# 第 9 章 Spark SQL 结构化数据处理

## ★本章导读★

在前面章节介绍了 Spark Core 的核心部分 RDD。Spark 数据处理方式除了 RDD 之外，还有 DataFrame 和 Spark SQL。本章将对 DataFrame 和 Spark SQL 数据处理进行详细介绍，并且结合实战，对数据挖掘常涉及到的增删改查等数据处理操作进行展示。通过对 DataFrame 数据处理的学习可以为后续 Spark 机器学习奠定基础。

## ★知识要点★

通过本章内容的学习，读者将掌握以下知识：

- DataFrame、Spark SQL 的概念和特点
- DataFrame、Spark SQL 常用数据的处理方法
- DataFrame 与 RDD 之间的转换处理

# 9.1 Spark SQL 与 DataFrame

### 9.1.1 什么是 Spark SQL

Spark SQL 是 Spark 用于结构化数据（Structured Data）处理的 Spark 模块。与基本的 Spark RDD API 不同, Spark SQL 的抽象数据类型为 Spark 提供了关于数据结构和正在执行的计算的更多信息。在内部，Spark SQL 使用这些额外的信息去做一些额外的优化。

有多种方式与 Spark SQL 进行交互，比如：SQL 和 Dataset API。当计算结果的时候，使用的是相同的执行引擎，不依赖我们使用哪种 API 或者语言。这种统一也就意味着开发者可以很容易在不同的 API 之间进行切换，这些 API 提供了最自然的方式来表达给定的转换。

Spark SQL 提供了 2 个编程抽象，类似 Spark Core 中的 RDD，分别是：DataFrame、DataSet。

### 9.1.2 Spark SQL 的特点

Spark SQL 具有如下特点。

1. Integrated()

批注 [雷1]: 需翻译成中文

Spark SQL 具有易整合的特点，可以无缝的整合 SQL 查询和 Spark 编程，如下图 9-1 所示。

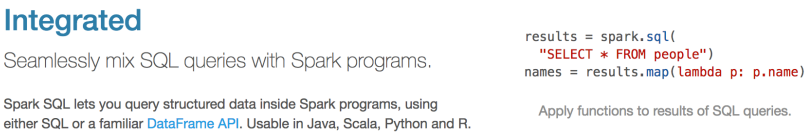


图 9-1 Spark SQL 易整合特点官网介绍

批注 [雷2]: 添加图注说明

2. Uniform Data Access

Spark SQL 具有统一的数据访问方式，可以使用相同的方式连接不同的数据源，如下图 9-2 所示。

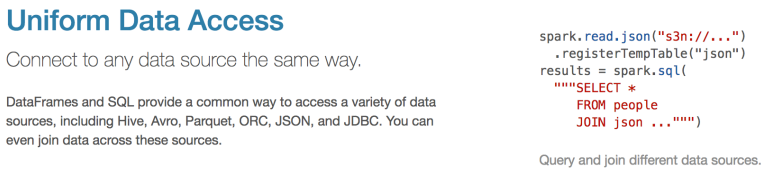



图 9-2 Spark SQL 统一数据访问特点官网介绍

3. Hive Integration

不同于 Spark 1.x 版本，在 Spark 2.x 版本中，已经集成了 Hive，在已有的仓库上可以直接运行 SQL 或者 HiveQL，如下图 9-3 所示。

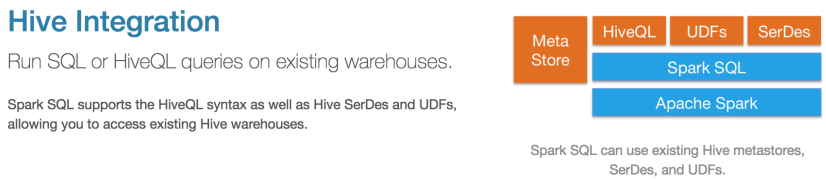



图 9-3 Spark SQL 集成 Hive 特点官网介绍

4. Standard Connectivity(标准的连接方式)

通过 JDBC 或者 ODBC 来连接数据库等其他数据工具，如下图 9-4 所示。

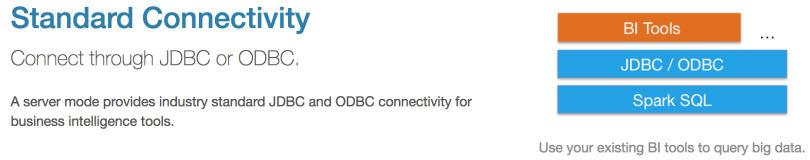



图 9-4 Spark SQL 标准连接方式官网介绍

### 9.1.3 什么是 DataFrame

关于 DataFrame 我们同样可以在 Spark 官网中得到详细定义。与 RDD 类似，DataFrame 也是一个分布式数据容器。然而 DataFrame 更像传统数据库的二维表格，除了数据以外，还记录数据的结构信息，即 schema。同时，DataFrame 支持嵌套数据类型，struct、array 和 map。从 API 易用性的角度上看，DataFrame API 提供的是一套高层的关系操作，比函数式的 RDD API 要更加友好，门槛更低。

批注 [雷3]: 删除该图片，图片内容需表述在正文中

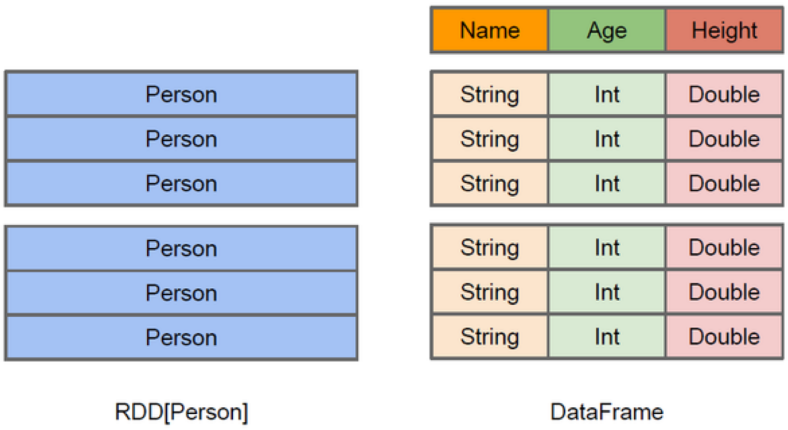



图 9-5 RDD 与 DataFrame 结构对比

如上图 9-5 所示，直观地体现了 DataFrame 和 RDD 的区别。左侧的 RDD[Person]虽然以 Person 为类型参数，但 Spark 框架本身不了解 Person 类的内部结构。而右侧的 DataFrame 却提供了详细的结构信息，使得 Spark SQL 可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。

DataFrame 是为数据提供了 Schema 的视图。可以把它当做数据库中的一张表来对待，DataFrame 也是懒执行的性能上比 RDD 要高，主要原因是优化的执行计划，查询计划可通过 Spark catalyst optimiser 进行优化。

简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程，如下图 9-6 所示。

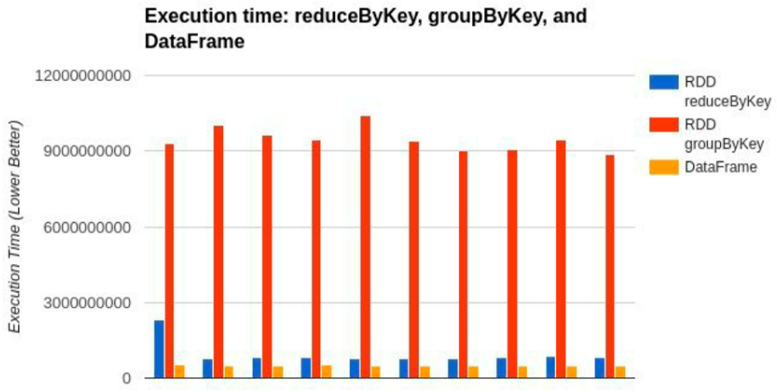



图 9-6 RDD 与 DataFrame 执行时间对比

### 9.1.4 什么是 DataSet

DataSet 是 DataFrame API 的一个扩展，是 SparkSQL 最新的数据抽象(1.6 新增)。DataSet 具有用户友好的 API 风格，既具有类型安全检查也具有 DataFrame 的查询优化特性。Dataset 支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。在 DataSet 中，可以使用样例类来

定义数据的结构信息，样例类中每个属性的名称直接映射到 DataSet 中的字段名称。

前面小节讲到的 DataFrame 是 DataSet 的特列，DataFrame=DataSet[Row]，所以可以通过 as 方法将 DataFrame 转换为 DataSet。Row 是一个类型，跟 Car、Person 这些的类型一样，所有的表结构信息都用 Row 来表示。DataFrame 只是知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没办法在编译的时候检查是否类型失败的，比如用户可以对一个 String 进行减法操作，在执行的时候才报错，而 DataSet 不仅仅知道字段，而且知道字段类型，所以有更严格的错误检查。就跟 JSON 对象和类对象之间的类比。

# 9.2 DataFrame 快速入门

### 9.2.1 创建 SparkSession

本节将进行 Spark DataFrame 的实战讲解，首先我们需要启动集群和 Pyspark Notebook，然后为本章节创建 Notebook 文件，同样，本章节启动的 Pyspark Notebook 为 local 模式。

步骤 1：启动 Hadoop 集群，代码如下所示。

```
cd  /usr/local/src/ hadoop-2.6.5/sbin
./start-all.sh
```

步骤 2：启动 Pyspark Notebook，代码如下所示。

```
PYSPARK_DRIVER_PYTHON=ipython PYSPARK_DRIVER_PYTHON_OPTS="notebook" pyspark
```

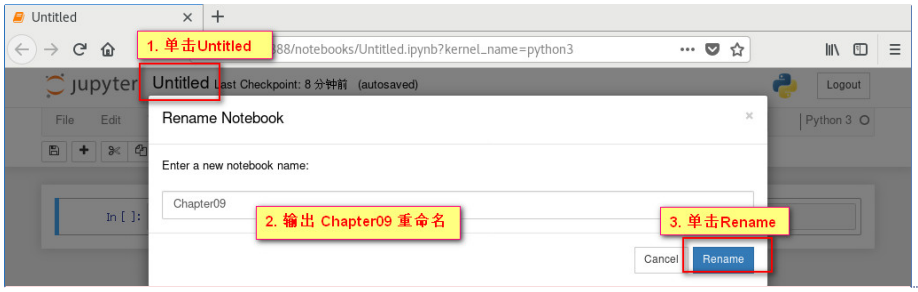
步骤 3：新建 Notebook 文件

在启动并打开 Jupyter 之后，点击【New】创建 Notebook 文件，结果如图 9-7 所示。


图 9-7 创建 Notebook 文件

步骤 4：重命名 Notebook

接下来，为 Notebook 文件进行重命名，结果如图 9-8 所示。




批注 [雷4]: 图片内容需要正文中描述出来

批注 [HZ5R4]: 这个具体操作方法在前面章节中已经一步一步介绍了，这里只是重复性的简单单击、修改名字的操作，所以只在图例中重点标注了一下。

图 9-8 重命名 Notebook

步骤 4：创建 SparkSession

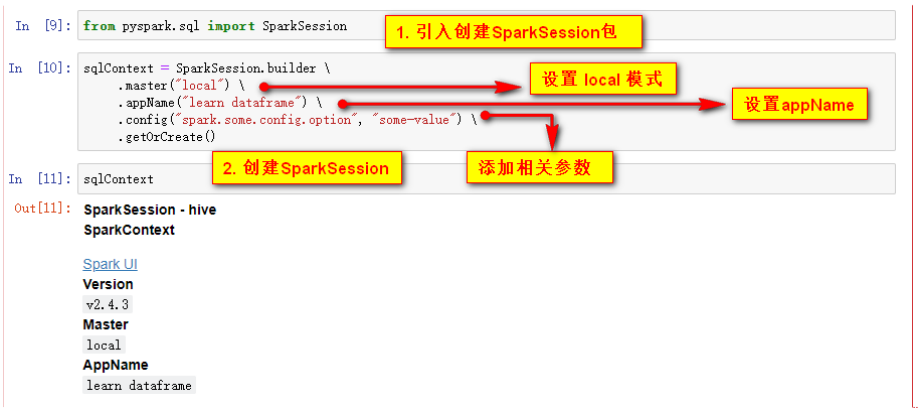
本章节是对 Spark DataFrame 以及 Spark SQL 的实战，实质均是对结构化数据的处理，下面我们创建本章节的 SparkSession。在 Spark 2.0 之前的版本，SparkSQL 提供两种 SQL 查询起始，而 sqlContext 是 Spark SQL 的入口，用于 Spark 自己提供的 SQL 查询；HiveContext 是 Hive 的入口，用于连接 Hive 的查询。对于 Spark 2.0 之后的版本，统一使用 SparkSession 作为入口，实质上是 SQLContext 和 HiveContext 的组合，所以在 SQLContext 和 HiveContext 上可用的 API 在 SparkSession 上同样是可以使用的。

