Day7 3月6日

软件51 庞建业 2151601012

HDFS文件系统操作学习

基本原理

（1）分布式文件系统，它所管理的文件是被切块存储在若干台datanode服务器上.（2）hdfs提供了一个统一的目录树来定位hdfs中的文件，客户端访问文件时只要指定目录树的路径即可，不用关心文件的具体物理位置.（3）每一个文件的每一个切块，在hdfs集群中都可以保存多个备份（默认 3 份），在hdfs-site.xml中，dfs.replication的value的数量就是备份的数量.（4）hdfs中有一个关键进程服务进程：namenode，它维护了一个hdfs的目录树及hdfs目录结构与文件真实存储位置的映射关系（元数据）.而datanode服务进程专门负责接收和管理 " 文件块 " - ｂｌｏｃｋ.默认大小为１２８M(可配置),(dfs.blocksize).（老版本的hadoop的默认block是64M的）

常用操作

hdfs的根目录,通过这条命令可以查看hdfs存储的文件，在hadoop的安装目录下面:

```
bin/hdfs dfs -ls /

hdfs dfs -chmod 600 /test.txt        //设置文件权限

bin/hdfs dfs -df -h /        //查看磁盘空间

bin/hdfs dfs -mkdir /aaa   //新建文件夹

bin/hdfs dfs  -tail             //查看文件尾部
```

两个块通过合并可以形成一个新的文件，然后可以通过解压进行验证

```
touch full hadoop.tar.gz
cat blk_1073741825 >> hadoop.tar.gz
cat blk_1073741826 >> hadoop.tar.gz
tar -xvzf  hadoop.tar.gz
```

在hadoop的安装目录中，直接用命令来存取文件可以通过下面的命令：

存文件：

```
./hdfs dfs -put /home/admin1/桌面/test.txt  hdfs://localhost:9000/
```

取文件：

```
./hdfs dfs -get
hdfs://localhost:9000/test.txt
```

hdfs的权限控制：

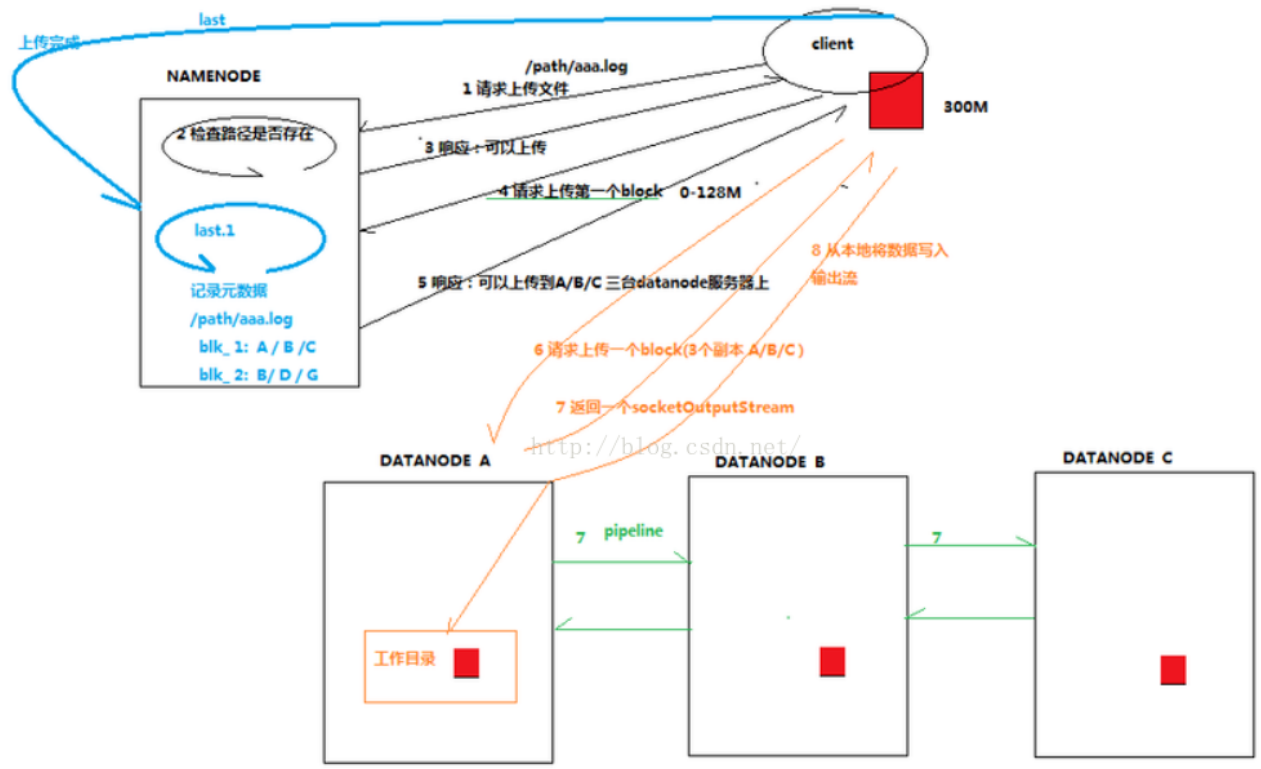
```
bin/hdfs dfs -chown  aa:bgroup  /test.txt

//将test.txt文件的权限改为aa用户的bgroup组
```

