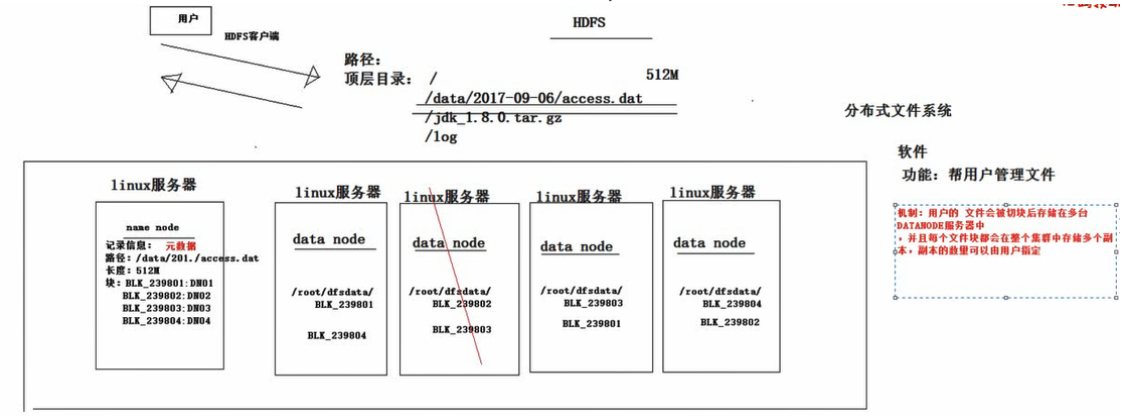
day1

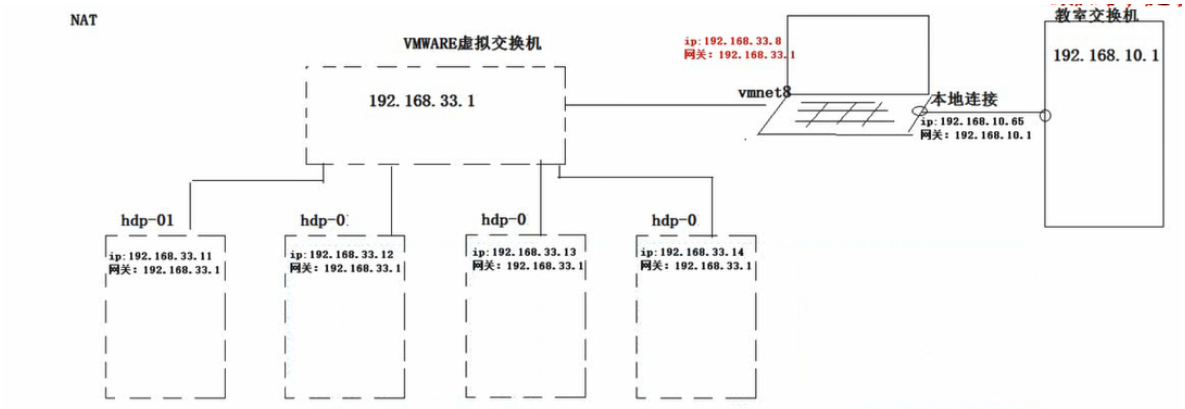
HDFS整体工作机制

hadoop的3个核心组件

- 分布式文件系统：HDFS-实现将文件分布式存储在很多服务器上
- 分布式运算变成框架：MAPREDUCE-实现在很多机器上分布式并行运算
- 分布式资源调度平台：YARN-帮助用户调度大量的mapreduce程序



服务器网络配置



虚拟机中三种不同的网络连接方式

1. bridged(桥接模式)
   在这种模式下，VMWare虚拟出来的操作系统就像是局域网中的一台独立的主机，它可以访问网内任何一台机器。在桥接模式下，你需要手工为虚拟 系统配置IP地址、子网掩码，而且还要和宿主机器处于同一网段，这样虚拟系统才能和宿主机器进行通信。同时，由于这个虚拟系统是局域网中的一个独立的主机 系统，那么就可以手工配置它的TCP/IP配置信息，以实现通过局域网的网关或路由器访问互联网。
2. host-only(主机模式)
   Host-Only模式将虚拟机与外网隔开，使得虚拟机成为一个独立的系统，只与主机相互通讯。

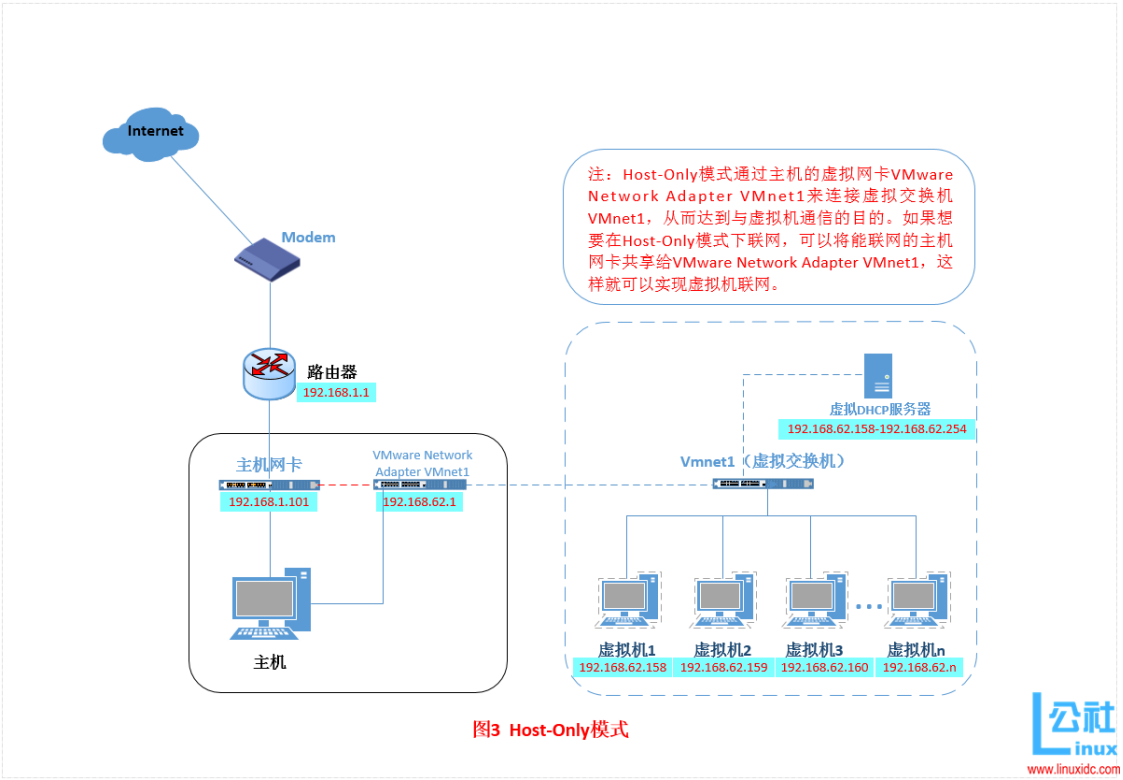


图3 Host-Only模式

如果要使得虚拟机能联网，我们可以将主机网卡共享给VMware Network Adapter VMnet1网卡，从而达到虚拟机联网的目的。

3. NAT(网络地址转换模式)
   此种方式下，虚拟机并不真实的存在于网络中，所以宿主机无法ping通虚拟机，虚拟机彼此间也不通。但是通过nat虚拟机可以访问互联网，且可以访问宿主机以及宿主机同网络中的其他主机。轻松实现上网，不占用网段中的IP地址。宿主机不能访问虚拟机，同网段中的主机无法找到虚拟机。

- virtualbox的网络连接

	桥接	NAT	host-only
虚拟机与宿主机	彼此互通，处于同一网段	虚拟机能访问宿主机；宿主机不能访问虚拟机	虚拟机不能访问宿主机；宿主机能访问虚拟机
虚拟机与虚拟机	彼此互通，处于同一网段	彼此不通	彼此互通，处于同一网段
虚拟机与其他主机	彼此互通，处于同一网段	虚拟机能访问其他主机；其他主机不能访问虚拟机	彼此不通；需要设置
虚拟机与互联网	虚拟机可以上网	虚拟机可以上网	彼此不通；需要设置

- virtualbox设置桥接模式

```
vi /etc/sysconfig/network-scripts/ifcfg-enp0s3
BOOTPROTO=static
ONBOOT=yes
IPADDR=192.168.0.10
GATEWAY=192.168.0.1
DNS1=8.8.8.8
DNS2=114.114.114.114
```

基础环境配置

解压到特定文件夹：

```
tar -zxvf jar_1.8.0.tar.gz -C /usr/java
```

jdk的安装出现如下问题：

```
/lib/ld-linux.so.2: bad ELF interpreter: No such file or directory
```