SparkSession 内部封装了 SparkContext，所以计算实际上是由 SparkContext 完成的。

当我们使用 spark-shell 的时候, spark 会自动的创建一个叫做 spark 的 SparkSession, 就像我们以前可以自动获取到一个 sc 来表示 SparkContext，这就是我们在前面介绍 RDD 相关章节中使用到的 sc。首先我们需要从 pyspark.sql 中引入 SparkSession 的库；然后，通过 SparkSession 中的 build 方法为 SparkSession 设置参数，通过方法设置的参数主要包括：

- master()：设置 SparkSession 的运行模式，这里为“local”即本地模式
- appName()：为该 SparkSession 设置 AppName，可以理解为该任务的名字
- config()：为 SparkSession 设置其他相关参数，该方法的参数格式为(key, value)形式

具体实现代码如图 9-9 所示。




批注 [雷6]: 图片内容需要正文中描述出来

图 9-9 创建 SparkSession

### 9.2.2 创建 DataFrame

有了 SparkSession 之后，通过 SparkSession 有 3 种方式来创建 DataFrame:

1. 通过 Spark 的数据源创建；
2. 通过已知的 RDD 来创建；
3. 通过查询一个 Hive 表来创建。

Spark 可以通过多种数据源的方式创建 Spark DataFrame，Spark 支持的数据源如下图 9-10 所示。

```
>>> spark.read.
spark.read.csv(       spark.read.jdbc(      spark.read.load(      spark.read.options(   spark.read.parquet(   spark.read.table(
spark.read.format(    spark.read.json(      spark.read.option(    spark.read.orc(       spark.read.schema(    spark.read.text(
```

图 9-10 Spark 可读取的数据源类型

PySpark 读取数据使用的函数为 read 方法，下面我们通过读取不同格式数据源来实现 DataFrame 的创建工作。本小节读取的文件均为本地文件，通过替换路径可以实现读取 HDFS 文件，读者可以自行尝试。

步骤 1：获取本地路径

本节代码涉及的数据文件存储在 /data/resources 目录下，如下图 9-11 所示。

```
In [1]: pwd = !pwd
        data_path = "file://" + list(pwd)[0] + '/data/resources'
        print(data_path)

        file:///root/pyspark-book/data/resources
```

批注 [雷7]: 图片内容需要正文中描述出来，下面相同问题同样处理

图 9-11 获取本地工程路径

步骤 2：读取文件

（1） 读取 json 文件

json 文件是我们经常使用的一种 Key-Value 格式的文件，读取 json 文件创建 DataFrame，这里我们读取了步骤 1 中数据目录下的 people.json 文件，具体代码如下图 9-12 所示。

```
In [2]: spark.read.json(f"{data_path}/people.json")
Out[2]: DataFrame[age: bigint, family: bigint, mobile: string, name: string, score: array<bigint>]
```

图 9-12 Spark 读取 json 文件

（2） 读取 parquet 文件

Parquet 文件是一种压缩数据格式文件，这里我们读取了步骤 1 中数据目录下的 people.parquet 文件，如下图 9-13 所示。

```
In [7]: spark.read.parquet(f"{data_path}/people.parquet")
Out[7]: DataFrame[name: string, favorite_color: string, favorite_numbers: array<int>]
```

图 9-13 Spark 读取 parquet 文件

（3） 读取 csv 文件

这里我们读取了步骤 1 中数据目录下的 people.csv 文件，如下图 9-14 所示。

```
In [8]: spark.read.csv(f"{data_path}/people.csv", header=True)
Out[8]: DataFrame[ratings: string, age: string, experience: string, family: string, mobile: string]
```

图 9-14 Spark 读取 csv 文件参数

与读取其他文件格式不同的是，读取 csv 文件通常需要通过一些参数指定读取的 csv 文件，常用的参数如下图 9-15 所示。

```
Signature: spark.read.csv(path, schema=None, sep=None, encoding=None, quote=None, escape=None, comment=None, header=None, inferSchem
a=None, ignoreLeadingWhiteSpace=None, ignoreTrailingWhiteSpace=None, nullValue=None, nanValue=None, positiveInf=None, negativeInf=No
ne, dateFormat=None, timestampFormat=None, maxColumns=None, maxCharsPerColumn=None, maxMalformedLogPerPartition=None, mode=None, col
umnNameOfCorruptRecord=None, multiLine=None, charToEscapeQuoteEscaping=None, samplingRatio=None, enforceSchema=None, emptyValue=Non
e)
Docstring:
Loads a CSV file and returns the result as a  :class:`DataFrame`.
```

图 9-15 Spark 读取 csv 文件参数

如上图所示，schema 指的是为数据指定字段类型，sep 是读取数据字段直接分隔符，header 是指是否读取首行（列名），inferSchema 是指是否根据输入数据自动设置字段类型。

（4） 读取 orc 文件

orc 数据格式是优化后的列式记录，是一种二进制的文件格式，读取 people.orc 文件代码如图 9-16 所示。

```
In [9]: spark.read.orc(f"{data_path}/people.orc")
Out[9]: DataFrame[name: string, favorite_color: string, favorite_numbers: array<int>]
```

图 9-16 Spark 读取 orc 文件参数

（5） 读取 txt 文件

使用 read 方法读取数据文件创建的数据格式均为 DataFrame 格式，例如读取 csv 文件，如下图所示。实现代码如下图 9-17 所示。

```
In [10]: spark.read.text(f"{data_path}/people.txt")
Out[10]: DataFrame[value: string]
```

图 9-17 读取 csv 文件

### 9.2.3 展示 DataFrame 初体验

在上一小节中，我们通过读取不同格式的数据文件创建 Spark DataFrame，本小节将对 Spark DataFrame 从数据内容和结构上进行展示，使读者对 Spark DataFrame 有更为直观的认识。我们以读取 json 文件创建 Spark DataFrame 为例进行介绍。

批注 [雷8]: 语意不完整，需审核后修改

步骤 1：创建 DataFrame

读取代码如下。

```
df = spark.read.json(f"{data_path}/people.json")
```

步骤 2：查看 DataFrame 数据结构

通过 printSchema()函数可以查看 DataFrame 都有哪些字段以及每个字段的类型，代码如下图 9-18 所示。

```
In [12]: df.printSchema()
root
 |— age: long (nullable = true)
 |— family: long (nullable = true)
 |— mobile: string (nullable = true)
 |— name: string (nullable = true)
 |— score: array (nullable = true)
 |    |— element: long (containsNull = true)
```

图 9-18 查看 DataFrame 数据结构

步骤 3：查看 DataFrame 数据内容

不同于 RDD 的查看方式，DataFrame 使用 show()函数方法打印数据，通过参数指定要查看的 DataFrame 行数，默认是 20 行，这里我们只打印查看前 5 行，代码如下图 9-19 所示。


图 9-19 查看 DataFrame 内容

步骤 3：查看 DataFrame 分区

同样，我们可以查看 DataFrame 的分区情况，这里需要通过.rdd 方法将 DataFrame 转为 RDD 进行查看，更为详细的 DataFrame 与 RDD 之间的转换将在后续小节中进行介绍如下图 9-20 所示。

```
In [14]: df.rdd.getNumPartitions()
Out[14]: 1
```

图 9-20 查看 DataFrame 分区数

步骤 4：查看 DataFrame 类型

类似于 printSchema()方法，使用.dtypes 方法可以查看 DataFrame 每个字段的类型，输出结果为 List 类型，每个元素为 Tuple 类型，分别代表字段名和字段类型，如下图 9-21 所示。

```
In [15]: df.dtypes
Out[15]: [('age', 'bigint'),
          ('family', 'bigint'),
          ('mobile', 'string'),
          ('name', 'string'),
          ('score', 'array<bigint>')]
```

数字array类型，数组中的元素为bigint类型

图 9-21 dtypes 查看 DataFrame 字段类型

步骤 5：查看 DdataFrame 基本统计信息

这里和 RDD 的方法非常相似，均通过 describe()方法来查看 DataFrame 的基本统计情况，包括 count、mean、max 等统计量，代码如下图 9-22 所示。

```
In [17]: df.describe().show()
+-------+------------------+------------------+------+----+
|summary|               age|            family|mobile|name|
+-------+------------------+------------------+------+----+
|  count|                 7|                 8|     8|   8|
|   mean|22.571428571428573|             5.125|  null|null|
| stddev| 6.924765768773768|1.9594095320493148|  null|null|
|    min|                14|                 3|   TCL|Andy|
|    max|                31|                 9|  vivo|sira|
+-------+------------------+------------------+------+----+
```

图 9-22 DdataFrame 基本统计信息

### 9.2.4 使用 DataFrame 查询数据

上一小节中，我们对 DataFrame 的展示有了初步体验，在本小节中将进一步对 DataFrame 数据查询方式进行深入实战。

当我们读取的数据集较大，包含的列较多，而我们只想对部分字段进行打印查看是否正确，这时我们可以使用 select 方法进行数据查询。

批注 [雷9]: 后面列举的不同方法，需要整体说明介绍。

（1） select 方法查询部分字段

使用 select 方法可以读取 DataFrame 部分字段，可以通过传入所需读取列的列表，也可以直接使用“DataFrame 名称.列名”的方法进行读取，如下图 9-23、9-24 所示。

```
In [18]: df.select("age", "mobile").show(5)
+---+------+
|age|mobile|
+---+------+
| 31|   ios|
| 30| ios15|
| 21| nokia|
| 14|  vivo|
| 14|  vivo|
+---+------+
only showing top 5 rows
```

图 9-23 列名列表方式读取

```
In [20]: df.select(df.age, df.mobile).show(5)
+---+------+
|age|mobile|
+---+------+
| 31|   ios|
| 30| ios15|
| 21| nokia|
| 14|  vivo|
| 14|  vivo|
+---+------+
only showing top 5 rows
```

图 9-24 “DataFrame 名称.列名”方式读取

（2） limit 方法限制读取行数

除了通过向 show()传入读取行数的方法之外，我们还可以通过 limit()方法限制读取的行数，方法同 show()方法相同，如下图 9-25 所示。

```
In [19]: df.select("age", "mobile").limit(2).show()
+---+------+
|age|mobile|
+---+------+
| 31|   ios|
| 30| ios15|
+---+------+
```

图 9-25 limit 方法限制读取

（3） selectExpr 方法运算后读取

使用 selectExpr 方法可以在 select 的基础上对列进行运算后读取输出，例如我们在读取的时候希望计算 mobile 的长度，如下图 9-26 所示。

```
In [22]: df.selectExpr("name", "length(mobile)").show()
+-------+----------------+
|   name|length(mobile)|
+-------+----------------+
|Michael|               3|
|   Andy|               5|
| Justin|               5|
|  Berta|               4|
|  Berta|               4|
|   Bela|               5|
|   sira|               3|
|   John|               2|
+-------+----------------+
```

图 9-26 selectExpr 方法运算后读取

（4） alias 方法为列设置别名

在查询数据集的时候，有时存在某些字段/列名太长，如上一代码中使用 selectExpr()函数计算 mobile 列的长度，我们希望通过为该列设置别名的方式进行读取查看，这时可以使用 alias()函数，该函数的参数即为要设置的别名，如下图 9-27 所示。

```
In [21]: df.select(df.age, df.mobile.alias("mo")).show(5)
+---+-----+
|age|   mo|
+---+-----+
| 31|  ios|
| 30|ios15|
| 21|nokia|
| 14| vivo|
| 14| vivo|
+---+-----+
only showing top 5 rows
```

图 9-27 alias 方法为列设置别名

（5） filter/where 方法过滤数据读取

使用 limit()或 show()方法只能从行数上限制读取的 DataFrame，当我们需要从数据的角度进行限制过滤，这时就需要使用 filter/where 方法，filter 和 where 方法使用完全一致，如下图 9-28 所示。

```
In [23]: df.filter("age<28").show()
+---+------+------+------+---------+
|age|family|mobile|  name|    score|
+---+------+------+------+---------+
| 21|     3| nokia|Justin|[3, 5, 6]|
| 14|     6|  vivo| Berta|[2, 3, 6]|
| 14|     6|  vivo| Berta|[2, 3, 6]|
| 26|     3| honor|  Bela|[4, 3, 6]|
| 22|     5|   TCL|  sira|[3, 5, 5]|
+---+------+------+------+---------+
```

图 9-28 filter 过滤年龄小于 28 的数据

```
In [24]: df.where("age<28").show()
+---+------+------+------+---------+
|age|family|mobile|  name|    score|
+---+------+------+------+---------+
| 21|     3| nokia|Justin|[3, 5, 6]|
| 14|     6|  vivo| Berta|[2, 3, 6]|
| 14|     6|  vivo| Berta|[2, 3, 6]|
| 26|     3| honor|  Bela|[4, 3, 6]|
| 22|     5|   TCL|  sira|[3, 5, 5]|
+---+------+------+------+---------+
```

