**北风网大数据**

# 1 Linux基础

## 第一节 虚拟机配置

1、Hadoop前置课程

\* Linux 系统，基本命令

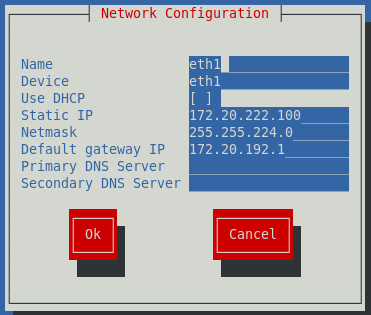
\* java语言，JSE相关知识

\* MySQL基本的DML和DDL

    SQL on Hadoop

2、克隆虚拟机或者直接打开已有虚拟机：

配置网卡时，把Name和Device都改变，配置好三项内容即可。



3、添加用户

[root@water01 ~]# useradd beifeng2

[root@water01 ~]# passwd beifeng2

\*\* 修改用户密码：

[root@water01 ~]# passwd beifeng

Changing password for user beifeng.

New password: 【输入新密码】

***修改主机名，此时主机名为localhost，不合适***     ===

[hadoop@localhost ~]$ sudo vi /etc/sysconfig/network

--------------------------------------

NETWORKING=yes

HOSTNAME=localhost.localdomain

--------------------------------------

将HOSTNAME=localhost.localdomain修改为

HOSTNAME=water01        【water01是修改后的主机名】

使文件生效：

[hadoop@localhost ~]$ sudo hostname water01

[hadoop@localhost ~]$ exit          【重新登录，可能会退回root用户，要切换到hadoop用户】

\*\* 若要通过主机名访问机器，需要域名解析器

修改配置文件：

[hadoop@water01 ~]$ sudo vi /etc/hosts

加入：

172.21.197.100      water01

接下来可以操作：

[hadoop@water01 ~]$ ping water01

4、主机名与IP地址映射

1)windows端操作：

C:\Windows\System32\drivers\etc\hosts

添加主机名和IP地址

172.20.222.100 water01

在虚拟机端：

[hadoop@water01 ~]$ sudo vi /etc/hosts

加入：

172.21.197.100      water01

5、安装Ultraedit软件，UltraEdit的菜单栏乱码问题：

【解决方案】删除IDMComp

IDMComp就在Roaming目录下C:\Users\Administrator\AppData\Roaming\IDMComp

或者C:\Documents and Settings\Administrator\AppData\Roaming\IDMComp两种方法一样的。

6、远程工具

    \*\* 远程命令行

        SecureCRT

    \*\* 远程FTP

        File Transfer Protocol

        FileZilla

    \*\* 远程编辑工具

        Notepad++

        UltraEdit

    \*\* 远程桌面工具

        Xmanager4

## 第二节 Linux基础

1、用户和组、文件

Linux系统上，创建用户的时候，默认情况会给我们创建一个用户组（名称与用户名相同）

    \*\*\*  文件

    在Linux下，文件类型（常见有三种）

        \* 文件

        \* 目录（类似win7的文件夹）

        \* 连接（类似win7下快捷方式）

    文件的归属

        \* 拥有者  owner        u

        \*属于组  group          g

        \* 其他人  other          o

    二进制数字之和的方式

        R        W        X

        2^2    2^1        2^0

2、Linux下软件的安装方式

第一种方式：

        RPM命令

####  检查某个软件是否安装

#    rpm -qa|grep java

#### 卸载已安装的软件

#  rpm -e --nodeps xxxxx

#### 安装软件

#  rpm -ivh xxx.rpm

第二种方式：

    tar

    源码

    编译

不推荐大家用源码编译方式安装

---------------------------

zip软件

    ## 解压

    $  unzip xxx.zip

    ## 压缩

    $  zip yy.zip file

tar软件

\* 解压

    ## 解压到当前目录

    $ tar -zxvf xxx.tar.gz

    ## 解压到指定目录

    $ tar -zxvf xxx.tar.gz -C dir        (dir指的是目录)

\* 压缩

    $ tar -zcvf zzzz.tar.gz dir

 第三种方式：

    yum安装

\* 需要配置源

\* 解决了软件包依赖关系以及各个软件的安装顺序

实践：安装jdk

1）查看系统是32位还是64位

[root@water01 ~]# getconf LONG\_BIT

32

2）为用户添加sudo权限

切换到root用户下：

[root@water01 ~]# vi /etc/sudoers

## Allow root to run any commands anywhere

root    ALL=(ALL)       ALL

hadoop  ALL=(ALL)       ALL

beifeng ALL=(ALL)       ALL

3）解压jdk

[beifeng@water01 ~]$ tar -zxvf jdk-7u65-linux-i586.tar.gz

[beifeng@water01 ~]$ mv jdk1.7.0\_65 /home/beifeng/app    【换目录】

[beifeng@water01 ~]$ ./java -version        【查看jdk版本】

4）环境变量配置

[beifeng@water01 ~]$ sudo vi /etc/profile

export JAVA\_HOME=/home/beifeng/app/jdk1.7.0\_65    【添加在文件尾部】

export export PATH=$PATH:$JAVA\_HOME/bin

[beifeng@water01 ~]$ source /etc/profile    【使修改生效】

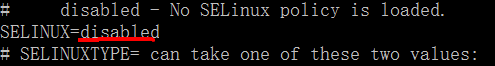
[beifeng@water01 ~]$ java -version        【测试】

##  关闭防火墙

[beifeng@water01 ~]$ sudo service iptables status

[beifeng@water01 ~]$ sudo service iptables stop

[beifeng@water01 ~]$ sudo chkconfig iptables off         【设置永久关闭】



 ##  关闭selinux

[beifeng@water01 ~]$ sudo vi /etc/sysconfig/selinux

使生效要重启

调度功能：crontab

3、Linux自带的crontab定时调度

在beifeng用户下创建定时任务

    功能：没分钟执行一次，将时间写入指定文件夹

[beifeng@water01 ~]$ crontab -e

###  first crontab

\*/1 \* \* \* \* /bin/date >> /home/beifeng/bf-log.txt

删除所有的定时任务：

[beifeng@water01 ~]$ crontab -r    【不建议用】

[beifeng@water01 ~]$ crontab -l     【查看crontab】

crontab基本定义：

语法：

\*/1 \* \* \* \* /bin/date >> /home/beifeng/bf-log.txt

\* \* \* \* \* command

说明：

    1）留个字段之间，用逗号隔开

    2）字段含义    【分时日月周】

    \*        \*        \*        \*        \*

     分：1-59，每十分钟  \*/10

            时：0-23， \*/2

                        日：1-31

                                月：1-12

                                            星期：0-6

举例：

###  每天21:30执行

30 21 \* \* \* cmd01

###  每个月1,11,21的2:30执行

30 2 1,11,21 \* \* cmd02

### 每周六或者周日，1:45执行

45 1 \* \* 6,0 cmd03

### 每天20：00-23：30，每半个小时执行一次

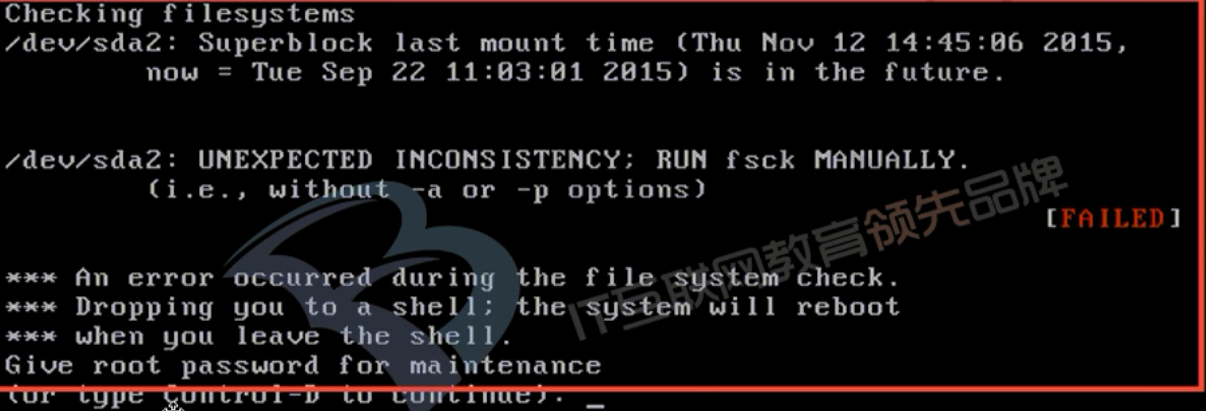
0,30 20-30 \* \* \* cmd04

### 每个小时执行一次

\* \*/1 \* \* \* cmd05

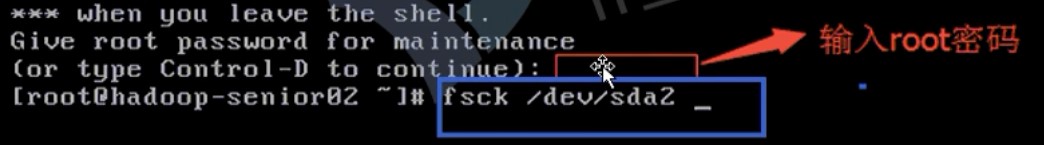
4、shell基础

前奏：虚拟机长时间不关会报错



原因：虚拟机磁盘有损坏

解决办法：回车



 输入命令：fsck /dev/sda2        --磁盘修复

若还有问题继续如此操作

\*\* shell脚本中：第一行常用来指定哪个程序来编译和执行脚本

如：    #！/bin/bash    #!/bin/sh

\*\* 变量：

根据作用域分为：本地变量（类似于java中局部变量）和环境变量

本地变量只在创建的shell脚本中有用，而环境变量则在所有用户中都有用，故也称全局变量

注：环境变量也分为当前用户环境变量和所有用户环境变量

全局环境变量：/etc/profile

当前用户环境变量：

[beifeng@water01 ~]$ cat .bash\_profile

创建和显示变量：

[beifeng@water01 scripts]$ name=beifeng

[beifeng@water01 scripts]$ echo $name

beifeng

[beifeng@water01 scripts]$ echo ${name}

beifeng

清除变量：

[beifeng@water01 scripts]$ unset name

显示当前用户所有变量：

[beifeng@water01 scripts]$ set

\* 环境变量（全局变量）需要大写

编写一个脚本：

#!/bin/shecho "hello,shell!!"

执行shell脚本方式：

[beifeng@water01 scripts]$ chmod u+x test.sh

[beifeng@water01 scripts]$ . test.sh    【有空格】

hello,shell!!

[beifeng@water01 scripts]$ sh test.sh

hello,shell!!