解决方案：

```
sudo yum install glibc.i686
```

添加环境变量

```
vim /etc/profile
export JAVA_HOME=/usr/java/jdk1.8.0_172
export PATH=$JAVA_HOME/bin:$PATH

//执行以下命令文件profile生效(或重启服务器reboot)
source /etc/profile
//执行这个就可以看到环境路径了
echo $PATH
```

域名映射

```
vim /etc/hosts
192.168.0.10    hdp-1
```

关闭防火墙

```
service iptables stop
chkconfig iptables off
```

安装hdfs集群

ssh免密登陆

```
ssh-keygen  //生成密钥
ssh-copy-id hdp-1  //自己也要一份密钥,因为ssh自己的时候也要输出密码
ssh-copy-id hdp-2  //给其他机器发送密钥
```

修改配置文件

核心参数:

1. 指定hadoop的默认文件系统为:hdfs
2. 指定hdfs的namenode节点为哪台机器
3. 指定namenode软件存储元数据的本地目录
4. 指定datanode软件存放文件块的本地目录

hadoop配置文件

配置hadoop目录到PATH

```
export HADOOP_HOME=/usr/hadoop/hadoop-2.9.2
export PATH=$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH  //$开始:结束
```

配置文件在：/hadoop安装目录/etc/hadoop/

1. 修改hadoop-env.sh

```
export JAVA_HOME=/usr/java/jdk1.8.0_60
```

2. 修改core-site.xml

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://hdp-1:9000/</value>
    </property>
</configuration>
```

3. 修改hdfs-site.xml

```
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/hdpdata/name</value>
    </property>

    <property>
        <name>dfs.datanode.name.dir</name>
        <value>/hdpdata/data</value>
    </property>

    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>hdp-2:50090</value>
    </property>
</configuration>
```

启动hadoop

初始化

要在hdp-1上执行hadoop的一个命令来初始化namenode的元数据存储目录

```
hadoop namenode -format
```

启动

```
hadoop-daemon.sh start namenode  //在hdp-1上
http://hdp-1:50070  //在windows中用浏览器访问namenode提供的web端口:50070
hadoop-daemon.sh start datanode  //启动datanode(在任意地方)
```

在namenode机器上启动集群(slaves方法),配置etc/hadoop/slaves文件

```
hdp-1
hdp-2
```

启动、停止命令

```
start-dfs.sh
stop-dfs.sh  //要开启ssh免秘钥,因为使用ssh的方式去启动
```

hdfs客户端的基本操作

- cat命令

```
cat a >> file.tgz
cat b >> file.tgz
```

将a追加到file文件,再将b追加到file文件。通过这个可以将hdfs的分块文件手动合并成原文件。

- 上传文件到hdfs系统

```
hadoop fs -put 本地目录 hdfs目录
```

- 从hdfs系统下载文件

```
hadoop fs -get hdfs目录 本地目录
```

- 在hdfs中创建文件夹

```
hadoop fs -mkdir -p /adir/bdir/afile
```

- 移动hdfs文件(更名)

```
hadoop fs -mv /hdfs路径 /hdfs另一个路径
```

- 删除hdfs中的文件或文件夹

```
hadoop fs -rm -r /aaa
```

- 查看目录

```
hadoop fs -ls /aaa
```

- 查看hdfs中的文本文件内容

```
hadoop fs -cat /demo.txt
hadoop fs -tail -f /demo.txt
```

- 复制hdfs中的文件到hdfs的另一目录

```
hadoop fs -cp /hdfs路径1 /hdfs路径2
```

- 查看空间

```
hadoop fs -df
```

java客户端api的基本使用

导入jar包

file -> Modules -> + -> 相关依赖

- common.jar
- common.jar的相关依赖:common/lib
- hdfs-client.jar
- hdfs-client.jar的相关依赖:hdfs/lib

查看目录的java程序

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

import java.net.URI;

public class hdfsClientDemo {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("dfs.replication", "2");
        conf.set("dfs.blocksize", "64m");
        FileSystem fs = FileSystem.get(new URI("hdfs://192.168.0.10:9000/"),
conf, "root");
        FileStatus[] listStatus = fs.listStatus(new Path("/"));
        for(FileStatus list:listStatus){
            System.out.println(list.getPath().getName());
        }
    }
}
```

Configuration参数对象的机制:构造时,会加载jar包中的默认配置xx-default.xml,再加载用户配置的xx-site.xml,覆盖掉默认参数,构造完成之后,还可以conf.set("p","v"),会再次覆盖用户配置文件中的参数值。

windows上开发hadoop程序

1. 解压hadoop编译后的文件
2. 下载对应版本的hadoop.dll和winutils.exe文件到hadoop/bin文件夹下
3. 将hadoop添加到环境变量
4. 配置hadoop中环境的hadoop-env.cmd中的JAVA_HOME

hdfs的java客户端常用操作

增、删、改、查、上传和下载等。

day2

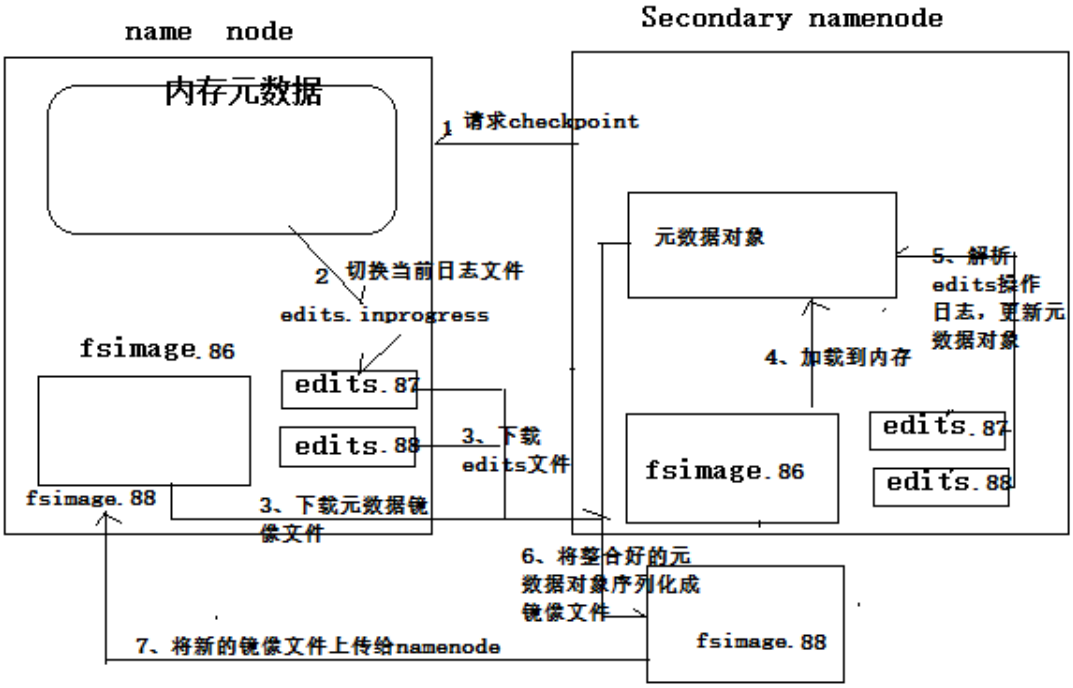
HDFS工作机制

namenode元数据管理要点

1. 什么是元数据?
   hdfs的目录结构及每一个文件的块信息(块的id,块的副本数量,块的存放位置<datanode>)
2. 元数据由谁负责管理?
   namenode
3. namenode把元数据记录在哪里?

