# MapReduce实验部分

## 实验一 Hadoop安装与配置实验

【实验目的】

通过本实验，学习如何在Linux环境下配置Hadoop运行环境，在本地和伪分布式下安装和配置Hadoop，以及如何运行本地/伪分布式Hadoop实例。为后续的大数据处理和分析工作做好准备。

【实验内容】

1. 安装Linux系统（建议通过虚拟机）。
2. 在Linux系统配置Hadoop基础环境。
3. 安装Java环境
4. 安装Hadoop
5. Hadoop单机配置
6. Hadoop伪分布式配置

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：Java JDK 1.8、Hadoop 3.1.3
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间用于安装Hadoop。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

内容1：请参考<https://blog.csdn.net/m0_51913750/article/details/131604868>

内容2-6：请参考<https://dblab.xmu.edu.cn/blog/2441/>

补充内容：为方便实验二操作，请补充以下环境变量，将hadoop的bin目录添加到环境变量，以方便通过hadoop、hdfs等指令直接调用

1. vim ~/.bashrc #编辑环境变量配置文件
2. # 在末尾添加以下内容后保存退出

export HADOOP\_OPTS=-Djava.library.path=$HADOOP\_HOME/lib

export PATH=$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin:$PATH

1. source ~/.bashrc # 使.bashrc文件的配置立即生效
2. hadoop version # 如果正确显示版本号，则配置成功

# MapReduce实验部分

## 实验二 HDFS的配置、启动和使用

【实验目的】

本实验旨在帮助学生深入理解HDFS在Hadoop体系结构中的角色，以及如何使用Hadoop提供的Shell命令和Java API来操作HDFS。通过此实验，学生将能够熟练地使用HDFS管理大规模数据，包括文件的存储、检索和操作。

【实验内容】

利用Hadoop提供的Shell命令或Java API完成以下任务：

1. 向HDFS中上传任意文本文件，如果指定的文件在HDFS中已经存在，由用户指定是追加到原有文件末尾还是覆盖原有的文件；
2. 从HDFS中下载指定文件，如果本地文件与要下载的文件名称相同，则自动对下载的文件重命名；
3. 将HDFS中指定文件的内容输出到终端中；
4. 显示HDFS中指定的文件的读写权限、大小、创建时间、路径等信息；
5. 给定HDFS中某一个目录，输出该目录下的所有文件的读写权限、大小、创建时间、路径等信息，如果该文件是目录，则递归输出该目录下所有文件相关信息；
6. 提供一个HDFS内的文件的路径，对该文件进行创建和删除操作。如果文件所在目录不存在，则自动创建目录；
7. 提供一个HDFS的目录的路径，对该目录进行创建和删除操作。创建目录时，如果目录文件所在目录不存在则自动创建相应目录；删除目录时，由用户指定当该目录不为空时是否还删除该目录；
8. 向HDFS中指定的文件追加内容，由用户指定内容追加到原有文件的开头或结尾；
9. 删除HDFS中指定的文件；
10. 删除HDFS中指定的目录，由用户指定目录中如果存在文件时是否删除目录；
11. 在HDFS中，将文件从源路径移动到目的路径。

【实验环境】

1. 操作系统：Linux（推荐使用Ubuntu或CentOS）
2. 软件环境：Java JDK、Hadoop
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间用于安装Hadoop。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

