第3章 热门品类Top10分析

学习目标

- 了解数据集分析，能够描述用户行为数据中包含的信息；
- 熟悉实现思路分析，能够描述热门品类 Top10 分析的实现思路；
- 掌握实现热门品类 Top10 分析，能够编写用于实现热门品类 Top10 分析的 Spark 程序；
- 掌握运行 Spark 程序，能够将 Spark 程序提交到 YARN 集群运行。

品类是指商品所属的分类，例如服装、电子产品、图书等。进行热门品类 Top10 分析的目的在于从用户行为数据中挖掘出排名前 10 的最受用户喜爱的品类。通过对热门品类的了解，企业可以调整其销售策略，提高对这些品类的投入，优化促销活动，从而更好地满足消费者需求，提高销售额。本章将讲解如何对电商网站的用户行为数据进行分析，从而统计出排名前 10 的品类。

3.1 数据集分析

热门品类 Top10 分析使用的数据集为某电商网站在 2022 年 11 月产生的用户行为数据，这些数据存储在文件 user_session.txt 中，该文件的每一行数据都记录了商品和用户相关的特定行为。下面，以文件 user_session.txt 中的一条用户行为数据为例进行详细分析，具体内容如下。

```
{"user_session":"000007b4-6d31-4590-b88f-0f68d1cee73c","event_type":"view","category_id":"2053013554415534427","user_id":"572115980","product_id":"1801873","address_name":"NewYork","event_time":"2022-11-16 08:12:52"}
```

从上述内容中可以看出，文件 user_session.txt 中的每一条用户行为数据都以 JSON 对象的形式存在，该对象包含多个键值对，每个键值对代表着不同的信息。下面，通过解读这些键，介绍用户行为数据中各项信息的含义。

- user_session：表示用户行为的唯一标识。
- event_type：表示用户行为的类型，包括 view（查看）、cart（加入购物车）和 purchase（购买）。

- category_id：表示品类的唯一标识。
- user_id：表示用户的唯一标识。
- product_id：表示商品的唯一标识。
- address_name：表示产生用户行为的区域。
- event_time：表示产生用户行为的时间。

3.2　实现思路分析

实现热门品类 Top10 分析的核心在于统计不同品类的商品被查看、加入购物车和购买的次数，然后按照特定排序规则对统计结果进行处理，以获取排名前 10 的热门品类。在本项目中，实现热门品类 Top10 分析的排序规则如下。首先，根据不同品类的商品被查看的次数进行降序排序；然后，根据不同品类的商品被加入购物车的次数进行降序排序；最后，根据不同品类的商品被购买的次数进行降序排序。下面，通过图 3-1 详细描述本项目中热门品类 Top10 分析的实现思路。

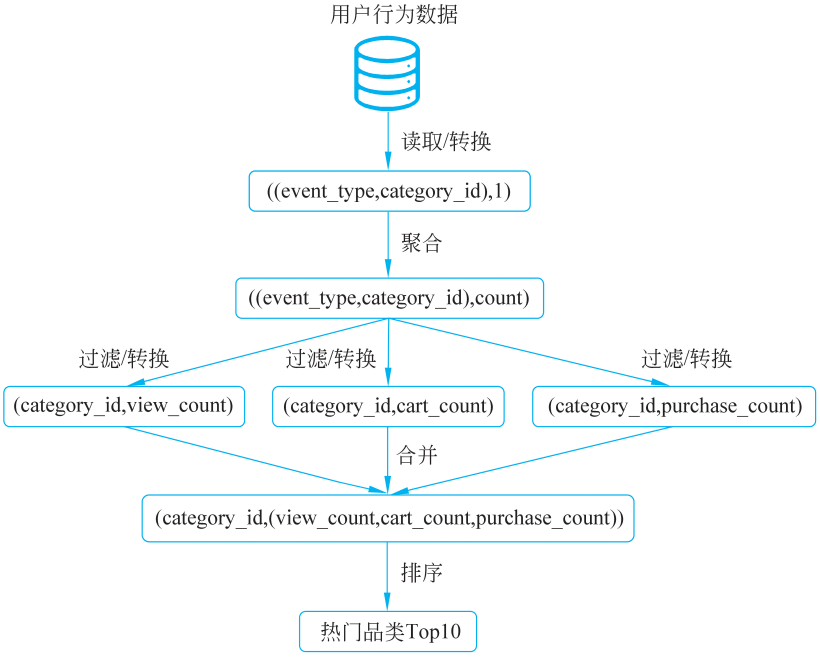

图 3-1　热门品类 Top10 分析的实现思路

针对热门品类 Top10 分析的实现思路进行如下讲解。

- 读取/转换：读取用户行为数据，提取其中的用户行为类型(event_type)和品类唯一标识(category_id)。为了便于后续统计不同品类的商品被查看、加入购物车和购买的次数，我们将提取的数据转换为元组。元组的第一个元素包含用户行为类型和品类唯一标识，第二个元素为 1，用于标识当前品类的商品被查看、加入购物车或购买的次数。

- 聚合：统计不同品类的商品被查看、加入购物车和购买的次数，并生成新的元组。该元组的第一个元素包含用户行为类型和品类唯一标识，第二个元素为当前品类的商品被查看、加入购物车或购买的次数的统计结果(count)。
- 过滤/转换：为了方便后续识别不同品类商品在不同用户行为下的统计结果，我们根据用户行为类型将聚合结果划分为三部分。这样，我们可以分别得到各品类商品被查看、加入购物车和购买次数的统计结果。接着，我们对过滤得到的结果进行转换，生成新的元组。该元组的第一个元素为品类唯一标识，第二个元素为当前品类的商品被查看(view_count)、加入购物车(cart_count)或购买(purchase_count)的次数的统计结果。
- 合并：根据商品的唯一标识对过滤/转换得到的三部分数据进行合并，生成新的元组。该元组的第一个元素为商品的唯一标识；第二个元素包含当前品类的商品被查看、加入购物车和购买的次数的统计结果。
- 排序：根据排序规则对合并结果进行降序排序，并获取排序结果的前 10 行数据，从而得到热门品类 Top10。

