项 目 3

Spark Core 数据分析

Spark Core 是 Spark 的核心模块，提供了分布式任务调度、内存计算、容错等基础功能。Spark Core 中的 SparkContext 对象用于连接到分布式集群并管理任务的执行，可将应用程序分解为一系列可执行的任务，并将这些任务分配给集群中的不同节点以实现并行计算。与传统的基于磁盘的 MapReduce 相比，Spark Core 采用内存计算，可以在内存中处理大规模数据集，大大提升了计算速度和效率。同时 Spark Core 具有良好的容错性，能够在系统故障时自动恢复丢失的数据或任务。

在本项目中，将通过完成 4 个具体的任务，系统学习 Spark Core 编程基础，包括 Spark 开发环境的搭建、RDD 的创建、常见算子的使用、RDD 的存储等。

【学习目标】

知识目标

1. 掌握 Spark 开发环境的部署方法。
2. 掌握创建 RDD 的方法。
3. 掌握 RDD 常见算子的使用方法。
4. 掌握键值对 RDD 的概念及创建方法。
5. 掌握键值对 RDD 常见算子的使用方法。
6. 掌握将数据存储为不同格式文件的方法。
7. 掌握自定义分区的使用方法。

能力目标

1. 能够独立完成 Spark 开发环境的部署。
2. 能够创建 RDD。
3. 能够使用 RDD 常见算子完成数据分析。
4. 能够创建键值对 RDD。
5. 能够使用键值对 RDD 常见算子完成数据分析。
6. 能够将 RDD 存储为不同格式的文件。
7. 能够实现自定义分区功能。

素质目标

1. 培养学生在解决实际问题时进行创新的能力。
2. 培养学生通过分工合作、知识共享，提升沟通与协调能力。
3. 培养学生的伦理意识，引导学生关注数据隐私、安全和公平问题。

【建议学时】

8 学时。

任务1 单词计数

【任务提出】

在集成开发环境 IntelliJ IDEA 中完成 Spark 环境的部署，搭建好项目后，编写 Spark 代码以实现单词计数功能。代码完成后将程序打包提交到 Hadoop 集群上，使用 Spark 本地模式运行程序，计算 HDFS 上 word.txt 文件中每个单词的出现次数。WordCount 输入文件内容和输出结果如图 3.1 所示。

```
Hello Spark              (hello,2)
Hello Hadoop Spark       (world,1)
Spark World              (spark,1)
```

图 3.1 WordCount 输入文件内容和输出结果

【任务分析】

本任务需要在 IntelliJ IDEA 中创建并部署 Spark 环境，搭建好项目后，编写程序实现单词计数。在本地测试无误后，将程序打包发送到集群上运行。具体的任务实施步骤如下。

1. 安装 IntelliJ IDEA。
2. 在 IDEA 中安装 Scala 插件。
3. 创建 Maven 项目，并导入 Spark Core 依赖。
4. 编写 Scala 代码实现单词计数，在本地测试无误后将程序打包。
5. 将 Jar 包发送到集群，使用 Spark 本地模式运行程序，观察计算结果。

【知识准备】

3.1 下载并安装 IntelliJ IDEA

IntelliJ IDEA 是由 JetBrains 公司开发的一款集成开发环境(IDE)，支持 Java、Kotlin、Groovy、Scala 等多种编程语言的开发。它以其强大的功能、智能的代码编辑器和出色的用户体验而闻名。本书使用的软件版本为 2022.3.3 社区版。

3.1.1 下载及安装

进入 IntelliJ IDEA 官网，找到社区版进行下载，按照以下步骤进行安装。

(1) 双击安装包，单击 Next 按钮，弹出图 3.2 所示对话框，选择安装路径。

(2) 单击 Next 按钮，弹出图 3.3 所示对话框，选择是否创建桌面快捷方式。

(3) 单击 Next 按钮，继续单击 Install 按钮完成安装。

图 3.2　设置安装目录

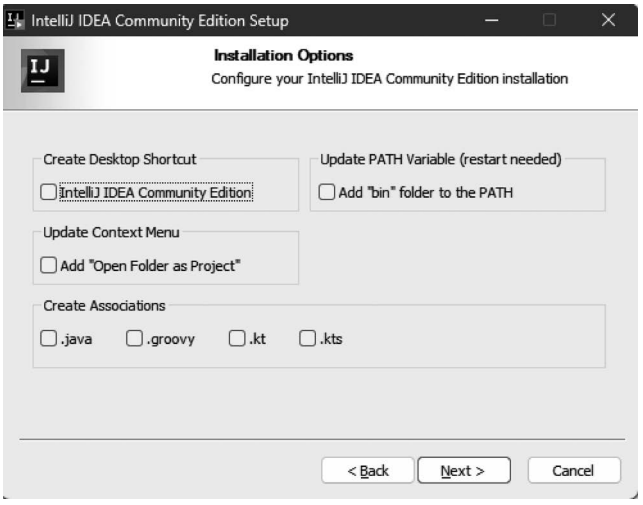

图 3.3　设置安装选项

3.1.2　安装 Scala 插件


视频讲解

安装完成后打开 IDEA，安装 Scala 插件，可选择在线安装或离线安装。

在线安装 Scala 插件步骤如下。

(1) 单击左侧的 Plugins 选项卡，右侧会显示 Marketplace 在线插件列表，如图 3.4 所示。

(2) 单击 Scala 右侧的 Install 按钮，会自动下载并安装 Scala 插件。安装完成后重启 IDEA。

离线安装 Scala 插件步骤如下。

(1) 下载所需要安装的 Scala 插件，也可以从官网链接找到 2022.3.3 版本进行下载。

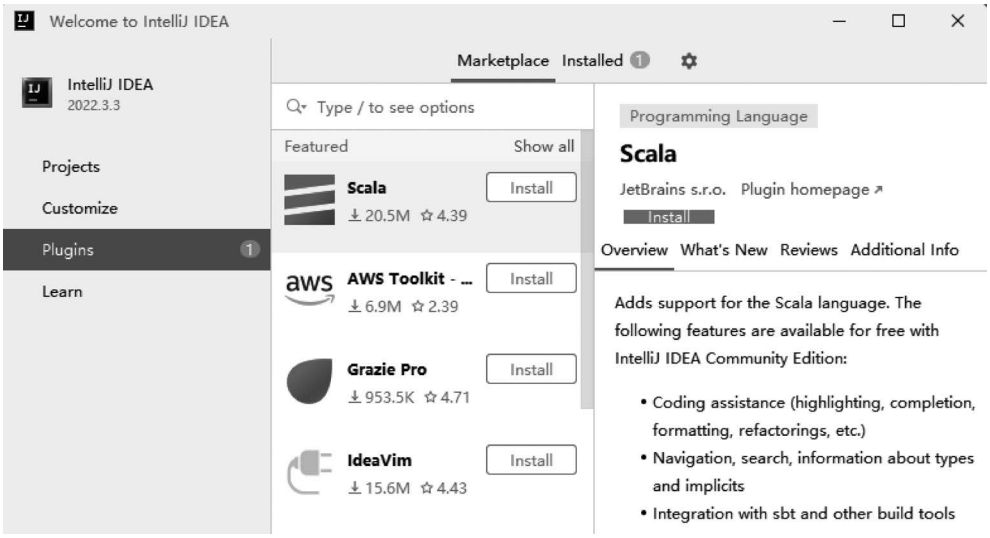

图 3.4 在线安装 Scala 插件

(2) 单击 Plugins 选项卡,从本地磁盘选择已下载的 Scala 插件,如图 3.5 所示。

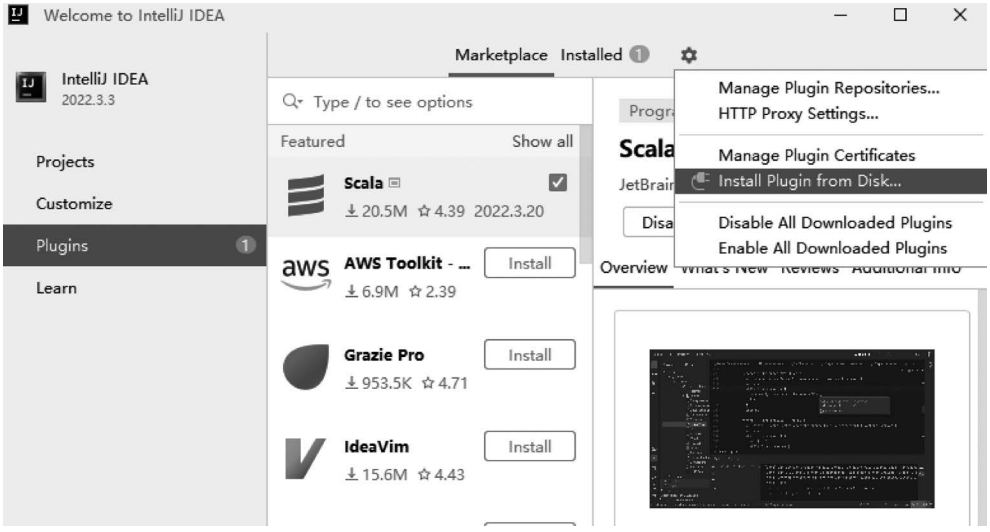

图 3.5 离线安装 Scala 插件

(3) 从本地选择 Scala 插件后执行安装操作,安装完成后重启 IntelliJ IDEA。

3.2 部署 Spark 编程环境

首先创建 Spark 项目。安装 Scala 插件后,创建 Maven 项目以编写 Spark 程序。项目创建步骤如下。

(1) 单击 New Project 选项,在弹出的对话框中设置项目名称等,如图 3.6 所示。

在 Name 处填写项目名,注意名称中不应包含中文且不以数字开头。选择项目所在路径,Language 选择 Java,Build system 选择 Maven,JDK 选择本地安装的 Java 1.8 版本。