注意：实验中所用软件版本可能因时间不同而有所变化，建议根据实际情况选择最新版本进行安装。

【实验步骤】

HDFS使用教程：<https://dblab.xmu.edu.cn/blog/290/>

1. 向HDFS中上传任意文本文件，如果指定的文件在HDFS中已经存在，由用户指定是追加到原有文件末尾还是覆盖原有的文件；

|  |
| --- |
| Shell命令:  检查文件是否存在: hdfs dfs -test -e text.txt(执行完这一句不会输出结果，需要继续输入命令  " echo $?")  追加命令: hdfs dfs -appendToFile local.txt text.txt  覆盖命令1: hdfs dfs -copyFromLocal -f local.txt text.txt  覆盖命令2: hdfs dfs -cp -f file:///home/hadoop/local.txt text.txt  也可以使用如下命令实现：  （如下代码可视为一行代码，在终端中输入第一行代码后，直到输入 fi 才会真正执行）：  if $(hdfs dfs -test -e text.txt);  then $(hdfs dfs -appendToFile local.txt text.txt);  else $(hdfs dfs -copyFromLocal -f local.txt text.txt);  fi |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 判断路径是否存在  \*/  public static boolean test(Configuration conf, String path) throws IOException {  FileSystem fs = FileSystem.get(conf);  return fs.exists(new Path(path));  }  /\*\*  \* 复制文件到指定路径  \* 若路径已存在，则进行覆盖  \*/  public static void copyFromLocalFile(Configuration conf, String localFilePath, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path localPath = new Path(localFilePath);  Path remotePath = new Path(remoteFilePath);  /\* fs.copyFromLocalFile 第一个参数表示是否删除源文件，第二个参数表示是否覆盖 \*/  fs.copyFromLocalFile(false, true, localPath, remotePath);  fs.close();  }    /\*\*  \* 追加文件内容  \*/  public static void appendToFile(Configuration conf, String localFilePath, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  /\* 创建一个文件读入流 \*/  FileInputStream in = new FileInputStream(localFilePath);  /\* 创建一个文件输出流，输出的内容将追加到文件末尾 \*/  FSDataOutputStream out = fs.append(remotePath);  /\* 读写文件内容 \*/  byte[] data = new byte[1024];  int read = -1;  while ( (read = in.read(data)) > 0 ) {  out.write(data, 0, read);  }  out.close();  in.close();  fs.close();  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String localFilePath = "/home/hadoop/text.txt"; // 本地路径  String remoteFilePath = "/user/hadoop/text.txt"; // HDFS路径  String choice = "append"; // 若文件存在则追加到文件末尾  // String choice = "overwrite"; // 若文件存在则覆盖    try {  /\* 判断文件是否存在 \*/  Boolean fileExists = false;  if (HDFSApi.test(conf, remoteFilePath)) {  fileExists = true;  System.out.println(remoteFilePath + " 已存在.");  } else {  System.out.println(remoteFilePath + " 不存在.");  }  /\* 进行处理 \*/  if ( !fileExists) { // 文件不存在，则上传  HDFSApi.copyFromLocalFile(conf, localFilePath, remoteFilePath);  System.out.println(localFilePath + " 已上传至 " + remoteFilePath);  } else if ( choice.equals("overwrite") ) { // 选择覆盖  HDFSApi.copyFromLocalFile(conf, localFilePath, remoteFilePath);  System.out.println(localFilePath + " 已覆盖 " + remoteFilePath);  } else if ( choice.equals("append") ) { // 选择追加  HDFSApi.appendToFile(conf, localFilePath, remoteFilePath);  System.out.println(localFilePath + " 已追加至 " + remoteFilePath);  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 从HDFS中下载指定文件，如果本地文件与要下载的文件名称相同，则自动对下载的文件重命名；

|  |
| --- |
| Shell命令：  if $(hdfs dfs -test -e file:///home/hadoop/text.txt);  then $(hdfs dfs -copyToLocal text.txt ./text2.txt);  else $(hdfs dfs -copyToLocal text.txt ./text.txt);  fi |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 下载文件到本地  \* 判断本地路径是否已存在，若已存在，则自动进行重命名  \*/  public static void copyToLocal(Configuration conf, String remoteFilePath, String localFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  File f = new File(localFilePath);  /\* 如果文件名存在，自动重命名(在文件名后面加上 \_0, \_1 ...) \*/  if (f.exists()) {  System.out.println(localFilePath + " 已存在.");  Integer i = 0;  while (true) {  f = new File(localFilePath + "\_" + i.toString());  if (!f.exists()) {  localFilePath = localFilePath + "\_" + i.toString();  break;  }  }  System.out.println("将重新命名为: " + localFilePath);  }    // 下载文件到本地  Path localPath = new Path(localFilePath);  fs.copyToLocalFile(remotePath, localPath);  fs.close();  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String localFilePath = "/home/hadoop/text.txt"; // 本地路径  String remoteFilePath = "/user/hadoop/text.txt"; // HDFS路径    try {  HDFSApi.copyToLocal(conf, remoteFilePath, localFilePath);  System.out.println("下载完成");  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 将HDFS中指定文件的内容输出到终端中；

