# 第3章

# Spark SQL 离线数据处理

## 学习目标

### 知识目标

- 理解 DataFrame 基本原理和常用创建方法（集合/CSV 文件等）
- 掌握 DataFrame 的查看，以及数据查询和处理方法
- 掌握 DataFrame 视图表的 SQL 查询
- 了解 Spark 数据类型转换
- 了解 RDD、DataFrame、Dataset 之间的异同和相互转换关系

### 能力目标

- 会使用 Spark SQL 解决人口信息统计问题
- 会使用 Spark SQL 分析电影评分数据
- 会使用 Spark SQL 处理基本的数据分析问题

### 素质目标

- 培养良好的学习态度和学习习惯
- 培养良好的人际沟通和团队协作能力
- 树立正确的科学精神和培养创新意识

## 3.1 引言

Spark SQL 是用来处理“结构化数据”的功能组件，提供了一种访问多数据源的通用方法，支持 CSV、JSON、Parquet、Hive、JDBC 等数据类型。所谓结构化数据，可理解为以“固定格式”存在的类似数据库表的数据，其行和列清晰地定义了数据的属性，也被称为模式（schema）数据。Spark SQL 设计的 DataFrame 支持直接执行 SQL 语句，充当分布式 SQL 查询引擎，实现对结构化数据的便捷处理。Spark SQL 还支持无缝地将标准 SQL 语句与 Spark 程序混合编写的做法，允许将结构化数据变成底层的 RDD 分布式数据集，这种紧密的集成有助于轻松地运行 SQL 查询，以及执行复杂的分析算法。

Spark SQL 的技术特点主要包括：第一，引入 DataFrame/Dataset 数据类型，其相当于包含字段结构信息的 RDD，可以像传统数据库表一样定义数据的结构；第二，在应用程序中可以混合使用不同来源的数据，比如将来自 JSON 的数据和来自 MySQL 的数据进行连接或合并；第三，内置了查询优化框架，把 SQL 语句解析成 RDD 的逻辑执行计划，在底层实现高效的计算。

## 3.2 DataFrame 基本原理

通过第 2 章的学习可知，RDD 是 Spark 的一个核心内容，也是整个 Spark 框架的基石。RDD 为 Spark 实现分布式计算提供了一个优秀的抽象数据模型，但也存在一些局限性，比如没有内置结构化数据的优化引擎，典型实例就是从文件读取数据后，每个 RDD 的数据元素只是一个普通的字符串，对数据本身的字段结构信息一无所知，也不清楚数据的原本类型。相比之下，Spark SQL 采用了 DataFrame 数据类型，它包含数据的结构细节，通过 Spark SQL 提供的标准接口就能获得有关 RDD 的数据结构和执行计算等信息，从而使得 Spark 可以对 DataFrame 的数据源以及作用于 DataFrame 的操作进行优化，达到提升计算效率的目的。Spark SQL 具备处理大规模结构化数据的能力，同时在使用上比 RDD 更简单。

从本质上说，DataFrame 仍然是一种以 RDD 为基础的分布式数据集，所以也被称为 SchemaRDD，即带有字段结构信息的 RDD。DataFrame 在使用上类似传统数据库的二维表，支持从多种数据源创建数据，包括结构化文件、外部数据库、Hive 表等。所以，我们可以简单地将 DataFrame 理解为一张数据表，即

DataFrame（数据表）= schema（字段结构）+ Data（数据行 RDD）

RDD 是一种分布式的对象集合，对象内部的数据结构是未知的，需要经过反序列化处理后才能知道存储的是什么内容，这就导致 RDD 的数据在转换时效率较低。DataFrame 的每一列都附带字段的名称和类型，还提供了比 RDD 更丰富的功能操作，不仅可以实现 RDD 的绝大多数功能，处理数据也变得更加简单。在使用 DataFrame 功能方法或 SQL 语句处理数据时，Spark 优化器会自动对其进行优化，即使编写的代码或 SQL 语句不够理想，Spark 应用程序也能高效地执行。

除此之外，Spark 还设计了一种 Dataset 数据类型，它不仅与 DataFrame 一样包含数据和字段的结构信息，还结合了 RDD 和 DataFrame 的优点，既支持很多类似 RDD 的功能方法，又支持 DataFrame 的所有操作，使用起来很像 RDD，执行效率和空间资源利用率也更高，可用来方便地处理结构化和非结构化数据。Dataset 可以存放任意类型的对象，当 Dataset 存储的数据是 Row 对象（行）时，此时它就相当于 DataFrame。所以，DataFrame 实际上是 Dataset 的一个特例。RDD、DataFrame、Dataset 三者的关系如图 3-1 所示。

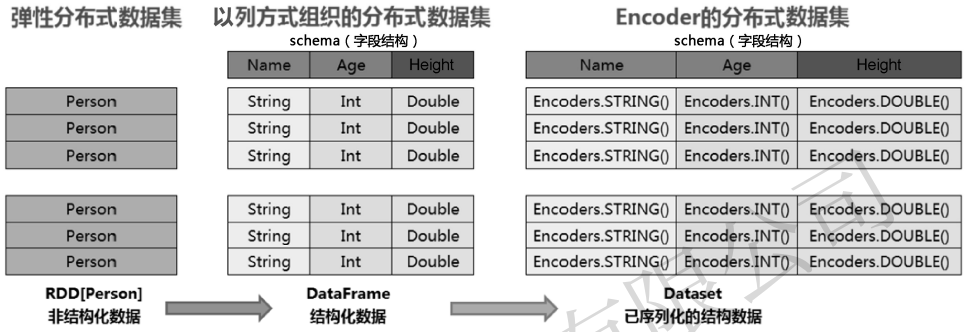


图 3-1　RDD、DataFrame、Dataset 三者的关系

不过，Dataset 需要通过构造 JVM 对象才能使用，所以它目前仅支持 Scala 和 Java 这两种原生的编程语言。自 Spark 2.2 开始，DataFrame 和 Dataset 的 API 已基本统一，两者的使用也相差无几，都建立在 RDD 基础之上，同时能与 RDD 进行相互的转换。考虑到 DataFrame 与数据库表相似，所以之后也经常把 DataFrame 称为“数据表”或“DataFrame 数据集”。

## 3.3　Spark SQL 常用操作

### 3.3.1　DataFrame 的基本创建

#### 1. 使用集合创建 DataFrame

在编写代码时，经常会使用 Scala 集合来创建 DataFrame 进行功能测试，只需要调用 createDataFrame()方法就可以得到一个 DataFrame 对象。下面以实例来演示如何创建 DataFrame，其中用到了 Spark 的自动类型推断机制来确定各字段的数据类型。

打开一个 Linux 终端窗体，在其中执行 spark-shell 命令以启动 SparkShell 交互式编程环境。

```
val  data = List(
       (11, "LingLing", 19, "Hangzhou"),
       (22, "MeiMei", 22, "Shanghai"),
       (33, "Sansan", 23, "Nanjing") )
val  df = spark.createDataFrame(data).
       toDF("id","name","age","address")
df.getClass
df.count
df.show
df.printSchema
```

◇ 构造元组集合，每个元组代表一个人的信息，相当于数据表的一行，即一条记录

◇ 创建 DataFrame，并设定字段结构，Spark 默认自动推断出各字段的数据类型
◇ getClass()方法用于显示对象变量的类型
◇ 获取 df 对象包含的数据行数
◇ 显示数据内容，默认最多显示 20 行
◇ 输出 df 对象的字段结构信息

```
scala>
scala> val  data = List(
     |        (11, "LingLing", 19, "Hangzhou"),
     |        (22, "MeiMei", 22, "Shanghai"),
     |        (33, "Sansan", 23, "Nanjing") )
ddata: List[(Int, String, Int, String)] = List((11,LingLing,19,Hangzhou), (22,MeiMei,22,Shanghai), (33,Sansan,23,Nanjing))
scala> val  df = spark.createDataFrame(data).toDF("id","name","age","address")
df: org.apache.spark.sql.DataFrame = [id: int, name: string ... 2 more fields]
scala> df.getClass
res3: Class[_ <: org.apache.spark.sql.DataFrame] = class org.apache.spark.sql.Dataset
scala> df.count
res4: Long = 3
scala> df.show
+---+--------+---+--------+
| id|    name|age| address|
+---+--------+---+--------+
| 11|LingLing| 19|Hangzhou|
| 22|  MeiMei| 22|Shanghai|
| 33|  Sansan| 23| Nanjing|
+---+--------+---+--------+
scala> df.printSchema
root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = false)
 |-- address: string (nullable = true)
scala>
```

注意：最左侧的竖杠符号|，是在连续输入多行代码时自动出现的提示符，无须手动输入，下同

含义：字段名、数据类型、字段是否允许为空



在该实例中，data 列表包含 3 个人的信息（元组数据），每个元素有 4 个字段，通过调用 createDataFrame()方法创建一个 org.apache.spark.sql.DataFrame 类型的对象，count()和 show()方法分别用于输出其中的数据行数和数据内容，printSchema()方法用于将 DataFrame 对象的字段结构信息显示出来，其中的 id 和 age 由 Spark 自动推断为 integer 类型，name 和 address 由 Spark 自动推断为 string 类型，括号中的 nullable=true/false 代表该字段是否允许为空。

需要注意的是，createDataFrame()方法是通过 spark 对象（SparkSession 类型）来调用的，而不是此前在创建 RDD 时使用的 sc 对象（SparkContext 类型）。实际上，SparkSession 已经成为 Spark 2.0 之后应用程序的统一入口点，简化了在各种场合下对 Spark 运行环境的访问，而且 sc 对象也可以通过 SparkSession 得到。例如：

```
rdd1 = sc.parallelize([1,2,3,4])
rdd1 = spark.sparkContext.parallelize([1,2,3,4])
```

在这里容易发现，sc 变量与 spark.sparkContext 实际上是等价的。

【学习提示】

在编写 Spark 应用程序时，尽量统一使用 SparkSession 来操作 Spark 的各种功能，除非 Spark 早期版本的程序代码使用 SparkContext。

除了使用 createDataFrame()方法的自动类型推断机制确定字段的数据类型，我们还可以通过一个 Scala 的样例类来设定元素的数据类型。

```
case  class  Person(id: Long, name: String, age: Int, address: String)
```

◇ 这里定义了一个 Person 样例类（case  class），包含 id、name、age、address 这 4 个成员变量。Scala 的样例类专为不可变数据封装设计，便于简洁地定义一个保存数据的类

```
val data = List(
       Person(11, "LingLing", 19, "Hangzhou"),
       Person(22, "MeiMei", 22, "Shanghai"),
       Person(33, "Sansan", 23, "Nanjing") )
val df = spark.createDataFrame(data)
df.show
df.printSchema
```

◇ 构造一个 Person 类型元素的列表

◇ 通过 List 创建 DataFrame

◇ 显示 df 对象的数据内容

◇ 输出字段结构信息

```
scala>
scala> case  class  Person(id: Long, name: String, age: Int, address: String)
defined class Person
scala> val  data = List(
    |        Person(11, "LingLing", 19, "Hangzhou"),
    |        Person(22, "MeiMei", 22, "Shanghai"),
    |        Person(33, "Sansan", 23, "Nanjing") )
data: List[Person] = List(Person(11,LingLing,19,Hangzhou), Person(22,MeiMei,
22,Shanghai), Person(33,Sansan,23,Nanjing))
scala> val df = spark.createDataFrame(data)
df: org.apache.spark.sql.DataFrame = [id: bigint, name: string ... 2 more
fields]
scala> df.show
+---+--------+---+--------+
| id|    name|age| address|
+---+--------+---+--------+
| 11|LingLing| 19|Hangzhou|
| 22|  MeiMei| 22|Shanghai|
| 33|  Sansan| 23| Nanjing|
+---+--------+---+--------+
scala> df.printSchema
root
 |-- id: long (nullable = false)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = false)
 |-- address: string (nullable = true)
scala>
```

构造 Person 样例类对象列表，可省略 new 关键字

这里定义的 Person 样例类充当了 DataFrame 的字段结构信息，由此可以看出，Spark 可以直接通过带有类型信息的数据集合构造 DataFrame。

此外，DataFrame 的字段结构信息还能够通过 StructType 来设定，下面是一个具体实例。

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

val  data = List(
       Row(11, "LingLing", 19, "Hangzhou"),
       Row(22, "MeiMei", 22, "Shanghai"),
       Row(33, "Sansan", 23, "Nanjing") )
val  myrdd = sc.parallelize(data)

val  myschema = (new StructType).
       add("id", "int", true).
       add("name", "string", true).
       add("age", "int", true).
       add("address", "string", true)

val  df = spark.createDataFrame(myrdd, myschema)
df.show
df.printSchema
```

◇ 导入 Row 类，以及 org.apache.spark.sql.types 包中的所有类

◇ 定义一个 Row 类型元素的列表，并将其转换为 RDD，为创建 DataFrame 做准备

◇ 设定数据的字段结构信息，包含字段名、数据类型、字段是否允许为空
◇ StructType 代表结构类型，int 表示整型，string 表示字符串，true 表示允许为空

◇ 通过 myrdd 和 myschema 创建 DataFrame

```
scala>
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._
scala> val  data = List(
     |        Row(11, "LingLing", 19, "Hangzhou"),
     |        Row(22, "MeiMei", 22, "Shanghai"),
     |        Row(33, "Sansan", 23, "Nanjing") )
data: List[org.apache.spark.sql.Row] = List([11,LingLing,19,Hangzhou], [22,MeiMei,22,Shanghai], [33,Sansan,23,Nanjing])
scala> val  myrdd = sc.parallelize(data)
myrdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = ParallelCollectionRDD[0] at parallelize at <console>:28
scala> val  myschema = (new StructType).
     |       add("id", "int", true).
     |       add("name", "string", true).
     |       add("age", "int", true).
     |       add("address", "string", true)
myschema: org.apache.spark.sql.types.StructType = StructType(StructField(id,IntegerType,true),StructField(name,StringType,true),StructField(age,IntegerType,true),StructField(address,StringType,true))
```

Row 对象，代表一行数据

通过“.”符号构成链式操作形式，其中的 add()方法用来添加字段结构信息

