# MapReduce实验部分

## 实验一 Hadoop与spark的安装与配置实验

【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
7. 安装Spark (选择和Hadoop集成的安装包)

【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

内容7:

1. 用浏览器访问spark.apache.org，打开documentation下面latest release页面，阅读Spark概述。下载spark-3.4.2-bin-hadoop3.tgz

2. tar解压到home目录某文件下: tar -xzvf spark-3.4.2-bin-hadoop3.tgz

3. 参考文档说明，选择单机模式(standalone deplogy mode)，spark也可以单独运行。

4. bin目录下./spark-shell运行。

若分开安装Hadoop和Spark，请参考<https://dblab.xmu.edu.cn/blog/804/>

补充内容：为方便后续实验操作

I. 安装好spark后配置以下环境变量

1. # 在.bashrc末尾添加以下内容后保存退出

export SPARK\_HOME=/path/to/spark

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

2. source ~/.bashrc # 使.bashrc文件的配置立即生效

3. hadoop version # 如果正确显示版本号，则配置成功

II. 配置Spark和Hadoop

确保Spark可以找到Hadoop的配置文件。将Hadoop的配置文件复制到Spark的conf目录下：

cp $HADOOP\_HOME/etc/hadoop/core-site.xml $SPARK\_HOME/conf/

cp $HADOOP\_HOME/etc/hadoop/hdfs-site.xml $SPARK\_HOME/conf/

# 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。

注： 每一个操作后续的scala代码需要在spark中执行且完成前面实验一对spark环境的配置。

【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 |
| scala代码 （spark\_shell）：  import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, FSDataOutputStream, Path}  import java.io.{BufferedInputStream, FileInputStream}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  val localFilePath = "/path/to/local/file.txt" // 本地路径  val hdfsFilePath = "/user/hadoop/file.txt" // HDFS路径  val choice = "append" // 可选值: "append" 或 "overwrite"  // 检查文件是否存在  if (fs.exists(new Path(hdfsFilePath))) {  if (choice == "append") {  // 追加文件内容  val inputStream = new BufferedInputStream(new FileInputStream(localFilePath))  val outputStream: FSDataOutputStream = fs.append(new Path(hdfsFilePath))  val buffer = new Array  var bytesRead = inputStream.read(buffer)  while (bytesRead > 0) {  outputStream.write(buffer, 0, bytesRead)  bytesRead = inputStream.read(buffer)  }  inputStream.close()  outputStream.close()  println(s"Content from $localFilePath appended to $hdfsFilePath.")  } else if (choice == "overwrite") {  // 覆盖文件  fs.copyFromLocalFile(new Path(localFilePath), new Path(hdfsFilePath))  println(s"File $localFilePath copied to $hdfsFilePath.")  }  } else {  // 上传文件  fs.copyFromLocalFile(new Path(localFilePath), new Path(hdfsFilePath))  println(s"File $localFilePath copied to $hdfsFilePath.")  }  // 关闭文件系统  fs.close() |

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 $(hdfs dfs -copyToLocal text.txt ./text.txt);  fi |
| scala代码 （spark\_shell）：  import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, Path}  import java.io.File  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  val remoteFilePath = "/user/hadoop/text.txt"  var localFilePath = "/path/to/local/text.txt"  // 检查本地文件是否存在并重命名  val localFile = new File(localFilePath)  if (localFile.exists()) {  println(s"$localFilePath 已存在.")  var i = 0  while (localFile.exists()) {  localFilePath = s"/path/to/local/text\_${i}.txt"  i += 1  }  println(s"将重新命名为: $localFilePath")  }  // 下载文件到本地  val remotePath = new Path(remoteFilePath)  val localPath = new Path(localFilePath)  fs.copyToLocalFile(remotePath, localPath)  println(s"File $remoteFilePath copied to $localFilePath.")  fs.close() |