图 9-29 where 过滤年龄小于 28 的数据

当然我们可以使用类似于 SQL 语句的方式过滤数据中的空值或进行匹配过滤，如下图 9-30 所示。

```
In [25]: df.filter("age is not null").show()
```

```
+---+------+------+-------+----------+
|age|family|mobile|   name|     score|
+---+------+------+-------+----------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 26|     3| honor|   Bela|[4, 3, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
+---+------+------+-------+----------+
```

图 9-30 过滤掉 null 值数据

```
In [27]: df.filter("name like '%Be%'").show()
```

```
+---+------+------+-----+----------+
|age|family|mobile| name|     score|
+---+------+------+-----+----------+
| 14|     6|  vivo|Berta|[2, 3, 6]|
| 14|     6|  vivo|Berta|[2, 3, 6]|
| 26|     3| honor| Bela|[4, 3, 6]|
+---+------+------+-----+----------+
```

图 9-31 like 匹配过滤数据

上图代码表示过滤 name 列中间存在“Be”的数据。

除了使用 SQL 语句的方式之外，我们还可以使用函数算子的方法进行过滤，如下图所示，如下图 9-32 所示。

```
In [26]: df.filter(df["family"].between(4,6)).show()
```

```
+---+------+------+-------+----------+
|age|family|mobile|   name|     score|
+---+------+------+-------+----------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
+---+------+------+-------+----------+
```

图 9-32 过滤 family 值在 4 和 6 之间的数据

（6） sample 方法采样读取

sample 采用读取的使用类似于 RDD 中的 sample，代码如下图 9-33 所示。

```
In [28]: df.sample(withReplacement=False, fraction=0.5, seed=6).show()
```

```
+---+------+------+-------+----------+
|age|family|mobile|   name|     score|
+---+------+------+-------+----------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
+---+------+------+-------+----------+
```

图 9-33 sample 采样读取

### 9.2.5 使用 DataFrame 增加数据

本小节开始将进入关于 DataFrame 真正的数据处理部分，Spark 为开发者提供了自带的函数以提高开发效率，Spark 自带的 DataFrame 相关函数库在下面代码所示的类中。

```
pyspark.sql.functions
```

在开始 DataFrame 数据处理之前， 我们需要引入该函数库，并为该函数库设置别名，代码如下所示。

```
import pyspark.sql.functions as F
```

（1） 使用 select 为 DataFrame 增加一列

如前面小节所介绍，select、selectExpr 均可以为 DataFrame 新增列，并进行简单的运算，例如我们根据 age 列来计算用户的出生年月，那么我们需要按行进行 2022-age 值的计算，Spark 可以方便的直接对一列值进行相同的计算，即直接使用 2022 减去 age 列，在 DataFrame 中，通过 df.age 可以表示 age 列，如下图 9-34 所示。

```
In [31]: df.select("name", "age", "mobile", (2022-df.age)).show(5)
+-------+---+------+-------------+
|   name|age|mobile|(2022 - age)|
+-------+---+------+-------------+
|Michael| 31|   ios|         1991|
|   Andy| 30| ios15|         1992|
| Justin| 21| nokia|         2001|
|  Berta| 14|  vivo|         2008|
|  Berta| 14|  vivo|         2008|
+-------+---+------+-------------+
only showing top 5 rows
```

对 age 列每个元素执行与2022的减法运算

图 9-34 使用 select 方法新增一列

（2） 使用 withColumn 为 DataFrame 增加一列

在实际开发中，withColumn 是最常使用的为 DataFrame 新增一列的方法。withColumn 有两个参数，第一个参数为所要新增列的列名，第二个参数为新增一列逻辑，可以使用简单的列计算，也可以传入复杂的 UDF 函数，关于 UDF 函数将在后续小节中将详细介绍，如下图 9-35 所示。

```
In [32]: df.withColumn("birth", 2021-df.age).show()
+---+------+------+-------+------------+-----+
|age|family|mobile|   name|       score|birth|
+---+------+------+-------+------------+-----+
| 31|     4|   ios|Michael| [1, 6, 7]| 1990|
| 30|     5| ios15|   Andy| [1, 2, 8]| 1991|
| 21|     3| nokia| Justin| [3, 5, 6]| 2000|
| 14|     6|  vivo|  Berta| [2, 3, 6]| 2007|
| 14|     6|  vivo|  Berta| [2, 3, 6]| 2007|
| 26|     3| honor|   Bela| [4, 3, 6]| 1995|
| 22|     5|   TCL|   sira| [3, 5, 5]| 1999|
|null|     9|    mi|   John|[2, 13, 6]| null|
+---+------+------+-------+------------+-----+
```

图 9-35 使用 withColumn 方法新增一列

（3） 使用 withColumn 为 DataFrame 增加一列相同值

上述两种为 DataFrame 新增一列的共同点在于：均是有 DataFrame 现有列参与计算生成的。那么，如果我们需要为 DataFrame 增加一列固定值，应该如何实现呢？这时就可以使用到 Spark 自带的函数 lit()函数，函数的参数为所要设置的值，可以为整型、字符串等，如下图 9-36 所示。

批注 [雷10]: 后面列举的不同方法，需要整体说明介绍。

批注 [雷11]: 图片内容需在正文中表述

```
In [33]: df.withColumn("school", F.lit("high school")).show()
+---+------+------+-----+---------+-----------+
|age|family|mobile| name|    score|     school|
+---+------+------+-----+---------+-----------+
| 31|     4|   ios|Michael|[1, 6, 7]|high school|
| 30|     5| ios15|  Andy|[1, 2, 8]|high school|
| 21|     3| nokia|Justin|[3, 5, 6]|high school|
| 14|     6|  vivo| Berta|[2, 3, 6]|high school|
| 14|     6|  vivo| Berta|[2, 3, 6]|high school|
| 26|     3| honor|  Bela|[4, 3, 6]|high school|
| 22|     5|   TCL|  sira|[3, 5, 5]|high school|
|null|     9|    mi|  John|[2, 13, 6]|high school|
+---+------+------+-----+---------+-----------+
```


图 9-36 使用 withColumn 增加一列相同值

### 9.2.6 使用 DataFrame 修改数据

接下来介绍如何修改 DataFrame 数据。前面小节提到的 alias()方法就是最简单的修改 DataFrame 的应用，即修改 DataFrame 的列名。除了使用 alias()方法之外，我们还可以使用 withColumnRenamed()方法对列进行重命名，使用 DataFrame 修改数据详情如下所示。

（1） 使用 withColumnRenamed 修改列名

withColumnRenamed()函数有两个参数，第一个参数为要修改的列原名，第二个参数为要修改的新列名，如下图 9-37 所示。

```
In [40]: df.withColumnRenamed("mobile", "phone").show()
+---+------+-----+-----+---------+
|age|family|phone| name|    score|
+---+------+-----+-----+---------+
| 31|     4|  ios|Michael|[1, 6, 7]|
| 30|     5|ios15|  Andy|[1, 2, 8]|
| 21|     3|nokia|Justin|[3, 5, 6]|
| 14|     6| vivo| Berta|[2, 3, 6]|
| 14|     6| vivo| Berta|[2, 3, 6]|
| 26|     3|honor|  Bela|[4, 3, 6]|
| 22|     5|  TCL|  sira|[3, 5, 5]|
|null|     9|   mi|  John|[2, 13, 6]|
+---+------+-----+-----+---------+
```

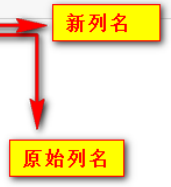



图 9-37 使用 withColumnRenamed 修改列名字

那么除了可以修改列名之外，我们还可以修改列的字段类型。

（2） 使用 cast 修改字段类型

对列修改类型，我们可以使用“DataFrame 名称.列名.cast(字段类型)”的方式，也可以使用“DataFrame 名称[列名].列名.cast(字段类型)”的方式进行修改，如下图 9-38 所示。

```
In [36]: df.select(df.age.cast("float").alias("age")).show(5)
+----+
| age|
+----+
|31.0|
|30.0|
|21.0|
|14.0|
|14.0|
+----+
only showing top 5 rows
```

使用 DataFrame.列名

使用 DataFrame["列名"]

```
In [37]: df.withColumn("age", df["age"].cast("float")).show(5)
+----+------+------+--------+----------+
| age|family|mobile|    name|     score|
+----+------+------+--------+----------+
|31.0|     4|   ios|Michael|[1, 6, 7]|
|30.0|     5| ios15|   Andy|[1, 2, 8]|
|21.0|     3| nokia| Justin|[3, 5, 6]|
|14.0|     6|  vivo|  Berta|[2, 3, 6]|
|14.0|     6|  vivo|  Berta|[2, 3, 6]|
+----+------+------+--------+----------+
only showing top 5 rows
```

图 9-38 使用 cast 修改字段类型

在实际数据中，经常是存在空值或者 null 值，通常的做法是对这些数据进行过滤，但是有些业务场景，需要保留这些数据，那么就需要对这些空值或者 null 值赋予指定的值，便于后续的数据分析处理。例如我们对 DataFrame 中所有空值/null 值修改为固定值 3。

（3） 修改 null 值为固定值

这里使用的方法是 na.fill()，其中 fill()方法的参数代表要将数据中的空值/null 值设置的值，如下图 9-39 所示。

批注 [雷12]: 需对图片内容进行详细说明

```
In [34]: df.na.fill(3).show()
+---+------+------+--------+----------+
|age|family|mobile|    name|     score|
+---+------+------+--------+----------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 26|     3| honor|   Bela|[4, 3, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
|  3|     9|    mi|   John|[2, 13, 6]|
+---+------+------+--------+----------+
```

图 9-39 修改 null 值为固定值

（4） 使用 lit()方法修改某列值

同样，使用 withColumn()方法和 lit()方法可以将某列所有值进行统一修改，在前面小节中我们已经简单使用过 F.lit()函数，F.lit()函数中参数可以是整型、浮点型、字符串等格式数据，可以简单理解为创建 DataFrame 中的“常量”。本例代码如下图 9-40 所示。

```
In [39]: df.withColumn("mobile", F.lit("ios-15")).show()
```

```
+---+------+------+-------+------------+
|age|family|mobile|   name|       score|
+---+------+------+-------+------------+
| 31|     4|ios-15|Michael| [1, 6, 7]|
| 30|     5|ios-15|   Andy| [1, 2, 8]|
| 21|     3|ios-15| Justin| [3, 5, 6]|
| 14|     6|ios-15|  Berta| [2, 3, 6]|
| 14|     6|ios-15|  Berta| [2, 3, 6]|
| 26|     3|ios-15|   Bela| [4, 3, 6]|
| 22|     5|ios-15|   sira| [3, 5, 5]|
|null|     9|ios-15|   John|[2, 13, 6]|
+---+------+------+-------+------------+
```

图 9-40 修改 DataFrame 某列所有值

### 9.2.7 使用 DataFrame 筛选数据

本小节介绍 DataFrame 删除数据的处理方法。通常删除数据包括筛选某列数据、筛选非空值/null 值以及删除重复元素处理等，详情如下所示。

（1） 筛选非空值/null 值

除了使用前面小节介绍的 filter 方法去除空值/null 值之外，针对空值/null 值可以使用 DataFrame 自带的函数方法进行处理，代码如下图 9-41 所示。

```
In [41]: df.na.drop().show()
```

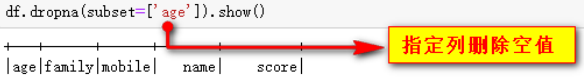
```
+---+------+------+-------+------------+
|age|family|mobile|   name|       score|
+---+------+------+-------+------------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 26|     3| honor|   Bela|[4, 3, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
+---+------+------+-------+------------+
```

图 9-41 筛选非 DataFrame 空值/null 值

（2） 筛选指定列非空值/null 值

使用.na.drop()方法可以将整个 DataFrame 中的空值/null 值进行删除，但是在某些场景中，我们需要对指定列进行删除，这时可以使用 dropna()方法，其参数为要删除空值/null 值的列名，代码如下图 9-42 所示。

```
In [42]: df.dropna(subset=['age']).show()
```




```
+---+------+------+-------+------------+
|age|family|mobile|   name|       score|
+---+------+------+-------+------------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 26|     3| honor|   Bela|[4, 3, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
+---+------+------+-------+------------+
```

图 9-42 筛选指定列非空值/null 值

（3） DataFrame 去重

下面我们介绍对 DataFrame 去重的操作，可是使用 dropDuplicates()方法进行去重，该方法参数为空时是对整个 DataFrame 进行去重，当我们需要按某列进行去重时，可以通过参数传入列名进行去重，值得注意的是，dropDuplicates()可以传入 List，即可以按指定多列进行去重，当然也可以只按某一列进行去重，如下图 9-43 所示。

```
In [7]: df.dropDuplicates(['family']).show()
```

```
+----+------+------+-------+------------+
| age|family|mobile|   name|       score|
+----+------+------+-------+------------+
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|null|     9|    mi|   John|[2, 13, 6]|
|  30|     5| ios15|   Andy| [1, 2, 8]|
|  21|     3| nokia| Justin| [3, 5, 6]|
|  31|     4|   ios|Michael| [1, 6, 7]|
+----+------+------+-------+------------+
```