```
scala> val  df = spark.createDataFrame(myrdd, myschema)
df: org.apache.spark.sql.DataFrame = [id: int, name: string ... 2 more fields]
scala> df.show
+---+--------+---+--------+
| id|    name|age| address|
+---+--------+---+--------+
| 11|LingLing| 19|Hangzhou|
| 22|  MeiMei| 22|Shanghai|
| 33|  Sansan| 23| Nanjing|
+---+--------+---+--------+
scala> df.printSchema
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- address: string (nullable = true)
scala>
```

在上述代码中，DataFrame 字段结构信息是通过 StructType 来设定的，StructType 的 add() 方法的参数就是具体的字段结构信息（StructField），分别代表字段名、字段类型，以及字段是否允许为空。这里 myschema 变量的赋值，还可以表示成下面的形式。

```
val  myschema = StructType(
    StructField("id", IntegerType, true) ::
    StructField("name", StringType, true) ::
    StructField("age", IntegerType, true) ::
    StructField("address", StringType, true) :: Nil
)
```

或者：

```
val  myschema = StructType(
  List(
     StructField("id", IntegerType, nullable = true),
     StructField("name", StringType, nullable = true),
     StructField("age", IntegerType, nullable = true),
     StructField("address", StringType, nullable = true)
  )
)
```

**【学习提示】**

Scala 的 List 实际上是一种链表结构，里面的每个元素类似火车的车厢前后连接在一起，代码中的 Nil 可理解为一个空列表，::是 List 的一个方法，功能是在列表开头添加元素。

**【随堂练习】**

通过 3 种不同的途径，创建一个如下所示的 DataFrame，并将数据显示出来。

```
+-------+----+---+-----+
|user_id|name|age|score|
+-------+----+---+-----+
|     a1|秀儿| 12| 56.5|
|     a2|小丁| 15| 23.0|
|     a3|小梅| 23| 84.0|
|     a4|小筱|  9| 93.5|
+-------+----+---+-----+
```

## Scala【学习 Scala——类和对象】

（1）类是创建对象的蓝图，对象是类的具体实例。类是抽象的，对象是具体的，每个对象都有自己的独立内存空间。类相当于一个用来定义“对象中的方法和变量”的软件模板。

（2）Scala 的类定义实例如下。

```
class  Point(x0: Int, y0: Int) {
   var  x: Int = x0
   var  y: Int = y0

   def  move(dx: Int, dy: Int) {    // 将点移到一个新的位置
      x = x + dx
      y = y + dy
   }
}
```

上述代码定义了一个名为 Point 的类，其中有两个成员变量 x 和 y，以及一个无返回值的方法 move()，与类名定义在一起的叫作主构造方法。Scala 的类定义可以有参数，其被称为类参数（比如这里的 x0 和 y0），类参数在整个类中都可以使用。

接下来，我们可以使用 new 来实例化类以创建对象，并调用对象中的方法，还可以访问对象中的变量。

```
val  pt = new  Point(10, 20);
pt.move(50, 50);
```

（3）Scala 的类继承与 Java 的相似，都是使用 extends 关键字。例如：

```
class  Location(override  val  x0: Int,
                override  val  y0: Int,
                val  z0: Int) extends Point(x0, y0) {
   var  z: Int = z0

   def  move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
   }
}
```

这个 Location 类继承了 Point 类，后者称为父类/基类，前者称为子类。Location 类继承了

Point 类的两个成员变量（x 和 y），并新增了一个成员变量 z。Scala 的子类会继承父类的所有属性和方法，不过子类只允许继承一个父类，这一点与 Java 相似。

### 2. 使用 CSV 文件创建 DataFrame

Spark SQL 读取 CSV 文件数据并将其转换为 DataFrame 的做法非常简单，这是因为 CSV 本质上就是一个结构化的文本文件，文件的每行代表一条完整的数据记录，数据的字段内容用逗号分隔，Spark SQL 不仅能够自动识别这些分隔符，还能自动推断各字段的数据类型。不过，有的 CSV 文件的第 1 行是标题，从第 2 行开始才是数据，而有的 CSV 文件只有数据，并不包含标题，所以在读取时要区别对待。

下面以读取/home/spark 主目录中的 people1.csv 和 people2.csv 文件为例，演示如何从 CSV 文件中读取数据并将其转换为 DataFrame。读者可自行创建这两个文件并将其上传到虚拟机中（或使用 vi 编辑器在虚拟机的主目录中直接创建）。people1.csv 和 people2.csv 文件的内容如下。

people1.csv

```
1  id,name,age,address
2  11,LingLing,19,Hangzhou
3  22,MeiMei,22,Shanghai
4  33,Sansan,23,Nanjing
```

people2.csv

```
1  11,LingLing,19,Hangzhou
2  22,MeiMei,22,Shanghai
3  33,Sansan,23,Nanjing
```

在 SparkShell 交互式编程环境中输入下面的代码。

```
val  df1 = spark.read.
       option("header", "true").
       option("inferSchema", "true").
       csv("file:///home/spark/people1.csv")
val  df2 = spark.read.
       csv("file:///home/spark/people2.csv")
df1.printSchema
df2.printSchema
df1.show
df2.show
```

◇ 从 people1.csv 文件中读取数据，header 参数代表标题行，inferSchema 参数代表是否自动推断字段的数据类型
◇ 从 people2.csv 文件中直接读取数据，默认不包含标题行
◇ 分别输出 df1 和 df2 对象的字段结构信息及其包含的数据内容

```
scala> val  df1 = spark.read.
    |         option("header", "true").
    |         option("inferSchema", "true").
    |         csv("file:///home/spark/people1.csv")
df1: org.apache.spark.sql.DataFrame = [id: int, name: string ... 2 more fields]
scala> val  df2 = spark.read.
    |         csv("file:///home/spark/people2.csv")
df2: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 2 more fields]
scala> df1.printSchema
root
 |-- id: integer (nullable = true)
```

```
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- address: string (nullable = true)
scala> df2.printSchema
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
scala> df1.show
+---+--------+---+--------+
| id|    name|age| address|
+---+--------+---+--------+
| 11|LingLing|  19|Hangzhou|
| 22|  MeiMei| 22|Shanghai|
| 33|  Sansan| 23| Nanjing|
+---+--------+---+--------+
scala> df2.show
+---+--------+---+--------+
|_c0|     _c1|_c2|     _c3|
+---+--------+---+--------+
| 11|LingLing| 19|Hangzhou|
| 22|  MeiMei|  22|Shanghai|
| 33|  Sansan|  23| Nanjing|
+---+--------+---+--------+
scala>
```

不包含标题行的 CSV 文件，字段名会自动设为 _c0、_c1、_c2、_c3



在使用 CSV 文件创建 df1 时，通过 option()方法设置了 header 和 inferSchema 这两个参数，这样 Spark SQL 就会自动识别标题行并推断出各字段的数据类型。对 df2 来说，因为 people2.csv 文件不包含标题行，所以在不预设字段名的前提下，字段名由 Spark SQL 自动设定（如_c0、_c1、_c2 等这样的名称），数据类型默认为 string。在这种情况下，通常需要人为设定字段的结构信息，比如下面的代码实例。

```
val  myschema = "id  long, name  string, age  int, address  string"
val  df2 = spark.read.
       schema(myschema).
       csv("file:///home/spark/people2.csv")
df2.printSchema
df2.show
```

◇ 字段结构信息定义，包含字段名、字段类型，类似数据库的表结构定义
◇ 从 people2.csv 文件中读取数据，字段结构信息设为 myschema

```
scala>
scala> val  myschema = "id  long, name  string, age  int, address  string"
myschema: String = id long, name string, age int, address string
scala> val  df2 = spark.read.
     |         schema(myschema).
```

```
    |         csv("file:///home/spark/people2.csv")
df2: org.apache.spark.sql.DataFrame = [id: bigint, name: string ... 2 more
fields]
scala> df2.printSchema
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- address: string (nullable = true)
scala> df2.show
+---+--------+---+--------+
| id|    name|age| address|
+---+--------+---+--------+
| 11|LingLing| 19|Hangzhou|
| 22|  MeiMei| 22|Shanghai|
| 33|  Sansan| 23| Nanjing|
+---+--------+---+--------+
scala>
```

**【学习提示】**

如果遇到 CSV 文件的标题行的字段名为中文的情况，则可以通过 schema()方法重新设定字段名。此外，在使用 CSV 文件创建 DataFrame 时，还可以借助 option()方法设定更多的参数，包括分隔符、文件编码等。下面的代码演示了这种情况的处理方法。

```
val  cameraDF = spark.read
   .option("encoding", "GBK")          // 设定文件编码为 GBK 字符集，以避免出现乱码
   .option("header", true)                 // 文件包含标题行
   .option("inferSchema", true)            // 自动推断字段的数据类型
   .option("sep", ",")                     // 设定分隔符为逗号（CSV 文件默认设置）
   .schema("sxtid string, sxtxlh string") // 重新设定字段名
   .csv("hdfs:///data/camera_info.csv")    // 读取数据文件（HDFS 文件或本地文件等）
```

**【随堂练习】**

在主目录中创建一个 stus.csv 文件，文件中包含以下数据内容，将其转换为 DataFrame。

stus.csv

```
1  user_id,name,age,score
2  a1,秀儿,12,56.5
3  a2,小丁,15,23.0
4  a3,小梅,23,84.0
5  a4,小筱,9,93.5
```

## 3.3.2　DataFrame 的查看

DataFrame 的查看主要包括查看数据表的字段结构信息、数据内容这两方面的操作，常用

的查看算子及属性如表 3-1 所示。这些算子都是 Action 算子（类似 RDD 的 Action 算子），由它们可以启动 Transformation 算子的执行。

表 3-1　DataFrame 常用的查看算子及属性

| 算子或属性名称 | 功能描述 |
|---|---|
| printSchema | 输出 DataFrame 的字段结构信息 |
| columns（属性） | 返回 DataFrame 的字段名 |
| dtypes（属性） | 返回 DataFrame 的字段名及字段类型 |
| count | 获取 DataFrame 的数据行数 |
| show | 显示 DataFrame 的数据内容，可指定显示行数，以及当字段内容超长时是否截断显示 |
| first、head | 获取 DataFrame 的首行数据内容，返回 Row 类型对象 |
| take、takeAsList | 获取 DataFrame 前 *n* 行数据内容，返回 Row 类型对象数组 |
| collect | 获取 DataFrame 的所有行数据内容，返回 Row 类型对象数组，只适用于数据量小的场合 |
| foreach | 每次对一行数据应用指定的处理函数并返回结果，内存占用较少 |
| foreachPartition | 与 foreach 类似，但它以分区为单位批量处理数据，对分区中的数据应用处理函数 |

下面依次介绍表 3-1 中的部分常用查看算子及属性。先准备后续要用的 DataFrame。

```
val  data = List( (6, "DingDing", 18, 88, "M"),
                  (3, "KeKe",    18, 90, "F"),
                  (2, "FeiFei",  16, 60, null),
                  (4, "JiaJia",  24, 92, "M"),
                  (1, "MeiMei",  20, 95, "F") )
val  df = spark.createDataFrame(data).
       toDF("id", "name", "age", "score", "gender")
```

◇ 定义元组集合，代表 5 个人的信息

◇ 根据 data 和字段结构信息创建 DataFrame

```
scala>
scala> val  data = List((6, "DingDing", 18, 88, "M"),
     |                  (3, "KeKe",    18, 90, "F"),
     |                  (2, "FeiFei",  16, 60, null),
     |                  (4, "JiaJia",  24, 92, "M"),
     |                  (1, "MeiMei",  20, 95, "F"))
data: List[(Int, String, Int, Int, String)] = List((6,DingDing,18,88,M), (3,KeKe,
18,90,F), (2,FeiFei,16,60,null), (4,JiaJia,24,92,M), (1,MeiMei,20,95,F))
scala> val  df = spark.createDataFrame(data).
     |         toDF("id", "name", "age", "score", "gender")
df: org.apache.spark.sql.DataFrame = [id: int, name: string ... 3 more fields]
scala>
```

下面分别演示其中的部分算子，以及 columns、dtypes 属性的用法。

```
df.printSchema()
df.columns
df.dtypes
```

◇ 输出 DataFrame 的字段结构信息，方法无参数也可省略括号

◇ 返回 DataFrame 的字段名，columns 是一个属性

◇ 返回 DataFrame 的字段名及字段类型，dtypes 也是一个属性

```
scala>
scala> df.printSchema()
root
```

```
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = false)
 |-- score: integer (nullable = false)
 |-- gender: string (nullable = true)
scala> df.columns
res1: Array[String] = Array(id, name, age, score, gender)
scala> df.dtypes
res2: Array[(String, String)] = Array((id,IntegerType), (name,StringType), (age,
IntegerType), (score,IntegerType), (gender,StringType))
scala>
```

| | |
|---|---|
| df.count | ◇ 获取 DataFrame 的数据行数 |
| df.show | ◇ 显示 DataFrame 的数据内容，默认显示前 20 行 |
| df.show(2) | ◇ 指定显示 DataFrame 的前两行数据内容 |
| df.show(2, false) | ◇ 显示前两行数据内容，且字段内容长度若超过 20 个字符，则不截断显示（false 表示不截断，true 表示截断） |

```
scala>
scala> df.count
res3: Long = 5
scala> df.show
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  6|DingDing|  18|   88|     M|
|  3|   KeKe|  18|   90|     F|
|  2|  FeiFei| 16|   60|  NULL|
|  4|  JiaJia| 24|   92|     M|
|  1|  MeiMei| 20|   95|     F|
+---+--------+---+-----+------+
scala> df.show(2)
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  6|DingDing|  18|   88|     M|
|  3|   KeKe|  18|   90|     F|
+---+--------+---+-----+------+
only showing top 2 rows
scala> df.show(2, false)
+---+--------+---+-----+------+
|id |name    |age|score|gender|
+---+--------+---+-----+------+
|6  |DingDing|18  |88   |M     |
|3  |KeKe    |18  |90   |F     |
+---+--------+---+-----+------+
```

由于这里的字段内容都很短，所以 false 参数体现不出超长截断的效果

```
only showing top 2 rows
scala>
```

这里的 show()方法可以根据需要使用，包括设定显示的数据行数，以及当字段内容超长时是否截断显示（避免输出内容出现排版错乱）等。

| | |
|---|---|
| df.first | ◇ 获取 DataFrame 的首行数据内容，返回 Row 类型对象 |
| df.take(2) | ◇ 获取 DataFrame 前 2 行数据内容，返回 Row 类型对象数组 |
| df.collect | ◇ 获取 DataFrame 的所有行数据内容，返回 Row 类型对象数组 |