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 |
| scala代码 （spark\_shell）：  import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, Path}  import java.io.{BufferedReader, InputStreamReader}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定HDFS文件路径  val remoteFilePath = "/user/hadoop/text.txt"  val remotePath = new Path(remoteFilePath)  // 打开HDFS文件并读取内容  val inputStream = fs.open(remotePath)  val reader = new BufferedReader(new InputStreamReader(inputStream))  // 输出文件内容到终端  var line: String = reader.readLine()  while (line != null) {  println(line)  line = reader.readLine()  }  // 关闭资源  reader.close()  inputStream.close()  fs.close() |

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 |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, FileStatus, Path}  import java.text.SimpleDateFormat  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定HDFS文件路径  val remoteFilePath = "/user/hadoop/text.txt"  val remotePath = new Path(remoteFilePath)  // 获取文件状态信息  val fileStatus: FileStatus = fs.getFileStatus(remotePath)  // 显示文件详细信息  val permissions = fileStatus.getPermission  val size = fileStatus.getLen  val modificationTime = fileStatus.getModificationTime  val path = fileStatus.getPath  val dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")  val modificationDate = dateFormat.format(modificationTime)  println(s"Path: $path")  println(s"Permissions: $permissions")  println(s"Size: $size bytes")  println(s"Modification Time: $modificationDate")  // 关闭文件系统  fs.close() |

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 |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, LocatedFileStatus, Path, RemoteIterator}  import java.text.SimpleDateFormat  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定HDFS目录路径  val remoteDir = "/user/hadoop"  val dirPath = new Path(remoteDir)  // 递归获取目录下的所有文件  val remoteIterator: RemoteIterator[LocatedFileStatus] = fs.listFiles(dirPath, true)  // 输出每个文件的信息  val dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")  while (remoteIterator.hasNext()) {  val fileStatus = remoteIterator.next()  val path = fileStatus.getPath.toString  val permissions = fileStatus.getPermission.toString  val size = fileStatus.getLen  val modificationTime = fileStatus.getModificationTime  val modificationDate = dateFormat.format(modificationTime)    println(s"Path: $path")  println(s"Permissions: $permissions")  println(s"Size: $size bytes")  println(s"Modification Time: $modificationDate")  println()  }  // 关闭文件系统  fs.close() |

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 |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, FSDataOutputStream, Path}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 定义文件和目录路径  val remoteFilePath = "/user/hadoop/input/text.txt"  val remoteDir = "/user/hadoop/input"  // 检查路径是否存在  def pathExists(conf: Configuration, path: String): Boolean = {  val fs = FileSystem.get(conf)  val exists = fs.exists(new Path(path))  fs.close()  exists  }  // 创建目录  def mkdir(conf: Configuration, remoteDir: String): Boolean = {  val fs = FileSystem.get(conf)  val dirPath = new Path(remoteDir)  val result = fs.mkdirs(dirPath)  fs.close()  result  }  // 创建文件  def touchz(conf: Configuration, remoteFilePath: String): Unit = {  val fs = FileSystem.get(conf)  val remotePath = new Path(remoteFilePath)  val outputStream: FSDataOutputStream = fs.create(remotePath)  outputStream.close()  fs.close()  }  // 删除文件  def rm(conf: Configuration, remoteFilePath: String): Boolean = {  val fs = FileSystem.get(conf)  val remotePath = new Path(remoteFilePath)  val result = fs.delete(remotePath, false)  fs.close()  result  }  // 创建或删除文件  if (pathExists(hadoopConf, remoteFilePath)) {  rm(hadoopConf, remoteFilePath)  println(s"文件已删除: $remoteFilePath")  } else {  if (!pathExists(hadoopConf, remoteDir)) {  mkdir(hadoopConf, remoteDir)  println(s"目录已创建: $remoteDir")  }  touchz(hadoopConf, remoteFilePath)  println(s"文件已创建: $remoteFilePath")  } |