视频讲解

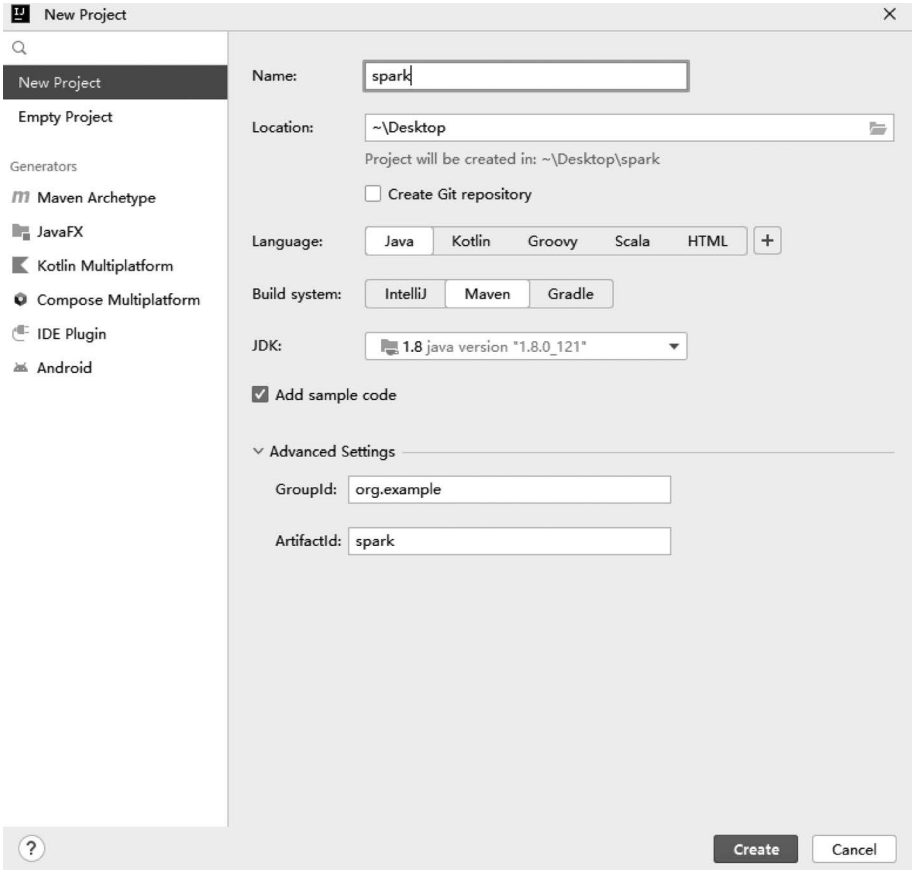

图 3.6　创建项目

(2) 设置完成后单击 Create 按钮,完成项目创建。

创建好 Maven 项目后,需要进行 Scala 编程环境的配置。具体步骤如下。

(1) 选中项目,右击选择 Add Frameworks Support,在弹出的对话框中选择 Scala,如图 3.7 所示,单击 OK 按钮关闭对话框。

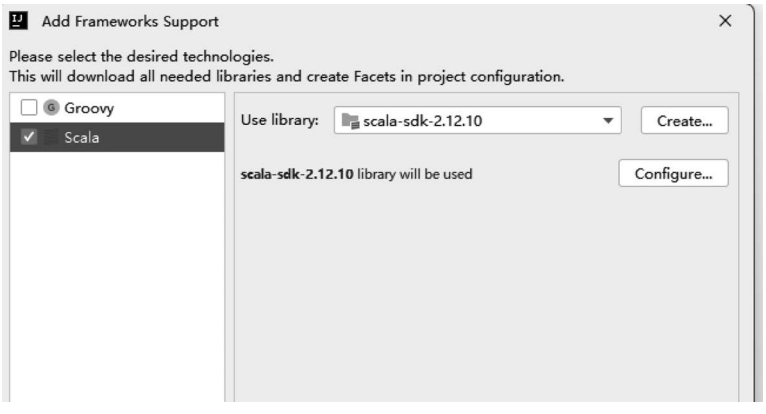

图 3.7　添加 Scala 框架支持

（2）将项目中默认生成的 java 文件夹进行重命名，如图 3.8 所示。

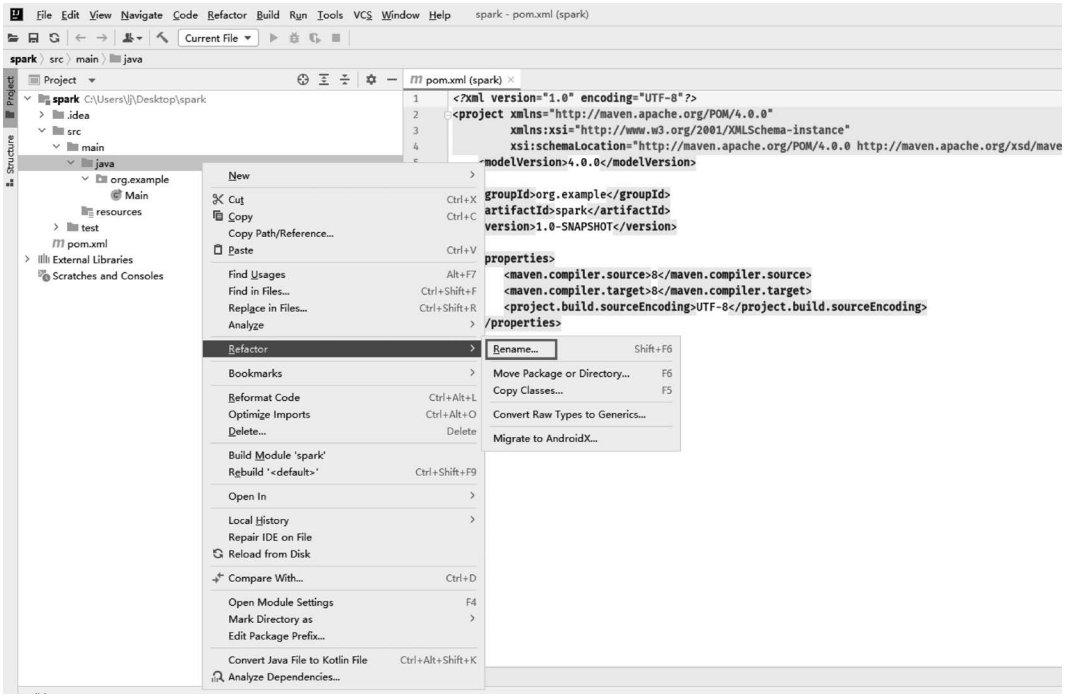

图 3.8　重命名

将文件名重命名为“scala”，如图 3.9 所示。

（3）将已有的包和类文件删除，选择新创建的“scala”文件夹，右击可新建包、新建 Scala 类，如图 3.10 所示。

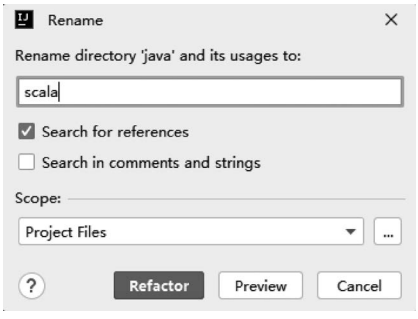

图 3.9　重命名为 scala 文件夹

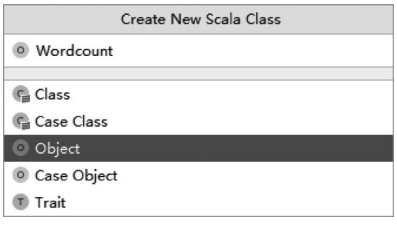

图 3.10　新建 Scala 类

（4）在进行 Spark 编程时，需要用到第三方依赖包。将依赖名写入 pom.xml 文件后，Maven 项目会根据配置文件，自动从镜像地址下载到本地。Maven 默认的镜像地址为国外网站，需要将该地址更改为国内阿里云镜像，并且配置本地库路径。右击项目右侧的 Maven 选项卡，在弹出的窗口中单击 图标，弹出图 3.11 所示对话框，用于设置 settings 文件和本地库路径。

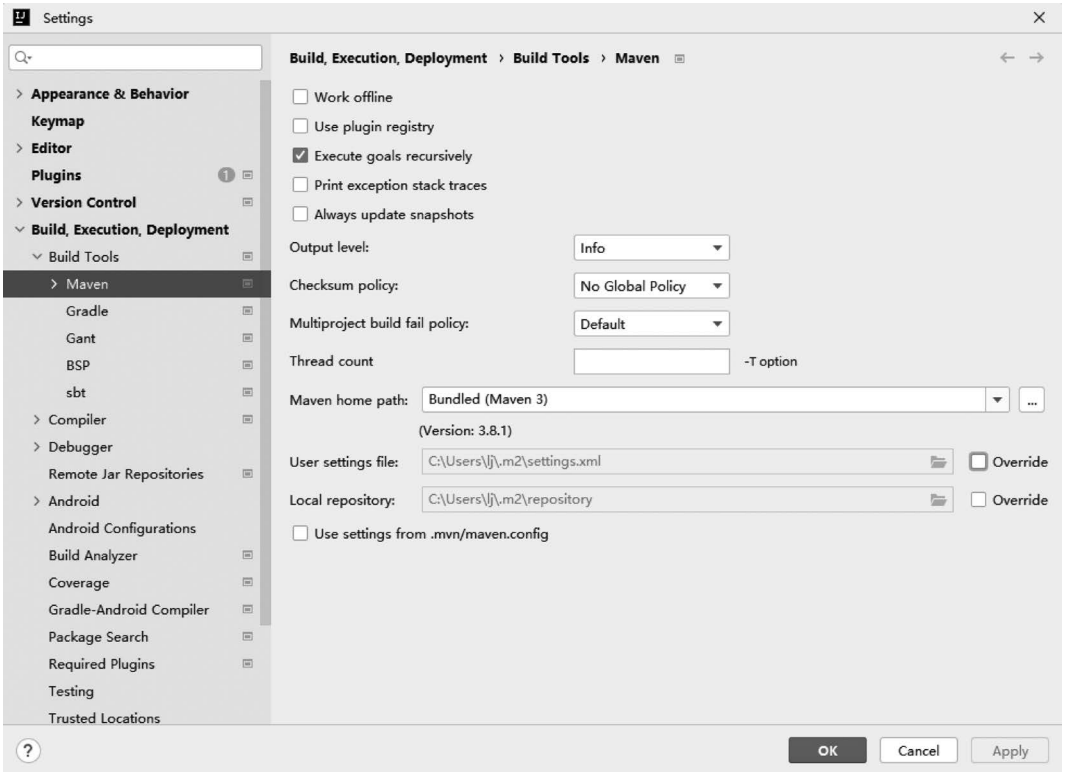

图 3.11　默认 settings 文件及本地库 Local respository 地址

打开该 settings.xml 文件，找到第 158 行，修改镜像地址。

```
158.  <--将原有的 mirror 注释掉
159.  <mirror>
160.  <id>mirrorId</id>
161.  <mirrorOf>repositoryId</mirrorOf>
162.  <name>Human Readable Name for this Mirror.</name>
163.  <url>http://my.repository.com/repo/path</url>
164.  </mirror>
165.  -->
166.  <mirror>
167.  <id>nexus</id>
168.  <mirrorOf>*</mirrorOf>
169.  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
170.  </mirror>
171.  <mirror>
172.  <id>nexus-public-snapshots</id>
173.  <mirrorOf>public-snapshots</mirrorOf>
174.  <url>http://maven.aliyun.com/nexus/content/repositories/snapshots/</url>
175.  </mirror>
```

找到第 53 行，修改本地库路径。

```
53.  <localRepository>E:/lj/course/maven/respository</localRepository>
```

修改完成后，回到 IntelliJ IDEA，选中 Override 选项，选择更改后的 settings. xml 文件，本地库路径会随之发生更改，检查无误后单击 OK 按钮，如图 3. 12 所示。

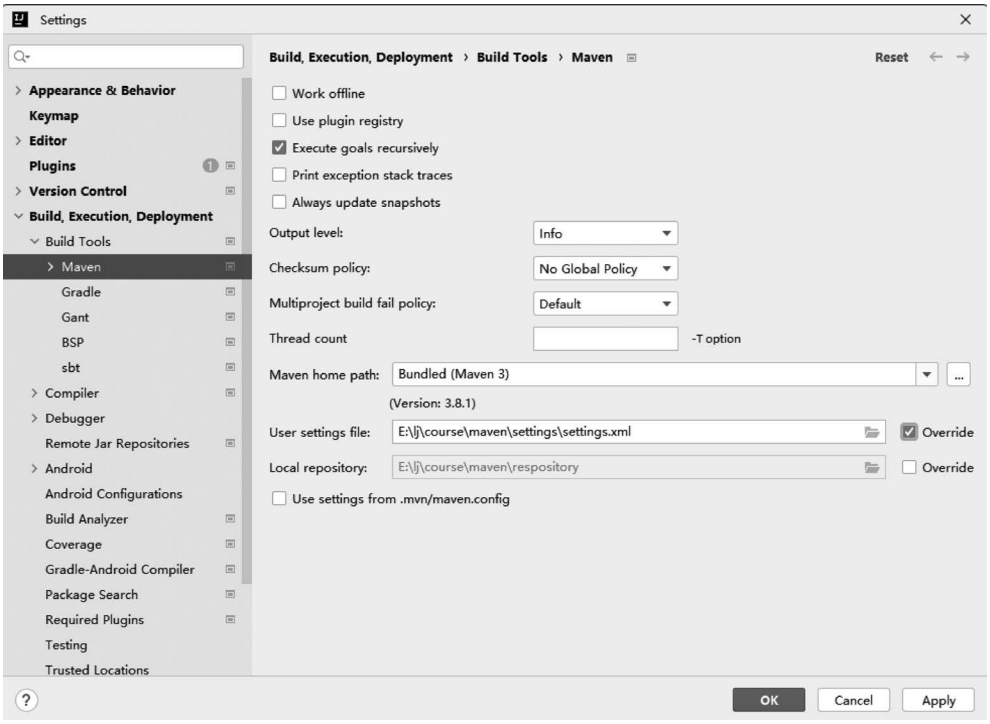

图 3. 12　更新后的 settiings. xml 文件和本地库路径

（5）打开项目的 pom. xml 文件，以增加 spark-core 依赖为例。本项目使用的 Scala 版本为 2. 12，Spark 版本为 3. 1. 1。依赖的版本号应与所用软件版本号保持一致。更改完成后，单击屏幕右侧 m 按钮导入依赖。

```
  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<!-- 增加依赖 -->
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.1.1</version>
    </dependency>
  </dependencies>
```

导入依赖后，在左侧“External Libraries”可以观察到已导入的依赖包，如图 3. 13 所示。

在编写程序之前，需要确保“scala”文件夹为项目根路径，在 IntelliJ IDEA 编辑器中为浅蓝色，如果为普通灰色文件夹，需要选中该文件夹，右击“Mark Directory As”，在弹出的菜单中选择“Sources Root”。