位置参量列表：



# 2大数据基础Hadoop 2.X

## 2.1 初识Hadoop 2.X

### 1 Hadoop简介

\*\* SQL on hadoop

1）Hive

2）Prestore

3）Impala

4）Phoneix（基于HBase）

5）Spark SQL

\*\* 来源：

GFS        -->   HDFS

MapReduce    -->   MapReduce

BigTable    -->   HBase

\* MapReduce

    对海量数据的处理

    分布式

    思想：

        分而治之

        大数据分为小的数据集

        每个数据集，进行逻辑业务处理（map）

        合并统计数据结果（reduce）

\* HDFS

    存储海量数据

    分布式

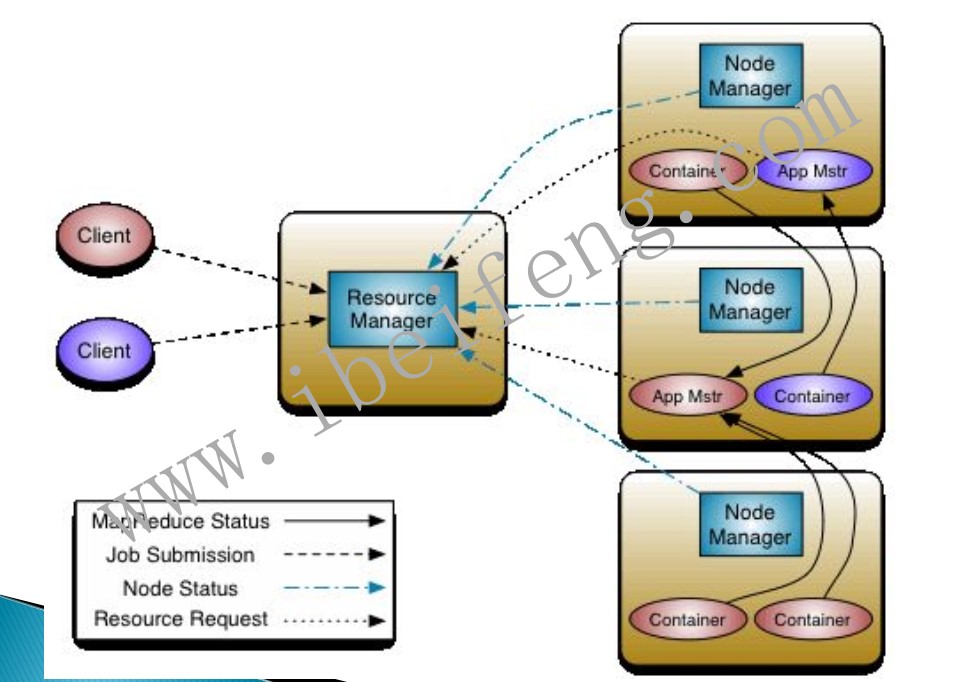
    安全性（副本数量3）

\* YARN

    分布式资源管理框架

    \* 管理每个集群的资源（内核，CPU核数）

        YARN 架构







\*\* 离线计算框架 MapReduce

# 将计算分为两个阶段，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

# Shuffle连接Map和Reduce两个阶段

1）Map Task将数据写到本地磁盘上

2）Reduce Task从每个Map Task上读取一份数据

# 仅适合离线批量处理

1）具有很好的容错性和扩展性

2）适合简单的批量处理

# 缺点明显

启动开销大，过多使用磁盘导致效率低下等

### 2.Hadoop2.x概述

a: HDFS系统架构图

主从节点来解决这种问题；

\*NameNode是主节点，存储文件的元数据如文件名，文件目录结构，文件属性（生成时间，副本数，文件权限），以及每个文件的块列表和块所在DataNode等;====》NameNode的元数据中的一部分存放在内存中的，在进程当中，另一部分存放在本地磁盘（fsimage：镜像文件和edits：编辑日志）

\*DataNode在本地文件系统存储文件块数据，以及块的校验和；

\*Secondary NameNode 用来监控HDFS状态的辅助后台程序，每隔一段时间获取HDFS元数据的快照。

b: Yarn架构图

\*ResourceManager:处理客户端请求；启动/监控ApplicationMaster;监控NodeManager;资源分配与调度

\*ApplicationMaster:数据切分；为应用程序申请资源，并分配给内部任务；任务监控与容错

\*NodeManager:单个节点上的资源管理；处理来自ResourceManager的命令；处理来自ApplicationMaster的命令

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

c:离线计算框架MapReduce

\*MapReduce on Yarn

d:通过实际需求介绍Hadoop的生态系统

\*数据库的需求：用的比较多的框架Sqoop(SQL to Hadoop),关系型的数据库把数据存储到HDFS上。

\*日志文件的收集：Flume

\*任务的管理与调度：Oozie

\*MapReduce不是那么好些，门口比较高，又有了Hive。

\*整个集群资源的管理界面：Clodera Manager

\*协作框架：Zookeeper

### 3 hadoop 安装

#### 1 普通用户增加到 sudo 组

# vim /etc/sudoers

# 添加行

hadoop ALL=(ALL) ALL

#### 2 修改主机名

## 修改主机名文件

$ sudo vim /etc/sysconfig/network

HOSTNAME=hadoop1

## 修改hosts文件

#### 3 安装jdk

$ tar -zxvf jdk-7u79-linux-x64.tar.gz

$ mv jdk1.7.0\_79 jdk

## 配置环境变量

$ sudo vim /etc/profile

## JAVA\_HOME

export JAVA\_HOME=/home/hadoop/app/jdk

export PATH=$PATH:/home/hadoop/app/jdk/bin:$PATH ##用绝对路径

## 立即生效

$ source /etc/profile

## 查看安装情况

$ java -version

#### 4 安装hadoop

$ rm -rf hadoop-2.7.0.tar.gz

$ mv hadoop-2.7.0 hadoop

## 修改配置文件

所有配置文件都在 /home/hadoop/app/hadoop/etc/hadoop 目录下

1 修改 hadoop-env.sh

export JAVA\_HOME=/home/hadoop/app/jdk

可执行example

#### 5 hadoop安装模式

* 独立模式

没有运行的守护程序，所有程序运行在一个JVM中。适合开发期间运行MapReduce程序，源于他的易于测试和调试。

* 伪分布式模式

守护程序运行在本地主机，模拟一个小规模集群。

* 完全分布式模式

守护程序运行在多个主机的集群上。

#### 6 本地模式

1 修改core-site.xml文件

## 创建临时文件目录 /home/hadoop/app/hadoop/data/tmp

<configuration>

 <property>

        <name>fs.defaultFS</name>

                <value>hdfs://hadoop1:8020</value>

 </property>

<property>

        <name>hadoop.tmp.dir</name>

        <value>/home/hadoop/app/hadoop/data/tmp</value>

 </property>

</configuration>

2 配置 hdfs-site.xml

hdfs文件副本数

<configuration>

<property>

        <name>fs.replication</name>

                <value>1</value>

</property>

<property>

        <name>dfs.namenode.secondary.http-address</name>

                <value>hadoop3:50090</value>

</property>

</configuration>

3 mapred-site.xml

mv mapred-site.xml.template mapred-site.xml     --修改文件名称

vi mapred-site.xml

<!-- 指定mr运行在yarn上 -->

<configuration>

<property>

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

</property>

</configuration>

4 yarn-site.xml

[hadoop@water02 hadoop]$ vi yarn-site.xml

<!-- 指定YARN的老大（ResourceManager）的地址 -->

<configuration>

<property>

        <name>yarn.resourcemanager.hostname</name>

        <value>hadoop1</value>

</property>

<!-- reducer获取数据的方式 -->

<property>

        <name>yarn.nodemanager.aux-services</name>

        <value>mapreduce\_shuffle</value>

</property>

</configuration>

5 格式化

$ bin/hdfs namenode -format

【备注】如果要重新格式化，请在data/tmp目录下删除文件

６ 启动守护进程

## 启动namenode

$ sbin/hadoop-daemon.sh start namenode

## 启动datanode

$ sbin/hadoop-daemon.sh start datanode

sbin/start-all.sh

centos6永久关闭防火墙

chkconfig iptables off

正确关闭步骤：

service iptables stop

chkconfig iptables off

windows 也要关闭防火墙

挂壁senlinux

# vim /etc/selinux/config

SELINUX=disabled

web界面访问

<http://192.168.246.133:50070>

8 文件操作

bin/hdfs dfs -mkdir -p /user/hadoop/mapreduce/wordcount/input

## 上传文件

$ bin/hdfs dfs -put wcinput/wc.input /user/hadoop/mapreduce/wordcount/input

## 查看目录

$ bin/hdfs dfs -ls -R /

## 执行example例程

$ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar wordcount /user/hadoop/mapreduce/wordcount/input/ /user/hadoop/mapreduce/wordcount/output

【报错】 Name node is in safe mode.

namenode在安全模式下

【解决方案】

bin/hadoop dfsadmin -safemode leave

!!!注意

用户可以通过dfsadmin -safemode value 来操作安全模式，参数value的说明如下：

enter - 进入安全模式

leave - 强制NameNode离开安全模式

get - 返回安全模式是否开启的信息

wait - 等待，一直到安全模式结束。

#### 7 部署YARN及在YARN上运行mapreduce

1 修改yarn-env.sh -- **不执行，执行报错**

export JAVA\_HOME=/home/hadop/app/jdk

2 修改yarn-site.xml

<property>

        <name>yarn.resourcemanager.hostname</name>

        <value>hadoop1</value>

</property>

<!-- reducer获取数据的方式 -->

<property>

        <name>yarn.nodemanager.aux-services</name>

        <value>mapreduce\_shuffle</value>

</property>

3 修改slaves

hadoop1

4 修改mapred-site.xml

<configuration>

<property>

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

</property>

</configuration>

5 执行

$ bin/yarn jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar wordcount /user/hadoop/mapreduce/wordcount/input/ /user/hadoop/mapreduce/wordcount/output

#### 8 历史服务器配置启动查看

历史服务器historyserver

查看已经运行完成的MapReduce作业记录，比如用了多少个Map,用了多少个Reduce，作业提交时间，作业启动时间，作业完成时间等信息。

\*\*启动命令：

$ sbin/mr-jobhistory-daemon.sh start historyserver

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

\*\*Web Ui:

$ http://hadoop1:19888/

\*\* Yarn的日志聚集功能配置使用，Log Aggregation  
    聚集：应用程序完成之后，将日志信息上传到HDFS系统

备注：配置yarn-site.xml，以实现历史服务器historyserver配置和Yarn的日志聚集功能配置

 <property>

        <name>yarn.log-aggregation-enable</name>

                <value>true</value>

 </property>

  <property>

        <name>yarn.log-aggregation.retain-seconds</name>

                <value>640800</value>

 </property>

#### 9 Hadoop两类配置文件讲解及垃圾回收

\*\* 默认配置文件:对应的jar包中

    core-default.xml

    hdfs-default.xml

    yarn-default.xml

    mapred-default.xml

\*\* 自定义配置文件:$HADOOP\_HOME/etc/hadoop

    core-site.xml

    hdfs-site.xml

    yarn-site.xml

    mapred-site.xml

\*\* 垃圾回收的设置

在core-site.xml中添加

<property>

   <name>fs.trash.interval</name>

   <value>10080</value>

</property>

备注：垃圾保留7天，单位：分钟

### 4 无密码登陆

$ ssh-keygen

$ cd .ssh

$ ls

id\_rsa id\_rsa.pub

$ cat id\_rsa.pub >> authorized\_keys

$ ssh localhost

## 为了避免出错，先不要执行此命令。因为执行后，会在.ssh/目录下产生一个known\_hosts文件。拷贝到其他主机后悔报错出问题。

赋权限：

chmod 600 authorized\_keys

可以直接登录到localhost

hadoop@hadoop1:~/.ssh$ ls

authorized\_keys id\_rsa id\_rsa.pub

#### 拷贝到其他主机

scp -r .ssh hadoop@hadoop1:/home/hadoop

## 2.2 深入Hadoop 2.X

### 2.1.1 HDFS架构详解

**谷歌GFS论文中文**：

MapReduce：  http://blog.csdn.net/active1001/archive/2007/07/02/1675920.aspx  
GFS:                 http://blog.csdn.net/xuleicsu/archive/2005/11/10/526386.aspx  
BigTale:             <http://blog.csdn.net/accesine960/archive/2006/02/09/595628.aspx>

