2 集群安全模式&磁盘修复

1) 安全模式:

文件系统只接受读数据请求，而不接受删除、修改等变更请求

2) 进入安全模式场景

- NameNode 在加载镜像文件和编辑日志期间处于安全模式；
- NameNode 再接收 DataNode 注册时，处于安全模式



3) 退出安全模式条件

dfs.namenode.safemode.min.datanodes:最小可用 datanode 数量， 默认 0

dfs.namenode.safemode.threshold-pct:副本数达到最小要求的 block 占系统总 block 数的百分比， 默认 0.999f。 (只允许丢一个块)

dfs.namenode.safemode.extension:稳定时间， 默认值 30000 毫秒， 即 30 秒

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

5) 案例 1：启动集群进入安全模式

(1) 重新启动集群

```
[atguigu@hadoop102 subdir0]$ myhadoop.sh stop
[atguigu@hadoop102 subdir0]$ myhadoop.sh start
```

(2) 集群启动后，立即来到集群上删除数据，提示集群处于安全模式

The screenshot shows the Hadoop Web UI's 'Browse Directory' page. At the top, there is a red-bordered warning message: "Cannot delete /NOTICE.txt. Name node is in safe mode. The reported blocks 0 needs additional 4 blocks to reach the threshold 0.9990 of total blocks 5. The minimum number of live datanodes is not required. Safe mode will be turned off automatically once the thresholds have been reached. NamenodeHostName:hadoop102". Below the message is a file listing for the root directory. The table has columns: Permission, Owner, Group, Size, Last Modified, Replication, Block Size, and Name. A single file named 'NOTICE.txt' is listed with the following details:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	atguigu	supergroup	21.35 KB	Feb 14 10:58	3	128 MB	NOTICE.txt

6) 案例 2：磁盘修复

需求：数据块损坏，进入安全模式，如何处理

(1) 分别进入 hadoop102、hadoop103、hadoop104 的 /opt/module/hadoop-3.1.3/data/dfs/data/current 目录，统一删除某 2 个块信息

```
[atguigu@hadoop102 subdir0]$ pwd
/opt/module/hadoop-3.1.3/data/dfs/data/current/BP-1015489500-192.168.10.102-
1611909480872/current/finalized/subdir0/subdir0
```

```
[atguigu@hadoop102 subdir0]$ rm -rf blk_1073741847
blk_1073741847_1023.meta
[atguigu@hadoop102 subdir0]$ rm -rf blk_1073741865
blk_1073741865_1042.meta
```

说明：hadoop103/hadoop104 重复执行以上命令

(2) 重新启动集群

```
[atguigu@hadoop102 subdir0]$ myhadoop.sh stop
[atguigu@hadoop102 subdir0]$ myhadoop.sh start
```

(3) 观察 <http://hadoop102:9870/dfshealth.html#tab-overview>

Summary

Security is off.

Safe mode is ON The reported blocks 5 needs additional 1 blocks to reach the threshold 0.9990 of total blocks 7. The minimum number of live datanodes is not required. Safe mode will be turned off automatically once the thresholds have been reached.

说明：安全模式已经打开，块的数量没有达到要求。

(4) 离开安全模式

```
[atguigu@hadoop102 subdir0]$ hdfs dfsadmin -safemode get
Safe mode is ON
[atguigu@hadoop102 subdir0]$ hdfs dfsadmin -safemode leave
Safe mode is OFF
```

(5) 观察 <http://hadoop102:9870/dfshealth.html#tab-overview>

The screenshot shows the HDFS Health Overview page at <http://hadoop102:9870/dfshealth.html#tab-overview>. A message box indicates there are 2 missing blocks. The list of missing blocks includes 'blk_1073741847 /tmp/logs/atguigu/logs-tfile/application_1611912355782_0001/hadoop103_45020' and 'blk_1073741865 /input/word.txt'. A note at the bottom suggests checking logs or running fsck.

(6) 将元数据删除

The screenshot shows the Browse Directory page at <http://hadoop102:9870/>. It lists a single entry: a file named 'hadoop103_45020' with size 96.69 KB, replication 3, and block size 128 MB. The delete icon for this file is highlighted with a red box.

Browse Directory

/input								Go!			
Show 25 entries								Search:			
	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name			
<input type="checkbox"/>	-rw-r--r--	atguigu	supergroup	36 B	Jan 30 18:34	3	128 MB	word.txt			

Showing 1 to 1 of 1 entries

Previous 1 Next

(7) 观察 <http://hadoop102:9870/dfshealth.html#tab-overview>, 集群已经正常

7) 案例 3:

需求：模拟等待安全模式

(1) 查看当前模式

```
[atguigu@hadoop102 hadoop-3.1.3]$ hdfs dfsadmin -safemode get
Safe mode is OFF
```

(2) 先进入安全模式

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hdfs dfsadmin -safemode enter
```

(3) 创建并执行下面的脚本

在/opt/module/hadoop-3.1.3 路径上，编辑一个脚本 safemode.sh

```
[atguigu@hadoop102 hadoop-3.1.3]$ vim safemode.sh

#!/bin/bash
hdfs dfsadmin -safemode wait
hdfs dfs -put /opt/module/hadoop-3.1.3/README.txt /

[atguigu@hadoop102 hadoop-3.1.3]$ chmod 777 safemode.sh

[atguigu@hadoop102 hadoop-3.1.3]$ ./safemode.sh
```

(4) 再打开一个窗口，执行

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hdfs dfsadmin -safemode leave
```

(5) 再观察上一个窗口

```
Safe mode is OFF
```

(6) HDFS 集群上已经有上传的数据了

	-rw-r--r--	atguigu	supergroup	1.33 KB	Feb 14 09:48	3	128 MB	README.txt		
--	------------	---------	------------	---------	--------------	---	--------	------------	--	--

6.3 慢磁盘监控

“慢磁盘”指的时写入数据非常慢的一类磁盘。其实慢性磁盘并不少见，当机器运行时间长了，上面跑的任务多了，磁盘的读写性能自然会退化，严重时就会出现写入数据延时的

问题。

如何发现慢磁盘？

正常在 HDFS 上创建一个目录，只需要不到 1s 的时间。如果你发现创建目录超过 1 分钟及以上，而且这个现象并不是每次都有。只是偶尔慢了一下，就很有可能存在慢磁盘。
可以采用如下方法找出是哪块磁盘慢：

1) 通过心跳未联系时间。

一般出现慢磁盘现象，会影响到 DataNode 与 NameNode 之间的心跳。正常情况心跳时间间隔是 3s。超过 3s 说明有异常。

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓ hadoop102:9866 (192.168.202.102:9866)	http://hadoop102:9864	1s	2m	88.34 GB	19	1.98 GB (2.24%)	3.1.3
✓ hadoop103:9866 (192.168.202.103:9866)	http://hadoop103:9864	1s	2m	88.34 GB	19	1.98 GB (2.24%)	3.1.3
✓ hadoop104:9866 (192.168.202.104:9866)	http://hadoop104:9864	1s	2m	88.34 GB	19	1.9 GB (2.16%)	3.1.3

2) fio 命令，测试磁盘的读写性能

(1) 顺序读测试

```
[atguigu@hadoop102 ~]# sudo yum install -y fio
[atguigu@hadoop102 ~]# sudo fio -
filename=/home/atguigu/test.log -direct=1 -iodepth 1 -thread -
rw=read -ioengine=psync -bs=16k -size=2G -numjobs=10 -
runtime=60 -group_reporting -name=test_r

Run status group 0 (all jobs):
  READ: bw=360MiB/s (378MB/s), 360MiB/s-360MiB/s (378MB/s-378MB/s),
  io=20.0GiB (21.5GB), run=56885-56885msec
```

结果显示，磁盘的总体顺序读速度为 **360MiB/s**。

(2) 顺序写测试

```
[atguigu@hadoop102 ~]# sudo fio -
filename=/home/atguigu/test.log -direct=1 -iodepth 1 -thread -
rw=write -ioengine=psync -bs=16k -size=2G -numjobs=10 -
runtime=60 -group_reporting -name=test_w

Run status group 0 (all jobs):
  WRITE: bw=341MiB/s (357MB/s), 341MiB/s-341MiB/s (357MB/s-
  357MB/s), io=19.0GiB (21.4GB), run=60001-60001msec
```

结果显示，磁盘的总体顺序写速度为 **341MiB/s**。

(3) 随机写测试

```
[atguigu@hadoop102 ~]# sudo fio -
filename=/home/atguigu/test.log -direct=1 -iodepth 1 -thread -
```

```
rw=randwrite -ioengine=psync -bs=16k -size=2G -numjobs=10 -
runtime=60 -group_reporting -name=test_randw
```

```
Run status group 0 (all jobs):
  WRITE: bw=309MiB/s (324MB/s), 309MiB/s-309MiB/s (324MB/s-324MB/s),
  io=18.1GiB (19.4GB), run=60001-60001msec
```

结果显示，磁盘的总体随机写速度为 **309MiB/s**。

(4) 混合随机读写：

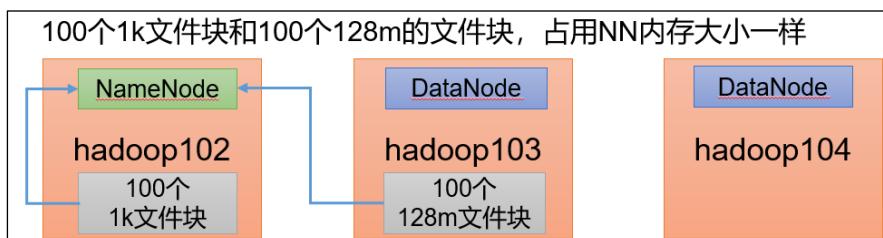
```
[atguigu@hadoop102 ~]# sudo fio -
filename=/home/atguigu/test.log -direct=1 -iodepth 1 -thread -
rw=randrw -rwmixread=70 -ioengine=psync -bs=16k -size=2G -
numjobs=10 -runtime=60 -group_reporting -name=test_r_w -
ioscheduler=noop

Run status group 0 (all jobs):
  READ: bw=220MiB/s (231MB/s), 220MiB/s-220MiB/s (231MB/s-
231MB/s), io=12.9GiB (13.9GB), run=60001-60001msec
  WRITE: bw=94.6MiB/s (99.2MB/s), 94.6MiB/s-94.6MiB/s
(99.2MB/s-99.2MB/s), io=5674MiB (5950MB), run=60001-60001msec
```

结果显示，磁盘的总体混合随机读写，读速度为 **220MiB/s**，写速度 **94.6MiB/s**。

6.4 小文件归档

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在 NameNode 的内存中，因此 HDFS 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数据块的大小无关。例如，一个 1MB 的文件设置为 128MB 的块存储，实际使用的是 1MB 的磁盘空间，而不是 128MB。

2) 解决存储小文件办法之一

HDFS 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，HDFS 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-3.1.3]$ start-yarn.sh
```

(2) 归档文件

把目录里面的所有文件归档成一个叫 input.har 的归档文件，并把归档后文件存储到/output 路径下。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop archive -archiveName input.har -p /input /output
```

(3) 查看归档

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls /output/input.har
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls har://output/input.har
```

(4) 解归档文件

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -cp har://output/input.har/* /
```

第 7 章 HDFS—集群迁移

7.1 Apache 和 Apache 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 distcp 命令实现两个 Hadoop 集群之间的递归数据复制

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hadoop distcp
hdfs://hadoop102:8020/user/atguigu/hello.txt
hdfs://hadoop105:8020/user/atguigu/hello.txt
```

7.2 Apache 和 CDH 集群间数据拷贝



尚硅谷大数据技术
之集群迁移 (Apache)

第 8 章 MapReduce 生产经验

8.1 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、网络

2) I/O 操作优化

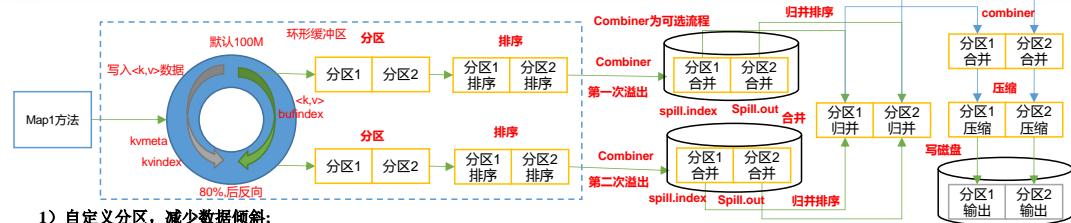
(1) 数据倾斜

(2) Map 运行时间太长，导致 Reduce 等待过久

(3) 小文件过多

8.2 MapReduce 常用调优参数

MapReduce 优化 (上)



1) 自定义分区，减少数据倾斜；
定义类，继承Partitioner接口，重写getPartition方法

2) 减少溢写的次数

mapreduce.task.io.sort.mb
Shuffle的环形缓冲区大小，默认100m，可以提高到200m
mapreduce.map.sort.spill.percent
环形缓冲区溢出的阈值，默认80%，可以提高的90%

3) 增加每次Merge合并次数

mapreduce.task.io.sort.factor默认10，可以提高到20

4) 在不影响业务结果的前提下可以提前采用Combiner

job.setCombinerClass(xxxReducer.class);

5) 为了减少磁盘IO，可以采用Snappy或者LZO压缩

conf.setBoolean("mapreduce.map.output.compress", true);
conf.setClass("mapreduce.map.output.compress.codec", SnappyCodec.class, CompressionCodec.class);

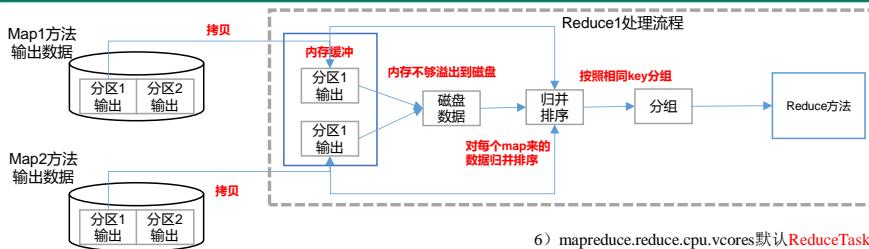
6) mapreduce.map.memory.mb 默认MapTask内存上限1024MB。
可以根据128m数据对应1G内存原则提高该内存。

7) mapreduce.map.java.opts：控制MapTask堆内存大小。（如果内存不够，报：java.lang.OutOfMemoryError）

8) mapreduce.map.cpu.vcores 默认MapTask的CPU核数1。计算密集型任务可以增加CPU核数

9) 异常重试
mapreduce.map.maxattempts每个Map Task最大重试次数，一旦重试次数超过该值，则认为Map Task运行失败，**默认值：4**。根据机器性能适当提高。

MapReduce优化 (下)



- 1) mapreduce.reduce.shuffle.parallelcopies 每个Reduce去Map中拉取数据的并行数，**默认值是5**。可以提高到**10**。
- 2) mapreduce.reduce.shuffle.input.buffer.percent Buffer大小占Reduce可用内存的比例，**默认值0.7**。可以提高到**0.8**
- 3) 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**。可以提高到**0.75**
- 4) mapreduce.reduce.memory.mb 默认ReduceTask内存上限**1024MB**，根据128m数据对应1G内存原则，**适当提高内存到4-6G**
- 5) mapreduce.reduce.java.opts: 控制ReduceTask堆内存大小。（如果内存不够，报：java.lang.OutOfMemoryError）
- 6) mapreduce.reduce.cpu.vcores 默认ReduceTask的CPU核数1个。可以提高到**2-4个**
- 7) mapreduce.reduce.maxattempts 每个Reduce Task最大重试次数，一旦重试次数超过该值，则认为Map Task运行失败，**默认值：4**。
- 8) mapreduce.job.reduce.slowstart.completedmaps 当MapTask完成的比例达到该值后才会为ReduceTask申请资源。**默认是0.05**。
- 9) mapreduce.task.timeout 如果一个Task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该Task处于Block状态，可能是卡住了，也许永远会卡住，为了防止因为用户程序永远Block住不退出，则强制设置了一个该超时时间（单位毫秒），**默认是600000（10分钟）**。如果你的程序对每条输入数据的处理时间过长，建议将该参数调大。
- 10) 如果可以不用Reduce，尽可能不用

天下没有难学的技术

8.3 MapReduce 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。

[2K----- [2K[31;1m VERTICES: 23/25 [=====>>-] 99% ELAPSED TIME: 5217.72 s [22;0m[2K [31A[2K----- [2K[36;1m VERTICES MODE STATUS TOTAL COMPLETED RUNNING PENDING FAILED KILLED [22;0m[2K [2KMap 6 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 7 container SUCCEEDED 31 31 0 0 0 0 0 0 Map 8 container SUCCEEDED 10 10 0 0 0 0 0 0 Map 9 container SUCCEEDED 2 2 0 0 0 0 0 0 Map 10 container SUCCEEDED 43 43 0 0 0 0 0 0 Map 11 container SUCCEEDED 130 130 0 0 0 0 0 0 Map 12 container SUCCEEDED 130 130 0 0 0 0 0 0 Map 13 container SUCCEEDED 60 60 0 0 0 0 0 0 Map 14 container SUCCEEDED 130 130 0 0 0 0 0 0 Map 15 container SUCCEEDED 2 2 0 0 0 0 0 0 Map 16 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 17 container SUCCEEDED 2 2 0 0 0 0 0 0 Map 18 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 1 container SUCCEEDED 156 156 0 0 0 0 0 0 Reducer 5 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 19 container SUCCEEDED 23 23 0 0 0 0 0 0 Map 20 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 21 container SUCCEEDED 2 2 0 0 0 0 0 0 Map 22 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 23 container SUCCEEDED 1 1 0 0 0 0 0 0 Map 24 container SUCCEEDED 1 1 0 0 0 0 0 0 Reducer 2 container SUCCEEDED 1009 1009 0 0 0 0 0 0 Map 25 container SUCCEEDED 60 60 0 0 0 0 0 0 Reducer 3 container RUNNING 1009 1008 1 0 0 0 0 Reducer 4 container RUNNING 1 0 1 0 0 0 0 0 [2K----- [2K[31;1m VERTICES: 23/25 [=====>>-] 99% ELAPSED TIME: 5222.73 s [22;0m[2K [31A[2K-----]
[2K----- [2K[31;1m VERTICES: 23/25 [=====>>-] 99% ELAPSED TIME: 5217.72 s [22;0m[2K [31A[2K----- [2K[36;1m VERTICES MODE STATUS TOTAL COMPLETED RUNNING PENDING FAILED KILLED [22;0m[2K [2KMap 6 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 7 container SUCCEEDED 31 31 0 0 0 0 0 0 0 Map 8 container SUCCEEDED 10 10 0 0 0 0 0 0 0 Map 9 container SUCCEEDED 2 2 0 0 0 0 0 0 0 Map 10 container SUCCEEDED 43 43 0 0 0 0 0 0 0 Map 11 container SUCCEEDED 130 130 0 0 0 0 0 0 0 Map 12 container SUCCEEDED 130 130 0 0 0 0 0 0 0 Map 13 container SUCCEEDED 60 60 0 0 0 0 0 0 0 Map 14 container SUCCEEDED 130 130 0 0 0 0 0 0 0 Map 15 container SUCCEEDED 2 2 0 0 0 0 0 0 0 Map 16 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 17 container SUCCEEDED 2 2 0 0 0 0 0 0 0 Map 18 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 1 container SUCCEEDED 156 156 0 0 0 0 0 0 0 Reducer 5 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 19 container SUCCEEDED 23 23 0 0 0 0 0 0 0 Map 20 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 21 container SUCCEEDED 2 2 0 0 0 0 0 0 0 Map 22 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 23 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Map 24 container SUCCEEDED 1 1 0 0 0 0 0 0 0 Reducer 2 container SUCCEEDED 1009 1009 0 0 0 0 0 0 0 Map 25 container SUCCEEDED 60 60 0 0 0 0 0 0 0 Reducer 3 container RUNNING 1009 1008 1 0 0 0 0 Reducer 4 container RUNNING 1 0 1 0 0 0 0 0 0 [2K----- [2K[31;1m VERTICES: 23/25 [=====>>-] 99% ELAPSED TIME: 5222.73 s [22;0m[2K [31A[2K-----]

2) 减少数据倾斜的方法

(1) 首先检查是否空值过多造成的数据倾斜

生产环境，可以直接过滤掉空值；如果想保留空值，就自定义分区，将空值加随机数打散。最后再二次聚合。

(2) 能在 map 阶段提前处理，最好先在 Map 阶段处理。如：Combiner、MapJoin

(3) 设置多个 reduce 个数

第 9 章 Hadoop-Yarn 生产经验

9.1 常用的调优参数

1) 调优参数列表

(1) Resourcemanager 相关

```
yarn.resourcemanager.scheduler.client.thread-count ResourceManager 处理调度器请求的线程数量  
yarn.resourcemanager.scheduler.class 配置调度器
```

(2) Nodemanager 相关

yarn.nodemanager.resource.memory-mb	NodeManager 使用内存数
yarn.nodemanager.resource.system-reserved-memory-mb	NodeManager 为系统保留多少内存，和上一个参数二者取一即可
yarn.nodemanager.resource.cpu-vcores	NodeManager 使用 CPU 核数
yarn.nodemanager.resource.count-logical-processors-as-cores	是否将虚拟核数当作 CPU 核数
yarn.nodemanager.resource.pcores-vcores-multiplier	虚拟核数和物理核数乘数，例如：4 核 8 线程，该参数就应设为 2
yarn.nodemanager.resource.detect-hardware-capabilities	是否让 yarn 自己检测硬件进行配置
yarn.nodemanager.pmem-check-enabled	是否开启物理内存检查限制 container
yarn.nodemanager.vmem-check-enabled	是否开启虚拟内存检查限制 container
yarn.nodemanager.vmem-pmem-ratio	虚拟内存物理内存比例

(3) Container 容器相关

yarn.scheduler.minimum-allocation-mb	容器最小内存
yarn.scheduler.maximum-allocation-mb	容器最大内存
yarn.scheduler.minimum-allocation-vcores	容器最小核数
yarn.scheduler.maximum-allocation-vcores	容器最大核数

2) 参数具体使用案例

详见《尚硅谷大数据技术之 Hadoop (Yarn)》，第 2.1 节。

9.2 容量调度器使用

详见《尚硅谷大数据技术之 Hadoop (Yarn)》，第 2.2 节。

9.3 公平调度器使用

详见《尚硅谷大数据技术之 Hadoop (Yarn)》，第 2.3 节。

第 10 章 Hadoop 综合调优

10.1 Hadoop 小文件优化方法

10.1.1 Hadoop 小文件弊端

HDFS 上每个文件都要在 NameNode 上创建对应的元数据，这个元数据的大小约为 150byte，这样当小文件比较多的时候，就会产生很多的元数据文件，**一方面会大量占用 NameNode 的内存空间，另一方面就是元数据文件过多，使得寻址索引速度变慢。**

小文件过多，在进行 MR 计算时，会生成过多切片，需要启动过多的 MapTask。每个 MapTask 处理的数据量小，**导致 MapTask 的处理时间比启动时间还小，白白消耗资源。**

10.1.2 Hadoop 小文件解决方案

1) 在数据采集的时候，就将小文件或小批数据合成大文件再上传 HDFS (数据源头)

2) Hadoop Archive (存储方向)

是一个高效的将小文件放入 HDFS 块中的文件存档工具，能够将多个小文件打包成一个 HAR 文件，从而达到减少 NameNode 的内存使用

3) CombineTextInputFormat (计算方向)

CombineTextInputFormat 用于将多个小文件在切片过程中生成一个单独的切片或者少量的切片。

4) 开启 uber 模式，实现 JVM 重用 (计算方向)

默认情况下，每个 Task 任务都需要启动一个 JVM 来运行，如果 Task 任务计算的数据量很小，我们可以让同一个 Job 的多个 Task 运行在一个 JVM 中，不必为每个 Task 都开启一个 JVM。

(1) 未开启 uber 模式，在 /input 路径上上传多个小文件并执行 wordcount 程序

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar  
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar  
wordcount /input /output2
```

(2) 观察控制台

```
2021-02-14 16:13:50,607 INFO mapreduce.Job: Job job_1613281510851_0002  
running in uber mode : false
```

(3) 观察 <http://hadoop103:8088/cluster>

Show 20 ▾ entries				Show 20 ▾ entries	
ID	User	Name	Application Type	Attempt ID	Started
application_1613281510851_0002	atguigu	word count	MAPREDUCE	appattempt_1613281510851_0002_000001	Sun Feb 14 16:13:45 +0800 2021

Total Allocated Containers: 5
Each table cell represents the number of NodeLocal/RackLocal/OffSwitch containers satisfied by NodeLocal/RackLocal/OffSwitch resource requests.