3.3 实现热门品类 Top10 分析

3.3.1 环境准备

在进行某项事务前，充足的准备能够使我们更好地发挥自己的潜力，提高自身能力和素质。这包括深入了解相关知识，积累相关经验，以及适时地进行规划。通过充分准备，我们能够为自己创造更多机会，增加成功的可能性，并提高对事务的把控能力和执行效果。

本项目主要使用 Scala 语言在集成开发工具 IntelliJ IDEA 中实现 Spark 程序。在开始实现 Spark 程序之前，需要在本计算机中安装 JDK 和 Scala 并配置系统环境变量，以及在 IntelliJ IDEA 中安装 Scala 插件、创建项目和导入依赖。关于安装 JDK 和 Scala 并配置系统环境变量的操作读者可参考本书提供的补充文档。接下来，主要以 IntelliJ IDEA 中执行的一系列相关操作进行讲解，具体内容如下。

1. 安装 Scala 插件

默认情况下，IntelliJ IDEA 并不支持 Scala 语言。因此，在集成开发工具 IntelliJ IDEA 中使用 Scala 语言实现 Spark 程序前，需要通过安装 Scala 插件来添加相应的支持。接下来，将讲解如何在 IntelliJ IDEA 中安装 Scala 插件，具体操作步骤如下。

(1) 打开 IntelliJ IDEA，进入 Welcome to IntelliJ IDEA 界面，如图 3-2 所示。

(2) 在图 3-2 所示界面中，单击 Plugins 选项，在对话框中部的搜索栏内输入 Scala，搜索 Scala 相关插件，如图 3-3 所示。

需要说明的是，如果读者在打开 IntelliJ IDEA 时，直接进入具体项目的界面，那么可以在 IntelliJ IDEA 的工具栏中依次单击 File、Settings 选项打开 Settings 对话框，在该对话框的左侧单击 Plugins 选项进行搜索 Scala 相关插件的操作。

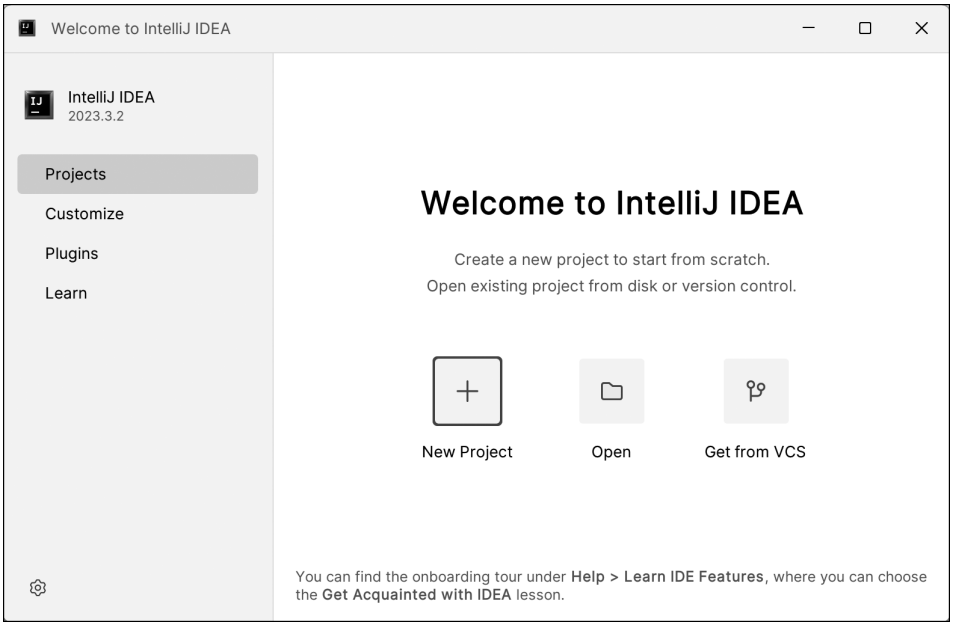

图 3-2　Welcome to IntelliJ IDEA 界面

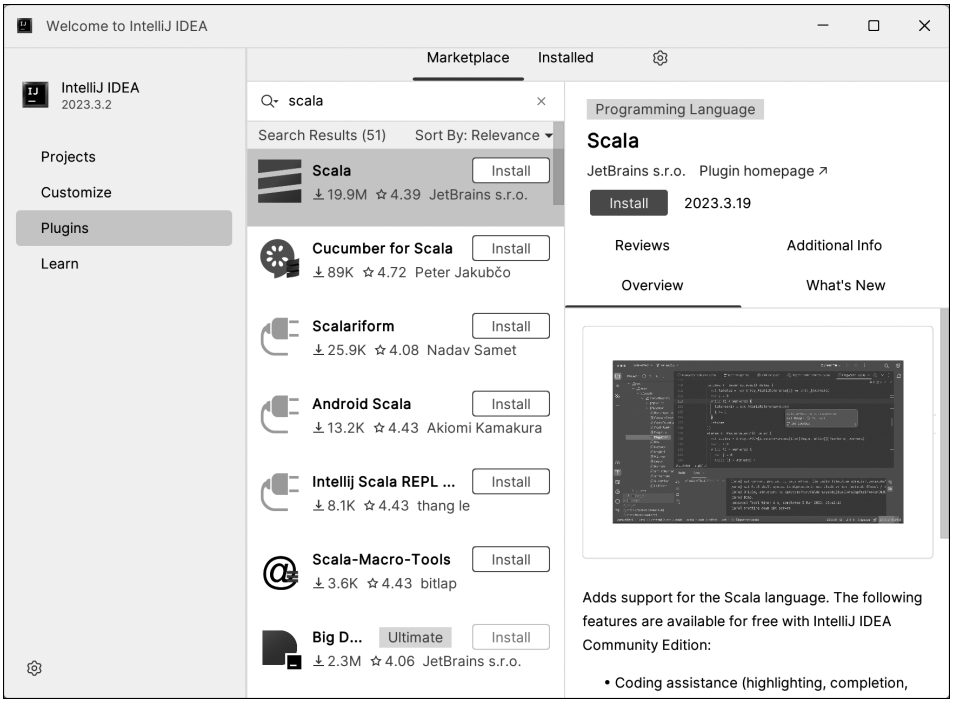

图 3-3　搜索 Scala 相关插件

(3) 在图 3-3 所示界面中，找到名为 Scala 的插件，单击其右方的 Install 按钮安装 Scala 插件。Scala 插件安装完成的效果如图 3-4 所示。