```
scala>
scala> df.first
res7: org.apache.spark.sql.Row = [6,DingDing,18,88,M]
scala> df.take(2)
res8: Array[org.apache.spark.sql.Row] = Array([6,DingDing,18,88,M], [3,KeKe,18,
90,F])
scala> df.collect
res9: Array[org.apache.spark.sql.Row] = Array([6,DingDing,18,88,M], [3,KeKe,18,
90,F], [2,FeiFei,16,60,null], [4,JiaJia,24,92,M], [1,MeiMei,20,95,F])
scala>
```

上面获取的 DataFrame 数据行返回的都是 Row 类型对象或对象数组，每个 Row 类型对象的属性值就是对应数据行的字段内容。

下面演示 foreach 算子的简单用法，实例代码如下。

```
import  org.apache.spark.sql.Row

def  myprocess(x: Row): Unit = {
   val name = x.getString(1).toUpperCase
   println(name)
}

df.foreach( row => myprocess(row) )
   或简化为：
df.foreach( myprocess(_) )
```

◇ 定义一个函数，用于执行数据行的具体处理工作，这里将 name 字段的内容转换成大写字母形式

◇ Row 中的字段顺序从 0 开始，1 代表 name 字段，也可通过字段名获取字段值

◇ 自定义函数中的 x.getString(1)等效于 x.getAs[String]("name")

◇ 通过 foreach()方法对数据行应用处理函数。myprocess(_)还可再简化为：myprocess _

```
scala>
scala> import  org.apache.spark.sql.Row
import org.apache.spark.sql.Row
scala> def  myprocess(x: Row): Unit = {
     |    val name = x.getString(1).toUpperCase
     |    println(name)
     | }
myprocess: (x: org.apache.spark.sql.Row)Unit
scala> df.foreach( row => myprocess(row) )
DINGDING
KEKE
FEIFEI
```

调用字符串的 toUpperCase()方法将 name 字段的内容转换为大写字母形式

```
JIAJIA
MEIMEI
scala>
```

调用 foreach()方法的目的是对每行数据进行处理，具体处理过程定义在 myprocess()函数中。

**【学习提示】**

与 RDD 类似，如果是将 DataFrame 的数据写入数据库之类的操作，此时就应该考虑使用 foreachPartition 算子，这是因为它是以分区为单位批量处理数据的。如果使用 foreach 算子，则在处理每条数据时都要创建一个数据库连接，用完再断开，这样反反复复地创建和关闭数据库连接会产生极大的消耗。

**【随堂练习】**

分别将下面 DataFrame 的字段结构信息、字段名、数据行数和第 1 条数据内容输出，并使用 foreach()方法把每名学生的分数按四舍五入的方式显示出来。

```
+-------+----+---+-----+
|user_id|name|age|score|
+-------+----+---+-----+
|     a1|秀儿| 12| 56.5|
|     a2|小丁| 15| 23.0|
|     a3|小梅| 23| 84.0|
|     a3|小筱|  9| 93.5|
+-------+----+---+-----+
```

### 3.3.3　DataFrame 的数据操作（DSL）

由于 DataFrame 是建立在 RDD 基础上的，因此 RDD 的很多转换操作和行动操作在 DataFrame 中也是支持的，包括转换操作、行动操作这两大类 API，它们被统称为 DSL（Domain Specific Language，特定领域语言），实际就是类似 RDD 的算子，代码上则表现为一些方法/函数。判断是 DataFrame 的转换操作还是行动操作，可以根据返回的结果是 Spark SQL 数据类型（如 DataFrame 等）还是普通的 Scala 数据类型来确定。同样地，DataFrame 的转换操作也是延迟执行的，只有在遇到行动算子时才会启动计算过程。

为方便起见，本节大部分实例代码将统一使用下面的 DataFrame，对应的变量名为 df 对象（取自 DataFrame 一词的两个字母）。

```
val  data = List( (6, "DingDing", 18, 88, "M"),
                  (3, "KeKe",     18, 90, "F"),
                  (2, "FeiFei",   16, 60, null),
                  (4, "JiaJia",   24, 92, "M"),
                  (1, "MeiMei",   20, 95, "F") )
val  df = spark.createDataFrame(data).
       toDF("id","name","age","score","gender")
```

◇ 每次在退出 SparkShell 交互式编程环境后，都需要重新执行这几行代码才能使用 df 对象

◇ 定义元组集合，代表 5 个人的信息
◇ 创建实例数据 DataFrame

```
scala>
scala> val  data = List( (6, "DingDing", 18, 88, "M"),
    |                    (3, "KeKe",   18, 90, "F"),
    |                    (2, "FeiFei", 16, 60, null),
```

```
    |                     (4, "JiaJia", 24, 92, "M"),
    |                     (1, "MeiMei", 20, 95, "F") )
data: List[(Int, String, Int, Int, String)] = List((6,DingDing,18,88,M), (3,KeKe,
18,90,F), (2,FeiFei,16,60,null), (4,JiaJia,24,92,M), (1,MeiMei,20,95,F))
scala> val  df = spark.createDataFrame(data).
    |          toDF("id", "name", "age", "score", "gender")
df: org.apache.spark.sql.DataFrame = [id: int, name: string ... 3 more fields]
scala>
```

【学习提示】

*若在本节后续内容的学习中遇到 df 变量未初始化的问题，则重新执行一遍这里的代码来创建 DataFrame，并对 df 变量进行赋值。*

### 1. DataFrame 的数据查询

DataFrame 的数据查询操作涉及对数据的筛选、排序、分组、查询等，相关的算子如表 3-2 所示，这些操作返回的都是一个新的 DataFrame 数据集。

表 3-2　DataFrame 常用的数据查询算子

| 算子名称 | 功能描述 |
| --- | --- |
| where/filter | 根据指定条件得到符合要求的 DataFrame 数据集，类似 SQL 语句中的 WHERE 子句，条件表达式可以使用 NOT、AND、OR 等 |
| sort/orderBy | 按指定的条件对 DataFrame 数据集进行排序，得到一个新的 DataFrame 数据集，类似 SQL 语句中的 ORDER BY 子句 |
| groupBy | 按指定的条件对 DataFrame 数据集进行分组，得到一个新的 DataFrame 数据集，类似 SQL 语句中的 GROUP BY 子句 |
| select/selectExpr | 对 DataFrame 数据集中的数据行进行查询，得到一个新的 DataFrame 数据集，类似 SQL 语句中的 SELECT 子句。selectExpr 算子支持使用表达式 |

下面通过实例予以说明。

（1）where 和 filter 数据筛选。

where 和 filter 都用于筛选符合条件的数据行，两者是等价的。where()和 filter()方法需要提供一个过滤条件的参数，该参数有两种形式：一种是形如 "age>18" 的字符串；另一种是使用 $"age">=18 这种 Column 对象的表示形式（以“$”符号开头的字段名）。

下面是 where 算子的使用实例，继续在 SparkShell 交互式编程环境中输入下面的代码（注意：若 df 变量未初始化，则应先按照前面数据准备的代码进行赋值，下同）。

```
val  df1 = df.where("age>=18")
val  df2 = df.where("age>=18  and  score>90")
val  df3 = df.where($"age">=18)
val  df4 = df3.where($"score">90)
df1.show
df2.show
df3.show
df4.show
```

◇ 查询 age>=18 的数据行
◇ 查询 age>=18 且 score>90 的数据行
◇ 查询 age>=18 的数据行，$"age"代表 age 字段这一列
◇ 继续查询 df3 中 score>90 的数据行
◇ 如果条件是字符串值，则可以在字符串里面使用单引号。例如：df.filter( "sal>1000 and job=='MANAGER'")

```
scala>
scala> val df1 = df.where("age>=18")
```

```
df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val df2 = df.where("age>=18  and  score>90")
df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val df3 = df.where($"age">=18)
df3: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val df4 = df3.where($"score">90)
df4: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> df1.show
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  6|DingDing|  18|   88|     M|
|  3|   KeKe|   18|   90|     F|
|  4|  JiaJia|  24|   92|     M|
|  1|  MeiMei|  20|   95|     F|
+---+--------+---+-----+------+
scala> df2.show
+---+------+---+-----+------+
| id|  name|age|score|gender|
+---+------+---+-----+------+
|  4|JiaJia| 24|   92|     M|
|  1|MeiMei| 20|   95|     F|
+---+------+---+-----+------+
scala> df3.show
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  6|DingDing| 18|   88|     M|
|  3|   KeKe| 18|   90|     F|
|  4|  JiaJia| 24|   92|     M|
|  1|  MeiMei|  20|   95|     F|
+---+--------+---+-----+------+
scala> df4.show
+---+------+---+-----+------+
| id|  name|age|score|gender|
+---+------+---+-----+------+
|  4|JiaJia| 24|   92|     M|
|  1|MeiMei| 20|   95|     F|
+---+------+---+-----+------+
scala>
```

$"age" 代表 age 字段这一列

在这个实例中，df1 至 df3 都是从原始数据表 df 中通过 where 操作得到的数据行，df4 则是从 df3 中再次筛选得到的数据行。where()和 filter()方法的字符串形式的参数与 SQL 语句的写法是相似的，所以使用 SQL 语句形式的字符串条件还是比较直观的。

（2）sort 和 orderBy 数据排序。

DataFrame 可以通过 sort()或 orderBy()方法对数据进行排序，在调用这两个方法时需要指定排序依据，即“按照什么字段”进行排序，它们在功能上是一样的。

下面是 sort()方法的使用实例，在使用时 sort()和 orderBy()方法可以相互替换。继续在 SparkShell 交互式编程环境中输入下面的代码。

```
val  df1 = df.sort("age")                          ◇ 按年龄排序，默认按升序排列
val  df2 = df.sort($"age")                         ◇ 按年龄排序，默认按升序排列
val  df3 = df.sort("age", "score")                 ◇ 按年龄和分数排序，默认均按升序排列
val  df4 = df.sort($"age".desc)                    ◇ 按年龄降序排列
val  df5 = df.sort($"age".asc, $"score".desc)      ◇ 按年龄升序、分数降序的方式排列
df1.show
df5.show
```

```
scala>
scala> val  df1 = df.sort("age")
df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val  df2 = df.sort($"age")
df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val  df3 = df.sort("age", "score")
df3: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val  df4 = df.sort($"age".desc)
df4: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> val  df5 = df.sort($"age".asc, $"score".desc)
df5: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name:
string ... 3 more fields]
scala> df1.show
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  2|  FeiFei| 16|   60|  NULL|
|  6|DingDing| 18|   88|     M|
|  3|    KeKe| 18|   90|     F|
|  1|  MeiMei| 20|   95|     F|
|  4|  JiaJia| 24|   92|     M|
+---+--------+---+-----+------+
scala> df5.show
```

$"age"是一个 Column 对象，调用的是 desc()方法

```
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  2|  FeiFei| 16|   60|  NULL|
|  3|    KeKe| 18|   90|     F|
|  6|DingDing| 18|   88|     M|
|  1|  MeiMei| 20|   95|     F|
|  4|  JiaJia| 24|   92|     M|
+---+--------+---+-----+------+
scala>
```

在上述代码中，指定的字段默认都是以升序的方式排列的。如果要指定排序方式，则可以通过“$”指定字段名对应的列（Column 对象），后面再跟 asc()或 desc()方法，如$"score".desc()或$"score".desc 等。

（3）groupBy 数据分组。

顾名思义，groupBy 用于对 DataFrame 的数据行依据某种规则进行分组，返回的是 GroupedData 类型的对象。groupBy()方法通常要与聚合函数一起使用，可以是 Spark 内置的聚合函数或自定义函数。常用的简单聚合函数如下。

- count()：统计数据行的数量。
- mean()、avg()：求字段的平均值。
- max()、min()：求字段的最大值和最小值。
- sum()：求字段的累加和。

下面直接通过实例代码予以说明。继续在 SparkShell 交互式编程环境中输入下面的代码。

```
val  df1 = df.groupBy("gender").count()
val  df2 = df.groupBy("gender").agg(
              Map(
                 "age" -> "mean",
                 "score" -> "max"
              )
           )
df1.show
df2.show
```

◇ 按性别统计人数
◇ 按性别聚合分组，分别统计平均年龄和最高分
◇ Map 代表一个字典数据集合，其中包含的是键值对类型的数据

```
scala>
scala> val  df1 = df.groupBy("gender").count()
df1: org.apache.spark.sql.DataFrame = [gender: string, count: bigint]
scala> val  df2 = df.groupBy("gender").agg(
     |               Map(
     |                  "age" -> "mean",
     |                  "score" -> "max"
     |               )
     |            )
df2: org.apache.spark.sql.DataFrame = [gender: string, avg(age): double ... 1
more field]
```

设定要聚合的字段及操作名称，mean 用于求平均值，max 用于求最大值

```
scala> df1.show
+------+-----+
|gender|count|
+------+-----+
|     F|    2|
|     M|    2|
|  NULL|    1|
+------+-----+
scala> df2.show
+------+--------+----------+
|gender|avg(age)|max(score)|
+------+--------+----------+
|     F|    19.0|        95|
|     M|    21.0|        92|
|  NULL|    16.0|        60|
+------+--------+----------+
scala>
```

（4）select 和 selectExpr 数据查询。

select 和 selectExpr 算子用于从 DataFrame 数据集中查询指定的字段，并返回一个新的 DataFrame 数据集。另外，selectExpr 算子还支持对指定字段进行处理，如取绝对值、四舍五入等。也就是说，它可以与表达式一起使用。

select()和 selectExpr()方法的实例代码如下。

```
df.select("*").show                          ◇ 查询数据集的所有字段
df.select("name", "age").show                ◇ 查询数据集的 name、age 字段
df.selectExpr("age*2").show                  ◇ 查询数据集的 age 字段，并将字段值乘以 2
df.selectExpr("age*2  as  newage").show      ◇ 将查询的 age*2 字段的别名设置为 newage
scala>
scala> df.select("*").show
+---+--------+---+-----+------+
| id|    name|age|score|gender|
+---+--------+---+-----+------+
|  6|DingDing| 18|   88|     M|
|  3|   KeKe| 18|    90|     F|
|  2|  FeiFei| 16|   60|  NULL|
|  4|  JiaJia| 24|   92|     M|
|  1|  MeiMei| 20|   95|     F|
+---+--------+---+-----+------+
scala> df.select("name", "age").show
+--------+---+
|    name|age|
+--------+---+
|DingDing| 18|
```

select()和 selectExpr()方法在涉及多个字段时，每个字段都应单独写成方法的一个参数