(4) 开启 uber 模式，在 mapred-site.xml 中添加如下配置

```
<!-- 开启 uber 模式， 默认关闭 -->
<property>
    <name>mapreduce.job.ubertask.enable</name>
    <value>true</value>
</property>

<!-- uber 模式中最大的 mapTask 数量， 可向下修改 -->
<property>
    <name>mapreduce.job.ubertask.maxmaps</name>
    <value>9</value>
</property>
<!-- uber 模式中最大的 reduce 数量， 可向下修改 -->
<property>
    <name>mapreduce.job.ubertask.maxreduces</name>
    <value>1</value>
</property>
<!-- uber 模式中最大的输入数据量， 默认使用 dfs.blocksize 的值， 可向下修改 -->
<property>
    <name>mapreduce.job.ubertask.maxbytes</name>
    <value></value>
</property>
```

(5) 分发配置

```
[atguigu@hadoop102 hadoop]$ xsync mapred-site.xml
```

(6) 再次执行 wordcount 程序

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar
wordcount /input /output2
```

(7) 观察控制台

```
2021-02-14 16:28:36,198 INFO mapreduce.Job: Job
job_1613281510851_0003 running in uber mode : true
```

(8) 观察 <http://hadoop103:8088/cluster>

Total Allocated Containers: 1
Each table cell represents the number of NodeLocal/RackLocal/OffSwitch containers satisfied by NodeLocal/RackLocal/OffSwitch resource requests.

10.2 测试 MapReduce 计算性能

使用 Sort 程序评测 MapReduce

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

注：一个虚拟机不超过 150G 磁盘尽量不要执行这段代码

- (1) 使用 RandomWriter 来产生随机数，每个节点运行 10 个 Map 任务，每个 Map 产生大约 1G 大小的二进制随机数

```
[atguigu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar randomwriter random-data
```

- (2) 执行 Sort 程序

```
[atguigu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar sort random-data sorted-data
```

- (3) 验证数据是否真正排好序了

```
[atguigu@hadoop102 mapreduce]$ hadoop jar /opt/module/hadoop-3.1.3/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-3.1.3-tests.jar testmapredsort -sortInput random-data -sortOutput sorted-data
```

10.3 企业开发场景案例

10.3.1 需求

- (1) 需求：从 1G 数据中，统计每个单词出现次数。服务器 3 台，每台配置 4G 内存，4 核 CPU，4 线程。

- (2) 需求分析：

$1G / 128m = 8$ 个 MapTask； 1 个 ReduceTask； 1 个 mrAppMaster

平均每个节点运行 10 个 / 3 台 \approx 3 个任务 (4 3 3)

10.3.2 HDFS 参数调优

- (1) 修改：hadoop-env.sh

```
export HDFS_NAMENODE_OPTS="-Dhadoop.security.logger=INFO,RFAS -Xmx1024m"
export HDFS_DATANODE_OPTS="-Dhadoop.security.logger=ERROR,RFAS -Xmx1024m"
```

- (2) 修改 hdfs-site.xml

```
<!-- NameNode 有一个工作线程池， 默认值是 10 -->
<property>
    <name>dfs.namenode.handler.count</name>
    <value>21</value>
</property>
```

- (3) 修改 core-site.xml

```
<!-- 配置垃圾回收时间为 60 分钟 -->
<property>
    <name>fs.trash.interval</name>
```

```
<value>60</value>
</property>
```

(4) 分发配置

```
[atguigu@hadoop102 hadoop]$ xsync hadoop-env.sh hdfs-site.xml
core-site.xml
```

10.3.3 MapReduce 参数调优

(1) 修改 mapred-site.xml

```
<!-- 环形缓冲区大小， 默认 100m -->
<property>
  <name>mapreduce.task.io.sort.mb</name>
  <value>100</value>
</property>

<!-- 环形缓冲区溢写阈值， 默认 0.8 -->
<property>
  <name>mapreduce.map.sort.spill.percent</name>
  <value>0.80</value>
</property>

<!-- merge 合并次数， 默认 10 个 -->
<property>
  <name>mapreduce.task.io.sort.factor</name>
  <value>10</value>
</property>

<!-- maptask 内存， 默认 1g； maptask 堆内存大小默认和该值大小一致
mapreduce.map.java.opts -->
<property>
  <name>mapreduce.map.memory.mb</name>
  <value>-1</value>
  <description>The amount of memory to request from the
scheduler for each map task. If this is not specified or is
non-positive, it is inferred from mapreduce.map.java.opts and
mapreduce.job.heap.memory-mb.ratio. If java-opts are also not
specified, we set it to 1024.
  </description>
</property>

<!-- matask 的 CPU 核数， 默认 1 个 -->
<property>
  <name>mapreduce.map.cpu.vcores</name>
  <value>1</value>
</property>

<!-- matask 异常重试次数， 默认 4 次 -->
<property>
  <name>mapreduce.map.maxattempts</name>
  <value>4</value>
</property>

<!-- 每个 Reduce 去 Map 中拉取数据的并行数。默认值是 5 -->
<property>
  <name>mapreduce.reduce.shuffle.parallelcopies</name>
```

```
<value>5</value>
</property>

<!-- Buffer 大小占 Reduce 可用内存的比例，默认值 0.7 -->
<property>
    <name>mapreduce.reduce.shuffle.input.buffer.percent</name>
    <value>0.70</value>
</property>

<!-- Buffer 中的数据达到多少比例开始写入磁盘，默认值 0.66。 -->
<property>
    <name>mapreduce.reduce.shuffle.merge.percent</name>
    <value>0.66</value>
</property>

<!-- reducetask 内存，默认 1g；reducetask 堆内存大小默认和该值大小一致
      mapreduce.reduce.java.opts -->
<property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>-1</value>
    <description>The amount of memory to request from the scheduler for each reduce task. If this is not specified or is non-positive, it is inferred from mapreduce.reduce.java.opts and mapreduce.job.heap.memory-mb.ratio.
      If java-opts are also not specified, we set it to 1024.</description>
</property>

<!-- reducetask 的 CPU 核数，默认 1 个 -->
<property>
    <name>mapreduce.reduce.cpu.vcores</name>
    <value>2</value>
</property>

<!-- reducetask 失败重试次数，默认 4 次 -->
<property>
    <name>mapreduce.reduce.maxattempts</name>
    <value>4</value>
</property>

<!-- 当 MapTask 完成的比例达到该值后才会为 ReduceTask 申请资源。默认是 0.05
      -->
<property>
    <name>mapreduce.job.reduce.slowstart.completedmaps</name>
    <value>0.05</value>
</property>

<!-- 如果程序在规定的默认 10 分钟内没有读到数据，将强制超时退出 -->
<property>
    <name>mapreduce.task.timeout</name>
    <value>600000</value>
</property>
```

(2) 分发配置

```
[atguigu@hadoop102 hadoop]$ xsync mapred-site.xml
```

10.3.4 Yarn 参数调优

(1) 修改 yarn-site.xml 配置参数如下：

```
<!-- 选择调度器， 默认容量 -->
<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>

<!-- ResourceManager 处理调度器请求的线程数量，默认 50；如果提交的任务数大于 50，可以增加该值，但是不能超过 3 台 * 4 线程 = 12 线程（去除其他应用程序实际不能超过 8） -->
<property>
    <description>Number      of      threads      to      handle      scheduler
interface.</description>
    <name>yarn.resourcemanager.scheduler.client.thread-count</name>
    <value>8</value>
</property>

<!-- 是否让 yarn 自动检测硬件进行配置，默认是 false，如果该节点有很多其他应用程序，建议手动配置。如果该节点没有其他应用程序，可以采用自动 -->
<property>
    <description>Enable auto-detection of node capabilities such as
memory and CPU.
</description>
    <name>yarn.nodemanager.resource.detect-hardware-capabilities</name>
    <value>false</value>
</property>

<!-- 是否将虚拟核数当作 CPU 核数， 默认是 false，采用物理 CPU 核数 -->
<property>
    <description>Flag to determine if logical processors(such as
hyperthreads) should be counted as cores. Only applicable on Linux
when yarn.nodemanager.resource.cpu-vcores is set to -1 and
yarn.nodemanager.resource.detect-hardware-capabilities is true.
</description>
    <name>yarn.nodemanager.resource.count-logical-processors-as-
cores</name>
    <value>false</value>
</property>

<!-- 虚拟核数和物理核数乘数， 默认是 1.0 -->
<property>
    <description>Multiplier to determine how to convert phyiscal cores to
vcores. This value is used if yarn.nodemanager.resource.cpu-vcores
is set to -1(which implies auto-calculate vcores) and
yarn.nodemanager.resource.detect-hardware-capabilities is set to true.
The number of vcores will be calculated as number of CPUs * multiplier.
</description>
    <name>yarn.nodemanager.resource.pcores-vcores-multiplier</name>
    <value>1.0</value>
</property>

<!-- NodeManager 使用内存数， 默认 8G， 修改为 4G 内存 -->
<property>
    <description>Amount of physical memory, in MB, that can be allocated
for containers. If set to -1 and
yarn.nodemanager.resource.detect-hardware-capabilities is true, it is
automatically calculated(in case of Windows and Linux).
</description>
```

```
In other cases, the default is 8192MB.  
</description>  
<name>yarn.nodemanager.resource.memory-mb</name>  
<value>4096</value>  
</property>  
  
<!-- nodemanager 的 CPU 核数, 不按照硬件环境自动设定时默认是 8 个, 修改为 4 个 -->  
<property>  
    <description>Number of vcores that can be allocated  
    for containers. This is used by the RM scheduler when allocating  
    resources for containers. This is not used to limit the number of  
    CPUs used by YARN containers. If it is set to -1 and  
    yarn.nodemanager.resource.detect-hardware-capabilities is true, it is  
    automatically determined from the hardware in case of Windows and Linux.  
    In other cases, number of vcores is 8 by default.</description>  
    <name>yarn.nodemanager.resource.cpu-vcores</name>  
    <value>4</value>  
</property>  
  
<!-- 容器最小内存, 默认 1G -->  
<property>  
    <description>The minimum allocation for every container request at the  
    RM in MBs. Memory requests lower than this will be set to the value of  
    this property. Additionally, a node manager that is configured to have  
    less memory than this value will be shut down by the resource manager.  
    </description>  
    <name>yarn.scheduler.minimum-allocation-mb</name>  
    <value>1024</value>  
</property>  
  
<!-- 容器最大内存, 默认 8G, 修改为 2G -->  
<property>  
    <description>The maximum allocation for every container request at the  
    RM in MBs. Memory requests higher than this will throw an  
    InvalidResourceRequestException.  
    </description>  
    <name>yarn.scheduler.maximum-allocation-mb</name>  
    <value>2048</value>  
</property>  
  
<!-- 容器最小 CPU 核数, 默认 1 个 -->  
<property>  
    <description>The minimum allocation for every container request at the  
    RM in terms of virtual CPU cores. Requests lower than this will be set to  
    the value of this property. Additionally, a node manager that is configured  
    to have fewer virtual cores than this value will be shut down by the  
    resource manager.  
    </description>  
    <name>yarn.scheduler.minimum-allocation-vcores</name>  
    <value>1</value>  
</property>  
  
<!-- 容器最大 CPU 核数, 默认 4 个, 修改为 2 个 -->  
<property>  
    <description>The maximum allocation for every container request at the  
    RM in terms of virtual CPU cores. Requests higher than this will throw an  
    InvalidResourceRequestException.</description>  
    <name>yarn.scheduler.maximum-allocation-vcores</name>  
    <value>2</value>  
</property>  
  
<!-- 虚拟内存检查, 默认打开, 修改为关闭 -->
```

```
<property>
    <description>Whether virtual memory limits will be enforced for
    containers.</description>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>

<!-- 虚拟内存和物理内存设置比例,默认 2.1 -->
<property>
    <description>Ratio between virtual memory to physical memory when
    setting memory limits for containers. Container allocations are
    expressed in terms of physical memory, and virtual memory usage is
    allowed to exceed this allocation by this ratio.
    </description>
    <name>yarn.nodemanager.vmem-pmem-ratio</name>
    <value>2.1</value>
</property>
```

(2) 分发配置

```
[atguigu@hadoop102 hadoop]$ xsync yarn-site.xml
```

10.3.5 执行程序

(1) 重启集群

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/stop-yarn.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

(2) 执行 WordCount 程序

```
[atguigu@hadoop102      hadoop-3.1.3]$      hadoop      jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.3.jar
wordcount /input /output
```

(3) 观察 Yarn 任务执行页面

<http://hadoop103:8088/cluster/apps>

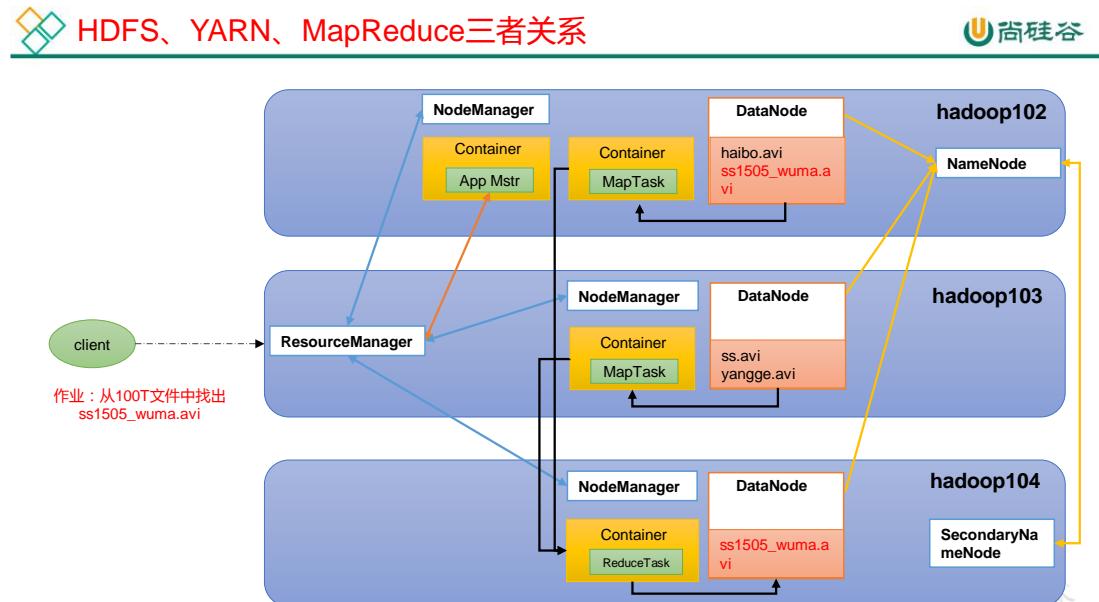
尚硅谷大数据技术之 Hadoop 源码解析

(作者: 尚硅谷大数据研发部)

版本: V3.3

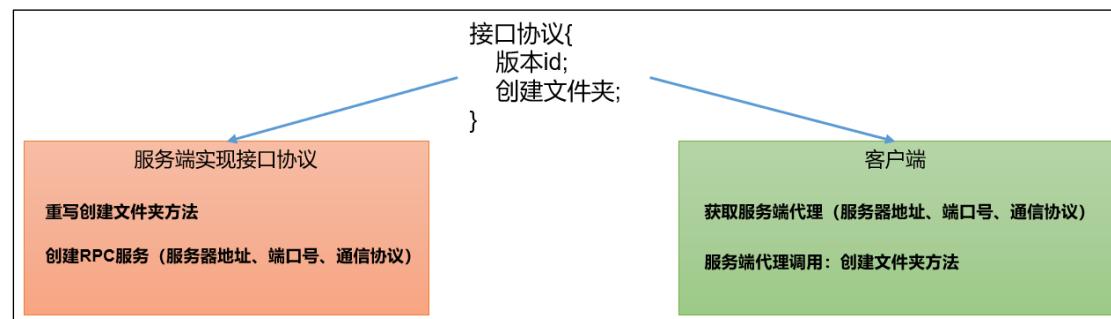
第 0 章 RPC 通信原理解析

0) 回顾



1) 需求：

模拟 RPC 的客户端、服务端、通信协议三者如何工作的



2) 代码编写：

(1) 在 HDFSClient 项目基础上创建包名 com.atguigu.rpc

(2) 创建 RPC 协议

```
package com.atguigu.rpc;

public interface RPCProtocol {
    long versionID = 666;
```

```
    void mkdirs(String path);  
}
```

(3) 创建 RPC 服务端

```
package com.atguigu.rpc;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.ipc.RPC;  
import org.apache.hadoop.ipc.Server;  
  
import java.io.IOException;  
  
public class NNServer implements RPCProtocol {  
  
    @Override  
    public void mkdirs(String path) {  
        System.out.println("服务端, 创建路径" + path);  
    }  
  
    public static void main(String[] args) throws IOException {  
  
        Server server = new RPC.Builder(new Configuration())  
            .setBindAddress("localhost")  
            .setPort(8888)  
            .setProtocol(RPCProtocol.class)  
            .setInstance(new NNServer())  
            .build();  
  
        System.out.println("服务器开始工作");  
  
        server.start();  
    }  
}
```

(4) 创建 RPC 客户端

```
package com.atguigu.rpc;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.ipc.RPC;  
  
import java.io.IOException;  
import java.net.InetSocketAddress;  
  
public class HDFSClient {  
  
    public static void main(String[] args) throws IOException {  
        RPCProtocol client = RPC.getProxy(  
            RPCProtocol.class,  
            RPCProtocol.versionID,  
            new InetSocketAddress("localhost", 8888),  
            new Configuration());  
  
        System.out.println("我是客户端");  
  
        client.mkdirs("/input");  
    }  
}
```

}

3) 测试

(1) 启动服务端

观察控制台打印：服务器开始工作

在控制台 Terminal 窗口输入， jps， 查看到 NNServer 服务

(2) 启动客户端

观察客户端控制台打印：我是客户端

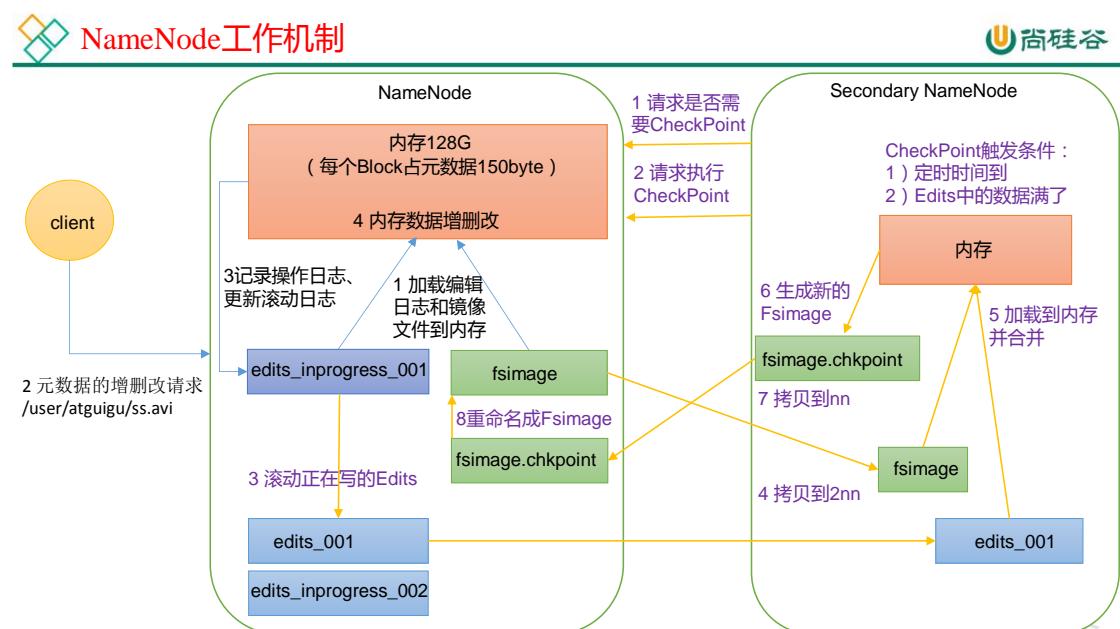
观察服务端控制台打印：服务端，创建路径/input

4) 总结

RPC 的客户端调用通信协议方法，方法的执行在服务端：

通信协议就是接口规范。

第 1 章 NameNode 启动源码解析





0) 在 pom.xml 中增加如下依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>

    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>3.1.3</version>
    </dependency>

    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs-client</artifactId>
        <version>3.1.3</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

1) **ctrl + n 全局查找 namenode**, 进入 NameNode.java

NameNode 官方说明

NameNode serves as both directory namespace manager and "inode table" for the Hadoop DFS. There is a single NameNode running in any DFS deployment. (Well, except when there is a second backup/failover NameNode, or when using federated NameNodes.) The NameNode controls two critical tables: 1) filename->blocksequence (namespace) 2) block->machinelist ("inodes") The first table is stored on disk and is very precious. The second table is rebuilt every time the NameNode comes up. 'NameNode' refers to both this class as well as the 'NameNode server'. The 'FSNamesystem' class actually performs most of the filesystem management. The majority of the 'NameNode' class itself is concerned with exposing the IPC interface and the HTTP server to the outside world, plus some configuration management. NameNode implements the ClientProtocol interface, which allows clients to ask for DFS services. ClientProtocol is not designed for direct use by authors of DFS client code. End-users

should instead use the FileSystem class. NameNode also implements the DatanodeProtocol interface, used by DataNodes that actually store DFS data blocks. These methods are invoked repeatedly and automatically by all the DataNodes in a DFS deployment. NameNode also implements the NamenodeProtocol interface, used by secondary namenodes or rebalancing processes to get partial NameNode state, for example partial blocksMap etc.

2) ctrl + f, 查找 main 方法

NameNode.java

```
public static void main(String argv[]) throws Exception {
    if (DFSUtil.parseHelpArgument(argv, NameNode.USAGE, System.out, true)) {
        System.exit(0);
    }

    try {
        StringUtils.startupShutdownMessage(NameNode.class, argv, LOG);
        // 创建 NameNode
        NameNode namenode = createNameNode(argv, null);
        if (namenode != null) {
            namenode.join();
        }
    } catch (Throwable e) {
        LOG.error("Failed to start namenode.", e);
        terminate(1, e);
    }
}
```

点击 **createNameNode**

```
public static NameNode createNameNode(String argv[], Configuration conf)
    throws IOException {
    ...
    StartupOption startOpt = parseArguments(argv);
    if (startOpt == null) {
        printUsage(System.err);
        return null;
    }
    setStartupOption(conf, startOpt);

    boolean aborted = false;
    switch (startOpt) {
        case FORMAT:
            aborted = format(conf, startOpt.getForceFormat(),
                startOpt.getInteractiveFormat());
            terminate(aborted ? 1 : 0);
            return null; // avoid javac warning
        case GENCLUSTERID:
            ...
        default:
            DefaultMetricsSystem.initialize("NameNode");
            // 创建 NameNode 对象
            return new NameNode(conf);
    }
}
```

点击 **NameNode**

```
public NameNode(Configuration conf) throws IOException {
    this(conf, NamenodeRole.NAMENODE);
```

```
}

protected NameNode(Configuration conf, NamenodeRole role)
    throws IOException {
    ...
    try {
        initializeGenericKeys(conf, nsId, namenodeId);
        initialize(getConf());
    ...
    } catch (IOException e) {
        this.stopAtException(e);
        throw e;
    } catch (HadoopIllegalArgumentException e) {
        this.stopAtException(e);
        throw e;
    }
    this.started.set(true);
}
```

点击 initialize

```
protected void initialize(Configuration conf) throws IOException {
    ...
    if (NamenodeRole.NAMENODE == role) {
        // 启动 HTTP 服务端 (9870)
        startHttpServer(conf);
    }

    // 加载镜像文件和编辑日志到内存
    loadNamesystem(conf);
    startAliasMapServerIfNecessary(conf);

    // 创建 NN 的 RPC 服务端
    rpcServer = createRpcServer(conf);

    initReconfigurableBackoffKey();

    if (clientNamenodeAddress == null) {
        // This is expected for MiniDFSCluster. Set it now using
        // the RPC server's bind address.
        clientNamenodeAddress =
            NetUtils.getHostPortString(getNameNodeAddress());
        LOG.info("Clients are to use " + clientNamenodeAddress + " to access"
            + " this namenode/service.");
    }
    if (NamenodeRole.NAMENODE == role) {
        httpServer.setNameNodeAddress(getNameNodeAddress());
        httpServer.setFSImage(getFSImage());
    }

    // NN 启动资源检查
    startCommonServices(conf);
    startMetricsLogger(conf);
}
```

1.1 启动 9870 端口服务

- 1) 点击 startHttpServer

NameNode.java

```
private void startHttpServer(final Configuration conf) throws IOException {
    httpServer = new NameNodeHttpServer(conf, this, getHttpServerBindAddress(conf));
    httpServer.start();
    httpServer.setStartupProgress(startupProgress);
}

protected InetSocketAddress getHttpServerBindAddress(Configuration conf) {
    InetSocketAddress bindAddress = getHttpServerAddress(conf);