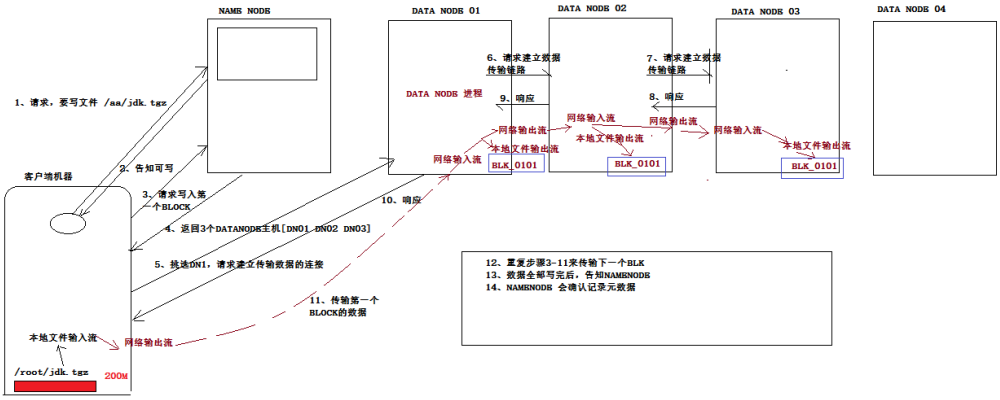
- namenode的实时的完整的元数据存储在内存中;
- namenode还会在磁盘中(dfs.namenode.name.dir)存储内存元数据在某个时间点上的镜像文件;
- namenode会把引起元数据变化的客户端操作记录在edits日志文件中;

checkpoint机制(元数据管理机制)

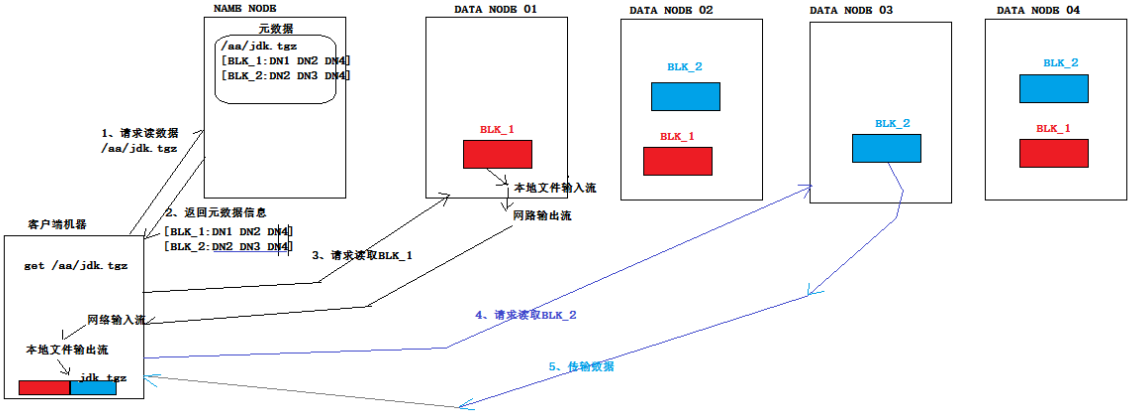


文件传输过程

写数据:

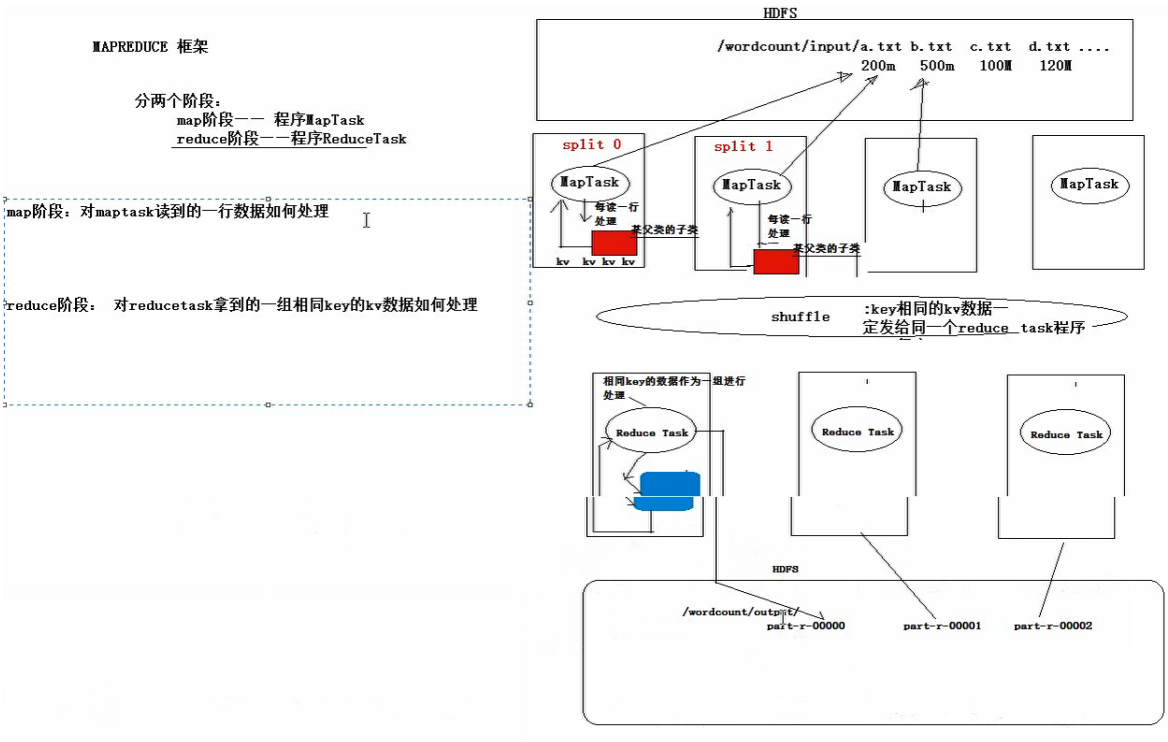


读数据:



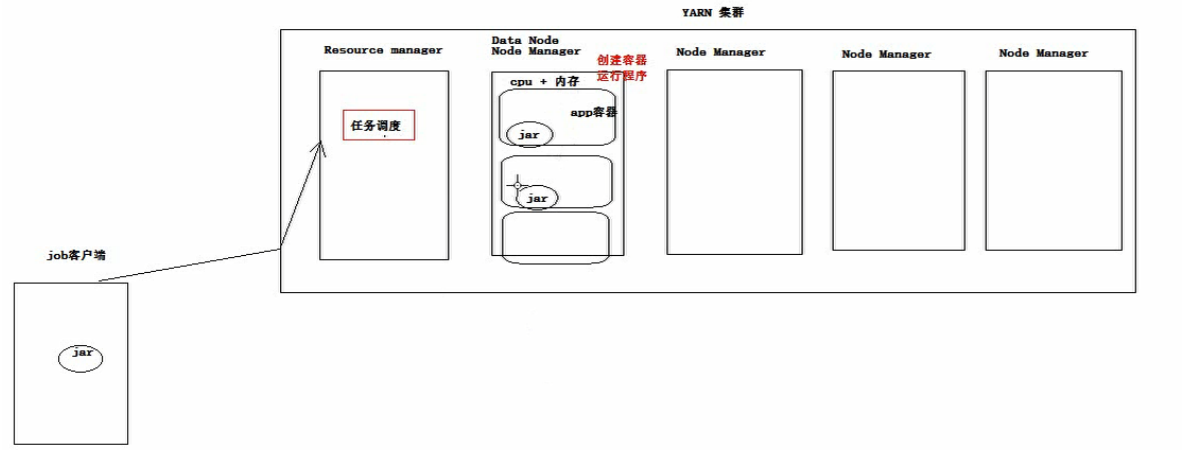
day3

mapreduce的计算框架



YARN集群

YARN集群简介



YAEN集群主要由Resource manager和Node Manager两个部分组成，Resource manager负责客户端提交的任务的调度，一般部署在单独的服务器上，宕机则客户端无法提交任务。Node manager会创建多个容器来具体处理客户端提交的任务，Node manager可以和Data Node放在一台服务器上，当处理的数据正好在该Data Node上时，可以减少数据网络传输部分的时间。

YARN集群的安装启动

修改yarn-site配置文件，指定resourcemanager的服务器，每台Node manager服务器也需要配置。

```
<configuration>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>hdp-1</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

- Nodemanager会在slaves配置文件中有的服务器上启动。
- 启动和停止yarn集群

```
start-yarn.sh
stop-yarn.sh
hdp-1:8088  //yarn集群的地址
```

注意:start-yarn不像namenode可以在任意机器启动，它都会读取配置文件去对应服务器启动namenode服务。start-yarn会在本地服务器上启动服务，因此要在配置文件写好的服务器上去启动start-yarn。

一个案例

JobSubmitter.java程序

```
package cn.edu360.mr.wc;
```

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
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;

/**
 * 用于提交mapreduce job的客户端程序
 * 功能：
 *   1、封装本次job运行时所需要的必要参数
 *   2、跟yarn进行交互，将mapreduce程序成功的启动、运行
 * @author ThinkPad
 *
 */
public class JobSubmitter {

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

        // 在代码中设置JVM系统参数，用于给job对象来获取访问HDFS的用户身份
        System.setProperty("HADOOP_USER_NAME", "root");


        Configuration conf = new Configuration();
        // 1、设置job运行时要访问的默认文件系统
        conf.set("fs.defaultFS", "hdfs://hdp-01:9000");
        // 2、设置job提交到哪去运行
        conf.set("mapreduce.framework.name", "yarn");
        conf.set("yarn.resourcemanager.hostname", "hdp-01");
        // 3、如果要从windows系统上运行这个job提交客户端程序，则需要加这个跨平台提交的参数
        conf.set("mapreduce.app-submission.cross-platform","true");

        Job job = Job.getInstance(conf);

        // 1、封装参数：jar包所在的位置
        job.setJar("d:/wc.jar");  //先要将jar包生成
        //job.setJarByClass(JobSubmitter.class);

        // 2、封装参数：  本次job所要调用的Mapper实现类、Reducer实现类
        job.setMapperClass(WordcountMapper.class);
        job.setReducerClass(WordcountReducer.class);

        // 3、封装参数：本次job的Mapper实现类、Reducer实现类产生的结果数据的key、value类
型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

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



        Path output = new Path("/wordcount/output");
```

```
        FileSystem fs = FileSystem.get(new URI("hdfs://hdp-
01:9000"),conf,"root");
        if(fs.exists(output)){
            fs.delete(output, true);
        }

        // 4、封装参数：本次job要处理的输入数据集所在路径、最终结果的输出路径
        FileInputFormat.setInputPaths(job, new Path("/wordcount/input"));
        FileOutputFormat.setOutputPath(job, output);  // 注意：输出路径必须不存在


        // 5、封装参数：想要启动的reduce task的数量
        job.setNumReduceTasks(2);

        // 6、提交job给yarn
        boolean res = job.waitForCompletion(true);

        System.exit(res?0:-1);

    }
}
```