一、HDFSD架构、启动过程

1、HDFS来源



2、HDFS的结构

3、NameNode



NameNode

\* 负责管理文件系统的名字空间[namespace]以及客户端对文件的访问。

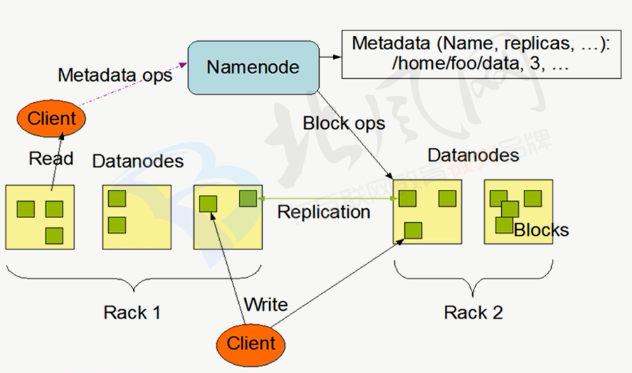
\* 存储元数据

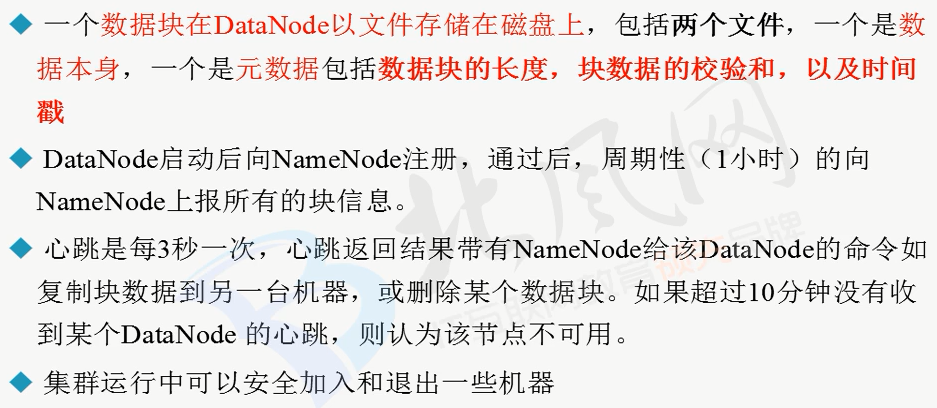
\* 接受来自datanode的心跳信号和块报告，监控datanode管理数据块的复制。

1. DateNode

\* 数据块在Datanode上以文件形式存储在磁盘上（a.数据本身；b,元数据）

\* DataNode向NameNode发送块报告

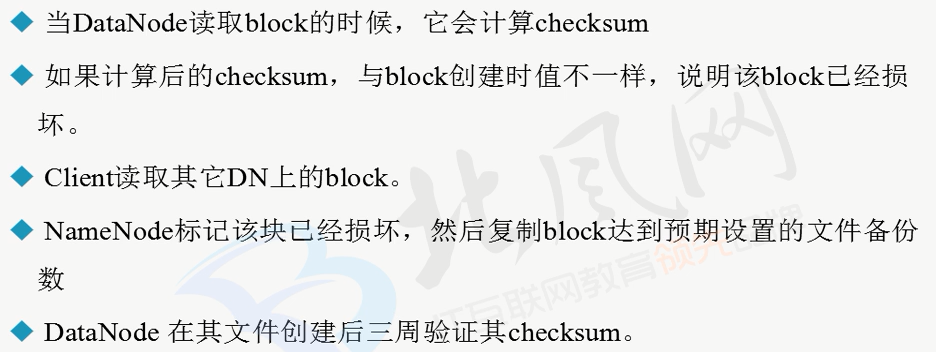




5、文件



6、数据损坏处理



### 2.2.2 Spark与Hadoop MR的异同

**总结**

\* spark中间数据 --> 内存，迭代计算效率高

mapreduce -->磁盘，影响性能

\* 容错性：

spark --> RDD [通过整个数据集的计算流程的血缘关系实现来重建]

mapreduce --> 重新计算，成本高

\* spark --> 通用：transformation和action，sparkstreamingGraphX

mapreduce --> map,reduce,其他模块缺乏

\* spark框架和生态复杂[RDD、血缘lineage、有向无环图DAG、stage划分]，功能强大高效，但是需要根据不同的业务场景调优

mapreduce 生态简单，但是运行较为稳定，适合长期后台运行。

1、spark把运算的中间数据存放在内存，迭代计算效率更高；mapreduce的中间结果需要落地，需要保存到磁盘，这样必然会有磁盘io操做，影响性能。

2、spark容错性高，它通过弹性分布式数据集RDD来实现高效容错，RDD是一组分布式的存储在节点内存中的只读性质的数据集，这些集合是弹性的，某一部分丢失或者出错，可以通过整个数据集的计算流程的血缘关系来实现重建；mapreduce的话容错可能只能重新计算了，成本较高。

3、spark更加通用，spark提供了transformation和action这两大类的多个功能api，另外还有流式处理sparkstreaming模块、图计算GraphX等等；mapreduce只提供了map和reduce两种操作，流计算以及其他模块的支持比较缺乏。

4、spark框架和生态更为复杂，首先有RDD、血缘lineage、执行时的有向无环图DAG、stage划分等等，

很多时候spark作业都需要根据不同业务场景的需要进行调优已达到性能要求；mapreduce框架及其生态相对较为简单，对性能的要求也相对较弱，但是运行较为稳定，适合长期后台运行。

### 2.2.3 HDFS操作

## 读取本地文件

$ bin/hdfs dfs -Dfs.defaultFS=file:/// -ls /

## 集群的状态信息

$ bin/hdfs dfsadmin -report

### 2.2.4 环境准备

#### 1 maven安装

参考文档

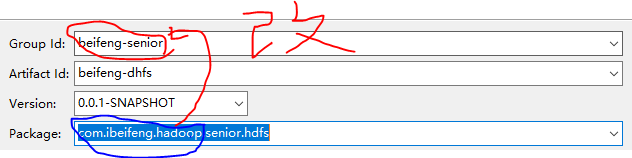
<https://www.cnblogs.com/wkrbky/p/6350334.html?utm_source=itdadao&utm_medium=referral>

eclipse maven配置

<https://www.cnblogs.com/tangshengwei/p/6341462.html>

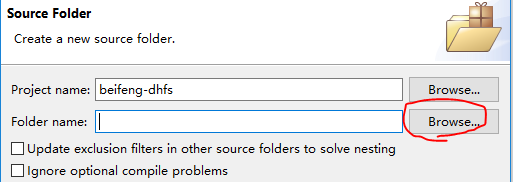
2 创建maven

new -- project -- maven project -- 默认



新建resources文件夹

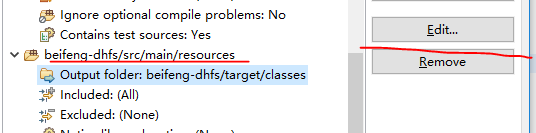
工程-- new -- source fouder --

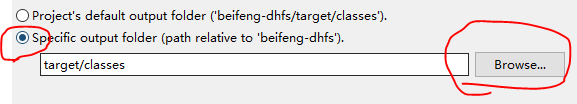


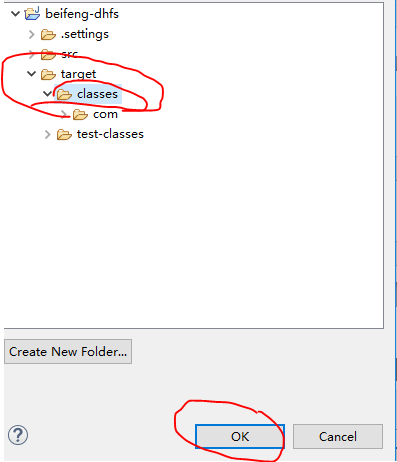


命名为resources

## 工程右击 -- build path -- Build Path Configure -- Source --







3 修改pom

<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.ibeifeng.hadoop</groupId>

<artifactId>beifeng-dhfs</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>beifeng-dhfs</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<hadoop.version>2.7.0</hadoop.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>${hadoop.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

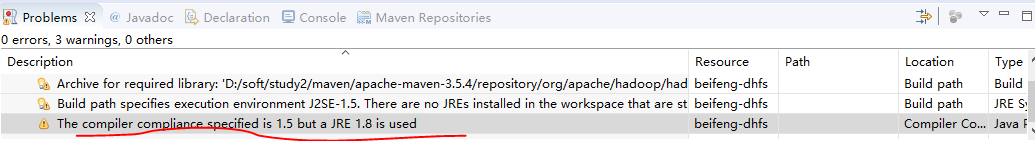
#### 2 maven工程上出现红感叹号

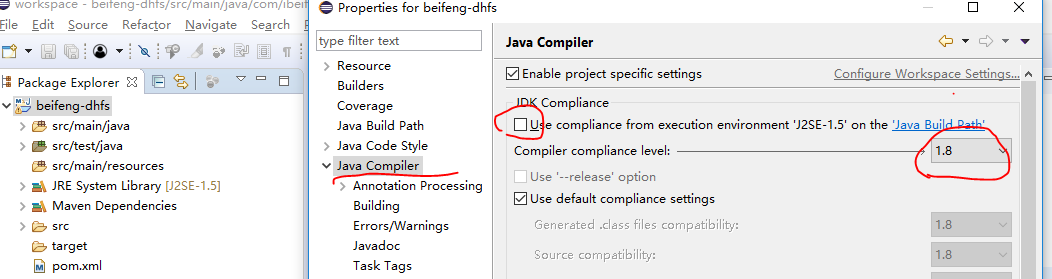
【解决方案】

<https://jingyan.baidu.com/article/86fae346f63b233c49121aa3.html>

<https://www.cnblogs.com/thomasbc/p/6860950.html>

另外





### 2.2.5 NameNode启动过程详解

namenode 存放整个文件系统的元数据。

NameNode

（存储在）

\* 内存

\* 本地磁盘

# fsimage [镜像文件]

# edits 编辑日志文件[存储文件系统元数据变化的信息（命名空间）]

格式化HDFS文件系统就是为了生成fsimage

0 -> format

1 -> fsimage

2 -> Start namenode

\* read fsimage

3 -> Start DataNode

\* 注册(向namenode注册)

\* block report（发送块报告给namenode）

4 -> hdfs mkdir dir ... -> write [edits]

5 -> hdfs put file ... -> write [edits]