    ...
    return bindAddress;
}

protected InetSocketAddress getHttpServerAddress(Configuration conf) {
    return getHttpAddress(conf);
}

public static InetSocketAddress getHttpAddress(Configuration conf) {
    return NetUtils.createSocketAddr(
        conf.getTrimmed(DFS_NAMENODE_HTTP_ADDRESS_KEY,
        DFS_NAMENODE_HTTP_ADDRESS_DEFAULT));
}

public static final String DFS_NAMENODE_HTTP_ADDRESS_DEFAULT = "0.0.0.0:" +
DFS_NAMENODE_HTTP_PORT_DEFAULT;

public static final int DFS_NAMENODE_HTTP_PORT_DEFAULT =
HdfsClientConfigKeys.DFS_NAMENODE_HTTP_PORT_DEFAULT;

int DFS_NAMENODE_HTTP_PORT_DEFAULT = 9870;
```

- 2) 点击 startHttpServer 方法中的 httpServer.start();

NameNodeHttpServer.java

```
void start() throws IOException {
    ...
    // Hadoop 自己封装了 HttpServer, 形成自己的 HttpServer2
    HttpServer2.Builder builder = DFSUtil.httpServerTemplateForNNAndJN(conf,
        httpAddr, httpsAddr, "hdfs",

    DFSConfigKeys.DFS_NAMENODE_KERBEROS_INTERNAL_SPNEGO_PRINCIPAL_KEY,
    DFSConfigKeys.DFS_NAMENODE_KEYTAB_FILE_KEY);
    ...

    httpServer = builder.build();

    ...
    httpServer.setAttribute(NAMENODE_ATTRIBUTE_KEY, nn);
```

```

httpServer.setAttribute(JspHelper.CURRENT_CONF, conf);
setupServlets(httpServer, conf);
httpServer.start();

...
}

```

点击 setupServlets

```

private static void setupServlets(HttpServer2 httpServer, Configuration conf) {
    httpServer.addInternalServlet("startupProgress",
        StartupProgressServlet.PATH_SPEC, StartupProgressServlet.class);
    httpServer.addInternalServlet("fsck", "/fsck", FsckServlet.class,
        true);
    httpServer.addInternalServlet("imagetransfer", ImageServlet.PATH_SPEC,
        ImageServlet.class, true);
}

```

1.2 加载镜像文件和编辑日志

1) 点击 loadNamesystem

NameNode.java

```

protected void loadNamesystem(Configuration conf) throws IOException {
    this.namesystem = FSNamesystem.loadFromDisk(conf);
}

static FSNamesystem loadFromDisk(Configuration conf) throws IOException {
    checkConfiguration(conf);

    FSImage fsImage = new FSImage(conf,
        FSNamesystem.getNamespaceDirs(conf),
        FSNamesystem.getNamespaceEditsDirs(conf));

    FSNamesystem namesystem = new FSNamesystem(conf, fsImage, false);
    StartupOption startOpt = NameNode.getStartupOption(conf);
    if (startOpt == StartupOption.RECOVER) {
        namesystem.setSafeMode(SafeModeAction.SAFEMODE_ENTER);
    }

    long loadStart = monotonicNow();
    try {
        namesystem.loadFSImage(startOpt);
    } catch (IOException ioe) {
        LOG.warn("Encountered exception loading fsimage", ioe);
        fsImage.close();
        throw ioe;
    }
    long timeTakenToLoadFSImage = monotonicNow() - loadStart;
    LOG.info("Finished loading FSImage in " + timeTakenToLoadFSImage + " msecs");
    NameNodeMetrics nnMetrics = NameNode.getNameNodeMetrics();
    if (nnMetrics != null) {
        nnMetrics.setFsImageLoadTime((int) timeTakenToLoadFSImage);
    }
    namesystem.getFSDirectory().createReservedStatuses(namesystem.getCTime());
    return namesystem;
}

```

```
}
```

1.3 初始化 NN 的 RPC 服务端

- 1) 点击 createRpcServer

```
NameNode.java
```

```
protected NameNodeRpcServer createRpcServer(Configuration conf)
    throws IOException {
    return new NameNodeRpcServer(conf, this);
}
```

```
NameNodeRpcServer.java
```

```
public NameNodeRpcServer(Configuration conf, NameNode nn)
    throws IOException {
    ...
    serviceRpcServer = new RPC.Builder(conf)
        .setProtocol(
            org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
        .setInstance(clientNNPBService)
        .setBindAddress(bindHost)
        .setPort(serviceRpcAddr.getPort())
        .setNumHandlers(serviceHandlerCount)
        .setVerbose(false)
        .setSecretManager(namesystem.getDelegationTokenSecretManager())
        .build();
    ...
}
```

1.4 NN 启动资源检查

- 1) 点击 startCommonServices

```
NameNode.java
```

```
private void startCommonServices(Configuration conf) throws IOException {
    namesystem.startCommonServices(conf, haContext);

    registerNNSMXBean();
    if (NamenodeRole.NAMENODE != role) {
        startHttpServer(conf);
        httpServer.setNameNodeAddress(getNameNodeAddress());
        httpServer.setFSImage(getFSImage());
    }
    rpcServer.start();
    try {
        plugins = conf.getInstances(DFS_NAMENODE_PLUGINS_KEY,
            ServicePlugin.class);
    } catch (RuntimeException e) {
        String pluginsValue = conf.get(DFS_NAMENODE_PLUGINS_KEY);
        LOG.error("Unable to load NameNode plugins. Specified list of plugins: " +
            pluginsValue, e);
        throw e;
    }
    ...
}
```

2) 点击 startCommonServices

FSNamesystem.java

```

void startCommonServices(Configuration conf, HAContext haContext) throws IOException {
    this.registerMBean(); // register the MBean for the FSNamesystemState
    writeLock();
    this.haContext = haContext;
    try {
        nnResourceChecker = new NameNodeResourceChecker(conf);
        // 检查是否有足够的磁盘存储元数据(fsimage(默认 100m) editLog(默认 100m))
        checkAvailableResources();

        assert !blockManager.isPopulatingReplQueues();
        StartupProgress prog = NameNode.getStartupProgress();
        prog.beginPhase(Phase.SAFEMODE);
        long completeBlocksTotal = getCompleteBlocksTotal();

        // 安全模式
        prog.setTotal(Phase.SAFEMODE, STEP_AWAITING_REPORTED_BLOCKS,
                     completeBlocksTotal);

        // 启动块服务
        blockManager.activate(conf, completeBlocksTotal);
    } finally {
        writeUnlock("startCommonServices");
    }

    registerMXBean();
    DefaultMetricsSystem.instance().register(this);
    if (inodeAttributeProvider != null) {
        inodeAttributeProvider.start();
        dir.setINodeAttributeProvider(inodeAttributeProvider);
    }
    snapshotManager.registerMXBean();
    InetSocketAddress serviceAddress = NameNode.getServiceAddress(conf, true);
    this.nameNodeHostName = (serviceAddress != null) ?
        serviceAddress.getHostName() : "";
}

```

点击 NameNodeResourceChecker

NameNodeResourceChecker.java

```

public NameNodeResourceChecker(Configuration conf) throws IOException {
    this.conf = conf;
    volumes = new HashMap<String, CheckedVolume>();

    // dfs.namenode.resource.du.reserved 默认值 1024 * 1024 * 100 => 100m
    duReserved = conf.getLong(DFSConfigKeys.DFS_NAMENODE_DU_RESERVED_KEY,
                             DFSConfigKeys.DFS_NAMENODE_DU_RESERVED_DEFAULT);

    Collection<URI> extraCheckedVolumes = Util.stringCollectionAsURIs(conf
        .getTrimmedStringCollection(DFSConfigKeys.DFS_NAMENODE_CHECKED_VOLUMES_KEY));

    Collection<URI> localEditDirs = Collections2.filter(

```

```

FSNamesystem.getNamespaceEditsDirs(conf),
new Predicate<URI>() {
    @Override
    public boolean apply(URI input) {
        if (input.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
            return true;
        }
        return false;
    }
};

// 对所有路径进行资源检查
for (URI editsDirToCheck : localEditDirs) {
    addDirToCheck(editsDirToCheck,
        FSNamesystem.getRequiredNamespaceEditsDirs(conf).contains(
            editsDirToCheck));
}

// All extra checked volumes are marked "required"
for (URI extraDirToCheck : extraCheckedVolumes) {
    addDirToCheck(extraDirToCheck, true);
}

minimumRedundantVolumes = conf.getInt(
    DFSConfigKeys.DFS_NAMENODE_CHECKED_VOLUMES_MINIMUM_KEY,
    DFSConfigKeys.DFS_NAMENODE_CHECKED_VOLUMES_MINIMUM_DEFAULT);
}

```

点击 checkAvailableResources

FNNamesystem.java

```

void checkAvailableResources() {
    long resourceCheckTime = monotonicNow();
    Preconditions.checkNotNull(nnResourceChecker != null,
        "nnResourceChecker not initialized");

    // 判断资源是否足够, 不够返回 false
    hasResourcesAvailable = nnResourceChecker.hasAvailableDiskSpace();

    resourceCheckTime = monotonicNow() - resourceCheckTime;
    NameNode.getNameNodeMetrics().addResourceCheckTime(resourceCheckTime);
}

```

NameNodeResourceChecker.java

```

public boolean hasAvailableDiskSpace() {
    return NameNodeResourcePolicy.areResourcesAvailable(volumes.values(),
        minimumRedundantVolumes);
}

```

NameNodeResourcePolicy.java

```

static boolean areResourcesAvailable(
    Collection<? extends CheckableNameNodeResource> resources,
    int minimumRedundantResources) {

    // TODO: workaround:
    // - during startup, if there are no edits dirs on disk, then there is
}

```

```
// a call to areResourcesAvailable() with no dirs at all, which was
// previously causing the NN to enter safemode
if (resources.isEmpty()) {
    return true;
}

int requiredResourceCount = 0;
int redundantResourceCount = 0;
int disabledRedundantResourceCount = 0;

// 判断资源是否充足
for (CheckableNameNodeResource resource : resources) {
    if (!resource.isRequired()) {
        redundantResourceCount++;
        if (!resource.isResourceAvailable()) {
            disabledRedundantResourceCount++;
        }
    } else {
        requiredResourceCount++;
        if (!resource.isResourceAvailable()) {
            // Short circuit - a required resource is not available. 不充足返回 false
            return false;
        }
    }
}

if (redundantResourceCount == 0) {
    // If there are no redundant resources, return true if there are any
    // required resources available.
    return requiredResourceCount > 0;
} else {
    return redundantResourceCount - disabledRedundantResourceCount >=
        minimumRedundantResources;
}

interface CheckableNameNodeResource {

    public boolean isResourceAvailable();

    public booleanisRequired();
}
```

ctrl + h, 查找实现类 CheckedVolume

NameNodeResourceChecker.java

```
public boolean isResourceAvailable() {

    // 获取当前目录的空间大小
    long availableSpace = df.getAvailable();

    if (LOG.isDebugEnabled()) {
        LOG.debug("Space available on volume " + volume + " is "
            + availableSpace);
    }
}
```

```
// 如果当前空间大小，小于 100m，返回 false
if (availableSpace < duReserved) {
    LOG.warn("Space available on volume '" + volume + "' is "
        + availableSpace +
        ", which is below the configured reserved amount " + duReserved);
    return false;
} else {
    return true;
}
```

1.5 NN 对心跳超时判断

Ctrl + n 搜索 namenode, ctrl + f 搜索 startCommonServices

点击 namesystem.startCommonServices(conf, haContext);

点击 blockManager.activate(conf, completeBlocksTotal);

点击 datanodeManager.activate(conf);

DatanodeManager.java

```
void activate(final Configuration conf) {
    datanodeAdminManager.activate(conf);
    heartbeatManager.activate();
}
```

DatanodeManager.java

```
void activate() {
    // 启动的线程，搜索 run 方法
    heartbeatThread.start();
}
```

public void run() {

```
    while(namesystem.isRunning()) {
        restartHeartbeatStopWatch();
        try {
            final long now = Time.monotonicNow();
            if (lastHeartbeatCheck + heartbeatRecheckInterval < now) {
                // 心跳检查
                heartbeatCheck();
                lastHeartbeatCheck = now;
            }
            if (blockManager.shouldUpdateBlockKey(now - lastBlockKeyUpdate)) {
                synchronized(HeartbeatManager.this) {
                    for(DatanodeDescriptor d : datanodes) {
                        d.setNeedKeyUpdate(true);
                    }
                }
                lastBlockKeyUpdate = now;
            }
        } catch (Exception e) {
            LOG.error("Exception while checking heartbeat", e);
        }
        try {
            Thread.sleep(5000); // 5 seconds
        } catch (InterruptedException ignored) {
```

```
        }
        // avoid declaring nodes dead for another cycle if a GC pause lasts
        // longer than the node recheck interval
        if (shouldAbortHeartbeatCheck(-5000)) {
            LOG.warn("Skipping next heartbeat scan due to excessive pause");
            lastHeartbeatCheck = Time.monotonicNow();
        }
    }

void heartbeatCheck() {
    final DatanodeManager dm = blockManager.getDatanodeManager();

    boolean allAlive = false;
    while (!allAlive) {
        // locate the first dead node.
        DatanodeDescriptor dead = null;

        // locate the first failed storage that isn't on a dead node.
        DatanodeStorageInfo failedStorage = null;

        // check the number of stale nodes
        int numOfStaleNodes = 0;
        int numOfStaleStorages = 0;
        synchronized(this) {
            for (DatanodeDescriptor d : datanodes) {
                // check if an excessive GC pause has occurred
                if (shouldAbortHeartbeatCheck(0)) {
                    return;
                }
                // 判断 DN 节点是否挂断
                if (dead == null && dm.isDatanodeDead(d)) {
                    stats.incrExpiredHeartbeats();
                    dead = d;
                }
                if (d.isStale(dm.getStaleInterval())) {
                    numOfStaleNodes++;
                }
                DatanodeStorageInfo[] storageInfos = d.getStorageInfos();
                for(DatanodeStorageInfo storageInfo : storageInfos) {
                    if (storageInfo.areBlockContentsStale()) {
                        numOfStaleStorages++;
                    }

                    if (failedStorage == null &&
                        storageInfo.areBlocksOnFailedStorage() &&
                        d != dead) {
                        failedStorage = storageInfo;
                    }
                }
            }
        }

        // Set the number of stale nodes in the DatanodeManager
        dm.setNumStaleNodes(numOfStaleNodes);
        dm.setNumStaleStorages(numOfStaleStorages);
    }
    ...
}
```

```
        }

    boolean isDatanodeDead(DatanodeDescriptor node) {
        return (node.getLastUpdateMonotonic() <
                (monotonicNow() - heartbeatExpireInterval));
    }

    private long heartbeatExpireInterval;
    // 10 分钟 + 30 秒
    this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval + 10 * 1000 *
    heartbeatIntervalSeconds;

    private volatile int heartbeatRecheckInterval;
    heartbeatRecheckInterval = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY,
        DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_DEFAULT);
    // 5 minutes

    private volatile long heartbeatIntervalSeconds;
    heartbeatIntervalSeconds = conf.getTimeDuration(
        DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
        DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT,
        TimeUnit.SECONDS);
    public static final long DFS_HEARTBEAT_INTERVAL_DEFAULT = 3;
```

1.6 安全模式

FSNamesystem.java

```
void startCommonServices(Configuration conf, HAContext haContext) throws IOException {
    this.registerMBean(); // register the MBean for the FSNamesystemState
    writeLock();
    this.haContext = haContext;
    try {
        nnResourceChecker = new NameNodeResourceChecker(conf);
        // 检查是否有足够的磁盘存储元数据(fsimage(默认 100m) editLog(默认 100m))
        checkAvailableResources();

        assert !blockManager.isPopulatingReplQueues();
        StartupProgress prog = NameNode.getStartupProgress();

        // 开始进入安全模式
        prog.beginPhase(Phase.SAFEMODE);

        // 获取所有可以正常使用的 block
        long completeBlocksTotal = getCompleteBlocksTotal();

        prog.setTotal(Phase.SAFEMODE, STEP_AWAITING_REPORTED_BLOCKS,
                    completeBlocksTotal);

        // 启动块服务
        blockManager.activate(conf, completeBlocksTotal);
    } finally {
        writeUnlock("startCommonServices");
```

```
}

registerMXBean();
DefaultMetricsSystem.instance().register(this);
if (inodeAttributeProvider != null) {
    inodeAttributeProvider.start();
    dir.setINodeAttributeProvider(inodeAttributeProvider);
}
snapshotManager.registerMXBean();
InetSocketAddress serviceAddress = NameNode.getServiceAddress(conf, true);
this.nameNodeHostName = (serviceAddress != null) ?
    serviceAddress.getHostName() : "";
}
```

点击 getCompleteBlocksTotal

```
public long getCompleteBlocksTotal() {
    // Calculate number of blocks under construction
    long numUCBlocks = 0;
    readLock();
    try {
        // 获取正在构建的 block
        numUCBlocks = leaseManager.getNumUnderConstructionBlocks();
        // 获取所有的块 - 正在构建的 block = 可以正常使用的 block
        return getBlocksTotal() - numUCBlocks;
    } finally {
        readUnlock("getCompleteBlocksTotal");
    }
}
```

点击 activate

```
public void activate(Configuration conf, long blockTotal) {
    pendingReconstruction.start();
    datanodeManager.activate(conf);

    this.redundancyThread.setName("RedundancyMonitor");
    this.redundancyThread.start();

    storageInfoDefragmenterThread.setName("StorageInfoMonitor");
    storageInfoDefragmenterThread.start();
    this.blockReportThread.start();

    mxBeanName = MBeans.register("NameNode", "BlockStats", this);

    bmSafeMode.activate(blockTotal);
}
```

点击 activate

```
void activate(long total) {
    assert namesystem.hasWriteLock();
    assert status == BMSSafeModeStatus.OFF;

    startTime = monotonicNow();

    // 计算是否满足块个数的阈值
    setBlockTotal(total);
```

```
// 判断 DataNode 节点和块信息是否达到退出安全模式标准
if (areThresholdsMet()) {
    boolean exitResult = leaveSafeMode(false);
    Preconditions.checkState(exitResult, "Failed to leave safe mode.");
} else {
    // enter safe mode
    status = BMSSafeModeStatus.PENDING_THRESHOLD;

    initializeReplQueuesIfNecessary();

    reportStatus("STATE* Safe mode ON.", true);
    lastStatusReport = monotonicNow();
}
}
```

点击 setBlockTotal

```
void setBlockTotal(long total) {
    assert namesystem.hasWriteLock();
    synchronized (this) {
        this.blockTotal = total;
        // 计算阈值：例如：1000 个正常的块 * 0.999 = 999
        this.blockThreshold = (long) (total * threshold);
    }

    this.blockReplQueueThreshold = (long) (total * replQueueThreshold);
}

this.threshold = conf.getFloat(DFS_NAMENODE_SAFEMODE_THRESHOLD_PCT_KEY,
    DFS_NAMENODE_SAFEMODE_THRESHOLD_PCT_DEFAULT);

public static final float DFS_NAMENODE_SAFEMODE_THRESHOLD_PCT_DEFAULT = 0.999f;
```

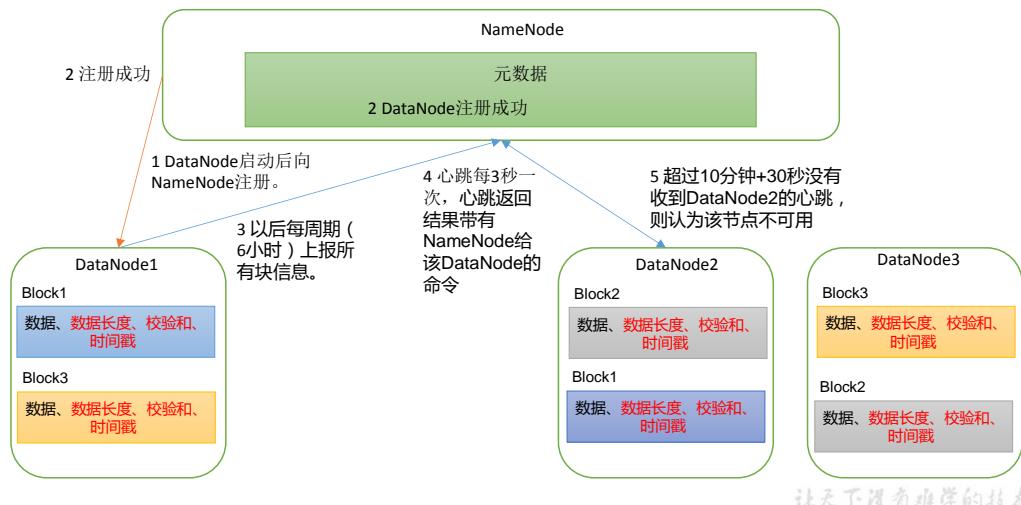
点击 areThresholdsMet

```
private boolean areThresholdsMet() {
    assert namesystem.hasWriteLock();
    // Calculating the number of live datanodes is time-consuming
    // in large clusters. Skip it when datanodeThreshold is zero.
    int datanodeNum = 0;

    if (datanodeThreshold > 0) {
        datanodeNum = blockManager.getDatanodeManager().getNumLiveDataNodes();
    }
    synchronized (this) {
        // 已经正常注册的块数 » = 块的最小阈值 » = 最小可用 DataNode
        return blockSafe >= blockThreshold && datanodeNum >= datanodeThreshold;
    }
}
```

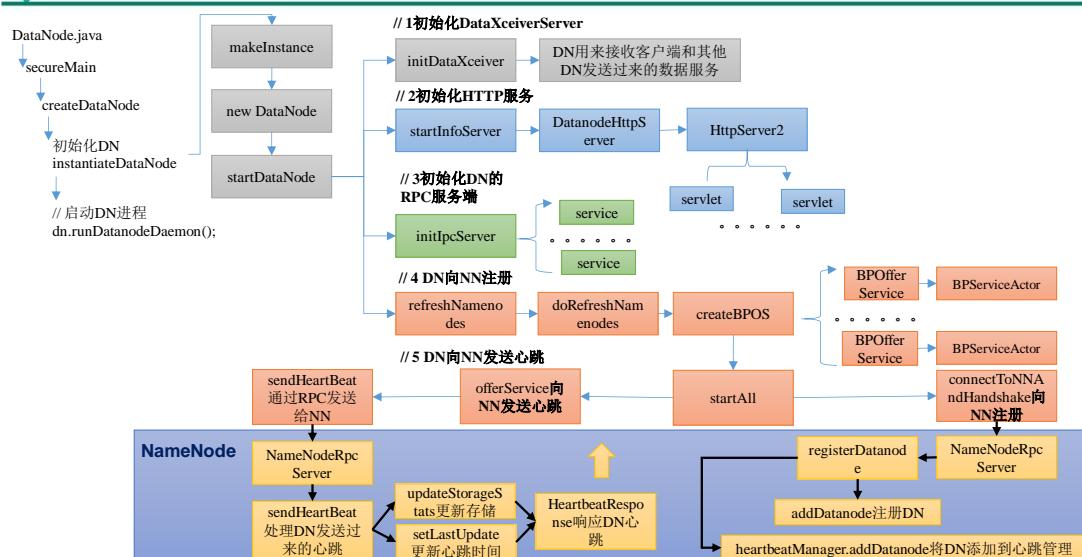
第 2 章 DataNode 启动源码解析

DataNode 工作机制



让天下没有难学的技术

DataNode 启动源码解析



0) 在 pom.xml 中增加如下依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>

    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>3.1.3</version>
    </dependency>
```

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs-client</artifactId>
    <version>3.1.3</version>
    <scope>provided</scope>
</dependency>
</dependencies>
```

- 1) **ctrl + n 全局查找 datanode，进入 DataNode.java**

DataNode 官方说明

DataNode is a class (and program) that stores a set of blocks for a DFS deployment. A single deployment can have one or many DataNodes. Each DataNode communicates regularly with a single NameNode. It also communicates with client code and other DataNodes from time to time. DataNodes store a series of named blocks. The DataNode allows client code to read these blocks, or to write new block data. The DataNode may also, in response to instructions from its NameNode, delete blocks or copy blocks to/from other DataNodes. The DataNode maintains just one critical table: block-> stream of bytes (of BLOCK_SIZE or less) This info is stored on a local disk. The DataNode reports the table's contents to the NameNode upon startup and every so often afterwards. DataNodes spend their lives in an endless loop of asking the NameNode for something to do. A NameNode cannot connect to a DataNode directly; a NameNode simply returns values from functions invoked by a DataNode. DataNodes maintain an open server socket so that client code or other DataNodes can read/write data. The host/port for this server is reported to the NameNode, which then sends that information to clients or other DataNodes that might be interested.

- 2) **ctrl + f，查找 main 方法**

DataNode.java

```
public static void main(String args[]) {
    if (DFSUtil.parseHelpArgument(args, DataNode.USAGE, System.out, true)) {
        System.exit(0);
    }

    secureMain(args, null);
}

public static void secureMain(String args[], SecureResources resources) {
    int errorCode = 0;
    try {
        StringUtils.startupShutdownMessage(DataNode.class, args, LOG);

        DataNode datanode = createDataNode(args, null, resources);

        ...
    } catch (Throwable e) {
        LOG.error("Exception in secureMain", e);
        terminate(1, e);
    } finally {
        LOG.warn("Exiting Datanode");
        terminate(errorCode);
    }
}

public static DataNode createDataNode(String args[], Configuration conf,
    SecureResources resources) throws IOException {
    // 初始化 DN
```