WordcountMapper.java程序

```
package cn.edu360.mr.wc;

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;

/**
 * KEYIN ： 是map task读取到的数据的key的类型，是一行的起始偏移量Long
 * VALUEIN:是map task读取到的数据的value的类型，是一行的内容String
 *
 * KEYOUT：是用户的自定义map方法要返回的结果kv数据的key的类型，在wordcount逻辑中，我们需要
返回的是单词String
 * VALUEOUT:是用户的自定义map方法要返回的结果kv数据的value的类型，在wordcount逻辑中，我们
需要返回的是整数Integer
 *
 *
 * 但是，在mapreduce中，map产生的数据需要传输给reduce，需要进行序列化和反序列化，而jdk中的原
生序列化机制产生的数据量比较冗余，就会导致数据在mapreduce运行过程中传输效率低下
 * 所以，hadoop专门设计了自己的序列化机制，那么，mapreduce中传输的数据类型就必须实现hadoop自
己的序列化接口
 *
 * hadoop为jdk中的常用基本类型Long String Integer Float等数据类型封住了自己的实现了
hadoop序列化接口的类型：LongWritable,Text,IntWritable,FloatWritable
 *
 *
 *
 *
 * @author ThinkPad
 *
 */
```

```
public class WordcountMapper extends Mapper<LongWritable, Text, Text,
IntWritable>{

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

        // 切单词
        String line = value.toString();
        String[] words = line.split(" ");
        for(String word:words){
            context.write(new Text(word), new IntWritable(1));

        }
    }
}
```

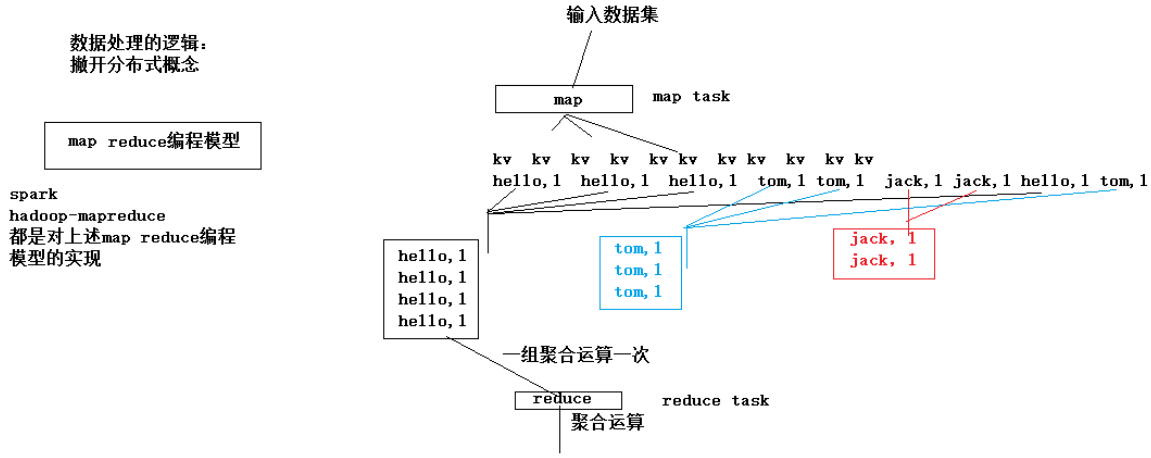
WordcountReducer.java程序

```
package cn.edu360.mr.wc;

import java.io.IOException;
import java.util.Iterator;

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

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


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


        int count = 0;

        Iterator<IntWritable> iterator = values.iterator();
        while(iterator.hasNext()){

            IntWritable value = iterator.next();
            count += value.get();
        }


        context.write(key, new IntWritable(count));

    }
}
```

注意:jobsubmitter需要在jar包在设置的路径上打包成功才能运行。

day4

mapreduce编程模型



程序提交方式1

day3的方式，在windows提交到yarn上运行。有数据传输的时间消耗。

程序提交方式2

将打包好的jar包上传到服务器,使用hadoop jar的方式运行,注意配置文件mapred-sit.xml中应该配置mapreduce.framework.name参数默认yarn上运行。

```
hadoop jar xx.jar aa  //aa是主类的名字
```

如果要在hadoop集群的某台机器上启动这个job提交客户端的话conf里面就不需要指定fs.defaultFS mapreduce.framework.name因为在集群机器上用hadoop jar xx.jar cn.edu360.mr.wc.JobSubmitter2命令来启动客户端main方法时，hadoop jar这个命令会将所在机器上的hadoop安装目录中的jar包和配置文件加入到运行时的classpath中那么，我们的客户端main方法中的new Configuration()语句就会加载classpath中的配置文件，自然就有了fs.defaultFS和mapreduce.framework.name和yarn.resourcemanager.hostname这些参数配置。

程序提交方式3

在windows本地localhost上运行(开发和调试程序)。需要windows配置好hadoop的环境。

```
conf.set("fs.defaultFS", "file:///"); //使用本地文件系统
conf.set("mapreduce.framework.name", "local");
```

运行mapreduce程序主要有两个设置:

1. 设置文件系统(hdfs或者local)
2. 设置mapreduce运行的地址(yarn或者local)

自定义类型的序列化

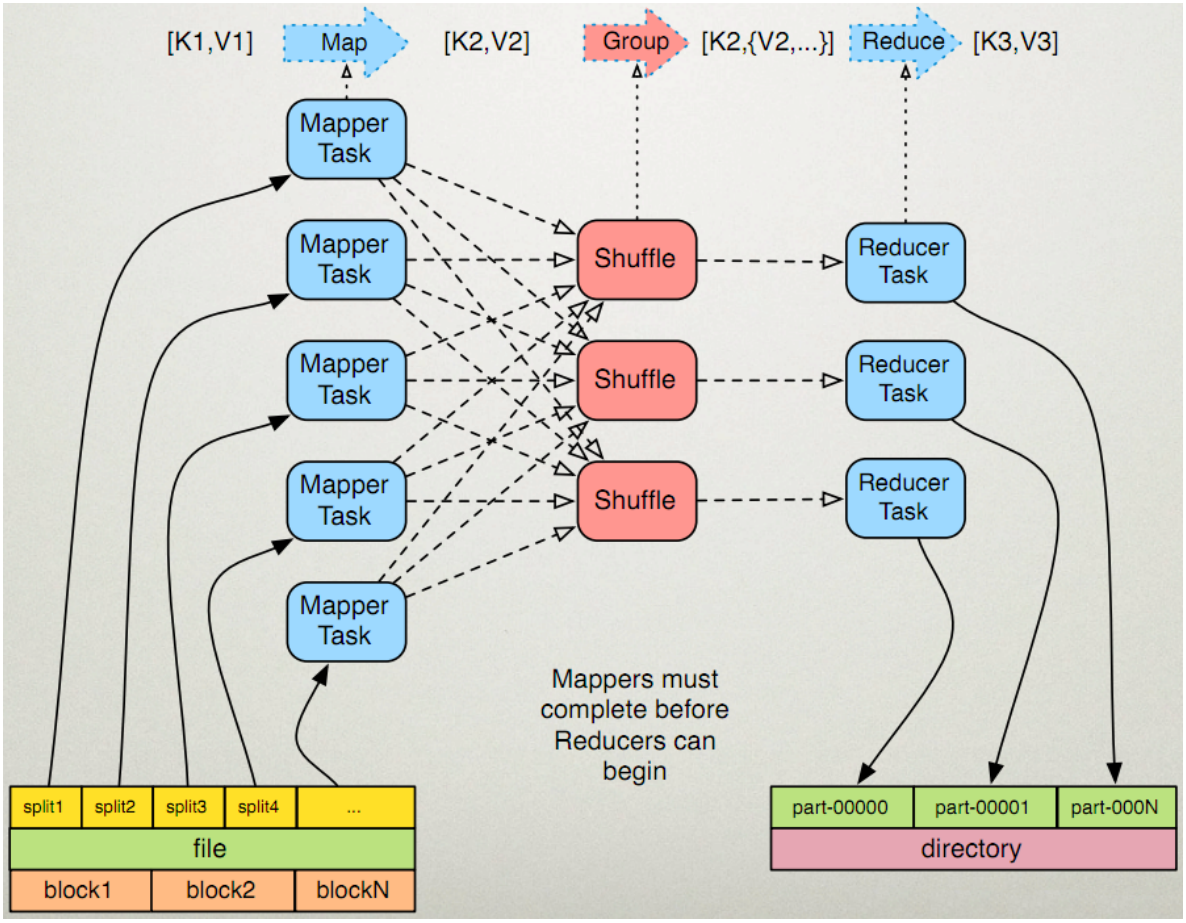
```
package cn.edu360.mr.flow;

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