```
|    KeKe| 18|
|  FeiFei| 16|
|  JiaJia| 24|
|  MeiMei| 20|
+--------+---+
scala> df.selectExpr("age*2").show
+---------+
|(age * 2)|
+---------+
|       36|
|       36|
|       32|
|       48|
|       40|
+---------+
scala> df.selectExpr("age*2  as  newage").show
+------+
|newage|
+------+
|    36|
|    36|
|    32|
|    48|
|    40|
+------+
scala>
```

代码的最后两行调用了 selectExpr()方法，其参数是 SQL 形式的字段，如果有多个字段，则每个字段都是 selectExpr()方法的字符串参数，类似 df.select("name", "age")的做法。此外，Select()和 selectExpr()方法还可以结合 DataFrame 的其他操作方法一起使用，比如先执行 where()方法再执行 select()方法。

```
val  df1 = df.where("age>18").select("name")
df1.show()
```

◇ 先过滤 age>18 的数据行，在此基础上再查询其 name 字段

```
scala>
scala> val  df1 = df.where("age>18").select("name")
df1: org.apache.spark.sql.DataFrame = [name: string]
scala> df1.show()
+------+
|  name|
+------+
|JiaJia|
|MeiMei|
+------+
scala>
```

综上所述，DataFrame 中很多操作方法的功能与传统的 SQL 语句是相似的。

**【随堂练习】**

下面是一个包含几名学生信息的 DataFrame 数据集。

```
val  df_stu = spark.createDataFrame(
                List(
                   ("a1", "秀儿", 12, 56.5), ("a2", "小丁", 15, 23.0),
                   ("a3", "小梅", 23, 84.0), ("a4", "小筱", 9,  93.5)
                )
             ).toDF("user_id", "name", "age", "score")
df_stu.show
```

```
>>> df_stu.show
+-------+----+---+-----+
|user_id|name|age|score|
+-------+----+---+-----+
|     a1|秀儿| 12| 56.5|
|     a2|小丁| 15| 23.0|
|     a3|小梅| 23| 84.0|
|     a4|小筱|  9| 93.5|
+-------+----+---+-----+
```

（1）筛选出 age<15 的数据行。

（2）按分数排序输出所有数据行。

（3）求出最高分和平均年龄。

### 2. DataFrame 的数据处理

DataFrame 提供了一系列用于数据处理的功能操作，包括去除重复行、删除字段、添加列、连接数据行等，相关的算子如表 3-3 所示。不过要提醒的是，类似 RDD 中的做法，DataFrame 的数据处理操作并不会改变它自身的数据内容，而是生成一个新的 DataFrame。此外，由于 Spark 是基于分布式的手段对数据进行处理的，因此 DataFrame 不能简单地直接操作某列数据。

表 3-3　DataFrame 常用的数据处理算子

| 算子名称 | 功能描述 |
| --- | --- |
| distinct | 去除完全重复的数据行 |
| dropDuplicates | 根据指定的字段去除重复的数据行 |
| dropna | 去除某些字段值为 null 的数据行 |
| drop | 删除数据集的某些字段，保留其他字段 |
| limit | 限定返回的结果数据集的数据行数 |
| intersect、intersectAll | 按数据行的字段顺序返回两个数据集的交集，等同于 SQL 语句的 INTERSECT。intersectAll 还会保留重复记录 |
| union/unionAll、unionByName | 返回两个数据集的并集，均保留重复记录。union/unionAll 等同于 SQL 语句的 UNION ALL，按数据行的字段位置顺序合并。unionByName 则按数据行的字段名合并 |
| exceptAll | 返回两个数据集的差集 |
| join | 将两个数据集连接起来，分为内连接、左外连接、右外连接、全外连接等方式 |
| withColumn | 添加列，或替换现有的同名列 |
| withColumnRenamed | 修改数据集的字段名 |
| col、apply | 获取指定列，返回 Column 对象，通常用于参与其他计算 |

下面介绍表 3-3 中部分算子的使用方法。

（1）distinct 和 dropDuplicates 去重。

distinct()和 dropDuplicates()方法都用来删除重复的数据行，但两者有一点区别。下面是 DataFrame 的两种数据行去重的操作实例。

```
val  df1 = spark.createDataFrame( List(

("Alice",5,Some(80)),("Alice",5,Some(80)),
       ("Alice",9,Some(80)),("Tom",12,None) )
     ).toDF("name","age","height")
df1.show
df1.distinct.show
df1.dropDuplicates("name","height").show
```

◇ 构造一个 DataFrame
◇ Some 代表有值，None 代表无值（与 null 不等同）
◇ distinct 算子用于去除完全重复的数据行，生成新的 DataFrame
◇ dropDuplicates 算子用于去除 name 和 height 字段相同的行

```
scala>
scala> val  df1 = spark.createDataFrame( List(
    |        ("Alice",5,Some(80)),("Alice",5,Some(80)),
    |        ("Alice",9,Some(80)),("Tom",12,None) )
    |      ).toDF("name","age","height")
df1: org.apache.spark.sql.DataFrame = [name: string, age: int ... 1 more field]
scala> df1.show
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  5|    80|
|Alice|  5|    80|
|Alice|  9|    80|
|  Tom| 12|  NULL|
+-----+---+------+
scala> df1.distinct.show
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  5|    80|
|Alice|  9|    80|
|  Tom| 12|  NULL|
+-----+---+------+
scala> df1.dropDuplicates("name", "height").show
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  5|    80|
|  Tom| 12|  NULL|
+-----+---+------+
scala>
```

构造 4 条记录，Some 代表有值（括号里面是具体值），None 代表无值

去除完全重复的行，得到一个新的 DataFrame

去除 name 和 height 这两个字段相同的行

在这个实例中，调用 distinct()和 dropDuplicates()方法去除重复的数据行，前者要求数据行的每个字段都相同才会去除，后者确定指定的字段相同后才去除。

（2）dropna 和 drop 按列删除。

如果 DataFrame 中存在不完整的数据行，比如某些行缺失部分字段值，此时就可以通过 dropna()方法来过滤这种数据行，或者根据需要“剪掉”某些字段，这也是 ETL 数据清洗工作中常见的一种做法。值得一提的是，无论施加怎样的操作，原始的 DataFrame 都不会发生变化。继续沿用上面 df1 数据集的代码，实例如下。

```
df1.na.drop.show                    ◇ 删除包含 null 值的数据行，生成新的 DataFrame
df1.drop("height").show             ◇ 剪掉 height 字段（影响所有记录）
df1.drop("age","height").show       ◇ 剪掉 age 和 height 字段（影响所有记录）
```

```
scala>
scala> df1.na.drop.show
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  5|    80|
|Alice|  5|    80|
|Alice|  9|    80|
+-----+---+------+
scala> df1.drop("height").show
+-----+---+
| name|age|
+-----+---+
|Alice|  5|
|Alice|  5|
|Alice|  9|
|  Tom| 12|
+-----+---+
scala> df1.drop("age","height").show
+-----+
| name|
+-----+
|Alice|
|Alice|
|Alice|
|  Tom|
+-----+
scala>
```

此外，对存在字段值缺失的数据行还可以使用 fill()方法填充缺失的字段值，这里给出一段实例代码。

```
val  df2 = df1.na.fill(50)                  // 空字段全部填为 50
val  df2 = df1.na.fill(false)               // 空字段全部填为 false
```

```
val  df2 = df1.na.fill(
       Map("age"->50, "name"->"unknown"))  // 通过 Map 设定不同空字段的填充值
```

（3）limit 限定数据行数。

limit()方法用来限定 DataFrame 的数据行数，以便进行后续的处理。例如：

| | |
|---|---|
| val  df2 = df1.limit(2)<br>df2.count<br>df2.show | ◇ 限定 DataFrame 的数据行数，生成新的 DataFrame<br>◇ 获取当前 DataFrame 的数据行数 |

```
scala>
scala> val  df2 = df1.limit(2)
df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [name: string,
age: int ... 1 more field]
scala> df2.count
res0: Long = 2
scala> df2.show
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  5|    80|
|Alice|  5|    80|
+-----+---+------+
scala>
```

这里是当前 DataFrame 的数据行数

其中，limit()方法返回一个新的 DataFrame，通过这种方式减少了后续处理的数据量。limit()与 show()方法是不同的，show()方法只可显示指定数量的数据内容，并不影响 DataFrame 的实际数据行数。

（4）withColumn 和 withColumnRenamed 按列处理。

withColumn()方法用来在当前 DataFrame 基础上添加一列，或替换现有的同名列，并返回一个新的 DataFrame。如果在新增或替换时要修改字段的值，则可以对它进行简单的运算或借助函数进行修改。withColumnRenamed()方法则用于修改现有列的字段名，列数据保持不变。继续沿用上面 df2 数据集的代码，实例如下。

```
import  org.apache.spark.sql.functions.{col,upper,lit}
```

| | |
|---|---|
| val  df3 = df2.withColumnRenamed("age", "newage")<br>df3.show | ◇ 修改字段名 age 为 newage<br>◇ 查看修改后的 DataFrame 数据内容 |
| val  df4 = df2.withColumn("newname", upper(col("name")))<br>val  df5 = df2.withColumn("name", upper(col("name")))<br>val  df6 = df2.withColumn("addr", lit("Hangzhou"))<br>df4.show<br>df5.show<br>df6.show<br><br>df2.show | ◇ 新增一列 newname，内容为原 name 列数据的大写字母<br>◇ 替换 name 列，内容为原 name 列数据的大写字母<br>◇ 新增 addr 列，内容为字面值 Hangzhou<br><br>◇ 再次查看 df2，证明未改变过 |

◇ 第一行的 import 语句的目的是导入 col()、upper()、lit()这几个函数，也可写成下面的形式：

```
import  org.apache.spark.sql.functions._
```

这样就可以使用 org.apache.spark.sql.functions 包中的任意函数了

◇ Spark 还支持自动隐式转换处理，比如先执行下面导入隐式转换功能的语句：

```
import  spark.implicits._
```

然后就可以使用$"name"来代替 col("name")这种字段列的表达形式

```
scala>
scala> import  org.apache.spark.sql.functions.{col,upper,lit}
import org.apache.spark.sql.functions.{col, upper, lit}
scala> val  df3 = df2.withColumnRenamed("age", "newage")
df3: org.apache.spark.sql.DataFrame = [name: string, newage: int ... 1 more
field]
scala> df3.show
+-----+------+------+
| name|newage|height|
+-----+------+------+
|Alice|     5|    80|
|Alice|     5|    80|
+-----+------+------+
scala> val  df4 = df2.withColumn("newname", upper(col("name")))
df4: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]
scala> val  df5 = df2.withColumn("name", upper(col("name")))
df5: org.apache.spark.sql.DataFrame = [name: string, age: int ... 1 more field]
scala> val  df6 = df2.withColumn("addr", lit("Hangzhou"))
df6: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]
scala> df4.show
+-----+---+------+-------+
| name|age|height|newname|
+-----+---+------+-------+
|Alice|  5|    80|  ALICE|
|Alice|  5|    80|  ALICE|
+-----+---+------+-------+
scala> df5.show
+-----+---+------+
| name|age|height|
+-----+---+------+
|ALICE|  5|    80|
|ALICE|  5|    80|
+-----+---+------+
scala> df6.show
+-----+---+------+--------+
| name|age|height|    addr|
+-----+---+------+--------+
|Alice|  5|    80|Hangzhou|
```

```
|Alice|  5|    80|Hangzhou|
+-----+---+------+--------+
scala>
scala> df2.show
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  5|    80|
|Alice|  5|    80|
+-----+---+------+
scala>
```

上述代码用到了 col()、lit()、upper()函数，它们都可以用于操作 DataFrame 的字段。其中，col()函数用于获取一列数据并返回 Column 对象，lit()函数用于将常量值转换为字段，upper()函数用于将字段转换为大写字母形式，还有其他一些类似的函数也都是在 org.apache.spark.sql.functions 包中定义的。

（5）intersectAll 交集和 unionByName 并集。

intersectAll()方法用于获取两个 DataFrame 的交集，得到的是两者按字段顺序排列的相同数据行，即只看字段位置而不看字段名，且交集可能存在重复数据行。unionByName()方法用于获取两个 DataFrame 的并集，得到的是将两者数据行按字段名合并到一起的记录。实例代码如下。

```
val  df1 = spark.createDataFrame(
        List( (1, 2, 3), (4, 5, 6) )
    ).toDF("c0", "c1", "c2")
val  df2 = spark.createDataFrame(
        List( (4, 5, 6) )
    ).toDF("c1", "c2", "c0")
df1.intersectAll(df2).show
df1.unionByName(df2).show
```

◇ 创建 DataFrame，字段顺序为 c0 c1 c2

◇ 创建 DataFrame，字段顺序为 c1 c2 c0（顺序有变化）

◇ 求 df1 和 df2 的交集（按字段顺序求交集）
◇ 求 df1 和 df2 的并集（按字段名求并集）

```
scala>
scala> val  df1 = spark.createDataFrame(
    |         List( (1, 2, 3), (4, 5, 6) )
    |      ).toDF("c0", "c1", "c2")
df1: org.apache.spark.sql.DataFrame = [c0: int, c1: int ... 1 more field]
scala> val  df2 = spark.createDataFrame(
    |         List( (4, 5, 6) )
    |      ).toDF("c1", "c2", "c0")
df2: org.apache.spark.sql.DataFrame = [c1: int, c2: int ... 1 more field]
scala> df1.intersectAll(df2).show
+---+---+---+
| c0| c1| c2|
+---+---+---+
|  4|  5|  6|
```

按字段顺序求交集

```
+---+---+---+
scala> df1.unionByName(df2).show
+---+---+---+
| c0| c1| c2|
+---+---+---+
|  1|  2|  3|
|  4|  5|  6|
|  6|  4|  5|
+---+---+---+
scala>
```

按字段名求并集

综上所述，intersectAll()和 unionByName()方法都是用来执行集合运算的，前者是按数据行的字段顺序求相同记录（不考虑字段名），后者则是按字段名进行记录的合并。

（6）join 连接处理。

join 算子用于连接两个 DataFrame 的数据行，包括内连接、左外连接、右外连接、全外连接等方式，默认采用内连接的方式。在使用 join()方法时，可以按一个或多个字段进行连接，下面是一个简单的内连接实例。