```
    DataNode dn = instantiateDataNode(args, conf, resources);

    if (dn != null) {
        // 启动 DN 进程
        dn.runDatanodeDaemon();
    }
    return dn;
}

public static DataNode instantiateDataNode(String args [], Configuration conf,
    SecureResources resources) throws IOException {
    ...
    ...
    return makeInstance(dataLocations, conf, resources);
}

static DataNode makeInstance(Collection<StorageLocation> dataDirs,
    Configuration conf, SecureResources resources) throws IOException {
    ...
    ...
    return new DataNode(conf, locations, storageLocationChecker, resources);
}

DataNode(final Configuration conf,
    final List<StorageLocation> dataDirs,
    final StorageLocationChecker storageLocationChecker,
    final SecureResources resources) throws IOException {
    super(conf);
    ...
    ...

    try {
        hostName = getHostName(conf);
        LOG.info("Configured hostname is {}", hostName);
        // 启动 DN
        startDataNode(dataDirs, resources);
    } catch (IOException ie) {
        shutdown();
        throw ie;
    }
    ...
}
}

void startDataNode(List<StorageLocation> dataDirectories,
    SecureResources resources
    ) throws IOException {
    ...
    ...
    // 创建数据存储对象
    storage = new DataStorage();

    // global DN settings
    registerMXBean();
    // 初始化 DataXceiver
    initDataXceiver();

    // 启动 HttpServer
    startInfoServer();
}
```

```

pauseMonitor = new JvmPauseMonitor();
pauseMonitor.init(getConf());
pauseMonitor.start();

// BlockPoolTokenSecretManager is required to create ipc server.
this.blockPoolTokenSecretManager = new BlockPoolTokenSecretManager();

// Login is done by now. Set the DN user name.
dnUserName = UserGroupInformation.getCurrentUser().getUserName();
LOG.info("dnUserName = {}", dnUserName);
LOG.info("supergroup = {}", supergroup);

// 初始化 RPC 服务
initIpcServer();

metrics = DataNodeMetrics.create(getConf(), getDisplayName());
peerMetrics = dnConf.peerStatsEnabled ?
    DataNodePeerMetrics.create(getDisplayName(), getConf()) : null;
metrics.setJvmMetrics().setPauseMonitor(pauseMonitor);

ecWorker = new ErasureCodingWorker(getConf(), this);
blockRecoveryWorker = new BlockRecoveryWorker(this);

// 创建 BlockPoolManager
blockPoolManager = new BlockPoolManager(this);
// 心跳管理
blockPoolManager.refreshNamenodes(getConf());

// Create the ReadaheadPool from the DataNode context so we can
// exit without having to explicitly shutdown its thread pool.
readaheadPool = ReadaheadPool.getInstance();
saslClient = new SaslDataTransferClient(dnConf.getConf(),
    dnConf.saslPropsResolver, dnConf.trustedChannelResolver);
saslServer = new SaslDataTransferServer(dnConf, blockPoolTokenSecretManager);
startMetricsLogger();

if (dnConf.diskStatsEnabled) {
    diskMetrics = new DataNodeDiskMetrics(this,
        dnConf.outliersReportIntervalMs);
}
}
}

```

2.1 初始 DataXceiverServer

点击 initDataXceiver

```

private void initDataXceiver() throws IOException {
    // dataXceiverServer 是一个服务，DN 用来接收客户端和其他 DN 发送过来的数据服务
    this.dataXceiverServer = new Daemon(threadGroup, xserver);
    this.threadGroup.setDaemon(true); // auto destroy when empty

    ...
}

```

2.2 初始 HTTP 服务

点击 startInfoServer();

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

DataNode.java

```
private void startInfoServer()
    throws IOException {
    // SecureDataNodeStarter will bind the privileged port to the channel if
    // the DN is started by JSVC, pass it along.
    ServerSocketChannel httpServerChannel = secureResources != null ?
        secureResources.getHttpServerChannel() : null;

    httpServer = new DatanodeHttpServer(getConf(), this, httpServerChannel);
    httpServer.start();
    if (httpServer.getHttpAddress() != null) {
        infoPort = httpServer.getHttpAddress().getPort();
    }
    if (httpServer.getHttpsAddress() != null) {
        infoSecurePort = httpServer.getHttpsAddress().getPort();
    }
}
```

DatanodeHttpServer.java

```
public DatanodeHttpServer(final Configuration conf,
    final DataNode datanode,
    final ServerSocketChannel externalHttpChannel)
throws IOException {

    ...
    HttpServer2.Builder builder = new HttpServer2.Builder()
        .setName("datanode")
        .setConf(confForInfoServer)
        .setACL(new AccessControlList(conf.get(DFS_ADMIN, " ")))
        .hostName(getHostnameForSpnegoPrincipal(confForInfoServer))
        .addEndpoint(URI.create("http://localhost:" + proxyPort))
        .setFindPort(true);
    ...
}
```

2.3 初始化 DN 的 RPC 服务端

点击 initIpcServer

DataNode.java

```
private void initIpcServer() throws IOException {
    InetSocketAddress ipcAddr = NetUtils.createSocketAddr(
        getConf().getTrimmed(DFS_DATANODE_IPC_ADDRESS_KEY));

    ...
    ipcServer = new RPC.Builder(getConf())
        .setProtocol(ClientDatanodeProtocolPB.class)
        .setInstance(service)
        .setBindAddress(ipcAddr.getHostName())
        .setPort(ipcAddr.getPort())
        .setNumHandlers(
            getConf().getInt(DFS_DATANODE_HANDLER_COUNT_KEY,
                DFS_DATANODE_HANDLER_COUNT_DEFAULT)).setVerbose(false)
        .setSecretManager(blockPoolTokenSecretManager).build();
    ...
}
```

```
}
```

2.4 DN 向 NN 注册

点击 refreshNamenodes

```
BlockPoolManager.java
void refreshNamenodes(Configuration conf)
    throws IOException {
    ...
    synchronized (refreshNamenodesLock) {
        doRefreshNamenodes(newAddressMap, newLifelineAddressMap);
    }
}

private void doRefreshNamenodes(
    Map<String, Map<String, InetSocketAddress>> addrMap,
    Map<String, Map<String, InetSocketAddress>> lifelineAddrMap)
    throws IOException {
    ...
    synchronized (this) {
        ...
        // Step 3. Start new nameservices
        if (!toAdd.isEmpty()) {
            for (String nsToAdd : toAdd) {
                ...
                BPOfferService bpos = createBPOS(nsToAdd, addrs, lifelineAddrs);
                bpByNameserviceId.put(nsToAdd, bpos);
                offerServices.add(bpos);
            }
            startAll();
        }
        ...
    }
}

protected BPOfferService createBPOS(
    final String nameserviceId,
    List<InetSocketAddress> nnAddrs,
    List<InetSocketAddress> lifelineNnAddrs) {
    // 根据 NameNode 个数创建对应的服务
    return new BPOfferService(nameserviceId, nnAddrs, lifelineNnAddrs, dn);
}
```

点击 startAll()

```
synchronized void startAll() throws IOException {
    try {
        UserGroupInformation.getLoginUser().doAs(
            new PrivilegedExceptionAction<Object>() {
                @Override
                public Object run() throws Exception {
```

```

        for (BPOfferService bpos : offerServices) {
            // 启动服务
            bpos.start();
        }
        return null;
    }
});
} catch (InterruptedException ex) {
    ...
}
}

```

点击 start()

BPOfferService.java

```

void start() {
    for (BPServiceActor actor : bpServices) {
        actor.start();
    }
}

```

点击 start()

BPServiceActor.java

```

void start() {
    ...
    bpThread = new Thread(this);
    bpThread.setDaemon(true); // needed for JUnit testing
    // 表示开启一个线程，所有查找该线程的 run 方法
    bpThread.start();

    if (lifelineSender != null) {
        lifelineSender.start();
    }
}

```

ctrl + f 搜索 run 方法

```

public void run() {
    LOG.info(this + " starting to offer service");

    try {
        while (true) {
            // init stuff
            try {
                // setup storage
                // 向 NN 注册
                connectToNNAndHandshake();
                break;
            } catch (IOException ioe) {
                // Initial handshake, storage recovery or registration failed
                runningState = RunningState.INIT_FAILED;
                if (shouldRetryInit()) {
                    // Retry until all namenode's of BPOS failed initialization
                    LOG.error("Initialization failed for " + this +
                            + ioe.getLocalizedMessage());
                    // 注册失败，5s 后重试
                }
            }
        }
    }
}

```

```

        sleepAndLogInterruptions(5000, "initializing");
    } else {
        runningState = RunningState.FAILED;
        LOG.error("Initialization failed for " + this + ". Exiting. ", ioe);
        return;
    }
}
...
while (shouldRun()) {
    try {
        // 发送心跳
        offerService();
    } catch (Exception ex) {
        ...
    }
}

private void connectToNNAndHandshake() throws IOException {
    // get NN proxy 获取 NN 的 RPC 客户端对象
    bpNamenode = dn.connectToNN(nnAddr);

    // First phase of the handshake with NN - get the namespace
    // info.
    NamespaceInfo nsInfo = retrieveNamespaceInfo();

    // Verify that this matches the other NN in this HA pair.
    // This also initializes our block pool in the DN if we are
    // the first NN connection for this BP.
    bpos.verifyAndSetNamespaceInfo(this, nsInfo);

    /* set thread name again to include NamespaceInfo when it's available. */
    this.bpThread.setName(formatThreadName("heartbeating", nnAddr));

    // 注册
    register(nsInfo);
}

DatanodeProtocolClientSideTranslatorPB connectToNN(
    InetSocketAddress nnAddr) throws IOException {
    return new DatanodeProtocolClientSideTranslatorPB(nnAddr, getConf());
}

```

DatanodeProtocolClientSideTranslatorPB.java

```

public DatanodeProtocolClientSideTranslatorPB(InetSocketAddress nameNodeAddr,
    Configuration conf) throws IOException {
    RPC.setProtocolEngine(conf, DatanodeProtocolPB.class,
        ProtobufRpcEngine.class);
    UserGroupInformation ugi = UserGroupInformation.getCurrentUser();
    rpcProxy = createNamenode(nameNodeAddr, conf, ugi);
}

private static DatanodeProtocolPB createNamenode(
    InetSocketAddress nameNodeAddr, Configuration conf,
    UserGroupInformation ugi) throws IOException {

```

```

    return RPC.getProxy(DatanodeProtocolPB.class,
        RPC.getProtocolVersion(DatanodeProtocolPB.class), nameNodeAddr, ugi,
        conf, NetUtils.getSocketFactory(conf, DatanodeProtocolPB.class));
}

```

点击 register

BPServiceActor.java

```

void register(NamespaceInfo nsInfo) throws IOException {
    // 创建注册信息
    DatanodeRegistration newBpRegistration = bpos.createRegistration();

    LOG.info(this + " beginning handshake with NN");

    while (shouldRun()) {
        try {
            // Use returned registration from namenode with updated fields
            // 把注册信息发送给 NN (DN 调用接口方法, 执行在 NN)
            newBpRegistration = bpNamenode.registerDatanode(newBpRegistration);
            newBpRegistration.setNamespaceInfo(nsInfo);
            bpRegistration = newBpRegistration;
            break;
        } catch(EOFException e) { // namenode might have just restarted
            LOG.info("Problem connecting to server: " + nnAddr + ":" +
                e.getLocalizedMessage());
            sleepAndLogInterrups(1000, "connecting to server");
        } catch(SocketTimeoutException e) { // namenode is busy
            LOG.info("Problem connecting to server: " + nnAddr);
            sleepAndLogInterrups(1000, "connecting to server");
        }
    }
    .....
}

```

ctrl + n 搜索 NameNodeRpcServer

NameNodeRpcServer.java

ctrl + f 在 NameNodeRpcServer.java 中搜索 registerDatanode

```

public DatanodeRegistration registerDatanode(DatanodeRegistration nodeReg)
    throws IOException {
    checkNNStartup();
    verifySoftwareVersion(nodeReg);
    // 注册 DN
    namesystem.registerDatanode(nodeReg);

    return nodeReg;
}

```

FSNamesystem.java

```

void registerDatanode(DatanodeRegistration nodeReg) throws IOException {
    writeLock();
    try {
        blockManager.registerDatanode(nodeReg);
    } finally {
        writeUnlock("registerDatanode");
    }
}

```

```
}

BlockManager.java
public void registerDatanode(DatanodeRegistration nodeReg)
    throws IOException {
    assert namesystem.hasWriteLock();
    datanodeManager.registerDatanode(nodeReg);
    bmSafeMode.checkSafeMode();
}

public void registerDatanode(DatanodeRegistration nodeReg)
    throws DisallowedDatanodeException, UnresolvedTopologyException {
    ...
    // register new datanode 注册 DN
    addDatanode(nodeDescr);
    blockManager.getBlockReportLeaseManager().register(nodeDescr);
    // also treat the registration message as a heartbeat
    // no need to update its timestamp
    // because its is done when the descriptor is created
    // 将 DN 添加到心跳管理
    heartbeatManager.addDatanode(nodeDescr);
    heartbeatManager.updateDnStat(nodeDescr);
    incrementVersionCount(nodeReg.getSoftwareVersion());
    startAdminOperationIfNecessary(nodeDescr);
    success = true;
    ...
}

void addDatanode(final DatanodeDescriptor node) {
    // To keep host2DatanodeMap consistent with datanodeMap,
    // remove from host2DatanodeMap the datanodeDescriptor removed
    // from datanodeMap before adding node to host2DatanodeMap.
    synchronized(this) {
        host2DatanodeMap.remove(datanodeMap.put(node.getDatanodeUuid(), node));
    }

    networktopology.add(node); // may throw InvalidTopologyException
    host2DatanodeMap.add(node);
    checkIfClusterIsNowMultiRack(node);
    resolveUpgradeDomain(node);

    ...
}
```

2.5 向 NN 发送心跳

点击 BPSERVICEActor.java 中的 run 方法中的 offerService 方法

```
BPSERVICEActor.java
private void offerService() throws Exception {
    while (shouldRun()) {
        ...
        HeartbeatResponse resp = null;
        if (sendHeartbeat) {
            boolean requestBlockReportLease = (fullBlockReportLeaseId == 0) &&
```

```
        scheduler.isBlockReportDue(startTime);
        if (!dn.areHeartbeatsDisabledForTests()) {
            // 发送心跳信息
            resp = sendHeartBeat(requestBlockReportLease);
            assert resp != null;
            if (resp.getFullBlockReportLeaseId() != 0) {
                if (fullBlockReportLeaseId != 0) {
                    .....
                }
                fullBlockReportLeaseId = resp.getFullBlockReportLeaseId();
            }
            .....
        }
    }

HeartbeatResponse sendHeartBeat(boolean requestBlockReportLease)
    throws IOException {
    .....
    // 通过 NN 的 RPC 客户端发送给 NN
    HeartbeatResponse response = bpNamenode.sendHeartbeat(bpRegistration,
        reports,
        dn.getFSDataset().getCacheCapacity(),
        dn.getFSDataset().getCacheUsed(),
        dn.getXmitsInProgress(),
        dn.getXceiverCount(),
        numFailedVolumes,
        volumeFailureSummary,
        requestBlockReportLease,
        slowPeers,
        slowDisks);
    .....
}
```

ctrl + n 搜索 NameNodeRpcServer

NameNodeRpcServer.java

ctrl + f 在 NameNodeRpcServer.java 中搜索 sendHeartbeat

```
public HeartbeatResponse sendHeartbeat(DatanodeRegistration nodeReg,
    StorageReport[] report, long dnCacheCapacity, long dnCacheUsed,
    int xmitsInProgress, int xceiverCount,
    int failedVolumes, VolumeFailureSummary volumeFailureSummary,
    boolean requestFullBlockReportLease,
    @Nonnull SlowPeerReports slowPeers,
    @Nonnull SlowDiskReports slowDisks) throws IOException {

    checkNNStartup();
    verifyRequest(nodeReg);

    // 处理 DN 发送的心跳
    return namesystem.handleHeartbeat(nodeReg, report,
        dnCacheCapacity, dnCacheUsed, xceiverCount, xmitsInProgress,
        failedVolumes, volumeFailureSummary, requestFullBlockReportLease,
```

```

        slowPeers, slowDisks);
    }

HeartbeatResponse handleHeartbeat(DatanodeRegistration nodeReg,
    StorageReport[] reports, long cacheCapacity, long cacheUsed,
    int xceiverCount, int xmitsInProgress, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary,
    boolean requestFullBlockReportLease,
    @Nonnull SlowPeerReports slowPeers,
    @Nonnull SlowDiskReports slowDisks) throws IOException {
    readLock();
    try {
        //get datanode commands
        final int maxTransfer = blockManager.getMaxReplicationStreams()
            - xmitsInProgress;
        // 处理 DN 发送过来的心跳
        DatanodeCommand[] cmds = blockManager.getDatanodeManager().handleHeartbeat(
            nodeReg, reports, getBlockPoolId(), cacheCapacity, cacheUsed,
            xceiverCount, maxTransfer, failedVolumes, volumeFailureSummary,
            slowPeers, slowDisks);

        long blockReportLeaseId = 0;
        if (requestFullBlockReportLease) {
            blockReportLeaseId = blockManager.requestBlockReportLeaseId(nodeReg);
        }
        //create ha status
        final NNHAStatusHeartbeat haState = new NNHAStatusHeartbeat(
            haContext.getState().getServiceState(),
            getFSImage().getCorrectLastAppliedOrWrittenTxId());

        // 响应 DN 的心跳
        return new HeartbeatResponse(cmds, haState, rollingUpgradeInfo,
            blockReportLeaseId);
    } finally {
        readUnlock("handleHeartbeat");
    }
}

```

点击 handleHeartbeat

```

DatanodeManager.java
public DatanodeCommand[] handleHeartbeat(DatanodeRegistration nodeReg,
    StorageReport[] reports, final String blockPoolId,
    long cacheCapacity, long cacheUsed, int xceiverCount,
    int maxTransfers, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary,
    @Nonnull SlowPeerReports slowPeers,
    @Nonnull SlowDiskReports slowDisks) throws IOException {
    ...
    heartbeatManager.updateHeartbeat(nodeinfo, reports, cacheCapacity,
        cacheUsed, xceiverCount, failedVolumes, volumeFailureSummary);
    ...
}

```

```

HeartbeatManager.java
synchronized void updateHeartbeat(final DatanodeDescriptor node,
    StorageReport[] reports, long cacheCapacity, long cacheUsed,
    int xceiverCount, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary) {

```

```
stats.subtract(node);
blockManager.updateHeartbeat(node, reports, cacheCapacity, cacheUsed,
    xceiverCount, failedVolumes, volumeFailureSummary);
stats.add(node);
}

BlockManager.java
void updateHeartbeat(DatanodeDescriptor node, StorageReport[] reports,
    long cacheCapacity, long cacheUsed, int xceiverCount, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary) {

    for (StorageReport report: reports) {
        providedStorageMap.updateStorage(node, report.getStorage());
    }
    node.updateHeartbeat(reports, cacheCapacity, cacheUsed, xceiverCount,
        failedVolumes, volumeFailureSummary);
}

DatanodeDescriptor.java
void updateHeartbeat(StorageReport[] reports, long cacheCapacity,
    long cacheUsed, int xceiverCount, int volFailures,
    VolumeFailureSummary volumeFailureSummary) {
    updateHeartbeatState(reports, cacheCapacity, cacheUsed, xceiverCount,
        volFailures, volumeFailureSummary);
    heartbeatedSinceRegistration = true;
}

void updateHeartbeatState(StorageReport[] reports, long cacheCapacity,
    long cacheUsed, int xceiverCount, int volFailures,
    VolumeFailureSummary volumeFailureSummary) {
    // 更新存储
    updateStorageStats(reports, cacheCapacity, cacheUsed, xceiverCount,
        volFailures, volumeFailureSummary);
    // 更新心跳时间
    setLastUpdate(Time.now());
    setLastUpdateMonotonic(Time.monotonicNow());
    rollBlocksScheduled(getLastUpdateMonotonic());
}

private void updateStorageStats(StorageReport[] reports, long cacheCapacity,
    long cacheUsed, int xceiverCount, int volFailures,
    VolumeFailureSummary volumeFailureSummary) {
    long totalCapacity = 0;
    long totalRemaining = 0;
    long totalBlockPoolUsed = 0;
    long totalDfsUsed = 0;
    long totalNonDfsUsed = 0;
    ...
    setCacheCapacity(cacheCapacity);
    setCacheUsed(cacheUsed);
    setXceiverCount(xceiverCount);
    this.volFailures = volFailures;
    this.volumeFailureSummary = volumeFailureSummary;
    for (StorageReport report : reports) {

        DatanodeStorageInfo storage =
            storageMap.get(report.getStorage().getStorageID());
    }
}
```

```
if (checkFailedStorages) {
    failedStorageInfos.remove(storage);
}

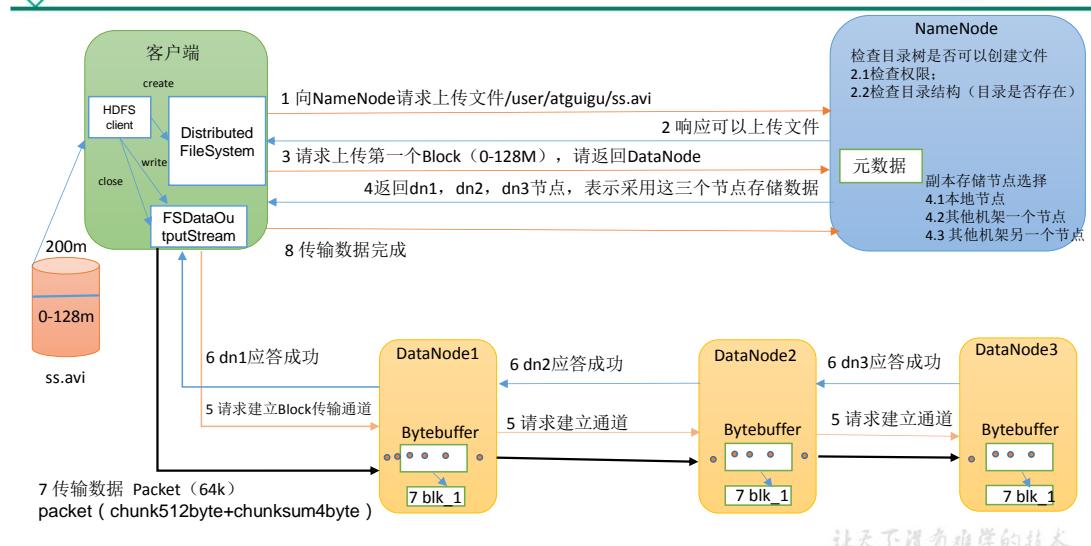
storage.receivedHeartbeat(report);
// skip accounting for capacity of PROVIDED storages!
if (StorageType.PROVIDED.equals(storage.getStorageType())) {
    continue;
}

totalCapacity += report.getCapacity();
totalRemaining += report.getRemaining();
totalBlockPoolUsed += report.getBlockPoolUsed();
totalDfsUsed += report.getDfsUsed();
totalNonDfsUsed += report.getNonDfsUsed();
}

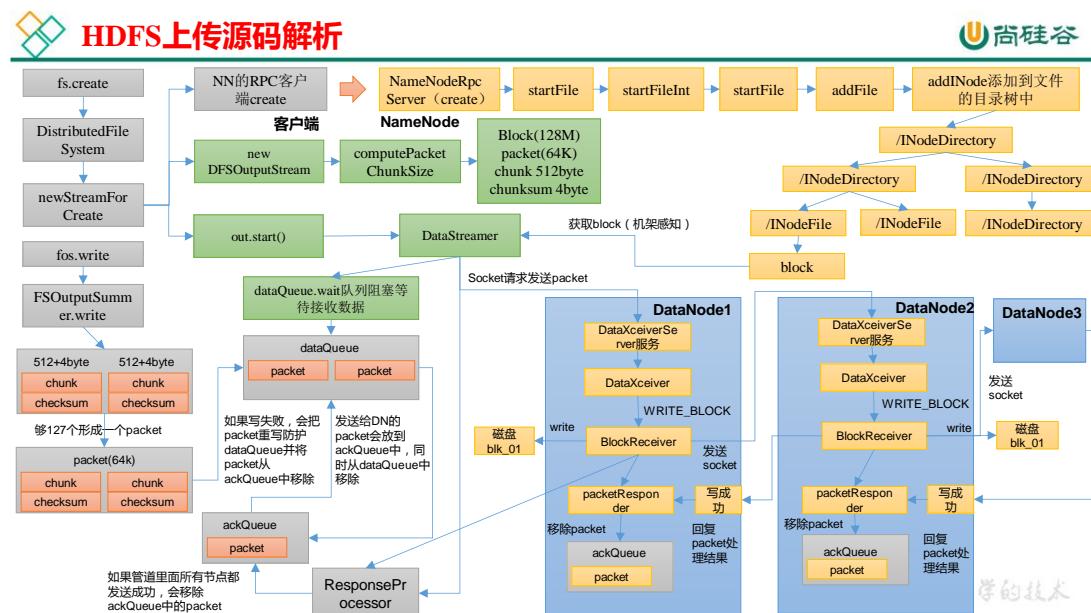
// Update total metrics for the node.
// 更新存储相关信息
setCapacity(totalCapacity);
setRemaining(totalRemaining);
setBlockPoolUsed(totalBlockPoolUsed);
setDfsUsed(totalDfsUsed);
setNonDfsUsed(totalNonDfsUsed);
if (checkFailedStorages) {
    updateFailedStorage(failedStorageInfos);
}
long storageMapSize;
synchronized (storageMap) {
    storageMapSize = storageMap.size();
}
if (storageMapSize != reports.length) {
    pruneStorageMap(reports);
}
}
```

第3章 HDFS上传源码解析

HDFS的写数据流程



 HDFS上传源码解析



3.1 create 创建过程

添加依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
```

```
<version>3.1.3</version>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs-client</artifactId>
    <version>3.1.3</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.30</version>
</dependency>
</dependencies>
```