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 |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, LocatedFileStatus, Path, RemoteIterator}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 检查路径是否存在  def pathExists(conf: Configuration, path: String): Boolean = {  val fs = FileSystem.get(conf)  val exists = fs.exists(new Path(path))  fs.close()  exists  }  // 判断目录是否为空  def isDirEmpty(conf: Configuration, remoteDir: String): Boolean = {  val fs = FileSystem.get(conf)  val dirPath = new Path(remoteDir)  val remoteIterator: RemoteIterator[LocatedFileStatus] = fs.listFiles(dirPath, true)  val isEmpty = !remoteIterator.hasNext  fs.close()  isEmpty  }  // 创建目录  def mkdir(conf: Configuration, remoteDir: String): Boolean = {  val fs = FileSystem.get(conf)  val dirPath = new Path(remoteDir)  val result = fs.mkdirs(dirPath)  fs.close()  result  }  // 删除目录  def rmDir(conf: Configuration, remoteDir: String, recursive: Boolean): Boolean = {  val fs = FileSystem.get(conf)  val dirPath = new Path(remoteDir)  val result = fs.delete(dirPath, recursive)  fs.close()  result  }  // 指定HDFS目录路径和是否强制删除  val remoteDir = "/user/hadoop/input"  val forceDelete = true // true 表示强制删除，false 表示非空时不删除  // 判断目录是否存在，不存在则创建，存在则删除  if (!pathExists(hadoopConf, remoteDir)) {  mkdir(hadoopConf, remoteDir)  println(s"创建目录: $remoteDir")  } else {  if (isDirEmpty(hadoopConf, remoteDir) || forceDelete) {  rmDir(hadoopConf, remoteDir, forceDelete)  println(s"删除目录: $remoteDir")  } else {  println(s"目录不为空，不删除: $remoteDir")  }  } |

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 |
| //追加到末尾  import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, FSDataOutputStream, Path}  import java.io.{BufferedInputStream, FileInputStream}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定HDFS文件路径  val remoteFilePath = "/user/hadoop/text.txt"  val localFilePath = "/path/to/local/file.txt"  // 追加内容到文件末尾  val remotePath = new Path(remoteFilePath)  val inputStream = new BufferedInputStream(new FileInputStream(localFilePath))  val outputStream: FSDataOutputStream = fs.append(remotePath)  val buffer = new Array  var bytesRead = inputStream.read(buffer)  while (bytesRead > 0) {  outputStream.write(buffer, 0, bytesRead)  bytesRead = inputStream.read(buffer)  }  inputStream.close()  outputStream.close()  println(s"内容已追加到文件末尾: $remoteFilePath")  // 关闭文件系统  fs.close()  //追加到开头  import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, FSDataOutputStream, Path}  import java.io.{BufferedInputStream, FileInputStream, FileOutputStream, PrintWriter}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定HDFS文件路径  val remoteFilePath = "/user/hadoop/text.txt"  val localTmpPath = "/path/to/local/tmp.txt"  val content = "新追加的内容\n"  // 判断文件是否存在  val remotePath = new Path(remoteFilePath)  if (!fs.exists(remotePath)) {  println(s"文件不存在: $remoteFilePath")  } else {  // 移动文件到本地  fs.copyToLocalFile(remotePath, new Path(localTmpPath))  fs.delete(remotePath, false)    // 创建新文件并写入新内容  val outputStream: FSDataOutputStream = fs.create(remotePath)  outputStream.write(content.getBytes)    // 追加原内容到新文件  val inputStream = new BufferedInputStream(new FileInputStream(localTmpPath))  val buffer = new Array  var bytesRead = inputStream.read(buffer)  while (bytesRead > 0) {  outputStream.write(buffer, 0, bytesRead)  bytesRead = inputStream.read(buffer)  }  inputStream.close()  outputStream.close()  println(s"内容已追加到文件开头: $remoteFilePath")  }  // 删除本地临时文件  new File(localTmpPath).delete()  // 关闭文件系统  fs.close() |

1. 删除HDFS中指定的文件；

|  |
| --- |
| Shell命令：  hdfs dfs -rm text.txt |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, Path}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定要删除的HDFS文件路径  val remoteFilePath = "/user/hadoop/text.txt"  val remotePath = new Path(remoteFilePath)  // 删除文件  val result = fs.delete(remotePath, false)  if (result) {  println(s"文件删除: $remoteFilePath")  } else {  println("操作失败（文件不存在或删除失败）")  }  // 关闭文件系统  fs.close() |