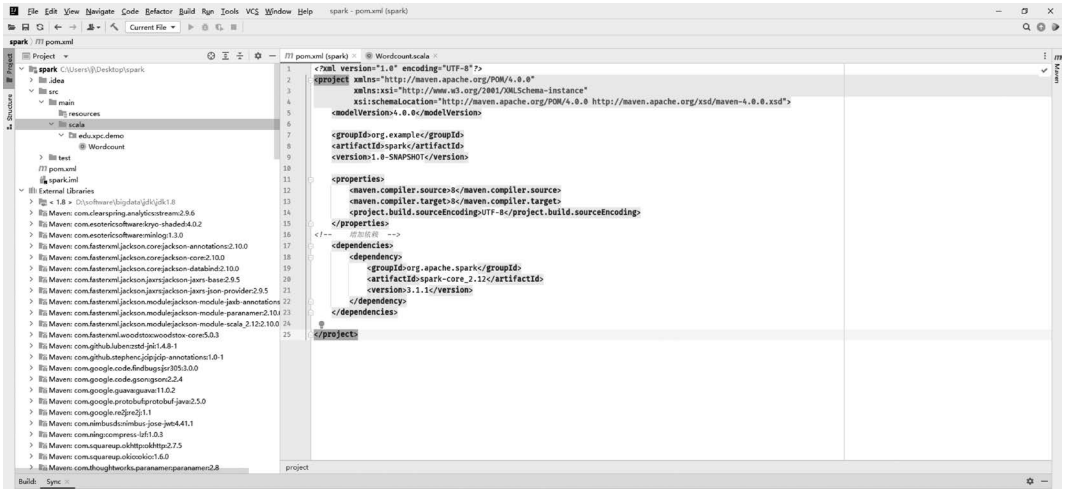

图 3.13　导入 Spark-core 依赖后的项目结构

3.3　程序打包并在集群运行

视频讲解

代码测试时通常直接在 IntelliJ IDEA 中运行程序。在实际生产环境下,需要处理的数据规模较大,所消耗资源较多时,需要将 Spark 程序打包运行,可以提高程序的可移植性、运行效率和部署简易性。

程序打包时需要在 pom.xml 文件中添加打包依赖,并完成导入。

```
<build>
    <sourceDirectory>src/main/scala</sourceDirectory>
    <testSourceDirectory>src/test/scala</testSourceDirectory>
    <plugins>
      <plugin>
        <groupId>net.alchim31.maven</groupId>
        <artifactId>scala-maven-plugin</artifactId>
        <version>3.2.2</version>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
            <configuration>
              <args>
                <arg>-dependencyfile</arg>
                <arg>${project.build.directory}/.scala_dependencies</arg>
              </args>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
```

```
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.4.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <filters>
                <filter>
                  <artifact>*:*</artifact>
                  <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                  </excludes>
                </filter>
              </filters>
              <transformers>
                 <transformer implementation="org.apache.maven.plugins.shade.
resource.ManifestResourceTransformer">
              </transformer>
            </transformers>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

在 IntelliJ IDEA 右侧 Maven 选项卡中单击项目名称下的 Lifecycle，再双击 package 即可完成打包。双击后，在左侧项目架构处生成文件夹 target，在该文件夹下生成两个 jar 包文件，如图 3.14 所示。

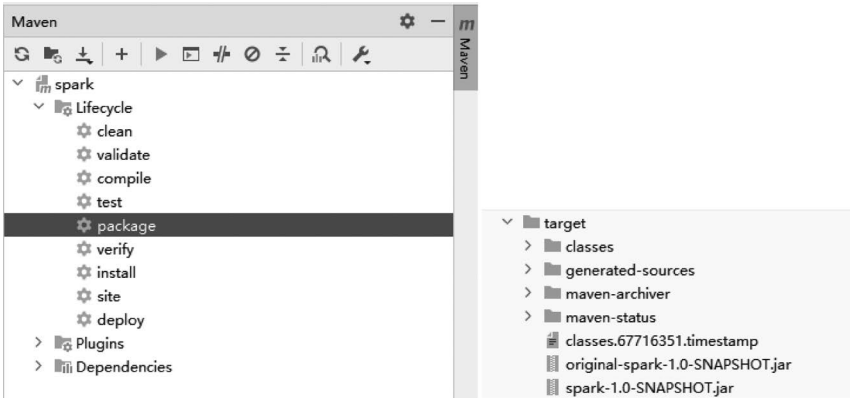

图 3.14 对代码进行打包，在 target 中生成 jar 包

其中，original-spark-1.0-SNAPSHOT.jar 包中主要包含程序代码，文件较小；spark-1.0-SNAPSHOT.jar 中将程序所需依赖进行了打包，文件较大。如果运行 jar 包的集群已经安装了所有依赖，则仅运行 original-spark-1.0-SNAPSHOT.jar 即可；如果环境中依赖不全，则需要使用 spark-1.0-SNAPSHOT.jar 运行程序。

打包的第二种方法是通过点击 File->File Structure->Project Settings->Artifacts 进行，依次选择 Add->JAR->From modules with dependencies，如图 3.15 所示。

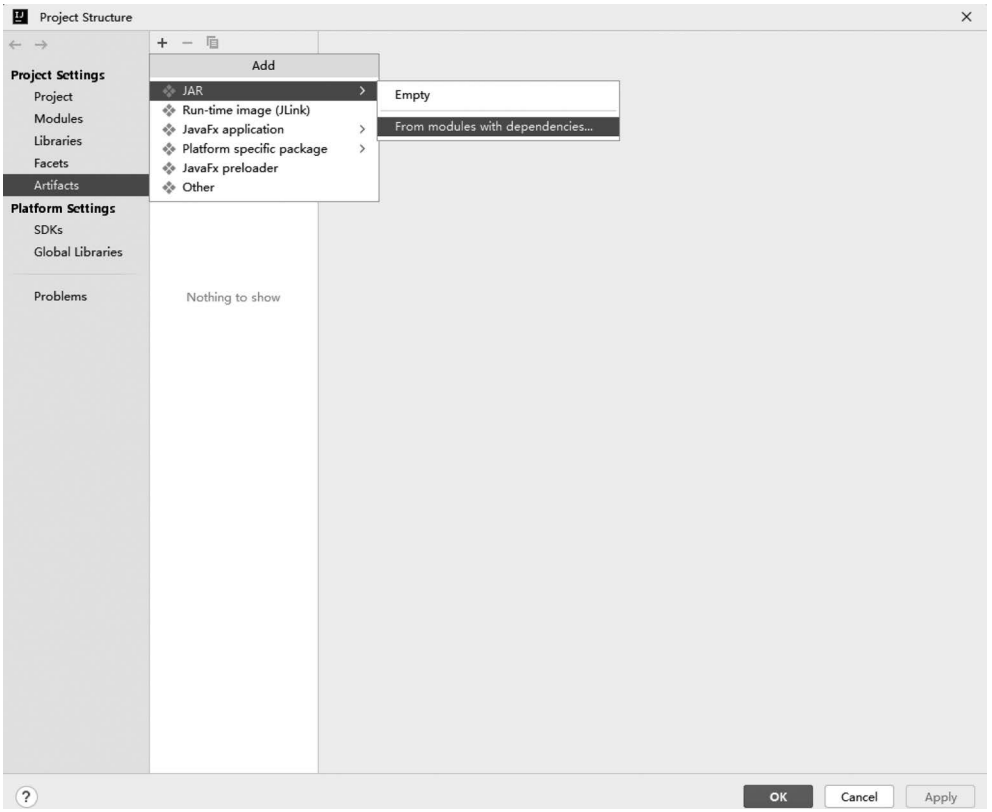

图 3.15　使用 Project Structure 进行打包

在弹出的对话框依次选择 Module、Main Class，如图 3.16 所示。

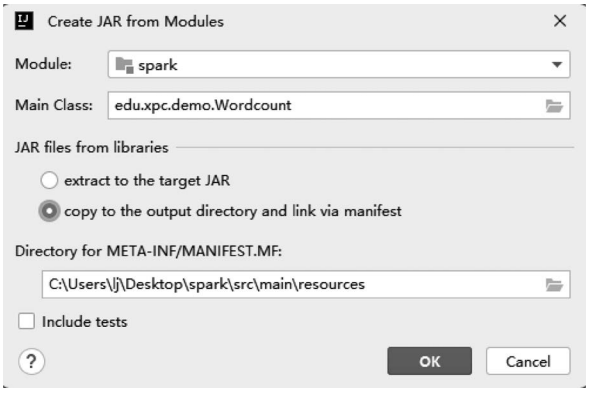

图 3.16　选择打包所需模块及主类

选择打包时所包含的依赖，如图 3.17 所示。

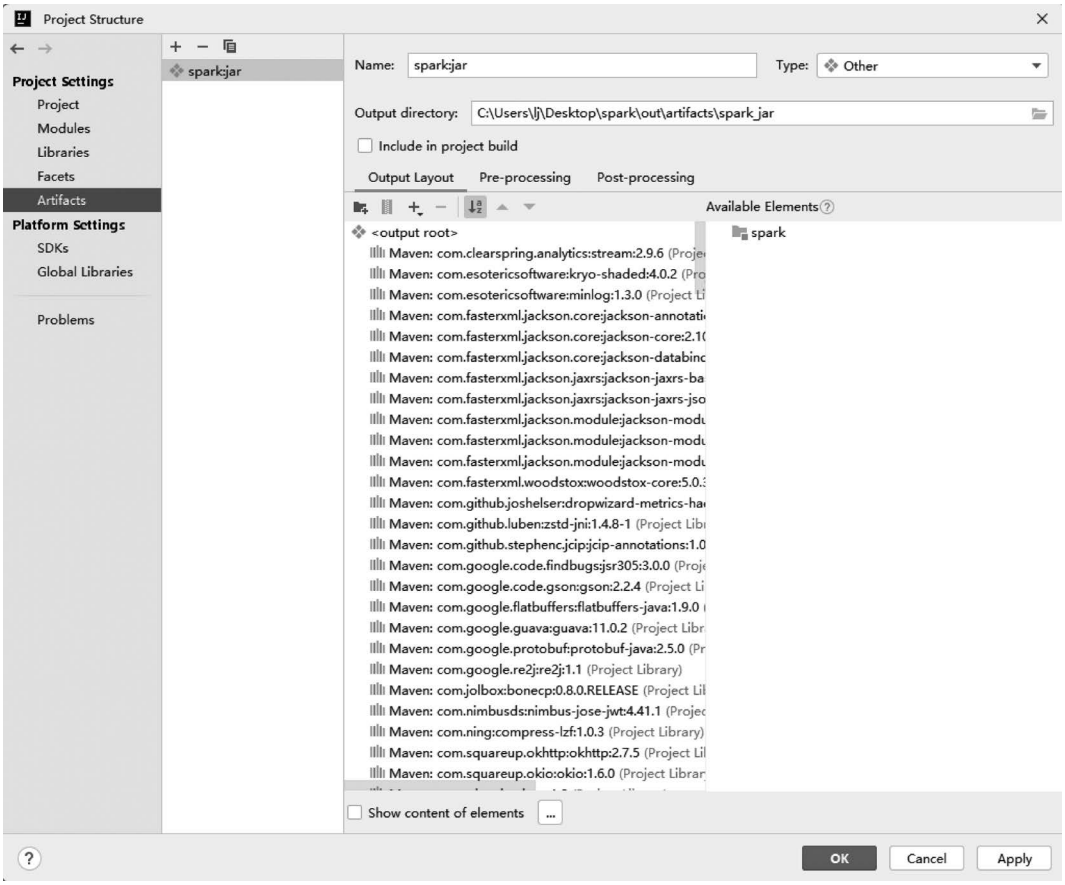

图 3.17 选择打包所包含的依赖

如果要运行的集群中包含程序所需要的所有依赖，则打包时可以将这些外部依赖删除，只保留代码，这样产生的 jar 包比较小，传输方便。选择完成后单击 OK 按钮。再单击右侧的 Maven 选项卡，依次选择 Lifecycle->package 按钮进行打包。

将 jar 包上传到虚拟机中，使用 spark-submit 指令即可运行程序。

【任务实现】

1. 步骤分析

首先，需要部署 IntelliJ IDEA 的 Spark 编程环境，创建项目；然后读取源文件并对数据进行处理，处理完成后将其保存为文本文件，确认代码中的输入文件路径和输出文件路径是作为主函数参数传递进去的。

2. 完成任务

(1) 启动 IntelliJ IDEA，创建 Maven 项目，添加 Scala 插件。

(2) 在 pom.xml 文件中加入 spark-core 依赖。

(3) 在项目中新建包 edu. xpc. demo,在该包中新建 scala 类,类名为 WordCount,在 WordCount. scala 中编写程序,代码如下。

```
package edu.xpc.demo

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

object Wordcount {
 def main(args: Array[String]): Unit = {
  //创建 SparkConf 对象,设置 AppName 为 WordCount,本地测试运行模式为 local[*]
  val sparkConf = new SparkConf().setAppName("WordCount").setMaster("local[*]")
  //创建 SparkContext 对象 sc
  val sc = new SparkContext(sparkConf)
  //设置 sc 日志级别
  sc.setLogLevel("ERROR")
  //读取本地文件 word.txt,将其转换为 rdd
  val rddinput = sc.textFile(args(0))
  //进行单词计数计算
  val rddout = rddinput.flatMap(_.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
  //打印结果
  rddout.foreach(println(_))
  //保存结果
  rddout.saveAsTextFile(args(1))
 }
}
```

将代码中的输入路径和输出路径设置成函数的参数,在运行 jar 包的时候在运行指令中输入该参数。

(4) 在 pom. xml 文件中加入打包依赖,将程序打包并上传到集群。

(5) 执行 jar 包运行指令,观察结果。

```
[root@master spark-3.1.1-bin-hadoop3.2]#bin/spark-submit --master local[*]
--class edu. xpc. demo. Wordcount  /usr/local/src/data/original - spark - 1. 0 -
SNAPSHOT.jar hdfs://master:9000/word.txt hdfs://master:9000/out
```

或

