写

Java MapReduce

WordCount

```
public class WordCount {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        Configuration cfg = new Configuration();
        Job job = Job.getInstance(cfg);
        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCountMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);

        job.setReducerClass(WordCountReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileSystem fs = FileSystem.get(cfg);
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        boolean flag = job.waitForCompletion(true);
        if (flag) {
            System.out.println("任务执行成功！");
        }

    }

    public static class WordCountMapper extends Mapper<LongWritable, Text, Text,
LongWritable> {
        @Override
        protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
            String linecontext = value.toString();
            String[] words = linecontext.split(" ");
            for(String word:words) {
                outkey.set(word);
                outvalue.set(1);
                context.write(outkey, outvalue);
            }

        }
    }

    public static class WordCountReducer extends Reducer<Text, LongWritable,
Text, LongWritable> {
        @Override
        protected void reduce(Text key, Iterable<LongWritable> iter, Context
context) throws IOException, InterruptedException {
            long sum = 0;
            for(LongWritable i:iter) {
```

```
                sum+=i.get();
            }

            outkey.set(key);
            outvalue.set(sum);
            context.write(outkey, outvalue);
        }
    }
}
```

在WordCount中使用combiner

```
job.setCombinerClass(WordCountReducer.class);
```

在WordCount中使用Partitioner

```
job.setNumReduceTasks(3);
job.setPartitionerClass(WordsFrequenciesPartitioner.class);

    public class WordsFrequenciesPartitioner extends Partitioner<Text,
LongWritable>{
        @Override
        public intgetPartition(Text key, LongWritable value, int numOfReducer){
            return key.toString().length() % numOfReducer;
        }
    }
```

排序

Map端Sort

```
public class MySort extends WritableComparator{
    public MySort() {
        super(IntWritable.class,true);
    }
    public int compare(WritableComparable a,WritableComparable b) {
        IntWritable v1=(IntWritable)a;
        IntWritable v2=(IntWritable)b;
        return v2.compareTo(v1);
    }
}
job.setSortComparatorClass()
```

reduce端group

```
public class MyGroupSort extends WritableComparator {
    public MyGroupSort() {
        super(IntWritable.class, true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        IntWritable v1 = (IntWritable) a;
        IntWritable v2 = (IntWritable) b;
        if (v1.get() > 10) {
            return 0;
```

```
        } else {
            return -1;
        }
    }
}
job.setGroupingComparatorClass()
```