在图 3-4 所示界面中，单击 Restart IDE 按钮，打开 IntelliJ IDEA and Plugin Updates

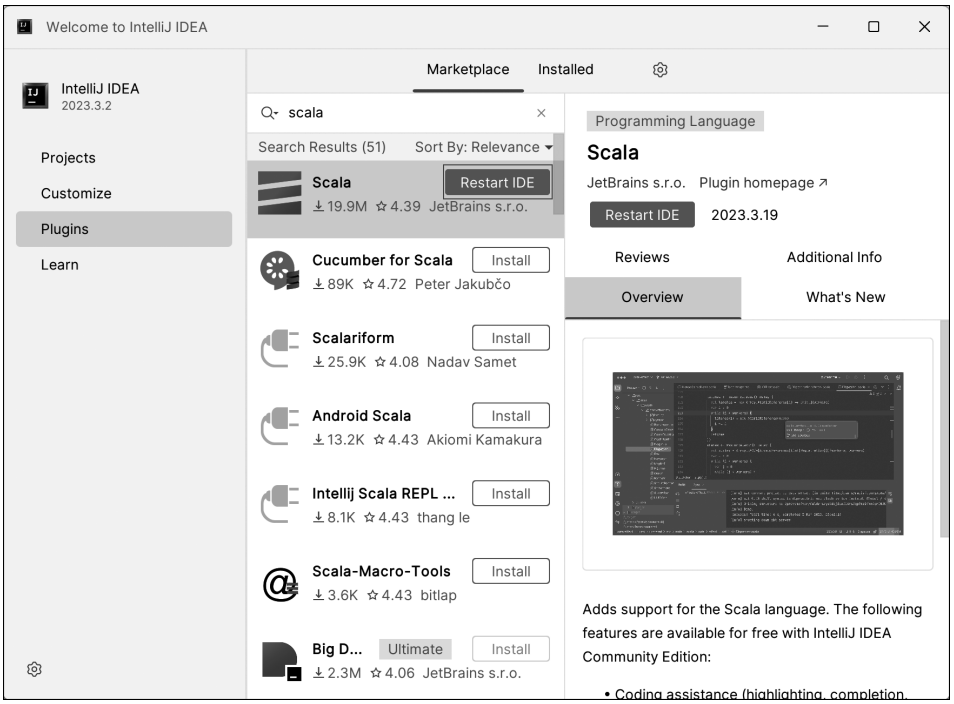

图 3-4 Scala 插件安装完成的效果

对话框。在该对话框中单击 Restart IDE 按钮重启 IntelliJ IDEA 使 Scala 插件生效。

至此完成了在 IntelliJ IDEA 中安装 Scala 插件的操作。

需要注意的是,搜索 Scala 相关插件的操作需要确保本地计算机处于联网状态。如果读者在进行搜索 Scala 相关插件的操作时,网络连接正常,但无法显示搜索结果,那么可以在图 3-4 中单击⚙按钮,在弹出的菜单中选择 HTTP Proxy Settings 选项,打开 HTTP Proxy 对话框,在该对话框中进行相关配置。HTTP Proxy 对话框配置完成的效果如图 3-5 所示。

在图 3-5 所示界面中,单击 OK 按钮后,重新打开 IntelliJ IDEA,再次尝试搜索 Scala 相关插件的操作。

2. 创建项目

在 IntelliJ IDEA 中基于 Maven 创建项目 SparkProject,具体操作步骤如下。

(1) 在 IntelliJ IDEA 的 Welcome to IntelliJ IDEA 界面中,单击 New Project 按钮,打开 New Project 对话框,在该对话框中配置项目的基本信息,具体内容如下。

① 在 Name 输入框中指定项目名称为 SparkProject。

② 在 Location 输入框中指定项目的存储路径为 D:\develop。

③ 在 JDK 下拉框中选择使用的 JDK 为本地安装的 JDK。

④ 在 Archetype 下拉框中选择 Maven 项目的模板为 org.apache.maven.archetypes:maven-archetype-quickstart。