```
[root@master spark-3.1.1-bin-hadoop3.2]#bin/spark-submit --master local[*]
--class edu. xpc. demo. Wordcount  /usr/local/src/data/spark - 1. 0 - SNAPSHOT. jar
hdfs://master:9000/word.txt hdfs://master:9000/out1
```

程序的输入文件为集群根目录下的 word. txt 文件,两个 jar 包的输出分别为集群上 /out 和/out1 路径。执行完成后,在浏览器的地址栏中输入“master:9870”,观察集群输出,如图 3. 18 所示。

分别使用指令查看文件内容。

```
[root@master spark-3.1.1-bin-hadoop3.2]#hdfs dfs -cat /out/*
2024-04-30 09:42:45,283 INFO sasl.SaslDataTransferClient: SASL encryption trust
check: localHostTrusted = false, remoteHostTrusted = false
(Hello,2)
```

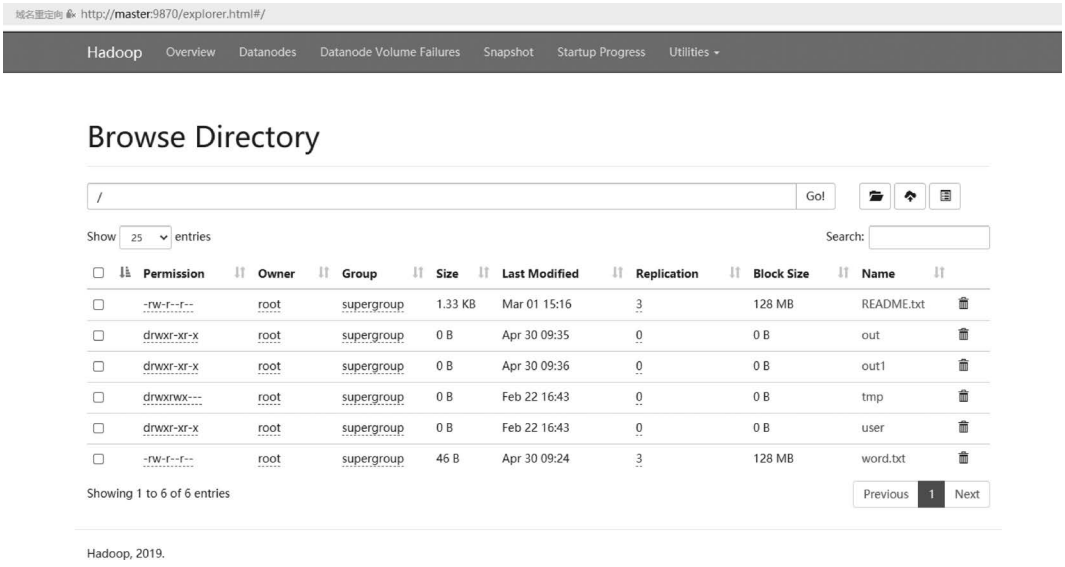

图 3.18 jar 包运行完成后 HDFS 上的文件结构

```
(World,1)
(Spark,3)
(Hadoop,1)
[root@master spark-3.1.1-bin-hadoop3.2]#hdfs dfs -cat /out1/*
2024-04-30 09:42:50,065 INFO sasl.SaslDataTransferClient: SASL encryption trust
check: localHostTrusted = false, remoteHostTrusted = false
(Hello,2)
(World,1)
(Spark,3)
(Hadoop,1)
```

可以看到,两个 jar 包运行的结果是一样的,都正确计算出了单词的个数。

【任务总结】

通过本任务的学习,掌握了如何在 IntelliJ IDEA 中部署 Spark 开发环境,完成代码后将程序打包上传至集群运行。在整个任务实施过程中需注意以下几点。

(1) 在部署 Spark 环境导入依赖时,需要注意依赖包中的各软件版本号,应确保与集群环境的版本号一致。在更新完 pom.xml 文件后,要手动将新增加的依赖导入项目中,否则不生效。

(2) 在使用 spark-submit 指令运行程序时,其中的--class 应为全类名,在书写时要将所在的包名一并写上。

(3) 运行 jar 包的输出路径,必须为一个不存在的路径。如果该路径已存在,程序会报错。

【巩固练习】

一、单选题

1. 在 IntelliJ IDEA 中，如何运行打包好的 Spark 应用程序？（　　）
   A. 在终端中使用 spark-submit 命令运行生成的 jar 文件
   B. 在 IntelliJ IDEA 中直接单击 Run 按钮
   C. 在项目根目录下执行生成的 jar 文件
   D. 在 IntelliJ IDEA 中无法运行打包好的 Spark 应用程序
2. 在 IntelliJ IDEA 中，如何导入 Spark 相关的依赖库？（　　）
   A. 在项目根目录下手动添加 jar 包
   B. 在 Project Structure 中选择 Libraries 选项卡，然后单击“＋”按钮添加依赖，或使用 pom.xml 文件添加依赖导入
   C. 在 IntelliJ IDEA 中无法导入 Spark 相关的依赖库
   D. 在项目根目录下创建一个名为“lib”的文件夹，将所有依赖的 jar 包放入其中
3. （　　）是 IntelliJ IDEA 插件管理工具。
   A. Plugin Manager　　B. Extension Manager
   C. PackageManager　　D. Marketplace

二、判断题

1. IntelliJ IDEA 支持多种编程语言，包括 Java、Python、Scala 等。（　　）
2. IntelliJ IDEA 提供了对多种版本控制工具的支持，包括 Git、SVN 等。（　　）

三、简答题

1. 简要说明在 IntelliJ IDEA 中创建 Maven 项目的步骤。
2. 简要说明如何在 Spark 项目中导入 Maven 依赖。
3. 简要说明 Spark 项目打包运行的具体步骤。

【任务拓展】

将单词计数生成的 jar 包使用 Spark-on-Yarn 模式在集群上运行，观察不同模式下 jar 包运行的结果。

任务 2　统计交易额

【任务提出】

online_retail_Ⅱ.txt 数据集封装了一家总部位于英国的某零售公司的所有销售数据，

该公司专注于多种场合的礼品销售。本数据集包含这家公司的部分交易数据，共541998条记录，涵盖产品代码、产品描述、交易数量、交易时间等多个维度。通过分析该数据集，可以深入了解该公司的销售模式、客户行为、产品受欢迎程度等信息，进而为公司的业务决策提供有力支持。

该数据集字段说明如下。

1. InvoiceNo 发票号码：描述每笔交易分配的唯一6位数发票号码，如果该号码以字母c开头，则表示这是一个退货订单。

2. StockCode 库存编号：用于描述每个不同的产品，是该产品分配的唯一5位整数代码，用于标识和追踪库存中的指定商品。

3. Description 产品信息：对每件产品的简略描述，通常包括产品的名称及类型，用于提供所购买产品的描述信息。

4. Quantity 产品数量：每笔交易中每种产品的数量。

5. InvoiceDate 发票日期和时间：每笔交易发生的日期和时间，可用于分析销售季节性、趋势、工作日与周末的差异等。

6. UnitPrice 产品单价：描述单位产品的价格。

7. CustomterID 顾客编号：为每个客户分配的唯一5位整数编号，可用于分析客户购买行为、客户忠诚度、细分客户群等。

8. Country 国家/地区：描述每个客户所在国家或地区的名称，可用于分析地理销售模式、国际销售趋势等。

现在需要在IntelliJ IDEA中编写程序，读取该文件，统计交易额最高的前10个订单程序运行结果如图3.19所示。

```
**************交易额最高的前10个订单及金额结果如下******************
(581483,168469.6)
(541431,77183.6)
(574941,52940.93999999999)
(576365,50653.909999999996)
(556444,38970.0)
(567423,31698.159999999996)
(556917,22775.930000000008)
(572209,22206.0)
(567381,22104.8)
(563614,21880.439999999995)
```

图3.19 统计交易额最高的前10个订单程序运行结果

【任务分析】

本任务需要读取源数据，创建RDD，并根据数据格式分割出各字段，统计各订单的交易额，找出交易额最高的前10个订单。具体的任务实施分析如下。

1. 编写Spark Core程序，读取源文件，创建RDD。

2. 数据清洗，去除已取消的订单，保留有效订单数据。

3. 对原始数据进行分割，获取相关字段，并进行相应的类型转换。

4. 按照订单号分组，计算各订单的交易额。

5. 按照交易额排序，取交易额最高的前 10 个订单，并打印。

【知识准备】

3.4 创建 RDD

每个 Spark 应用程序都包含一个驱动程序 Driver，它运行用户的主函数，并在集群中执行各种并行操作。Spark 提供的主要抽象是 RDD，它是一个元素集合，分布在集群的各节点上，能够进行并行操作。RDD 可以通过 Hadoop 文件系统或其他任何支持的文件系统中的文件来创建，也可以来自驱动程序中现有的 Scala 集合，还可以通过对已有的 RDD 进行转换来创建。同时，Spark 支持用户将 RDD 持久化到内存中，以便在并行操作中高效地重用。另外，RDD 可以自动从节点故障中恢复。

Spark 中的另一个抽象是共享变量，可用于并行操作。默认情况下，当 Spark 并行运行一个函数作为不同节点上的一组任务时，它会将函数中使用的每个变量的副本传输给每个任务。有时，需要在任务之间或在任务与驱动程序之间共享变量。共享变量有助于优化 Spark 程序的性能，特别是在分布式环境下，可以有效避免数据的重复计算和不必要的网络传输。

Spark 的核心概念是 RDD，它是一个容错的元素集合，可以并行操作。创建 RDD 有两种方式：一是在驱动程序中并行化现有集合；二是引用外部存储系统中的数据集，如共享文件系统、HDFS、HBase 或任何提供 Hadoop InputFormat 的数据源。

3.4.1 在驱动程序中并行化现有集合创建 RDD


视频讲解

在 Spark 驱动程序中，首先需要创建 Spark Context 对象。在 Spark Shell 界面中，该对象默认命名为 sc，如图 3.20 所示。

```
[root@master spark-3.1.1-bin-hadoop3.2]# bin/spark-shell
24/05/10 15:32:36 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/05/10 15:32:47 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Spark context Web UI available at http://master:4041
Spark context available as 'sc' (master = local[*], app id = local-1715326367825).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.1.1
      /_/

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_231)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

图 3.20 Spark Shell 界面

在程序中，需要使用如下代码创建 Spark Context 对象。

```
//setAppName 用于设置程序名，setMaster 用于设置程序运行模式
val sparkConf = new SparkConf().setAppName("createRDD").setMaster("local[*]")
val sc = new SparkContext(sparkConf)
```

RDD 可以通过在驱动程序中的现有集合上调用 Spark Context 的 parallelize 方法来创建。该集合的元素会被复制，形成一个可以并行操作的分布式数据集。该方法有两个参数

输入。

(1) 要转换的集合,必须为 Seq 集合。

(2) 分区个数,默认为该 Application 分配到的资源的 CPU 数。

例 3.1 创建一个包含数字 1 到 5 的 RDD。

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
val distData2 = sc.parallelize(data, 2)      //分区数为 2
```

3.4.2 引用外部存储系统中的数据集创建 RDD


视频讲解

Spark 可以从 Hadoop 支持的任何存储源(包括本地文件系统、HDFS、Cassandra、HBase、Amazon S3 等)创建 RDD。通过使用 Spark Context 的 textFile 方法创建文本文件 RDD。该方法接受文件的 URI,同时可以设置分区个数。

例 3.2 读取本地文件创建 RDD,同时设置分区数为 2。

```
val words: RDD[String] = sc.textFile("data/word.txt", 2)
```

例 3.3 读取 HDFS 文件创建 RDD,同时设置分区数为 2。