|  |
| --- |
| Shell命令：  hdfs dfs -cat text.txt |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 读取文件内容  \*/  public static void cat(Configuration conf, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  FSDataInputStream in = fs.open(remotePath);  BufferedReader d = new BufferedReader(new InputStreamReader(in));  String line = null;  while ( (line = d.readLine()) != null ) {  System.out.println(line);  }  d.close();  in.close();  fs.close();  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteFilePath = "/user/hadoop/text.txt"; // HDFS路径    try {  System.out.println("读取文件: " + remoteFilePath);  HDFSApi.cat(conf, remoteFilePath);  System.out.println("\n读取完成");  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 显示HDFS中指定的文件的读写权限、大小、创建时间、路径等信息；

|  |
| --- |
| Shell命令：  hdfs dfs -ls -h text.txt |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  import java.text.SimpleDateFormat;  public class HDFSApi {  /\*\*  \* 显示指定文件的信息  \*/  public static void ls(Configuration conf, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  FileStatus[] fileStatuses = fs.listStatus(remotePath);  for (FileStatus s : fileStatuses) {  System.out.println("路径: " + s.getPath().toString());  System.out.println("权限: " + s.getPermission().toString());  System.out.println("大小: " + s.getLen());  /\* 返回的是时间戳,转化为时间日期格式 \*/  Long timeStamp = s.getModificationTime();  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  String date = format.format(timeStamp);  System.out.println("时间: " + date);  }  fs.close();  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteFilePath = "/user/hadoop/text.txt"; // HDFS路径    try {  System.out.println("读取文件信息: " + remoteFilePath);  HDFSApi.ls(conf, remoteFilePath);  System.out.println("\n读取完成");  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 给定HDFS中某一个目录，输出该目录下的所有文件的读写权限、大小、创建时间、路径等信息，如果该文件是目录，则递归输出该目录下所有文件相关信息；

|  |
| --- |
| Shell命令：  hdfs dfs -ls -R -h /user/hadoop |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  import java.text.SimpleDateFormat;  public class HDFSApi {  /\*\*  \* 显示指定文件夹下所有文件的信息（递归）  \*/  public static void lsDir(Configuration conf, String remoteDir) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  /\* 递归获取目录下的所有文件 \*/  RemoteIterator<LocatedFileStatus> remoteIterator = fs.listFiles(dirPath, true);  /\* 输出每个文件的信息 \*/  while (remoteIterator.hasNext()) {  FileStatus s = remoteIterator.next();  System.out.println("路径: " + s.getPath().toString());  System.out.println("权限: " + s.getPermission().toString());  System.out.println("大小: " + s.getLen());  /\* 返回的是时间戳,转化为时间日期格式 \*/  Long timeStamp = s.getModificationTime();  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  String date = format.format(timeStamp);  System.out.println("时间: " + date);  System.out.println();  }  fs.close();  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteDir = "/user/hadoop"; // HDFS路径    try {  System.out.println("(递归)读取目录下所有文件的信息: " + remoteDir);  HDFSApi.lsDir(conf, remoteDir);  System.out.println("读取完成");  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 提供一个HDFS内的文件的路径，对该文件进行创建和删除操作。如果文件所在目录不存在，则自动创建目录；