New Project 对话框配置完成的效果如图 3-6 所示。

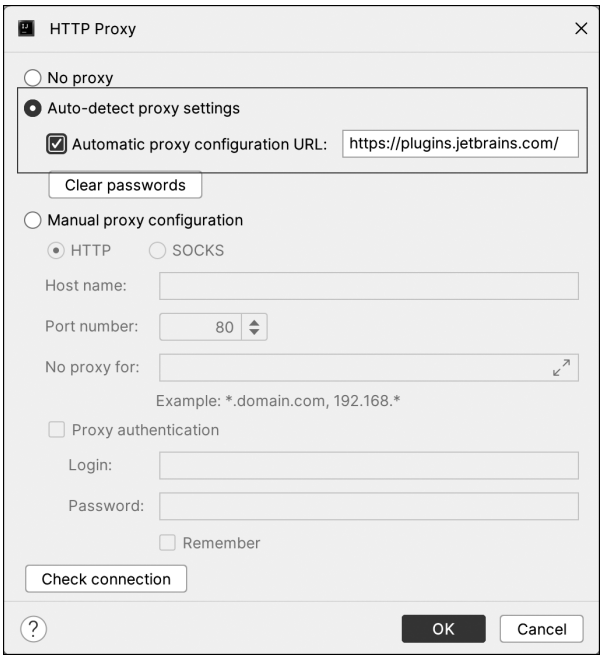

图 3-5　HTTP Proxy 对话框配置完成的效果

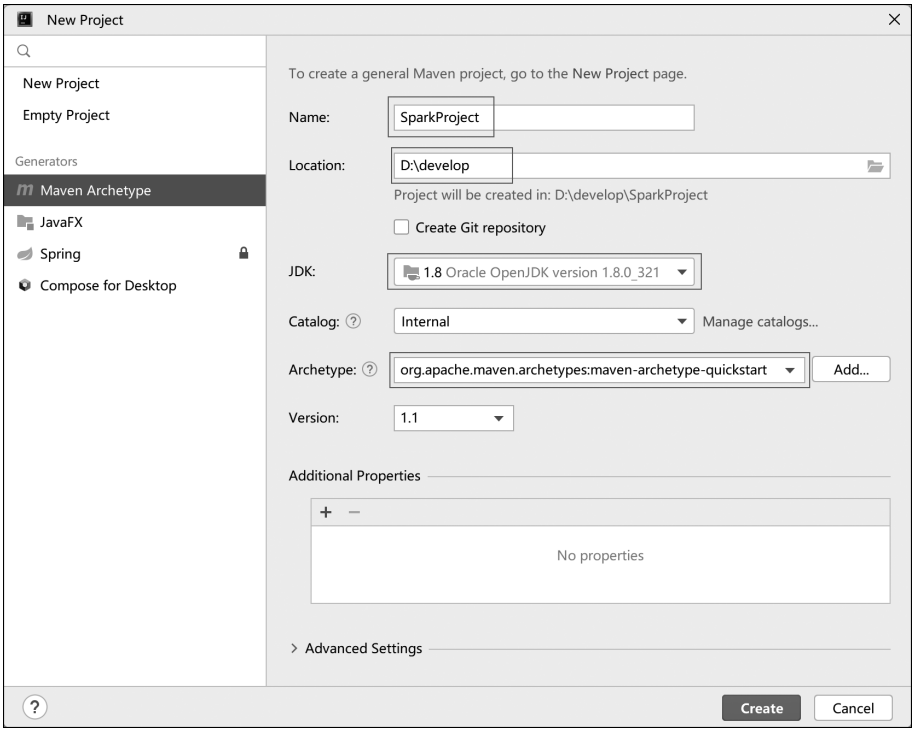

图 3-6　New Project 对话框配置完成的效果

需要说明的是,根据 IntelliJ IDEA 版本的不同,New Project 对话框显示的内容会存在差异。读者在创建项目时,需要根据实际显示的内容来配置项目的基本信息。

(2) 在图 3-6 所示界面中,单击 Create 按钮创建项目 SparkProject。项目 SparkProject 创建完成的效果如图 3-7 所示。

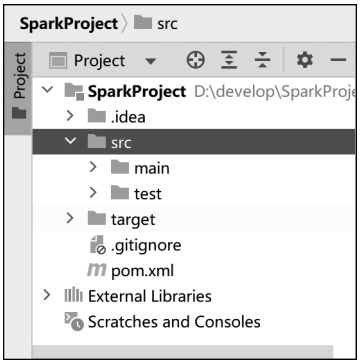

图 3-7 项目 SparkProject 创建完成的效果

(3) 在 SparkProject 项目的 src/main 目录中新建一个名为 scala 的文件夹,该文件夹将用于存放与本项目相关的 Scala 源代码文件,操作步骤如下。

① 选中并右击 main 文件夹,在弹出的菜单中依次单击 New、Directory 选项打开 New Directory 对话框。在该对话框的输入框中输入 scala,如图 3-8 所示。

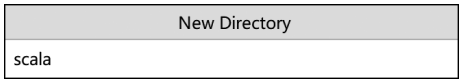

图 3-8 New Directory 对话框

② 在图 3-8 所示界面中按 Enter 键创建 scala 文件夹,如图 3-9 所示。

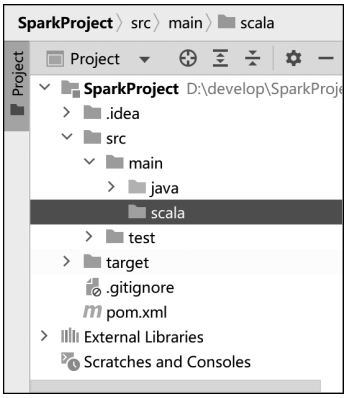

图 3-9 创建 scala 文件夹

(4) 新建的 scala 文件夹需要被标记为 Sources Root(源代码根目录)才可以存放 Scala 源代码文件。在图 3-9 所示界面中,选中并右击 scala 文件夹,在弹出的菜单依次单击 Mark Directory as、Sources Root 选项。成功标记为 Sources Root 后,scala 文件夹的颜色将变为蓝色。

（5）由于项目 SparkProject 是基于 Maven 创建的，默认并不提供对 Scala 语言的支持，所以需要为项目 SparkProject 添加 Scala SDK，操作步骤如下。

① 在 IntelliJ IDEA 的工具栏中依次单击 File、Project Structure 选项，会弹出 Project Structure 对话框，单击该对话框内左侧的 Libraries 选项，如图 3-10 所示。

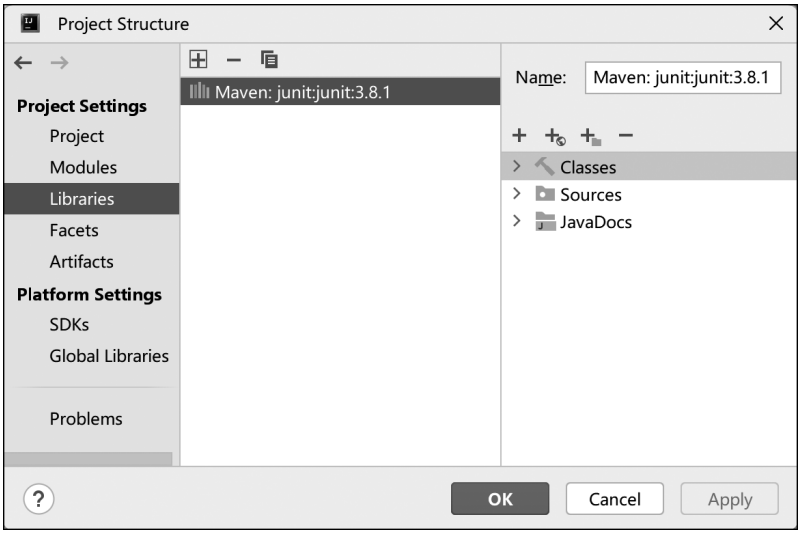

图 3-10　Project Structure 对话框

② 在图 3-10 所示界面中，单击上方的＋按钮并在弹出的菜单栏中选择 Scala SDK 选项，会弹出 Select JAR's for the new Scala SDK 对话框，在该对话框中选择 Location 为 System 的一项，表示通过选择本地操作系统中安装的 Scala 添加 Scala SDK，如图 3-11 所示。