```
val hdfswords = sc.textFile("hdfs://master:9000/word.txt", 2)
```

默认情况下,该方法会为文件的每个块创建一个分区,可以通过参数设置更大的分区数。

3.5 RDD 的常用转换操作

RDD 支持两种类型的操作:转换(Transformations)操作和行动(Actions)操作。转换操作从现有 RDD 创建一个新的 RDD,而行动操作在对 RDD 执行计算后将一个值返回给驱动程序。例如,map 是一种转换操作,它通过一个函数将每个数据集元素传递,并返回表示结果的新 RDD。另一方面,reduce 是一种行动操作,它使用某个函数聚合 RDD 的所有元素,并将最终结果返回给驱动程序。

Spark 中的所有转换操作都是惰性的,因为它们不会立即计算结果。相反,它们只是记住了应用到某个基本数据集(如一个文件)上的转换操作。转换操作只有在某个行动操作需要将结果返回给 Driver 时才会被计算。RDD 的常用转换操作如表 3.1 所示。

表 3.1 RDD 的常用转换操作

转 换 操 作	说 明
map(func)	将源数据集中的每个元素使用函数 func 进行处理,返回一个新的分布式数据集
filter(func)	返回一个新的数据集,通过函数 func 计算返回 true 的原数据集元素组成
flatMap(func)	类似于 map,但每个输入项可以映射到 0 个或多个输出项,因此 func 应返回一个序列而不是单个项

续表

转换操作	说明
distinct([numPartitions]))	返回一个新的数据集,由原始数据集去重后的元素组成
repartition(numPartitions)	重新设置 RDD 的分区数。会随机重排 RDD 中的数据,以创建更多或更少的分区,并在这些分区之间进行平衡
groupBy(keyfunc)	返回一个分组项的 RDD。每个组由一个键和映射到该键的元素序列组成
sortBy(keyfunc)	返回按照给定键函数对该 RDD 进行排序的结果

这些操作类的方法会返回一个新的 RDD,用于后续继续进行转换操作或行动操作。

3.5.1 使用 map 操作转换 RDD

map 操作是最基本的转换操作,用于将 RDD 中的每个元素进行转换,得到一个新的 RDD。常见的应用场景有以下三种。

(1) 对每个字符串类型的元素做切割、重组等操作。

(2) 获取部分字段。

(3) 对每个元素进行类型转换、数据结构的转换等。

例 3.4 map 的使用。将一个包含 5 个 Int 类型元素的 RDD 中的每个元素乘以 10 后的结果保存到新的 RDD 中。

```
val list = List(1,2,3,4,5)
val value = sc.parallelize(list)
val result = value.map(x => x *10)
result.foreach(println(_))
//********************运行结果********************
10
30
40
20
50
```

例 3.5 map 的灵活运用。现有多名同学的基本信息,包括姓名、分数。将每位同学的分数取出来并进行类型转换,最后打印结果。

```
val stu = List("Kevin,92","Lucy,84","Mark,87")
val rddstu = sc.parallelize(stu)
val rddscore = rddstu.map(x => x.split(",")(1).toInt)
rddscore.foreach(println(_))
//********************运行结果********************
87
92
84
```

3.5.2 使用 filter 操作过滤 RDD

filter 操作用于对 RDD 中的元素进行过滤,对原数据集中的每个元素应用函数进行计算,如果函数结果返回 true,则保留该元素。经过过滤后的集合元素个数会小于或等于原始

数据集，但不会改变原数据集的元素类型及格式。filter 通常用于对数据进行清洗等预处理。

例 3.6 filter 的使用。将包含 5 个元素的 RDD 中的偶数类元素过滤出来，保存为新的数据集并进行打印。

```
val list = List(1,2,3,4,5)
val value = sc.parallelize(list)
val result = value.filter(x => x % 2 == 0)
result.foreach(println(_))
//********************运行结果********************
4
2
```

例 3.7 filter 的灵活运用。将上述学生成绩中，成绩大于 90 分的学生信息过滤出来并打印。

```
rddstu
 .filter(x=>x.split(",")(1).toInt > 90)
 .foreach(println(_))
//********************运行结果********************
Kevin,92
```

filter 方法并不会改变数据的结构，计算得到的结果为原始数据集的子集。

3.5.3 使用 flatMap 操作转换 RDD

flatMap 与 map 类似，不同的是，flatMap 将函数应用于 RDD 的每个元素，然后将返回迭代器的结果中的所有元素取出构成新的 RDD。例如在单词计数案例中，输入的每个元素为一行字符串，如“hello world”，使用 flatMap 对该字符串进行切割得到数组，再将数组中的每个元素取出，在新的数据集中就会保存为两个元素，分别为“hello”和“world”。因此，经过 flatMap 操作后，新数据集中的元素个数往往会远远大于原数据集。

例 3.8 单词计数中的 flatMap。在单词计数任务中，需要对每一行文本进行切割得到每个单词。

```
val lines = sc.parallelize(List("hello world","hello spark world"))
lines.flatMap(line => line.split(" "))
 .foreach(println(_))
//********************运行结果********************
hello
world
hello
spark
world
```

原数据集中仅包含“hello world”和“hello spark world”两个长字符串，flatMap 使用空格对原字符串进行切割，得到数组后依次取出每个元素，这样就获取到了每个单词，新集合中共包含 5 个元素。

3.5.4 使用 distinct 操作对 RDD 去重

distinct 操作用于去除 RDD 中的重复元素，这也是数据清洗的一种常见操作。

例 3.9 distinct 操作。原数据集中包含重复元素，使用 distinct 操作去除重复元素，并打印新数据集的结果。

```
val value1 = sc.parallelize(List(1, 1, 3, 2))
value1.distinct().foreach(println(_))
//**********运行结果**********
2
3
1
```

观察结果数据集，可以看到该操作实现了数据的去重。

3.5.5 使用 repartition 操作重新设置 RDD 分区数

repartition 方法用于重新设置 RDD 分区数，该方法接受一个 Int 类型的整数作为参数，返回一个新的数据集。

例 3.10 repartition 操作。首先打印 value1 的分区个数，再使用 repartition 操作将分区个数设置为 1，再重新打印设置之后的分区个数。

```
println(value1.partitions.size)
println(value1.repartition(1).partitions.size)
//**********运行结果**********
8
1
```

由代码运行结果可知，默认情况下数据的分区数为程序分配到的 CPU 个数，此处为 8。重新设置为 1 个分区后，新的分区数被成功设置为 1。

3.5.6 使用 groupBy 操作对 RDD 分组

groupBy 操作用于对 RDD 中的元素进行分组，返回一个由分组项组成的 RDD，每个组由一个键和映射到该键的元素序列组成。该操作中的函数会对数据集中每个元素进行计算，根据计算结果将元素分到不同的组中。结果集中的键对应该组的组名，也就是函数的不同的计算结果；结果集中对应的元素序列为该组中所包含的元素列表。

例 3.11 groupBy 操作。对数据集(1,1,3,2)中的元素按奇偶性进行分组，将所有奇数分到一组，所有偶数分到另一组。

```
value1.groupBy(x => x % 2 == 0).foreach(println(_))
//**********运行结果**********
(true,CompactBuffer(2))
(false,CompactBuffer(1, 1, 3))
```

该数据被分成了两组，结果集中的 true 表示对该组中的元素应用对 2 取模的函数结果为 true，true 对应的序列中包含元素 2；false 表示对该组中的元素应用该函数结果为 false，

该组中对应的元素包括1,1,3。此操作成功地对数据进行了分组。

3.5.7 使用 sortBy 操作对 RDD 元素排序

sortBy 用于对 RDD 中的元素进行排序。该操作包含以下 3 个参数。

(1) 排序函数,根据函数获取元素中的排序字段。

(2) 排序方向,为布尔类型,默认为升序,若要设置为降序,将该参数设置为 false。

(3) 分区个数,为 Int 类型,该参数为排序后的 RDD 的分区个数。默认情况下,排序后的分区个数与排序之前的个数相等。

sortBy 操作仅对分区内的数据进行排序,因此只能实现局部排序。如果需要对数据进行全局排序,可以通过将分区个数设置为 1 的方式实现。

例 3.12 sortBy 操作。对 rddstu 中的元素按照成绩进行倒序排序。

```
val stu = List("Kevin,92","Lucy,84","Mark,87")
val rddstu = sc.parallelize(stu)
rddstu.sortBy(x=>x.split(",")(1).toInt,false).foreach(println(_))
//********************运行结果********************
Mark,87
Lucy,84
Kevin,92
```

该示例首先通过 split 函数切割学生信息,得到成绩字段后将其转换为数值类型,再按照成绩进行排序。运行结果显示,分数并没有按照预想的由高到低进行排列,这是因为该操作默认为分区内排序,仅保证每个分区内的顺序是正确的。如果希望完成全局排序,则需要在排序的时候将分区个数设置为 1。

例 3.13 sortBy 操作重新设置分区个数,实现全局排序。

```
rddstu.sortBy(x=>x.split(",")(1).toInt,false,1).foreach(println(_))
//********************运行结果********************
Kevin,92
Mark,87
Lucy,84
```

重新排序后,数据按照成绩字段由高到低重新排列。

3.6 RDD 的常用行动操作

行动操作输入一个 RDD,返回一个数值、数组或其他类型的计算结果。前面章节代码中用到的 foreach 方法就是一种行动操作。RDD 常用的行动操作如表 3.2 所示。

表 3.2 RDD 常用的行动操作

行 动 操 作	说 明
reduce(func)	使用一个函数 func(接受两个参数并返回一个参数)对数据集的元素进行聚合。该函数应满足交换律和结合律,以便在并行计算中正确地进行计算
collect()	将数据集的所有元素作为一个数组返回到 Driver

续表

行 动 操 作	说　　明
count()	返回数据集的元素个数
first()	返回数据集的第一个元素
take(n)	返回数据集的前 n 个元素
foreach(func)	对数据集的每个元素运行函数 func

只有遇到行动操作,程序才会被触发执行。

3.6.1　使用 reduce 操作对 RDD 元素进行聚合

reduce 操作使用函数对 RDD 中的元素进行聚合。该函数以两个元素作为参数,返回两个元素聚合后的结果。

例 3.14　reduce 操作。使用 reduce 操作对 RDD 中的元素求和。

```
val value = sc.parallelize(List(1, 2, 3, 4))
val i = value.reduce((a, b) => a + b)
println(i)
//********************运行结果********************
10
```

该操作依次将 RDD 中的各元素相加,最终得到所有元素的和。reduce 操作可用于对元素个数求和、求最大值、求最小值等。

3.6.2　使用 collect 操作获取 RDD 元素集合

collect 操作返回一个包含 RDD 中所有元素的数组。

例 3.15　collect 操作,将 RDD 转换为一个数组。

```
val array: Array[Int] = value.collect()
```

需要注意的是,在数据集较大时不要使用该操作,因为它会将所有数据加载到 Driver 端的内存,数据集过大会导致内存溢出等问题。

3.6.3　使用 count 操作获取 RDD 元素个数

count 操作返回 RDD 中的元素个数,并将结果保存在 Long 类型的变量中。

例 3.16　count 操作。计算 RDD 中的元素个数并打印。

```
val array: Array[Int] = value.collect()
val l = value.count()
println(l)
//********************运行结果********************
4
```

3.6.4　使用 first 操作获取 RDD 第一个元素

若需要获取 RDD 中的第一个元素,可使用 first 操作实现。

例 3.17 frist 操作。获取 RDD 中的第一个元素并打印。

```
println(value.first())
//*********************运行结果*********************
1
```

3.6.5 使用 take 操作获取 RDD 前 n 个元素

take 操作返回 RDD 的前 n 个元素，并将结果保存在数组中。该操作从第一个分区开始扫描，并使用该分区的结果来评估满足个数条件所需的额外分区数量。

例 3.18 take 操作。获取 RDD 中的前三个元素并打印。

```
   val array: Array[Int] = value.take(3)
array.foreach(println(_))
//*********************运行结果*********************
1
2
3
```

需要注意的是，只有当预期的结果数组较小时，才建议使用此方法，因为所有数据都会加载到 Driver 的内存中。由于内部实现的复杂性，如果在一个包含 Nothing 或 Null 的 RDD 上调用此方法，它将引发异常。

3.6.6 使用 foreach 操作计算 RDD 每个元素

foreach 操作将函数应用于 RDD 中的每个元素。

例 3.19 foreach 操作。依次打印 RDD 中的每个元素。

```
value.foreach(println(_))
//*********************运行结果*********************
1
2
4
3
```

3.7 集合类操作

RDD 是一个分布式的数据集合，因此也有一些集合类的操作方法，例如并集、交集、补集、笛卡儿积等。这些方法最终都会返回一个 RDD，因此都属于转换操作。集合类操作方法如表 3.3 所示。

表 3.3 集合类操作方法

集合类操作	说　明
union(otherDataset)	返回两个 RDD 的并集
intersection(otherDataset)	返回两个 RDD 的交集
substract(otherDataset)	返回一个 RDD，元素为源数据集去除与参数数据集相同元素后剩余的结果
cartesian(otherDataset)	返回一个包含源数据集和参数数据集元素的笛卡儿积

3.7.1 使用 unoin 操作合并多个 RDD

union 操作返回两个 RDD 的并集，并且不会对结果进行去重。

例 3.20 union 操作。计算两个 RDD 的并集并打印。

```
val rdd1 = sc.parallelize(List(1, 2, 4))
val rdd2 = sc.parallelize(List(3, 4, 5, 6))
rdd1.union(rdd2).foreach(println(_))
//*********************运行结果*********************
1
2
4
3
4
5
6
```

3.7.2 使用 intersection 操作计算交集

intersection 操作返回两个 RDD 的交集。

例 3.21 intersection 操作。计算两个 RDD 的交集，并打印结果。

```
rdd1.intersection(rdd2).foreach(println(_))
//*********************运行结果*********************
4
```

为了找到两个数据集中相同元素的集合，Spark 首先会将两个 RDD 中的元素重新分配到合适的分区，然后每个分区会比较 rdd1 和 rdd2 中的元素并找到其交集，最终将计算结果进行合并。因为涉及数据的重新分配，因此会触发 Shuffle 过程，这种操作可能会因网络延迟和 I/O 开销导致系统性能下降，尤其对于大数据集，需要谨慎使用该操作。

3.7.3 使用 substract 操作计算差集

substract 操作返回一个 RDD，元素为源数据集去除与参数数据集相同元素后剩余的结果。

例 3.22 substract 操作。