Shell操作整理

```
--appendToFile  ----追加一个文件到已经存在的文件末尾
-->hadoop fs  -appendToFile  ./hello.txt
hdfs://hadoop-server01:9000/hello.txt
可以简写为：
Hadoop fs  -appendToFile  ./hello.txt /hello.txt

-cat  ---显示文件内容
-->hadoop fs -cat /hello.txt
-chgrp
-chmod
-chown
上面三个跟linux中的用法一样

-->hadoop fs -chmod 666 /hello.txt
-copyFromLocal    #从本地文件系统中拷贝文件到hdfs路径去
-->hadoop fs  -copyFromLocal  ./jdk.tar.gz /aaa/
-copyToLocal      #从hdfs拷贝到本地
Eg:  hadoop fs -copyToLocal /aaa/jdk.tar.gz

-count          #统计一个指定目录下的文件节点数量
-->hadoop fs -count /aaa/
-cp              #从hdfs的一个路径拷贝hdfs的另一个路径

hadoop fs -cp/aaa/jdk.tar.gz /bbb/jdk.tar.gz.2
-createSnapshot
-deleteSnapshot
-renameSnapshot
以上三个用来操作hdfs文件系统目录信息快照


-->hadoop fs -createSnapshot /
```

```
-df               #统计文件系统的可用空间信息
-du
-->hadoop fs -df -h /
-->hadoop fs -du -s -h /aaa/*

-get              #等同于copyToLocal，就是从hdfs下载文件到本地
-getmerge             #合并下载多个文件
--> 比如hdfs的目录 /aaa/下有多个文件:log.1, log.2,log.3,...
hadoop fs -getmerge /aaa/log.* ./log.sum

-help             #输出这个命令参数手册
-ls                  #显示目录信息
-->hadoop fs -ls hdfs://hadoop-server01:9000/
这些参数中，所有的hdfs路径都可以简写
-->hadoop fs -ls /   等同于上一条命令的效果

-mkdir              #在hdfs上创建目录
-->hadoop fs -mkdir -p /aaa/bbb/cc/dd

-moveFromLocal            #从本地剪切粘贴到hdfs
-moveToLocal              #从hdfs剪切粘贴到本地
-mv                      #在hdfs目录中移动文件
-put                #等同于copyFromLocal
-rm                #删除文件或文件夹
--> hadoop fs -rm -r/aaa/bbb/
-rmdir                 #删除空目录

-setrep                #设置hdfs中文件的副本数量
-->hadoop fs -setrep 3 /aaa/jdk.tar.gz
-stat                  #显示一个文件或文件夹的元信息
-tail                  #显示一个文件的末尾
-text                  #以字符形式打印一个文件的内容
```