图 9-43 DataFrame 去重

除了使用上面的方法之外，同样可以使用 distinc()方法进行去重，读者可以自行验证。

### 9.2.8 使用 DataFrame 合并数据

本小节介绍 DataFrame 的合并，用于多个 DataFrame 的操作，例如 join、union()等操作，详情如下所示。

（1） union 合并 DataFrame

首先我们通过 filter()方法将 df 拆分成两个 DataFrame，然后再使用 union()方法进行合并，从而可以验证数据的正确性，拆分 DataFrame 代码如下。

```
df1 = df.filter("family > 4")
df2 = df.filter("family < 5")
```

接下来，使用 union()方法将 df1 和 df2 进行合并连接。

```
In [47]: df1.union(df2).show()
```

```
+----+------+------+-------+------------+
| age|family|mobile|   name|       score|
+----+------+------+-------+------------+
|  30|     5| ios15|   Andy| [1, 2, 8]|
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|  22|     5|   TCL|   sira| [3, 5, 5]|
|null|     9|    mi|   John|[2, 13, 6]|
|  31|     4|   ios|Michael| [1, 6, 7]|
|  21|     3| nokia| Justin| [3, 5, 6]|
|  26|     3| honor|   Bela| [4, 3, 6]|
+----+------+------+-------+------------+
```

图 9-44 合并 DataFrame

（2） join 连接 DataFrame

join()方法是在数据分析中经常使用到的数据关联的方法，通过关联将两组数据进行连接，得到一个更大的数据集，下面我们使用 join()方法关联两个 DataFrame，同样我们首先将 DataFrame 拆分成两个 DataFrame，这里使用了 distinct()方法进行去重，代码如下所示。

```
df_n = df.distinct()
df3 = df_n.select("name", "age", "mobile")
df4 = df_n.select("name", "family", "score")
```

从上面代码可以看出，df1 和 df2 由不同的列组成，且两者拥有共同列，即 name 列。

接下来，使用 union()方法将 df1 和 df2 进行合并连接。

```
In [49]: df3.join(df4, df3.name == df4.name, "inner").show()
```

```
+-------+---+------+-------+------+-------------+
|   name|age|mobile|   name|family|        score|
+-------+---+------+-------+------+-------------+
| Justin| 21| nokia| Justin|     3| [3, 5, 6]|
|  Berta| 14|  vivo|  Berta|     6| [2, 3, 6]|
|   sira| 22|   TCL|   sira|     5| [3, 5, 5]|
|   Bela| 26| honor|   Bela|     3| [4, 3, 6]|
|   Andy| 30| ios15|   Andy|     5| [1, 2, 8]|
|Michael| 31|   ios|Michael|     4| [1, 6, 7]|
|   John|null|    mi|   John|     9|[2, 13, 6]|
+-------+---+------+-------+------+-------------+
```

图 9-45 join 关联 DataFrame

上述代码程序说明如表 9-1 所示

表 9-1 join 代码说明

| 程序代码 | 解释 |
|---|---|
| **df3.join(df4** | df3 和 df4 进行 join 操作 |
| **df3.name == df4.name** | join 关联条件指定 |
| **"inner"** | 设置 join 关联方式为"inner" |

当然，当两个 DataFrame 关联条件的字段名字相同时，可以使用简便的方法进行关联，代码如下图 9-46 所示。

```
In [50]: df3.join(df4, ["name"], "inner").show()
```

```
+-------+---+------+------+-------------+
|   name|age|mobile|family|        score|
+-------+---+------+------+-------------+
| Justin| 21| nokia|     3| [3, 5, 6]|
|  Berta| 14|  vivo|     6| [2, 3, 6]|
|   sira| 22|   TCL|     5| [3, 5, 5]|
|   Bela| 26| honor|     3| [4, 3, 6]|
|   Andy| 30| ios15|     5| [1, 2, 8]|
|Michael| 31|   ios|     4| [1, 6, 7]|
|   John|null|    mi|     9|[2, 13, 6]|
+-------+---+------+------+-------------+
```

图 9-46 join 关联 DataFrame

### 9.2.9 使用 DataFrame 进阶处理

本小节将介绍 DataFrame 进阶处理，对基本的数据增删改查之外的补充，例如数据分析处理中经常涉及使用的排序、分组聚合等操作，详情如下所示。

（1） DataFrame orderBy 排序

使用 orderBy()方法可以实现对 DataFrame 的排序，同样参数可以使用字符串指定列名，也可以使用“DataFrame 名称.列名”的方式。下面按 age 字段对 DataFrame 进行排序，如下图 9-47 所示。

```
In [53]: df = df.na.drop()
         df.orderBy("age").show()
```

删除空值数据

```
+---+------+------+-----+---------+
|age|family|mobile| name|    score|
+---+------+------+-----+---------+
| 14|     6|  vivo|Berta|[2, 3, 6]|
| 14|     6|  vivo|Berta|[2, 3, 6]|
| 21|     3| nokia|Justin|[3, 5, 6]|
| 22|     5|   TCL| sira|[3, 5, 5]|
| 26|     3| honor| Bela|[4, 3, 6]|
| 30|     5| ios15| Andy|[1, 2, 8]|
| 31|     4|   ios|Michael|[1, 6, 7]|
+---+------+------+-----+---------+
```

批注 [雷13]: 图片内容需要正文中描述出来

批注 [HZ14R13]: 这个在上面一段话中已经描述了，图中标的注释框只是对额外补充对无关内容加以说明，并不是例子中要讲解的内容。

图 9-47 DataFrame 排序

orderBy()方法默认是升序排序，通过 ascending 参数可以指定是否升序排序，其参数值为布尔类型，如下图 9-48 所示。

```
In [11]: df.orderBy("age", ascending=False).show()
```

降序排序

```
+---+------+------+-----+---------+
|age|family|mobile| name|    score|
+---+------+------+-----+---------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15| Andy|[1, 2, 8]|
| 26|     3| honor| Bela|[4, 3, 6]|
| 22|     5|   TCL| sira|[3, 5, 5]|
| 21|     3| nokia|Justin|[3, 5, 6]|
| 14|     6|  vivo|Berta|[2, 3, 6]|
| 14|     6|  vivo|Berta|[2, 3, 6]|
+---+------+------+-----+---------+
```

批注 [雷15]: 图片内容需要正文中描述出来

图 9-48 DataFrame 降序排序

更进一步，我们可以直接通过“DataFrame 名称.列名.desc()”的方法对指定列名进行排序方式的指定，代码如下图 9-49 所示。

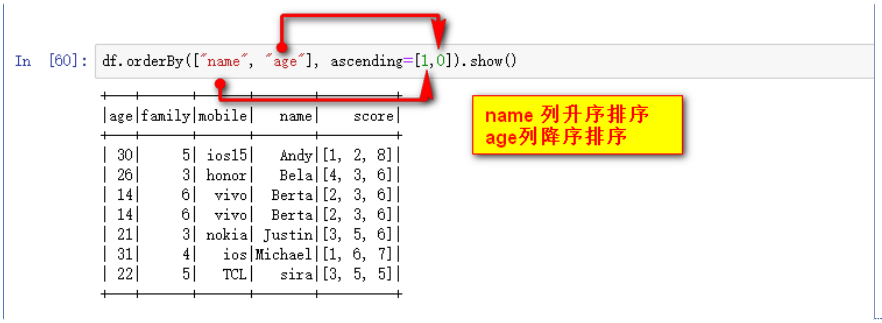
```
In [59]: df.orderBy(df.age.desc()).show()
```

指定列调用 desc()方法

```
+---+------+------+-----+---------+
|age|family|mobile| name|    score|
+---+------+------+-----+---------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15| Andy|[1, 2, 8]|
| 26|     3| honor| Bela|[4, 3, 6]|
| 22|     5|   TCL| sira|[3, 5, 5]|
| 21|     3| nokia|Justin|[3, 5, 6]|
| 14|     6|  vivo|Berta|[2, 3, 6]|
| 14|     6|  vivo|Berta|[2, 3, 6]|
+---+------+------+-----+---------+
```

图 9-49 指定 age 降序排序

这里我们只按 age 字段进行排序，当我们需要按多个字段进行排序，并且排序方式不同，应该如何实现呢？例如我们要按 name 进行升序排序，并且按照 age 进行降序排序。ascending 可以通过数组的方式传入要排序的方式，代码如下图 9-50 所示。




批注 [雷16]: 图片内容需要正文中描述出来

批注 [HZ17R16]: 在下面一段话中已经进行了解释和描述

图 9-50 DataFrame 多字段排序

如上图代码所示，ascending 参数为一个 List 列表，其中元素的索引和前面传入排序字段列表的索引相当于，即 name 列对应的 ascending=1，age 列对应的 ascending=0，即为按 name 升序排序，且按 age 降序排序。

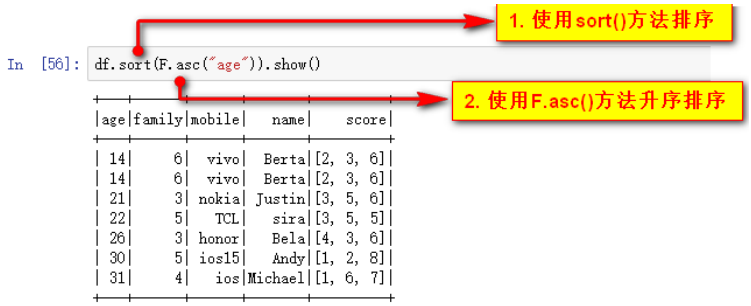
当然当只对 age 字段进行排序时同样可以使用这种传参方式，如下图 9-51 所示。

```
In [13]: df.orderBy("age", ascending=[0]).show()
+---+------+------+-------+---------+
|age|family|mobile|   name|    score|
+---+------+------+-------+---------+
| 31|     4|   ios|Michael|[1, 6, 7]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 26|     3| honor|   Bela|[4, 3, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
+---+------+------+-------+---------+
```

图 9-51 ascending 参数指定 age 字段排序

（2） DataFrame sort 排序

除了使用 orderBy()方法之外，我们还可以使用 sort()方法对 DataFrame 排序。sort()的使用方式和 orderBy 相同，这里我们使用另一种指定 age 列升序排序的方法，使用 Spark 为我们提供的升序排序函数，代码如下图 9-52 所示。




```
+---+------+------+-------+---------+
|age|family|mobile|   name|    score|
+---+------+------+-------+---------+
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|
| 21|     3| nokia| Justin|[3, 5, 6]|
| 22|     5|   TCL|   sira|[3, 5, 5]|
| 26|     3| honor|   Bela|[4, 3, 6]|
| 30|     5| ios15|   Andy|[1, 2, 8]|
| 31|     4|   ios|Michael|[1, 6, 7]|
+---+------+------+-------+---------+
```

图 9-52 sort 排序

这里我们使用了函数库中的排序方法，即 F.asc()，其参数为 DataFrame 中的列名，代表按该列进行升序排序。

（3） DataFrame groupBy 分组

在数据分析处理中，我们经常涉及分组统计的需求，对某些字段进行分组统计，类似于 RDD 的方法，DataFrame 同样使用 groupBy()方法对 DataFrame 进行分组。首先，我们对 family 列进行分组统计，代码如下图 9-53 所示。

```
In [62]: df.groupBy("family").count().show()
+------+-----+
|family|count|
+------+-----+
|     6|    2|
|     5|    2|
|     3|    2|
|     4|    1|
+------+-----+
```

图 9-53 groupBy 分组统计

上面代码直接使用 count()聚合函数对分组后的数据进行统计。Spark 还为我们提供了另一种分组统计的聚合函数，即 agg()，在 agg()函数里，我们可以指定对不同列的聚合方式，例如根据 family 进行分组，对 mobile 列进行统计数量，代码如下图 9-54 所示。

```
In [63]: df.groupBy("family").agg({"mobile":"count"}).show()
+------+-------------+
|family|count(mobile)|
+------+-------------+
|     6|            2|
|     5|            2|
|     3|            2|
|     4|            1|
+------+-------------+
```

2. 对mobile列指定count聚合方式

1. 调用agg()聚合方法

图 9-54 agg 聚合函数统计数据量

如上图代码所示，在 agg()函数里传入的可以理解为一个字典类型的参数，key 为要聚合的字段名，value 为聚合的方式，可以编写多个字段不同的聚合方式，在本例中代表对“mobile”列进行“count”统计聚合。

当然，我们也可以使用 Spark 为我们提供的函数库中的聚合函数进行统计，F.max()和 F.avg()作用分别为求最大值和平均值，参数均为 DataFrame 中存在的列名，代码如下图 9-55 所示。

```
In [64]: df.groupBy("family").agg(F.max("age"), F.avg("age")).show()
+------+--------+--------+
|family|max(age)|avg(age)|
+------+--------+--------+
|     6|      14|    14.0|
|     5|      30|    26.0|
|     3|      26|    23.5|
|     4|      31|    31.0|
+------+--------+--------+
```