```
rdd1.subtract(rdd2).foreach(println(_))
//*********************运行结果*********************
2
1
```

rdd1 中包含元素 1、2 以及 4，与 rdd2 中的相同元素为 4，去掉该相同元素后的结果为 1 和 2。

3.7.4 使用 cartesian 操作计算笛卡儿积

cartesian 操作返回 RDD 和另一个 RDD 的笛卡儿积，即所有元素对(a，b)的 RDD，其中

a 在原 RDD 中，b 在参数 RDD 中。

例 3.23 cartesian 操作。

```
val rdd1 = sc.parallelize(List(1, 2, 4))
val rdd3 = sc.parallelize(List("a", "b"))
rdd1.cartesian(rdd3).foreach(println(_))
//********************运行结果********************
(1,a)
(1,b)
(2,b)
(2,a)
(4,a)
(4,b)
```

结果集中的元素个数为两个 RDD 元素个数的乘积，值为两个 RDD 中元素组成的键值对。

【任务实现】

1. 步骤分析

首先，需要读取源文件并将其转换成 RDD，将得到的数据集中的每一行进行切割，并清洗无效订单数据。接着计算每个订单的交易额，最后按照交易额由高到低的顺序打印前 10 个订单的信息。

2. 完成任务

(1) 启动 IntelliJ IDEA，右击项目文件夹，选择相应的 package，新建 scala 类。

(2) 打开类文件，在代码编辑窗口中输入如下代码。

```
package edu.xpc.sparkcore
import org.apache.spark.{SparkConf, SparkContext}

object Rw2 {
 def main(args: Array[String]): Unit = {
  val sparkConf = new SparkConf().setAppName("WordCount").setMaster("local[*]")
  val sc = new SparkContext(sparkConf)
  val rddretail = sc.textFile("data/online_retail_II.txt")
  println("**********交易额最高的前 10 个订单及金额结果如下**********")
  rddretail.map(x=>x.split("\t"))
   .filter(x=> !x(0).toLowerCase.startsWith("c"))
   .map(x=>(x(0),x(3).toDouble *x(5).toDouble))
   .groupBy(x=>x._1)
   .map(x=>(x._1,x._2.map(x=>x._2)))
   .map(x=>(x._1,x._2.sum))
   .sortBy(x=>x._2,false,1)
   .take(10)
   .foreach(println(_))
  sc.stop()
 }
}
```

(3) 测试源代码,运行当前程序,结果如图 3.19 所示。

【任务总结】

通过本任务的学习,读者可以掌握如何创建 RDD,以及如何使用 RDD 的各种算子进行数据的转换和分析。在编程过程中,需要注意以下几点。

(1) 在进行数据分析前,往往需要对数据进行清洗。数据清洗不仅可以去除不满足规范的数据,还能提取关键字段,是大数据处理流程中必不可少的环节。

(2) 在使用 RDD 的算子时,应注意各算子的应用场景,这样才能做到灵活运用,举一反三。

【巩固练习】

一、单选题

1. (　　)不是 RDD 的转换算子。

A. filter()　　B. map()　　C. count()　　D. flatMap()

2. 算子(　　)可以对 RDD 进行去重操作。

A. union()　　B. distinct()

C. reduceByKey()　　D. groupByKey()

3. 算子(　　)可以对 RDD 进行扁平化操作。

A. map()　　B. flatMap()　　C. filter()　　D. reduceByKey()

二、判断题

1. filter()函数可以对 RDD 中的元素进行筛选,并返回一个新的 RDD。　　(　　)

2. map()函数可以将 RDD 中的每个元素通过一个函数映射为多个元素,并返回一个新的 RDD。　　(　　)

3. sortBy()函数无法对 RDD 中的元素进行降序排序操作。　　(　　)

4. 在 Spark 中,RDD 是不可修改的数据集。　　(　　)

5. Spark 中的 RDD 可以包含任何类型的数据。　　(　　)

6. 在 Spark 中,RDD 的操作是懒执行的,只有在遇到转换操作时才会真正执行计算。　　(　　)

7. 在 Spark 中,RDD 是一种分布式数据集,可以跨多台机器进行计算。　　(　　)

三、编程题

1. 现有几位学生的分数,依次存放在列表中,分别是("kevin,85,79","lucy,92,84","mark,80,69")。计算每位同学的总分并打印结果。

2. 按照编程题第 1 题计算的总分,给所有同学按分数由高到低排序并打印结果。

3. 统计每个学生的平均分数。

4. 计算每门课程的平均分数。

【任务拓展】

现有一份关于某个地区的空气质量和气象情况的记录。其中，PM2.5浓度是一种衡量空气中颗粒物浓度的指标，DEWP、TEMP、HUMI、PRES这些字段则描述了气象情况，而cbwd、Iws、precipitation、Iprec等字段则反映了风向、风速和降水情况。对PM2.5浓度和气象条件进行分析的结果，可用于推断污染源和制定改善空气质量的措施。通过分析降水量和累计降水量，可以预测洪水等自然灾害的发生。这份数据包含的字段说明如表3.4所示。

表3.4 空气质量数据字段说明

序　号	字 段 名	说　明
1	No	数据行号
2	year	数据的年份信息
3	month	数据的月份信息
4	day	数据的日期信息
5	hour	数据的小时信息
6	season	季节(1：春季，2：夏季，3：秋季，4：冬季)
7	PM_Dongsi	PM2.5浓度(ug/m^3)。Dongsi监测点数据
8	PM_Dongsihuan	PM2.5浓度(ug/m^3)。Dongsihuan监测点数据
9	PM_Nongzhanguan	PM2.5浓度(ug/m^3)。Nongzhanguan监测点数据
10	PM_US Post	PM2.5浓度(ug/m^3)。US Post监测点数据
11	DEWP	露点(摄氏度)
12	HUMI	湿度
13	PRES	气压(hPa)
14	TEMP	温度
15	cbwd	联合风向
16	Iws	累计风速(m/s)
17	precipitation	小时降水量(mm)
18	Iprec	累计降水量(mm)

请完成以下分析任务。

1. 将该文件上传至HDFS，读取该文件后创建RDD。

2. 删除文件中缺失值大于3个字段的数据，并打印删除条数。

3. 在第2步的基础上删除文件中关键字段PM_US Post、HUMI、precipitation、Iprec中任意字段为空的数据，并打印删除条数。

4. 计算每个季节的平均PM2.5浓度(PM2.5的值根据PM_US Post字段计算)，并按照季节的顺序输出结果。

5. 计算每个小时的平均湿度，并按照小时的顺序输出结果。

6. 找出PM2.5超过100的数据记录，并统计其出现次数。

7. 求出每个季节的最大降水量，并输出对应的季节和降水量值。

8. 将程序打包，上传到集群上运行，记录打印结果。

任务 3　商品交易量分析

【任务提出】

商品交易量分析可为商业决策提供有力支持，帮助企业和零售商更好地了解市场需求、优化库存、提高销售额和客户满意度。通过分析不同地区的畅销商品，企业可以了解市场趋势和消费者偏好，从而预测未来的市场需求。这有助于企业及时调整产品线，确保产品组合与市场需求相匹配。也可以帮助企业根据实际需求调整库存水平，避免库存积压或缺货现象，制定更具针对性的营销策略，如促销、广告、定价等，帮助企业在激烈的市场竞争中保持竞争优势。

本任务需要计算出每个地区每种商品的销量，然后找到销量最高的商品，返回的结果包含地区名、商品名及销量，将最终结果保存到文本文件中，如图 3.21 所示。

```
Australia,MINI PAINT SET VINTAGE ,2952
Portugal,POLKADOT PEN,240
United Kingdom,PAPER CRAFT , LITTLE BIRDIE,80995
Brazil,ROSES REGENCY TEACUP AND SAUCER ,24
Canada,RETRO COFFEE MUGS ASSORTED,504
Japan,RABBIT NIGHT LIGHT,3408
Cyprus,HEART DECORATION PAINTED ZINC ,384
Finland,CHILDRENS CUTLERY POLKADOT PINK,480
European Community,WHITE ROCKING HORSE HAND PAINTED,24
Netherlands,RABBIT NIGHT LIGHT,4801
Iceland,ICE CREAM SUNDAE LIP GLOSS,240
Singapore,CHRISTMAS TREE PAINTED ZINC ,384
Sweden,MINI PAINT SET VINTAGE ,2916
RSA,ASSORTED BOTTLE TOP  MAGNETS ,12
Norway,SMALL FOLDING SCISSOR(POINTED EDGE),576
Denmark,RED  HARMONICA IN BOX ,288
```

图 3.21　每个地区最畅销商品的计算结果

【任务分析】

本任务首先需要计算商品销量，然后按照地区进行分组，求该地区销量最高的商品。具体实施步骤如下。

1. 读取源数据创建 RDD，清洗掉无效记录。
2. 计算每个地区每种商品的销量。
3. 按地区分组，求出最高销量的商品。
4. 将结果保存到文本文件中。

【知识准备】

3.8 键值对 RDD 的创建


视频讲解

大多数 RDD 操作适用于包含任意类型对象的 RDD，但是一些特殊操作仅适用于键值对的 RDD。最常见的特殊操作是分布式“Shuffle”操作，例如，按键对元素进行分组或聚合。在 Scala 中，这些操作自动适用于包含 Tuple2 对象的 RDD。键值对操作位于 PairRDDFunctions 类中，它会自动将 RDD 中的元素包装成元组。

键值对 RDD 可直接创建，也可以由普通 RDD 通过 map()操作转换而来。

例 3.24 键值对 RDD 的创建。在单词计数的案例中，可以通过 map 将由单词组成的 RDD 转换成键值对 RDD。

```
val lines = sc.parallelize(List("hello world", "hello spark"))
val wordAndOne = lines.flatMap(x=>x.split(" "))        //元素变成每个单词
 .map(x=>(x,1))                                        //转换成键值对 RDD
wordAndOne.foreach(println(_))
//********************运行结果********************
(hello,1)
(world,1)
(hello,1)
(spark,1)
```

键值对 RDD 的元素是二元元组，元组的第一个元素为“键”，第二个元素为“值”，从而构成了键值对的结构。通过 keys 操作可以获取 RDD 的所有键，通过 values 操作可以获取 RDD 的所有值，二者均为转换操作。

3.9 键值对 RDD 的常用操作

部分转换操作仅适用于键值对 RDD，常用的转换操作如表 3.5 所示。

表 3.5 键值对 RDD 的常用转换操作

键值对常用操作	说　明
join(otherDataset, [numPartitions])	当在类型为(K,V)和(K,W)的数据集上调用时，返回一个数据集，其中包含每个键的所有元素对(K,(V,W))
groupByKey ([numPartitions])	当在类型为(K,V)的数据集上调用时，返回一个数据集，其中包含每个键对应的(K,Iterable<V>)对
reduceByKey(func, [numPartitions])	当在类型为(K,V)的数据集上调用时，使用给定的 reduce 函数 func(该函数必须是类型为(V,V)=>V 的函数)对每个键的值进行聚合，并返回一个(K,V)对的数据集
sortByKey ([ascending], [numPartitions])	当在类型为(K,V)的数据集上调用时，其中 K 实现了 Ordered 接口，它返回一个键按升序或降序排序的(K,V)对的数据集，具体排序顺序由布尔型参数 ascending 指定
countByKey()	该操作仅适用于类型为(K,V)的 RDD。它返回一个哈希映射，其中包含每个键的计数，类型为(K,Long)。这是一个行动操作

3.9.1 使用 join 操作连接键值对 RDD


视频讲解

join 操作用于连接两个键值对 RDD。它将两个具有相同键的数据放在同一个元组中，结果数据集只包含两个 RDD 中都存在的键的连接结果。例如，输入的两个 RDD 键值对分别为(K,V)和(K,W)，join 计算结果为(K,(V,W))。

例 3.25 join 操作连接两个键值对 RDD，并打印结果。

```
val rdd1 = sc.parallelize(List(("apple", 3), ("pear", 1), ("banana", 5)))
val rdd2 = sc.parallelize(List(("grape", 2), ("apple", 1)))
rdd1.join(rdd2).foreach(println(_))
//*********************运行结果*********************
(apple,(3,1))
```

若要实现外连接，可通过 leftOuterJoin、rightOuterJoin 和 fullOuterJoin 等操作实现。

3.9.2 使用 groupByKey 操作对键值对 RDD 进行分组


视频讲解

groupByKey 根据键对 RDD 中的元素进行分组。例如，输入的 RDD 键值对为(K,V)，分组结果为包含每个键的(K,Iterable<V>)对。

例 3.26 groupByKey 操作。在单词计数中按照单词对元素进行分组，并打印结果。