原理



hdfs写数据的流程
客户端 namenode datanodeA datanodeB
1、先请求上传文件（带路径）
2，检查路径是否存在
3、响应：可以上传
4、请求上传第一个block
5、响应、可以上传到a/b/c三台datanode服务器上

6、请求上传一个block（3个副本/a/b/c）随机选择
7、返回一个socket的输出流 8、A与B之间建立管道，pipeline，同时复制(A复制给B，B复制给C）。
9、当上传完成的时候要通知namenode
10、写元数据（记录上传的文件的block分别在哪些datanode上面）。

attention

1、如果传输过程中，有某个datanode出现了故障，那么当前的pipeline会被关闭，出现故障的datanode会从当前的pipeline中移除，剩余的block会继续剩下的datanode中继续以pipeline的形式传输，同时Namenode会分配一个新的datanode，保持replicas设定的数量。
2、关闭pipeline，将ack queue中的数据块放入data queue的开始。
3、当前的数据块在已经写入的数据节点中被元数据节点赋予新的标示，则错误节点重启后能够察觉其数据块是过时的，会被删除。
4、失败的数据节点从pipeline中移除，另外的数据块则写入pipeline中的另外两个数据节点。
5、元数据节点则被通知此数据块是复制块数不足，将来会再创建第三份备份。
6、客户端调用create()来创建文件
7、DistributedFileSystem用RPC调用元数据节点，在文件系统的命名空间中创建一个新的文件。
8、元数据节点首先确定文件原来不存在，并且客户端有创建文件的权限，然后创建新文件。
9、DistributedFileSystem返回DFSOutputStream，客户端用于写数据。
10、客户端开始写入数据，DFSOutputStream将数据分成块，写入data queue。
11、Data queue由Data Streamer读取，并通知元数据节点分配数据节点，用来存储数据块(每块默认复制3块)。分配的数据节点放在一个pipeline里。
12、Data Streamer将数据块写入pipeline中的第一个数据节点。第一个数据节点将数据块发送给第二个数据节点。第二个数据节点将数据发送给第三个数据节点。
13、DFSOutputStream为发出去的数据块保存了ack queue，等待pipeline中的数据节点告知数据已经写入成功。

Code

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.util.Progressable;


public class HDFSTest {

    public static void main(String[] args) throws Exception {
        try {
            // uploadToHdfs();
           // deleteFromHdfs();
            getDirectoryFromHdfs();
            //appendToHdfs();
            //readFromHdfs();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        finally
        {
            System.out.println("SUCCESS");
        }
    }
```

```
    /**上传文件到HDFS上去*/

    private static void uploadToHdfs() throws FileNotFoundException,IOException {
        String localSrc = "c://test//ttt.txt";
        String dst = "hdfs://192.168.31.130:9000/input/ttt.txt";
        InputStream in = new BufferedInputStream(new FileInputStream(localSrc));
        Configuration conf = new Configuration();
        // conf.set("hadoop.job.userhadoop.job.ugi", "root");
        // conf.set("hadoop.job.user", "root");
        FileSystem fs = FileSystem.get(URI.create(dst), conf);

        OutputStream out = fs.create(new Path(dst), new Progressable() {
```

```
            public void progress() {
                System.out.print(".");
            }
        });
        IOUtils.copyBytes(in, out, 4096, true);
    }
```

```
    /**从HDFS上读取文件*/
    private static void readFromHdfs() throws FileNotFoundException,IOException {
        String dst = "hdfs://192.168.31.131:9000/input/ttt.txt";
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(dst), conf);
        FSDataInputStream hdfsInStream = fs.open(new Path(dst));

        OutputStream out = new FileOutputStream("c:/test/t_fromHdfs.txt");
        byte[] ioBuffer = new byte[1024];
        int readLen = hdfsInStream.read(ioBuffer);

        while(-1 != readLen){
            out.write(ioBuffer, 0, readLen);
            readLen = hdfsInStream.read(ioBuffer);
        }
        out.close();
        hdfsInStream.close();
        fs.close();
    }
```