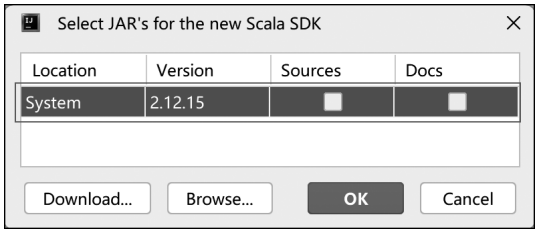

图 3-11　Select JAR's for the new Scala SDK 对话框

需要说明的是，若图 3-11 中未显示本地操作系统安装的 Scala，则可以单击 Browse 按钮通过浏览本地文件系统中 Scala 的安装目录添加 Scala SDK。

③ 在图 3-11 所示界面中，单击 OK 按钮，会弹出 Choose Modules 对话框，如图 3-12 所示。

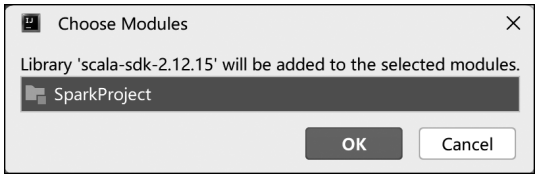

图 3-12　Choose Modules 对话框

在图 3-12 中,选择项目 SparkProject 后单击 OK 按钮返回 Project Structure 对话框。在该对话框中单击 Apply 按钮后单击 OK 按钮关闭 Project Structure 对话框。

3. 导入依赖和插件

在项目 SparkProject 的配置文件 pom.xml 中,添加用于实现本需求所需的依赖和插件。依赖添加完成的效果如文件 3-1 所示。

文件 3-1　pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>cn.itcast</groupId>
  <artifactId>SparkProject</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>SparkProject</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.3.0</version>
    </dependency>
    <dependency>
      <groupId>org.json</groupId>
      <artifactId>json</artifactId>
      <version>20230227</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-shaded-client</artifactId>
      <version>2.4.9</version>
    </dependency>
```

```
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-common</artifactId>
      <version>3.3.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>net.alchim31.maven</groupId>
        <artifactId>scala-maven-plugin</artifactId>
        <version>3.2.2</version>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>3.1.0</version>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

在文件 3-1 中,第 22～41 行代码用于添加实现本需求所需的依赖。其中,第 22～26 行代码添加的依赖为 Spark 核心依赖,第 27～31 行代码添加的依赖为 JSON 依赖,第 32～36 行代码添加的依赖为 HBase 客户端依赖,第 37～41 行代码添加的依赖为 Hadoop 核心依赖。

第 43～77 行代码用于添加实现本需求所需的插件。其中,第 45～56 行代码添加的 scala-maven-plugin 插件用于支持 Scala 编译和构建 Scala 项目,第 57～75 行代码添加的 maven-assembly-plugin 插件用于将项目的所有依赖和资源打包成一个独立的可执行的 jar 文件。

依赖添加完成后,确认添加的依赖是否存在于项目 SparkProject 中,在 IntelliJ IDEA 主界面的右侧单击 Maven 选项卡展开 Maven 面板,在 Maven 面板双击 Dependencies 折叠项,如图 3-13 所示。

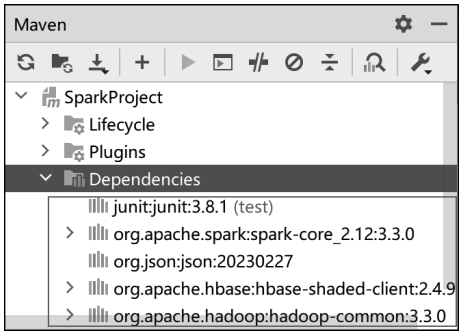

图 3-13 Maven 面板

从图 3-13 中可以看出,依赖已经成功添加到项目 SparkProject 中。如果这里未显示添加的依赖,则可以在图 3-13 中单击 按钮重新加载 pom.xml 文件。

3.3.2 实现 Spark 程序

在项目 SparkProject 的 src/main/scala 目录下,新建了一个名为 cn.itcast.top10 的包。在 cn.itcast.top10 包中创建一个名为 CategoryTop10 的 Scala 单例对象,在该单例对象中实现热门品类 Top10 分析的 Spark 程序,具体实现过程如下。

(1) 在单例对象 CategoryTop10 中添加 main()方法,用于定义 Spark 程序的实现逻辑,具体代码如文件 3-2 所示。

文件 3-2 CategoryTop10.scala

```
1  package cn.itcast.top10
2  object CategoryTop10 {
3    def main(args: Array[String]): Unit ={
4        //实现逻辑
5    }
6  }
```

(2) 在 Spark 程序中创建 SparkConf 对象 conf,用于配置 Spark 程序的参数。在单

例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val conf =new SparkConf().setAppName("CategoryTop10")
```

上述代码指定 Spark 程序的名称为 CategoryTop10。

(3) 在 Spark 程序中基于 SparkConf 对象创建 SparkContext 对象 sc,用于管理 Spark 程序的执行。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

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

(4) 在 Spark 程序中,通过 SparkContext 对象 sc 的 textFile()方法从文件系统中读取用户行为数据,并将其存储到 RDD 对象 textFileRDD 中。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val textFileRDD =sc.textFile(args(0))
```

上述代码中,使用 args(0)来代替用户行为数据的具体路径,以便于将 Spark 程序提交到 YARN 集群运行时,可以更加灵活地通过 spark-submit 命令的参数来指定用户行为数据的具体路径。