图 9-55 最大值、平均值聚合统计

在实际业务中，我们常常遇到非具体数字类的聚合统计，例如我们要根据 family 分组，统计相同 family 人数的都有哪些 name 的用户，这时我们可以使用 Spark 为我们提供的 collect_list 聚合函数，代码实现如下图 9-56 所示。

```
In [65]: df.groupBy("family").agg(F.collect_list("name")).show()
```

```
+------+-----------------+
|family|collect_list(name)|
+------+-----------------+
|     6|  [Berta, Berta]|
|     5|    [Andy, sira]|
|     3|  [Justin, Bela]|
|     4|       [Michael]|
+------+-----------------+
```

图 9-56 collect_list 聚合

由上图可以看出，聚合的结果为一个数组列表，里面每个元素为 name 的值。例如对于 family 为 6 的用户（name）有 Berta, Berta，这时我们遇到了另一个问题，这两个 name 相同，有可能是一条重复数据，我们只需要不同的 name 的列表，应该如何操作呢？我们可以使用 collect_set 聚合函数，它的作用相当于对 collect_list 的结果进行了去重，代码如下图 9-57 所示。

```
In [66]: df.groupBy("family").agg(F.collect_set("name")).show()
```

```
+------+----------------+
|family|collect_set(name)|
+------+----------------+
|     6|        [Berta]|
|     5|    [Andy, sira]|
|     3|  [Bela, Justin]|
|     4|       [Michael]|
+------+----------------+
```

图 9-57 collect_set 聚合

（4） DataFrame crosstab 交叉分析

我们经常使用的 Excel 表格是一个二维的表，我们希望根据 DataFrame 数据得到一个类似于 Excel 的二维表格，可以对某两个字段进行可直观的分析，这时我们可以使用 crosstab 交叉分析，代码如下图 9-58 所示。

```
In [68]: df.stat.crosstab("name", "age").show()
```

```
+---------+---+---+---+---+---+---+
|name_age| 14| 21| 22| 26| 30| 31|
+---------+---+---+---+---+---+---+
|    Bela|  0|  0|  0|  1|  0|  0|
|    Andy|  0|  0|  0|  0|  1|  0|
| Michael|  0|  0|  0|  0|  0|  1|
|  Justin|  0|  1|  0|  0|  0|  0|
|   Berta|  2|  0|  0|  0|  0|  0|
|    sira|  0|  0|  1|  0|  0|  0|
+---------+---+---+---+---+---+---+
```

图 9-58 crosstab 交叉分析

从上图结果可以看出，我们对 DataFrame 中 age 和 name 两列进行交叉分析，生成的新 DataFrame 的每一列为 age 包含的值，每一行为 name 包含的值，例如第一行，我们可以看出，name 为 Bela 且年龄为 26 的人有 1 位，name 为 Berta 且年龄为 14 的人有 2 位。

（5） DataFrame explode 展开多行

下面我们介绍一行展开多行的方法，在 df 中 score 列为数组类型，我们可以理解为每个学生（name）语文、数学、外语三门课的成绩，我们希望能够将其展开到每一行中，即每门课的成绩单独为一行，便于其他分析处理。这时，我们需要使用 Spark 提供的 explode()函数来实现，代码如下图 9-59 所示。

```
In [72]: df.select("name", "score", F.explode("score")).show()
```

```
+--------+------------+---+
|    name|       score|col|
+--------+------------+---+
|Michael|[1, 6, 7]|  1|
|Michael|[1, 6, 7]|  6|
|Michael|[1, 6, 7]|  7|
|   Andy|[1, 2, 8]|  1|
|   Andy|[1, 2, 8]|  2|
|   Andy|[1, 2, 8]|  8|
| Justin|[3, 5, 6]|  3|
| Justin|[3, 5, 6]|  5|
| Justin|[3, 5, 6]|  6|
|  Berta|[2, 3, 6]|  2|
|  Berta|[2, 3, 6]|  3|
|  Berta|[2, 3, 6]|  6|
|  Berta|[2, 3, 6]|  2|
|  Berta|[2, 3, 6]|  3|
|  Berta|[2, 3, 6]|  6|
|   Bela|[4, 3, 6]|  4|
|   Bela|[4, 3, 6]|  3|
|   Bela|[4, 3, 6]|  6|
|   sira|[3, 5, 5]|  3|
|   sira|[3, 5, 5]|  5|
+--------+------------+---+
only showing top 20 rows
```

图 9-59 explode 展开多行

（6） to_json 与 struct 的结合使用

最后，我们介绍一种实际应用中经常涉及的构造数据方法，DataFrame 创建 json 数据。我们使用 to_json()与 struct()函数方法可以将 DataFrame 的每一行数据构造成 json 格式，首先通过 F.struct()函数将 DataFrame 每行 name 列和 age 列的数据组成一个多列组合成新 struct column；然后，再通过 F.json()将该 struct column 转换成 json 字符串或者说是 Python 字典，具体代码如下图 9-60 所示。

```
In [73]: df.withColumn(
             "jsonCol", F.to_json(F.struct([df.name, df.age]))
         ).show(truncate=False)
```

1. 将name和age列创建json字符串

2. 行内容完全显示

```
+---+------+------+--------+------------+-----------------------------+
|age|family|mobile|name    |score       |jsonCol                      |
+---+------+------+--------+------------+-----------------------------+
|31 |4     |ios   |Michael|[1, 6, 7]||{"name":"Michael","age":31}|
|30 |5     |ios15 |Andy    |[1, 2, 8]||{"name":"Andy","age":30}   |
|21 |3     |nokia |Justin |[3, 5, 6]||{"name":"Justin","age":21} |
|14 |6     |vivo  |Berta   |[2, 3, 6]||{"name":"Berta","age":14}  |
|14 |6     |vivo  |Berta   |[2, 3, 6]||{"name":"Berta","age":14}  |
|26 |3     |honor |Bela    |[4, 3, 6]||{"name":"Bela","age":26}   |
|22 |5     |TCL   |sira    |[3, 5, 5]||{"name":"sira","age":22}   |
+---+------+------+--------+------------+-----------------------------+
```

图 9-60 to_json 与 struct 的结合使用

批注 [雷18]: 图片内容需在正文中描述出来

### 9.2.10 自定义 UDF 函数

Spark 为用户提供了 UDF 函数的功能，即用户可以自定义实现函数的逻辑，应用在 DataFrame 的一列或多列上，从而生成新的一列。

UDF 函数同样位于 Spark 自带的函数库中，此外，我们还需要引入 Spark 结构化数据类型的库，代码如下所示。

```
from pyspark.sql.types import *
```

该库中包含了所有 Spark 结构化数据类型，如 StringType()、FloatType()等等。引入该库的目的在于，用户创建 UDF 函数需要指定返回值的类型。这里我们介绍对 name 进行小写字母转换并拼接原始 name 值的简单 UDF 函数的使用方法。

（1） 单列 UDF 函数使用

这里我们首先介绍一下创建 UDF 的基本步骤：

- 导入 Spark SQL 库函数和字段类型库，其中库函数导入在前面小节已经进行了介绍；
- 编写自定义 UDF 函数；
- 将编写的自定义 UDF 函数创建 Spark UDF 函数；
- 在 DataFrame 中使用所创建的 UDF 函数。

具体实现实例如下图 9-61 所示。

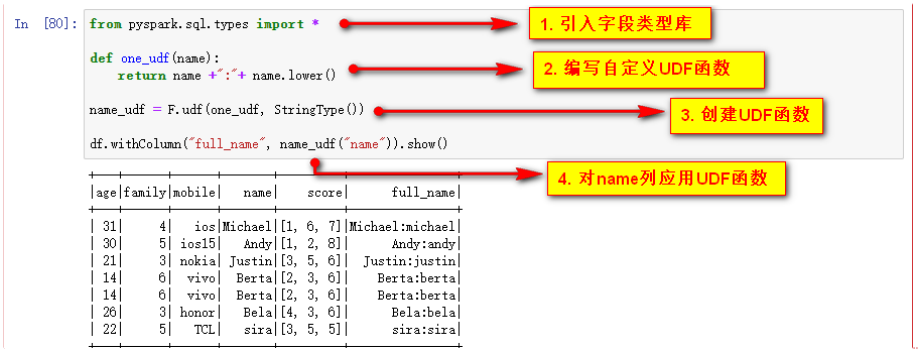



图 9-61 创建 UDF 函数

值得注意的是，F.udf()需要用户设置两个参数，第一个参数为自定义函数名，即不需要带括号和参数，第二个参数为该UDF函数返回值类型，在本例中返回值为字符串，所以第二个参数值为StringType()。

（2） 多列 UDF 函数使用

当 UDF 函数中需要传入多个字段列应该如何创建和使用呢，与创建单列 UDF 函数唯一不同之处在于自定义 UDF 函数的参数为多个，在使用 UDF 函数时传入的参数为 DataFrame 中多列的列名。本例中创建的 UDF 函数的功能为将两列数据合成一个 List 列表，具体实现代码如图 9-62 所示。

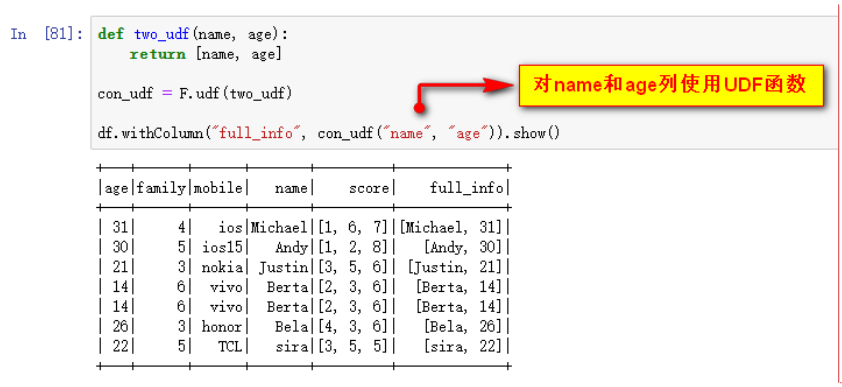



图 9-62 创建 UDF 函数

批注 [雷19]: 图片步骤内容需要在正文中描述出来

批注 [雷20]: 图片内容需在正文中描述出来

### 9.2.11 DataFrame 数据存储

DataFrame 可以存储多种格式，同样可以存储在本地文件目录或者 HDFS 文件目录中，下面分别对 DataFrame 存储 csv 和 parquet 格式文件进行介绍。

（1） 存储 csv 格式文件

在这里，存储 csv 格式文件使用的两个参数分别代表存储文件的路径和存储使用的模式 mode，本例中存储模式问“overwrite”，表示覆盖模式。

```
In [83]: df.select("name", "age").write.csv(f"{data_path}/"+"out_csv", mode="overwrite")
```


图 9-63 存储 csv 格式文件

使用如下图 9-64 命令查看存储结果。

```
In [84]: !ls -l data/resources/out_csv
总用量 4
-rw-r--r--. 1 root root 63 2月   5 20:12 part-00000-08b9d9a2-7cc7-4162-8d26-ba97f519d966-c000.csv
-rw-r--r--. 1 root root  0 2月   5 20:12 _SUCCESS
```

图 9-64 查看存储结果

注意，这里文件存储的文件名为系统自动命名的，和存储 HDFS 文件相同方式。

（2） 存储 parquet 格式文件

存储 parquet 文件代码中的参数含义和存储 csv 文件相同，具体代码如下所示。

```
df.write.parquet(f"{data_path}/"+"out_pqt", mode="overwrite")
```

使用同样的命令查看存储结果，如图 9-65 所示。

```
In [86]: !ls -l data/resources/out_pqt
总用量 4
-rw-r--r--. 1 root root 1689 2月   5 20:12 part-00000-4f4c5639-562a-4a47-b2f1-962465227a87-c000.snappy.parquet
-rw-r--r--. 1 root root    0 2月   5 20:12 _SUCCESS
```

图 9-65 查看存储结果

### 9.2.12 DataFrame 与 RDD 的交互

本小节将介绍 DataFrame 与 RDD 的交互，通过 RDD 可以创建 DataFrame，同样，DataFrame 可以转换为 RDD，进而使用 RDD 的函数方法进行数据处理。

（1） RDD 创建 DataFrame

步骤 1：创建 RDD

创建 RDD 的代码如下所示。

```
rdd = sc.parallelize([('Jones', 23), ('Tim', 18), ('Jane', 50), ('Mike',23), ('Trump', 13)])
rdd.collect()
```