```
    /**以append方式将内容添加到HDFS上文件的末尾;注意：文件更新，需要在hdfs-site.xml中添<property>
<name>dfs.append.support</name><value>true</value></property>*/
    private static void appendToHdfs() throws FileNotFoundException,IOException {
        String dst = "hdfs://192.168.31.130:9000/test/ttt.txt";
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(dst), conf);
        FSDataOutputStream out = fs.append(new Path(dst));

        int readLen = "zhangzk add by hdfs java api".getBytes().length;

        while(-1 != readLen){
            out.write("zhangzk add by hdfs java api".getBytes(), 0, readLen);
        }
        out.close();
        fs.close();
    }
```

```
    /**从HDFS上删除文件*/
    private static void deleteFromHdfs() throws FileNotFoundException,IOException {
        String dst = "hdfs://192.168.31.130:9000/input/ttt.txt";
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(dst), conf);
        fs.deleteOnExit(new Path(dst));
        fs.close();
    }
```

```
    /**遍历HDFS上的文件和目录*/
    private static void getDirectoryFromHdfs() throws FileNotFoundException,IOException {
        String dst = "hdfs://192.168.31.130:9000/input/";
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(dst), conf);
        FileStatus fileList[] = fs.listStatus(new Path(dst));
        int size = fileList.length;
        for(int i = 0; i < size; i++){
            System.out.println("name:" + fileList[i].getPath().getName() + "--/t/tsize:" +
fileList[i].getLen());
        }
        fs.close();
    }
}
```

NameNode工作机制

元数据放在内存中，同时需要备份，通过日志的记录（对元数据有修改操作时）写到磁盘中，用edits_ingropress,定期dump一次，当一旦断电时，很难恢复， 所以用sercondName定期把操作日志下载和内存镜像文件，然后就定期合并操作记录，（日志+fsimage)然后生成一个新的namenode镜像文件 (fs.image.ckpt) 最后上传给namenode,然后namenode重命名为fsimage.

元数据管理机制： 1、元数据有3种存储形式：内存、edits日志、fsimage 2、最完整最新的元数据一定是内存中的这一部分

RPC

> RPC 编程是在客户机和服务器实体之间进行可靠通信的最强大、最高效的方法之一。它为在分布式计算环境中运行的几乎所有应用程序提供基础。任何 RPC 客户机-服务器程序的重要实体都包括 IDL 文件（接口定义文件）、客户机 stub、服务器 stub 以及由客户机和服务器程序共用的头文件。客户机和服务器 stub 使用 RPC 运行时库通信。RPC 运行时库提供一套标准的运行时例程来支持 RPC 应用程序。在一般的应用程序中，被调用的过程在相同的地址空间中运行，并把结果返回给发出调用的过程。在分布式环境中，客户机和服务器在不同的机器上运行，客户端调用在服务器端运行的过程，并把结果发送回客户机。这称为远程过程调用 (RPC)，是 RPC 编程的基础。

```
public interface  RPCService {
    public static final long versionID=100L;  //定义通信接口的版本号
    public String userLogin(String username,String password);  //用户名和密码
}
```

```
public class RPCServiceImpl  implements RPCService {

    @Override
    public String userLogin(String username, String password) {
        return username+"  logged in successfully!";
    }
}
```

```
public class RPCController {

    public static void main(String[] args) throws IOException {
        RPCService serverceImpl=RPC.getProxy(RPCService.class,100,new
InetSocketAddress("ubuntu2",10000),new Configuration());
//100是指前面设置的版本号，InetSocketAddress中的是hdfs主机名和10000是通信端口
        String result=serverceImpl.userLogin("aaa", "aaa");  //设置用户用户名和密码
        System.out.println(result);
    }
}
```

test

```
public class ServerRunner {

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

        Builder builder=new RPC.Builder(new Configuration());

        builder.setBindAddress("ubuntu2");  //hdfs的主机名
        builder.setPort(10000);    //设置通信端口号

        builder.setProtocol(RPCService.class);
        builder.setInstance(new RPCServiceImpl());

        Server server=builder.build();
        server.start();   //开启服务
    }
}
```