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 |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, LocatedFileStatus, Path, RemoteIterator}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 判断目录是否为空  def isDirEmpty(conf: Configuration, remoteDir: String): Boolean = {  val fs = FileSystem.get(conf)  val dirPath = new Path(remoteDir)  val remoteIterator: RemoteIterator[LocatedFileStatus] = fs.listFiles(dirPath, true)  val isEmpty = !remoteIterator.hasNext  fs.close()  isEmpty  }  // 删除目录  def rmDir(conf: Configuration, remoteDir: String, recursive: Boolean): Boolean = {  val fs = FileSystem.get(conf)  val dirPath = new Path(remoteDir)  val result = fs.delete(dirPath, recursive)  fs.close()  result  }  val remoteDir = "/user/hadoop/input" // HDFS目录  val forceDelete = true // 是否强制删除  if (!isDirEmpty(hadoopConf, remoteDir) && !forceDelete) {  println("目录不为空，不删除")  } else {  if (rmDir(hadoopConf, remoteDir, forceDelete)) {  println(s"目录已删除: $remoteDir")  } else {  println("操作失败")  }  } |

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 |
| import org.apache.hadoop.conf.Configuration  import org.apache.hadoop.fs.{FileSystem, Path}  // 创建Hadoop配置和文件系统对象  val hadoopConf = new Configuration()  hadoopConf.set("fs.defaultFS", "hdfs://localhost:9000")  val fs = FileSystem.get(hadoopConf)  // 指定源文件和目的文件路径  val remoteFilePath = "/user/hadoop/text.txt"  val remoteToFilePath = "/user/hadoop/new.txt"  val srcPath = new Path(remoteFilePath)  val dstPath = new Path(remoteToFilePath)  // 移动文件  val result = fs.rename(srcPath, dstPath)  if (result) {  println(s"文件 $remoteFilePath 已移动到 $remoteToFilePath")  } else {  println(s"操作失败（源文件不存在或移动失败）")  }  // 关闭文件系统  fs.close() |

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

【实验内容】

1. Scala程序实现WordCount，掌握RDD 和函数式编程思想。
2. 使用命令行执行WordCount程序。
3. 使用Eclipse编译、打包WordCount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、Spark3.4
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。

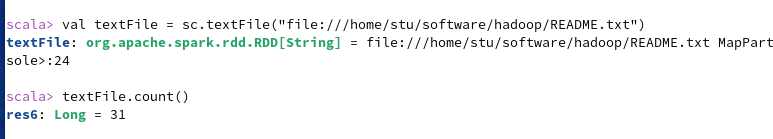
【实验步骤】

1. 使用命令行执行scale代码，注意双引号格式，在spark-shell中读取Linux系统本地文件**file:///home/stu/software/hadoop/README.txt, 统计**“Hadoop”单词的个数。

**scala> val sc = new SparkContext( )**

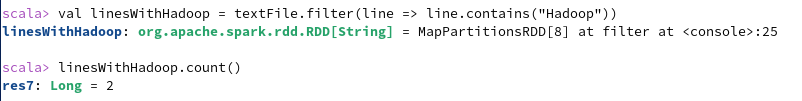
**scala> val textFile = sc.textFile(“file:///home/stu/software/hadoop/README.txt”)**

**scala> textFile.count()**

****

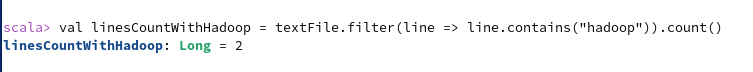
**scala> val linesWithHadoop = textFile.filter(line => line.contains(“Hadoop”))**