```
wordAndOne.groupByKey().foreach(println(_))
//*********************运行结果*********************
(hello,CompactBuffer(1, 1))
(spark,CompactBuffer(1))
(world,CompactBuffer(1))
```

该示例将相同单词的键值对元素放到了同一组，其中“hello”单词出现了两次，分组结果的键为该单词，值为两个 1 构成的可变序列。

如果需要分组后执行每个键上的聚合(如求和或平均值)，使用 reduceByKey 或 aggregateByKey 将获得更好的性能。默认情况下，输出中的并行级别取决于父 RDD 的分区数，但可通过可选的 numPartitions 参数来设置不同的分区个数。

3.9.3 使用 reduceByKey 操作对键值对 RDD 进行聚合


视频讲解

reduceByKey 操作用于合并具有相同键的值，该操作只对值进行处理。在进行处理时，该操作需要接收一个函数，相同键的值会根据函数进行合并，返回一个新的键值对元素。与 groupByKey 类似，可以通过可选的第二个参数来配置 reduce 任务的数量。

例 3.27 reduceByKey 操作。在单词计数案例中，可以直接通过 reduceByKey 操作对元素进行聚合，计算值的和，求出每个单词的个数。

```
wordAndOne.reduceByKey((a,b)=>a+b).foreach(println(_))
//*********************运行结果*********************
(spark,1)
(hello,2)
(world,1)
```

在该例子中，reduceByKey 会首先将相同键的两个值进行求和，新产生的和与下一个相同键的值继续求和，最终计算完所有的键值对元素，从而实现了单词计数功能。

3.9.4 使用 sortByKey 操作对键值对 RDD 进行排序

sortByKey 操作用于对键值对 RDD 中的元素按照键进行排序。该操作包含以下两个参数。

(1) 排序方向，为布尔值类型。默认为 true，表示升序排序。若要实现降序排序，需要将该参数设置为 false。

(2) 排序后的分区个数，默认值与原 RDD 的分区个数相同。与 sortBy 操作一样，该操作仅对分区内的元素实现排序。如果想实现全局排序，需要将分区个数设置为 1。

例 3.28 sortByKey 操作。对 wordAndOne 中的元素进行排序并打印结果。

```
wordAndOne.sortByKey(false,1).foreach(println(_))
//**********运行结果**********
(world,1)
(spark,1)
(hello,1)
(hello,1)
```

示例中的键为字符串，因此数据集会按照字符的顺序进行排序。

3.9.5 使用 countByKey 操作按键计算键值对 RDD 个数

countByKey 操作用于按键统计元素个数，返回 Map[K,Long]类型的变量。该操作会将所有数据加载到 Driver 端，当数据集较大时可能会造成内存溢出。

例 3.29 countByKey 操作。

```
wordAndOne.countByKey().foreach(println(_))
//**********运行结果**********
(hello,2)
(world,1)
(spark,1)
```

本操作适用于较小的数据集。若想处理较大的数据集，建议使用 map()及 reduceByKey 操作实现，这些方法返回的是分布式的 RDD 数据集，不会将数据全部加载到 Driver 端的内存。

3.10 RDD 的输出操作


视频讲解

在实际生产环境中，对 RDD 进行处理之后，通常需要将结果保存，以便后续环节的分析与应用。与读取类似，RDD 保存格式包括文本文件、Sequence 文件和 Object 文件等。如果需要操作 json 文件或 csv 文件，需要导入第三方包，此处不再赘述。常见的 RDD 输出操作如表 3.6 所示。

表 3.6　常见的 RDD 输出操作

RDD 输出操作	说　明
saveAsTextFile(path)	将 RDD 写入给定路径下的文本文件中(一个或多个文件,取决于分区数量),该操作支持本地文件系统、HDFS 或其他 Hadoop 可支持的任何文件系统。Spark 会调用 toString 方法将文件中的每一行文本转换成字符串进行保存
saveAsSequenceFile(path) (Java and Scala)	将 RDD 写入给定路径下的 Hadoop Sequence 文件中
saveAsObjectFile(path) (Java and Scala)	使用 Java 序列化将 RDD 写入 Object 文件中

输出操作中的路径参数必须是一个不存在的路径,如果该文件夹已存在,程序会报错。保存的文件个数与 RDD 的分区数相同,如果需要将所有数据保存到一个文件中,需要将该 RDD 的分区数重置为 1。

例 3.30　saveAsTextFile 操作。将单词计数的结果保存到本地文本文件中。

```
//直接保存到 data/wordcount 下
wordAndOne.reduceByKey((a,b)=>a+b).saveAsTextFile("data/wordcount")
//重新设置 1 个分区后保存到 data/wordcount2 下
wordAndOne.reduceByKey((a,b)=>a+b).repartition(1).saveAsTextFile("data/
wordcount2")
```

RDD 输出结果如图 3.22 所示。

在 wordcount 文件夹下,共产生了 17 个文件。其中扩展名为“.crc”的文件为校验文件,_SUCCESS 为状态标识文件,剩余的 8 个以“part”开头的文件为数据文件,分别代表 0～7 分区的数据。将分区数重置为 1 后,仅产生一个 part-00000 文件,RDD 输出结果如图 3.23 所示。

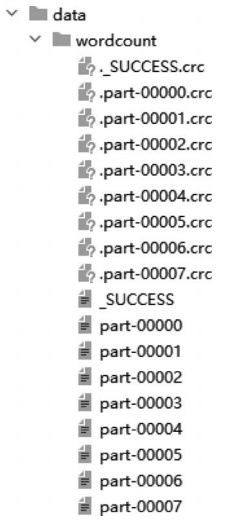

图 3.22　默认分区个数情况下的 RDD 输出结果

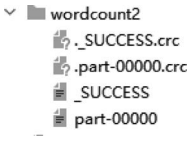

图 3.23　分区数重置为 1 后的 RDD 输出结果

【任务实现】

1. 步骤分析

在计算每个地区每种商品的销量时，可以使用键值对 RDD 进行聚合运算，需要将地区和商品联合起来作为键，统计每个地区的最高销量的商品时，又需要将地区取出作为键进行分组，因此该过程会涉及较多的 map()操作和一系列的分组、聚合操作，有助于更好地理解和运用键值对 RDD。

2. 完成任务

(1) 启动 IntelliJ IDEA，右击项目文件夹，选择相应的 package，新建 scala 类。

(2) 打开类文件，在代码编辑窗口中输入如下代码。

```
package edu.xpc.sparkcore
import org.apache.spark.{SparkConf, SparkContext}

object Rw3 {
 def main(args: Array[String]): Unit = {
  val sparkConf = new SparkConf().setAppName("WordCount").setMaster("local[*]")
  val sc = new SparkContext(sparkConf)
  val rddretail = sc.textFile("data/online_retail_II.txt")
  rddretail.map(x=>x.split("\t"))
   .filter(x=> !x(0).toLowerCase.startsWith("c"))
   .map(x=>((x(x.length-1),x(2)),x(3).toInt))  //((地区,商品名),销售数量)
   .reduceByKey(_+_)                           //求出每个地区每种商品的销售量
   .map(x=>(x._1._1,(x._1._2,x._2)))           //(地区,(商品名,销售数量))
   .groupByKey()                               //按地区分组
   .map(x=>{
    val country = x._1
    val list = x._2.toList.sortBy(x => -x._2)  //组内元素倒序排序
    (country,list(0))                          //(地区,最高销量的商品名及数量)
   })
.map(x=>x._1+","+x._2._1+","+x._2._2)
   .repartition(1)
   .saveAsTextFile("data/rw3")
 }
}
```

(3) 测试源代码，运行当前程序，部分结果文件内容如图 3.21 所示。

【任务总结】

通过本任务的学习，读者可以掌握键值对 RDD 的创建以及常用的操作。在使用键值对 RDD 时，需注意以下几点。

(1) 在做数据分析时，键值对 RDD 丰富的操作可以极大提高数据分析的效率，用户需

要根据具体需求确定键和值。

(2) reduce 聚合运算适用于求和、求最值等场景,不适用于求均值。因为均值不满足交换律和结合律。

(3) 涉及数学运算时,需要特别注意字段类型,必要时需要进行相应的类型转换。

【巩固练习】

一、单选题

1. 键值对 RDD 是指(　　)。

A. 一种特殊类型的 RDD　　B. 由键值对组成的数据集合

C. 只能存储整数键和字符串值　　D. RDD 的一种变换操作

2. 在 Spark 中,(　　)可以创建键值对 RDD。

A. 使用 collect 操作　　B. 使用 paralleize 操作

C. 使用 textFile 操作　　D. 使用 map()操作

3. 键值对 RDD 与普通单值 RDD 相比,其特点在于(　　)。

A. 键值对 RDD 可以进行更多类型的操作

B. 键值对 RDD 可以并行处理

C. 键值对 RDD 只支持整数键

D. 键值对 RDD 可以持久化到磁盘

4. 在键值对 RDD 中,(　　)可以对每个键的值进行累加操作。

A. 使用 collect 操作　　B. 使用 reduceByKey 操作

C. 使用 map()操作　　D. 使用 filter 操作

5. (　　)可以将两个键值对 RDD 进行关联操作。

A. 使用 collect 操作　　B. 使用 reduceByKey 操作

C. 使用 join 操作　　D. 使用 groupByKey 操作

二、判断题

1. 键值对 RDD 只支持基本的数学运算,如加法和减法。　　(　　)

2. 键值对 RDD 的值可以是一个列表或数组。　　(　　)

3. 键值对 RDD 可通过 collect 操作将数据收集到 Driver 端。　　(　　)

4. 键值对 RDD 的分区数量决定了计算任务的并行度。　　(　　)

5. 键值对 RDD 可以使用 groupByKey 操作将相同值的元素进行分组。　　(　　)

6. 可以使用 values 操作获取所有键值对的值。　　(　　)

三、编程题

1. 现有学生成绩存放在列表("bj1,kevin,82,76","bj1,mark,92,79","bj2,jack,85,69")中,计算每个班级的平均分。

2. 对于商品零售数据,计算该连锁商店在每个地区的总销售额,打印销售额最高的前 10 个地区。

【任务拓展】

分析电影评分数据有助于帮助人们了解电影的质量、受欢迎程度和观众口碑，对电影制作方、发行方和观众都具有重要的参考价值。

通过统计每年上映的电影数量和评分等指标，可以了解电影产业的发展趋势和受欢迎程度；通过计算导演执导的电影数量、评分数据和演员出演的电影分析，可以评估导演和演员的工作表现和知名度。

现有三张数据表，具体字段如表 3.7～表 3.9 所示。

表 3.7 电影表 movies.dat 数据字段说明

序　号	字 段 名	说　明
1	id	电影 ID
2	title	电影名
3	imdbID	imdbID
4	spanishTitle	西班牙语电影名
5	imdbPictureURL	imdb 图片地址
6	year	年份
7	rtID	烂番茄 id
8	rtAllCriticsRating	烂番茄收视率
9	rtAllCriticsNumReviews	烂番茄评论浏览数
10	rtAllCriticsNumFresh	烂番茄最新评论数
11	rtAllCriticsNumRotten	烂番茄总差评数
12	rtAllCriticsScore	烂番茄总评分
13	rtTopCriticsRating	烂番茄知名评论收视率
14	rtTopCriticsNumReviews	烂番茄知名评论浏览数
15	rtTopCriticsNumFresh	烂番茄最新知名评论数
16	rtTopCriticsNumRotten	烂番茄知名评论差评数
17	rtTopCriticsScore	烂番茄知名评论得分
18	rtAudienceRating	烂番茄观众收视率
19	rtAudienceNumRatings	烂番茄观众评价数
20	rtAudienceScore	烂番茄观众得分
21	rtPictureURL	烂番茄图片地址

表 3.8 电影导演表 movie_directors.dat 数据字段说明

序　号	字 段 名	说　明
1	movieID	电影 ID
2	directorID	导演 ID
3	directorName	导演名

表 3.9　电影演员表 movie_actors.dat 数据字段说明

序　号	字　段　名	说　明
1	movieID	电影 ID
2	actorID	演员 ID
3	actorName	演员名
4	ranking	排名

结合三张数据表，做如下数据分析。

1. 读取源数据，创建 RDD，并根据需求创建相应的键值对 RDD。

2. 统计每年上映的电影数量。

3. 计算每个导演执导的电影数量，并按照数量由高到低排序，结果包含导演名及电影数量。

4. 计算每个演员出演的电影数量，并按照数量由高到低排序，结果包含演员名及电影数量。

5. 统计每个年份烂番茄评分最高的电影。

6. 统计每个导演的电影平均烂番茄收视率，并按照收视率由高到低排序。

7. 计算每个导演的电影中，烂番茄评分最高的电影的评分，并将结果保存成文本文件。

任务 4　分区保存销售数据

【任务提出】

Spark RDD 的分区是指将数据集按照一定规则划分成多个逻辑数据块，每个分区都是一个独立的数据片段，能在集群的不同节点上并行处理。通过数据量大小和计算需求合理设置 RDD 的分区数量，可以优化 Spark 应用程序的性能和效率。分区数量过少可能导致负载不均衡，而分区数量过多会增加通信开销。默认情况下，Spark 会根据数据源的特性来确定 RDD 的分区数量。例如，对于文本文件，Spark 通常会对每一个输入文件创建一个分区，这样的默认分区策略可以确保每个输入文件都能被单独处理，从而实现了数据的并行处理。

Spark 的系统分区方式有两种，一种是哈希分区，另一种是范围分区。哈希分区是根据键值对 RDD 的键计算哈希值，然后利用该哈希值对分区数取模进行分区；范围分区是将一定范围的数据映射到一个分区中。用户也可以通过自定义分区器完成特定的分区要求。

本任务需要按照销售数据的年份进行分区，将销售数据保存到文本文件中。结果如图 3.24 和图 3.25 所示。

part-00000 × part-00001 × online_retail_II.txt ×