(5) 在 Spark 程序中,通过 map 算子对 RDD 对象 textFileRDD 进行转换操作,并将转换操作的结果存储到 RDD 对象 transformRDD。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
1  val transformRDD =textFileRDD.map(s =>{
2     //将读取的用户行为数据转换为 JSON 对象 json
3     val json =new JSONObject(s)
4     val category_id =json.getString("category_id")
5     val event_type =json.getString("event_type")
6     ((category_id, event_type), 1)
7  })
```

上述代码用于从用户行为数据中提取用户行为类型和品类唯一标识,并将其映射为包含两个元素的元组,该元组的第一个元素包含用户行为类型和品类唯一标识,第二个元素为 1。

(6) 在 Spark 程序中,通过 reduceByKey 算子对 RDD 对象 transformRDD 进行聚合操作,并将聚合操作的结果存储到 RDD 对象 aggregationRDD。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val aggregationRDD =transformRDD.reduceByKey(_ +_).cache()
```

上述代码中的聚合操作用于统计不同品类的商品被查看、加入购物车和购买的次数。由于后续需要对 RDD 对象 aggregationRDD 进行多次过滤操作,所以通过 cache()方法

将 RDD 对象 aggregationRDD 缓存到内存中。

(7) 在 Spark 程序中,通过 filter 算子对 RDD 对象 aggregationRDD 进行过滤操作,获取不同品类中商品被查看的次数,然后通过 map 算子对过滤操作的结果进行转换操作,并将转换操作的结果存储到 RDD 对象 getViewCategoryRDD。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val getViewCategoryRDD =aggregationRDD.filter(
action =>action._1._2 =="view"
).map(action =>(action._1._1, action._2))
```

上述代码中的转换操作用于将过滤结果映射为包含两个元素的元组,该元组的第一个元素为品类唯一标识,第二个元素为当前品类的商品被查看的次数。

(8) 在 Spark 程序中,通过 filter 算子对 RDD 对象 aggregationRDD 进行过滤操作,获取不同品类商品被加入购物车的次数,然后通过 map 算子对过滤操作的结果进行转换操作,并将转换操作的结果存储到 RDD 对象 getCartCategoryRDD。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val getCartCategoryRDD =aggregationRDD.filter(
  action =>action._1._2 =="cart"
).map(action =>(action._1._1, action._2))
```

上述代码中的转换操作用于将过滤结果映射为包含两个元素的元组,该元组的第一个元素为品类唯一标识,第二个元素为当前品类的商品被加入购物车的次数。

(9) 在 Spark 程序中,通过 filter 算子对 RDD 对象 aggregationRDD 进行过滤操作,获取不同品类商品被购买的次数,然后通过 map 算子对过滤操作的结果进行转换操作,并将转换操作的结果存储到 RDD 对象 getPurchaseCategoryRDD。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val getPurchaseCategoryRDD =aggregationRDD.filter(
  action =>action._1._2 =="purchase"
).map(action =>(action._1._1, action._2))
```

上述代码中的转换操作用于将过滤结果映射为包含两个元素的元组,该元组的第一个元素为品类唯一标识,第二个元素为当前品类的商品被购买的次数。

(10) 在 Spark 程序中,通过 leftOuterJoin 算子对 RDD 对象 getViewCategoryRDD、getCartCategoryRDD 和 getPurchaseCategoryRDD 进行合并操作,并将合并操作的最终结果存储到 RDD 对象 joinCategoryRDD。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val tmpJoinCategoryRDD =getViewCategoryRDD
```

```
    .leftOuterJoin(getCartCategoryRDD)
val joinCategoryRDD =tmpJoinCategoryRDD
    .leftOuterJoin(getPurchaseCategoryRDD)
```

上述代码中的合并操作用于将相同品类商品的不同行为类型的统计结果合并到RDD的同一元素中，并通过这3个RDD对象合并的先后顺序，明确不同行为类型统计结果在元素中的位置。

(11) 根据热门品类Top10分析的排序规则对合并操作的结果进行排序操作，具体实现过程如下。

① 在项目SparkProject的包cn.itcast.top10中创建一个名为CategorySortKey的Scala类，该类CategorySortKey需要实现Java提供的接口Comparable和Serializable，前者用于实现对象的比较，后者用于将对象转换成字节流进行传输。此外，在类CategorySortKey中还须要重写接口Comparable的compareTo()方法定义对象的比较规则，具体代码如文件3-3所示。

文件 3-3　CategorySortKey.scala

```
class CategorySortKey(
  val viewCount: Int,
  val cartCount: Int,
  val purchaseCount: Int
) extends Comparable[CategorySortKey] with Serializable {
  override def compareTo(other: CategorySortKey): Int ={
    val viewComparison =Integer.compare(viewCount, other.viewCount)
    if (viewComparison !=0) {
      viewComparison
    } else {
      val cartComparison =Integer.compare(cartCount, other.cartCount)
      if (cartComparison !=0) {
        cartComparison
      } else {
        Integer.compare(purchaseCount, other.purchaseCount)
      }
    }
  }
}
```

上述代码中，第6～18行代码通过重写接口Comparable的compareTo()方法定义对象的比较规则。比较规则为，首先比较不同品类的商品被查看的次数(viewCount)，若比较结果不相等，则返回1(大于的比较关系)或－1(小于的比较关系)；若比较结果相等，则进一步比较不同品类的商品被加入购物车的次数(cartCount)；若比较结果不相等，则返回1或－1；若比较结果仍然相等，则继续比较不同品类商品被购买的次数(purchaseCount)。

② 在Spark程序中，通过map算子，将存储在RDD对象joinCategoryRDD中的不同

品类的商品被查看、加入购物车和购买的次数映射到类 CategorySortKey 中进行比较,并将比较结果存储到 RDD 对象 transCategoryRDD 中。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val transCategoryRDD =joinCategoryRDD.map ({
  case (category_id, ((viewcount, cartcountOpt), purchasecountOpt)) =>
    val cartcount =cartcountOpt.getOrElse(0).intValue()
    val purchasecount =purchasecountOpt.getOrElse(0).intValue()
    val sortKey =new CategorySortKey(viewcount, cartcount, purchasecount)
    (sortKey, category_id)
})
```

上述代码中,getOrElse()方法用于处理可能出现的空值问题,防止空指针异常的发生。如果 getOrElse()方法的返回值为 None,则会将其替换为指定的默认值,这里指定的默认值为 0。

③ 在 Spark 程序中,通过 sortByKey 算子对 RDD 对象 transCategoryRDD 进行降序排序,并将降序排序的结果存储到 RDD 对象 sortedCategoryRDD 中。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val sortedCategoryRDD =transCategoryRDD.sortByKey(false)
```