|  |
| --- |
| Shell命令：  if $(hdfs dfs -test -d dir1/dir2);  then $(hdfs dfs -touchz dir1/dir2/filename);  else $(hdfs dfs -mkdir -p dir1/dir2 && hdfs dfs -touchz dir1/dir2/filename);  fi  删除文件：hdfs dfs -rm dir1/dir2/filename |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 判断路径是否存在  \*/  public static boolean test(Configuration conf, String path) throws IOException {  FileSystem fs = FileSystem.get(conf);  return fs.exists(new Path(path));  }    /\*\*  \* 创建目录  \*/  public static boolean mkdir(Configuration conf, String remoteDir) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  boolean result = fs.mkdirs(dirPath);  fs.close();  return result;  }  /\*\*  \* 创建文件  \*/  public static void touchz(Configuration conf, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  FSDataOutputStream outputStream = fs.create(remotePath);  outputStream.close();  fs.close();  }    /\*\*  \* 删除文件  \*/  public static boolean rm(Configuration conf, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  boolean result = fs.delete(remotePath, false);  fs.close();  return result;  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteFilePath = "/user/hadoop/input/text.txt"; // HDFS路径  String remoteDir = "/user/hadoop/input"; // HDFS路径对应的目录    try {  /\* 判断路径是否存在，存在则删除，否则进行创建 \*/  if ( HDFSApi.test(conf, remoteFilePath) ) {  HDFSApi.rm(conf, remoteFilePath); // 删除  System.out.println("删除路径: " + remoteFilePath);  } else {  if ( !HDFSApi.test(conf, remoteDir) ) { // 若目录不存在，则进行创建  HDFSApi.mkdir(conf, remoteDir);  System.out.println("创建文件夹: " + remoteDir);  }  HDFSApi.touchz(conf, remoteFilePath);  System.out.println("创建路径: " + remoteFilePath);  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 提供一个HDFS的目录的路径，对该目录进行创建和删除操作。创建目录时，如果目录文件所在目录不存在则自动创建相应目录；删除目录时，由用户指定当该目录不为空时是否还删除该目录；

|  |
| --- |
| Shell命令：  创建目录：hdfs dfs -mkdir -p dir1/dir2  删除目录（如果目录非空则会提示not empty，不执行删除）：hdfs dfs -rmdir dir1/dir2  强制删除目录：hdfs dfs -rm -R dir1/dir2 |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 判断路径是否存在  \*/  public static boolean test(Configuration conf, String path) throws IOException {  FileSystem fs = FileSystem.get(conf);  return fs.exists(new Path(path));  }  /\*\*  \* 判断目录是否为空  \* true: 空，false: 非空  \*/  public static boolean isDirEmpty(Configuration conf, String remoteDir) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  RemoteIterator<LocatedFileStatus> remoteIterator = fs.listFiles(dirPath, true);  return !remoteIterator.hasNext();  }    /\*\*  \* 创建目录  \*/  public static boolean mkdir(Configuration conf, String remoteDir) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  boolean result = fs.mkdirs(dirPath);  fs.close();  return result;  }    /\*\*  \* 删除目录  \*/  public static boolean rmDir(Configuration conf, String remoteDir) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  /\* 第二个参数表示是否递归删除所有文件 \*/  boolean result = fs.delete(dirPath, true);  fs.close();  return result;  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteDir = "/user/hadoop/input"; // HDFS目录  Boolean forceDelete = false; // 是否强制删除    try {  /\* 判断目录是否存在，不存在则创建，存在则删除 \*/  if ( !HDFSApi.test(conf, remoteDir) ) {  HDFSApi.mkdir(conf, remoteDir); // 创建目录  System.out.println("创建目录: " + remoteDir);  } else {  if ( HDFSApi.isDirEmpty(conf, remoteDir) || forceDelete ) { // 目录为空或强制删除  HDFSApi.rmDir(conf, remoteDir);  System.out.println("删除目录: " + remoteDir);  } else { // 目录不为空  System.out.println("目录不为空，不删除: " + remoteDir);  }  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 向HDFS中指定的文件追加内容，由用户指定内容追加到原有文件的开头或结尾；