```
(2010,536365    85123A  WHITE HANGING HEART T-LIGHT HOLDER  6   2010-12-1 8:26  2.55    17850   United Kingdom)
(2010,536365    71053   WHITE METAL LANTERN 6   2010-12-1 8:26  3.39    17850   United Kingdom)
(2010,536365    84406B  CREAM CUPID HEARTS COAT HANGER  8   2010-12-1 8:26  2.75    17850   United Kingdom)
(2010,536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE 6   2010-12-1 8:26  3.39    17850   United Kingdom)
(2010,536365    84029E  RED WOOLLY HOTTIE WHITE HEART.  6   2010-12-1 8:26  3.39    17850   United Kingdom)
(2010,536365    22752   SET 7 BABUSHKA NESTING BOXES    2   2010-12-1 8:26  7.65    17850   United Kingdom)
(2010,536365    21730   GLASS STAR FROSTED T-LIGHT HOLDER   6   2010-12-1 8:26  4.25    17850   United Kingdom)
(2010,536366    22633   HAND WARMER UNION JACK  6   2010-12-1 8:28  1.85    17850   United Kingdom)
(2010,536366    22632   HAND WARMER RED POLKA DOT   6   2010-12-1 8:28  1.85    17850   United Kingdom)
(2010,536368    22960   JAM MAKING SET WITH JARS    6   2010-12-1 8:34  4.25    13047   United Kingdom)
(2010,536368    22913   RED COAT RACK PARIS FASHION 3   2010-12-1 8:34  4.95    13047   United Kingdom)
(2010,536368    22912   YELLOW COAT RACK PARIS FASHION  3   2010-12-1 8:34  4.95    13047   United Kingdom)
(2010,536368    22914   BLUE COAT RACK PARIS FASHION    3   2010-12-1 8:34  4.95    13047   United Kingdom)
(2010,536367    84879   ASSORTED COLOUR BIRD ORNAMENT   32  2010-12-1 8:34  1.69    13047   United Kingdom)
(2010,536367    22745   POPPY'S PLAYHOUSE BEDROOM   6   2010-12-1 8:34  2.1 13047   United Kingdom)
(2010,536367    22748   POPPY'S PLAYHOUSE KITCHEN   6   2010-12-1 8:34  2.1 13047   United Kingdom)
(2010,536367    22749   FELTCRAFT PRINCESS CHARLOTTE DOLL   8   2010-12-1 8:34  3.75    13047   United Kingdom)
(2010,536367    22310   IVORY KNITTED MUG COSY  6   2010-12-1 8:34  1.65    13047   United Kingdom)
(2010,536367    84969   BOX OF 6 ASSORTED COLOUR TEASPOONS  6   2010-12-1 8:34  4.25    13047   United Kingdom)
```

图 3.24 年份为 2010 的分区 0 数据集

part-00000 × part-00001 × online_retail_II.txt ×

```
(2011,560773    84877B  GREEN ROUND COMPACT MIRROR  2   2011-7-20 16:17 2.46        United Kingdom)
(2011,560773    84920   PINK FLOWER FABRIC PONY 1   2011-7-20 16:17 3.29        United Kingdom)
(2011,560773    84923   PINK BUTTERFLY HANDBAG W BOBBLES    1   2011-7-20 16:17 4.13        United Kingdom)
(2011,560773    84951A  SET OF 4 PISTACHIO LOVEBIRD COASTER 1   2011-7-20 16:17 2.46        United Kingdom)
(2011,560773    84969   BOX OF 6 ASSORTED COLOUR TEASPOONS  2   2011-7-20 16:17 3.29        United Kingdom)
(2011,560773    84971S  SMALL HEART FLOWERS HOOK    1   2011-7-20 16:17 1.63        United Kingdom)
(2011,560773    84988   SET OF 72 PINK HEART PAPER DOILIES  2   2011-7-20 16:17 2.92        United Kingdom)
(2011,560773    84991   60 TEATIME FAIRY CAKE CASES 1   2011-7-20 16:17 1.25        United Kingdom)
(2011,560773    85061W  WHITE JEWELLED HEART DECORATION 1   2011-7-20 16:17 1.63        United Kingdom)
(2011,560773    85086A  CANDY SPOT HEART DECORATION 2   2011-7-20 16:17 0.83        United Kingdom)
(2011,560773    85093   CANDY SPOT EGG WARMER HARE  1   2011-7-20 16:17 0.83        United Kingdom)
(2011,560773    85095   THREE CANVAS LUGGAGE TAGS   1   2011-7-20 16:17 0.83        United Kingdom)
(2011,560773    85099B  JUMBO BAG RED RETROSPOT 6   2011-7-20 16:17 2.08        United Kingdom)
(2011,560773    85099F  JUMBO BAG STRAWBERRY    3   2011-7-20 16:17 2.08        United Kingdom)
(2011,560773    85123A  WHITE HANGING HEART T-LIGHT HOLDER  1   2011-7-20 16:17 5.79        United Kingdom)
(2011,560773    85131B  BEADED CRYSTAL HEART GREEN ON STICK 1   2011-7-20 16:17 0.83        United Kingdom)
(2011,560773    85132C  CHARLIE AND LOLA FIGURES TINS   1   2011-7-20 16:17 5.79        United Kingdom)
(2011,560773    85176   SEWING SUSAN 21 NEEDLE SET  4   2011-7-20 16:17 0.79        United Kingdom)
```

图 3.25 年份为 2011 的分区 1 数据集

【任务分析】

本任务需要将数据按照年份实现自定义分区。Spark RDD 只能对键值对 RDD 设置分区方式，因此需要先提取数据中的年份信息，将其作为键，零售数据作为值，将原始 RDD 转换成键值对 RDD，然后编写自定义分区器，最后将数据保存到文本文件中。

【知识准备】

3.11 自定义分区器

实现自定义分区器，需要继承 org. apache. spark. Partitioner 类并实现其中的两个方法。

(1) def numPartitions：Int：返回需要创建的分区个数。

(2) def getPartition(key：Any)：Int：对输入的键进行处理，返回该键的分区 ID。分


视频讲解

区 ID 范围是 0～(numPartitions－1)。

例 3.31 以班级为分区保存数据。现有一份学生列表,包含班级、学号、姓名字段,现需要以班级作为分区依据,将班级及学生姓名保存到文本文件中。

```
package edu.xpc.sparkcore

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

object PartitionDemo {
 def main(args: Array[String]): Unit = {
  val sparkConf = new SparkConf().setAppName("partitioner").setMaster("local[*]")
  val sc = new SparkContext(sparkConf)
  val stu = List("bj1,01,kevin","bj1,02,lucy","bj2,01,mark")
  val rddstu = sc.parallelize(stu).map(x=>x.split(",")).map(x=>(x(0),x(2)))
  val c = rddstu.keys.distinct().count().toInt
  rddstu.partitionBy(new MyPartitioner(c)).saveAsTextFile("data/stuparts")
 }
 class MyPartitioner(numParts:Int) extends Partitioner{
  override def numPartitions: Int = numParts

  override def getPartition(key: Any): Int = {
   key.hashCode() % numParts
  }
 }
}
```

上述示例中,定义了一个类 MyPartitioner,继承 Partitioner,并重写了 numPartitions()及 getPartition()方法。getPartition()方法根据键的哈希值对分区个数求模返回分区值。在 main()方法中,读取原始数据转换为 RDD 后,使用 map()方法将数据转换为键值对 RDD,键为班级(即分区字段),值为学生姓名,常量 c 为班级数(即分区数),将自定义类对象作为参数调用 partitionBy()方法,重新设置分区后保存为文本文件。结果如图 3.26 所示。

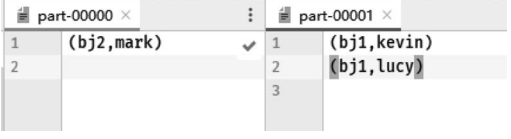

图 3.26 按班级进行分区运算结果

【任务实现】

1. 步骤分析

首先读取数据创建 RDD 后,将数据转换成键值对 RDD,并且以分区字段作为值。编写自定义分区类,当键值为不同年份时,设置对应的分区值,最后调用 partitionBy()方法实现自定义分区保存。

2. 完成任务

(1) 启动 IntelliJ IDEA,右击项目文件夹,选择相应的 package,新建 scala 类。

(2) 打开类文件,在代码编辑窗口中输入如下代码。

```
package edu.xpc.sparkcore
import org.apache.spark.{Partitioner, SparkConf, SparkContext}

object Rw4 {
 def main(args: Array[String]): Unit = {
  val sparkConf = new SparkConf().setAppName("Rw4").setMaster("local[*]")
  val sc = new SparkContext(sparkConf)
  val rddretail = sc.textFile("data/online_retail_II.txt")
  rddretail.map(s=>s.split("\t"))
   .map(x=>(x(4).substring(0,4),x.mkString("\t")))
   .partitionBy(new MyPartition(2))
   .saveAsTextFile("data/retail_parts")
  sc.stop()
 }
 class MyPartition(numParts:Int) extends Partitioner{
  override def numPartitions: Int = numParts

  override def getPartition(key: Any): Int = {
   key match {
    case "2010" => 0
    case "2011" => 1
    case _ => 0
   }
  }
 }
}
```

(3) 测试源代码,运行当前程序,部分结果文件内容如图 3.24 和图 3.25 所示。

【任务总结】

通过本任务的学习,读者可以掌握如何实现自定义分区。在对数据进行分区时,需要注意以下几点。

(1) 自定义分区器需要继承 org.apache.spark.Partitioner 类,并且实现其中的 numPartitions()和 getPartition()两个方法。

(2) 在 numPartitions()方法中,需要指定 RDD 的分区数目,需要确保分区数合理,以充分利用集群资源并避免数据倾斜。

(3) 在 getPartition()方法中,根据元素的键计算分区逻辑,返回每个元素应该被分配到的分区编号。需要合理设计数据的特征或键,保证正确分区。

【巩固练习】

一、单选题

1. 自定义 Spark RDD 分区器需要继承(　　)类。

A. org. apache. spark. RDDPartitioner　　B. org. apache. spark. Partition

C. org. apache. spark. Partitioner　　D. org. apache. spark. PartitionFunction

2. 自定义分区器中必须实现(　　)方法来指定 RDD 的分区数目。

A. numPartitions()　　B. getPartitionsCount()

C. partitionCount()　　D. countPartitions()

3. 在自定义分区器中,getPartition()方法的作用是(　　)。

A. 返回 RDD 的分区数目

B. 返回 RDD 中每个元素所在的分区编号

C. 返回 RDD 的分区索引

D. 返回 RDD 的分区器类型

4. 在自定义分区器中,相同的键应该被映射到(　　)。

A. 相同的节点　　B. 相同的 RDD 分区

C. 不同的节点　　D. 不同的 RDD 分区

二、判断题

1. 自定义分区器可以通过重写 hashCode()方法来实现分区逻辑。(　　)

2. 自定义分区器可以在运行时动态调整分区数目。(　　)

3. 自定义分区器的性能调优应该避免复杂的计算。(　　)

4. 自定义分区器中的 getPartition()方法返回的是一个分区对象。(　　)

5. 在自定义分区器中,分区数目的合理性对性能没有影响。(　　)

6. 自定义分区器中必须实现 getPartition()方法来指定 RDD 的分区数目。(　　)

三、编程题

现有数据存放在列表(1,2,3,4,5)中,将数据按照奇偶性分区保存。

【任务拓展】

现有一份全国城市平均气温数据,具体字段如表 3.10 所示。

表 3.10　平均气温数据字段说明

序　号	字 段 名	说　明
1	No	数据行号
2	year	数据所在年份
3	province	省份
4	provinceID	省份 ID

续表

序　号	字 段 名	说　明
5	city	城市
6	cityID	城市 ID
7	avgtemp	平均气温

根据城市的平均气温对城市分类。如果该城市平均气温小于或等于 10℃,则将该城市标记为“cold”;如果平均气温在 10～20℃,则将该城市标记为“warm”;否则标记为“hot”。按照不同的标记将数据分区存放。