天气案例

同年数据放在同一个文件中。文件中输出的结果按年升序，如果同年按月升序，如果同月按温度倒排序。

```
自定义Writable Key：
public class MyKey implements WritableComparable<MyKey>  {
    private int year;
    private int month;
    private double hot;
    public void readFields(DataInput datainput) throws IOException {
        this.year=datainput.readInt();
        this.month=datainput.readInt();
        this.hot=datainput.readDouble();
    }
    public void write(DataOutput dataoutput) throws IOException {
        dataoutput.writeInt(year);
        dataoutput.writeInt(month);
        dataoutput.writeDouble(hot);
    }
    ##########
    public int compareTo(MyKey o) {
        int r=  Integer.compare(this.getYear(), o.getYear());
        if(r==0){
            int r1=Integer.compare(this.getMonth(), o.getMonth());
            if(r1==0){
                return Double.compare(this.getHot(), o.getHot());
            }
            return r1;
        }
        return r;
    }
}
自定义Sort：
public class MySort extends WritableComparator{
    public MySort(){
        super(MyKey.class,true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        MyKey mykey1=(MyKey)a;
        MyKey mykey2=(MyKey)b;
        int r=Integer.compare(mykey1.getYear(),mykey2.getYear());
        if(r==0){
             int r1=Integer.compare(mykey1.getMonth(), mykey2.getMonth());
             if(r1==0){
                return -Double.compare(mykey1.getHot(), mykey2.getHot());
             }
                return r1;
        }
```

```
        return r;
    }
}
```
自定义Partitioner：
```
public class MyPartitioner extends Partitioner<MyKey, DoubleWritable> {
    @Override
    public int getPartition(MyKey key, DoubleWritable value, int numPartitions)
{
        return (key.getYear() -1949)%numPartitions;
    }
}
```
自定义Group：
```
public class MyGroup extends WritableComparator {
    public MyGroup(){
        super(MyKey.class,true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        MyKey mykey1=(MyKey)a;
        MyKey mykey2=(MyKey)b;
        int r=Integer.compare(mykey1.getYear(),mykey2.getYear());
        if(r==0){
            return Integer.compare(mykey1.getMonth(), mykey2.getMonth());
        }
        return r;
    }
}
```
Map函数：
```
public static class WeatherMapper extends Mapper<Text,Text,MyKey,DoubleWritable>
{
    SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    DoubleWritable dhot=new DoubleWritable();
    @Override
    protected void map(Text key, Text value, Context context)
            throws IOException, InterruptedException {
        Date d;
        try {
            d = sdf.parse( key.toString());
            Calendar c=Calendar.getInstance();
            c.setTime(d);
            int year=c.get(Calendar.YEAR);
            int month=c.get(Calendar.MONTH) +1;
            double
hot=Double.parseDouble(value.toString().substring(0,value.toString().lastIndexOf
("c")));
            MyKey mykey=new MyKey();
            mykey.setYear(year);
            mykey.setMonth(month);
            mykey.setHot(hot);
            dhot.set(hot);
            context.write(mykey, dhot);
        } catch (ParseException e) {
                e.printStackTrace();
        }
    }
}
```
Reduce函数：

```
public static class WeatherReducer extends Reducer<MyKey, DoubleWritable, Text,
NullWritable>{
    Text outkey=new Text();
    @Override
    protected void reduce(MyKey mykey, Iterable<DoubleWritable> iter, Context
context)
                throws IOException, InterruptedException {
        int i=0;
        for(DoubleWritable d:iter){
            i++;
            if(i>3){
                break;
            }
            String msg=mykey.getYear()+"\t"+mykey.getMonth()+"\t"+d.get();
            System.out.println("r:"+msg);
            outkey.set(msg);
            context.write(outkey, NullWritable.get());
        }
    }
}
```

Main函数：

```
public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
    Configuration cfg=new Configuration();
    cfg.set("fs.defaultFS", "hdfs://master:9000");
    Job job=Job.getInstance(cfg);
    job.setMapperClass(WeatherMapper.class);
    job.setReducerClass(WeatherReducer.class);
    job.setMapOutputKeyClass(MyKey.class);
    job.setMapOutputValueClass(DoubleWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(NullWritable.class);
    job.setNumReduceTasks(3);
    #############
    job.setPartitionerClass(MyPartitioner.class);
    job.setSortComparatorClass(MySort.class);
    job.setGroupingComparatorClass(MyGroup.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    #KeyValueTextInputFormat使用分隔符\t将每一行分割为key和value，如果没有找到分隔符，当
前行的内容作为key，value为空串
    job.setInputFormatClass(KeyValueTextInputFormat.class);
    FileSystem fs =FileSystem.get(cfg);
    if(fs.exists(new Path(args[1]))){
        fs.delete(new Path(args[1]), true);
    }
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    boolean flag=job.waitForCompletion(true);
    if(flag){
        System.out.println("success");
    }
}
```

Scala

Hello World

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!");
  }
}
```

基本数据类型和变量

```
#val不可变，var可变，两种变量声明时都必须初始化
val s:int=1
var s=1
```

输入输出

控制台输入输出

```
#输入
var i= readBoolean
var i=readInt()
var str=readLine("please input here:")  #参数为提示信息
#输出
println("两条语句位于同一行时不能省略中间的分号")
print("i=");print(i)
printf("I am a %d year old %s.",age,name)
#s插值字符串
printf(s"i am $age year old")
#f插值字符串
printf(f"i am $age%.2f year old")
```

读写文件

```
#写入文件，PrintWriter类提供了两个写方法
val out=new PrintWriter("test.txt")
out.println("xxx")
out.print("...")
out.close()
#读取文件，使用Source的fromFile和getLines方法
val in=Source.fromFile("test.txt")
val lines=in.getLines  #返回一个迭代器
for(line <- lines) println(line)
```

控制结构