所以说，namenode存储的元数据，一份在内存，一份在本地磁盘

=========

第二次启动HDFS

1 -> Start Namenode

\* read fsimage

\* read edits

\* 加载 fsimage 和 edits 到内存，生成新的fsimage [new]

\* 生成新的edits [空]

### 4 SecondaryNameNode

辅助namenode

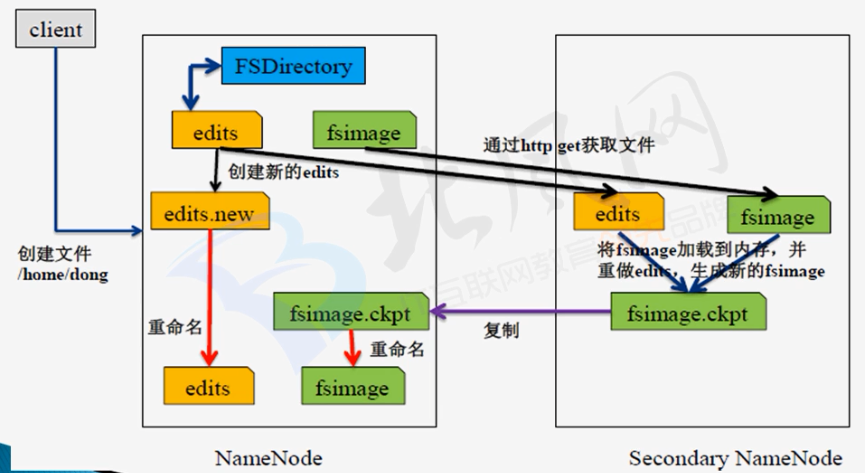
\* 内存

\* 本地磁盘

# fsimage

# edits

SecondaryNameNode**定期合并fsimage 和edits -> new fsimage**



### 5 HDFS 启动safemode

安全模式SafeMode

\* 等待DataNode向它发送块的报告

\* （NameNode汇总块的信息）

datanode blocks/total blocks = 99.99%

此时安全模式才会退出

操作（可执行）：

\* 查看系统文件

\* 不能改变文件系统的命名空间

\* 创建文件夹

\* 上传文件

\* 删除文件

### 6 手动进入SafeMode

用户可以通过dfsadmin -safemode value 来操作安全模式，参数value的说明如下：

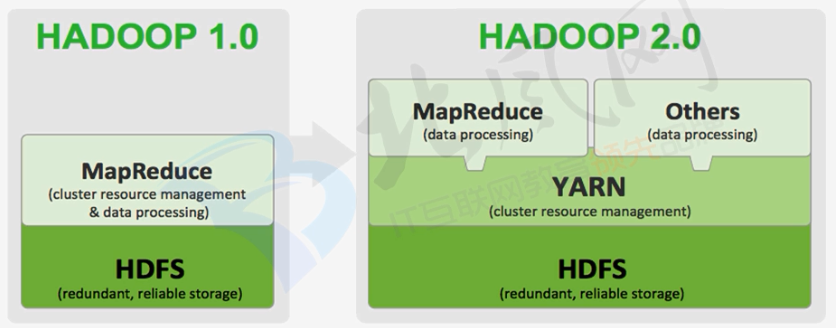
enter - 进入安全模式

leave - 强制NameNode离开安全模式

get - 返回安全模式是否开启的信息

wait - 等待，一直到安全模式结束。

### 7 YARN 组件功能



YARN：Cluster resource managment

对于hadop2,mapreduce仅仅是**运行在YARN上的一个应用框架**

hadoop1中mapreduce有单独进程，但是在hadoop2中，mapreduce无单独进程.

#### ResourceManager

全局的资源管理器，整个集群只有一个，负责集群资源的统一管理和调度分配。

功能：

\* 处理客户端请求

\* 启动/监控ApplicationMaster

\* 监控NodeManager

\* 资源分配与调度

#### NodeManager

整个集群有多个，负责单节点资源管理和使用

功能：

\* 单个节点上的资源管理和任务管理

\* 处理来自ResourceManager的命令

\* 处理来自ApplicationManager的命令

NM管理抽象容器，这些容器代表着可供一个特定应用程序使用的针对每个节点的资源。定时向RM汇报本节点上资源的使用情况和各个Container的运行状态。

#### ApplicationMaster

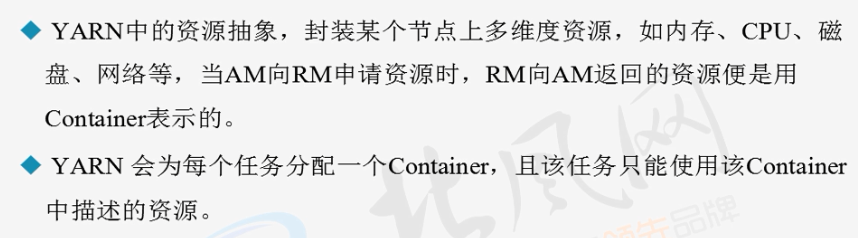
YARN中每个应用都会启动一个AM，负责向RM申请资源，请求NM启动container，并告诉container做什么事情。

管理一个在YARN内运行应用程序的每个实例。

负责协调来自RM的资源，并通过NM监视容器的执行和资源的使用（CPU、内存等资源的分配）

#### Container

资源容器。YARN中所有的应用都是在container之上运行的。AM也是在container上运行的，不过AM的container是RM申请的。



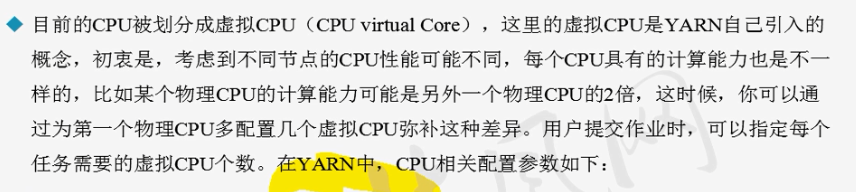
### 8 YARN 资源管理

YARN -- 资源调度、资源隔离

\* 资源调度 --> ResourceManager

\* 资源隔离 --> NodeManager





yarn配置文档

<http://hadoop.apache.org/docs/r2.5.2/hadoop-yarn/hadoop-yarn-common/yarn-default.xml>

#### hadoop各个版本文档集合

<http://hadoop.apache.org/docs/>

在 yarn-site.xml文件添加：

 <property>

        <name>yarn.nodemanager.resource.memory-mb</name>

                <value>4096</value>

 </property>

  <property>

        <name>yarn.nodemanager.resource.cpu-vcores</name>

                <value>2</value>

 </property>

### 9 MapReduce 工作过程

Hadoop的分布式计算框架（MapReduce）-- 适合离线计算

核心思想： 移动计算而不移动数据

MapReduce的执行步骤：

#### 1 Map任务处理

**1 split过程**

读取HDFS中的文件。每一行解析成一个<k,v>。每一个键值对调用一次map函数。

（<0,hello you>   <10,hello me> ）

**2 map过程**

覆盖map()，接收1产生的<k,v>，进行处理，转换为新的<k,v>输出。

（<hello,1> <you,1> <hello,1> <me,1>）

**3** [**Partitioner**](http://www.cnblogs.com/ahu-lichang/p/6657895.html)**分区**

对2 输出的<k,v>进行分区。默认分为一个区。[**Partitioner**](http://www.cnblogs.com/ahu-lichang/p/6657895.html)（分区的目的：1.根据业务需要，产生多个输出文件，2，多个reduce任务在运行，提高整体job的运行效率）

**4 排序分组**

对不同分区中的数据进行排序（按照k）、分组。

分组指的是相同key的value放到一个集合中。

排序后：<hello,1> <hello,1> <me,1> <you,1>

分组后：<hello,{1,1}><me,{1}><you,{1}>

**5** [**Combiner**](http://www.cnblogs.com/ahu-lichang/p/6657572.html)**归约**[可选]

对分组后的数据进行归约。

每一个map可能会产生大量的输出，combiner的作用就是在map端对输出先做一次合并，以减少传输到reducer的数据量。

combiner最基本是实现本地key的归并，combiner具有类似本地的reduce功能。 如果不用combiner，那么，所有的结果都是reduce完成，效率会相对低下。使用combiner，先完成的map会在本地聚合，提升速度。

#### 2 Reduce任务处理

**1 copy**

多个map任务的输出，按照不同的分区，通过网络copy到不同的reduce节点上。（shuffle）

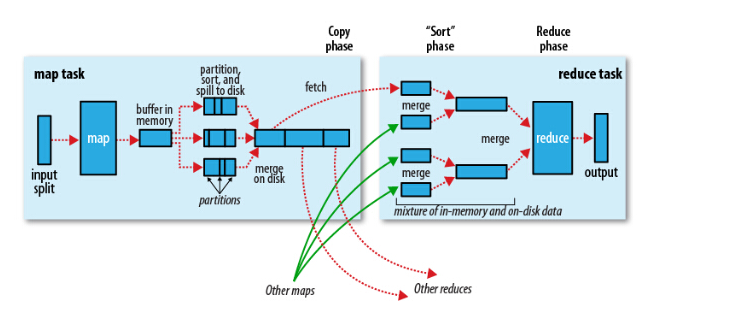
**2 merge**

对多个map的输出进行合并、排序。覆盖reduce函数，接收的是分组后的数据，实现自己的业务逻辑，处理后，产生新的<k,v>输出。

　<hello,2> <me,1> <you,1>

3 对reduce输出的<k,v>写到HDFS中。

#### 3 Shuffle [hadoop2]



从map到reduce的过程，被称之为shuffle过程.当map任务将数据output时，不仅仅是将结果输出到磁盘，它是将其写入内存缓冲区域，并进行一些预分类。

**## 1 Map端**

首先map任务的output过程是一个环状的内存缓冲区，缓冲区的大小默认为100MB，当写入内存的大小到达一定比例，默认为80%,便开始写入磁盘。

　　在写入磁盘之前，线程将会指定数据写入与reduce相应的patitions中，最终传送给reduce.在每个partition中，后台线程将会在内存中进行Key的排序，（如果代码中有combiner方法，则会在output时就进行sort排序，这里，如果只有少于3个写入磁盘的文件，combiner将会在outputfile前启动，如果只有一个或两个，那么将不会调用）

　 这里将map输出的结果进行压缩会大大减少磁盘IO与网络传输的开销。

　 随后这些paritions输出文件将会通过HTTP发送至reducers。

**## 2 Reduce端**

首先上面每个节点的map都将结果写入了本地磁盘中，现在reduce需要将map的结果通过集群拉取过来，这里要注意的是，需要等到所有map任务结束后,reduce才会对map的结果进行拷贝，由于reduce函数有少数几个复制线程，以至于它可以同时拉取多个map的输出结果。默认的为5个线程。

　　当所有map的任务结束后，applicationMaster通过心跳机制（heartbeat mechanism)，由它知道mapping的输出结果与机器host,所以reducer会定时的通过一个线程访问applicationmaster请求map的输出结果。

　　Map的结果将会被拷贝到reduce task的JVM的内存中，如果不够用，则会写入磁盘。当内存缓冲区的大小到达一定比例时或map的输出结果文件过多时，将会除法合并(merged)随之写入磁盘。