|  |
| --- |
| Shell命令：  追加到文件末尾：hdfs dfs -appendToFile local.txt text.txt  追加到文件开头：  （由于没有直接的命令可以操作，方法之一是先移动到本地进行操作，再进行上传覆盖）：  hdfs dfs -get text.txt  cat text.txt >> local.txt  hdfs dfs -copyFromLocal -f text.txt text.txt |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 判断路径是否存在  \*/  public static boolean test(Configuration conf, String path) throws IOException {  FileSystem fs = FileSystem.get(conf);  return fs.exists(new Path(path));  }  /\*\*  \* 追加文本内容  \*/  public static void appendContentToFile(Configuration conf, String content, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  /\* 创建一个文件输出流，输出的内容将追加到文件末尾 \*/  FSDataOutputStream out = fs.append(remotePath);  out.write(content.getBytes());  out.close();  fs.close();  }  /\*\*  \* 追加文件内容  \*/  public static void appendToFile(Configuration conf, String localFilePath, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  /\* 创建一个文件读入流 \*/  FileInputStream in = new FileInputStream(localFilePath);  /\* 创建一个文件输出流，输出的内容将追加到文件末尾 \*/  FSDataOutputStream out = fs.append(remotePath);  /\* 读写文件内容 \*/  byte[] data = new byte[1024];  int read = -1;  while ( (read = in.read(data)) > 0 ) {  out.write(data, 0, read);  }  out.close();  in.close();  fs.close();  }  /\*\*  \* 移动文件到本地  \* 移动后，删除源文件  \*/  public static void moveToLocalFile(Configuration conf, String remoteFilePath, String localFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  Path localPath = new Path(localFilePath);  fs.moveToLocalFile(remotePath, localPath);  }    /\*\*  \* 创建文件  \*/  public static void touchz(Configuration conf, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  FSDataOutputStream outputStream = fs.create(remotePath);  outputStream.close();  fs.close();  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteFilePath = "/user/hadoop/text.txt"; // HDFS文件  String content = "新追加的内容\n";  String choice = "after"; //追加到文件末尾  // String choice = "before"; // 追加到文件开头    try {  /\* 判断文件是否存在 \*/  if ( !HDFSApi.test(conf, remoteFilePath) ) {  System.out.println("文件不存在: " + remoteFilePath);  } else {  if ( choice.equals("after") ) { // 追加在文件末尾  HDFSApi.appendContentToFile(conf, content, remoteFilePath);  System.out.println("已追加内容到文件末尾" + remoteFilePath);  } else if ( choice.equals("before") ) { // 追加到文件开头  /\* 没有相应的api可以直接操作，因此先把文件移动到本地，创建一个新的HDFS，再按顺序追加内容 \*/  String localTmpPath = "/user/hadoop/tmp.txt";  HDFSApi.moveToLocalFile(conf, remoteFilePath, localTmpPath); // 移动到本地  HDFSApi.touchz(conf, remoteFilePath); // 创建一个新文件  HDFSApi.appendContentToFile(conf, content, remoteFilePath); // 先写入新内容  HDFSApi.appendToFile(conf, localTmpPath, remoteFilePath); // 再写入原来内容  System.out.println("已追加内容到文件开头: " + remoteFilePath);  }  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 删除HDFS中指定的文件；

|  |
| --- |
| Shell命令：  hdfs dfs -rm text.txt |
| Java命令：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 删除文件  \*/  public static boolean rm(Configuration conf, String remoteFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path remotePath = new Path(remoteFilePath);  boolean result = fs.delete(remotePath, false);  fs.close();  return result;  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteFilePath = "/user/hadoop/text.txt"; // HDFS文件    try {  if ( HDFSApi.rm(conf, remoteFilePath) ) {  System.out.println("文件删除: " + remoteFilePath);  } else {  System.out.println("操作失败（文件不存在或删除失败）");  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 删除HDFS中指定的目录，由用户指定目录中如果存在文件时是否删除目录；