```
if，while 一样
# for循环
for(i<- 1 to 5 by 2) println(i)
for(i<- 1 to 5 if i%2==0)println(i)
for(i<- 1 to 5;j<- 1 to 3)println(i*j)
# 生成集合
for(i<- 1 to 5) yield {print(i);i}
```

```
#循环控制
breakable{
..
if(..)break
..
}
```

数据结构

```
数组Array[T]
#可变，元素类型相同，可索引
val a1=new Array[Int](3)
a1(0)=1
val a2=Array('s','f','d')
val a3=new Array.ofDim[Int](3,4)

元组Tuple
#元素类型可不同，可索引(从1开始)
val t1=('BigDATA', 1, 3.5)
println(t1._1)

容器Collection
foreach方法用于对容器元素进行遍历操作

列表List
#不可变，元素类型相同
val li=List('h','y')
#使用::在已有列表前端增加元素，可以使用空列表对象Nil
val nl='z'::'s'::li

向量Vector
val v1=Vector(1,2)
val v2=v1:+5
val v3=5+:v2

Range
val r=new Range(1,5,2)  #1 to 5 by 2

集合Set
var se=Set("Hadoop","Bigdata")
se+="ahah"

映射Map
val uni=Map("Qingh"->"Top1","BeiDa"->"Top2")
println(uni("Qingh"))
使用.contains()方法确认是否存在键
可添加
uni("Jilin")="Good"
uni+=("Jilin"->"Good","WuHna"->"Great")

迭代器Iterator
val iter = Iterator(1,2,3)
while(iter.hasNext){
    println(iter.Next())
}
```

面向对象编程基础

类

private成员只对本类型和嵌套类型可见；protected成员对本类型和其继承类型都可见

Scala采用类似Java中的getter和setter方法，定义了两个成对的方法value和value_=进行读取和修改

```
#创建类实例
var c1=new Counter
var c2=new Counter(3)
c1.value=5
c1.ad(3)

#主构造器的参数前使用val或var关键字，Scala内部将自动为这些参数创建私有字段，并提供对应的访问方法；如果不希望将构造器参数成为类的字段，只需要省略关键字var或者val
class Counter{
    var value=0
    def ad(step:Int):Unit={value+=step}
    #返回类型为Unit，可以同时省略返回结果类型和等号，但不能省略大括号
    def ad2(step:Int){value+=step}

    def current():Int={value} #定义中有括号，调用时可带括号也可不带括号
    def current2:Int=value #定义中无括号，调用时只能无括号
    #当方法的返回结果类型可以从最后的表达式推断出时，可以省略结果类型
    def current3()=value

    #辅助构造器
    #辅助构造器的第一个表达式必须是调用一个此前已经定义的辅助构造器或主构造器
    def this(value:Int){
        this()
        this.value=value
    }
}
```

对象

```
# 单例对象
/*
当一个单例对象和它的同名类一起出现时，这时的单例对象被称为这个同名类的“伴生对象”。相应的类被称为这个单例对象的“伴生类”。类和它的伴生对象必须存在于同一个文件中，可以相互访问私有成员。
没有同名类的单例对象，被称为孤立对象。一般情况下，Scala程序的入口点main方法定义在一个孤立对象里。
*/
class Person(val name:String){
    private val id=Person.newid()
    def info(){
     println("name:%s id:%s",name,id)
    }
}
object Person{
    private var Lastid=0
    def newid()={Lastid+=1;Lastid}
    def main(args:Array[String]){
        val p1=new Person("LiLei")
```

```
        val p2 = Person("Lilei2")
        p1.info()
    }
    def apply(p:String)=new Person(p)
}

#apply方法
用()传递给类实例或单例对象一个或多个参数时默认使用该类或对象的apply方法
val p2 = Person("Lilei")

#update方法
当对带有括号并包括一到若干参数的对象进行赋值时，编译器将调用对象的update方法，并将括号里的参数和等号右边的值一起作为update方法的输入参数来执行调用
myStrArr(0) = "BigData"
```

继承