这时要注意，所有的map结果这时都是被压缩过的，需要先在内存中进行解压缩，以便后续合并它们。 最终reduce进行运算进行输出。

mapreduce [**MapReduce运行原理详解**](https://www.cnblogs.com/yangsy0915/p/5559969.html)

https://www.cnblogs.com/yangsy0915/p/5559969.html

### 10 MapReduce编程模型

**package** com.ibeifeng.hadoop.senior.mapreduce;

**import** java.io.IOException;

**import** java.util.StringTokenizer;

**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;

**public** **class** WordCount {

// step 1 : Mapper Class

**public** **static** **class** WordCountMapper **extends**

Mapper<LongWritable, Text, Text, IntWritable> {

// 输出单词

**private** Text mapOutputKey = **new** Text();

// 出现一次就记作一次

**private** IntWritable mapOutputValue = **new** IntWritable(1);

@Override

**public** **void** map(LongWritable key, Text value, Context context)

**throws** IOException, InterruptedException {

System.***out***.println("map-in-0-key: " + key.get() + " -- "

+ "map-in-value: " + value.toString());

// line value

// 获取文件每一行的<key,value>

String lineValue = value.toString();

// split

// 分割单词，以空格分割

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

//String太占空间内存

// StringTokenizer stringTokenizer = new StringTokenizer(lineValue);

// while((Boolean) StringTokenizer.hasMoreTokens()){

// String wordValue = stringTokenizer.nextToken();

// }

// iterator

// 将数组里面的每一个单词拿出来，一个个组成<key,value>

// 生成1

**for** (String str : strs) {

// set map output key

// 设置key

mapOutputKey.set(str);

// output

// 最终输出

context.write(mapOutputKey, mapOutputValue);

}

}

}

// step 2 : Reducer Class

**public** **static** **class** WordCountReducer **extends**

Reducer<Text, IntWritable, Text, IntWritable> {

**private** IntWritable outputValue = **new** IntWritable();

@Override

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

Context context) **throws** IOException, InterruptedException {

// temp : sum

// 定义一个临时变量

**int** sum = 0;

// iterator

// 对于迭代器中的值进行迭代累加，最后sum加完以后就是统计的次数

**for** (IntWritable value : values) {

// total

sum += value.get();

}

// set output value

outputValue.set(sum);

// output

context.write(key, outputValue);

}

}

// step 3 : Driver

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

Configuration configuration = **new** Configuration();

Job job = Job.*getInstance*(configuration, **this**.getClass()

.getSimpleName());

job.setJarByClass(WordCount.**class**);

// set job

// input

Path inpath = **new** Path(args[0]);

FileInputFormat.*addInputPath*(job, inpath);

// output

Path outpath = **new** Path(args[1]);

FileOutputFormat.*setOutputPath*(job, outpath);

// Mapper

job.setMapperClass(WordCountMapper.**class**);

job.setMapOutputKeyClass(Text.**class**);

job.setMapOutputValueClass(IntWritable.**class**);

// Reducer

job.setReducerClass(WordCountReducer.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

// submit job

**boolean** isSuccess = job.waitForCompletion(**true**);

**return** isSuccess ? 0 : 1;

}

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

// 传递两个参数，设置路径

// args = new String[] {

// // 参数1：输入路径

// "hdfs://hadoop1:8020/user/hadoop1/mapreduce/wordcount/input",

// // 参数2：输出路径

// "hdfs://hadoop1:8020/user/hadoop1/mapreduce/wordcount/output" };

// run job

**int** status = **new** WordCount().run(args);

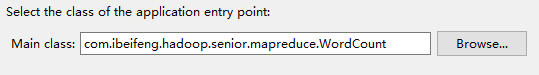
System.*exit*(status);

}

}

提交运行

1 将文件打包时注意：eclipse打jar包要next -> next -> 指定其类



2 提交运行

$ bin/yarn jar jars/wc1.jar /user/hadoop1/mapreduce/input0 /user/hadoop1/mapreduce/output4

input0里面存放待wordcount的文件，output4是输出路径，必须是不存在的路径。

3 查看结果

$ bin/hdfs dfs -text /user/hadoop1/mapreduce/output4/par\*

11 MapReduce数据类型



基本数据类型：

BooleanWritable :标准布尔型数值

ByteWritable ：单字节数值

DoubleWritable：双字节数值

FloatWritable：浮点数值

IntWritable：整形数

LongWritable：长整形

Text：使用UTF-8格式存储的文件

NullWritable:<key,value>中key或者value为空时使用

### 11 编程模型优化

#### 编程模型

**import** java.io.IOException;

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.conf.Configured;

**import** org.apache.hadoop.fs.Path;

**import** org.apache.hadoop.util.Tool;

**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.ToolRunner;

**public** **class** ModuleWordCount **extends** Configured **implements** Tool{

// map

**public** **static** **class** WordCountMap **extends** Mapper<LongWritable, Text, Text, IntWritable>{

@Override

**protected** **void** setup(Context context)

**throws** IOException, InterruptedException {

//Nothing

}

@Override

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

**throws** IOException, InterruptedException {

//**TODO**

}

@Override

**protected** **void** cleanup(

Context context)

**throws** IOException, InterruptedException {

//Nothing

}

}

//reduce

**public** **static** **class** WordCountReduce **extends** Reducer<Text, IntWritable, Text, IntWritable>{

@Override

**protected** **void** setup(Context context)

**throws** IOException, InterruptedException {

//Nothing

}

@Override

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

Context context)

**throws** IOException, InterruptedException {

//**TODO**

}

@Override

**protected** **void** cleanup(

Context context)

**throws** IOException, InterruptedException {

//Nothing

}

}

//driver

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

//得到配置信息

Configuration configurable=getConf();

//创建Job

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

//运行jar

job.setJarByClass(**this**.getClass());

//设置job

//1. input

Path inPath=**new** Path(args[0]);

FileInputFormat.*addInputPath*(job, inPath);

// FileInputFormat.addInputPath(job, inPath);

//2.map

job.setMapperClass(WordCountMap.**class**);

job.setMapOutputKeyClass(Text.**class**);

job.setMapOutputValueClass(IntWritable.**class**);

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Shuffle\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//分区

job.setParttitionerClass(cls);

//排序

job.setSortComparatorClass(cls);

//combiner

job.setCombinerClass(cls);

//分组

job.setGroupingComparatorClass(cls);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*/

//3.reduce

job.setReducerClass(WordCountReduce.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

//设置reduce数量

// job.setNumReduceTasks(数量);

//4.output

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

FileOutputFormat.*setOutputPath*(job, outPath);

//提交job

**boolean** completion = job.waitForCompletion(**true**);

**return** completion ? 0 : 1;

}

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

//得到配置信息

Configuration configurable=**new** Configuration();

//设置压缩

configurable.set("mapreduce.map.output.compress","true");

configurable.set("mapreduce.map.output.compress.codec","压缩方法");

**int** status=ToolRunner.*run*(configurable, **new** ModuleWordCount(), args);

System.*exit*(status);

}

}

#### 优化

1. 在map和reduce方法中添加了setup和cleanup方法，setup方法可以进行初始化，cleanup方法进行资源的关闭
2. shuffle阶段:



1. **压缩(compress)**:     这个可以在配置文件中配置，也可以在代码中进行设置

4)Reduce的个数设置，这个在实际应用中是比较重要的，默认是一个，并不是越多越好，而是尽量保持在一个波动范围内

### 12 MapReduce执行流程shuffle

#### 过程

\* step1:

input

InputFormat

\* 读取数据

\* 转换成<key,value>

FileInputForat

\* TextInputFormat

\* step2:

map

ModuleMapper自定义mapper类

map(KEYIN,VALUEIN,KEYOUT,VALUEOUT)

\* 默认情况下：

KEYIN：LongWritable

VALUE:TEXT

\* step3:

shuffle [洗牌]

\* process

\* map,output<key,value>

\* 最初存在memory中(有一定限制，若超过了这个限制-->next )

\* spill，溢写到磁盘中，可能有很多文件

\* 分区[指定哪些map给哪些reduce处理]

\* 排序

\* 很多小文件，spill

\* 合并[[**Combiner**](http://www.cnblogs.com/ahu-lichang/p/6657572.html)]，merge

\* 排序

大文件 --> Map Task 运行的机器的本地磁盘

---------------------------------------- map 任务结束

\* copy

Reduce Task ,回到Map Task 运行的机器上，拷贝要处理的数据[到reduce节点]

\* 合并，merge,排序

\* 分组group

将相同key的value放在一起

step4:reduce

reduce(KEYIN,VALUEIN,KEYOUT,VALUEOUT)

map输出<key,value>类型与reduce输入的数据类型一致

\* step5

output

outputFormat

FileOutputFormat

TextOutputFormat

每个<key,value>对输出一行，key与value间的分隔符为制表符\t

默认调用key和value的toString()方法

### 13 MapReduce 调优

#### 1 目的

充分的利用机器的性能，更快的完成mr程序的计算任务。甚至是在有限的机器条件下，能够支持运行足够多的mr程序。

#### 2 调优的总体概述

从mr程序的内部运行机制，我们可以了解到一个mr程序由mapper和reducer两个阶段组成，其中mapper阶段包括：

(1)数据的读取、

(2)map处理以及写出操作(排序和合并/sort&merge)，

而reducer阶段包含：

(1)对mapper端输出数据的获取、

(2)数据合并(sort&merge)、

(3)reduce处理以及写出操作。

那么在这七个子阶段中，能够进行较大力度的进行调优的就是

**map端的输出**、reducer端的数据合并以及reducer的个数这三个方面的调优操作。

也就是说虽然性能调优包括cpu、内存、磁盘io以及网络这四个大方面，但是从mr程序的执行流程中，我们可以知道主要有调优的是内存、磁盘io以及网络。在mr程序中调优，主要考虑的就是**减少网络传输和减少磁盘IO操作**，本笔记的mr调优主要包括：

服务器调优、代码调优、mapper调优、reducer调优以及runner调优这五个方面。

#### 3 配置文件参数调优

mapreduce调优

1. mapreduce.task.io.sort.factor --->mr程序进行合并排序的时候，打开的文件数量，默认为10个.