步骤 2：创建 DataFrame

使用 SparkSession 中的 createDataFrame 的方法将 RDD 转为 DataFrame，代码如下图 9-66 所示。

```
In [88]: df = spark.createDataFrame(rdd)

In [89]: df.show()
+-----+---+
|   _1| _2|
+-----+---+
|Jones| 23|
|  Tim| 18|
| Jane| 50|
| Mike| 23|
|Trump| 13|
+-----+---+
```

图 9-66 创建 DataFrame

步骤 3：指定 Shema 创建 DataFrame

可以发现，创建后的 DataFrame 没有列名，即自动创建了“_1”、“_2”列名。那么当我们需要按我们指定的列名进行创建时，我们可以通过 shema 参数来指定列名，该参数为一个变长 List 列表，代表要转换后的 DataFrame 列名的列表，代码如下图 9-67 所示。

批注 [雷21]: 语意不明，需修改

```
In [90]: df = spark.createDataFrame(rdd, schema=["name", "age"])
         df.show()
+-----+---+
| name|age|
+-----+---+
|Jones| 23|
|  Tim| 18|
| Jane| 50|
| Mike| 23|
|Trump| 13|
+-----+---+
```

指定列名

图 9-67 指定 Shema 创建 DataFrame

步骤 4：指定 Shema 创建 DataFrame

当我们将 RDD 转为 DataFrame 的时候，不仅想要自定义列名，还希望 DataFrame 可以按照我们指定的字段类型进行创建时，我们需要创建带有字段类型的 schma，代码如下图 9-68 所示。

```
In [92]: schema = StructType([
             StructField('name', StringType(), True),
             StructField('age', IntegerType(), True),
         ])
         df = spark.createDataFrame(rdd, schema)
         df.show()
+-----+---+
| name|age|
+-----+---+
|Jones| 23|
|  Tim| 18|
| Jane| 50|
| Mike| 23|
|Trump| 13|
+-----+---+
```

2.指定字段类型

3.该字段是否可以有null值

1. 指定列名

图 9-68 指定 Shema 创建 DataFrame

通过上面代码可以看出，创建带有字段类型的 schema 需要定义一个 StructType()类型变量，其参数为一个 List 列表，其中每个元素为 StructField()类型，指定了每列的列名和字段类型以及该字段是否可以存在 null 值。

步骤 5：toDF 函数创建 DataFrame

除了使用 createDataFrame 之外，我们可以使用 toDF()方法直接将 RDD 转换为 DataFrame，同样，

toDF()方法可以指定转换 DataFrame 的 Schema，代码如下图 9-69 所示。

```
In [94]: df = rdd.toDF(schema=["name", "age"])
         df.show()
```

```
+-----+---+
| name|age|
+-----+---+
|Jones| 23|
|  Tim| 18|
| Jane| 50|
| Mike| 23|
|Trump| 13|
+-----+---+
```

图 9-69 toDF 函数创建 DataFrame

（2） DataFrame 创建 RDD

在前面小节中我们已经使用到了 DataFrame 转 RDD 的方法，即.rdd，例如我们将 df 转为 rdd，代码如下图 9-70 所示。

```
In [95]: rdd = df.rdd
         rdd.collect()

Out[95]: [Row(name='Jones', age=23),
          Row(name='Tim', age=18),
          Row(name='Jane', age=50),
          Row(name='Mike', age=23),
          Row(name='Trump', age=13)]
```

图 9-70 DataFrame 转 RDD

注意，得到的 RDD 中存储的数据类型是 Row。

（3） 使用 RDD 中 foreach 函数

我们已经掌握了 DataFrame 转为 RDD 的方法，接下来我们可以将 DataFrame 转为 RDD 并使用 foreach 方法实现遍历输出，代码如下图 9-71 所示。

```
In [77]: def func(row):
             print("name is :", row["name"], "age is :", row["age"])

         df.rdd.foreach(func)
```

local模式输出结果应在控制台打印

```
name is : Andy age is : 30
name is : Justin age is : 21
name is : Berta age is : 14
name is : Berta age is : 14
name is : Bela age is : 26
name is : sira age is : 22
```

图 9-71 DataFrame 使用 foreach 函数

（4） 使用 RDD 中 map 函数

步骤 1：使用 map 函数处理 DataFrame

接下来，我们使用 RDD 的 map 函数实现将 DataFrame 中的 age 值乘以 10 和原 age 值一起输出，代码如下图 9-72 所示。

```
In [78]: df.rdd.map(lambda row : (row["age"], row["age"]*10)).take(5)
Out[78]: [(31, 310), (30, 300), (21, 210), (14, 140), (14, 140)]
```

图 9-72 DataFrame 使用 map 函数

步骤 2：生成 DataFrame

在上一步骤中我们使用 map 函数实现了对 DataFrame 的处理，从代码可以看出，生成数据的结果为 RDD 格式，我们可以通过 toDF()函数将 map 函数处理后的 RDD 转为 DataFrame，如图 9-73 所示。

```
In [79]: df_out = df.rdd.map(lambda row : (row["age"], row["age"]*10)).toDF(["age", "10_age"])
         df_out.show()
```

```
+---+------+
|age|10_age|
+---+------+
| 31|   310|
| 30|   300|
| 21|   210|
| 14|   140|
| 14|   140|
| 26|   260|
| 22|   220|
+---+------+
```

图 9-73 生成 DataFrame

# 9.3 Spark SQL 编程

### 9.3.1 Spark SQL 数据源创建

在前面章节，我们创建了 DataFrame，本节对 Spark SQL 的使用进行介绍。Spark SQL 的使用需要基于 DataFrame 创建的临时表 temp table 进行操作。

下面我们使用前面章节的 DataFrame 创建临时表，完成使用 Spark SQL 进行数据源的创建，也可以通过文件直接读取，详情如下所示。

（1） DataFrame 创建临时表

步骤 1：创建临时表

DataFrame 创建临时表代码如下。

```
df.registerTempTable("people_table")
```

上述代码完成了临时表的创建，临时表的表名为 people_table。

步骤 2：读取数据表

Spark SQL 读取 sqlContext.sql()，其中参数为 SQL 语句，需要 3 个双引号括住 SQL 语句，即："""SQL 语句"""，代码如图 9-74 所示。

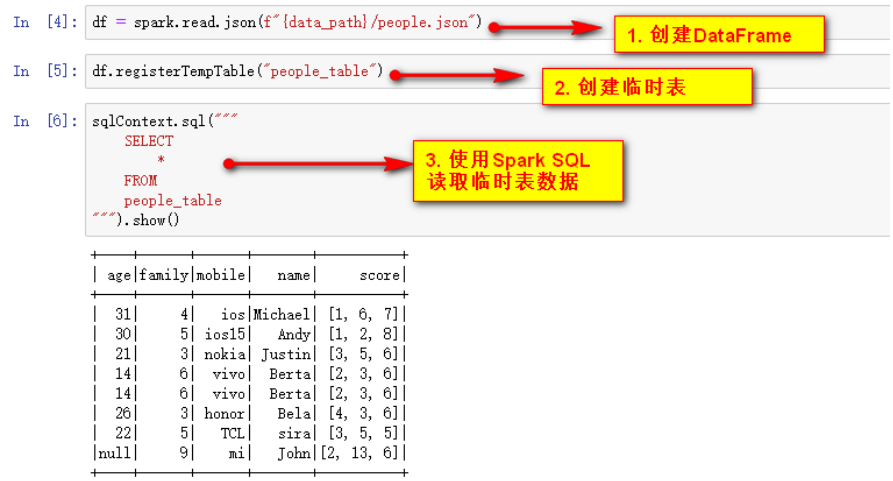



图 9-74 Spark SQL 读取临时表

（2） sqlContext 文件读取

除了上面通过 DataFrame 创建临时表的方法之外，我们还可以使用 sqlContext.sql()直接读取文件，代码如下图 9-75 所示。

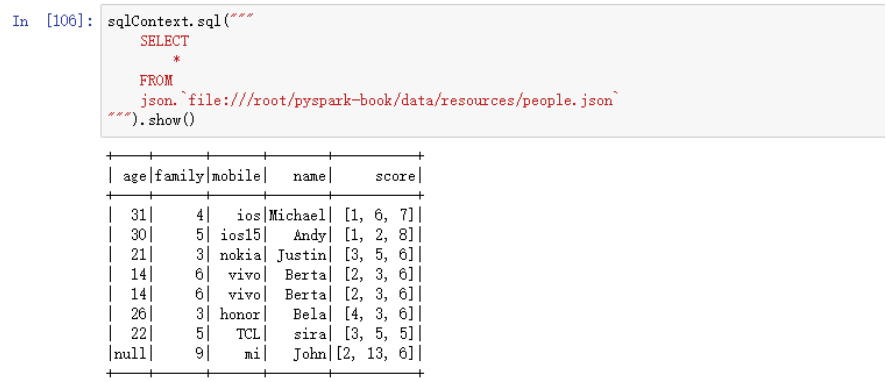



图 9-75 sqlContext.sql 读取 json 文件

注意，代码中'json'表示文件的格式，'.' 后面的文件具体路径需要用反引号括起来。

### 9.3.2 使用 Spark SQL 查询数据

Spark SQL 语法、内置函数和常规 SQL 语句基本一致，都可以便于开发者方便的使用 SQL 语句进行数据查询，详情如下所示。

（1） 查询指定字段

这里我们使用 select 语句查询 name、age 和 family 三列数据，实现代码如下图 9-76 所示。

```
In [107]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family
              FROM
              people_table
          """).show(5)
```

```
+-------+---+------+
|   name|age|family|
+-------+---+------+
|Michael| 31|     4|
|   Andy| 30|     5|
| Justin| 21|     3|
|  Berta| 14|     6|
|  Berta| 14|     6|
+-------+---+------+
only showing top 5 rows
```

图 9-76　Spark SQL 查询指定字段

（2）　LIMIT 关键字

通过 LIMIT 关键字可以限制查询的行数，如下图 9-77 所示。

```
In [108]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family
              FROM
              people_table
              LIMIT 5
          """).show()
```

```
+-------+---+------+
|   name|age|family|
+-------+---+------+
|Michael| 31|     4|
|   Andy| 30|     5|
| Justin| 21|     3|
|  Berta| 14|     6|
|  Berta| 14|     6|
+-------+---+------+
```

图 9-77 Spark SQL 使用 LIMIT 关键字

（3）　length 内置函数

使用 length 内置函数可以计算字段值的长度，如下图 9-78 所示。

```
In [110]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family,
                  length(name)
              FROM
              people_table
              LIMIT 5
          """).show()
```

```
+-------+---+------+--------------+
|   name|age|family|length(name)|
+-------+---+------+--------------+
|Michael| 31|     4|             7|
|   Andy| 30|     5|             4|
| Justin| 21|     3|             6|
|  Berta| 14|     6|             5|
|  Berta| 14|     6|             5|
+-------+---+------+--------------+
```

图 9-78 使用 length 内置函数

### 9.3.3 使用 Spark SQL 增加数据

使用 Spark SQL 语句增加计算字段也非常简单，只需在查询中增加新增计算字段的逻辑即可。这里同 DataFrame 增加字段相同，我们增加一列出生日期列，详情如下所示。

（1） 增加出生日期列

代码如下图 9-79 所示。

```
In [112]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family,
                  2022 - age
              FROM
              people_table
          """).show()
```

```
+-------+----+------+-------------------------------+
|   name| age|family|(CAST(2022 AS BIGINT) - age)|
+-------+----+------+-------------------------------+
|Michael|  31|     4|                           1991|
|   Andy|  30|     5|                           1992|
| Justin|  21|     3|                           2001|
|  Berta|  14|     6|                           2008|
|  Berta|  14|     6|                           2008|
|   Bela|  26|     3|                           1996|
|   sira|  22|     5|                           2000|
|   John|null|     9|                           null|
+-------+----+------+-------------------------------+
```

图 9-79 新增计算出生日期列

（2） 增加出生日期列并修改列名

在新增计算列的同时我们可以为该列的列名进行设置，如下图 9-80 所示。

```
In [113]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family,
                  (2022 - age) as birth
              FROM
              people_table
          """).show()
```

```
+-------+---+------+-----+
|   name|age|family|birth|
+-------+---+------+-----+
|Michael| 31|     4| 1991|
|   Andy| 30|     5| 1992|
| Justin| 21|     3| 2001|
|  Berta| 14|     6| 2008|
|  Berta| 14|     6| 2008|
|   Bela| 26|     3| 1996|
|   sira| 22|     5| 2000|
|   John|null|    9| null|
+-------+---+------+-----+
```

图 9-80 新增计算列并重命名

（3） 新增固定值列

在 DataFrame 章节中，我们为 DataFrame 新增了一列“school”列，所有行的值均相同。同样在 Spark SQL 里也可以很方便的实现，如下图 9-81 所示。

```
In [117]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family,
                  'high school' AS school
              FROM
              people_table
          """)\
          .show()
```

```
+-------+---+------+-----------+
|   name|age|family|     school|
+-------+---+------+-----------+
|Michael| 31|     4|high school|
|   Andy| 30|     5|high school|
| Justin| 21|     3|high school|
|  Berta| 14|     6|high school|
|  Berta| 14|     6|high school|
|   Bela| 26|     3|high school|
|   sira| 22|     5|high school|
|   John|null|    9|high school|
+-------+---+------+-----------+
```