```
#抽象类
包含未实现成员的类
abstract class Car(val name:String){
    var carBrand:String
    def info()
    def ex(){
        println{"haah"}
    }
}
#扩展类
class LAOS(name:String) extends Car(name){
    Override var carBrand="LAOS"
    def info(){
        print("I am LAOs")
    }
    Override def ex(){
        print("Override here")
    }
}
Object MyCar{
    def main(args:Array[String]){
        val myC=new LAOS("c1")
        c1.ex
    }
}
```

特质

一个类只能继承自一个超类，却可以实现多个特质

```
trait Car{
    var carBrand:String
    def info()
    def ex(){
        println{"haah"}
    }
}
trait Car2{
```

```
    var carName:String
    def info2()
}
class LAOS(carBrand:String,carName:String) extends Car with Car2{
    var carBrand:String=carBrand
    var carName:String=carName
    def info(){
        print("What the fuck")
    }
    def info2(){
        print("alright")
    }
}
```

模式匹配

```
#match语句
var s=1
val str = s match{
    case 2=> print("2");2
    case 1=> print("1");1
    case_=>print("啥也不是");0
}

#case类
case类是一种特殊的类，它们经过优化以被用于模式匹配
case class Car(brand: String, price: Int)
Scala为每一个case类自动生成一个伴生对象，其包括模板代码apply方法和unapply方法
object  Car{
    def apply(brand:String,price:Int)= new Car(brand,price)
    def unapply(c:Car):Option[(String,Int)]=Some((c.brand,c.price))
}
```

函数式编程基础

```
#Lambda表达式
(参数) => 表达式
num: Int => num * 2

#可以直接把匿名函数存放到变量
val myfunc:Int=>Int = num:Int=>num*2
print(myfunc(3))

#使用推断机制
val myfunc= num:Int=>num*2
val myfunc:Int=>Int = num=>num*2

#使用_简化函数表示
val myfunc=(_:Int)+(_:Int)     等效(a:Int,b:Int)=>a+b
val m2=m1.map(_*2)
```

针对容器的操作

```
#遍历
l1.foreach(x=>println(x._1+" "+x._2))  #_1和_2分别为x的key和value
l1.foreach(case(k,v)=>println(k+" "+v))

#映射
#map一对一映射，生成大小相同的容器但是类型可能不同
books.map(s => s.length)
#flatMap一对多映射
books.flatMap(s => s.toList)

#过滤
books.filter{_%2<5}

#规约
接受一个二元函数f作为参数
books.reduce(_+_)
```

WordCount

```
object WordCount {
    def main(args: Array[String]) {
        val dirfile=new File("testfiles")
        val files  = dirfile.listFiles
        val results = Map.empty[String,Int]
        for(file <-files) {
            val data= Source.fromFile(file)
            val strs =data.getLines.flatMap{s =>s.split(" ")}
            strs foreach { word =>
                if (results.contains(word))
                    results(word)+=1 else  results(word)=1
            }
        }
        results foreach{case (k,v) => println(s"$k:$v")}
    }
}
```

Spark

WordCount

```
object wordCount {
  def main(args: Array[String]): Unit = {
    //设置本机Spark配置
    val conf = new SparkConf().setAppName("wordCount").setMaster("local")
    //创建Spark上下
    val sc = new SparkContext(conf)
    //从文件中获取数据，结果作为许多rdd的模式取到（一行变一个RDD）
    val input = sc.textFile("D:\\code\\spark2\\WordCount\\data\\word.txt")
    //分析并排序输出统计结果
    input.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((x,
y) => x + y).sortBy(_._2,false).foreach(println _)
    #_._2表示用key排，false表示降序排
  }
}
```

RDD

基础

```
val conf=new SparkConf()
val sc=new SparkContext(conf)
```

RDD创建

```
# 从文件系统中加载数据创建RDD
val lines=sc.textFile("file:///usr/word.txt")
#hdfs://localhost:9000/user/word.txt

# 通过并行集合（数组）创建RDD
val lines=sc.parallelize(array)
```

转换

```
#filter
val lines=lines.filter(line=>line.contains("Spark"))