**scala> linesWithHadoop.count()**



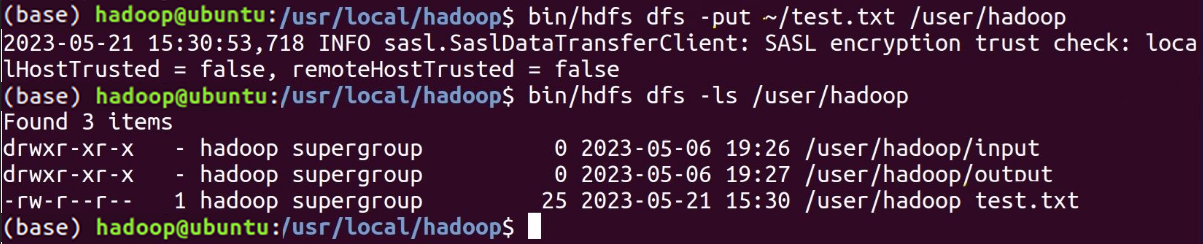
**上面两条语句可以通过链式操作合并为：**

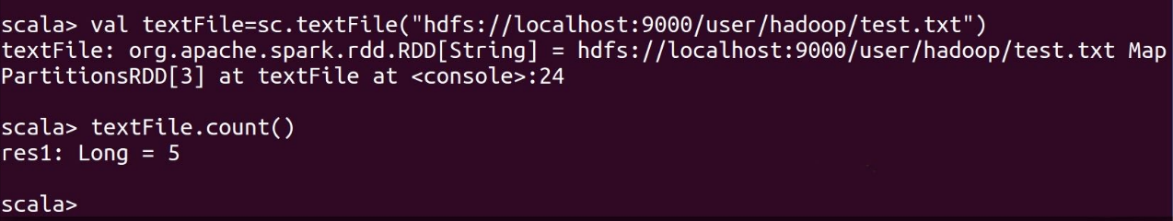
**scala> val linesCountWithHadoop = textFile.filter(line => line.contains(“Hadoop”)).count()**



1. 在spark-shell中读取HDFS系统文件“/user/hadoop/test.txt”（如果该文件不存在，请先创建），然后，统计出文件的行数；

**scala>** val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/test.txt")





1. 使用命令行来实现wordcount，即统计文件中各个单词的计数：

**scala> val sc = new SparkContext( )**

**scala> val textFile = sc.textFile(“file:///home/stu/software/hadoop/README.txt”)**

**scala> val wordCounts = textFile.flatMap(line => line.split(“ ”)).map(word => (word,1)).reduceByKey((a,b) => a+b)**

**scala> wordCounts.collect()**

**scala> wordCounts.foreach(println)**

1. 用scale语言实现wordcount

WordCount程序参考代码1：

import java.io.File

import scala.io.Source

object WordCountApp {  
  
 def main(args: Array[String]): Unit = {  
 //文件路径  
 val filePath = "file:///home/stu/software/hadoop/README.txt "  
 val codec = "utf-8"  
 //打开文件  
 val file = Source.fromFile(filePath, codec)   
 val wc = file.getLines().flatMap(\_.split("\t")).toList.map((\_, 1)).groupBy((\_.\_1)).mapValues(\_.size)

println(wc)  
 // 关闭文件  
 file.close()  
 }  
}

WordCount程序参考代码2：

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

/\*\* \* Scala原生实现wordcount \*/

object WordCount {

def main(args: Array[String]): Unit = {

val list = List("cw is cool", "wc is beautiful", "andy is beautiful", "mike is cool")

/\*\* \* 第一步，将list中的元素按照分隔符这里是空格拆分，然后展开 \* 先map(\_.split(" "))将每一个元素按照空格拆分 \* 然后flatten展开 \* flatmap即为上面两个步骤的整合 \*/

val res0 = list.map(\_.split(" ")).flatten

val res1 = list.flatMap(\_.split(" "))

println("第一步结果")

println(res0)

println(res1)

/\*\* \* 第二步是将拆分后得到的每个单词生成一个元组 \* k是单词名称，v任意字符即可这里是1 \*/

val res3 = res1.map((\_, 1))

println("第二步结果")

println(res3)

/\*\* \* 第三步是根据相同的key合并 \*/

val res4 = res3.groupBy(\_.\_1)

println("第三步结果")

println(res4)

/\*\* \* 最后一步是求出groupBy后的每个key对应的value的size大小，即单词出现的个数 \*/

val res5 = res4.mapValues(\_.size)

println("最后一步结果")

println(res5.toBuffer)

}

}

1. 试一试，是否还有其他的实现方式？

# 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。