图 9-81 新增固定值列

（4） withColumn 新增固定值列

我们可以使用 Spark SQL 和 withColumn 结合的方式来为数据新增一列“school”列，如下图 9-82 所示。

```
In [118]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family
              FROM
              people_table
          """)\
          .withColumn("school", F.lit("high school"))\
          .show()
```

```
+-------+---+------+-----------+
|   name|age|family|     school|
+-------+---+------+-----------+
|Michael|  31|     4|high school|
|   Andy|  30|     5|high school|
| Justin|  21|     3|high school|
|  Berta|  14|     6|high school|
|  Berta|  14|     6|high school|
|   Bela|  26|     3|high school|
|   sira|  22|     5|high school|
|   John|null|     9|high school|
+-------+---+------+-----------+
```

图 9-82 新增固定值列

### 9.3.4 使用 Spark SQL 修改数据

使用 Spark SQL 修改数据主要包括修改字段类型或列名和修改数据本身的值，当然，修改数据本身的值也可以为该列设置新字段名，不覆盖原有数据，详情如下所示。

（1） AS 关键字

使用 AS 关键字可以在查询数据的同时对字段进行重命名，AS 关键字同 SQL 语法完全一致，我们在上一小节中已经简单使用过了 AS 关键字，代码如下图 9-83 所示。

```
In [115]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family,
                  age as birth
              FROM
              people_table
          """).show()
```

```
+-------+---+------+-----+
|   name|age|family|birth|
+-------+---+------+-----+
|Michael|  31|     4|   31|
|   Andy|  30|     5|   30|
| Justin|  21|     3|   21|
|  Berta|  14|     6|   14|
|  Berta|  14|     6|   14|
|   Bela|  26|     3|   26|
|   sira|  22|     5|   22|
|   John|null|     9| null|
+-------+---+------+-----+
```

图 9-83 AS 关键字重名命列名

（2） 使用 CASE-WHEN 关键修改

和 DataFrame 相比，Spark SQL 编程可以使用更丰富的 SQL 语法函数，使用 CASE-WHEN 语句，

我们可以填充数据中的 null 值或对数据分段处理等等，如下图 9-84 所示。

```
In [8]: sqlContext.sql("""
            SELECT
                name,
                family,
                CASE WHEN
                    age is null then 666
                    else age
                END AS age
            FROM
            people_table
        """).show()
```

```
+-------+------+---+
|   name|family|age|
+-------+------+---+
|Michael|     4| 31|
|   Andy|     5| 30|
| Justin|     3| 21|
|  Berta|     6| 14|
|  Berta|     6| 14|
|   Bela|     3| 26|
|   sira|     5| 22|
|   John|     9|666|
+-------+------+---+
```

图 9-84 CASE-WHEN 语句

（3） 使用 CAST 关键字修改字段类型

使用 CAST 关键字，我们可以实现对字段的类型进行修改，代码如下图 9-85 所示。

```
In [116]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family,
                  CAST(age AS FLOAT)
              FROM
              people_table
          """).show()
```

```
+-------+----+------+----+
|   name| age|family| age|
+-------+----+------+----+
|Michael|  31|     4|31.0|
|   Andy|  30|     5|30.0|
| Justin|  21|     3|21.0|
|  Berta|  14|     6|14.0|
|  Berta|  14|     6|14.0|
|   Bela|  26|     3|26.0|
|   sira|  22|     5|22.0|
|   John|null|     9|null|
+-------+----+------+----+
```

图 9-85 CAST 关键字修改字段类型

（4） 使用 INITCAP 函数修改数据

类似于 Python 中的内置函数，我们可以使用 Spark SQL 的内置函数 INITCAP，实现对字符串的首字符的大小写转换，从而来修改数据，如下图 9-86 所示。

```
In [9]: sqlContext.sql("""
            SELECT
                name,
                age,
                family,
                INITCAP(mobile) AS mobile
            FROM
            people_table
        """).show()
```

```
+-------+----+------+------+
|   name| age|family|mobile|
+-------+----+------+------+
|Michael|  31|     4|   Ios|
|   Andy|  30|     5| Ios15|
| Justin|  21|     3| Nokia|
|  Berta|  14|     6|  Vivo|
|  Berta|  14|     6|  Vivo|
|   Bela|  26|     3| Honor|
|   sira|  22|     5|   Tcl|
|   John|null|     9|    Mi|
+-------+----+------+------+
```

图 9-86 INITCAP 函数修改数据

### 9.3.5 使用 Spark SQL 筛选数据

本小节将介绍使用 Spark SQL 来实现数据的筛选功能。这也是 SQL 里“增删改查”中重要的一项数据处理方式。

（1） WHERE 关键字

批注 [雷22]: 添加一段说明文

通 SQL 语法相同，Spark SQL 可以使用 WHERE 关键字进行数据过滤删除的查询，详情如下图 9-87、9-88 所示。

```
In [122]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family
              FROM
              people_table
              WHERE age is not null
          """).show()
```

```
+-------+---+------+
|   name|age|family|
+-------+---+------+
|Michael| 31|     4|
|   Andy| 30|     5|
| Justin| 21|     3|
|  Berta| 14|     6|
|  Berta| 14|     6|
|   Bela| 26|     3|
|   sira| 22|     5|
+-------+---+------+
```

图 9-87 WHERE 筛选非 null 值

```
In [121]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family
              FROM
              people_table
              WHERE age > 20
          """).show()
```

```
+-------+---+------+
|   name|age|family|
+-------+---+------+
|Michael| 31|     4|
|   Andy| 30|     5|
| Justin| 21|     3|
|   Bela| 26|     3|
|   sira| 22|     5|
+-------+---+------+
```

图 9-88 WHERE 筛选年龄大于 20 的数据

（2） .na.drop()函数的使用

同样的，我们可以结合 DataFrame 的函数进行 null 值的删除操作，如下图 9-89 所示。

```
In [123]: sqlContext.sql("""
              SELECT
                  name,
                  age,
                  family
              FROM
              people_table
          """)\
          .na.drop()\
          .show()
```

```
+-------+---+------+
|   name|age|family|
+-------+---+------+
|Michael| 31|     4|
|   Andy| 30|     5|
| Justin| 21|     3|
|  Berta| 14|     6|
|  Berta| 14|     6|
|   Bela| 26|     3|
|   sira| 22|     5|
+-------+---+------+
```

图 9-89 .na.drop()函数筛选非 null 值

（3） distinct 数据去重

使用 distinct 关键字可以实现对数据的去重操作，当然也可以结合使用 DataFrame 的去重函数来实现，如下图 9-90 所示。

```
In [124]: sqlContext.sql("""
              SELECT
                  distinct *
              FROM
              people_table
          """)\
          .show()
```

```
+---+------+------+--------+-------------+
|age|family|mobile|    name|        score|
+---+------+------+--------+-------------+
| 21|     3| nokia|  Justin| [3, 5, 6]|
| 31|     4|   ios| Michael| [1, 6, 7]|
| 22|     5|   TCL|    sira| [3, 5, 5]|
| 30|     5| ios15|    Andy| [1, 2, 8]|
| 14|     6|  vivo|   Berta| [2, 3, 6]|
| 26|     3| honor|    Bela| [4, 3, 6]|
|null|     9|    mi|    John|[2, 13, 6]|
+---+------+------+--------+-------------+
```

图 9-90 distinct 数据去重

### 9.3.6 使用 Spark SQL 合并数据

接下来，我们介绍 Spark SQL 如何实现数据合并的相关操作。

首先，我们同样将原始数据过滤 null 值之后进行数据表的拆分，代码如下。

```
df_n = df.distinct()
df_n.registerTempTable("people_unique")

sqlContext.sql("""
    SELECT
        name,
        age,
        mobile
    FROM
    people_unique
""").registerTempTable("table1")

sqlContext.sql("""
    SELECT
        name,
        family,
        score
    FROM
    people_unique
""").registerTempTable("table2")
```

上述代码将过滤 null 值之后的 DataFrame 通过 Spark SQL 语句创建了两个临时表 table1 和 table2。

（1） JOIN 数据关联

与 SQL 语句一样，使用 JOIN 关键字可以实现两个数据表的关联操作，如下图 9-91 所示。

```
In [127]: sqlContext.sql("""
              SELECT
                  *
              FROM table1 AS t1
              LEFT JOIN table2 AS t2
              ON t1.name == t2.name
              WHERE age is not null
          """).show()
```

```
+-------+---+------+-------+------+---------+
|   name|age|mobile|   name|family|    score|
+-------+---+------+-------+------+---------+
| Justin| 21| nokia| Justin|     3|[3, 5, 6]|
|  Berta| 14|  vivo|  Berta|     6|[2, 3, 6]|
|   sira| 22|   TCL|   sira|     5|[3, 5, 5]|
|   Bela| 26| honor|   Bela|     3|[4, 3, 6]|
|   Andy| 30| ios15|   Andy|     5|[1, 2, 8]|
|Michael| 31|   ios|Michael|     4|[1, 6, 7]|
+-------+---+------+-------+------+---------+
```

图 9-91 JOIN 数据关联

（2） concat_ws 数据合并

在这里，我们介绍一种经常使用到的内置函数 concat_ws，使用该函数可以按照指定字符将多列值进行合并拼接的操作，实现代码如下图 9-92 所示。

```
In [129]: sqlContext.sql("""
              SELECT
                  *,
                  concat_ws('-', name, age) AS info
              FROM
              people_unique
              WHERE age is not null
          """).show()
```

```
+---+------+------+-------+---------+---------+
|age|family|mobile|   name|    score|     info|
+---+------+------+-------+---------+---------+
| 21|     3| nokia| Justin|[3, 5, 6]|Justin-21|
| 14|     6|  vivo|  Berta|[2, 3, 6]| Berta-14|
| 22|     5|   TCL|   sira|[3, 5, 5]|  sira-22|
| 26|     3| honor|   Bela|[4, 3, 6]|  Bela-26|
| 30|     5| ios15|   Andy|[1, 2, 8]|  Andy-30|
| 31|     4|   ios|Michael|[1, 6, 7]|Michael-31|
+---+------+------+-------+---------+---------+
```

图 9-92 concat_ws 数据合并

### 9.3.7 使用 Spark SQL 进阶处理

接下来，我们介绍 Spark SQL 进阶处理的相关内容。主要包括排序、分组处理、数据转换等内容，详情如下所示。

（1） ORDER BY 排序

与 SQL 语法相同，Spark SQL 同样使用 ORDER BY 关键字实现对数据的排序，默认为升序排序，如下图 9-93 所示。

```
In [132]: sqlContext.sql("""
              SELECT
                  *
              FROM
              people_table
              ORDER BY age
          """).show()
```

```
+----+------+------+-------+----------+
| age|family|mobile|   name|     score|
+----+------+------+-------+----------+
|null|     9|    mi|   John|[2, 13, 6]|
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|  21|     3| nokia| Justin| [3, 5, 6]|
|  22|     5|   TCL|   sira| [3, 5, 5]|
|  26|     3| honor|   Bela| [4, 3, 6]|
|  30|     5| ios15|   Andy| [1, 2, 8]|
|  31|     4|   ios|Michael| [1, 6, 7]|
+----+------+------+-------+----------+
```

图 9-93 ORDER BY 升序排序

这里使用 DESC 关键字即可实现将数据按某列进行降序排序，代码如图 9-94 所示。

```
In [134]: sqlContext.sql("""
              SELECT
                  *
              FROM
              people_table
              ORDER BY age DESC
          """).show()
```

```
+----+------+------+-------+----------+
| age|family|mobile|   name|     score|
+----+------+------+-------+----------+
|  31|     4|   ios|Michael| [1, 6, 7]|
|  30|     5| ios15|   Andy| [1, 2, 8]|
|  26|     3| honor|   Bela| [4, 3, 6]|
|  22|     5|   TCL|   sira| [3, 5, 5]|
|  21|     3| nokia| Justin| [3, 5, 6]|
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|  14|     6|  vivo|  Berta| [2, 3, 6]|
|null|     9|    mi|   John|[2, 13, 6]|
+----+------+------+-------+----------+
```

图 9-94 ORDER BY 降序排序

（2） GROUP BY 分组聚合

使用 GROUP BY 进行数据分组聚合，这里同 DataFrame 相同，介绍三类常用聚合方式。

首先，我们使用 count 实现对分组数据的简单统计操作，如下图 9-95 所示。

```
In [136]: sqlContext.sql("""
              SELECT
                  family,
                  count(1)
              FROM
              people_table
              GROUP BY family
          """).show()
```

```
+------+--------+
|family|count(1)|
+------+--------+
|     6|       2|
|     9|       1|
|     5|       2|
|     3|       2|
|     4|       1|
+------+--------+
```