#map
val edd2=edd1.map(x=>x*2)
val edd2=edd1.map(x=>x.split(" "))  #生成元素种类为array

#faltMap
val words=lines.flatMap(x=>x.split(" "))

#groupByKey
应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)

#reduceByKey
rdd2=rdd1.reduceByKey((x,y)=>x+y)
```

动作

操作	含义
count()	返回数据集中的元素个数
collect()	以数组的形式返回数据集中的所有元素
first()	返回数据集中的第一个元素
take(n)	以数组的形式返回数据集中的前n个元素
reduce(func)	通过函数func（输入两个参数并返回一个值）聚合数据集中的元素
foreach(func)	将数据集中的每个元素传递到函数func中运行

持久化

```
persist(MEMORY_ONLY)              #=catch()
persist(MEMORY_AND_DISK)
unpersist()
```

分区

```
#创建RDD时手动指定分区个数
val rdd1=sc.parallelize(array,2)

#使用repariton方法重新设置分区个数
rdd1.repartition(3)

#自定义分区
需要定义一个类，这个自定义类需要继承org.apache.spark.Partitioner类，并实现下面三个：
numPartitions: Int  返回创建出来的分区数
getPartition(key: Any): Int  返回给定键的分区编号（0到numPartitions-1）
equals()  Java判断相等性的标准方法
```

```
import org.apache.spark.{Partitioner, SparkContext, SparkConf}
//自定义分区类，需要继承org.apache.spark.Partitioner类
class MyPartitioner(numParts:Int) extends Partitioner{
  //覆盖分区数
  override def numPartitions: Int = numParts
  //覆盖分区号获取函数
  override def getPartition(key: Any): Int = {
    key.toString.toInt%10
  }
}
object TestPartitioner {
  def main(args: Array[String]) {
    val conf=new SparkConf()
    val sc=new SparkContext(conf)
    //模拟5个分区的数据
    val data=sc.parallelize(1 to 10,5)
    //根据尾号转变为10个分区，分别写到10个文件
    data.map((_,1)).partitionBy(new
MyPartitioner(10)).map(_._1).saveAsTextFile("file:///usr/local/spark/mycode/rdd/
partitioner")
```

```
  }
}
```

综合实例

```
val lines=sc.textFile("file:///usr/line.txt")
val wordcount=lines.flatMap(line=>line.split(" ")).map(word=>
(word,1)).reduceByKey((a,b)=>a+b)
wordcount.collect()
wordcount.foreach(println)
```

键值对RDD

键值对RDD的创建

```
#多使用map()函数来实现
val lines=sc.textFile("file:///usr/input.txt")
val pair=lines.flatMap(line=>line.split(" ")).map(x=>(x,1))
```

常用的键值对RDD转换操作

```
#reduceByKey
输入为两个(k,v)的value值
pair.reduceByKey((x,y)=>x+y)

#groupByKey
对具有相同键的值进行分组，生成(k,iterator)
Pair.groupByKey().map(x => (x._1, x._2.sum))

#keys，values
把Pair RDD中的key返回形成一个新的RDD，value同理
pair.values

#sortByKey()
返回一个根据键排序的RDD
pair.sortByKey(false).collect()
#sortBy()
pair.sortBy(_._1,false).collect()

#mapValues()
pair.mapValues(x=>x+1)

#join
pair1.join(pair2)
```

综合实例

```
val rdd = sc.parallelize(Array(("spark",2),("hadoop",6),("hadoop",4),
("spark",6)))
rdd.mapValues(x => (x,1)).reduceByKey((x,y) => (x._1+y._1,x._2 +
y._2)).mapValues(x => (x._1 / x._2)).collect()
```

数据读写

文件数据读写

```
# 本地文件系统的数据读写
#读
val textFile = sc.textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
#写
textFile.saveAsTextFile("file:///usr/local/spark/mycode/wordcount/writeback")

# 分布式文件系统HDFS的数据读写
val textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
textFile.saveAsTextFile("writeback")

# JSON文件的读取
val jsonStr = sc.textFile("file:///usr/spark/examples/people.json")
jsonStr.map(s => JSON.parseFull(s))
```

五个案例

求Top值