```
val  df1 = spark.createDataFrame(
        List( ("a", 1),( "b",2),("c",3) )
     ).toDF("name","num1")
val  df2 = spark.createDataFrame(
        List( ("a", 1), ("b", 4) )
     ).toDF("name", "num2")
val  df3 = df1.join(df2, "name")
df3.show()
```

◇ 构造两个 DataFrame

◇ 按 name 字段执行 df1 和 df2 的内连接

```
scala>
scala> val  df1 = spark.createDataFrame(
    |         List(("a", 1),( "b",2),("c",3) )
    |       ).toDF("name","num1")
df1: org.apache.spark.sql.DataFrame = [name: string, num1: int]
scala> val  df2 = spark.createDataFrame(
    |         List( ("a", 1), ("b", 4) )
    |       ).toDF("name", "num2")
df2: org.apache.spark.sql.DataFrame = [name: string, num2: int]
scala> val  df3 = df1.join(df2, "name")
df3: org.apache.spark.sql.DataFrame = [name: string, num1: int ... 1 more field]
scala> df3.show()
+----+----+----+
|name|num1|num2|
+----+----+----+
|   a|   1|   1|
|   b|   2|   4|
+----+----+----+
scala>
```

df1 含有字段值"a"、"b"和"c"

df2 也含有字段值"a"和"b"

按 name 字段进行内连接

从运行结果容易看出，DataFrame 的内连接与 RDD 的内连接的操作效果是一致的。

**【随堂练习】**

下面是一个包含几名学生信息的 DataFrame，字段分别包含编号、姓名、年龄、综合分数。

```
val df_stu = spark.createDataFrame( List(
       ("a1", "秀儿", 12, 56.5), ("a2", "小丁", 15, 23.0),
       ("a3", "小梅", 23, 84.0), ("a4", "小筱", 9,  93.5) )
   ).toDF("user_id", "name", "age", "score")
df_stu.show()
```

（1）在 df_stu 的基础上新增一列，值为原综合分数字段值的一半。

```
+-------+----+---+-----+---------+
|user_id|name|age|score|score_new|
+-------+----+---+-----+---------+
|     a1|秀儿| 12| 56.5|    28.25|
|     a2|小丁| 15| 23.0|     11.5|
|     a3|小梅| 23| 84.0|     42.0|
|     a4|小筱|  9| 93.5|    46.75|
+-------+----+---+-----+---------+
```

（2）限定数据行数为两行。

```
+-------+----+---+-----+---------+
|user_id|name|age|score|score_new|
+-------+----+---+-----+---------+
|     a1|秀儿| 12| 56.5|    28.25|
|     a2|小丁| 15| 23.0|     11.5|
+-------+----+---+-----+---------+
```

（3）删除原有的 score 列。

```
+-------+----+---+---------+
|user_id|name|age|score_new|
+-------+----+---+---------+
|     a1|秀儿| 12|    28.25|
|     a2|小丁| 15|     11.5|
|     a3|小梅| 23|     42.0|
|     a4|小筱|  9|    46.75|
+-------+----+---+---------+
```

（4）将字段名 user_id 修改为 id。

```
+---+----+---+---------+
| id|name|age|score_new|
+---+----+---+---------+
| a1|秀儿| 12|    28.25|
| a2|小丁| 15|     11.5|
| a3|小梅| 23|     42.0|
| a4|小筱|  9|    46.75|
+---+----+---+---------+
```

（5）将 score_new 字段值增加 10。

```
+---+----+---+---------+
| id|name|age|score_new|
+---+----+---+---------+
| a1|秀儿| 12|    38.25|
| a2|小丁| 15|     21.5|
| a3|小梅| 23|     52.0|
| a4|小筱|  9|    56.75|
+---+----+---+---------+
```

### 3.3.4　DataFrame 的数据操作（SQL）

Spark SQL 除了支持 DSL 形式的数据操作 API 方法，还可以像 Hive 那样直接执行 SQL

语句，这为 Spark 在实际的数据分析工作中提供了极大的便利。Spark SQL 的 SQL 操作是通过 spark.sql()方法实现的，它以 SQL 字符串作为参数，返回一个 DataFrame。不过，Spark SQL 并不支持 SQL 事务、UPDATE 等操作，并且在执行 SQL 操作之前还应将 DataFrame 注册为一张临时视图表（TempView，相当于数据库中的表视图），也被称为“临时数据表”。换句话说，DataFrame 可以当作一张数据表来使用。通过在程序中调用 spark.sql()方法执行 SQL 操作，返回结果为一个新的 DataFrame。

同样地，本节先准备两组数据供后续的实例代码使用，分别是客户信息和购物记录。为方便起见，这两组数据分别存放于 sql_customers.csv 和 sql_shopping.csv 文件中，它们的内容如表 3-4 和表 3-5 所示。

表 3-4　客户信息（customers）

| user_id<br>（客户编号） | name<br>（姓名） | age<br>（年龄） | gender<br>（性别） | phone<br>（电话） | city<br>（城市） |
|---|---|---|---|---|---|
| 1 | 张大明 | 22 | 男 | 138****5678 | Hangzhou |
| 2 | 王芳 | 18 | 女 | 139****4321 | Shanghai |
| 3 | 王小文 | 19 | 男 | 138****3344 | Guangzhou |
| 4 | 陈婷婷 | 18 | 女 | 136****7788 | Hangzhou |
| 5 | 范丽丽 | 20 | 女 | 135****5432 | Hangzhou |

表 3-5　购物记录（shopping）

| sale_id<br>（订单编号） | user_id<br>（客户编号） | sale_date<br>（销售日期） | product_name<br>（产品名称） | price<br>（价格） | location<br>（发货地） |
|---|---|---|---|---|---|
| 1001 | 1 | 2023-10-26 | 智能手机 | 5000 | 深圳 |
| 1002 | 2 | 2023-10-26 | 化妆品套装 | 1200 | 上海 |
| 1003 | 3 | 2023-10-27 | 运动鞋 | 400 | 温州 |
| 1004 | 4 | 2023-10-27 | 时尚女包 | 1500 | 上海 |
| 1005 | 1 | 2023-10-28 | 笔记本电脑 | 8000 | 深圳 |
| 1006 | 1 | 2023-10-28 | 蓝牙耳机 | 300 | 深圳 |
| 1007 | 3 | 2023-10-29 | 书籍 | 200 | 杭州 |
| 1008 | 4 | 2023-10-29 | 儿童玩具 | 400 | 杭州 |
| 1009 | 5 | 2023-10-30 | 家居用品 | 2000 | 无锡 |

### 1. DataFrame 视图表的创建

在通过 SQL 操作 DataFrame 之前，必须先将当前 DataFrame 注册为一张临时视图表才能使用。DataFrame 临时视图表又分为“局部视图表”和“全局视图表”两种形式，前者仅限当前的 SparkSession 会话使用，后者可在当前 Spark 应用程序的所有 SparkSession 实例中访问。在使用全局视图表时，视图表名称需要附带一个“global_temp.”前缀，这是因为全局视图表是要绑定到 global_temp 这个系统保留数据库上的。不过，无论是哪一类视图表，它们在使用完毕后都会被自动删除，这也是被称为“临时视图表”的原因。

假定 sql_customers.csv 和 sql_shopping.csv 这两个数据文件已经上传至本地主目录/home/spark/中。在 SparkShell 交互式编程环境中输入下面的代码查看数据内容。

```
scala>
scala> val  df1 = spark.read.
    |               option("header", true).
    |               option("inferSchema", true).
    |               csv("file:///home/spark/sql_customers.csv")
df1: org.apache.spark.sql.DataFrame = [user_id: int, name: string ... 4 more
fields]
scala> val  df2 = spark.read.
    |               option("header", true).
    |               option("inferSchema", true).
    |               csv("file:///home/spark/sql_shopping.csv")
df2: org.apache.spark.sql.DataFrame = [sale_id: int, user_id: int ... 4 more
scala> df1.createOrReplaceTempView("customers")
scala> df2.createOrReplaceTempView("shopping")
scala> spark.sql("select  *  from  customers").show
+-------+------+---+------+-----------+---------+
|user_id|  name|age|gender|      phone|     city|
+-------+------+---+------+-----------+---------+
|      1| 张大明| 22|    男| 138****5678| Hangzhou|
|      2|   王芳| 18|    女| 139****4321| Shanghai|
|      3| 王小文| 19|    男| 138****3344|Guangzhou|
|      4| 陈婷婷| 18|    女| 136****7788| Hangzhou|
|      5| 范丽丽| 20|    女| 135****5432| Hangzhou|
+-------+------+---+------+-----------+---------+
scala> spark.sql("select  *  from  shopping").show
+-------+-------+----------+------------+-----+--------+
|sale_id|user_id| sale_date|product_name|price|location|
+-------+-------+----------+------------+-----+--------+
|   1001|      1|2023/10/26|    智能手机| 5000|    深圳|
|   1002|      2|2023/10/26|   化妆品套装| 1200|    上海|
|   1003|      3|2023/10/27|      运动鞋|  400|    温州|
|   1004|      4|2023/10/27|    时尚女包| 1500|    上海|
|   1005|      1|2023/10/28|   笔记本电脑| 8000|    深圳|
|   1006|      1|2023/10/28|    蓝牙耳机|  300|    深圳|
|   1007|      3|2023/10/29|        书籍|  200|    杭州|
|   1008|      4|2023/10/29|    儿童玩具|  400|    杭州|
|   1009|      5|2023/10/30|    家居用品| 2000|    无锡|
+-------+-------+----------+------------+-----+--------+
scala>
```

通过 CSV 数据文件创建 DataFrame，文件的第一行均为标题，且自动推断字段类型

分别创建两张局部视图表

通过SQL语句查询局部视图表中的数据

**【随堂练习】**

根据下面的数据构造一个 DataFrame，将其注册为 student 视图表，并将其中的数据内容通过 SQL 语句查询出来。

```
+-------+----+---+-----+
|user_id|name|age|score|
+-------+----+---+-----+
|     a1|秀儿| 12| 56.5|
|     a2|小丁| 15| 23.0|
|     a3|小梅| 23| 84.0|
|     a4|小筱|  9| 93.5|
+-------+----+---+-----+
```

### 2. DataFrame 视图表的 SQL 查询

通过 SQL 查询 DataFrame 的数据需要借助 spark.sql()方法，这个方法返回一个新的 DataFrame。在日常工作中，使用 SQL 语句操作数据是一种很普遍的方式。Spark SQL 实现了 SQL 语句的解析执行，因此只要构造了 SQL 语句就能充分利用 Spark 强大的分布式计算功能，这大大降低了使用 Spark 进行大数据处理与分析的门槛。

下面通过具体的实例来阐述 DataFrame 的 SQL 查询的基本用法，其中，df1 和 df2 对象是分别通过数据文件 sql_customers.csv 和 sql_shopping.csv 创建的 DataFrame，临时视图表 customers 和 shopping 则分别注册自 df1 和 df2 对象。为方便区分，这里构造的 SQL 语句中的关键字用大写字母形式表示（SQL 实际上是不区分字母大小写的）。

```
val  df1 = spark.read.
             option("header", true).
             option("inferSchema", true).
             csv("file:///home/spark/sql_customers.csv")
val  df2 = spark.read.
             option("header", true).
             option("inferSchema", true).
             csv("file:///home/spark/sql_shopping.csv")
df1.createOrReplaceTempView("customers")
df2.createOrReplaceTempView("shopping")
```

◇ 通过 sql_customers.csv 和 sql_shopping.csv 数据文件创建两个 DataFrame，文件的第一行均为标题，字段的数据类型由系统自动推断

◇ 创建在 SQL 语句中使用的 customers 和 shopping 视图表

**【学习提示】**

特别注意：因本节内容较多，如果退出了 SparkShell 交互式编程环境并再次进入，则必须重新执行上面这几行代码，才能在本节使用 df1 和 df2 对象，以及在 SQL 语句中使用 customers 和 shopping 视图表，否则运行接下来的代码会出现错误。

（1）SQL 基本查询。

SQL 基本查询包括 SELECT、SELECT DISTINCT、WHERE、ORDER BY 等。如果读者熟悉 SQL 语法，则很容易看出这部分内容与关系数据库的 SQL 语句基本是一样的。

① 查询指定的字段值。

```
val  df = spark.sql(
       "SELECT  name, city  FROM  customers")
df.show
```

◇ 从 customers 视图表中查询 name 和 city 字段

◇ 格式为 SELECT … FROM …

```
scala>
scala> val  df = spark.sql("SELECT  name,city  FROM  customers")
df: org.apache.spark.sql.DataFrame = [name: string, city: string]
scala> df.show
+------+---------+
```

```
|  name|     city|
+------+---------+
| 张大明| Hangzhou|
|   王芳| Shanghai|
| 王小文|Guangzhou|
| 陈婷婷| Hangzhou|
| 范丽丽| Hangzhou|
+------+---------+
scala>
```

这个实例从 customers 视图表中查询 name 和 city 两个字段，返回一个新的 DataFrame，并将查询到的数据显示出来。

② 查询无重复的字段值。

数据集中的某一列可能包含重复的字段值，如果希望仅列出不同的字段值，则可以增加 DISTINCT 关键字，以返回唯一的字段值或字段值组合。比如查询出所有不同城市的名称。

```
spark.sql(
    "SELECT  DISTINCT  city  FROM  customers").show
```

◇ 从 customers 视图表中查询不同的 city 值，distinct 意为“不同的”

```
scala>
scala> spark.sql("SELECT  DISTINCT  city  FROM  customers").show
+---------+
|     city|
+---------+
|Guangzhou|
| Hangzhou|
| Shanghai|
+---------+
scala>
```

③ 按条件查询。

所谓按条件查询，是指提取那些满足指定条件的数据行。按条件查询是通过 WHERE 子句实现的，条件表达式还可以与 AND、OR、NOT 组合使用。比如查询 user_id 为 1 的客户信息。

```
spark.sql("SELECT * FROM customers WHERE user_id=1").show
```

◇ 从 customers 视图表中查询 user_id=1 的客户信息

```
scala>
scala> spark.sql("SELECT  *  FROM  customers  WHERE  user_id=1").show
+-------+------+---+------+-----------+--------+
|user_id|  name|age|gender|      phone|    city|
+-------+------+---+------+-----------+--------+
|      1| 张大明| 22|    男| 138****5678|Hangzhou|
+-------+------+---+------+-----------+--------+
scala>
```

查询条件

④ 查询结果排序。

对查询出来的数据，如果希望按某个或某些字段排序，则需要使用 ORDER BY 关键字。ORDER BY 默认按升序排列，如果要按降序排列，则可在排序字段后面增加 DESC 关键字（默认为 ASC 升序排列）。比如，查询所有客户信息，并按 age 字段升序排列。