2. mapreduce.task.io.sort.mb ---> mr程序进行合并排序操作的时候或者mapper写数据的时候，内存大小，默认100M

3. mapreduce.map.sort.spill.percent ---> mr程序进行flush操作的阀值，默认0.80。

4. mapreduce.reduce.shuffle.parallelcopies --->mr程序reducer copy数据的线程数，默认5。

5. mapreduce.reduce.shuffle.input.buffer.percent --->reduce复制map数据的时候指定的内存堆大小百分比，默认为0.70，适当的增加该值可以减少map数据的磁盘溢出，能够提高系统能。

6. mapreduce.reduce.shuffle.merge.percent --->reduce进行shuffle的时候，用于启动合并输出和磁盘溢写的过程的阀值，默认为0.66。如果允许，适当增大其比例能够减少磁盘溢写次数，提

高系统性能。同mapreduce.reduce.shuffle.input.buffer.percent一起使用。

1. mapreduce.task.timeout --->mr程序的task执行情况汇报过期时间，默认600000(10分钟)，设置为0表示不进行该值的判断。

Uber模式也是一种优化：

1、概念：Uber模式是Hadoop2.0中实现的一种针对MR小作业的优化机制。

即如果作业足够小，则所有task在一个jvm（mrappmaster）中完成要比为每个task启动一个container更划算。

2.下面是该机制的相关参数，这些参数均为客户端配置。

在hadoop2.X版本中新增加了Uber模式运行MR

Hadoop中对Uber模式的定义：

1：mapreduce.job.ubertask.enable=true，首先开启Uber模式，默认是false

2：mapreduce.job.ubertask.maxmaps map任务数的阀值 9

mapreduce.job.ubertask.maxreduces reduce任务数的阀值 1

map的数量<=9， reduce<=1

3：所有的输入文件的总长度<=默认的块的大小（128M）

4：mapreduce.map.memory.mb(默认是1024)<=内存需求(内存需求的大小由yarn.app.mapreduce.am.resource.mb来决定，默认1536M)

5：cpu<=yarn.app.mapreduce.am.resource.cpu-vcores(默认1)

6：采用非链式方式运行MR

Uber模式优点：

针对多个小作业，开启uber模式，mapreduce会将所有的task任务放在一个JVM中完成，就需要每个task都去申请资源，启动一个Container容器，

而是多个task申请一份资源，资源会重复的利用，这样的话可以节省cpu及网络Io，磁盘Io的消耗，节省了job运行的时间。

#### 4 代码调优

代码调优，主要是mapper和reducer中，针对多次创建的对象，进行代码提出操作。这个和一般的java程序的代码调优一样。

#### 5 mapper调优

mapper调优主要就是就一个目标：减少输出量。我们可以通过增加combine阶段以及对输出进行压缩设置进行mapper调优。

1.combine介绍：

实现自定义combine要求继承reducer类，特点：

以map的输出key/value键值对作为输入输出键值对，作用是减少网络输出，在map节点上就合并一部分数据。

适用场景，map的输出是数值型的，方便进行统计。

2.压缩设置：

在提交job的时候分别设置启动压缩和指定压缩方式。mapreduce.map.output.compress--->设置是否启动map输出的压缩机制，默认为false。在需要减少网络传输的时候，可以设置为true。

#### 6 reducer调优

reducer调优主要是通过参数调优和设置reducer的个数来完成。

reducer个数调优：

要求：一个reducer和多个reducer的执行结果一致，不能因为多个reducer导致执行结果异常。

规则：一般要求在hadoop集群中的执行mr程序，map执行完成100%后，尽量早的看到reducer执行到33%，可以通过命令hadoop job -status job\_id或者web页面来查看。

原因： map的执行process数是通过inputformat返回recordread来定义的；而reducer是有三部分构成的，分别为读取mapper输出数据、合并所有输出数据以及reduce处理，其中第一步要依赖map的执行，所以在数据量比较大的情况下，一个reducer无法满足性能要求的情况下，我们可以通过调高reducer的个数来解决该问题。

优点：充分利用集群的优势。

缺点：有些mr程序没法利用多reducer的优点，比如获取top n的mr程序。

#### 7 runner调优

runner调优其实就是在提交job的时候设置job参数

1~8详见ActiveUserRunner(before和configure方法)，9详解TransformerBaseRunner(initScans方法)

1. mapred.child.java.opts: 修改childyard进程执行的jvm参数，针对map和reducer均有效，默认：-Xmx200m

2. mapreduce.map.java.opts： 需改map阶段的childyard进程执行jvm参数，默认为空，当为空的时候，使用mapred.child.java.opts。

3. mapreduce.reduce.java.opts：修改reducer阶段的childyard进程执行jvm参数，默认为空，当为空的时候，使用mapred.child.java.opts。

4. mapreduce.job.reduces ---> 修改reducer的个数，默认为1。可以通过job.setNumReduceTasks方法来进行更改。

5. mapreduce.map.speculative：是否启动map阶段的推测执行，默认为true。其实一般情况设置为false比较好。可通过方法job.setMapSpeculativeExecution来设置。

6. mapreduce.reduce.speculative：是否需要启动reduce阶段的推测执行，默认为true，其实一般情况设置为fase比较好。可通过方法job.setReduceSpeculativeExecution来设置。

7. mapreduce.map.output.compress --->设置是否启动map输出的压缩机制，默认为false。在需要减少网络传输的时候，可以设置为true。

8. mapreduce.map.output.compress.codec --->设置map输出压缩机制，默认为org.apache.hadoop.io.compress.DefaultCodec，推荐使用SnappyCodec(在之前版本中需要进行安装操作

9. hbase参数设置

由于hbase默认是一条一条数据拿取的，在mapper节点上执行的时候是每处理一条数据后就从hbase中获取下一条数据，通过设置cache值可以一次获取多条数据，减少网络数据传输。

#### 8 总结

参考文档：

<https://blog.csdn.net/WYpersist/article/details/80699372>

目的：充分的利用机器的性能，更快的完成mr程序的计算任务。甚至是在有限的机器条件下，能够支持运行足够多的mr程序。

主要从**map端的输出**、reducer端的数据合并以及reducer的个数这三个方面的调优操作。

性能调优包括cpu、内存、磁盘io以及网络这四个大方面。从mr程序的执行流程中，我们可以知道主要有调优的是内存、磁盘io以及网络。

**1 配置文件调优**

\* mapreduce参数调优

\* Uber模式

Uber模式是Hadoop2.0中实现的一种针对MR小作业的优化机制。

即如果作业足够小，则所有task在一个jvm（mr AppMaster）中完成要比为每个task启动一个container更划算。

**2 mapper调优**

mapper调优主要就是就一个目标：减少输出量。我们可以通过增加combine阶段以及对输出进行压缩(在提交job的时候分别设置启动压缩和指定压缩方式)设置进行mapper调优。

**3 reducer调优**

reducer调优主要是通过参数调优和设置reducer的个数来完成。

**4 runner调优**

runner调优其实就是在提交job的时候设置job参数。

具体实用策略如下：

<https://blog.csdn.net/b_x_p/article/details/78459828>

### 14 MapReduce数据倾斜

参考文档：

<https://blog.csdn.net/qq_35478489/article/details/79366123>

#### 数据倾斜

就是大量的相同key被partition分配到一个分区里,造成了'一个人累死,其他人闲死'的情况，效率低下。

一种是唯一值非常少，极少数值有非常多的记录值(唯一值少于几千)

一种是唯一值比较多，这个字段的某些值有远远多于其他值的记录数，但是它的占比也小于百分之一或千分之一

#### 解决方案

1 增加jvm内存

这适用于第一种情况

2 增加reduce的个数

这适用于第二种情况

（一个reduce执行了大量的工作,而如果我们增加了reduce的个数,这种情况相对来说会减轻很多）

3 自定义分区,

这需要用户自己继承partition类,指定分区策略,这种方式**效果比较显著**。

4 重新设计key,给key增加随机数。

重新设计key,有一种方案是在map阶段时给key加上一个随机数,有了随机数的key就不会被大量的分配到同一节点(小几率),待到reduce后再把随机数去掉即可。

5 使用combinner合并

combinner是在map阶段,reduce之前的一个中间阶段在这个阶段可以选择性的把大量的相同key数据先进行一个合并,可以看做是**local reduce**,然后再交给reduce来处理,这样做的好处很多,即减轻了map端向reduce端发送的数据量（减轻了网络带宽）,也减轻了map端和reduce端中间的shuffle阶段的数据拉取数量(本地化磁盘IO速率),**推荐使用这种方法。**

## 2.3 高级Hadoop 2.X

### 1 概述讲解

\* Distributed Mode

\* 伪分布式

一台机器，运行所有的守护进程

从节点DateNode,NodeManager

\* 完全分布式

有多个从节点

DateNodes

NodeManagers

配置文件

$HADOOP/etc/hadoop/slaves

### 2 环境准备

\* 3台虚拟机

|  |  |  |  |
| --- | --- | --- | --- |
|  | hadoop1 | hadoop2 | hadoop3 |
| IP | 192.168.246.133 | 192.168.246.134 | 192.168.246.135 |
| 内存 | 2G | 1G | 1G |
| CPU | 1 | 1 | 1 |
| hosts |  |  |  |
| HDFS |  |  |  |
|  | NameNode |  |  |
|  | DateNode | DateNode | DateNode |
|  |  |  | SecondaryNamenode |
| YARN |  |  |  |
|  |  | ResourceManager |  |
|  | NodeManager | NodeManager | NodeManager |
| MapReduce |  |  |  |
|  | JobHistoryServer |  |  |
|  |  |  |  |
|  |  |  |  |

1 复制虚拟机后修改IP

# vim /etc/sysconfig/network-scripts/ifcfg-eth0

DEVICE=**eth1 ##克隆虚拟机后网卡失效，修改为eth1即可**

TYPE=Ethernet

ONBOOT=**yes**

NM\_CONTROLLED=yes

BOOTPROTO=static

IPADDR=192.168.17.25

NETMASK=255.255.255.0

GATEWAY=192.168.17.2

DNS1=114.114.114.114

DNS2=8.8.8.8

# service network restart

2 修改字符界面启动

# vim /etc/inittab

id:3:initdefault:

3 配置文件

\* hdfs

\* core-site.xml --namenode

\* hdfs-site.xml --secondarynamenode

\* slaves -- datenodes

\* yarn

\* yarn-env.sh

\* yarn-site.xml

\* mapreduce

\* mapreduce-site.xml

1 hdfs-site.xml

\* 删除副本数设置

fs.replication

\* 添加secondarynamenode配置

<property>

        <name>dfs.namenode.secondary.http-address</name>

                <value>hadoop3:50090</value>

</property>

2 修改slaves

hadoop1

hadoop2

hadoop3

3 修改yarn-site.xml

<property>

        <name>yarn.resourcemanager.hostname</name>

        <value>**hadoop2**</value>

</property>

**4 注意重设密钥**

由于是拷贝虚拟机，原本的密钥会有问题，因此，需要删除后，重新生成密钥

$ ssh-keygen -t rsa --> 一直回车

## copy 密钥到本机

$ ssh-copy-id hadoop1

$ ssh-copy-id hadoop2

$ ssh-copy-id hadoop3

4 分发文件到各个节点

\* 删除share/doc目录，无用

\* 分发

## 前提，设置免密钥登陆

scp -r ./hadoop hadop@hadoop2:/home/hadoop/app/

分别在hadoop1 hadoop2 hadoop3执行

### 3 启动测试

1 格式化

$ bin/hdfs namenode -format

2 启动

$ sbin/start-all.sh

### 4 集群测试

#### 基本测试

服务启动是否可用

\* hdfs

读写操作

\* yarn

run jar

\* mapreduce

#### 基准测试

测试集群的性能

\* hdfs

写数据

读数据

（读写的速度）

#### 监控集群

Clouder Manager

\* 部署安装集群

\* 监控集群

\* 配置同步集群

\* 预警

### 5 集群时间同步

集群时间要同步

\* 找一台机器

时间服务器

\* 所有的机器与这台服务器进行时间同步

比如：每十分钟同步一次时间

1 ## 检查是否安装ntp

$ rpm -qa|grep ntp

2 修改配置文件

step1

修改：# vim /etc/ntp.conf

## 取消注释，并修改

restrict **192.168.246.0** mask 255.255.255.0 nomodify notrap

## 注释网络时间

# 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

## 添加

server 127.127.1.0

fudge 127.127.1.0 startum 10

step2

## 编辑

$ sudo vim /etc/sysconfig/ntpd

## 添加

SYNC\_HWCLOCK=yes

step3

## ntpd启动

# service ntpd status

# service ntpd start

# chkconfig ntpd on

step4

# 设置定时任务

# crontab -e

0-59/10 \* \* \* \* /usr/sbin/ntpdate hadoop1

同时在hadoop2 hadoop3 上也执行定时任务

备注：

可以在hadoop2、hadoop3上执行

/usr/sbin/ntpdate hadoop1

手动同步

### 6 分布式协作服务框架

#### 1 单机模式安装

$ tar -zxvf zookeeper-3.4.5.tar.gz

$ mv zookeeper-3.4.5 zookeeper

step1:

/home/hadoop/app/zookeeper/conf 下修改

$ mv zoo\_sample.cfg zoo.cfg

vim zoo.cfg -- 前提，创建路径 $ mkdir -p data/zkData

dataDir=/home/hadoop/app/zookeeper/data/zkData

dataLogDir=/home/hadoop/app/zookeeper/logs

# 启动

$ bin/zkServer.sh start

[hadoop@hadoop1 conf]$ jps

2466 QuorumPeerMain

2599 Jps

启动命令行

$ bin/zkCli.sh

【报错】

常看logs.out文件

配置JAVA\_HOME时出错，导致zookeeper显示启动成功，但是实际上启动失败

QuorumPeerMain 进程不存在

#### 2 分布式安装

step1:

# 配置文件zoo.cfg

vim conf/zoo.cfg

server.1=hadoop1:2888:3888

server.2=hadoop2:2888:3888

server.3=hadoop3:2888:3888

step2:

# 配置myid

# 在/home/hadoop/app/zookeeper/data/zkData目录下

echo 1 > myid

step3:

# 文件分发

scp -r ./zookeeper hadoop@hadoop2:/home/hadoop/app/

scp -r ./zookeeper hadoop@hadoop3:/home/hadoop/app/

step4:

# 修改myid

hadoop2 -- myid --> 2

hadoop3 -- myid --> 3

step5:

启动集群：

$ bin/zkServer.sh start

# 查看状态

$ bin/zkServer.sh status

### 7 HDFS HA

HDFS

分布式存储

\*NameNode

元数据

/usr/hadoop1/mapreduce/input/spark.txt

文件名称 路径 拥有者（所属者） 所属组 权限 副本数 ....

\* DataNode

128M

Block方式进行存储

本地磁盘

## 数据存放的地址

<property>

        <name>dfs.datanode.dir</name>

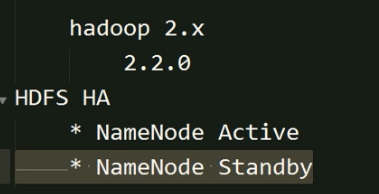
        <value>file://${hadoop.tmp.dir}/dfs/data</value>

</property>

\* Client

-> NameNode ->

万一NameNode挂了怎么办？？



\* hadop 2.x

2.2.0

HDFS HA [高可用HDFS]

\* NameNode Active

\* NameNode Standby

#### 1 背景

\* Hadoop2.0之前，在HDFS集群中NameNode存在单点故障（SPOF）.对于只有一个NameNode的集群，若NameNode机器出现故障，则整个集群无法使用，直至NameNode重启

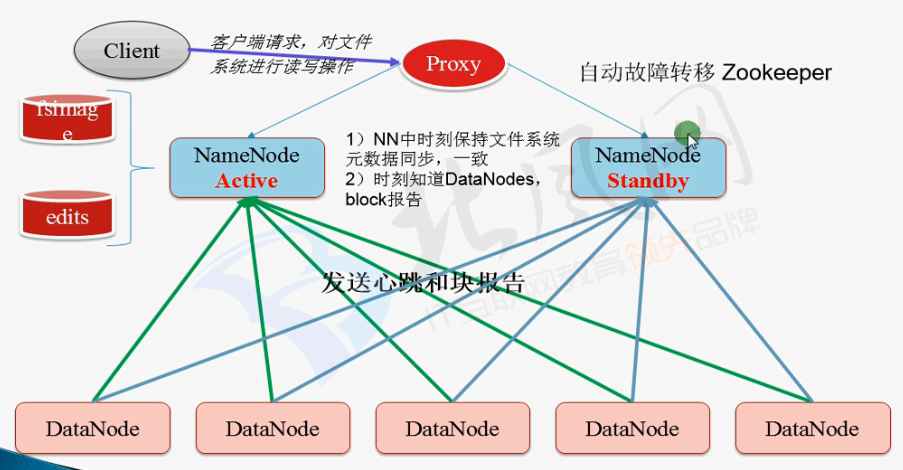
\* 影响：

NameNode机器发生意外，集群无法使用，直至管理员重启

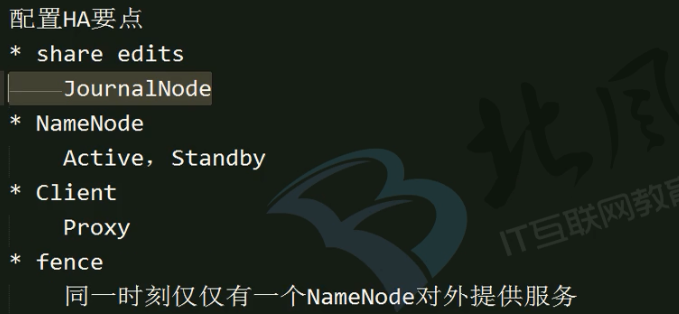
NameNode需要升级，包括软件，硬件等，此时集群无法使用

\* HDFS HA功能通过**配置Active/Standby 两个NameNode实现集群中对NameNode的热备份**来解决上述问题。

#### 2 Hadoop HA设计



### 8 配置HDFS HA



使用sshfence方式

两个NameNode之间能够无密码登录

133（NameNode）ssh --- 134

134（NameNode）ssh --- 133

## 基于分布式配置文件配置

1 备份分布式配置

hadoop1 hadoop2 hadoop3 上将 etc/hadoop --> 备份为 dist-hadoop

hadoop1 hadoop2 hadoop3 上将 data/tmp --> 备份为 dist-tmp

$ cp -r hadoop dist-hadoop

$ mv tmp dist-tmp

$ mkdir tmp

2 修改配置文件

## step1 - hdfs-site.xml

删除secondarynamenode配置

添加

<configuration>

<property>

<name>dfs.nameservices</name>

<value>ns1</value>

</property>

<property>

<name>dfs.ha.namenodes.ns1</name>

<value>nn1,nn2</value>

</property>

<!--################# NameNode RPC ADDRESS ################-->

<property>

<name>dfs.namenode.rpc-address.ns1.nn1</name>

<value>hadoop1:8020</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn2</name>

<value>hadoop2:8020</value>

</property>

<!--################# NameNode HTTP WEB ADDRESS ################-->

<property>

<name>dfs.namenode.http-address.ns1.nn1</name>

<value>hadoop1:50070</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn2</name>