```
import java.io.IOException;

import org.apache.hadoop.io.Writable;

/**
 * 本案例的功能：演示自定义数据类型如何实现hadoop的序列化接口
 * 1、该类一定要保留空参构造函数
 * 2、write方法中输出字段二进制数据的顺序　要与　readFields方法读取数据的顺序一致
 *
 * @author ThinkPad
 *
 */
public class FlowBean implements Writable {

    private int upFlow;
    private int dFlow;
    private String phone;
    private int amountFlow;

    public FlowBean(){}

    public FlowBean(String phone, int upFlow, int dFlow) {
        this.phone = phone;
        this.upFlow = upFlow;
        this.dFlow = dFlow;
        this.amountFlow = upFlow + dFlow;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public int getUpFlow() {
        return upFlow;
    }

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

    public int getdFlow() {
        return dFlow;
    }

    public void setdFlow(int dFlow) {
        this.dFlow = dFlow;
    }

    public int getAmountFlow() {
        return amountFlow;
    }

    public void setAmountFlow(int amountFlow) {
        this.amountFlow = amountFlow;
```

```
    }

    /**
     * hadoop系统在序列化该类的对象时要调用的方法
     */
    @Override
    public void write(DataOutput out) throws IOException {

        out.writeInt(upFlow);
        out.writeUTF(phone);
        out.writeInt(dFlow);
        out.writeInt(amountFlow);

    }

    /**
     * hadoop系统在反序列化该类的对象时要调用的方法
     */
    @Override
    public void readFields(DataInput in) throws IOException {
        this.upFlow = in.readInt();
        this.phone = in.readUTF();
        this.dFlow = in.readInt();
        this.amountFlow = in.readInt();
    }

    @Override
    public String toString() {

        return this.phone + ","+this.upFlow +","+ this.dFlow +"," +
this.amountFlow;
    }

}
```

day5

mapreduce原理



map的个数问题

https://www.cnblogs.com/junneyang/p/5850440.html

输入分片(input split):在进行map计算之前，mapreduce会根据输入文件计算输入分片（input split），每个输入分片（input split）针对一个map任务，输入分片（input split）存储的并非数据本身，而是一个分片长度和一个记录数据的位置的数组。

1. 默认map个数
   如果不进行任何设置，默认的map个数是和blcok_size相关的。

$$default\_num = total\_size/block\_size$$

2. 期望大小
   可以通过参数mapred.map.tasks来设置程序员期望的map个数，但是这个个数只有在大于default_num的时候，才会生效。

$$goal\_num = mapred.map.tasks$$

3. 设置处理的文件大小
   可以通过mapred.min.split.size 设置每个task处理的文件大小，但是这个大小只有在大于block_size的时候才会生效。

$$split\_size = max(mapred.min.split.size, block\_size)$$

$$split\_num = total\_size/split\_size$$

4. 计算的map个数

$$compute\_map\_num = min(split\_num, max(default\_num, goal\_num))$$

除了这些配置以外，mapreduce还要遵循一些原则。 mapreduce的每一个map处理的数据是不能跨越文件的，也就是说$min_map_num >= input_file_num$。 所以，最终的map个数应该为：

$$final\_map\_num = max(compute\_map\_num, input\_file\_num)$$
总结：

1. 如果想增加map个数，则设置mapred.map.tasks 为一个较大的值。
2. 如果想减小map个数，则设置mapred.min.split.size 为一个较大的值。
3. 如果输入中有很多小文件，依然想减少map个数，则需要将小文件merger为大文件，然后使用准则2。

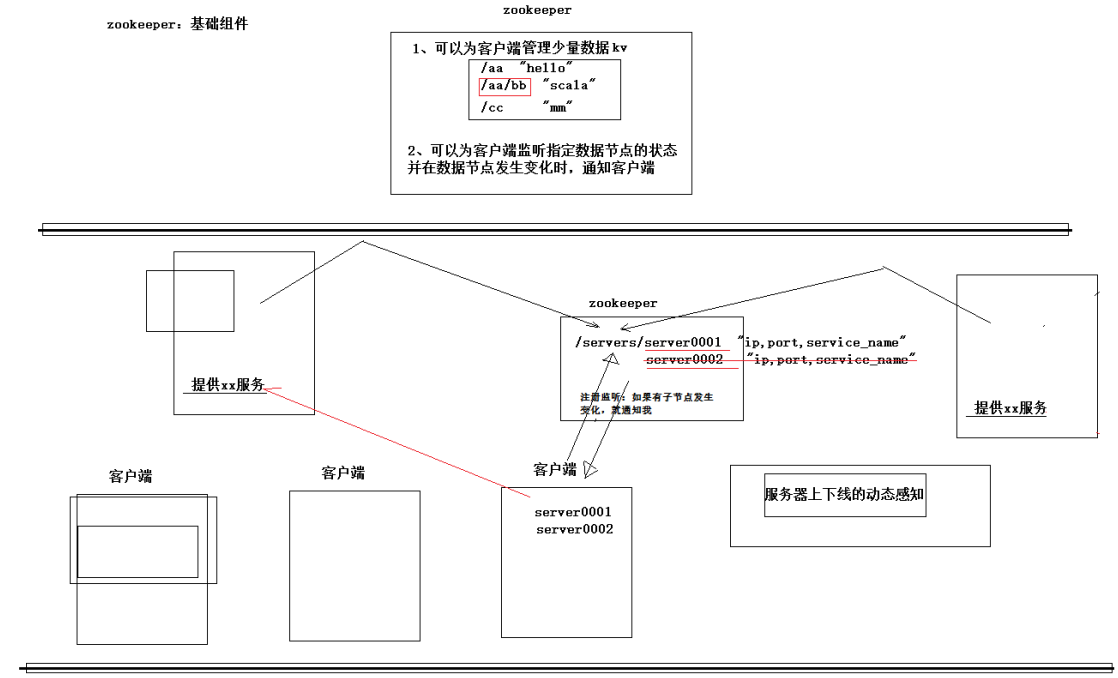
day6

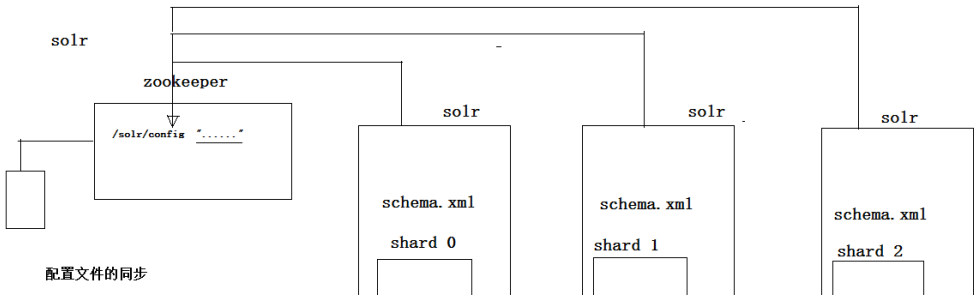
HA:高可用

引入入zookeeper

zookeeper

zookeeper的两个功能和应用场景:





1. 服务器上下线动态感知的场景
2. 配置文件同步管理的场景

zookeeper集群的搭建

1. 下载zookeeper，注意不要下成源码了。
2. 修改配置文件

```
cp zoo_sample.cfg zoo.cfg
vim zoo.cfg
dataDir=/zkdata  //指定数据存放文件夹
server.1=hdp-1:2888:3888
server.2=hdp-2:2888:3888
server.3=hdp-3:2888:3888  //指定集群
```

3. 手动创建文件夹和节点编号

```
mkdir /zkdata
echo 1 > /zkdata/myid  //第二台机器为2，第三台为3...
```

4. 启动zookeeper和查看节点状态