3.1.1 DN 向 NN 发起创建请求

用户自己写的代码

```
@Test
public void testPut2() throws IOException {
    FSDataOutputStream fos = fs.create(new Path("/input"));

    fos.write("hello world".getBytes());
}
```

FileSystem.java

```
public FSDataOutputStream create(Path f) throws IOException {
    return create(f, true);
}

public FSDataOutputStream create(Path f, boolean overwrite)
    throws IOException {

    return create(f, overwrite,
        getConf().getInt(IO_FILE_BUFFER_SIZE_KEY,
            IO_FILE_BUFFER_SIZE_DEFAULT),
        getDefaultReplication(f),
        getDefaultBlockSize(f));
}

public FSDataOutputStream create(Path f,
    boolean overwrite,
    int bufferSize,
    short replication,
    long blockSize) throws IOException {

    return create(f, overwrite, bufferSize, replication, blockSize, null);
}
```

```
public FSDataOutputStream create(Path f,
    boolean overwrite,
    int bufferSize,
    short replication,
    long blockSize,
    Progressable progress
) throws IOException {

    return this.create(f, FsCreateModes.applyUMask(
        FsPermission.getFileDefault(), FsPermission.getUMask(getConf())),
        overwrite, bufferSize, replication, blockSize, progress);
}

public abstract FSDataOutputStream create(Path f,
    FsPermission permission,
    boolean overwrite,
    int bufferSize,
    short replication,
    long blockSize,
    Progressable progress) throws IOException;
```

选中 create，点击 **ctrl+h**，找到实现类 `DistributedFileSystem.java`，查找 create 方法。

`DistributedFileSystem.java`

```
@Override
public FSDataOutputStream create(Path f, FsPermission permission,
    boolean overwrite, int bufferSize, short replication, long blockSize,
    Progressable progress) throws IOException {

    return this.create(f, permission,
        overwrite ? EnumSet.of(CreateFlag.CREATE, CreateFlag.OVERWRITE)
            : EnumSet.of(CreateFlag.CREATE), bufferSize, replication,
            blockSize, progress, null);
}

@Override
public FSDataOutputStream create(final Path f, final FsPermission permission,
    final EnumSet<CreateFlag> cflags, final int bufferSize,
    final short replication, final long blockSize,
    final Progressable progress, final ChecksumOpt checksumOpt)
throws IOException {

    statistics.incrementWriteOps(1);
    storageStatistics.incrementOpCounter(OpType.CREATE);
    Path absF = fixRelativePart(f);

    return new FileSystemLinkResolver<FSDataOutputStream>() {

        @Override
        public FSDataOutputStream doCall(final Path p) throws IOException {

            // 创建获取了一个输出流对象
            final DFSOutputStream dfsos = dfs.create(getPathName(p), permission,
                cflags, replication, blockSize, progress, bufferSize,
                checksumOpt);
            // 这里将上面创建的 dfsos 进行包装并返回
        }
    };
}
```

```

        return dfs.createWrappedOutputStream(dfsos, statistics);
    }

    @Override
    public FSDataOutputStream next(final FileSystem fs, final Path p)
        throws IOException {
        return fs.create(p, permission, cflags, bufferSize,
            replication, blockSize, progress, checksumOpt);
    }
}.resolve(this, absF);
}

```

点击 create，进入 DFSClient.java

```

public DFSOutputStream create(String src, FsPermission permission,
    EnumSet<CreateFlag> flag, short replication, long blockSize,
    Progressable progress, int buffersize, ChecksumOpt checksumOpt)
    throws IOException {

    return create(src, permission, flag, true,
        replication, blockSize, progress, buffersize, checksumOpt, null);
}

public DFSOutputStream create(String src, FsPermission permission,
    EnumSet<CreateFlag> flag, boolean createParent, short replication,
    long blockSize, Progressable progress, int buffersize,
    ChecksumOpt checksumOpt, InetSocketAddress[] favoredNodes)
    throws IOException {

    return create(src, permission, flag, createParent, replication, blockSize,
        progress, buffersize, checksumOpt, favoredNodes, null);
}

public DFSOutputStream create(String src, FsPermission permission,
    EnumSet<CreateFlag> flag, boolean createParent, short replication,
    long blockSize, Progressable progress, int buffersize,
    ChecksumOpt checksumOpt, InetSocketAddress[] favoredNodes,
    String ecPolicyName) throws IOException {

    checkOpen();

    final FsPermission masked = applyUMask(permission);
    LOG.debug("{}: masked={}", src, masked);

    final DFSOutputStream result = DFSOutputStream.newStreamForCreate(this,
        src, masked, flag, createParent, replication, blockSize, progress,
        dfsClientConf.createChecksum(checksumOpt),
        getFavoredNodesStr(favoredNodes), ecPolicyName);

    beginFileLease(result.getFileId(), result);

    return result;
}

```

点击 newStreamForCreate，进入 DFSOutputStream.java

```

static DFSOutputStream newStreamForCreate(DFSClient dfsClient, String src,
    FsPermission masked, EnumSet<CreateFlag> flag, boolean createParent,
    short replication, long blockSize, Progressable progress,

```

```

DataChecksum checksum, String[] favoredNodes, String ecPolicyName)
throws IOException {

    try (TraceScope ignored =
        dfsClient newPathTraceScope("newStreamForCreate", src)) {
        HdfsFileStatus stat = null;

        // Retry the create if we get a RetryStartFileException up to a maximum
        // number of times
        boolean shouldRetry = true;
        int retryCount = CREATE_RETRY_COUNT;

        while (shouldRetry) {
            shouldRetry = false;
            try {
                // DN 将创建请求发送给 NN (RPC)
                stat = dfsClient.namenode.create(src, masked, dfsClient.clientName,
                    new EnumSetWritable<>(flag), createParent, replication,
                    blockSize, SUPPORTED_CRYPTO_VERSIONS, ecPolicyName);
                break;
            } catch (RemoteException re) {
                ...
            }
        }
        Preconditions.checkNotNull(stat, "HdfsFileStatus should not be null!");
        final DFSOutputStream out;

        if(stat.getErasureCodingPolicy() != null) {
            out = new DFSStripedOutputStream(dfsClient, src, stat,
                flag, progress, checksum, favoredNodes);
        } else {
            out = new DFSOutputStream(dfsClient, src, stat,
                flag, progress, checksum, favoredNodes, true);
        }

        // 开启线程 run, DataStreamer extends Daemon extends Thread
        out.start();

        return out;
    }
}

```

3.1.2 NN 处理 DN 的创建请求

1) 点击 create

ClientProtocol.java

```

HdfsFileStatus create(String src, FsPermission masked,
    String clientName, EnumSetWritable<CreateFlag> flag,
    boolean createParent, short replication, long blockSize,
    CryptoProtocolVersion[] supportedVersions, String ecPolicyName)
throws IOException;

```

2) Ctrl + h 查找 create 实现类，点击 NameNodeRpcServer，在 NameNodeRpcServer.java 中搜索 create

NameNodeRpcServer.java

```
public HdfsFileStatus create(String src, FsPermission masked,
    String clientName, EnumSetWritable<CreateFlag> flag,
    boolean createParent, short replication, long blockSize,
    CryptoProtocolVersion[] supportedVersions, String ecPolicyName)
    throws IOException {
    // 检查 NN 启动
    checkNNStartup();
    ... ...

    HdfsFileStatus status = null;
    try {
        PermissionStatus perm = new PermissionStatus(getRemoteUser()
            .getShortUserName(), null, masked);
        // 重要
        status = namesystem.startFile(src, perm, clientName, clientMachine,
            flag.get(), createParent, replication, blockSize, supportedVersions,
            ecPolicyName, cacheEntry != null);
    } finally {
        RetryCache.setState(cacheEntry, status != null, status);
    }

    metrics.incrFilesCreated();
    metrics.incrCreateFileOps();
    return status;
}
```

FSNamesystem.java

```
HdfsFileStatus startFile(String src, PermissionStatus permissions,
    String holder, String clientMachine, EnumSet<CreateFlag> flag,
    boolean createParent, short replication, long blockSize,
    CryptoProtocolVersion[] supportedVersions, String ecPolicyName,
    boolean logRetryCache) throws IOException {

    HdfsFileStatus status;
    try {
        status = startFileInt(src, permissions, holder, clientMachine, flag,
            createParent, replication, blockSize, supportedVersions, ecPolicyName,
            logRetryCache);
    } catch (AccessControlException e) {
        logAuditEvent(false, "create", src);
        throw e;
    }
    logAuditEvent(true, "create", src, status);
    return status;
}

private HdfsFileStatus startFileInt(String src,
    PermissionStatus permissions, String holder, String clientMachine,
    EnumSet<CreateFlag> flag, boolean createParent, short replication,
    long blockSize, CryptoProtocolVersion[] supportedVersions,
    String ecPolicyName, boolean logRetryCache) throws IOException {
    ...
    stat = FSDirWriteFileOp.startFile(this, iip, permissions, holder,
        clientMachine, flag, createParent, replication, blockSize, feInfo,
        toRemoveBlocks, shouldReplicate, ecPolicyName, logRetryCache);
}
```

```
    ... ...
}

static HdfsFileStatus startFile(
    ... ...)
    throws IOException {

    ... ...
    FSDirectory fsd = fsn.getFSDirectory();

    // 文件路径是否存在校验
    if (iip.getLastINode() != null) {
        if (overwrite) {
            List<INode> toRemoveINodes = new ChunkedArrayList<>();
            List<Long> toRemoveUCFiles = new ChunkedArrayList<>();
            long ret = FSDirDeleteOp.delete(fsd, iip, toRemoveBlocks,
                toRemoveINodes, toRemoveUCFiles, now());
            if (ret >= 0) {
                iip = INodesInPath.replace(iip, iip.length() - 1, null);
                FSDirDeleteOp.incrDeletedFileCount(ret);
                fsn.removeLeasesAndINodes(toRemoveUCFiles, toRemoveINodes, true);
            }
        } else {
            // If lease soft limit time is expired, recover the lease
            fsn.recoverLeaseInternal(FSNamesystem.RecoverLeaseOp.CREATE_FILE, iip,
                src, holder, clientMachine, false);
            throw new FileAlreadyExistsException(src + " for client "
                + clientMachine + " already exists");
        }
    }
    fsn.checkFsObjectLimit();
    INodeFile newNode = null;
    INodesInPath parent = FSDirMkdirOp.createAncestorDirectories(fsd, iip, permissions);
    if (parent != null) {
        // 添加文件元数据信息
        iip = addFile(fsd, parent, iip.getLastLocalName(), permissions,
            replication, blockSize, holder, clientMachine, shouldReplicate,
            ecPolicyName);
        newNode = iip != null ? iip.getLastINode().asFile() : null;
    }
    ... ...
    setNewINodeStoragePolicy(fsd.getBlockManager(), iip, isLazyPersist);
    fsd.getEditLog().logOpenFile(src, newNode, overwrite, logRetryEntry);
    if (NameNode.stateChangeLog.isDebugEnabled()) {
        NameNode.stateChangeLog.debug("DIR* NameSystem.startFile: added " +
            src + " inode " + newNode.getId() + " " + holder);
    }
    return FSDirStatAndListingOp.getFileInfo(fsd, iip, false, false);
}

private static INodesInPath addFile(
    FSDirectory fsd, INodesInPath existing, byte[] localName,
    PermissionStatus permissions, short replication, long preferredBlockSize,
    String clientName, String clientMachine, boolean shouldReplicate,
    String ecPolicyName) throws IOException {
```

```

Preconditions.checkNotNull(existing);
long modTime = now();
INodesInPath newiip;
fsd.writeLock();
try {
    ...
    newiip = fsd.addINode(existing, newNode, permissions.getPermission());
} finally {
    fsd.writeUnlock();
}
...
return newiip;
}

INodesInPath addINode(INodesInPath existing, INode child,
                      FsPermission modes)
throws QuotaExceededException, UnresolvedLinkException {
cacheName(child);
writeLock();
try {
    // 将数据写入到 INode 的目录树中
    return addLastINode(existing, child, modes, true);
} finally {
    writeUnlock();
}
}

```

3.1.3 DataStreamer 启动流程

NN 处理完 DN 请求后，再次回到 DN 端，启动对应的线程

DFSOutputStream.java

```

static DFSOutputStream newStreamForCreate(DFSClient dfsClient, String src,
                                         FsPermission masked, EnumSet<CreateFlag> flag, boolean createParent,
                                         short replication, long blockSize, Progressable progress,
                                         DataChecksum checksum, String[] favoredNodes, String ecPolicyName)
throws IOException {
    ...
    // DN 将创建请求发送给 NN (RPC)
    stat = dfsClient.namenode.create(src, masked, dfsClient.clientName,
                                     new EnumSetWritable<>(flag), createParent, replication,
                                     blockSize, SUPPORTED_CRYPTO_VERSIONS, ecPolicyName);
    ...
    // 创建输出流
    out = new DFSOutputStream(dfsClient, src, stat,
                             flag, progress, checksum, favoredNodes, true);
    // 开启线程 run, DataStreamer extends Daemon extends Thread
    out.start();
    ...
    return out;
}

```

点击 DFSOutputStream

```
protected DFSOutputStream(DFSClient dfsClient, String src,
```

```

HdfsFileStatus stat, EnumSet<CreateFlag> flag, Progressable progress,
    DataChecksum checksum, String[] favoredNodes, boolean createStreamer) {
    this(dfsClient, src, flag, progress, stat, checksum);
    this.shouldSyncBlock = flag.contains(CreateFlag.SYNC_BLOCK);

    // Directory => File => Block(128M) => packet(64K) => chunk (chunk 512byte +
    chunksum 4byte)
    computePacketChunkSize(dfsClient.getConf().getWritePacketSize(),
        bytesPerChecksum);

    if (createStreamer) {
        streamer = new DataStreamer(stat, null, dfsClient, src, progress,
            checksum, cachingStrategy, byteArrayManager, favoredNodes,
            addBlockFlags);
    }
}

```

1) 点击 newStreamForCreate 方法中的 out.start(), 进入 DFSOutputStream.java

```

protected synchronized void start() {
    getStreamer().start();
}

protected DataStreamer getStreamer() {
    return streamer;
}

```

点击 DataStreamer, 进入 DataStreamer.java

```

class DataStreamer extends Daemon {
    ...
}

```

点击 Daemon, 进入 Daemon.java

```

public class Daemon extends Thread {
    ...
}

```

说明: out.start(); 实际是开启线程, 点击 DataStreamer, 搜索 run 方法

DataStreamer.java

```

@Override
public void run() {

    long lastPacket = Time.monotonicNow();
    TraceScope scope = null;
    while (!streamerClosed && dfsClient.clientRunning) {
        // if the Responder encountered an error, shutdown Responder
        if (errorState.hasError()) {
            closeResponder();
        }

        DFSPacket one;
        try {
            // process datanode IO errors if any
            boolean doSleep = processDatanodeOrExternalError();

            final int halfSocketTimeout = dfsClient.getConf().getSocketTimeout()/2;

```

```

synchronized (dataQueue) {
    // wait for a packet to be sent.
    ...
    try {
        // 如果 dataQueue 里面没有数据，代码会阻塞在这儿
        dataQueue.wait(timeout);
    } catch (InterruptedException e) {
        LOG.warn("Caught exception", e);
    }
    doSleep = false;
    now = Time.monotonicNow();
}
...
// 队列不为空，从队列中取出 packet
one = dataQueue.getFirst(); // regular data packet
SpanId[] parents = one.getTraceParents();
if (parents.length > 0) {
    scope = dfsClient.getTracer().
        newScope("dataStreamer", parents[0]);
    scope.getSpan().setParents(parents);
}
...
}
...
}
}

```

3.2 write 上传过程

3.1.1 向 DataStreamer 的队列里面写数据

1) 用户写的代码

```

@Test
public void testPut2() throws IOException {
    FSDataOutputStream fos = fs.create(new Path("/input"));

    fos.write("hello world".getBytes());
}

```

2) 点击 write

```

FilterOutputStream.java
public void write(byte b[]) throws IOException {
    write(b, 0, b.length);
}

public void write(byte b[], int off, int len) throws IOException {
    if ((off | len | (b.length - (len + off)) | (off + len)) < 0)
        throw new IndexOutOfBoundsException();

    for (int i = 0 ; i < len ; i++) {
        write(b[off + i]);
    }
}

public void write(int b) throws IOException {

```

```
    out.write(b);
}
```

3) 点击 write

OutputStream.java

```
public abstract void write(int b) throws IOException;
```

ctrl + h 查找 write 实现类，选择 FSOOutputSummer.java，在该类中查找 write

FSOutputSummer.java

```
public synchronized void write(int b) throws IOException {
    buf[count++] = (byte)b;
    if(count == buf.length) {
        flushBuffer();
    }
}

protected synchronized void flushBuffer() throws IOException {
    flushBuffer(false, true);
}

protected synchronized int flushBuffer(boolean keep,
    boolean flushPartial) throws IOException {
    int bufLen = count;
    int partialLen = bufLen % sum.getBytesPerChecksum();
    int lenToFlush = flushPartial ? bufLen : bufLen - partialLen;

    if (lenToFlush != 0) {
        // 向队列中写数据
        // Directory => File => Block(128M) => package(64K) => chunk (chunk 512byte +
        chunksum 4byte)
        writeChecksumChunks(buf, 0, lenToFlush);

        if (!flushPartial || keep) {
            count = partialLen;
            System.arraycopy(buf, bufLen - count, buf, 0, count);
        } else {
            count = 0;
        }
    }

    // total bytes left minus unflushed bytes left
    return count - (bufLen - lenToFlush);
}

private void writeChecksumChunks(byte b[], int off, int len)
throws IOException {

    // 计算 chunk 的校验和
    sum.calculateChunkedSums(b, off, len, checksum, 0);
    TraceScope scope = createWriteTraceScope();

    // 按照 chunk 的大小遍历数据
    try {
        for (int i = 0; i < len; i += sum.getBytesPerChecksum()) {
```

```

        int chunkLen = Math.min(sum.getBytesPerChecksum(), len - i);
        int ckOffset = i / sum.getBytesPerChecksum() * getChecksumSize();

        // 一个 chunk 一个 chunk 的将数据写入队列
        writeChunk(b, off + i, chunkLen, checksum, ckOffset,
                   getChecksumSize());
    }
} finally {
    if (scope != null) {
        scope.close();
    }
}
}

protected abstract void writeChunk(byte[] b, int bOffset, int bLen,
                                    byte[] checksum, int checksumOffset, int checksumLen) throws IOException;

```

ctrl + h 查找 writeChunk 实现类 DFSOutputStream.java

```

protected synchronized void writeChunk(byte[] b, int offset, int len,
                                       byte[] checksum, int ckoff, int cklen) throws IOException {

    writeChunkPrepare(len, ckoff, cklen);

    // 往 packet 里面写 chunk 的校验和 4byte
    currentPacket.writeChecksum(checksum, ckoff, cklen);

    // 往 packet 里面写一个 chunk 512 byte
    currentPacket.writeData(b, offset, len);

    // 记录写入 packet 中的 chunk 个数，累计到 127 个 chuck，这个 packet 就满了
    currentPacket.incNumChunks();
    getStreamer().incBytesCurBlock(len);

    // If packet is full, enqueue it for transmission
    if (currentPacket.getNumChunks() == currentPacket.getMaxChunks() ||
        getStreamer().getBytesCurBlock() == blockSize) {
        enqueueCurrentPacketFull();
    }
}

```

```

synchronized void enqueueCurrentPacketFull() throws IOException {
    LOG.debug("enqueue full {}, src={}, bytesCurBlock={}, blockSize={}",
              + " appendChunk={}, {}", currentPacket, src, getStreamer()
              .getBytesCurBlock(), blockSize, getStreamer().getAppendChunk(),
              getStreamer());

    enqueueCurrentPacket();

    adjustChunkBoundary();

    endBlock();
}

void enqueueCurrentPacket() throws IOException {
    getStreamer().waitAndQueuePacket(currentPacket);
    currentPacket = null;
}

```

```
}

void waitAndQueuePacket(DFSPacket packet) throws IOException {
    synchronized (dataQueue) {
        try {
            // 如果队列满了，等待
            // If queue is full, then wait till we have enough space
            boolean firstWait = true;
            try {
                while (!streamerClosed && dataQueue.size() + ackQueue.size() >
                    dfsClient.getConf().getWriteMaxPackets()) {
                    if (firstWait) {
                        Span span = Tracer.getCurrentSpan();
                        if (span != null) {
                            span.addTimelineAnnotation("dataQueue.wait");
                        }
                        firstWait = false;
                    }
                    try {
                        dataQueue.wait();
                    } catch (InterruptedException e) {
                        ...
                    }
                }
            } finally {
                Span span = Tracer.getCurrentSpan();
                if ((span != null) && (!firstWait)) {
                    span.addTimelineAnnotation("end.wait");
                }
            }
            checkClosed();
            // 如果队列没满，向队列中添加数据
            queuePacket(packet);
        } catch (ClosedChannelException ignored) {
        }
    }
}
```

DataStreamer.java

```
void queuePacket(DFSPacket packet) {
    synchronized (dataQueue) {
        if (packet == null) return;
        packet.addTraceParent(Tracer.getCurrentSpanId());

        // 向队列中添加数据
        dataQueue.addLast(packet);

        lastQueuedSeqno = packet.getSeqno();
        LOG.debug("Queued {}, {}", packet, this);

        // 通知队列添加数据完成
        dataQueue.notifyAll();
    }
}
```

3.1.2 建立管道之机架感知（块存储位置）

Ctrl + n 全局查找 DataStreamer，搜索 run 方法

DataStreamer.java

```
@Override
public void run() {

    long lastPacket = Time.monotonicNow();
    TraceScope scope = null;
    while (!streamerClosed && dfsClient.clientRunning) {
        // if the Responder encountered an error, shutdown Responder
        if (errorState.hasError()) {
            closeResponder();
        }

        DFSPacket one;
        try {
            // process datanode IO errors if any
            boolean doSleep = processDatanodeOrExternalError();

            final int halfSocketTimeout = dfsClient.getConf().getSocketTimeout()/2;
            synchronized (dataQueue) {
                // wait for a packet to be sent.
                long now = Time.monotonicNow();
                while ((!shouldStop() && dataQueue.size() == 0 &&
                        (stage != BlockConstructionStage.DATA_STREAMING ||
                         now - lastPacket < halfSocketTimeout)) || doSleep) {
                    long timeout = halfSocketTimeout - (now - lastPacket);
                    timeout = timeout <= 0 ? 1000 : timeout;
                    timeout = (stage == BlockConstructionStage.DATA_STREAMING)?
                        timeout : 1000;
                    try {
                        // 如果 dataQueue 里面没有数据，代码会阻塞在这儿
                        dataQueue.wait(timeout); // 接收到 notify 消息
                    } catch (InterruptedException e) {
                        LOG.warn("Caught exception", e);
                    }
                    doSleep = false;
                    now = Time.monotonicNow();
                }
                if (shouldStop()) {
                    continue;
                }
                // get packet to be sent.
                if (dataQueue.isEmpty()) {
                    one = createHeartbeatPacket();
                } else {
                    try {
                        backOffIfNecessary();
                    } catch (InterruptedException e) {
                        LOG.warn("Caught exception", e);
                    }
                    // 队列不为空，从队列中取出 packet
                    one = dataQueue.getFirst(); // regular data packet
                }
            }
        }
    }
}
```

```
SpanId[] parents = one.getTraceParents();
if (parents.length > 0) {
    scope = dfsClient.getTracer().
        newScope("dataStreamer", parents[0]);
    scope.getSpan().setParents(parents);
}
}

// get new block from namenode.
if (LOG.isDebugEnabled()) {
    LOG.debug("stage=" + stage + ", " + this);
}
if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
    LOG.debug("Allocating new block: {}", this);
    // 步骤一：向 NameNode 申请 block 并建立数据管道
    setPipeline(nextBlockOutputStream());
    // 步骤二：启动 ResponseProcessor 用来监听 packet 发送是否成功
    initDataStreaming();
} else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
    setupPipelineForAppendOrRecovery();
    if (streamerClosed) {
        continue;
    }
    initDataStreaming();
}

long lastByteOffsetInBlock = one.getLastByteOffsetBlock();
if (lastByteOffsetInBlock > stat.getBlockSize()) {
    throw new IOException("BlockSize " + stat.getBlockSize() +
        " < lastByteOffsetInBlock, " + this + ", " + one);
}
...
// send the packet
SpanId spanId = SpanId.INVALID;
synchronized (dataQueue) {

    // move packet from dataQueue to ackQueue
    if (!one.isHeartbeatPacket()) {
        if (scope != null) {
            spanId = scope.getSpanId();
            scope.detach();

            one.setTraceScope(scope);
        }
        scope = null;
        // 步骤三：从 dataQueue 把要发送的这个 packet 移除出去
        dataQueue.removeFirst();
        // 步骤四：然后往 ackQueue 里面添加这个 packet
        ackQueue.addLast(one);
        packetSendTime.put(one.getSeqno(), Time.monotonicNow());
        dataQueue.notifyAll();
    }
}

LOG.debug("{} sending {}", this, one);
```

```
// write out data to remote datanode
try (TraceScope ignored = dfsClient.getTracer().newScope("DataStreamer#writeTo", spanId)) {
    // 将数据写出去
    one.writeTo(blockStream);
    blockStream.flush();
} catch (IOException e) {
    errorState.markFirstNodeIfNotMarked();
    throw e;
}
...
}
```