图 9-95 count 分组统计

Spark SQL 同样可以使用 DataFrame 的 collect_list 和 collect_set 函数来实现将分组数据聚合为一个数组，如下图 9-96、9-97 所示。

```
In [138]: sqlContext.sql("""
              SELECT
                  family,
                  COLLECT_LIST(name)
              FROM
              people_table
              GROUP BY family
          """).show()
```

```
+------+--------------------+
|family|collect_list(name)|
+------+--------------------+
|     6|    [Berta, Berta]|
|     9|            [John]|
|     5|      [Andy, sira]|
|     3|    [Justin, Bela]|
|     4|         [Michael]|
+------+--------------------+
```

图 9-96 collect_list 分组聚合

```
In [139]: sqlContext.sql("""
              SELECT
                  family,
                  COLLECT_SET(name)
              FROM
              people_table
              GROUP BY family
          """).show()
```

```
+------+--------------------+
|family|collect_set(name)|
+------+--------------------+
|     6|          [Berta]|
|     9|           [John]|
|     5|     [Andy, sira]|
|     3|   [Bela, Justin]|
|     4|        [Michael]|
+------+--------------------+
```

图 9-97 collect_set 分组聚合

（3） EXPLODE 列转多行

下面我们使用 EXPLODE 函数实现将一列数组类型的数据拆分成多行内容，如下图 9-98 所示。

```
In [141]: sqlContext.sql("""
              SELECT
                  name,
                  score,
                  EXPLODE(score) AS col
              FROM
              people_unique
          """).show()
```

```
+-------+---------+---+
|   name|    score|col|
+-------+---------+---+
| Justin|[3, 5, 6]|  3|
| Justin|[3, 5, 6]|  5|
| Justin|[3, 5, 6]|  6|
|  Berta|[2, 3, 6]|  2|
|  Berta|[2, 3, 6]|  3|
|  Berta|[2, 3, 6]|  6|
|   sira|[3, 5, 5]|  3|
|   sira|[3, 5, 5]|  5|
|   sira|[3, 5, 5]|  5|
|   Bela|[4, 3, 6]|  4|
|   Bela|[4, 3, 6]|  3|
|   Bela|[4, 3, 6]|  6|
|   Andy|[1, 2, 8]|  1|
|   Andy|[1, 2, 8]|  2|
|   Andy|[1, 2, 8]|  8|
|Michael|[1, 6, 7]|  1|
|Michael|[1, 6, 7]|  6|
|Michael|[1, 6, 7]|  7|
|   John|[2, 13, 6]|  2|
|   John|[2, 13, 6]| 13|
+-------+---------+---+
only showing top 20 rows
```

图 9-98 EXPLODE 列转多行

（4） TO_JSON 函数

在 DataFrame 的章节中，我们介绍了 to_json()函数，可以实现将每行数据转换成一行 json 字符串，在 Spark SQL 中 TO_JSON 函数需要配合 MAP 函数共同来实现该功能，如下图 9-99 所示。

```
In [143]: sqlContext.sql("""
              SELECT
                  *,
                  TO_JSON(MAP("NAME" , name, "AGE", age)) AS json
              FROM
              people_table
          """).show(truncate=False)
```

```
+---+------+------+-------+---------+-----------------------------+
|age|family|mobile|name   |score    |json                         |
+---+------+------+-------+---------+-----------------------------+
|31 |4     |ios   |Michael|[1, 6, 7]|{"NAME":"Michael","AGE":"31"}|
|30 |5     |ios15 |Andy   |[1, 2, 8]|{"NAME":"Andy","AGE":"30"}   |
|21 |3     |nokia |Justin |[3, 5, 6]|{"NAME":"Justin","AGE":"21"} |
|14 |6     |vivo  |Berta  |[2, 3, 6]|{"NAME":"Berta","AGE":"14"}  |
|14 |6     |vivo  |Berta  |[2, 3, 6]|{"NAME":"Berta","AGE":"14"}  |
|26 |3     |honor |Bela   |[4, 3, 6]|{"NAME":"Bela","AGE":"26"}   |
|22 |5     |TCL   |sira   |[3, 5, 5]|{"NAME":"sira","AGE":"22"}   |
|null|9    |mi    |John   |[2, 13, 6]|{"NAME":"John","AGE":null}  |
+---+------+------+-------+---------+-----------------------------+
```

图 9-99 TO_JSON 函数的使用

### 9.3.8 自定义 UDF 函数

在一些场景中，Spark SQL 内置的函数无法满足我们对数据处理的需求，这时，我们可以在 Spark SQL 中使用 UDF 函数来实现数据处理的目的。

在 Spark SQL 中使用 UDF 函数需要对 UDF 函数进行注册，UDF 函数的注册需要调用下面代码所示的函数。

```
spark.udf.register(name, f, returnType=None)
```

其中，name 是指要注册的 UDF 函数的名字，f 为编写的 UDF 函数，returnType 为 UDF 返回值类型。

接下来，和 DataFrame 的处理逻辑相同，首先对单一列进行 UDF 函数处理，将 name 列进行大小写转换并拼接原始 name 值，详情如下所示。

（1） 单列 UDF 函数使用

如下图 9-100 所示。

```
In [144]: def one_udf(name):
              return name +":"+ name.lower()

In [145]: spark.udf.register("name_udf", one_udf, StringType())
Out[145]: <function __main__.one_udf(name)>

In [146]: sqlContext.sql("""
              SELECT
                  *,
                  name_udf(name) AS full_name
              FROM
              people_table
          """).show()
```

1. 编写UDF函数

2. 注册UDF函数

3. 调用UDF函数

```
+---+------+------+-------+----------+-------------+
|age|family|mobile|   name|     score|    full_name|
+---+------+------+-------+----------+-------------+
| 31|     4|   ios|Michael| [1, 6, 7]|Michael:michael|
| 30|     5| ios15|   Andy| [1, 2, 8]|    Andy:andy|
| 21|     3| nokia| Justin| [3, 5, 6]|Justin:justin|
| 14|     6|  vivo|  Berta| [2, 3, 6]|  Berta:berta|
| 14|     6|  vivo|  Berta| [2, 3, 6]|  Berta:berta|
| 26|     3| honor|   Bela| [4, 3, 6]|    Bela:bela|
| 22|     5|   TCL|   sira| [3, 5, 5]|    sira:sira|
|null|    9|    mi|   John|[2, 13, 6]|    John:john|
+---+------+------+-------+----------+-------------+
```

图 9-100 Spark SQL 使用 UDF 函数

（2） 多列 UDF 函数使用

如下图 9-101 所示

```
In [147]: def two_udf(name, age):
              return [name, age]

In [148]: spark.udf.register("con_udf", two_udf, ArrayType(StringType()))

Out[148]: <function __main__.two_udf(name, age)>

In [150]: sqlContext.sql("""
              SELECT
                  *,
                  con_udf(name, age) AS full_info
              FROM
              people_table
          """).show()
```

```
+---+------+------+-------+---------+-------------+
|age|family|mobile|   name|    score|    full_info|
+---+------+------+-------+---------+-------------+
| 31|     4|   ios|Michael|[1, 6, 7]|[Michael, 31]|
| 30|     5| ios15|   Andy|[1, 2, 8]|   [Andy, 30]|
| 21|     3| nokia| Justin|[3, 5, 6]| [Justin, 21]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|  [Berta, 14]|
| 14|     6|  vivo|  Berta|[2, 3, 6]|  [Berta, 14]|
| 26|     3| honor|   Bela|[4, 3, 6]|   [Bela, 26]|
| 22|     5|   TCL|   sira|[3, 5, 5]|   [sira, 22]|
|null|    9|    mi|   John|[2, 13, 6]|     [John,]|
+---+------+------+-------+---------+-------------+
```

图 9-101 Spark SQL 使用 UDF 函数

# 9.4 RDD、DataFrame 比较

DataFrame 是在 SparkSQL 中 Spark 为我们提供的抽象。那么 DataFrame 和 RDD 有什么区别呢？首先从版本的产生上来看：RDD 产生于 Spark 1.0 版本，而 Dataframe 产生于 Spark 1.3 版本。

DataFrame 和 RDD 有如下共同点：

1. RDD、DataFrame 都是 Spark 平台下的分布式弹性数据集，为处理超大型数据提供便利；
2. 二者都有惰性机制，在进行创建、转换，如 map 方法时，不会立即执行，只有在遇到 Action 如 foreach 时，二者才会开始遍历运算；
3. 二者都会根据 Spark 的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存溢出；
4. 二者都有 partition 的概念；
5. 二者有许多共同的函数，如 map、filter，排序等。

DataFrame 和 RDD 有如下区别：

1. RDD 一般和 Spark MLib 同时使用；
2. RDD 不支持 Spark SQL 操作；
3. 与 RDD 不同，DataFrame 每一行的类型固定为 Row，每一列的值没法直接访问，只有通过解析才能获取各个字段的值；
4. DataFrame 一般不与 Spark MLib 同时使用。

最后，RDD 与 DataFrame 的转换可以通过下图 9-102 进行概括。

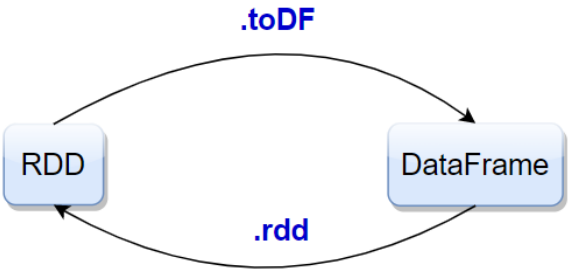



图 9-102 RDD 与 DataFrame 的转换

# ★回顾思考★

**01　Spark SQL 有哪些特点？**

答：包括如下：
（1）Integrated，易整合；
（2）Uniform Data Access，统一的数据访问方式；
（3）Hive Integration，集成 Hive；
（4）Standard Connectivity，标准的连接方式。

**02　Spark SQL 的两个抽象类是？**

答：DataFrame 和 DataSet。

**03 RDD 和 DataFrame 的不同点？**

答：（1）RDD 一般和 Spark MLib 同时使用；
（2）RDD 不支持 Spark SQL 操作；
（3）与 RDD 不同，DataFrame 每一行的类型固定为 Row，每一列的值没法直接访问，只有通过解析才能获取各个字段的值；
（4）DataFrame 一般不与 Spark MLib 同时使用。

# ★练习★

**一、选择题**

1. 使用下面哪个方法可以将 RDD 转为 DataFrame（ ）
   A. todf()
   B. to_DF()
   C. ToDF()
   D. DF()
2. 下面哪个语句可以实现 DataFrame 过滤掉 null 值数据行（ ）
   A. nadrop()
   B. na.drop()
   C. drop.na()
   D. filterna()
3. 下面哪个函数方法可以对 DataFrame 创建临时表（ ）
   A. registerTempTable ()
   B. createrTempTable ()
   C. registerTable ()
   D. registerGlobalTable ()
4. 假设已有编写的 UDF 函数 func(arg)，返回值为整数类型，下面注册 UDF 函数正确的是（ ）
   A. my_udf = F.udf(func(),int)
   B. my_udf = F.udf(func(), Integer())
   C. my_udf() = F.udf(func,Integer())
   D. my_udf = F.udf(func, IntegerType())
5. 假设已有编写的 UDF 函数 func(arg)，返回值为字符串类型，下面注册 UDF 函数正确的是（ ）
   A. spark.udf.register(my_func, func, StringType())
   B. spark.udf.register(my_func, func(), StringType())
   C. spark.udf.register(my_func, func, StrType())
   D. spark.udf.register(my_func(), func, StringType())

**二、填空题**

1.只有 DataFrame 可以使用 UDF 函数。（×）

2. filter()和 where()都可以实现对 DataFrame 的过滤操作。（√）

**三、实战练习**

**任务 1：**完整实现 DataFrame 的数据处理操作代码实战。

请参考本章 9.2 节内容和本书配套第 9 章 Notebook 代码文件。

**任务 2：**完整实现 Spark SQL 的数据处理操作代码实战。

请参考本章 9.3 节内容和本书配套第 9 章 Notebook 代码文件。

批注 [雷23]: 完成答案

批注 [HZ24R23]: 在第 9 章代码文件里

# 本章小结

本章分别对 Spark DataFrame、Spark SQL 以及 DataSet 的概念和它们之前的区别进行了介绍，这部

分内容可能晦涩不便于理解，本书以实战为主，读者可以对概念介绍有个初步了解后通过代码实战的方式加深对 DataFrame 与 Spark SQL 的理解。

DataFrame 与 RDD 有共同点也有区别，在 Spark 数据挖掘与数据分析中，机器学习算法的应用越来越广泛，而 Spark ML 又是 Spark 机器学习的趋势，DataFrame 可以很好的支持 Spark ML 的应用，所以掌握 Spark DataFrame 的灵活使用，可以为后续 Spark 机器学习奠定良好的基础。