```
bin/zkServer.sh start
bin/zkServer.sh status
```

注意:启动第一台的时候是没有节点被选成leader的,zookeeper是不能正常的运行的，当第二个节点启动的时候，zookeeper会选第二个节点为leader，第一个节点自然就成为了follower，此时zookeeper开始正常运行。启动第三个节点时，第三个节点自然成为follower。当kill掉第二个节点时，第三个节点会被选为leader。再kill掉第三个节点，则zookeeper无法正常运行。

zookeeper常见问题

1. 为什么zookeeper的节点配置的个数必须是奇数个?
   https://blog.csdn.net/gaochao1995/article/details/39613431
   zookeeper有这样一个特性：集群中只要有过半的机器是正常工作的，那么整个集群对外就是可用的。也就是说如果有2个zookeeper，那么只要有1个死了zookeeper就不能用了，因为1没有过半，所以2个zookeeper的死亡容忍度为0；同理，要是有3个zookeeper，一个死了，还剩下2个正常的，过半了，所以3个zookeeper的容忍度为1；同理你多列举几个：2->0;3->1;4->1;5->2;6->2会发现一个规律，2n和2n-1的容忍度是一样的，都是n-1，所以为了更加高效，何必增加那一个不必要的zookeeper呢。
2. leader选举简述
   https://www.cnblogs.com/shuaiandjun/p/9383655.html
   目前有5台服务器，每台服务器均没有数据，它们的编号分别是1,2,3,4,5,按编号依次启动，它们的选择举过程如下：

- 服务器1启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器1的状态一直属于Looking(选举状态)。
- 服务器2启动，给自己投票，同时与之前启动的服务器1交换结果，由于服务器2的编号大所以服务器2胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是LOOKING。
- 服务器3启动，给自己投票，同时与之前启动的服务器1,2交换信息，由于服务器3的编号最大所以服务器3胜出，此时投票数正好大于半数，所以服务器3成为领导者，服务器1,2成为小弟。
- 服务器4启动，给自己投票，同时与之前启动的服务器1,2,3交换信息，尽管服务器4的编号大，但之前服务器3已经胜出，所以服务器4只能成为小弟。
- 服务器5启动，后面的逻辑同服务器4成为小弟。

shell脚本启动zookeeper

```
#!/bin/bash
for host in hdp-1 hdp-2 hdp-3
do
echo "${host}:${1}ing..."
ssh $host "source /etc/profile;/root/usr/zookeeper/zookeeper3.5.1/bin $1"
done
sleep 2
for host in hdp-1 hdp-2 hdp-3
do
echo "${host}:${1}ing..."
ssh $host "source /etc/profile;/root/usr/zookeeper/zookeeper3.5.1/bin status"
done
```

启动命令

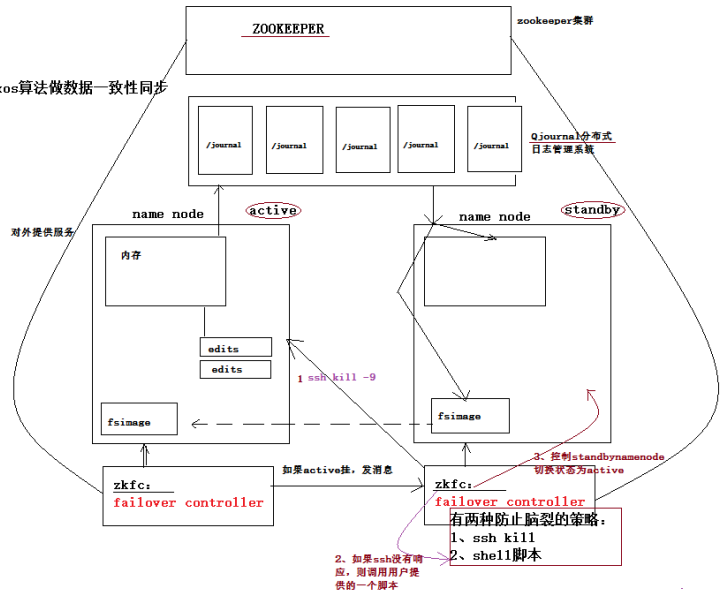
```
./zkmanage.sh start
```

day7

hadoop中HA机制

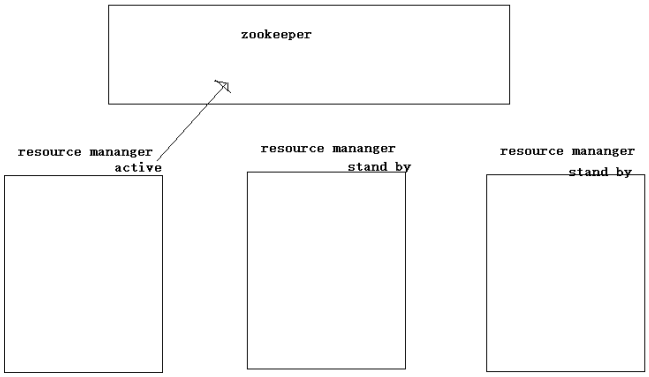
HA ：解决系统的单点故障问题

HDFS：namenode单点故障



YARN：

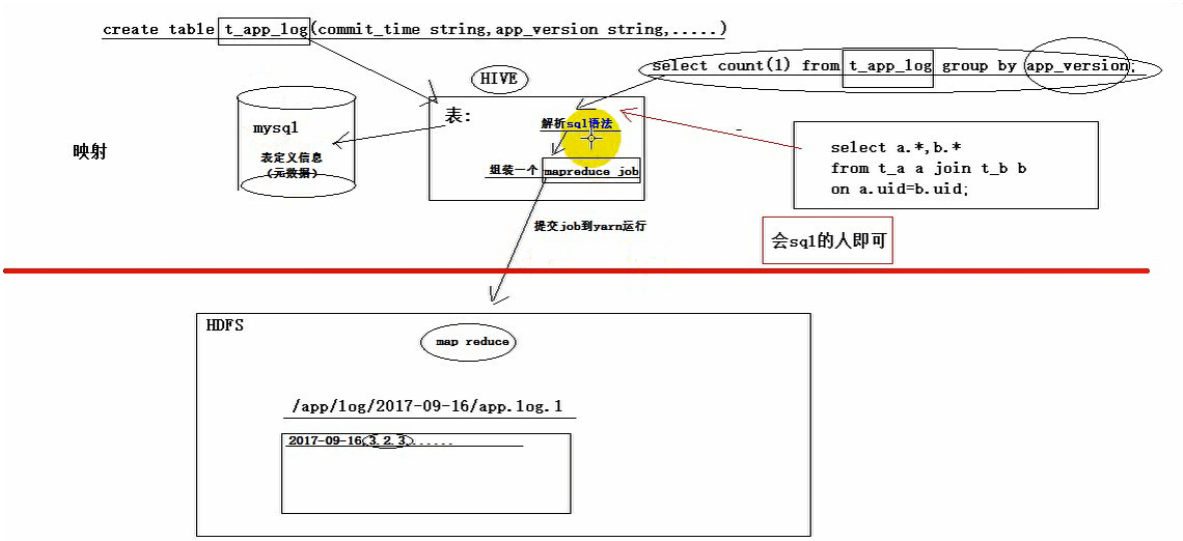
Resource manager
单点故障



---

hive

hive的基本功能和概念



- HIVE是一个可以将sql翻译为MR程序的工具。
- HIVE支持用户将HDFS上的文件映射为表结构，然后用户就可以输入SQL对这些表（HDFS上的文件）进行查询分析。
- HIVE将用户定义的库、表结构等信息存储hive的元数据库（可以是本地derby，也可以是远程mysql）中。
- HIVE看起来就像一个大的数据库
  可以建表：

1. 表定义信息会被记录到hive的元数据库中(mysql中)。
2. 会在HDFS上的hive库目录中创建一个跟表名一致的文件夹。
3. 往表目录中放入文件，表就有了数据(数据在hdfs中)。

hive的安装

将mysql作为元数据库

https://www.jianshu.com/p/b206e12d74c7

1. 下载bundle版本mysql
2. 查看linux上是否已经安装了mysql,有则卸载。

```
rpm -qa|grep mariadb
rpm -e --nodeps mariadb-libs-5.5.56-2.el7.x86_64
```

3. 安装mysql5.7所需要的依赖

```
yum install libaio
yum install perl
yum install net-tools
```

4. 解压到mysql文件夹

```
tar -xvf mysql-5.7.24-1.el7.x86_64.rpm-bundle.tar -C mysql
```

5. 安装mysql

```
rpm -ivh mysql-community-common-5.7.24-1.el7.x86_64.rpm
rpm -ivh mysql-community-libs-5.7.24-1.el7.x86_64.rpm
rpm -ivh mysql-community-client-5.7.24-1.el7.x86_64.rpm
rpm -ivh mysql-community-server-5.7.24-1.el7.x86_64.rpm
```

6. 查看mysql状态