```
spark.sql("SELECT * FROM customers ORDER BY age").show
```

◇ order by 意为“按…排序”

```
scala>
scala> spark.sql("SELECT * FROM customers ORDER BY age").show
+-------+------+---+------+-----------+---------+
|user_id|  name|age|gender|      phone|     city|
+-------+------+---+------+-----------+---------+
|      2|   王芳| 18|    女| 139****4321| Shanghai|
|      4| 陈婷婷| 18|    女| 136****7788| Hangzhou|
|      3| 王小文| 19|    男| 138****3344|Guangzhou|
|      5| 范丽丽| 20|    女| 135****5432| Hangzhou|
|      1| 张大明| 22|    男| 138****5678| Hangzhou|
+-------+------+---+------+-----------+---------+
scala>
```

**【随堂练习】**

使用“1. DataFrame 视图表的创建”的“随堂练习”中注册的 student 视图表构造 SQL 语句，实现以下功能。

① 查询所有学生的姓名。

② 查询年龄超过 12 岁的学生。

③ 查询分数大于 60 分的学生，并按分数从高到低排序。

（2）SQL 高级查询。

针对 DataFrame 的高级查询可使用 SELECT…LIMIT、SELECT…LIKE 等。下面通过代码实例阐述它们的使用方法。

① 查询指定数量的记录。

SELECT…LIMIT 可以限定查询返回的行数，这对拥有大量数据的数据集来说比较有用。比如查询 customers 视图表中的前两个客户信息。

```
spark.sql("SELECT * FROM customers LIMIT 2").show
```

◇ limit 意为“限制”

```
scala>
scala> spark.sql("SELECT * FROM customers LIMIT 2").show
+-------+------+---+------+-----------+--------+
|user_id|  name|age|gender|      phone|    city|
+-------+------+---+------+-----------+--------+
|      1| 张大明| 22|    男| 138****5678|Hangzhou|
|      2|   王芳| 18|    女| 139****4321|Shanghai|
+-------+------+---+------+-----------+--------+
```

上面这条 SQL 语句还可以加入各种筛选条件，或者按要求排序。也就是说，设定要返回

的行数，只需在 SQL 语句的末尾加上 LIMIT 关键字及限定数值即可，其他仍可按一般的 SQL 语法处理。例如：

```
scala>
scala> spark.sql(
     |    """ SELECT  *  FROM  customers
     |          WHERE  age < 20
     |          ORDER  BY  city  DESC  LIMIT  2""").show
+-------+------+---+------+-----------+--------+
|user_id|  name|age|gender|      phone|    city|
+-------+------+---+------+-----------+--------+
|      2|   王芳| 18|    女| 139****4321|Shanghai|
|      4| 陈婷婷| 18|    女| 136****7788|Hangzhou|
+-------+------+---+------+-----------+--------+
```

② 模糊查询。

模糊查询是指字段值是否符合某种匹配模式，而不是精确匹配。SQL 语句中的 LIKE 关键字可用于模糊匹配 WHERE 字段的内容。比如查询电话号码“以 138 开头”的所有客户信息。

```
spark.sql(""" SELECT  *  FROM  customers
                WHERE  phone  LIKE  '138%' """).show
```

◇ like 意为“相似”
◇ %代表任意字符

```
scala>
scala> spark.sql(""" SELECT  *  FROM  customers
     |                WHERE  phone  LIKE  '138%' """).show
+-------+------+---+------+-----------+---------+
|user_id|  name|age|gender|      phone|     city|
+-------+------+---+------+-----------+---------+
|      1| 张大明| 22|    男| 138****5678| Hangzhou|
|      3| 王小文| 19|    男| 138****3344|Guangzhou|
+-------+------+---+------+-----------+---------+
scala>
```

SQL 语句中的字符串值用单引号引起来

**【学习提示】**

SQL 查询在执行模糊匹配时，百分号（%）代表 0 或任意多个字符，下画线（_）代表单个任意字符。

③ IN 和 BETWEEN 查询。

SQL 的 IN 关键字支持在 WHERE 子句中规定多个值，BETWEEN 关键字用于选取介于两个值之间的范围。例如，下面的第一条 SQL 语句用于查询 city 字段值为“Hangzhou”和“Shanghai”的所有客户信息。第二条 SQL 语句用于查询 age 字段值介于 18 ~ 20 之间的所有客户信息。

```
spark.sql("""SELECT  *  FROM  customers  WHERE
               city  IN  ('Hangzhou', 'Shanghai') """).show
spark.sql("""SELECT  *  FROM  customers  WHERE
               age  BETWEEN  18  AND  20 """).show
```

◇ between…and 意为“在…之间”

```
scala>
scala> spark.sql(
    |    """ SELECT  *  FROM  customers
    |          WHERE  city  IN  ('Hangzhou', 'Shanghai') """).show
+-------+------+---+------+-----------+--------+
|user_id|  name|age|gender|      phone|    city|
+-------+------+---+------+-----------+--------+
|      1| 张大明| 22|    男| 138****5678|Hangzhou|
|      2|   王芳| 18|    女| 139****4321|Shanghai|
|      4| 陈婷婷| 18|    女| 136****7788|Hangzhou|
|      5| 范丽丽| 20|    女| 135****5432|Hangzhou|
+-------+------+---+------+-----------+--------+
scala> spark.sql(
    |    """ SELECT  *  FROM  customers
    |          WHERE  age  BETWEEN  18  AND  20 """).show
+-------+------+---+------+-----------+---------+
|user_id|  name|age|gender|      phone|     city|
+-------+------+---+------+-----------+---------+
|      2|   王芳| 18|    女|139****4321| Shanghai|
|      3| 王小文| 19|    男|138****3344|Guangzhou|
|      4| 陈婷婷| 18|    女|136****7788| Hangzhou|
|      5| 范丽丽| 20|    女|135****5432| Hangzhou|
+-------+------+---+------+-----------+---------+
scala>
```

**【随堂练习】**

使用“1. DataFrame 视图表的创建”的“随堂练习”中注册的 student 视图表构造 SQL 语句，实现以下功能。

① 查询分数排在前两名的学生信息。

② 查询姓名以“小”字开头的学生信息，并按分数由高到低排序。

③ 查询姓名中不包含“小”字的学生信息。

④ 查询年龄介于 12 ~ 20 之间的学生信息。

（3）SQL 连接查询。

SQL 连接查询使用 JOIN 关键字，按指定的字段将来自两张或两张以上数据表的数据行结合起来。其中，INNER JOIN 是最常用的连接方式，用于从多张数据表中根据它们之间的关系提取匹配的记录，此外，还有 LEFT JOIN、RIGHT JOIN、FULL OUTER JOIN 等连接方式。例如，查询客户购买过的产品及销售日期。

```
spark.sql("""SELECT  a.name, b.product_name, b.sale_date
               FROM  customers  a  INNER  JOIN  shopping  b
               ON  a.user_id = b.user_id """).show
```

◇ INNER JOIN 为内连接

```
scala>
scala> spark.sql(""" SELECT  a.name, b.product_name, b.sale_date
```

```
     |                  FROM  customers  a  INNER  JOIN  shopping  b
     |                  ON  a.user_id = b.user_id """).show
+------+------------+----------+
|  name|product_name| sale_date|
+------+------------+----------+
| 张大明|      智能手机| 2023/10/26|
|   王芳|     化妆品套装| 2023/10/26|
| 王小文|       运动鞋| 2023/10/27|
| 陈婷婷|      时尚女包| 2023/10/27|
| 张大明|     笔记本电脑| 2023/10/28|
| 张大明|      蓝牙耳机| 2023/10/28|
| 王小文|         书籍| 2023/10/29|
| 陈婷婷|      儿童玩具| 2023/10/29|
| 范丽丽|      家居用品| 2023/10/30|
+------+------------+----------+
scala>
```

INNER JOIN 需要指定按什么字段进行连接

上述代码用到了 SQL 别名机制以简化 SQL 语句的表达形式，其中 customers 的别名被设为 a，shopping 的别名被设为 b。此外，我们还可以根据需要设置字段的别名，字段别名会出现在查询的结果中。

**【随堂练习】**

基于“1. DataFrame 视图表的创建”的“随堂练习”中注册的 student 视图表新增一张课程成绩 course 表，构造 SQL 语句，实现以下功能，并对比它们的运行结果有什么异同。

```
+-------+----+---+-----+     +---+-----+------+----+
|user_id|name|age|score|     | id|spark|hadoop|java|
+-------+----+---+-----+     +---+-----+------+----+
|     a1|秀儿| 12| 56.5|     | a3|   80|    78|  90|
|     a2|小丁| 15| 23.0|     | a1|   30|    66|  70|
|     a3|小梅| 23| 84.0|     | a2|   60|    70|  60|
|     a4|小筱|  9| 93.5|     | a5|   76|    90|  80|
+-------+----+---+-----+     +---+-----+------+----+
```

① 使用 INNER JOIN 查询所有学生的课程成绩。

② 使用 LEFT JOIN 查询所有学生的课程成绩。

③ 使用 RIGHT JOIN 查询所有学生的课程成绩。

④ 使用 FULL OUTER JOIN 查询所有学生的课程成绩。

（4）SQL 嵌套查询/子查询。

对于简单的数据业务，使用基本的查询功能就可以解决问题。如果问题不是那么直接，可能就需要通过多次嵌套查询来解决。例如，查询价格（price 字段）超过 1000 元的产品是哪些客户购买的。

```
spark.sql("""SELECT  *  FROM  customers  WHERE  user_id  IN
       (SELECT  user_id  FROM  shopping  WHERE  price>1000)"""
   ).show
scala>
```

◇ 嵌套查询子句为 SQL 语句中括号部分的内容

```
scala> spark.sql("""SELECT  *  FROM  customers  WHERE  user_id  IN
     |            (SELECT  user_id  FROM  shopping  WHERE  price>1000)""").show
+-------+------+---+------+----------+--------+
|user_id|  name|age|gender|     phone|    city|
+-------+------+---+------+----------+--------+
|      1| 张大明| 22|    男| 138****5678|Hangzhou|
|      2|   王芳| 18|    女| 139****4321|Shanghai|
|      4| 陈婷婷| 18|    女| 136****7788|Hangzhou|
|      5| 范丽丽| 20|    女| 135****5432|Hangzhou|
+-------+------+---+------+----------+--------+
scala>
```

嵌套查询执行时，会生成一张临时表，这张表只有一个 user_id 字段

这里的“(SELECT user_id FROM shopping WHERE price>1000)”得到的是仅有一个字段的临时表，然后使用 IN 关键字将其转换为一个查询条件，从而得到二次查询的结果。

**【随堂练习】**

使用“（3）SQL 连接查询”的“随堂练习”中的 student 表和 course 表构造 SQL 语句，查询 Spark 和 Hadoop 课程成绩均超过 70 分的学生信息。

（5）SQL 聚合查询。

与关系数据库类似，Spark SQL 也实现了常用的聚合函数功能，包括 count()计数、avg()求平均值、max()求最大值、min()求最小值等。例如，求 shopping 视图表中 price 字段的平均值。

```
spark.sql(
"SELECT  avg(price)  AS  average_price  FROM  shopping"
).show
```

◇ avg()函数用于求平均值

```
scala>
scala> spark.sql("SELECT  avg(price)  AS  average_price  FROM  shopping").show
+------------------+
|     average_price|
+------------------+
|2111.1111111111113|
+------------------+
scala>
```

**【随堂练习】**

使用“（3）SQL 连接查询”的“随堂练习”中的 student 表和 course 表构造 SQL 语句，查询 Spark 课程成绩超过该课程平均分的学生信息。

（6）SQL 分组统计查询。

分组统计查询是数据处理中经常会遇到的操作。在 Spark SQL 中，分组统计是通过关键字 GROUP BY 来实现的，分组统计通常要结合聚合函数一起使用。例如，统计 customers 视图表中各年龄段的人数。

```
spark.sql(""" SELECT  age, count(age)  AS  nums
              FROM  customers  GROUP  BY  age """).show
```

◇ group by 意为“按…分组”

```
scala>
scala> spark.sql(""" SELECT  age, count(age)  AS  nums  FROM  customers
     |                  GROUP  BY  age """).show
+---+----+
|age|nums|
+---+----+
| 22|   1|
| 20|   1|
| 19|   1|
| 18|   2|
+---+----+
scala>
```

nums 是统计结果字段的别名

**【随堂练习】**

统计 shopping 视图表中每个发货地的平均价格信息。

（7）用户自定义函数。

Spark SQL 内置的 org.apache.spark.sql.functions 模块中包含各种可直接使用的数据处理函数，包括聚合函数、标量值函数等，以支持对 DataFrame 的行或列数据进行相应处理。表 3-6 列出了 org.apache.spark.sql.functions 模块中的部分常用内置函数。

表 3-6　org.apache.spark.sql.functions 模块中的部分常用内置函数

| 类型 | 常用内置函数 |
| --- | --- |
| 字符串函数 | lower()、upper()、substr()、concat()、startsWith()、regexp_replace()、regexp_extract() |
| 数学函数 | abs()、ceil()、floor()、log()、round()、sqrt() |
| 统计函数 | avg()、max()、min()、mean()、count()、stddev() |
| 日期函数 | datediff()、date_add()、from_utc_timestamp() |
| 编解码函数 | md5()、sha1()、sha2() |
| 窗口函数 | over()、rank()、dense_rank()、row_number()、percent_rank()、lead()、lag()、ntile() |

虽然 Spark SQL 提供了很多内置函数供用户使用，但在实际工作中仍有不少问题是内置函数无法解决的，或者使用内置函数实现起来较为烦琐，此时可能要通过用户自定义函数来完成更复杂的逻辑。所谓用户自定义函数（User Defined Function，UDF），是指通过 Spark 支持的编程语言定义一个函数传递给 Spark SQL，使用起来就像内置的 abs()、lower()等函数一样。在用户自定义函数中，可以灵活运用编程语言提供的各种函数、方法、模块库等实现所需功能，从而不受 Spark SQL 的限制。

除了用户自定义函数，Spark SQL 还支持用户自定义的聚合函数（User Defined Aggregate Functions，UDAF）和用户自定义的表生成函数（User Defined Table Generating Function，UDTF）。其中，UDF 相当于最基本的函数，提供了对 SQL 字段的转换功能，是“一行输入一行输出”；UDAF 代表聚合函数，为 Spark SQL 提供了对字段的聚合功能，类似 max()、min()、count()等函数，是“多行输入一行输出”；UDTF 代表表生成函数，是“一行输入多行输出”。

下面介绍最基本的 UDF 实例，其他两种自定义函数处理起来相对要更复杂。