点击 nextBlockOutputStream

```
protected LocatedBlock nextBlockOutputStream() throws IOException {
    LocatedBlock lb;
    DatanodeInfo[] nodes;
    StorageType[] nextStorageTypes;
    String[] nextStorageIDs;
    int count = dfsClient.getConf().getNumBlockWriteRetry();
    boolean success;
    final ExtendedBlock oldBlock = block.getCurrentBlock();
    do {
        errorState.resetInternalError();
        lastException.clear();

        DatanodeInfo[] excluded = getExcludedNodes();
        // 向 NN 获取向哪个 DN 写数据
        lb = locateFollowingBlock(
            excluded.length > 0 ? excluded : null, oldBlock);

        // 创建管道
        success = createBlockOutputStream(nodes, nextStorageTypes, nextStorageIDs,
            0L, false);
        ...
    } while (!success && --count >= 0);

    if (!success) {
        throw new IOException("Unable to create new block.");
    }
    return lb;
}

private LocatedBlock locateFollowingBlock(DatanodeInfo[] excluded,
    ExtendedBlock oldBlock) throws IOException {
    return DFSOutputStream.addBlock(excluded, dfsClient, src, oldBlock,
        stat.getFileId(), favoredNodes, addBlockFlags);
}

static LocatedBlock addBlock(DatanodeInfo[] excludedNodes,
    DFSClient dfsClient, String src, ExtendedBlock prevBlock, long fileId,
    String[] favoredNodes, EnumSet<AddBlockFlag> allocFlags)
throws IOException {
    ...
    // 向 NN 获取向哪个 DN 写数据
}
```

```

        return dfsClient.namenode.addBlock(src, dfsClient.clientName, prevBlock,
                                         excludedNodes, fileId, favoredNodes, allocFlags);
    }

    ...

    LocatedBlock addBlock(String src, String clientName,
                          ExtendedBlock previous, DatanodeInfo[] excludeNodes, long fileId,
                          String[] favoredNodes, EnumSet<AddBlockFlag> addBlockFlags)
    throws IOException;
}

```

ctrl + h 点击 NameNodeRpcServer， 在该类中搜索 addBlock

NameNodeRpcServer.java

```

public LocatedBlock addBlock(String src, String clientName,
                            ExtendedBlock previous, DatanodeInfo[] excludedNodes, long fileId,
                            String[] favoredNodes, EnumSet<AddBlockFlag> addBlockFlags)
    throws IOException {
    checkNNStartup();
    LocatedBlock locatedBlock = namesystem.getAdditionalBlock(src, fileId,
                                                               clientName, previous, excludedNodes, favoredNodes, addBlockFlags);
    if (locatedBlock != null) {
        metrics.incrAddBlockOps();
    }
    return locatedBlock;
}

```

FSNamesystrm.java

```

LocatedBlock getAdditionalBlock(
    String src, long fileId, String clientName, ExtendedBlock previous,
    DatanodeInfo[] excludedNodes, String[] favoredNodes,
    EnumSet<AddBlockFlag> flags) throws IOException {
    final String operationName = "getAdditionalBlock";
    NameNode.stateChangeLog.debug("BLOCK* getAdditionalBlock: {} inodeId {}" +
        " for {}", src, fileId, clientName);

    ...

    // 选择块存储位置
    DatanodeStorageInfo[] targets = FSDirWriteFileOp.chooseTargetForNewBlock(
        blockManager, src, excludedNodes, favoredNodes, flags, r);

    ...
    return lb;
}

static DatanodeStorageInfo[] chooseTargetForNewBlock(
    BlockManager bm, String src, DatanodeInfo[] excludedNodes,
    String[] favoredNodes, EnumSet<AddBlockFlag> flags,
    ValidateAddBlockResult r) throws IOException {
    ...
    return bm.chooseTarget4NewBlock(src, r.numTargets, clientNode,
                                    excludedNodesSet, r.blockSize,
                                    favoredNodesList, r.storagePolicyID,
                                    r.blockType, r.ecPolicy, flags);
}

public DatanodeStorageInfo[] chooseTarget4NewBlock(... ...)
    throws IOException {
}

```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
... ...
final DatanodeStorageInfo[] targets = blockplacement.chooseTarget(src,
    numOfReplicas, client, excludedNodes, blocksize,
    favoredDatanodeDescriptors, storagePolicy, flags);

...
return targets;
}

DatanodeStorageInfo[] chooseTarget(String src,
    int numOfReplicas, Node writer,
    Set<Node> excludedNodes,
    long blocksize,
    List<DatanodeDescriptor> favoredNodes,
    BlockStoragePolicy storagePolicy,
    EnumSet<AddBlockFlag> flags) {

    return chooseTarget(src, numOfReplicas, writer,
        new ArrayList<DatanodeStorageInfo>(numOfReplicas), false,
        excludedNodes, blocksize, storagePolicy, flags);
}

public abstract DatanodeStorageInfo[] chooseTarget(String srcPath,
    int numOfReplicas,
    Node writer,
    List<DatanodeStorageInfo> chosen,
    boolean returnChosenNodes,
    Set<Node> excludedNodes,
    long blocksize,
    BlockStoragePolicy storagePolicy,
    EnumSet<AddBlockFlag> flags);
```

Ctrl + h 查找 chooseTarget 实现类 BlockPlacementPolicyDefault.java

```
public DatanodeStorageInfo[] chooseTarget(String srcPath,
    int numOfReplicas,
    Node writer,
    List<DatanodeStorageInfo> chosenNodes,
    boolean returnChosenNodes,
    Set<Node> excludedNodes,
    long blocksize,
    final BlockStoragePolicy storagePolicy,
    EnumSet<AddBlockFlag> flags) {

    return chooseTarget(numOfReplicas, writer, chosenNodes, returnChosenNodes,
        excludedNodes, blocksize, storagePolicy, flags, null);
}

private DatanodeStorageInfo[] chooseTarget(int numOfReplicas,
    Node writer,
    List<DatanodeStorageInfo> chosenStorage,
    boolean returnChosenNodes,
    Set<Node> excludedNodes,
    long blocksize,
    final BlockStoragePolicy storagePolicy,
    EnumSet<AddBlockFlag> addBlockFlags,
    EnumMap<StorageType, Integer> sTypes) {
```

```
....  
  
int[] result = getMaxNodesPerRack(chosenStorage.size(), numOfReplicas);  
numOfReplicas = result[0];  
int maxNodesPerRack = result[1];  
  
for (DatanodeStorageInfo storage : chosenStorage) {  
    // add localMachine and related nodes to excludedNodes  
    // 获取不可用的 DN  
    addToExcludedNodes(storage.getDatanodeDescriptor(), excludedNodes);  
}  
  
List<DatanodeStorageInfo> results = null;  
Node localNode = null;  
boolean avoidStaleNodes = (stats != null  
    && stats.isAvoidingStaleDataNodesForWrite());  
//  
boolean avoidLocalNode = (addBlockFlags != null  
    && addBlockFlags.contains(AddBlockFlag.NO_LOCAL_WRITE)  
    && writer != null  
    && !excludedNodes.contains(writer));  
// Attempt to exclude local node if the client suggests so. If no enough  
// nodes can be obtained, it falls back to the default block placement  
// policy.  
  
// 有数据正在写，避免都写入本地  
if (avoidLocalNode) {  
    results = new ArrayList<>(chosenStorage);  
    Set<Node> excludedNodeCopy = new HashSet<>(excludedNodes);  
    if (writer != null) {  
        excludedNodeCopy.add(writer);  
    }  
    localNode = chooseTarget(numOfReplicas, writer,  
        excludedNodeCopy, blocksize, maxNodesPerRack, results,  
        avoidStaleNodes, storagePolicy,  
        EnumSet.noneOf(StorageType.class), results.isEmpty(), sTypes);  
    if (results.size() < numOfReplicas) {  
        // not enough nodes; discard results and fall back  
        results = null;  
    }  
}  
if (results == null) {  
    results = new ArrayList<>(chosenStorage);  
    // 真正的选择 DN 节点  
    localNode = chooseTarget(numOfReplicas, writer, excludedNodes,  
        blocksize, maxNodesPerRack, results, avoidStaleNodes,  
        storagePolicy, EnumSet.noneOf(StorageType.class), results.isEmpty(),  
        sTypes);  
}  
  
if (!returnChosenNodes) {  
    results.removeAll(chosenStorage);  
}  
  
// sorting nodes to form a pipeline  
return getPipeline()
```

```
(writer != null && writer instanceof DatanodeDescriptor) ? writer
    : localNode,
  results.toArray(new DatanodeStorageInfo[results.size()]));
}

private Node chooseTarget(int numOfReplicas,
  ... ...) {

  writer = chooseTargetInOrder(numOfReplicas, writer, excludedNodes, blocksize,
    maxNodesPerRack, results, avoidStaleNodes, newBlock, storageTypes);
  ...
}

protected Node chooseTargetInOrder(int numOfReplicas,
  Node writer,
  final Set<Node> excludedNodes,
  final long blocksize,
  final int maxNodesPerRack,
  final List<DatanodeStorageInfo> results,
  final boolean avoidStaleNodes,
  final boolean newBlock,
  EnumMap<StorageType, Integer> storageTypes)
throws NotEnoughReplicasException {
  final int numOfResults = results.size();
  if (numOfResults == 0) {
    // 第一个块存储在当前节点
    DatanodeStorageInfo storageInfo = chooseLocalStorage(writer,
      excludedNodes, blocksize, maxNodesPerRack, results, avoidStaleNodes,
      storageTypes, true);

    writer = (storageInfo != null) ? storageInfo.getDatanodeDescriptor()
      : null;

    if (--numOfReplicas == 0) {
      return writer;
    }
  }
  final DatanodeDescriptor dn0 = results.get(0).getDatanodeDescriptor();
  // 第二个块存储在另外一个机架
  if (numOfResults <= 1) {
    chooseRemoteRack(1, dn0, excludedNodes, blocksize, maxNodesPerRack,
      results, avoidStaleNodes, storageTypes);
    if (--numOfReplicas == 0) {
      return writer;
    }
  }
  if (numOfResults <= 2) {
    final DatanodeDescriptor dn1 = results.get(1).getDatanodeDescriptor();
    // 如果第一个和第二个在同一个机架，那么第三个放在其他机架
    if (clusterMap.isOnSameRack(dn0, dn1)) {
      chooseRemoteRack(1, dn0, excludedNodes, blocksize, maxNodesPerRack,
        results, avoidStaleNodes, storageTypes);
    } else if (newBlock){
      // 如果是新块，和第二个块存储在同一个机架
      chooseLocalRack(dn1, excludedNodes, blocksize, maxNodesPerRack,
        results, avoidStaleNodes, storageTypes);
    }
  }
}
```

```

    } else {
        // 如果不是新块，放在当前机架
        chooseLocalRack(writer, excludedNodes, blocksize, maxNodesPerRack,
                         results, avoidStaleNodes, storageTypes);
    }
    if (--numOfReplicas == 0) {
        return writer;
    }
}
chooseRandom(numOfReplicas, NodeBase.ROOT, excludedNodes, blocksize,
             maxNodesPerRack, results, avoidStaleNodes, storageTypes);
return writer;
}

```

3.1.3 建立管道之 Socket 发送

点击 nextBlockOutputStream

```

protected LocatedBlock nextBlockOutputStream() throws IOException {
    LocatedBlock lb;
    DatanodeInfo[] nodes;
    StorageType[] nextStorageTypes;
    String[] nextStorageIDs;
    int count = dfsClient.getConf().getNumBlockWriteRetry();
    boolean success;
    final ExtendedBlock oldBlock = block.getCurrentBlock();
    do {
        errorState.resetInternalError();
        lastException.clear();

        DatanodeInfo[] excluded = getExcludedNodes();
        // 向 NN 获取向哪个 DN 写数据
        lb = locateFollowingBlock(
            excluded.length > 0 ? excluded : null, oldBlock);

        // 创建管道
        success = createBlockOutputStream(nodes, nextStorageTypes, nextStorageIDs,
                                         0L, false);
        ...
    } while (!success && --count >= 0);

    if (!success) {
        throw new IOException("Unable to create new block.");
    }
    return lb;
}

boolean createBlockOutputStream(DatanodeInfo[] nodes,
                               StorageType[] nodeStorageTypes, String[] nodeStorageIDs,
                               long newGS, boolean recoveryFlag) {
    ...
    // 和 DN 创建 socket
    s = createSocketForPipeline(nodes[0], nodes.length, dfsClient);

    // 获取输出流，用于写数据到 DN
    OutputStream unbufOut = NetUtils.getOutputStream(s, writeTimeout);
}

```

```

// 获取输入流，用于读取写数据到 DN 的结果
InputStream unbufIn = NetUtils.getInputStream(s, readTimeout);

IOStreamPair saslStreams = dfsClient.saslClient.socketSend(s,
    unbufOut, unbufIn, dfsClient, accessToken, nodes[0]);
unbufOut = saslStreams.out;
unbufIn = saslStreams.in;
out = new DataOutputStream(new BufferedOutputStream(unbufOut,
    DFSUtilClient.getSmallBufferSize(dfsClient.getConfiguration())));
blockReplyStream = new DataInputStream(unbufIn);

// 发送数据
new Sender(out).writeBlock(blockCopy, nodeStorageTypes[0], accessToken,
    dfsClient.clientName, nodes, nodeStorageTypes, null, bcs,
    nodes.length, block.getNumBytes(), bytesSent, newGS,
    checksum4WriteBlock, cachingStrategy.get(), isLazyPersistFile,
    (targetPinnings != null && targetPinnings[0]), targetPinnings,
    nodeStorageIDs[0], nodeStorageIDs);
...
}

public void writeBlock(...) throws IOException {
...
send(out, Op.WRITE_BLOCK, proto.build());
}

```

3.1.4 建立管道之 Socket 接收

Ctrl +n 全局查找 DataXceiverServer.java，在该类中查找 run 方法

```

public void run() {
    Peer peer = null;
    while (datanode.shouldRun && !datanode.shutdownForUpgrade) {
        try {
            // 接收 socket 的请求
            peer = peerServer.accept();

            // Make sure the xceiver count is not exceeded
            int curXceiverCount = datanode.getXceiverCount();
            if (curXceiverCount > maxXceiverCount) {
                throw new IOException("Xceiver count " + curXceiverCount
                    + " exceeds the limit of concurrent xcivers: "
                    + maxXceiverCount);
            }
            // 客户端每发送一个 block，都启动一个 DataXceiver 去处理 block
            new Daemon(datanode.threadGroup,
                DataXceiver.create(peer, datanode, this))
                    .start();
        } catch (SocketTimeoutException ignored) {
            ...
        }
    }
}

```

点击 DataXceiver (线程), 查找 run 方法

```
public void run() {
    int opsProcessed = 0;
    Op op = null;

    try {
        synchronized(this) {
            xceiver = Thread.currentThread();
        }
        dataXceiverServer.addPeer(peer, Thread.currentThread(), this);
        peer.setWriteTimeout(datanode.getDnConf().socketWriteTimeout);
        InputStream input = socketIn;
        try {
            IOStreamPair saslStreams = datanode.saslServer.receive(peer, socketOut,
                socketIn, datanode.getXferAddress().getPort(),
                return;
        }

        super.initialize(new DataInputStream(input));

        do {
            updateCurrentThreadName("Waiting for operation #" + (opsProcessed + 1));

            try {
                if (opsProcessed != 0) {
                    assert dnConf.socketKeepaliveTimeout > 0;
                    peer.setReadTimeout(dnConf.socketKeepaliveTimeout);
                } else {
                    peer.setReadTimeout(dnConf.socketTimeout);
                }
                // 读取这次数据的请求类型
                op = readOp();
            } catch (InterruptedException ignored) {
                // Time out while we wait for client rpc
                break;
            } catch (EOFException | ClosedChannelException e) {
                // Since we optimistically expect the next op, it's quite normal to
                // get EOF here.
                LOG.debug("Cached {} closing after {} ops. " +
                    "This message is usually benign.", peer, opsProcessed);
                break;
            } catch (IOException err) {
                incrDatanodeNetworkErrors();
                throw err;
            }

            // restore normal timeout
            if (opsProcessed != 0) {
                peer.setReadTimeout(dnConf.socketTimeout);
            }

            opStartTime = monotonicNow();
            // 根据操作类型处理我们的数据
            processOp(op);
        }
    }
}
```

```

        ++opsProcessed;
    } while ((peer != null) &&
        (!peer.isClosed() && dnConf.socketKeepaliveTimeout > 0));
    } catch (Throwable t) {
        ...
    }
}

protected final void processOp(Op op) throws IOException {
    switch(op) {
        ...
        case WRITE_BLOCK:
            opWriteBlock(in);
            break;
        ...
        default:
            throw new IOException("Unknown op " + op + " in data stream");
    }
}

private void opWriteBlock(DataInputStream in) throws IOException {
    final OpWriteBlockProto proto = OpWriteBlockProto.parseFrom(vintPrefixed(in));
    final DatanodeInfo[] targets = PBHelperClient.convert(proto.getTargetsList());
    TraceScope traceScope = continueTraceSpan(proto.getHeader(),
        proto.getClass().getSimpleName());
    try {
        writeBlock(PBHelperClient.convert(proto.getHeader().getBaseHeader().getBlock()),
            PBHelperClient.convertStorageType(proto.getStorageType()),
            PBHelperClient.convert(proto.getHeader().getBaseHeader().getToken()),
            proto.getHeader().getClientName(),
            targets,
            PBHelperClient.convertStorageTypes(proto.getTargetStorageTypesList(),
                targets.length),
            PBHelperClient.convert(proto.getSource()),
            fromProto(proto.getStage()),
            proto.getPipelineSize(),
            proto.getMinBytesRcvd(), proto.getMaxBytesRcvd(),
            proto.getLatestGenerationStamp(),
            fromProto(proto.getRequestedChecksum()),
            (proto.hasCachingStrategy() ?
                getCacheStrategy(proto.getCacheStrategy()) :
                CacheStrategy.newDefaultStrategy()),
            (proto.hasAllowLazyPersist() ? proto.getAllowLazyPersist() : false),
            (proto.hasPinning() ? proto.getPinning() : false),
            (PBHelperClient.convertBooleanList(proto.getTargetPinningsList())),
            proto.getStorageId(),
            proto.getTargetStorageIdsList().toArray(new String[0]));
    } finally {
        if (traceScope != null) traceScope.close();
    }
}

```

Ctrl +alt +b 查找 `writeBlock` 的实现类 DataXceiver.java

```

public void writeBlock(...) throws IOException {
    ...
    try {
        final Replica replica;

```

```
if (isDatanode ||  
    stage != BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {  
    // open a block receiver  
    // 创建一个 BlockReceiver  
    setCurrentBlockReceiver(getBlockReceiver(block, storageType, in,  
        peer.getRemoteAddressString(),  
        peer.getLocalAddressString(),  
        stage, latestGenerationStamp, minBytesRcvd, maxBytesRcvd,  
        clientname, srcDataNode, datanode, requestedChecksum,  
        cachingStrategy, allowLazyPersist, pinning, storageId));  
    replica = blockReceiver.getReplica();  
} else {  
    replica = datanode.data.recoverClose(  
        block, latestGenerationStamp, minBytesRcvd);  
}  
storageUuid = replica.getStorageUuid();  
isOnTransientStorage = replica.isOnTransientStorage();  
  
//  
// Connect to downstream machine, if appropriate  
// 继续连接下游的机器  
if (targets.length > 0) {  
    InetSocketAddress mirrorTarget = null;  
    // Connect to backup machine  
    mirrorNode = targets[0].getXferAddr(connectToDnViaHostname);  
    LOG.debug("Connecting to datanode {}", mirrorNode);  
    mirrorTarget = NetUtils.createSocketAddr(mirrorNode);  
  
    // 向新的副本发送 socket  
    mirrorSock = datanode.newSocket();  
    try {  
  
        ... ...  
        if (targetPinnings != null && targetPinnings.length > 0) {  
            // 往下游 socket 发送数据  
            new Sender(mirrorOut).writeBlock(originalBlock, targetStorageTypes[0],  
                blockToken, clientname, targets, targetStorageTypes,  
                srcDataNode, stage, pipelineSize, minBytesRcvd, maxBytesRcvd,  
                latestGenerationStamp, requestedChecksum, cachingStrategy,  
                allowLazyPersist, targetPinnings[0], targetPinnings,  
                targetStorageId, targetStorageIds);  
        } else {  
            new Sender(mirrorOut).writeBlock(originalBlock, targetStorageTypes[0],  
                blockToken, clientname, targets, targetStorageTypes,  
                srcDataNode, stage, pipelineSize, minBytesRcvd, maxBytesRcvd,  
                latestGenerationStamp, requestedChecksum, cachingStrategy,  
                allowLazyPersist, false, targetPinnings,  
                targetStorageId, targetStorageIds);  
        }  
  
        mirrorOut.flush();  
  
        DataNodeFaultInjector.get().writeBlockAfterFlush();  
  
        // read connect ack (only for clients, not for replication req)  
        if (isClient) {  
    }
```

```
BlockOpResponseProto connectAck =  
    BlockOpResponseProto.parseFrom(PBHelperClient.vintPrefixed(mirrorIn));  
mirrorInStatus = connectAck.getStatus();  
firstBadLink = connectAck.getFirstBadLink();  
if (mirrorInStatus != SUCCESS) {  
    LOG.debug("Datanode {} got response for connect" +  
        "ack from downstream datanode with firstbadlink as {}",  
        targets.length, firstBadLink);  
}  
...  
  