④ 在 Spark 程序中,通过 take()方法获取 RDD 对象 sortedCategoryRDD 的前 10 个元素,即热门品类 Top10 分析的结果,并将这 10 个元素存储在数组 top10Category 中。在单例对象 CategoryTop10 的 main()方法中添加如下代码。

```
val top10Category =sortedCategoryRDD.take(10)
```

3.3.3 数据持久化

通过上一节内容实现的 Spark 程序仅仅获取了热门品类 Top10 的分析结果。为了便于后续进行数据可视化,并确保分析结果的长期存储,需要进行数据持久化操作。本项目使用 HBase 作为数据持久化工具。接下来,分步骤讲解如何将热门品类 Top10 的分析结果存储到 HBase 的表中,具体操作步骤如下。

(1) 为了避免实现本项目后续需求的数据持久化操作时,重复编写操作 HBase 的相关代码,这里在项目 SparkProject 的 src/main/scala 目录中,新建了一个名为 cn.itcast.hbase 的包。在包 cn.itcast.hbase 中创建一个名为 HBaseConnect 的 Scala 单例对象,在该单例对象中实现操作 HBase 的相关代码,具体如文件 3-4 所示。

文件 3-4 HBaseConnect.scala

```
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.hbase._
import org.apache.hadoop.hbase.client._
```

```
4  import java.io.IOException
5  object HBaseConnect {
6    //创建 Configuration 对象 hbaseConf,用于指定 HBase 的相关配置
7    val hbaseConf: Configuration =HBaseConfiguration.create()
8    //指定 ZooKeeper 集群中每个 ZooKeeper 服务的地址
9    hbaseConf.set("hbase.zookeeper.quorum", "spark01,spark02,spark03")
10   //指定 ZooKeeper 服务的端口号
11   hbaseConf.set("hbase.zookeeper.property.clientPort", "2181")
12   var conn: Connection =_
13   try {
14     //根据 HBase 的配置信息创建 HBase 连接
15     conn =ConnectionFactory.createConnection(hbaseConf)
16   }
17   catch {
18     case e: IOException =>e.printStackTrace()
19   }
20   def getHBaseAdmin: Admin ={
21     var hbaseAdmin: Admin =null
22     try {
23       hbaseAdmin =conn.getAdmin
24     } catch {
25       case e: MasterNotRunningException =>e.printStackTrace()
26       case e: ZooKeeperConnectionException =>e.printStackTrace()
27     }
28     hbaseAdmin
29   }
30   def getConnection: Connection =conn
31   def closeConnection(): Unit ={
32     if (conn !=null) {
33       try {
34         conn.close()
35       } catch {
36       case e: IOException =>e.printStackTrace()
37       }
38     }
39   }
40 }
```

上述代码中,第 20～29 行代码定义的 getHBaseAdmin()方法用于通过 HBase 连接获取 Admin 对象,该对象用于操作 HBase 的表。第 30 行代码定义的 getConnection()方法用于获取 HBase 连接。第 31～39 行代码定义的 closeConnection()方法用于关闭 HBase 连接以释放资源。

需要注意的是,如果在运行项目 SparkProject 的环境中未配置虚拟机 Spark01、

Spark02 和 Spark03 的主机名与 IP 地址的映射关系,那么在配置 ZooKeeper 集群地址时将主机名替换为具体的 IP 地址。

(2) 为了避免实现本项目后续需求的数据持久化操作时,重复编写操作 HBase 表的相关代码,这里在项目 SparkProject 的包 cn.itcast.hbase 中新建一个名为 HBaseUtils 的 Scala 单例对象,在该单例对象中实现操作 HBase 表的相关代码,具体如文件 3-5 所示。

文件 3-5 HBaseUtils.scala

```
import org.apache.hadoop.hbase.TableName
import org.apache.hadoop.hbase.client._
import org.apache.hadoop.hbase.util.Bytes
object HBaseUtils {
  def createtable(tableName: String, columnFamilys: String*): Unit ={
    //创建 Admin 对象 admin,用于操作 HBase 的表
    val admin: Admin =HBaseConnect.getHBaseAdmin
    //判断表是否存在
    if (admin.tableExists(TableName.valueOf(tableName))) {
      //禁用表
      admin.disableTable(TableName.valueOf(tableName))
      //删除表
      admin.deleteTable(TableName.valueOf(tableName))
    }
    //指定表名
    val tableDescriptorBuilder =TableDescriptorBuilder
      .newBuilder(TableName.valueOf(tableName))
    for (cf <-columnFamilys) {
        val columnDescriptor =ColumnFamilyDescriptorBuilder
      .newBuilder(Bytes.toBytes(cf)).build()
      //向表中添加列族
      tableDescriptorBuilder.setColumnFamily(columnDescriptor)
    }
    val tableDescriptor =tableDescriptorBuilder.build()
    //创建表
    admin.createTable(tableDescriptor)
    admin.close()
  }
  def putsToHBase(
                  tableName: String, rowkey: String,
                  cf: String, columns: Array[String],
                  values: Array[String]): Unit ={
    //基于表名创建 Table 对象 table,用于管理表的数据
    val table: Table =HBaseConnect.getConnection
      .getTable(TableName.valueOf(tableName))
```

```
36      //创建 Put 对象 puts,用于根据行键向表中插入数据
37      val puts: Put =new Put(Bytes.toBytes(rowkey))
38      for (i <-columns.indices) {
39        puts.addColumn(
40          //指定列族
41          Bytes.toBytes(cf),
42          //指定列
43          Bytes.toBytes(columns(i)),
44          //指定数据
45          Bytes.toBytes(values(i))
46        )
47      }
48      table.put(puts)
49      table.close()
50    }
51  }
```

上述代码中,第 5～28 行代码定义了 createtable()方法,用于在 HBase 中创建表。该方法接收 tableName 和 columnFamilys 两个参数,其中参数 tableName 用于指定表名,参数 columnFamilys 用于通过可变参数列表指定表中多个列族的名称。