```
// 自定义一个处理函数，传入一个 String 并返回一个新的 String
def  convert(s: String): String = {
    return "#" + s.toUpperCase + "#"
}

// 用户自定义函数要先在 Spark SQL 中注册才能使用
spark.udf.register("myconvert", convert(_) )

// 在 SQL 中调用用户自定义函数
spark.sql(
    "SELECT  name,myconvert(city)  FROM  customers").show
```

◇ 自定义一个 convert() 函数，将字符串转换为大写字母且前后各用一个“#”符号连接

◇ 注册用户自定义函数（为避免混淆，这里使用了新名字）

◇ 可以像使用 Spark SQL 内置函数一样调用 myconvert() 函数

```
scala>
scala> def  convert(s: String): String = {
     |      return "#" + s.toUpperCase + "#"
     |  }
convert: (s: String)String
scala> spark.udf.register("myconvert", convert(_) )
res10: org.apache.spark.sql.expressions.UserDefinedFunction = SparkUserDefined
Function($Lambda$5000/460845124@4070d628,StringType,List(Some(class[value[0]:
string])),Some(class[value[0]: string]),Some(myconvert),true,true)
scala> spark.sql("SELECT  name, myconvert(city)  FROM  customers").show
+------+---------------+
|  name|myconvert(city)|
+------+---------------+
| 张大明|     #HANGZHOU#|
|   王芳|     #SHANGHAI#|
| 王小文|    #GUANGZHOU#|
| 陈婷婷|     #HANGZHOU#|
| 范丽丽|     #HANGZHOU#|
+------+---------------+
scala>
```

自定义一个 convert()函数

在 Spark SQL 中注册用户自定义函数

在 SQL 语句中调用用户自定义函数

从运行结果容易看出，city 字段的内容已经变成原字段值的大写字母形式且前后各多了一个“#”符号。

Spark SQL 的用户自定义函数不仅可以应用在 SQL 中，而且支持 DSL 形式的 API 操作。下面是一个在 DSL 操作中调用用户自定义函数的代码实例。

```
val  convudf =
       spark.udf.register("myconvert", convert(_) )
df1.select($"name", convudf($"city")).show
```

◇ 注册用户自定义函数，返回一个用户自定义函数对象

◇ 调用 convudf 函数对象

```
scala>
scala> val  convudf = spark.udf.register("myconvert", convert(_) )
convudf: org.apache.spark.sql.expressions.UserDefinedFunction = SparkUserDefin
edFunction($Lambda$5021/382878114@770f48fd,StringType,List(Some(class[value[
0]: string])),Some(class[value[0]: string]),Some(myconvert),true,true)
```

```
scala> df1.select($"name", convudf($"city")).show
+------+---------------+
|  name|myconvert(city)|
+------+---------------+
| 张大明|      #HANGZHOU#|
|   王芳|      #SHANGHAI#|
| 王小文|     #GUANGZHOU#|
| 陈婷婷|      #HANGZHOU#|
| 范丽丽|      #HANGZHOU#|
+------+---------------+
scala>
```

在 DSL 操作中直接调用 convudf 函数对象。这里不能使用"myconvert"，因为它只是一个字符串名字

【随堂练习】

使用“1. DataFrame 视图表的创建”的“随堂练习”中注册的 student 视图表，自定义一个函数，实现将 score 自动转换为 5 级分制，即低于 60 分为不及格，60～69 分为及格，70～79 分为中等，80～89 分为良好，90 分以上为优秀。例如：

```
+-------+----+---+-----+
|user_id|name|age|score|
+-------+----+---+-----+
|     a1|秀儿| 12| 56.5|
|     a2|小丁| 15| 23.0|
|     a3|小梅| 23| 84.0|
|     a4|小筱|  9| 93.5|
+-------+----+---+-----+
```

```
+-------+----+---+---------+
|user_id|name|age|new_score|
+-------+----+---+---------+
|     a1|秀儿| 12|   不及格|
|     a2|小丁| 15|   不及格|
|     a3|小梅| 23|     良好|
|     a4|小筱|  9|     优秀|
+-------+----+---+---------+
```

**提示：**先使用 withColumn()方法在 DataFrame 中新增一个 new_score 字段（数据类型为字符串），然后仿照上述实例代码编写一个用户自定义函数，实现“按 score 字段设置 new_score 字段值”的功能。



## 3.4　Spark SQL 数据处理实例

### 3.4.1　人口信息统计实例

假定有一批包含 600 万人口信息的数据存储在当前主目录的 people_info.csv 文本文件中，每行数据代表一个人的基本信息，其中的 3 个字段分别是编号、性别（F/M，分别对应女性和男性）、身高（单位为 cm），部分样本数据内容如下。

```
spark@ubuntu:~$ head people_info.csv
1,F,180
2,M,146
3,F,198
4,F,196
5,M,197
6,F,202
7,M,184
8,M,153
9,M,218
10,M,214
```

```
spark@ubuntu:~$ tail people_info.csv
5999991,F,155
5999992,M,140
5999993,M,137
5999994,F,181
5999995,M,205
5999996,M,150
5999997,F,192
5999998,M,164
5999999,M,206
6000000,F,203
```

现要求使用 Spark SQL 完成以下数据分析任务。

（1）统计男性身高超过 170cm 以及女性身高超过 165cm 的总人数。

（2）按照性别分组统计男性和女性人数。

（3）统计身高大于 210cm 的前 50 名男性，并按身高从高到低排序。

（4）统计男性的平均身高。

（5）统计女性身高的最大值。

根据问题要求，我们先简单分析一下数据，这是一个以逗号分隔的标准 CSV 格式的文本文件，且不存在标题行，因此需要构造数据的字段结构信息（id、gender 和 height）并将其转换成 DataFrame。当数据准备完毕后，就可以针对每个分析任务编写 SQL 语句，其都是一些基本的 SQL 查询和统计操作。

在 SparkShell 交互式编程环境中输入下面的代码。

定义字段结构信息

```
scala>
scala> val  pepschema = "id  LONG, gender  STRING, height  INT"
pepschema: String = id  LONG, gender  STRING, height  INT
scala> val  df = spark.read.
     |        schema(pepschema).
     |        csv("file:///home/spark/people_info.csv")
df: org.apache.spark.sql.DataFrame = [id: bigint, gender: string ... 1 more
field]
scala> df.createOrReplaceTempView("people_info")
scala>
```

将 DataFrame 注册为临时视图表

上述代码首先将 people_info.csv 文件加载进来并转换成 DataFrame 对象，相当于一张包含 600 万条数据的表格，然后将其注册为 people_info 视图表。

接下来考虑第 1 个问题，即按照要求的条件统计人数，这需要用到 SQL 的 count()统计函数，代码如下。

```
scala>
scala> spark.sql(
     |    """ SELECT  count(id)  FROM  people_info
     |          WHERE  (height>170  and  gender='M')
     |             OR  (height>165 and gender='F') """).show
+---------+
|count(id)|
+---------+
|  3091252|
+---------+
scala>
```

统计男性身高超过 170cm 以及女性身高超过 165cm 的总人数。需要注意 WHERE 的两个主条件应使用 OR 运算，而不是 AND

第 2 个问题实际是一个分组统计问题，可以先通过 GROUP BY 关键字按指定字段进行分组，然后将分组后的字段应用 count()、avg()、max()等聚合函数。这里的分组统计功能实现代码如下。

```
scala>
scala> spark.sql(""" SELECT  gender, count(gender)  FROM  people_info
     |                 GROUP  BY  gender """).show
+------+-------------+
|gender|count(gender)|
+------+-------------+
|     F|      2999577|
|     M|      3000423|
+------+-------------+
scala>
```

按照性别分组统计男性和女性人数

第 3 个问题是统计身高大于 210cm 的前 50 名男性，并按身高从高到低排序，因此需要使用 ORDER BY 和 LIMIT 关键字，实现代码如下。

```
scala>
scala> spark.sql(
     |    """ SELECT  *  FROM  people_info
     |          WHERE  height>210  AND  gender='M'
     |          ORDER  BY  height  DESC
     |          LIMIT  50 """).show
+-------+------+------+
|     id|gender|height|
+-------+------+------+
|3191220|     M|   219|
|   2031|     M|   219|
|3193350|     M|   219|
|   6784|     M|   219|
...
scala>
```

将身高大于 210cm 的男性从高到低排序，并取前 50 名

第 4 个问题是统计男性的平均身高，需要使用 avg()函数，实现代码如下。

```
scala>
scala> spark.sql(
     |   "SELECT  avg(height)  FROM  people_info  WHERE  gender='M' ").show
+------------------+
|       avg(height)|
+------------------+
|169.48521191845282|
+------------------+
scala>
```

统计男性的平均身高

第 5 个问题是统计女性身高的最大值，这个问题也比较简单，实现代码如下。

```
scala>
scala> spark.sql(
     |    "SELECT  max(height)  FROM  people_info  WHERE  gender='F' ").show
```

统计女性身高的最大值

```
+-----------+
|max(height)|
+-----------+
|        219|
+-----------+
scala>
```

至此，人口信息的各项统计任务就全部完成了。本实例的内容比较基础，侧重点主要在于如何按照问题要求将正确的 SQL 语句构造出来，读者在学习时可以对照数据库相关课程中的 SQL 内容，以加深对上述代码的理解。

### 3.4.2 电影评分数据分析实例

MovieLens 是一个关于电影评分的公开数据集，里面包含了大量的 IMDB（Internet Movie DataBase，互联网电影数据库）的电影评分信息，所以经常被用作推荐系统、机器学习算法的测试数据集。MovieLens 数据集的 movies.csv 和 ratings.csv 文件中分别存放了电影信息、电影的用户评分数据，其中的部分样本数据如下（第 1 行都是标题行）。

```
spark@ubuntu:~/ml-25m$ head movies.csv
movieId,title,genres
1,Toy Story (1995),Adventure|Animation|Children|Com
2,Jumanji (1995),Adventure|Children|Fantasy
3,Grumpier Old Men (1995),Comedy|Romance
4,Waiting to Exhale (1995),Comedy|Drama|Romance
5,Father of the Bride Part II (1995),Comedy
6,Heat (1995),Action|Crime|Thriller
7,Sabrina (1995),Comedy|Romance
8,Tom and Huck (1995),Adventure|Children
9,Sudden Death (1995),Action
```

```
spark@ubuntu:~/ml-25m$ head ratings.csv
userId,movieId,rating,timestamp
1,296,5.0,1147880044
1,306,3.5,1147868817
1,307,5.0,1147868828
1,665,5.0,1147878820
1,899,3.5,1147868510
1,1088,4.0,1147868495
1,1175,3.5,1147868826
1,1217,3.5,1147878326
1,1237,5.0,1147868839
```

假定这两个文件置于/home/spark/ml-25m 目录（如果没有该目录，则应先用 mkdir 命令创建，然后将用到的两个 CSV 文件上传到里面）中，其中，movies.csv 文件的大小约为 2.9MB，包含 6 万多部电影的信息，数据格式为[movieId,title,genres]，即[电影 ID,电影名称,电影类别]；ratings.csv 文件中包含电影的用户评分数据，大小约为 647MB，数据格式为[userId,movieId,rating,timestamp]，即[用户 ID,电影 ID,用户评分,时间戳]。这里的用户评分采用 5 星制且按半颗星的规模递增（0.5 ~ 5），且每个用户只能对同一部电影评分一次，时间戳则是自 1970 年 1 月 1 日 0 点到用户提交评分时经过的毫秒数。

现要求使用 Spark SQL 完成以下数据分析任务。

（1）查找电影评分排名前 10 的电影。

（2）查找电影评分次数超过 5000 次，且平均评分排名前 10 的电影及其对应的平均评分。

为了解决这两个问题，我们先对它们进行简单的分析。第 1 个问题实际是获取电影的关注度信息，因此首先要统计出每部电影的评分次数（即发布评分的用户数），然后对其进行降序排列，获取前 10 部电影。第 2 个问题包含两个条件：一是电影评分次数超过 5000 次，二是平均评分排名位于前 10。

第 1 个问题的实现，可以将问题分成两步查询的方式。在 SparkShell 交互式编程环境中输入下面的代码。

```
scala>
scala> val  df1 = spark.read.
     |      option("header", true).
     |      option("inferSchema", true).
     |      csv("file:///home/spark/ml-25m/movies.csv")
df1: org.apache.spark.sql.DataFrame = [movieId: int, title: string ...
scala> val df2 = spark.read.
     |      option("header", true).
     |      option("inferSchema", true).
     |     csv("file:///home/spark/ml-25m/ratings.csv")
df2: org.apache.spark.sql.DataFrame = [userId: int, movieId: int ... 2 more
fields]
scala> df1.createOrReplaceTempView("movies")
scala> df2.createOrReplaceTempView("ratings")
scala>
scala> val  df_max10 = spark.sql(
     | """ SELECT  movieId, count(userId)  AS  cnt  FROM  ratings
     |      GROUP  BY  movieId
     |      ORDER  BY  cnt  DESC  LIMIT  10 """)
df_max10: org.apache.spark.sql.DataFrame = [movieId: int, cnt: bigint]
scala> df_max10.createOrReplaceTempView("max10_ratings")
scala>
scala>
```

分别从 movies.csv 和 ratings.csv 文件中读取数据，header 为标题行，inferSchema 用于设置自动推断字段类型

分别将两个 DataFrame 注册为临时视图表

按 movieId 字段进行分组，计算每部电影的评分用户数 cnt，因为每个用户对每部电影只能评分一次

按评分用户数 cnt 进行降序排列，获取前 10 部电影

将查询得到的数据集注册为临时视图表

上面是第 1 步，先按 movieId 字段对数据进行分组，将每部电影的评分用户数统计出来，然后对评分用户数进行降序排列并获取前 10 部电影，最后将查询出来的数据集注册成一张 max10_ratings 临时视图表。