```
service mysqld status
```

出现dead说明没有启动mysql服务。

```
service mysqld start
```

7. 设置账户密码等级和密码

```
grep password /var/log/mysqld.log  #查看临时密码
mysql -uroot -p  #使用临时密码登陆mysql -h [host] -u [username] -p [password]
set global validate_password_policy=LOW;  #设置密码等级
set global validate_password_length=6;  #设置密码长度
set password = password("123456");  #修改密码
```

8. 开启远程连接,允许远程连接数据库

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '123456' WITH GRANT
OPTION;
```

9. 配置mysql配置文件
   打开配置文件my.cnf

```
vi /etc/my.cnf
```

```
lower_case_table_names=1  #配置表名不区分大小写
character-set-server=utf8  #设置为默认编码为utf8
init_connect='SET NAMES utf8'
max_connections=1024  #设置最大连接数
```

```
service mysqld restart  #重启mysql 重启配置才能生效
```

安装hive

1. 配置hive配置文件hive-site.xml

```
<configuration>
<property>
<name>javax.jdo.option.ConnectionURL</name>
<value>jdbc:mysql://hdp-1:3306/hive?createDatabaseIfNotExist=true</value>
<description>JDBC connect string for a JDBC metastore</description>
</property>
```

```
<property>
<name>javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
<description>Driver class name for a JDBC metastore</description>
</property>

<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>root</value>
<description>username to use against metastore database</description>
</property>

<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>123456</value>
<description>password to use against metastore database</description>
</property>
</configuration>
```

2. 将mysql-connector-java.jar的驱动包传到hive的lib下。
   https://dev.mysql.com/downloads/connector/j/5.1.html

使用hive

交互式使用hive

1. 启动hdfs和yarn

```
start-dfs.sh
start-yarn.sh
```

2. 启动hive

```
bin/hive
```

启动服务端和使用客户端使用hive

1. 启动hive的服务端

```
nohup bin/hiveserver2 >> hive.log 2>&1 &
```

2. 客户端连接hive的服务器

```
bin/beeline -u jdbc:hive2://hdp-1:10000 -n root
```

3. 使用引号和文件进行hive查询

```
hive -e "select * from table_name;"  #hive可以使用一次性命令的方式来执行给定的hql语句
hive -f test.hql  #将hive写入到文件中
```

test.hql文件

```
select * from table_name;
```

hive的基本语法

建库

hive中有一个默认的库:
库名: default
库目录: hdfs://hdp-1:9000/usr/hive/warehouse
新建库:

```
create database db_order;
```

库建好后，在hdfs中会生成一个库目录:
hdfs://hdp-1:9000/usr/hive/warehouse/db_order.db

建表

```
use db_order;
create table t_order(id string, create_time, amount float, uid string);
```

建表后，会在所属的目录中生成一个表目录
/user/hive/warehouse/db_order.db/t_order
只是，这样建表的话，hive会认为表数据文件中的字段分隔符为^A

正确的建表语句为:

```
create table t_order(id string, create_time string, amount float, uid string)
row format delimited
fields terminated by ',';
```

这样就指定了，我们的表数据文件中的字段分隔符为,。

建立一个相同结构的表:

```
create table table1 like table2;
```

建立一个有相同结构并且有一定数据的表:

```
create table table1
as
select * from table2 where id>10;
```

删除表

```
drop table t_order;
```

内部表和外部表

内部表(MANAGED_TABLE):表目录按照hive的规范来部署，位于hive的仓库目录/user/hive/warehouse中。

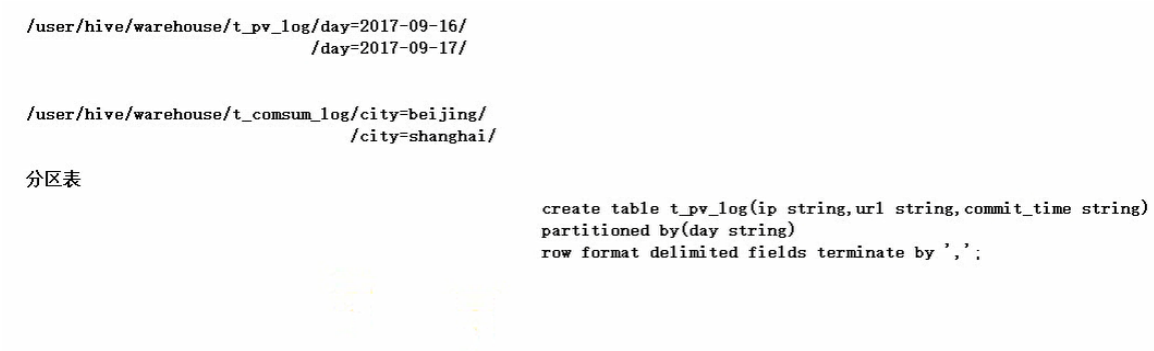
外部表(EXTERNAL_TABLE):表目录由建表用户自己制定

```
create external table t_access(ip string, url string, access_time string)
row format delimited
fields terminated by ','
location '/access/log';
```

外部表和内部表的特性差别:

1. 内部表的目录在hive的仓库目录中,外部表的目录由用户制定。
2. drop一个内部表时:hive会清除相关元数据,并删除表数据目录。
3. drop一个外部表时:hive只会清除相关元数据。

分区表



1. 创建带分区的表

```
create table t_access(ip string, url string, access_time string)
partitioned by(dt string)
row format delimited
fields terminated by ',';
```

注意:分区字段不能是表定义中已存在的字段

2. 向分区中导入数据

```
load data local inpath '/root/access1.log' into table t_access
partition(dt='20170101');
load data local inpath '/root/access2.log' into table t_access
partition(dt='20170102');
```

导入数据

1. 手动用hdfs命令,将文件放入表目录;
2. 在hive的交互式shell中用hive命令来导入本地数据到表目录

```
hive>load data local inpath '/data/order.dat' into table t_order;
```

3. 用hive命令导入hdfs中的数据文件到表目录

```
hive>load data inpath '/access.log' into table t_access;
```

注意:导本地文件和导hdfs文件的区别:
本地文件导入表:复制
hdfs文件导入表:移动

联接表

创建数据

```
create table t_a(name string,numb int)
row format delimited
fields terminated by ',';

create table t_b(name string,nick string)
row format delimited
fields terminated by ',';

load data local inpath '/root/hivetest/a.txt' into table t_a;
load data local inpath '/root/hivetest/b.txt' into table t_b;
```

1. 内连接
   笛卡尔积

```
select a.*,b.*
from t_a a inner join t_b b;
```

指定join条件

```
select a.*,b.*
from
t_a a join t_b b on a.name=b.name;
```

2. 左外连接（左连接）

```
select a.*,b.*
from
t_a a left outer join t_b b on a.name=b.name;
```

3. 右外连接（右连接）

```
select a.*,b.*
from
t_a a right outer join t_b b on a.name=b.name;
```

4. 全外连接

```
select a.*,b.*
from
t_a a full outer join t_b b on a.name=b.name;
```

5. 左半连接

```
select a.*
from
t_a a left semi join t_b b on a.name=b.name;
```

hive数据类型

数字类型

TINYINT (1-byte signed integer, from -128 to 127)
SMALLINT (2-byte signed integer, from -32,768 to 32,767)
INT/INTEGER (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)
BIGINT (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
FLOAT (4-byte single precision floating point number)
DOUBLE (8-byte double precision floating point number)

日期时间类型

TIMESTAMP
DATE

字符串类型

STRING
VARCHAR
CHAR

混杂类型

BOOLEAN
BINARY (Note: Only available starting with Hive 0.8.0)

复合类型

1. array数组类型

arrays: ARRAY<data_type> (Note: negative values and non-constant expressions are allowed as of Hive 0.14.)
file文件

```
战狼2,吴京:吴刚:龙母,2017-08-16
三生三世十里桃花,刘亦菲:痒痒,2017-08-20
```

```
create table t_movie(moive_name string,actors array<string>,first_show date)
row format delimited fields terminated by ','
collection items terminated by ':';
```

常用查询:

```
select * from t_movie;
select moive_name,actors[0] from t_movie;
select moive_name,actors from t_movie where array_contains(actors,'吴刚');
select moive_name,size(actors) from t_movie;
```

2. map类型

maps: MAP<primitive_type, data_type> (Note: negative values and non-constant expressions are allowed as of Hive 0.14.)
file文件

```
1,zhangsan,father:xiaoming#mother:xiaohuang#brother:xiaoxu,28
2,lisi,father:mayun#mother:huangyi#brother:guanyu,22
3,wangwu,father:wangjianlin#mother:ruhua#sister:jingtian,29
4,mayun,father:mayongzhen#mother:angelababy,26
```

```
create table t_person(id int,name string,family_members map<string,string>,age
int)
row format delimited fields terminated by ','
collection items terminated by '#'
map keys terminated by ':';
```