//update metrics  
datanode.getMetrics().addWriteBlockOp(elapsed());  
datanode.getMetrics().incrWritesFromClient(peer.isLocal(), size);  
}  
  
BlockReceiver getBlockReceiver(  
    final ExtendedBlock block, final StorageType storageType,  
    final DataInputStream in,  
    final String inAddr, final String myAddr,  
    final BlockConstructionStage stage,  
    final long newGs, final long minBytesRcvd, final long maxBytesRcvd,  
    final String clientname, final DatanodeInfo srcDataNode,  
    final DataNode dn, DataChecksum requestedChecksum,  
    CachingStrategy cachingStrategy,  
    final boolean allowLazyPersist,  
    final boolean pinning,  
    final String storageId) throws IOException {  
return new BlockReceiver(block, storageType, in,  
    inAddr, myAddr, stage, newGs, minBytesRcvd, maxBytesRcvd,  
    clientname, srcDataNode, dn, requestedChecksum,  
    cachingStrategy, allowLazyPersist, pinning, storageId);  
}  
  
BlockReceiver(final ExtendedBlock block, final StorageType storageType,  
    final DataInputStream in,  
    final String inAddr, final String myAddr,  
    final BlockConstructionStage stage,  
    final long newGs, final long minBytesRcvd, final long maxBytesRcvd,  
    final String clientname, final DatanodeInfo srcDataNode,  
    final DataNode datanode, DataChecksum requestedChecksum,  
    CachingStrategy cachingStrategy,  
    final boolean allowLazyPersist,  
    final boolean pinning,  
    final String storageId) throws IOException {  
...  
if (isDatanode) { //replication or move  
    replicaHandler =  
        datanode.data.createTemporary(storageType, storageId, block, false);  
} else {  
    switch (stage) {  
    case PIPELINE_SETUP_CREATE:  
        // 创建管道  
        replicaHandler = datanode.data.createRbw(storageType, storageId,  
            block, allowLazyPersist);  
    }  
}
```

```
    datanode.notifyNamenodeReceivingBlock(
        block, replicaHandler.getReplica().getStorageUuid());
    break;
    ...
    default: throw new IOException("Unsupported stage " + stage +
        " while receiving block " + block + " from " + inAddr);
}
}
...
}

public ReplicaHandler createRbw(
    StorageType storageType, String storageId, ExtendedBlock b,
    boolean allowLazyPersist) throws IOException {
try (AutoCloseableLock lock = datasetLock.acquire()) {
    ...
    if (ref == null) {
        ref = volumes.getNextVolume(storageType, storageId, b.getNumBytes());
    }
    FsVolumeImpl v = (FsVolumeImpl) ref.getVolume();
    // create an rbw file to hold block in the designated volume

    if (allowLazyPersist && !v.isTransientStorage()) {
        datanode.getMetrics().incrRamDiskBlocksWriteFallback();
    }

    ReplicaInPipeline newReplicaInfo;
    try {
        // 创建输出流的临时写文件
        newReplicaInfo = v.createRbw(b);
        if (newReplicaInfo.getReplicaInfo().getState() != ReplicaState.RBW) {
            throw new IOException("CreateRBW returned a replica of state "
                + newReplicaInfo.getReplicaInfo().getState()
                + " for block " + b.getBlockId());
        }
    } catch (IOException e) {
        IOUtils.cleanup(null, ref);
        throw e;
    }
    volumeMap.add(b.getBlockPoolId(), newReplicaInfo.getReplicaInfo());
    return new ReplicaHandler(newReplicaInfo, ref);
}
}

public ReplicaHandler createRbw(
    StorageType storageType, String storageId, ExtendedBlock b,
    boolean allowLazyPersist) throws IOException {
try (AutoCloseableLock lock = datasetLock.acquire()) {
    ...
    if (ref == null) {
        // 可能有多个临时写文件
        ref = volumes.getNextVolume(storageType, storageId, b.getNumBytes());
    }
}
```

```

    }

    FsVolumeImpl v = (FsVolumeImpl) ref.getVolume();
    // create an rbw file to hold block in the designated volume

    if (allowLazyPersist && !v.isTransientStorage()) {
        datanode.getMetrics().incrRamDiskBlocksWriteFallback();
    }

    ReplicaInPipeline newReplicaInfo;
    try {
        // 创建输出流的临时写文件
        newReplicaInfo = v.createRbw(b);
        if (newReplicaInfo.getReplicaInfo().getState() != ReplicaState.RBW) {
            throw new IOException("CreateRBW returned a replica of state "
                + newReplicaInfo.getReplicaInfo().getState()
                + " for block " + b.getBlockId());
        }
    } catch (IOException e) {
        IOUtils.cleanup(null, ref);
        throw e;
    }

    volumeMap.add(b.getBlockPoolId(), newReplicaInfo.getReplicaInfo());
    return new ReplicaHandler(newReplicaInfo, ref);
}
}

public ReplicaInPipeline createRbw(ExtendedBlock b) throws IOException {
    File f = createRbwFile(b.getBlockPoolId(), b.getLocalBlock());
    LocalReplicaInPipeline newReplicaInfo = new ReplicaBuilder(ReplicaState.RBW)
        .setBlockId(b.getBlockId())
        .setGenerationStamp(b.getGenerationStamp())
        .setFsVolume(this)
        .setDirectoryToUse(f.getParentFile())
        .setBytesToReserve(b.getNumBytes())
        .buildLocalReplicaInPipeline();
    return newReplicaInfo;
}
}

```

3.1.5 客户端接收 DN 写数据应答 Response

Ctrl + n 全局查找 DataStreamer，搜索 run 方法

DataStreamer.java

```

@Override
public void run() {

    long lastPacket = Time.monotonicNow();
    TraceScope scope = null;
    while (!streamerClosed && dfsClient.clientRunning) {
        // if the Responder encountered an error, shutdown Responder
        if (errorState.hasError()) {
            closeResponder();
        }
    }
}

```

```
DFSPacket one;
try {
    // process datanode IO errors if any
    boolean doSleep = processDatanodeOrExternalError();

    final int halfSocketTimeout = dfsClient.getConf().getSocketTimeout()/2;
    synchronized (dataQueue) {
        // wait for a packet to be sent.
        long now = Time.monotonicNow();
        while ((!shouldStop() && dataQueue.size() == 0 &&
               (stage != BlockConstructionStage.DATA_STREAMING ||
                now - lastPacket < halfSocketTimeout)) || doSleep) {
            long timeout = halfSocketTimeout - (now - lastPacket);
            timeout = timeout <= 0 ? 1000 : timeout;
            timeout = (stage == BlockConstructionStage.DATA_STREAMING)?
                      timeout : 1000;
            try {
                // 如果 dataQueue 里面没有数据，代码会阻塞在这儿
                dataQueue.wait(timeout); // 接收到 notify 消息
            } catch (InterruptedException e) {
                LOG.warn("Caught exception", e);
            }
            doSleep = false;
            now = Time.monotonicNow();
        }
        if (shouldStop()) {
            continue;
        }
        // get packet to be sent.
        if (dataQueue.isEmpty()) {
            one = createHeartbeatPacket();
        } else {
            try {
                backOffIfNecessary();
            } catch (InterruptedException e) {
                LOG.warn("Caught exception", e);
            }
            // 队列不为空，从队列中取出 packet
            one = dataQueue.getFirst(); // regular data packet
            SpanId[] parents = one.getTraceParents();
            if (parents.length > 0) {
                scope = dfsClient.getTracer().
                    newScope("dataStreamer", parents[0]);
                scope.getSpan().setParents(parents);
            }
        }
    }

    // get new block from namenode.
    if (LOG.isDebugEnabled()) {
        LOG.debug("stage=" + stage + ", " + this);
    }
    if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
        LOG.debug("Allocating new block: {}", this);
        // 步骤一：向 NameNode 申请 block 并建立数据管道
    }
}
```

```
        setPipeline(nextBlockOutputStream());
        // 步骤二：启动 ResponseProcessor 用来监听 packet 发送是否成功
        initDataStreaming();
    } else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
        LOG.debug("Append to block {}", block);
        setupPipelineForAppendOrRecovery();
        if (streamerClosed) {
            continue;
        }
        initDataStreaming();
    }

    long lastByteOffsetInBlock = one.getLastByteOffsetBlock();
    if (lastByteOffsetInBlock > stat.getBlockSize()) {
        throw new IOException("BlockSize " + stat.getBlockSize() +
            " < lastByteOffsetInBlock, " + this + ", " + one);
    }

    if (one.isLastPacketInBlock()) {
        // wait for all data packets have been successfully acked
        synchronized (dataQueue) {
            while (!shouldStop() && ackQueue.size() != 0) {
                try {
                    // wait for acks to arrive from datanodes
                    dataQueue.wait(1000);
                } catch (InterruptedException e) {
                    LOG.warn("Caught exception", e);
                }
            }
        }
        if (shouldStop()) {
            continue;
        }
        stage = BlockConstructionStage.PIPELINE_CLOSE;
    }

    // send the packet
    SpanId spanId = SpanId.INVALID;
    synchronized (dataQueue) {

        // move packet from dataQueue to ackQueue
        if (!one.isHeartbeatPacket()) {
            if (scope != null) {
                spanId = scope.getSpanId();
                scope.detach();

                one.setTraceScope(scope);
            }
            scope = null;
        }
        // 步骤三：从 dataQueue 把要发送的这个 packet 移除出去
        dataQueue.removeFirst();
        // 步骤四：然后往 ackQueue 里面添加这个 packet
        ackQueue.addLast(one);
        packetSendTime.put(one.getSeqno(), Time.monotonicNow());
        dataQueue.notifyAll();
    }
}
```

```
        }

        LOG.debug("{} sending {}", this, one);

        // write out data to remote datanode
        try (TraceScope ignored = dfsClient.getTracer().
            newScope("DataStreamer#writeTo", spanId)) {
            // 将数据写出去
            one.writeTo(blockStream);
            blockStream.flush();
        } catch (IOException e) {
            errorState.markFirstNodeIfNotMarked();
            throw e;
        }
        lastPacket = Time.monotonicNow();

        // update bytesSent
        long tmpBytesSent = one.getLastByteOffsetBlock();
        if (bytesSent < tmpBytesSent) {
            bytesSent = tmpBytesSent;
        }

        if (shouldStop()) {
            continue;
        }

        // Is this block full?
        if (one.isLastPacketInBlock()) {
            // wait for the close packet has been acked
            synchronized (dataQueue) {
                while (!shouldStop() && ackQueue.size() != 0) {
                    dataQueue.wait(1000); // wait for acks to arrive from datanodes
                }
            }
            if (shouldStop()) {
                continue;
            }

            endBlock();
        }
        if (progress != null) { progress.progress(); }

        // This is used by unit test to trigger race conditions.
        if (artificialSlowdown != 0 && dfsClient.clientRunning) {
            Thread.sleep(artificialSlowdown);
        }
    } catch (Throwable e) {
        ...
    } finally {
        if (scope != null) {
            scope.close();
            scope = null;
        }
    }
}

closeInternal();
}
```

```

private void initDataStreaming() {
    this.setName("DataStreamer for file " + src +
        " block " + block);
    ...
    response = new ResponseProcessor(nodes);
    response.start();
    stage = BlockConstructionStage.DATA_STREAMING;
}

```

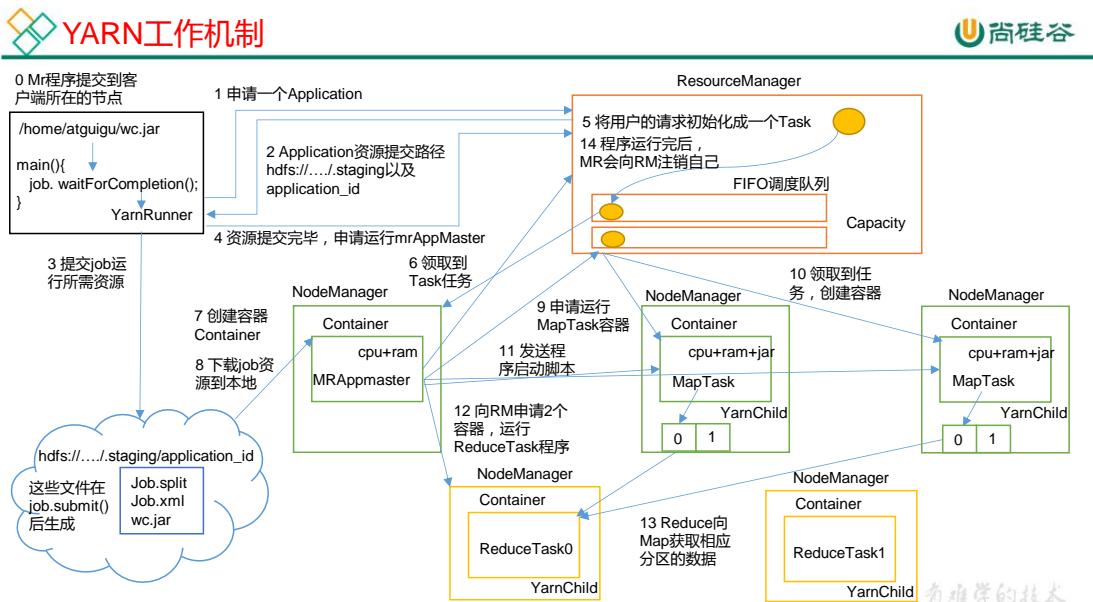
点击 response 再点击 ResponseProcessor, **ctrl + f** 查找 run 方法

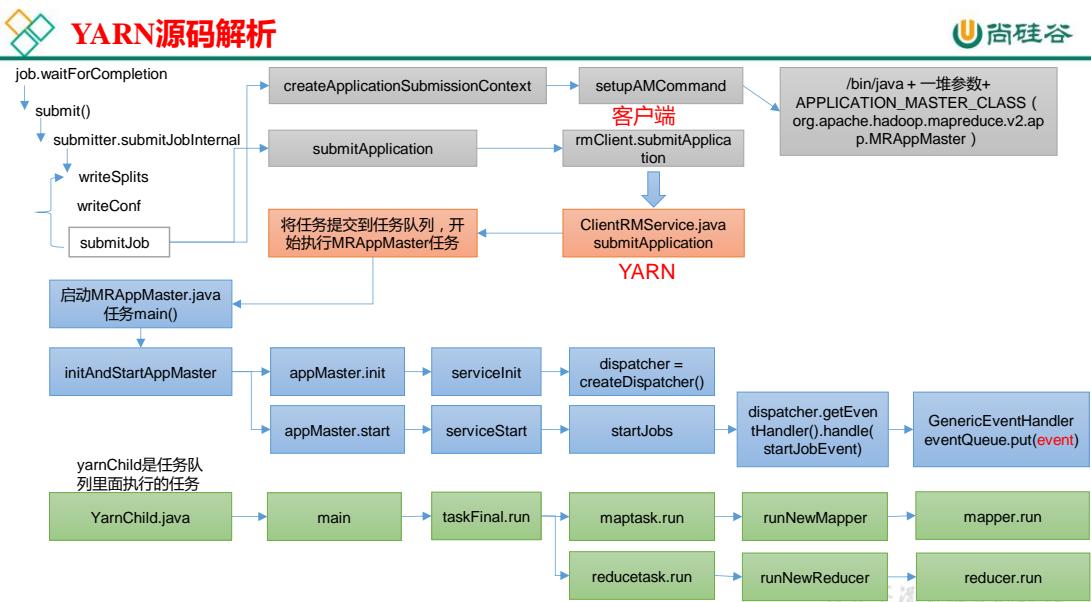
```

public void run() {
    ...
    ackQueue.removeFirst();
    packetSendTime.remove(seqno);
    dataQueue.notifyAll();
    ...
}

```

第 4 章 Yarn 源码解析





4.1 Yarn 客户端向 RM 提交作业

- 在 wordcount 程序的驱动类中点击

Job.java

```

boolean result = job.waitForCompletion(true);

public boolean waitForCompletion(boolean verbose
                                ) throws IOException, InterruptedException,
                                         ClassNotFoundException {
    if (state == JobState.DEFINE) {
        submit();
    }
    if (verbose) {
        monitorAndPrintJob();
    } else {
        // get the completion poll interval from the client.
        int completionPollIntervalMillis =
            Job.getCompletionPollInterval(cluster.getConf());
        while (!isComplete()) {
            try {
                Thread.sleep(completionPollIntervalMillis);
            } catch (InterruptedException ie) {
            }
        }
    }
    return.isSuccessful();
}

public void submit()
    throws IOException, InterruptedException, ClassNotFoundException {
    ensureState(JobState.DEFINE);
    setUseNewAPI();
    connect();
    final JobSubmitter submitter =
        getJobSubmitter(cluster.getFileSystem(), cluster.getClient());
    status = ugi.doAs(new PrivilegedExceptionAction<JobStatus>() {
        public JobStatus run() throws IOException, InterruptedException,
                                     ClassNotFoundException {
    
```

```

        return submitter.submitJobInternal(Job.this, cluster);
    }
});

state = JobState.RUNNING;
LOG.info("The url to track the job: " + getTrackingURL());
}

```

点击 submitJobInternal()

JobSubmitter.java

```

JobStatus submitJobInternal(Job job, Cluster cluster)
throws ClassNotFoundException, InterruptedException, IOException {
... ...
status = submitClient.submitJob(
    jobId, submitJobDir.toString(), job.getCredentials());
... ...
}

public JobStatus submitJob(JobID jobId, String jobSubmitDir, Credentials
ts) throws IOException, InterruptedException;

```

2) 创建提交环境

ctrl + alt +B 查找 submitJob 实现类, YARNRunner.java

```

public JobStatus submitJob(JobID jobId, String jobSubmitDir, Credentials
ts)
throws IOException, InterruptedException {

addHistoryToken(ts);
// 创建提交环境:
ApplicationSubmissionContext appContext =
    createApplicationSubmissionContext(conf, jobSubmitDir, ts);

// Submit to ResourceManager
try {
    // 向 RM 提交一个应用程序, appContext 里面封装了启动 mrappMaster 和运行 container
    // 的命令
    ApplicationId applicationId =
        resMgrDelegate.submitApplication(appContext);

    // 获取提交响应
    ApplicationReport appMaster = resMgrDelegate
        .getApplicationReport(applicationId);

    String diagnostics =
        (appMaster == null ?
            "application report is null" : appMaster.getDiagnostics());
    if (appMaster == null
        || appMaster.getYarnApplicationState() ==
YarnApplicationState.FAILED
        || appMaster.getYarnApplicationState() ==
YarnApplicationState.KILLED) {
        throw new IOException("Failed to run job : " +
            diagnostics);
    }
    return clientCache.getClient(jobId).getJobStatus(jobId);
} catch (YarnException e) {
    throw new IOException(e);
}
}

```

```
public ApplicationSubmissionContext createApplicationSubmissionContext(
    Configuration jobConf, String jobSubmitDir, Credentials ts)
    throws IOException {
    ApplicationId applicationId = resMgrDelegate.getApplicationId();

    // Setup LocalResources
    // 封装了本地资源相关路径
    Map<String, LocalResource> localResources =
        setupLocalResources(jobConf, jobSubmitDir);

    // Setup security tokens
    DataOutputBuffer dob = new DataOutputBuffer();
    ts.writeTokenStorageToStream(dob);
    ByteBuffer securityTokens =
        ByteBuffer.wrap(dob.getData(), 0, dob.getLength());

    // Setup ContainerLaunchContext for AM container
    // 封装了启动 mrappMaster 和运行 container 的命令
    List<String> vargs = setupAMCommand(jobConf);
    ContainerLaunchContext amContainer = setupContainerLaunchContextForAM(
        jobConf, localResources, securityTokens, vargs);

    ...
    ...

    return appContext;
}

private List<String> setupAMCommand(Configuration jobConf) {
    List<String> vargs = new ArrayList<>(8);
    // Java 进程启动命令开始
    vargs.add(MRApps.crossPlatformifyMREnv(jobConf, Environment.JAVA_HOME)
        + "/bin/java");

    Path amTmpDir =
        new Path(MRApps.crossPlatformifyMREnv(conf, Environment.PWD),
            YarnConfiguration.DEFAULT_CONTAINER_TEMP_DIR);
    vargs.add("-Djava.io.tmpdir=" + amTmpDir);
    MRApps.addLog4jSystemProperties(null, vargs, conf);

    // Check for Java Lib Path usage in MAP and REDUCE configs
    warnForJavaLibPath(conf.get(MRJobConfig.MAP_JAVA_OPTS, ""),
        "map",
        MRJobConfig.MAP_JAVA_OPTS,
        MRJobConfig.MAP_ENV);
    warnForJavaLibPath(conf.get(MRJobConfig.MAPRED_MAP_ADMIN_JAVA_OPTS, ""),
        "map",
        MRJobConfig.MAPRED_MAP_ADMIN_JAVA_OPTS,
        MRJobConfig.MAPRED_ADMIN_USER_ENV);
    warnForJavaLibPath(conf.get(MRJobConfig.REDUCE_JAVA_OPTS, ""),
        "reduce",
        MRJobConfig.REDUCE_JAVA_OPTS,
        MRJobConfig.REDUCE_ENV);
    warnForJavaLibPath(conf.get(MRJobConfig.MAPRED_REDUCE_ADMIN_JAVA_OPTS,
        ""),
        "reduce",
        MRJobConfig.MAPRED_REDUCE_ADMIN_JAVA_OPTS,
        MRJobConfig.MAPRED_ADMIN_USER_ENV);

    // Add AM admin command opts before user command opts
    // so that it can be overridden by user
```

```

        String mrAppMasterAdminOptions = conf.get(MRJobConfig.MR_AM_ADMIN_COMMAND_OPTS,
            MRJobConfig.DEFAULT_MR_AM_ADMIN_COMMAND_OPTS);
        warnForJavaLibPath(mrAppMasterAdminOptions, "app master",
            MRJobConfig.MR_AM_ADMIN_COMMAND_OPTS,
            MRJobConfig.MR_AM_ADMIN_USER_ENV);
        vargs.add(mrAppMasterAdminOptions);

        // Add AM user command opts 用户命令参数
        String mrAppMasterUserOptions = conf.get(MRJobConfig.MR_AM_COMMAND_OPTS,
            MRJobConfig.DEFAULT_MR_AM_COMMAND_OPTS);
        warnForJavaLibPath(mrAppMasterUserOptions, "app master",
            MRJobConfig.MR_AM_COMMAND_OPTS, MRJobConfig.MR_AM_ENV);
        vargs.add(mrAppMasterUserOptions);

        if (jobConf.getBoolean(MRJobConfig.MR_AM_PROFILE,
            MRJobConfig.DEFAULT_MR_AM_PROFILE)) {
            final String profileParams = jobConf.get(MRJobConfig.MR_AM_PROFILE_PARAMS,
                MRJobConfig.DEFAULT_TASK_PROFILE_PARAMS);
            if (profileParams != null) {
                vargs.add(String.format(profileParams,
                    ApplicationConstants.LOG_DIR_EXPANSION_VAR + Path.SEPARATOR
                    + TaskLog.LogName.PROFILE));
            }
        }

        // 封装了要启动的 mrappmaster 全类名
        // org.apache.hadoop.mapreduce.v2.app.MRApplMaster
        vargs.add(MRJobConfig.APPLICATION_MASTER_CLASS);
        vargs.add("1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR +
            Path.SEPARATOR + ApplicationConstants.STDOUT);
        vargs.add("2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR +
            Path.SEPARATOR + ApplicationConstants.STDERR);
        return vargs;
    }
}

```

3) 向 Yarn 提交

点击 submitJob 方法中的 `submitApplication()`

YARNRunner.java

```

ApplicationId applicationId =
resMgrDelegate.submitApplication(appContext);

public ApplicationId
submitApplication(ApplicationSubmissionContext appContext)
    throws YarnException, IOException {
    return client.submitApplication(appContext);
}

```

`ctrl + alt +B` 查找 `submitApplication` 实现类, `YarnClientImpl.java`

```

public ApplicationId
submitApplication(ApplicationSubmissionContext appContext)
    throws YarnException, IOException {
    ApplicationId applicationId = appContext.getApplicationId();
    if (applicationId == null) {
        throw new ApplicationIdNotProvidedException(
            "ApplicationId is not provided in ApplicationSubmissionContext");
    }
    // 创建一个提交请求
}

```

```

SubmitApplicationRequest request =
    Records.newRecord(SubmitApplicationRequest.class);
request.setApplicationSubmissionContext(appContext);
...
...