```
scala>
scala> val  df_movie = spark.sql(
     | """ SELECT  a.movieId, a.title, a.genres, b.cnt
     |       FROM  movies  a, max10_ratings  b
     |       WHERE  a.movieId = b.movieId
     |       ORDER  BY  cnt  DESC """)
df_movie: org.apache.spark.sql.DataFrame = [movieId: int, title: string ...
scala> df_movie.show
[Stage 4:====================>                                  (2 + 2) / 6]
+-------+--------------------+--------------------+-----+
|movieId|               title|              genres|  cnt|
+-------+--------------------+--------------------+-----+
|    356| Forrest Gump (1994)|Comedy|Drama|Roma...| 81491|
|    318|Shawshank Redempt...|         Crime|Drama| 81482|
|    296| Pulp Fiction (1994)|Comedy|Crime|Dram...| 79672|
|    593|Silence of the La...|Crime|Horror|Thri...| 74127|
```

连接 max10_ratings 和 movies 这两张视图表，查询出这 10 部电影的详细信息，并按评分用户数进行降序排列

《阿甘正传》
《肖申克的救赎》

```
|   2571|  Matrix, The (1999)|Action|Sci-Fi|Thr...| 72674|
|    260|Star Wars: Episod...|Action|Adventure|...| 68717|
|    480|Jurassic Park (1993)|Action|Adventure|...| 64144|
|    527|Schindler's List ...|           Drama|War| 60411|
|    110|   Braveheart (1995)|    Action|Drama|War| 59184|
|   2959|   Fight Club (1999)|Action|Crime|Dram...| 58773|
+-------+--------------------+--------------------+-----+
scala>
```

《辛德勒的名单》

《勇敢的心》

由于第 1 步查询出来的前 10 部电影只有电影的 movieId，因此还需根据这个 movieId 从 movies 数据表中得到电影的详细信息。所以第 2 步所做的，就是通过内连接将每部电影关联的数据查询出来。

接下来，我们将以上两个步骤合并，直接通过嵌套查询的方式一次性实现。继续输入下面的代码。

```
scala>
scala> val  df_movie = spark.sql(
     | """ SELECT  a.movieId, a.title, a.genres, b.cnt
     |       FROM  movies  a,
     |         ( SELECT  movieId, count(userId)  AS  cnt  FROM  ratings
     |             GROUP  BY  movieId
     |             ORDER  BY  cnt  DESC  LIMIT  10 )  AS  b
     |      WHERE  a.movieId = b.movieId
     |      ORDER  BY  cnt  DESC """)
df_movie: org.apache.spark.sql.DataFrame = [movieId: int, title: string ...
scala> df_movie.show
[Stage 9:==================>                                (2 + 2) / 6]
+-------+--------------------+--------------------+-----+
|movieId|               title|              genres|  cnt|
+-------+--------------------+--------------------+-----+
|    356| Forrest Gump (1994)|Comedy|Drama|Roma...| 81491|
|    318|Shawshank Redempt...|         Crime|Drama| 81482|
|    296| Pulp Fiction (1994)|Comedy|Crime|Dram...| 79672|
|    593|Silence of the La...|Crime|Horror|Thri...| 74127|
|   2571|  Matrix, The (1999)|Action|Sci-Fi|Thr...| 72674|
|    260|Star Wars: Episod...|Action|Adventure|...| 68717|
|    480|Jurassic Park (1993)|Action|Adventure|...| 64144|
|    527|Schindler's List ...|           Drama|War| 60411|
|    110|   Braveheart (1995)|    Action|Drama|War| 59184|
|   2959|   Fight Club (1999)|Action|Crime|Dram...| 58773|
+-------+--------------------+--------------------+-----+
scala>
```

在嵌套的子查询中，通过 movieId 分组查询出评分用户数，并进行降序排列获取前 10 部电影，得到的查询结果当作表 b

连接表 a 和表 b，获取前 10 部电影的详细信息，其中，a 是 movies 表的别名，b 是嵌套查询生成的临时表

这里合并起来的嵌套查询，是将子查询直接嵌入外部查询中，子查询得到的数据集被命名为 b，然后完成两张表（a 和 b）的连接，从而得到前 10 部电影的详细信息，查询的结果与

单独分为两步的做法完全相同。

接下来考虑第 2 个问题，只需在第 1 个问题的基础上稍做调整，首先增加一个“评分次数大于 5000 次”的条件，再计算平均评分，并进行降序排列获取前 10 行，最后在聚合函数的基础上通过 HAVING 子句处理“评分次数大于 5000 次”的条件。继续在 SparkShell 交互式编程环境中输入下面的代码。

在嵌套的子查询中，通过 movieId 分组查询出评分用户数超过 5000 个的记录，并按评分的平均值进行降序排列，获取前 10 部电影，得到的查询结果当作表 b

分组统计中包含评分用户数、评分的平均值

连接表 a 和表 b，获取前 10 部电影的详细信息，其中 a 是 movies 表的别名，b 是嵌套查询生成的临时表

```
scala>
scala> val  df2_movie = spark.sql(
     | """ SELECT  a.movieId, a.title, a.genres, b.avgr
     |      FROM  movies  a,
     |      ( SELECT  movieId, count(userId)  AS  cnt, avg(rating)  AS  avgr
     |          FROM  ratings
     |          GROUP  BY  movieId
     |          HAVING  cnt > 5000
     |          ORDER  BY  avgr  DESC  LIMIT  10 )  AS  b
     |      WHERE  a.movieId = b.movieId
     |      ORDER  BY  avgr  DESC """)
df2_movie: org.apache.spark.sql.DataFrame = [movieId: int, title: string ...
scala> df2_movie.show
[Stage 14:=======================================>                 (4 + 2) / 6]
+-------+--------------------+--------------------+------------------+
|movieId|               title|              genres|              avgr|
+-------+--------------------+--------------------+------------------+
|    318|Shawshank Redempt...|       Crime|Drama| 4.413576004516335|
|    858|Godfather, The (1...|       Crime|Drama| 4.324336165187245|
|     50|Usual Suspects, T...|Crime|Mystery|Thr...| 4.284353213163313|
|   1221|Godfather: Part I...|       Crime|Drama|4.2617585117585115|
|   2019|Seven Samurai (Sh...|Action|Adventure|...|  4.25476920775043|
|    527|Schindler's List ...|         Drama|War| 4.247579083279535|
|   1203| 12 Angry Men (1957)|             Drama| 4.243014062405697|
|    904|  Rear Window (1954)|   Mystery|Thriller| 4.237947624243627|
|   2959|   Fight Club (1999)|Action|Crime|Dram...| 4.228310618821568|
|   1193|One Flew Over the...|             Drama| 4.2186616007543405|
+-------+--------------------+--------------------+------------------+
scala>
```

《肖申克的救赎》

《教父》

《辛德勒的名单》

至此，有关电影评分的数据分析问题已经全部得到解决。从上面的代码中可以看出，Spark SQL 大数据处理工作中一个非常重要的功底是根据问题需求编写合适的 SQL 语句。

## 3.5　DataFrame 创建和保存

与 RDD 类似，Spark SQL 提供了多种不同的途径来创建和保存 DataFrame。在 3.3.1 节

中，我们分别介绍过通过集合、CSV 文件等途径创建 DataFrame 的方法，本节介绍如何通过 JSON 文件、Parquet 文件创建 DataFrame。此外，当要持久化保存 DataFrame 时，Spark SQL 支持将其存储到 CSV、JSON 及 Parquet 等格式的数据文件中。

### 3.5.1 创建 DataFrame

#### 1. 通过 JSON 文件创建 DataFrame

首先准备一个空的 JSON 文件，然后在其中写入几条测试用的人口信息数据。在 Linux 终端窗体中输入下面的 Shell 命令和文件内容（注意：不是在 SparkShell 交互式编程环境中）。

```
spark@ubuntu-vm:~$
spark@ubuntu-vm:~$ cd  ~
spark@ubuntu-vm:~$ cat  >  people_info.json
{"id":1, "gender":"F", "height":180}
{"id":2, "gender":"M", "height":146}
{"id":3, "gender":"F", "height":198}
spark@ubuntu-vm:~$
```

切换到当前主目录，如果已在，则忽略该步

将测试数据输入 people_info.json 文件中

输入完毕后，在这里按 Ctrl+D 快捷键

切换到 SparkShell 交互式编程环境，输入下面的代码。

```
scala>
scala> val df1 = spark.read.json(
     |       "file:///home/spark/people_info.json")
df1: org.apache.spark.sql.DataFrame = [gender: string, height: bigint ...
scala> val df2 = spark.read.format("json").load(
     |       "file:///home/spark/people_info.json")
df2: org.apache.spark.sql.DataFrame = [gender: string, height: bigint ... 1
scala> df1.show
+------+------+---+
|gender|height| id|
+------+------+---+
|     F|   180|  1|
|     M|   146|  2|
|     F|   198|  3|
+------+------+---+
scala> df2.show
+------+------+---+
|gender|height| id|
+------+------+---+
|     F|   180|  1|
|     M|   146|  2|
|     F|   198|  3|
+------+------+---+
scala>
```

方式 1：使用 json()方法直接加载 JSON 文件

方式 2：首先使用 format()方法设定加载的文件类型，然后使用 load()方法加载 JSON 文件

这里列举了两种加载 JSON 文件的方法，其中，json()方法针对的是 JSON 格式的文件，load()方法则是一个通用的加载文件数据的方法，所以需要使用 format()方法设定文件类型，其支持 CSV、JSON、Parquet 等常见的数据文件格式，只需在 format()方法的括号里面填入"csv""json""parquet"等这样的字符串名称即可。

### 2. 通过 Parquet 文件创建 DataFrame

Parquet 是 Hadoop 生态圈的一种新型列式存储格式，兼容常见的大数据计算框架（如 Hadoop、Spark 等），与平台和编程语言无关，这也使得它的适用范围很广，只要相关语言有支持的类库就可以使用它。Parquet 目前是 Spark SQL 默认的存储格式，不过要注意的是，Parquet 文件不能直接使用 cat 或 vi 之类的文本工具查阅，否则会被识别为乱码形式，只有被 Spark 加载到应用程序以后才能出现正常的数据内容。

Spark 软件包中提供了一个 Parquet 的样例数据，其放置在 examples/src/main/resources 目录中。下面通过一个简单的实例演示如何加载 Parquet 文件，这里假定 Spark 软件包存放在/usr/local/spark 目录中，如果不是这个目录，就需要修改一下代码中的目录位置。在 SparkShell 交互式编程环境中输入下面的代码。

```
scala>
scala> val  df = spark.read.load(
    |   "file:///usr/local/spark/examples/src/main/resources/users.parquet")
df: org.apache.spark.sql.DataFrame = [name: string, favorite_color: string ...
scala> df.show
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          NULL|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
scala>
```

加载 users.parquet 文件并将其转换为 DataFrame

由此可见，Spark SQL 在加载默认格式的 Parquet 文件时直接调用 load()方法即可，无须通过 format()方法设置文件类型。

## 3.5.2　保存 DataFrame

DataFrame 的数据可以根据需要保存到 CSV、JSON、Parquet 等格式的文件中，Spark 官方推荐的是 Parquet 格式。在将 DataFrame 的数据保存到文件中时，还可以附带各种选项，比如是否保存 CSV 文件的标题行等。

在 SparkShell 交互式编程环境中输入下面的代码。

```
scala>
scala> val  data = List(
    |        (11, "LingLing", 19, "Hangzhou"),
    |        (22, "MeiMei", 22, "Shanghai"),
    |        (33, "Sansan", 23, "Nanjing") )
```

测试数据，创建包含 3 条记录的 DataFrame

```
data: List[(Int, String, Int, String)] = List((11,LingLing,19,Hangzhou),
(22,MeiMei,22,Shanghai), (33,Sansan,23,Nanjing))
scala> val  df = spark.createDataFrame(data).
     |       toDF("id","name","age","address")
df: org.apache.spark.sql.DataFrame = [id: int, name: string ... 2 more fields]
scala> df.write.parquet("file:///home/spark/stu_parquet/")
scala> df.write.option("header", true).
     |      csv("file:///home/spark/stu_csv1/")
scala> df.write.option("header", true).
     |      format("csv").
     |      save("file:///home/spark/stu_csv2/")
scala> df.write.json("file:///home/spark/stu_json/")
scala>
```

保存为 Parquet 格式的文件

分别用两种方法保存为 CSV 格式的文件，同时保留标题行。保存 DataFrame 时需要指定保存的目录路径而不是文件名，如果保存数据的目录事先存在，则应先将其删除

保存为 JSON 格式的文件

与读取文件类似，Spark SQL 提供了不同格式数据的保存方法，如 parquet()、csv()、json()等，还可以使用通用的 save()方法进行保存，只需指定保存的文件类型即可。当上面的代码执行完毕后，我们会发现主目录中存在 4 个以 stu 开头的目录，在 Linux 终端窗体中进入任意一个目录，即可查看具体保存的数据内容（注意：不是在 SparkShell 交互式编程环境中）。

```
spark@ubuntu-vm:~$
spark@ubuntu-vm:~$ ll  -d  stu*
drwxr-xr-x 2 spark spark 4096  2月 28 18:47 stu_csv1/
drwxr-xr-x 2 spark spark 4096  2月 28 18:47 stu_csv2/
drwxr-xr-x 2 spark spark 4096  2月 28 18:47 stu_json/
drwxr-xr-x 2 spark spark 4096  2月 28 18:47 stu_parquet/
spark@ubuntu-vm:~$ cd  stu_csv1
spark@ubuntu-vm:~/stu_csv1$ ls
part-00000-73518b52-208a-4b79-ac7a-72860af19c89-c000.csv  _SUCCESS
part-00001-73518b52-208a-4b79-ac7a-72860af19c89-c000.csv
spark@ubuntu-vm:~/stu_csv1$ cat  part-*
id,name,age,address
11,LingLing,19,Hangzhou
id,name,age,address
22,MeiMei,22,Shanghai
33,Sansan,23,Nanjing
spark@ubuntu-vm:~/stu_csv1$
```

确保在当前主目录中，列出以 stu 开头的目录，-d 代表仅列出目录名

进入 stu_csv1 目录并列出其中的文件

查看所有以 part-开头的文件内容

这里有两个 CPU 核，DataFrame 对应两个分区，所以显示的是两个文件的内容

## 3.6 RDD/DataFrame/Dataset 类型转换

Spark 的 RDD、DataFrame、Dataset 这几种关键的数据结构，是分布式数据处理领域非常出色的抽象设计，因此有必要对它们进行一些梳理，这也有助于读者进一步加深对 Spark 内部工作机制的理解。