```
/*
orderid,userid,payment,productid
求Top N个payment值
*/
object TopN {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("TopN").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")
    val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/rdd/examples",2)
    var num = 0;
    val result = lines.filter(line => (line.trim().length > 0) &&
(line.split(",").length == 4))
      .map(_.split(",")(2))
      .map(x => (x.toInt,""))
      .sortByKey(false)
      .map(x => x._1).take(5)
      .foreach(x => {
        num = num + 1
        println(num + "\t" + x)
      })
  }
}
```

求最大最小值

```
object MaxAndMin {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("MaxAndMin").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")
    val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/spark/chapter5",
2)
```

```
    val result = lines.filter(_.trim().length>0).map(line =>
("key",line.trim.toInt)).groupByKey().map(x => {
      var min = Integer.MAX_VALUE
      var max = Integer.MIN_VALUE
      for(num <- x._2){
        if(num>max){
          max = num
        }
        if(num<min){
          min = num
        }
      }
      (max,min)
    }).collect.foreach(x => {
      println("max\t"+x._1)
      println("min\t"+x._2)
    })
  }
}
```

文件排序

```
object FileSort {
    def main(args: Array[String]) {
        val conf = new SparkConf().setAppName("FileSort")
        val sc = new SparkContext(conf)
        val dataFile = "file:///usr/local/spark/mycode/rdd/data"
        val lines = sc.textFile(dataFile,3)
        var index = 0
        val result = lines.filter(_.trim().length>0).map(n=>
(n.trim.toInt,"")).partitionBy(new HashPartitioner(1)).sortByKey().map(t => {
          index += 1
            (index,t._1)
        })

result.saveAsTextFile("file:///usrl/local/spark/mycode/rdd/examples/result")
    }
}
```

二次排序

```
class SecondarySortKey(val first:Int,val second:Int) extends Ordered
[SecondarySortKey] with Serializable {
def compare(other:SecondarySortKey):Int = {
    if (this.first - other.first !=0) {
         this.first - other.first
    } else {
      this.second - other.second
    }
  }
}
object SecondarySortApp {
  def main(args:Array[String]){
```

```
     val conf = new
SparkConf().setAppName("SecondarySortApp").setMaster("local")
       val sc = new SparkContext(conf)
       val lines =
sc.textFile("file:///usr/local/spark/mycode/rdd/examples/file1.txt", 1)
       val pairWithSortKey = lines.map(line=>(new SecondarySortKey(line.split("
")(0).toInt, line.split(" ")(1).toInt),line))
       val sorted = pairWithSortKey.sortByKey(false)
       val sortedResult = sorted.map(sortedLine =>sortedLine._2)
       sortedResult.collect().foreach (println)
  }
}
```

连接操作

```
object SparkJoin {
  def main(args: Array[String]) {
    if (args.length != 3 ){
      println("usage is SparkJoin <rating> <movie> <output>")
      return
    }
   val conf = new SparkConf().setAppName("SparkJoin").setMaster("local")
   val sc = new SparkContext(conf)
   // Read rating from HDFS file
   val textFile = sc.textFile(args(0))
//extract (movieid, rating)
    val rating = textFile.map(line => {
        val fileds = line.split("::")
        (fileds(1).toInt, fileds(2).toDouble)
       })
 //get (movieid,ave_rating)
    val movieScores = rating
       .groupByKey()
       .map(data => {
         val avg = data._2.sum / data._2.size
         (data._1, avg)
       })
// Read movie from HDFS file
     val movies = sc.textFile(args(1))
     val movieskey = movies.map(line => {
       val fileds = line.split("::")
        (fileds(0).toInt, fileds(1))   //(MovieID,MovieName)
     }).keyBy(tup => tup._1)

     // by join, we get <movie, averageRating, movieName>
     val result = movieScores
       .keyBy(tup => tup._1)
       .join(movieskey)
       .filter(f => f._2._1._2 > 4.0)
       .map(f => (f._1, f._2._1._2, f._2._2._2))

    result.saveAsTextFile(args(2))
  }
}
```