<value>hadoop2:50070</value>

</property>

<!--################# NameNode shared edits ################-->

<property>

<name>dfs.namenode.shared.edits.dir</name>

<value>qjournal://hadoop1:8485;hadoop2:8485;hadoop3:8485/ns1</value>

</property>

<!--################# HDFS proxy client ################-->

<property>

<name>dfs.client.failover.proxy.provider.**ns1**</name>

<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>

</property>

<!--################# HDFS FENCE ################-->

<property>

<name>dfs.ha.fencing.methods</name>

<value>sshfence</value>

</property>

<property>

<name>dfs.ha.fencing.ssh.private-key-files</name>

<value>/home/hadoop/.ssh/id\_rsa</value>

</property>

<property>

<name>dfs.journalnode.edits.dir</name>

<value>/home/hadoop/app/hadoop/data/dfs/jn</value>

</property>

</configuration>

mkdir -p /home/hadoop/app/hadoop/data/dfs/jn

修改core-site.xml

修改fs.defaultFS配置

 <property>

        <name>fs.defaultFS</name>

                <value>hdfs://ns1</value>

 </property>

文件同步

$ scp -r etc/hadoop hadoop@hadoop2:/home/hadoop/app/hadoop/etc/

$ scp -r etc/hadoop hadoop@hadoop3:/home/hadoop/app/hadoop/etc/

### 9 HDFS HA启动

step1:多个JournalNode节点上启动journalnode服务

$ sbin/hadoop-daemon.sh start journalnode

$ jps

2824 Jps

2795 JournalNode

## 可以在logs目录下查看日志

step2:在[nn1]上，对其进行格式化，并启动

$ bin/hdfs namenode -format

$ sbin/hadoop-daemon.sh start namenode

$ jps

3043 Jps

2995 NameNode

2795 JournalNode

step3:在[nn2]上同步nn1的元数据信息 -- 在hadoop2执行

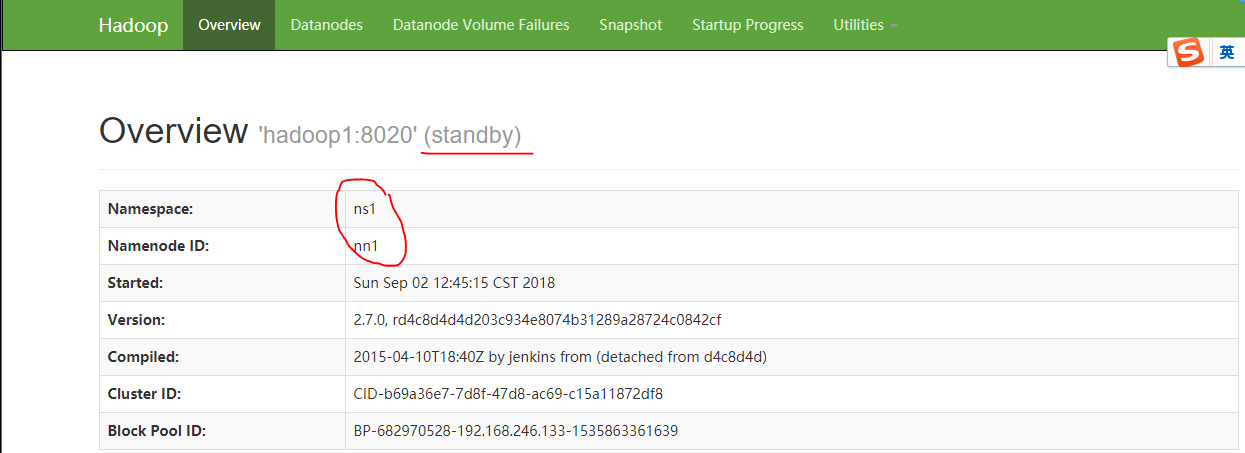
$ bin/hdfs namenode -bootstrapStandby

step4:在[nn2]上启动namenode

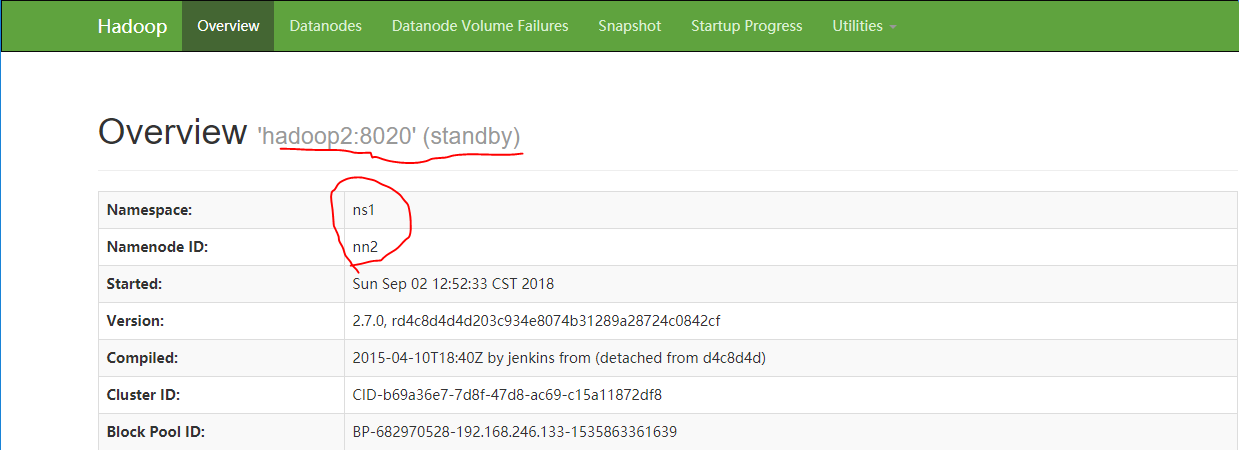
$ sbin/hadoop-daemon.sh start namenode

登录web ui界面查看

<http://hadoop1:50070>



[http://hadoop2:50070](http://hadoop1:50070)



step5: 启动datanode

-- 三台机器上分别执行

sbin/hadoop-daemon.sh start datanode

step6: haadmin命令

Usage: haadmin

[-transitionToActive [--forceactive] <serviceId>]

[-transitionToStandby <serviceId>]

[-failover [--forcefence] [--forceactive] <serviceId> <serviceId>]

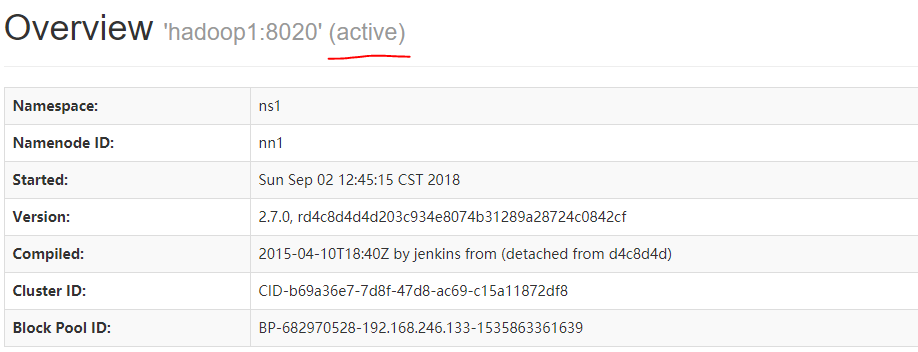
[-getServiceState <serviceId>]

[-checkHealth <serviceId>]

[-help <command>]

[hadoop@hadoop1 hadoop]$ bin/hdfs haadmin -transitionToActive nn1

[hadoop@hadoop1 hadoop]$ bin/hdfs haadmin -getServiceState nn1



## 都可以在hadoop1上执行

$ bin/hdfs haadmin -transitionToStandby nn1

$ bin/hdfs haadmin -getServiceState nn2

### 10 故障转移

HA 故障转移 Zookeeper

规划集群

1. 134 135

NameNode NameNode

ZKFC ZKFC

JournalNode JournalNode JournalNode

DataNode DataNode DataNode

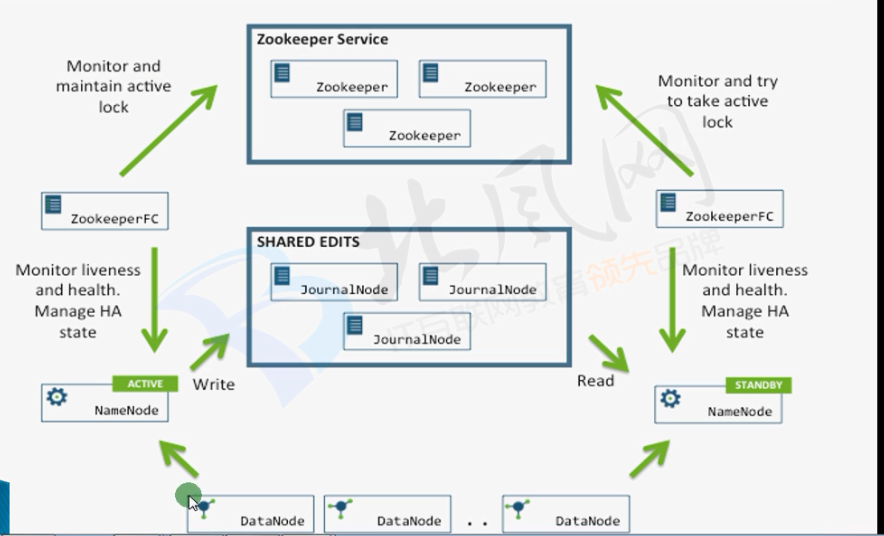
\* NameNodeq启动后都是Standby

选举一个为Active

\* 监控

ZKFC --

FailoverController



#### 配置文件

step1: hdfs-site.xml

# 添加

<!--########### automatic failover ###########-->

<property>

<name>dfs.ha.automatic-failover.enabled</name>

<value>true</value>

</property>

step2: core-site.xml

# 添加

<!--########### automatic failover ###########-->

<property>

<name>ha.zookeeper.quorum</name>

<value>hadoop1:2181,hadoop2:2181,hadoop3:2181</value>

</property>

step3: 同步文件

$ scp -r etc/hadoop/core-site.xml etc/hadoop/hdfs-site.xml hadoop@hadoop2:/home/hadoop/app/hadoop/etc/hadoop

$ scp -r etc/hadoop/core-site.xml etc/hadoop/hdfs-site.xml hadoop@hadoop2:/home/hadoop/app/hadoop/etc/hadoop

#### 启动

step1: 关闭所有HA服务

$ sbin/stop-dfs.sh

step2: 启动zookeeper集群

## 在三台机器上执行

$ bin/zkServer.sh start

stp3: 初始化HA在zookeepee中的状态

$ bin/hdfs zkfc -formatZK

step4: 启动HDFS任务

# 在hadoop1上面执行

$ sbin/start-dfs.sh

$ jps --> hadoop1

7283 NameNode

7780 DFSZKFailoverController

7077 QuorumPeerMain

7398 DataNode

7836 Jps

7599 JournalNode

[hadoop@hadoop2 ~]$ jps

5747 Jps

5430 DataNode

5240 QuorumPeerMain

5529 JournalNode

5353 NameNode

5291 **ZooKeeperMain**

5645 DFSZKFailoverController

此时查看web ui 发现 hadoop2 -- active ,hadoop1 -- standby

#### 验证

step1： 杀死进程

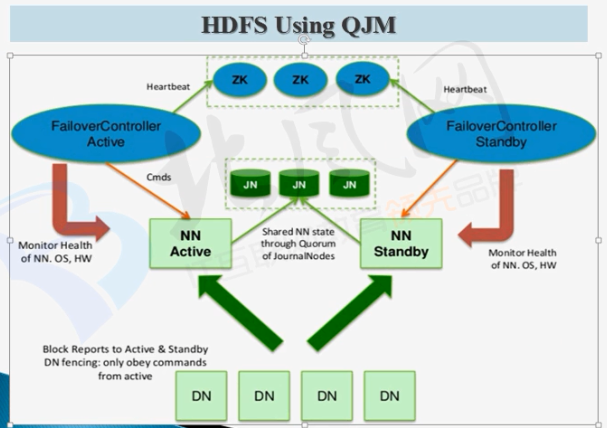
在hadoop2执行：

$ kill -9 5353

step2: 查看web ui发现hadoop1 --> active

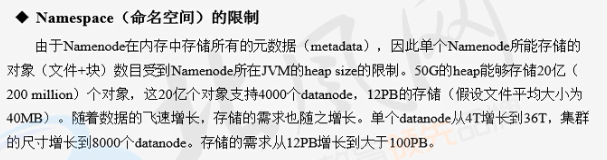
#### 说明

如果zookeeper集群挂了，HDFS集群依旧可以使用，只不过无法进行自动故障转移。



### 11 Hadoop2.x其他高级特性

NameSpace的限制



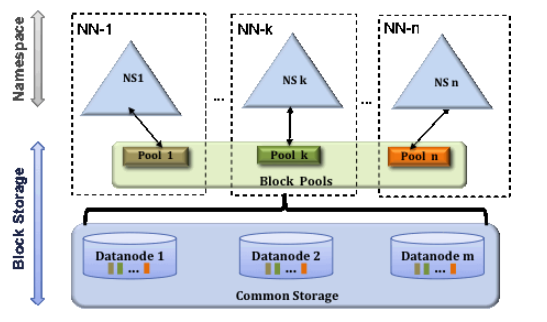
隔离问题

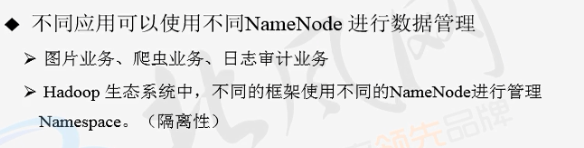
由于HDFS仅有一个NameNode,无法隔离各个程序，因此HDFS上的一个实验程序就很可能形象整个HDFS上运行的程序

性能的瓶颈

由于是单个NameNode的HDFS架构，因此整个HDFS文件系统的吞吐量受限于单个NameNode的吞吐量。

HDFS Federation[连门]





优势：

不同应用可以使用不同的NameNode进行数据管理

\* 图片业务、爬虫业务、日志审计业务

\* hadoop生态圈中，不同的框架使用不同的NameNode进行管理Nampspace(隔离性)

# 3 Hive入门

## 3.5 Hive面试题

1 求单月访问次数和总访问次数

1. 数据说明：

用户名，月份，访问次数

A,2015-01,5

A,2015-01,15

B,2015-01,5

A,2015-01,8

B,2015-01,25

A,2015-01,5

A,2015-02,4

A,2015-02,6

B,2015-02,10

B,2015-02,5

A,2015-03,16

A,2015-03,22

B,2015-03,23

B,2015-03,10

B,2015-03,1

1. 数据准备

\* 创建表

use myhive;

create external table if not exists t\_access(

uname string comment '用户名',

umonth string comment '月份',

ucount int comment '访问次数'

) comment '用户访问表'

row format delimited fields terminated by ","

location "/hive/t\_access";

\* 导入数据

load data local inpath "/home/hadoop/access.txt" into table t\_access;

# 8 Flink

https://blog.csdn.net/liguohuaBigdata/article/details/78511082