//TODO: YARN-1763:Handle RM failovers during the submitApplication call.
// 继续提交, 实现类是 ApplicationClientProtocolPBClientImpl
rmClient.submitApplication(request);

int pollCount = 0;
long startTime = System.currentTimeMillis();
EnumSet<YarnApplicationState> waitingStates =
    EnumSet.of(YarnApplicationState.NEW,
               YarnApplicationState.NEW_SAVING,
               YarnApplicationState.SUBMITTED);
EnumSet<YarnApplicationState> failToSubmitStates =
    EnumSet.of(YarnApplicationState.FAILED,
               YarnApplicationState.KILLED);
while (true) {
    try {
        // 获取提交给 Yarn 的反馈
        ApplicationReport appReport = getApplicationReport(applicationId);
        YarnApplicationState state = appReport.getYarnApplicationState();
        ...
    } catch (ApplicationNotFoundException ex) {
        // FailOver or RM restart happens before RMStateStore saves
        // ApplicationState
        LOG.info("Re-submit application " + applicationId + "with the " +
            "same ApplicationSubmissionContext");
        // 如果提交失败, 则再次提交
        rmClient.submitApplication(request);
    }
}

return applicationId;
}

```

ctrl + alt +B 查找 submitApplication 实现类, ClientRMService.java

```

public SubmitApplicationResponse submitApplication(
    SubmitApplicationRequest request) throws YarnException, IOException {
    ApplicationSubmissionContext submissionContext = request
        .getApplicationSubmissionContext();
    ApplicationId applicationId = submissionContext.getApplicationId();
    CallerContext callerContext = CallerContext.getCurrent();

    ...
    try {
        // call RMAppManager to submit application directly
        rmAppManager.submitApplication(submissionContext,
            System.currentTimeMillis(), user);

        LOG.info("Application with id " + applicationId.getId() +
            " submitted by user " + user);
        RMAuditLogger.logSuccess(user, AuditConstants.SUBMIT_APP_REQUEST,
            "ClientRMService", applicationId, callerContext,
            submissionContext.getQueue());
    } catch (YarnException e) {
        LOG.info("Exception in submitting " + applicationId, e);
        RMAuditLogger.logFailure(user, AuditConstants.SUBMIT_APP_REQUEST,
            e.getMessage(), "ClientRMService",
            "Exception in submitting application", applicationId, callerContext,

```

```

        submissionContext.getQueue());
    throw e;
}

return recordFactory
    .newRecordInstance(SubmitApplicationResponse.class);
}

```

4.2 RM 启动 MRAppMaster

- 0) 在 pom.xml 中增加如下依赖

```

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-app</artifactId>
    <version>3.1.3</version>
</dependency>

```

ctrl +n 查找 MRAppMaster，搜索 main 方法

```

public static void main(String[] args) {
    try {
        ...
        ...
        // 初始化一个 container
        ContainerId containerId = ContainerId.fromString(containerIdStr);
        ApplicationAttemptId applicationAttemptId =
            containerId.getApplicationAttemptId();
        if (applicationAttemptId != null) {
            CallerContext.setCurrent(new CallerContext.Builder(
                "mr_appmaster_" + applicationAttemptId.toString()).build());
        }
        long appSubmitTime = Long.parseLong(appSubmitTimeStr);

        // 创建 appMaster 对象
        MRAppMaster appMaster =
            new MRAppMaster(applicationAttemptId, containerId, nodeHostString,
                Integer.parseInt(nodePortString),
                Integer.parseInt(nodeHttpPortString), appSubmitTime);
        ...
        ...
        // 初始化并启动 AppMaster
        initAndStartAppMaster(appMaster, conf, jobUserName);
    } catch (Throwable t) {
        LOG.error("Error starting MRAppMaster", t);
        ExitUtil.terminate(1, t);
    }
}

protected static void initAndStartAppMaster(final MRAppMaster appMaster,
    final JobConf conf, String jobUserName) throws IOException,
    InterruptedException {
    ...
    conf.getCredentials().addAll(credentials);
    appMasterUgi.doAs(new PrivilegedExceptionAction<Object>() {
        @Override
        public Object run() throws Exception {
            // 初始化

```

```

        appMaster.init(conf);
        // 启动
        appMaster.start();
        if(appMaster.errorHappenedShutDown) {
            throw new IOException("Was asked to shut down.");
        }
        return null;
    }
}

public void init(Configuration conf) {
    ...
    synchronized (stateChangeLock) {
        if (enterState(STATE.INITED) != STATE.INITED) {
            setConfig(conf);
            try {
                // 调用 MRAppMaster 中的 serviceInit()方法
                serviceInit(config);
                if (isInState(STATE.INITED)) {
                    //if the service ended up here during init,
                    //notify the listeners
                    // 如果初始化完成，通知监听器
                    notifyListeners();
                }
            } catch (Exception e) {
                noteFailure(e);
                ServiceOperations.stopQuietly(LOG, this);
                throw ServiceStateException.convert(e);
            }
        }
    }
}

```

ctrl + alt +B 查找 serviceInit 实现类， MRAppMaster.java

```

protected void serviceInit(final Configuration conf) throws Exception {
    ...
    // 创建提交路径
    clientService = createClientService(context);

    // 创建调度器
    clientService.init(conf);

    // 创建 job 提交 RPC 客户端
    containerAllocator = createContainerAllocator(clientService, context);
    ...
}

```

点击 MRAppMaster.java 中的 initAndStartAppMaster 方法中的 appMaster.start();

```

public void start() {
    if (isInState(STATE.STARTED)) {
        return;
    }
    //enter the started state
    synchronized (stateChangeLock) {
        if (stateModel.enterState(STATE.STARTED) != STATE.STARTED) {

```

```

try {
    startTime = System.currentTimeMillis();
    // 调用 MRAppMaster 中的 serviceStart()方法
    serviceStart();
    if (isInState(STATE.STARTED)) {
        //if the service started (and isn't now in a later state), notify
        LOG.debug("Service {} is started", getName());
        notifyListeners();
    }
} catch (Exception e) {
    noteFailure(e);
    ServiceOperations.stopQuietly(LOG, this);
    throw ServiceStateException.convert(e);
}
}

protected void serviceStart() throws Exception {
    ...
    if (initFailed) {
        JobEvent initFailedEvent = new JobEvent(job.getID(),
JobEventType.JOB_INIT_FAILED);
        jobEventDispatcher.handle(initFailedEvent);
    } else {
        // All components have started, start the job.
        // 初始化成功后，提交 Job 到队列中
        startJobs();
    }
}

protected void startJobs() {
    /** create a job-start event to get this ball rolling */
    JobEvent startJobEvent = new JobStartEvent(job.getID(),
        recoveredJobStartTime);
    /** send the job-start event. this triggers the job execution. */
    // 这里将 job 存放到 yarn 队列
    // dispatcher = AsyncDispatcher
    // getEventHandler()返回的是 GenericEventHandler
    dispatcher.getEventHandler().handle(startJobEvent);
}

```

ctrl + alt +B 查找 handle 实现类， GenericEventHandler.java

```

class GenericEventHandler implements EventHandler<Event> {
    public void handle(Event event) {
        ...
        try {
            // 将 job 存储到 yarn 队列中
            eventQueue.put(event);
        } catch (InterruptedException e) {
            ...
        }
    };
}

```

4.3 调度器任务执行 (YarnChild)

1) 启动 MapTask

ctrl +n 查找 YarnChild, 搜索 main 方法

```
public static void main(String[] args) throws Throwable {
    Thread.setDefaultUncaughtExceptionHandler(new YarnUncaughtExceptionHandler());
    LOG.debug("Child starting");

    ...
    ...

    task = myTask.getTask();
    YarnChild.taskid = task.getTaskID();
    ...

    // Create a final reference to the task for the doAs block
    final Task taskFinal = task;
    childUGI.doAs(new PrivilegedExceptionAction<Object>() {
        @Override
        public Object run() throws Exception {
            // use job-specified working directory
            setEncryptedSpillKeyIfRequired(taskFinal);
            FileSystem.get(job).setWorkingDirectory(job.getWorkingDirectory());
            // 调用 task 执行 (maptask 或者 reducetask)
            taskFinal.run(job, umbilical); // run the task
            return null;
        }
    });
}
...
}
```

ctrl + alt +B 查找 run 实现类, maptask.java

```
public void run(final JobConf job, final TaskUmbilicalProtocol umbilical)
    throws IOException, ClassNotFoundException, InterruptedException {
    this.umbilical = umbilical;

    // 判断是否是 MapTask
    if (isMapTask()) {
        // If there are no reducers then there won't be any sort. Hence the map
        // phase will govern the entire attempt's progress.
        // 如果 reducetask 个数为零, maptask 占用整个任务的 100%
        if (conf.getNumReduceTasks() == 0) {
            mapPhase = getProgress().addPhase("map", 1.0f);
        } else {
            // If there are reducers then the entire attempt's progress will be
            // split between the map phase (67%) and the sort phase (33%).
            // 如果 reduceTask 个数不为零, MapTask 占用整个任务的 66.7% sort 阶段占比
            mapPhase = getProgress().addPhase("map", 0.667f);
            sortPhase = getProgress().addPhase("sort", 0.333f);
        }
    }
    ...
    if (useNewApi) {
        // 调用新的 API 执行 maptask
    }
}
```

```

        runNewMapper(job, splitMetaInfo, umbilical, reporter);
    } else {
        runOldMapper(job, splitMetaInfo, umbilical, reporter);
    }
    done(umbilical, reporter);
}

void runNewMapper(final JobConf job,
                  final TaskSplitIndex splitIndex,
                  final TaskUmbilicalProtocol umbilical,
                  TaskReporter reporter
) throws IOException, ClassNotFoundException,
       InterruptedException {
    ...
}

try {
    input.initialize(split, mapperContext);
    // 运行 maptask
    mapper.run(mapperContext);

    mapPhase.complete();
    setPhase(TaskStatus.Phase.SORT);
    statusUpdate(umbilical);
    input.close();
    input = null;
    output.close(mapperContext);
    output = null;
} finally {
    closeQuietly(input);
    closeQuietly(output, mapperContext);
}
}

```

Mapper.java (和 Map 联系在一起)

```

public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    try {
        while (context.nextKeyValue()) {
            map(context.getCurrentKey(), context.getCurrentValue(), context);
        }
    } finally {
        cleanup(context);
    }
}

```

2) 启动 ReduceTask

在 YarnChild.java 类中的 main 方法中 ctrl + alt +B 查找 run 实现类， reducetask.java

```

public void run(JobConf job, final TaskUmbilicalProtocol umbilical)
    throws IOException, InterruptedException, ClassNotFoundException {
    job.setBoolean(JobContext.SKIP_RECORDS, isSkipping());
    ...
}

if (useNewApi) {
    // 调用新 API 执行 reduce
    runNewReducer(job, umbilical, reporter, rIter, comparator,
}

```

```

        keyClass, valueClass);
    } else {
        runOldReducer(job, umbilical, reporter, rIter, comparator,
                      keyClass, valueClass);
    }

    shuffleConsumerPlugin.close();
    done(umbilical, reporter);
}

void runNewReducer(JobConf job,
                    final TaskUmbilicalProtocol umbilical,
                    final TaskReporter reporter,
                    RawKeyValueIterator rIter,
                    RawComparator<INKEY> comparator,
                    Class<INKEY> keyClass,
                    Class<INVALUE> valueClass
                    ) throws IOException, InterruptedException,
                           ClassNotFoundException {
    ...
    try {
        // 调用 reducetask 的 run 方法
        reducer.run(reducerContext);
    } finally {
        trackedRW.close(reducerContext);
    }
}

```

Reduce.java

```

public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    try {
        while (context.nextKey()) {
            reduce(context.getCurrentKey(), context.getValues(), context);
            // If a back up store is used, reset it
            Iterator<VALUEIN> iter = context.getValues().iterator();
            if(iter instanceof ReduceContext.ValueIterator) {
                ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore();
            }
        }
    } finally {
        cleanup(context);
    }
}

```

第 5 章 MapReduce 源码解析

说明：在讲 MapReduce 课程时，已经讲过源码，在这就不再赘述。

5.1 Job 提交流程源码和切片源码详解

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) 判断是本地运行环境还是 yarn 集群运行环境
        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();

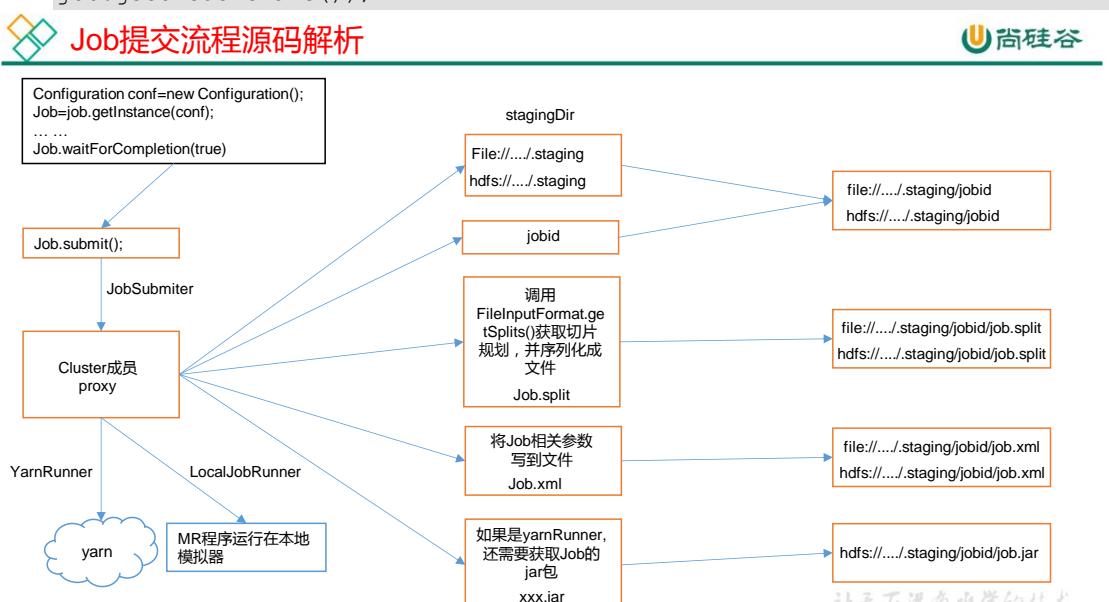
// 3) 拷贝 jar 包到集群
copyAndConfigureFiles(job, submitJobDir);
rUploader.uploadFiles(job, jobSubmitDir);

// 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 切片源码解析 (input.getSplits(job))

 FileInputFormat切片源码解析

- (1) 程序先找到你数据存储的目录。
- (2) 开始遍历处理(规划切片)目录下的每一个文件
- (3) 遍历第一个文件ss.txt
 - a) 获取文件大小fs.sizeOf(ss.txt)
 - b) 计算切片大小
computeSplitSize(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) InputSplit只记录了切片的元数据信息,比如起始位置、长度以及所在的节点列表等。
- (4) 提交切片规划文件到YARN上,YARN上的MrAppMaster就可以根据切片规划文件计算开启MapTask个数。

让天下没有难学的技术

5.2 MapTask & ReduceTask 源码解析

1) MapTask 源码解析流程

```
===== MapTask =====
context.write(k, NullWritable.get()); //自定义的 map 方法的写出, 进入
output.write(key, value);
//MapTask727 行, 收集方法, 进入两次
collector.collect(key, value, partitioner.getPartition(key, value, partitions));
HashPartitioner(); //默认分区器
collect() //MapTask1082 行 map 端所有的 kv 全部写出后会走下面的 close 方法
close() //MapTask732 行
collector.flush() // 溢出刷写方法, MapTask735 行, 提前打个断点, 进入
sortAndSpill() //溢写排序, MapTask1505 行, 进入
sorter.sort() QuickSort //溢写排序方法, MapTask1625 行, 进入
mergeParts(); //合并文件, MapTask1527 行, 进入
□ file.out
□ file.out.index
collector.close(); //MapTask739 行, 收集器关闭, 即将进入 ReduceTask
```

2) ReduceTask 源码解析流程

```
===== ReduceTask =====
if (isMapOrReduce()) //reduceTask324 行, 提前打断点
initialize() // reduceTask333 行, 进入
init(shuffleContext); // reduceTask375 行, 走到这需要先给下面的打断点
totalMaps = job.getNumMapTasks(); // ShuffleSchedulerImpl 第 120 行, 提前打断点
merger = createMergeManager(context); //合并方法, Shuffle 第 80 行
// MergeManagerImpl 第 232 235 行, 提前打断点
this.inMemoryMerger = createInMemoryMerger(); //内存合并
```

```

this.onDiskMerger = new OnDiskMerger(this); //磁盘合并
rlter = shuffleConsumerPlugin.run();
eventFetcher.start(); //开始抓取数据, Shuffle 第 107 行, 提前打断点
eventFetcher.shutDown(); //抓取结束, Shuffle 第 141 行, 提前打断点
copyPhase.complete(); //copy 阶段完成, Shuffle 第 151 行
taskStatus.setPhase(TaskStatus.Phase.SORT); //开始排序阶段, Shuffle 第 152 行
sortPhase.complete(); //排序阶段完成, 即将进入 reduce 阶段 reduceTask382 行
reduce(); //reduce 阶段调用的就是我们自定义的 reduce 方法, 会被调用多次
cleanup(context); //reduce 完成之前, 会最后调用一次 Reducer 里面的 cleanup 方法

```

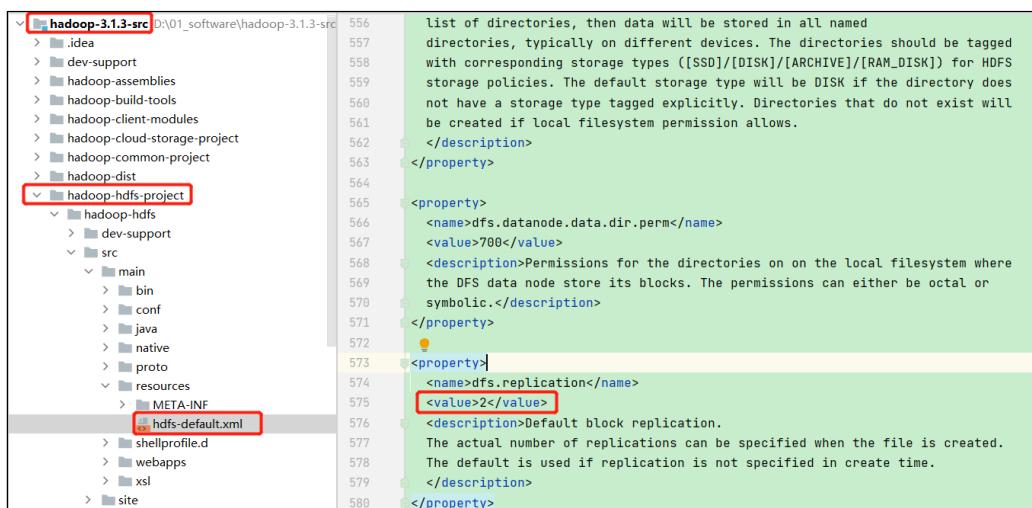
第 6 章 Hadoop 源码编译

6.1 前期准备工作

1) 官网下载源码

<https://hadoop.apache.org/release/3.1.3.html>

2) 修改源码中的 HDFS 副本数的设置



3) CentOS 虚拟机准备

(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）

- hadoop-3.1.3-src.tar.gz
- jdk-8u212-linux-x64.tar.gz
- apache-maven-3.6.3-bin.tar.gz
- protobuf-2.5.0.tar.gz (序列化的框架)

➤ cmake-3.17.0.tar.gz

6.2 工具包安装

注意：所有操作必须在 root 用户下完成

0) 分别创建 /opt/software/hadoop_source 和 /opt/module/hadoop_source 路径

1) 上传软件包到指定的目录，例如 /opt/software/hadoop_source

```
[root@hadoop101 hadoop_source]$ pwd
/opt/software/hadoop_source
[root@hadoop101 hadoop_source]$ ll
总用量 55868
-rw-rw-r--. 1 atguigu atguigu 9506321 3月 28 13:23 apache-maven-3.6.3-bin.tar.gz
-rw-rw-r--. 1 atguigu atguigu 8614663 3月 28 13:23 cmake-3.17.0.tar.gz
-rw-rw-r--. 1 atguigu atguigu 29800905 3月 28 13:23 hadoop-3.1.3-src.tar.gz
-rw-rw-r--. 1 atguigu atguigu 2401901 3月 28 13:23 protobuf-2.5.0.tar.gz
```

2) 解压软件包指定的目录，例如：/opt/module/hadoop_source

```
[root@hadoop101 hadoop_source]$ tar -zxvf apache-maven-3.6.3-bin.tar.gz -C /opt/module/hadoop_source/
[root@hadoop101 hadoop_source]$ tar -zxvf cmake-3.17.0.tar.gz -C /opt/module/hadoop_source/
[root@hadoop101 hadoop_source]$ tar -zxvf hadoop-3.1.3-src.tar.gz -C /opt/module/hadoop_source/
[root@hadoop101 hadoop_source]$ tar -zxvf protobuf-2.5.0.tar.gz -C /opt/module/hadoop_source/
[root@hadoop101 hadoop_source]$ pwd
/opt/module/hadoop_source
[root@hadoop101 hadoop_source]$ ll
总用量 20
drwxrwxr-x. 6 atguigu atguigu 4096 3月 28 13:25 apache-maven-3.6.3
drwxr-xr-x. 15 root root 4096 3月 28 13:43 cmake-3.17.0
drwxr-xr-x. 18 atguigu atguigu 4096 9月 12 2019 hadoop-3.1.3-src
drwxr-xr-x. 10 atguigu atguigu 4096 3月 28 13:44 protobuf-2.5.0
```

3) 安装 JDK

(1) 解压 JDK

```
[root@hadoop101 hadoop_source]# tar -zxvf jdk-8u212-linux-x64.tar.gz -C /opt/module/hadoop_source/
```

(2) 配置环境变量

```
[root@hadoop101 jdk1.8.0_212]# vim /etc/profile.d/my_env.sh
输入如下内容：
#JAVA_HOME
export JAVA_HOME=/opt/module/hadoop_source/jdk1.8.0_212
export PATH=$PATH:$JAVA_HOME/bin
```

(3) 刷新 JDK 环境变量

```
[root@hadoop101 jdk1.8.0_212]# source /etc/profile
```

(4) 验证 JDK 是否安装成功

```
[root@hadoop101 hadoop_source] $ java -version
java version "1.8.0_212"
Java(TM) SE Runtime Environment (build 1.8.0_212-b10)
Java HotSpot(TM) 64-Bit Server VM (build 25.212-b10, mixed mode)
```

4) 配置 maven 环境变量, maven 镜像, 并验证

(1) 配置 maven 的环境变量

```
[root@hadoop101 hadoop_source] # vim /etc/profile.d/my_env.sh
#MAVEN_HOME
MAVEN_HOME=/opt/module/hadoop_source/apache-maven-3.6.3
PATH=$PATH:$JAVA_HOME/bin:$MAVEN_HOME/bin

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

(2) 修改 maven 的镜像

```
[root@hadoop101 apache-maven-3.6.3] # vi conf/settings.xml

# 在 mirrors 节点中添加阿里云镜像
<mirrors>
  <mirror>
    <id>nexus-aliyun</id>
    <mirrorOf>central</mirrorOf>
    <name>Nexus aliyun</name>

    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
  </mirror>
</mirrors>
```

(3) 验证 maven 安装是否成功

```
[root@hadoop101 hadoop_source] # mvn -version
Apache Maven 3.6.3 (ceceddd343002696d0abb50b32b541b8a6ba2883f)
Maven home: /opt/module/hadoop_source/apache-maven-3.6.3
Java version: 1.8.0_212, vendor: Oracle Corporation, runtime: /opt/module/hadoop_source/jdk1.8.0_212/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "linux", version: "3.10.0-862.el7.x86_64", arch: "amd64", family: "unix"
```

5) 安装相关的依赖(注意安装顺序不可乱, 可能会出现依赖找不到问题)

(1) 安装 gcc make

```
[root@hadoop101 hadoop_source] # yum install -y gcc* make
```

(2) 安装压缩工具

```
[root@hadoop101 hadoop_source] # yum -y install snappy* bzip2* lzo* zlib*
lz4* gzip*
```

(3) 安装一些基本工具

```
[root@hadoop101 hadoop_source] # yum -y install openssl* svn ncurses*
autoconf automake libtool
```

(4) 安装扩展源, 才可安装 zstd

```
[root@hadoop101 hadoop_source] # yum -y install epel-release
```

(5) 安装 zstd

```
[root@hadoop101 hadoop_source] # yum -y install *zstd*
```

6) 手动安装 cmake

(1) 在解压好的 cmake 目录下, 执行 ./bootstrap 进行编译, 此过程需一小时请耐心等待

```
[root@hadoop101 cmake-3.17.0]$ pwd  
/opt/module/hadoop_source/cmake-3.17.0  
[atguigu@hadoop101 cmake-3.17.0]$ ./bootstrap
```

(2) 执行安装

```
[root@hadoop101 cmake-3.17.0]$ make && make install
```

(3) 验证安装是否成功

```
[root@hadoop101 cmake-3.17.0]$ cmake --version  
cmake version 3.17.0  
CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

7) 安装 protobuf, 进入到解压后的 protobuf 目录

```
[root@hadoop101 protobuf-2.5.0]$ pwd  
/opt/module/hadoop_source/protobuf-2.5.0
```

(1) 依次执行下列命令 --prefix 指定安装到当前目录

```
[root@hadoop101 protobuf-2.5.0]$ ./configure --  
prefix=/opt/module/hadoop_source/protobuf-2.5.0  
  
[root@hadoop101 protobuf-2.5.0]$ make && make install
```

(2) 配置环境变量

```
[root@hadoop101 protobuf-2.5.0]$ vim /etc/profile.d/my_env.sh  
输入如下内容  
PROTOC_HOME=/opt/module/hadoop_source/protobuf-2.5.0  
PATH=$PATH:$JAVA_HOME/bin:$MAVEN_HOME/bin:$PROTOC_HOME/bin
```

(3) 验证

```
[root@hadoop101 protobuf-2.5.0]$ source /etc/profile  
[root@hadoop101 protobuf-2.5.0]$ protoc --version  
libprotoc 2.5.0
```

8) 到此, 软件包安装配置工作完成。

6.3 编译源码

1) 进入解压后的 Hadoop 源码目录下

```
[root@hadoop101 hadoop-3.1.3-src]$ pwd  
/opt/module/hadoop_source/hadoop-3.1.3-src  
  
#开始编译  
[root@hadoop101 hadoop-3.1.3-src]$ mvn clean package -DskipTests -  
Pdist,native -Dtar
```

注意: 第一次编译需要下载很多依赖 jar 包, 编译时间会很久, 预计 1 小时左右, 最终成功是全部 SUCCESS, 爽!!!

```
[INFO] Apache Hadoop Client Packaging Integration Tests ... SUCCESS [  0.080 s]
[INFO] Apache Hadoop Distribution ..... SUCCESS [ 27.180 s]
[INFO] Apache Hadoop Client Modules ..... SUCCESS [  0.044 s]
[INFO] Apache Hadoop Cloud Storage ..... SUCCESS [  0.886 s]
[INFO] Apache Hadoop Cloud Storage Project ..... SUCCESS [  0.023 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:05 h
```

2) 成功的 64 位 hadoop 包在 /opt/module/hadoop_source/hadoop-3.1.3-src/hadoop-dist/target 下

```
[root@hadoop101 target]# pwd
/opt/module/hadoop_source/hadoop-3.1.3-src/hadoop-dist/target
```