第 29～50 行代码定义的 putsToHBase()方法用于在向 HBase 的指定表插入数据,该方法接收 tableName、rowkey、cf、columns 和 values 五个参数。其中,参数 tableName 用于指定表名,参数 rowkey 用于指定行键,参数 cf 用于指定列族的名称,参数 columns 用于通过数组指定表中多个列标识的名称;参数 values 用于通过数组指定表中多个列的数据。

(3) 在单例对象 CategoryTop10 中定义一个 top10ToHBase()方法,该方法用于向 HBase 的表 top10 中插入热门品类 Top10 的分析结果,具体代码如下。

```
1   def top10ToHBase(top10Category: Array[(CategorySortKey, String)]): Unit ={
2       //在 HBase 中创建表 top10 并向表中添加列族 top10_category
3       HBaseUtils.createtable("top10", "top10_category")
4       //创建数组 column,用于指定列标识的名称
5       val column =Array(
6         "category_id", "viewcount",
7         "cartcount", "purchasecount"
8       )
9       var viewcount =""
10      var cartcount =""
11      var purchasecount =""
12      var count =0
13      for ((top10, category_id) <-top10Category) {
```

```
14        count +=1
15        //获取当前品类中商品被查看的次数
16        viewcount =top10.viewCount.toString
17        //获取当前品类中商品被加入购物车的次数
18        cartcount =top10.cartCount.toString
19        //获取当前品类中商品被购买的次数
20        purchasecount =top10.purchaseCount.toString
21        //创建数组 value,用于指定插入的数据
22        val value =Array(category_id, viewcount, cartcount, purchasecount)
23        HBaseUtils.putsToHBase(
24          "top10",
25          s"rowkey_top$count",
26          "top10_category",
27          column,
28          value
29        )
30      }
31    }
```

上述代码中,第 23～29 行代码用于向 HBase 中表 top10 的列 top10_category:category_id、top10_category:viewcount、top10_category:cartcount 和 top10_category:purchasecount 插入数据,数据的内容依次为不同品类的唯一标识、不同品类中商品被查看的次数、不同品类中商品被加入购物车的次数和不同品类中商品被购买的次数。

(4) 在单例对象 CategoryTop10 的 main()方法中调用 top10ToHBase()方法并将 top10Category 作为参数传递,实现将热门品类 Top10 的分析结果插入 HBase 的表 top10 中,具体代码如下。

```
1   try {
2     CategoryTop10.top10ToHBase(top10Category)
3   } catch {
4     case e: Exception =>
5         e.printStackTrace()
6   }
```

(5) 在单例对象 CategoryTop10 的 main()方法中添加关闭 HBase 连接和 Spark 连接的代码,具体代码如下。

```
1   HBaseConnect.closeConnection()
2   sc.stop()
```

3.4 运行 Spark 程序

为了充分利用集群资源分析热门品类 Top10,本项目使用 spark-submit 命令将 Spark 程序提交到 YARN 集群运行,具体操作步骤如下。

1. 封装 jar 文件

在 IntelliJ IDEA 主界面的右侧单击 Maven 选项卡标签展开 Maven 面板。在 Maven 面板双击 Lifecycle 折叠项，如图 3-14 所示。

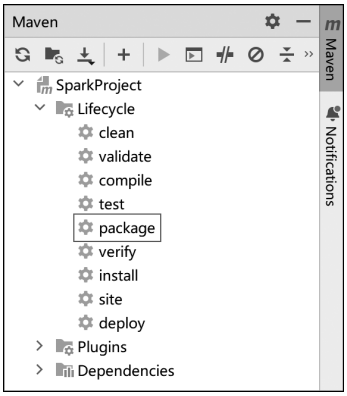

图 3-14　Maven 面板

在图 3-14 所示界面中，双击 package 选项将项目 SparkProject 封装为 jar 文件。项目 SparkProject 封装完成的效果如图 3-15 所示。

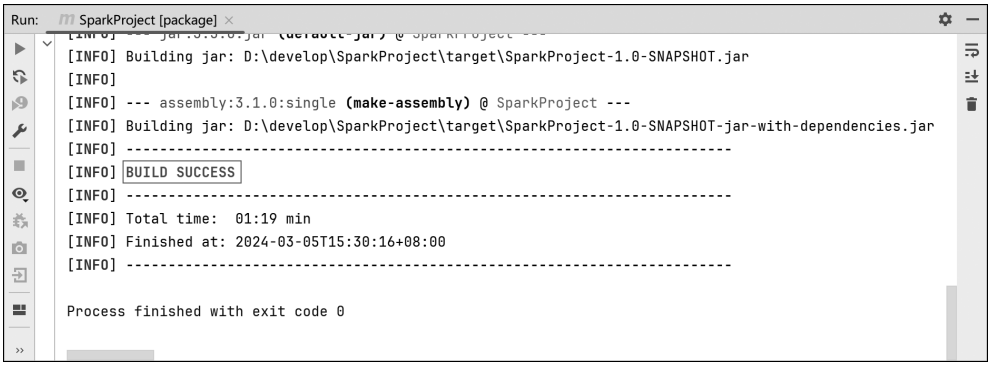

图 3-15　项目 SparkProject 封装完成的效果

从图 3-15 中可以看出，控制台输出 BUILD SUCCESS 提示信息，说明成功将项目 SparkProject 封装为 jar 文件 SparkProject-1.0-SNAPSHOT-jar-with-dependencies.jar 和 SparkProject-1.0-SNAPSHOT.jar。前者不仅包含项目 SparkProject 的源代码和编译后的 Scala 文件，还包含项目 SparkProject 的所有依赖；而后者仅包含项目 SparkProject 的源代码和编译后的 Scala 文件，不含项目 SparkProject 的依赖。这两个 jar 文件存储在 D:\develop\SparkProject\target 目录中。

本项目主要基于 SparkProject-1.0-SNAPSHOT-jar-with-dependencies.jar 来运行 Spark 程序。为了后续使用，这里将 SparkProject-1.0-SNAPSHOT-jar-with-dependencies.jar 重命名为 SparkProject.jar。

2. 启动大数据集群环境

在虚拟机 Spark01、Spark02 和 Spark03 依次启动 ZooKeeper 集群、Hadoop 集群和