常用查询:

```
select * from t_person;
## 取map字段的指定key的值
select id,name,family_members['father'] as father from t_person;

## 取map字段的所有key
select id,name,map_keys(family_members) as relation from t_person;

## 取map字段的所有value
select id,name,map_values(family_members) from t_person;
select id,name,map_values(family_members)[0] from t_person;
```

3. struct类型(结构类型)

structs: STRUCT<col_name : data_type, ...>
file文件:

```
1,zhangsan,18:male:beijing
2,lisi,28:female:shanghai
```

```
create table t_person_struct(id int,name string,info
struct<age:int,sex:string,addr:string>)
row format delimited fields terminated by ','
collection items terminated by ':';
```

常用查询:

```
select * from t_person_struct;
select id,name,info.age from t_person_struct;
```

常用内置函数

```
# 类型转换函数
select cast("5" as int) from dual;
select cast("2017-08-03" as date) ;
select cast(current_timestamp as date);
```

```
# 数学运算函数
select round(5.4) from dual;   ## 5
select round(5.1345,3) from dual;  ##5.135
select ceil(5.4) from dual; // select ceiling(5.4) from dual;   ## 6
select floor(5.4) from dual;  ## 5
select abs(-5.4) from dual;  ## 5.4
select greatest(3,5,6) from dual;  ## 6
select least(3,5,6) from dual;

# 字符串函数
substr(string, int start)   ## 截取子串
concat(string A, string B...)  ## 拼接字符串
length(string A)
split(string str, string pat)
# 示例：select split("192.168.33.44",".") from dual; 错误的，因为.号是正则语法中的特定字符
select split("192.168.33.44","\\.") from dual;
upper(string str) ##转大写

# unix时间戳转字符串
from_unixtime(bigint unixtime[, string format])

# 字符串转unix时间戳
unix_timestamp(string date, string pattern)
# 示例： select unix_timestamp("2017-08-10 17:50:30");
## 将字符串转成日期date
select to_date("2017-09-17 16:58:32");
```

表生成函数

1. 行转列函数:explode()

file文件:

```
1,zhangsan,化学:物理:数学:语文
2,lisi,化学:数学:生物:生理:卫生
3,wangwu,化学:语文:英语:体育:生物
```

建表:

```
create table t_stu_subject(id int,name string,subjects array<string>)
row format delimited fields terminated by ','
collection items terminated by ':';
```

使用explode()对数组字段"炸裂"

```
0: jdbc:hive2://hdp20-04:10000> select * from t_stu_subject;
+--------------------+----------------------+-----------------------------+--+
| t_stu_subject.id   | t_stu_subject.name   |   t_stu_subject.subjects    |
+--------------------+----------------------+-----------------------------+--+
| 1                  | zhangsan             | ["化学","物理","数学","语文"]       |
| 2                  | lisi                 | ["化学","数学","生物","生理卫生"]     |
| 3                  | wangwu               | ["化学","语文","英语","体育","生物"]  |
+--------------------+----------------------+-----------------------------+--+
3 rows selected (0.095 seconds)
0: jdbc:hive2://hdp20-04:10000> select explode(subjects) from t_stu_subject;
+-------+--+
|  col  |
+-------+--+
| 化学    |
| 物理    |
| 数学    |
| 语文    |
| 化学    |
| 数学    |
| 生物    |
| 生理卫生  |
| 化学    |
| 语文    |
| 英语    |
| 体育    |
| 生物    |
+-------+--+
```

```
select distinct tmp.sub
from
(select explode(subjects) as sub from t_stu_subject) tmp;
```

2. 表生成函数lateral view

```
select id,name,tmp.sub
from t_stu_subject lateral view explode(subjects) tmp as sub;
```

```
+-----+-----------+----------+--+
| id  |   name    | tmp.sub  |
+-----+-----------+----------+--+
| 1   | zhangsan  | 化学       |
| 1   | zhangsan  | 物理       |
| 1   | zhangsan  | 数学       |
| 1   | zhangsan  | 语文       |
| 2   | lisi      | 化学       |
| 2   | lisi      | 数学       |
| 2   | lisi      | 生物       |
| 2   | lisi      | 生理卫生     |
| 3   | wangwu    | 化学       |
| 3   | wangwu    | 语文       |
| 3   | wangwu    | 英语       |
| 3   | wangwu    | 体育       |
| 3   | wangwu    | 生物       |
+-----+-----------+----------+--+
```

day8

hbase基本的概念介绍

基本结构

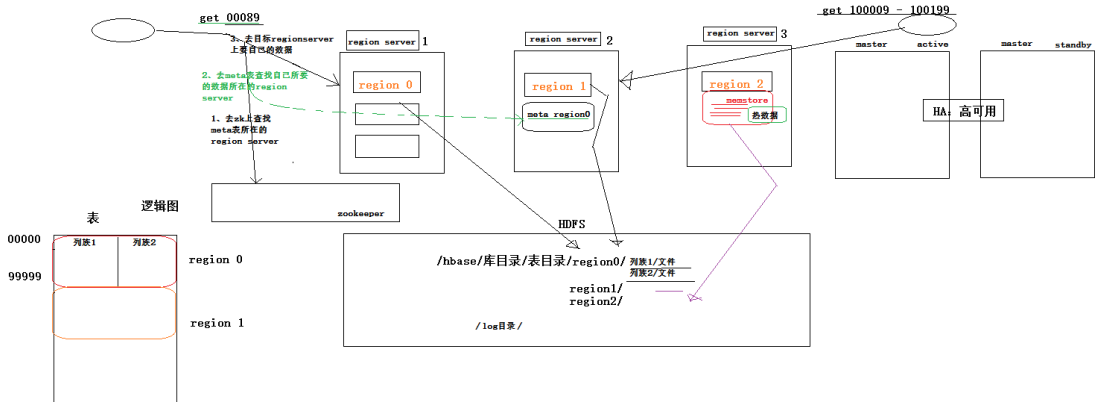
HBASE的特性：
数据的最终持久化存储是基于:HDFS-->存储容量可以随时在线扩容
HBASE的数据增删改查功能模块是：分布式系统--> hbase是一个分布式数据库系统
nosql

hbase的表结构：

表名：
行键：
列族：

rowkey: 行键	base_info 列族	extra_info 列族
001	name:zs, age:18, sex:male	hobbiy:read, addr:beijing
002	name:laowang, sex:male (cell)	
003	name:yangying v.1 (同一个数据可以保留多个版本) angelababy v.2 yangdama v.3	

hbase的整体工作机制(集群角色)



hbase集群的搭建

HBASE是一个分布式系统
其中有一个管理角色： HMaster(一般2台，一台active，一台backup)
其他的数据节点角色： HRegionServer(很多台，看数据容量)

1. 安装准备
   首先，要有一个HDFS集群，并正常运行； regionserver应该跟hdfs中的datanode在一起
   其次，还需要一个zookeeper集群，并正常运行
   然后，安装HBASE
   角色分配如下：
   hdp-1: namenode datanode regionserver hmaster zookeeper
   hdp-2: datanode regionserver zookeeper
   hdp-3: datanode regionserver zookeeper
2. 安装hbase
   修改hbase-env.sh

```
export JAVA_HOME=/root/apps/jdk1.7.0_67
export HBASE_MANAGES_ZK=false
```

修改hbase-site.xml

```
<configuration>
    <!-- 指定hbase在HDFS上存储的路径 -->
    <property>
```

```
        <name>hbase.rootdir</name>
        <value>hdfs://hdp-1:9000/hbase</value>
    </property>
    <!-- 指定hbase是分布式的 -->
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
    <!-- 指定zk的地址，多个用“,”分割 -->
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>hdp-1:2181,hdp-2:2181,hdp-3:2181</value>
    </property>
</configuration>
```

修改 regionservers

```
hdp-1
hdp-2
hdp-3
```

3. 启动hbase

```
bin/start-hbase.sh
```

启动完后，还可以在集群中找任意一台机器启动一个备用的master

```
bin/hbase-daemon.sh start master
```

新启的这个master会处于backup状态

4. 启动hbase的命令行客户端

```
bin/hbase shell
Hbase> list     // 查看表
Hbase> status   // 查看集群状态
Hbase> version  // 查看集群版本
```

hbase表模型的要点

1. 一个表，有表名
2. 一个表可以分为多个列族（不同列族的数据会存储在不同文件中）
3. 表中的每一行有一个“行键rowkey”，而且行键在表中不能重复
4. 表中的每一对kv数据称作一个cell
5. hbase可以对数据存储多个历史版本（历史版本数量可配置）
6. 整张表由于数据量过大，会被横向切分成若干个region（用rowkey范围标识），不同region的数据也存储在不同文件中
7. hbase会对插入的数据按顺序存储:

- 首先会按行键排序
- 同一行里面的kv会按列族排序，再按k排序