|  |
| --- |
| Shell命令：  删除目录（如果目录非空则会提示not empty，不执行删除）：hdfs dfs -rmdir dir1/dir2  强制删除目录：hdfs dfs -rm -R dir1/dir2 |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 判断目录是否为空  \* true: 空，false: 非空  \*/  public static boolean isDirEmpty(Configuration conf, String remoteDir) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  RemoteIterator<LocatedFileStatus> remoteIterator = fs.listFiles(dirPath, true);  return !remoteIterator.hasNext();  }    /\*\*  \* 删除目录  \*/  public static boolean rmDir(Configuration conf, String remoteDir, boolean recursive) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path dirPath = new Path(remoteDir);  /\* 第二个参数表示是否递归删除所有文件 \*/  boolean result = fs.delete(dirPath, recursive);  fs.close();  return result;  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteDir = "/user/hadoop/input"; // HDFS目录  Boolean forceDelete = false; // 是否强制删除    try {  if ( !HDFSApi.isDirEmpty(conf, remoteDir) && !forceDelete ) {  System.out.println("目录不为空，不删除");  } else {  if ( HDFSApi.rmDir(conf, remoteDir, forceDelete) ) {  System.out.println("目录已删除: " + remoteDir);  } else {  System.out.println("操作失败");  }  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

1. 在HDFS中，将文件从源路径移动到目的路径。

|  |
| --- |
| Shell命令：  hdfs dfs -mv text.txt text2.txt |
| Java代码：  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.\*;  import java.io.\*;  public class HDFSApi {  /\*\*  \* 移动文件  \*/  public static boolean mv(Configuration conf, String remoteFilePath, String remoteToFilePath) throws IOException {  FileSystem fs = FileSystem.get(conf);  Path srcPath = new Path(remoteFilePath);  Path dstPath = new Path(remoteToFilePath);  boolean result = fs.rename(srcPath, dstPath);  fs.close();  return result;  }    /\*\*  \* 主函数  \*/  public static void main(String[] args) {  Configuration conf = new Configuration();  conf.set("fs.default.name","hdfs://localhost:9000");  String remoteFilePath = "hdfs:///user/hadoop/text.txt"; // 源文件HDFS路径  String remoteToFilePath = "hdfs:///user/hadoop/new.txt"; // 目的HDFS路径    try {  if ( HDFSApi.mv(conf, remoteFilePath, remoteToFilePath) ) {  System.out.println("将文件 " + remoteFilePath + " 移动到 " + remoteToFilePath);  } else {  System.out.println("操作失败(源文件不存在或移动失败)");  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

# MapReduce实验部分

## 实验三 WordCount实验

【实验目的】

本实验旨在通过编写和执行基于MapReduce编程模型的WordCount程序，帮助学生深入理解MapReduce的工作原理，并学会使用Hadoop框架进行大规模数据处理。通过此实验，学生将能够掌握MapReduce编程的基本概念、编写简单的MapReduce程序以及运行它们在分布式环境中。

【实验内容】

1. 编写java程序实现WordCount，掌握Map & Reduce原理。
2. 使用命令行编译、打包Hadoop MapReduce程序。
3. 使用Eclipse编译、打包Hadoop MapReduce程序。
4. 查看程序执行结果。

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：Java JDK 1.8、Hadoop 3.1.3
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间用于安装Hadoop。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

使用命令行：编译请参考：<https://dblab.xmu.edu.cn/blog/83/>

使用Eclipse编译请参考：<https://dblab.xmu.edu.cn/blog/31/>

WordCount程序参考代码：

// 导入 Java 和 Hadoop 相关的库

import java.io.IOException;

import java.util.Iterator;

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.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.GenericOptionsParser;

public class WordCount {

public WordCount() {

}

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

// 设置 Hadoop Configuration

Configuration conf = new Configuration();

// 使用 GenericOptionsParser 获取命令行参数

String[] otherArgs = (new GenericOptionsParser(conf, args)).getRemainingArgs();

// 如果输入参数个数 <2 则返回错误提示

if(otherArgs.length < 2) {

System.err.println("Usage: wordcount <in> [<in>...] <out>");

System.exit(2);

}

// 设置 Hadoop Job

Job job = Job.getInstance(conf, "word count");

job.setJarByClass(WordCount.class);

job.setMapperClass(WordCount.TokenizerMapper.class);

job.setCombinerClass(WordCount.IntSumReducer.class);

job.setReducerClass(WordCount.IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

// 添加输入文件

for(int i = 0; i < otherArgs.length - 1; ++i) {

FileInputFormat.addInputPath(job, new Path(otherArgs[i]));

}

// 设置输出文件

FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));

// 提交任务并等待任务完成，如果成功则返回0，反之则返回1

System.exit(job.waitForCompletion(true)?0:1);

}

// 定义 SumReducer 用于计算每个单词出现的总次数

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public IntSumReducer() {

}

public void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, IntWritable, Text, IntWritable>.Context context) throws IOException, InterruptedException {

int sum = 0;

// 遍历所有IntWritable，求和得到单词的总出现次数

IntWritable val;

for(Iterator i$ = values.iterator(); i$.hasNext(); sum += val.get()) {

val = (IntWritable)i$.next();

}

// 写入结果

this.result.set(sum);

context.write(key, this.result);

}

}

// 定义 TokenizerMapper 用于将每行文本切分为单词，并输出每个单词及其出现次数（在该行文本中）

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

private static final IntWritable one = new IntWritable(1);

private Text word = new Text();

public TokenizerMapper() {

}

public void map(Object key, Text value, Mapper<Object, Text, Text, IntWritable>.Context context) throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

// 当还有更多单词时，继续获取下一个单词并输出

while(itr.hasMoreTokens()) {

this.word.set(itr.nextToken());

context.write(this.word, one);

}

}

}

}

# MapReduce实验部分

## 实验四 PageRank实验

【实验目的】

本实验旨在通过编写和执行基于MapReduce编程模型的PageRank程序，帮助学生深入理解MapReduce的工作原理，并学会使用Hadoop框架进行大规模数据处理。通过此实验，学生将能够掌握MapReduce编程的基本概念、编写简单的MapReduce程序以及运行它们在分布式环境中。

【实验内容】

实验的主要内容是在开源系统Hadoop上实现PageRank算法，进一步理解Map & Reduce原理。

PageRank算法是搜索引擎不断发展的产物，其核心思想是从许多优质的网页链接过来的网页，必定还是优质网页。为了区分网页之间的优劣，PageRank引入了一个值来评估一个网页的受欢迎程度，也就是PR值。PR值越高，说明该网页受欢迎程度越高。

算法开始设定所有网页为同一PR值，如果网页总数为N，则初始PR值一般都设置为1/N。之后通过如下公式对所有网页的PR值进行迭代计算。

https://images2017.cnblogs.com/blog/1219922/201709/1219922-20170930112525200-1983095258.png

其中，N表示网页总数，d是阻尼因子，通常设为0.85，PR(pi)表示网页pi的PR值，L(pi)表示网页pi链出网页的数目，在图论里成为出度。在有限次迭代后，所有网页的PR值会收敛到一个固定的值。当两次迭代之间PR值的改变量小于一个设定的阈值时，算法结束。

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：Java JDK 1.8、Hadoop 3.1.3
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间用于安装Hadoop。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【数据集】

SNAP-Stanford，含有281903个顶点和2312497条边：

<http://snap.stanford.edu/data/web-Stanford.html>

大家可以先构造一个小图来做程序调试。