

# Spark SQL

## 一、概述

1. `spark sql` 是用于操作结构化数据的程序包
  - 通过 `spark sql` , 可以使用 `SQL` 或者 `HQL` 来查询数据, 查询结果以 `Dataset/DataFrame` 的形式返回
  - 它支持多种数据源, 如 `Hive` 表、`Parquet` 以及 `JSON` 等
  - 它支持开发者将 `SQL` 和传统的 `RDD` 变成相结合
2. `Dataset` : 是一个分布式的数据集合
  - 它是 `Spark 1.6` 中被添加的新接口
  - 它提供了 `RDD` 的优点与 `Spark SQL` 执行引擎的优点
  - 它在 `Scala` 和 `Java` 中是可用的。 `Python` 不支持 `Dataset API` 。但是由于 `Python` 的动态特性, 许多 `DataSet API` 的优点已经可用
3. `DataFrame` : 是一个 `Dataset` 组成的指定列。
  - 它的概念等价于一个关系型数据库中的表
  - 在 `Scala/Python` 中, `DataFrame` 由 `DataSet` 中的 `RowS` (多个 `Row` ) 来表示。
4. 在 `spark 2.0` 之后, `SQLContext` 被 `SparkSession` 取代。

## 二、SparkSession

1. `spark sql` 中所有功能的入口点是 `SparkSession` 类。它可以用于创建 `DataFrame` 、注册 `DataFrame` 为 `table` 、在 `table` 上执行 `SQL` 、缓存 `table` 、读写文件等等。
2. 要创建一个 `SparkSession` , 仅仅使用 `SparkSession.builder` 即可:

```
from pyspark.sql import SparkSession
spark_session = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

3. `Builder` 用于创建 `SparkSession` , 它的方法有 (这些方法都返回 `self` ) :
  - `.appName(name)` : 给程序设定一个名字, 用于在 `Spark web UI` 中展示。如果未指定, 则 `spark` 会随机生成一个。
    - `name` : 一个字符串, 表示程序的名字
  - `.config(key=None, value=None, conf=None)` : 配置程序。这里设定的配置会直接传递给 `SparkConf` 和 `SparkSession` 各自的配置。
    - `key` : 一个字符串, 表示配置名
    - `value` : 对应配置的值
    - `conf` : 一个 `SparkConf` 实例有两种设置方式:

- 通过键值对设置：

```
SparkSession.builder.config("spark.some.config.option", "some-value")
```

- 通过已有的 `SparkConf` 设置：

```
SparkSession.builder.config(conf=SparkConf())
```

- `.enableHiveSupport()`：开启 `Hive` 支持。（`spark 2.0` 的新接口）
- `.master(master)`：设置 `spark master URL`。如：
  - `master=local`：表示单机本地运行
  - `master=local[4]`：表示单机本地4核运行
  - `master=spark://master:7077`：表示在一个 `spark standalone cluster` 上运行
- `.getOrCreate()`：返回一个已有的 `SparkSession` 实例；如果没有则基于当前 `builder` 的配置，创建一个新的 `SparkSession` 实例
  - 该方法首先检测是否有一个有效的全局默认 `SparkSession` 实例。如果有，则返回它；如果没有，则创建一个作为全局默认 `SparkSession` 实例，并返回它
  - 如果已有一个有效的全局默认 `SparkSession` 实例，则当前 `builder` 的配置将应用到该实例上

## 2.1 属性

1. `.builder` = `<pyspark.sql.session.Builder object at 0x7f51f134a110>`：一个 `Builder` 实例
2. `.catalog`：一个接口。用户通过它来 `create`、`drop`、`alter`、`query` 底层的数据库、`table` 以及 `function` 等
  - 可以通过 `SparkSession.catalog.cacheTable('tableName')`，来缓存表；通过 `SparkSession.catalog.uncacheTable('tableName')` 来从缓存中删除该表。
3. `.conf`：`spark` 的运行时配置接口。通过它，你可以获取、设置 `spark`、`hadoop` 的配置。
4. `.read`：返回一个 `DataFrameReader`，用于从外部存储系统中读取数据并返回 `DataFrame`
5. `.readStream`：返回一个 `DataStreamReader`，用于将输入数据流视作一个 `DataFrame` 来读取
6. `.sparkContext`：返回底层的 `SparkContext`
7. `.streams`：返回一个 `StreamingQueryManager` 对象，它管理当前上下文的所有活动的 `StreamingQuery`
8. `.udf`：返回一个 `UDFRegistration`，用于 `UDF` 注册
9. `.version`：返回当前应用的 `spark` 版本

## 2.2 方法

1. `.createDataFrame(data, schema=None, samplingRatio=None, verifySchema=True)`：从 `RDD`、一个列表、或者 `pandas.DataFrame` 中创建一个 `DataFrame`
  - 参数：
    - `data`：输入数据。可以为一个 `RDD`、一个列表、或者一个 `pandas.DataFrame`
    - `schema`：给出了 `DataFrame` 的结构化信息。可以为：

- 一个字符串的列表：给出了列名信息。此时每一列数据的类型从 `data` 中推断
  - 为 `None`：此时要求 `data` 是一个 `RDD`，且元素类型为 `Row`、`namedtuple`、`dict` 之一。此时结构化信息从 `data` 中推断（推断列名、列类型）
  - 为 `pyspark.sql.types.StructType`：此时直接指定了每一列数据的类型。
  - 为 `pyspark.sql.types.DataType` 或者 `datatype string`：此时直接指定了一列数据的类型，会自动封装成 `pyspark.sql.types.StructType`（只有一列）。此时要求指定的类型与 `data` 匹配（否则抛出异常）
  - `samplingRatio`：如果需要推断数据类型，则它指定了需要多少比例的行记录来执行推断。如果为 `None`，则只使用第一行来推断。
  - `verifySchema`：如果为 `True`，则根据 `schema` 检验每一行数据
    - 返回值：一个 `DataFrame` 实例
2. `.newSession()`：返回一个新的 `SparkSession` 实例，它拥有独立的 `SQLConf`、`registered temporary views and UDFs`，但是共享同样的 `SparkContext` 以及 `table cache`。
  3. `.range(start,end=None,step=1,numPartitions=None)`：创建一个 `DataFrame`，它只有一列。该列的列名为 `id`，类型为 `pyspark.sql.types.LongType`，数值为区间 `[start,end)`，间隔为 `step`（即：`list(range(start,end,step))`）
  4. `.sql(sqlQuery)`：查询 `SQL` 并以 `DataFrame` 的形式返回查询结果
  5. `.stop()`：停止底层的 `SparkContext`
  6. `.table(tableName)`：以 `DataFrame` 的形式返回指定的 `table`

### 三、DataFrame 创建

1. 在一个 `SparkSession` 中，应用程序可以从一个已经存在的 `RDD`、`HIVE` 表、或者 `spark` 数据源中创建一个 `DataFrame`

#### 3.1 从列表创建

1. 未指定列名：

```
l = [('Alice', 1)]
spark_session.createDataFrame(l).collect()
```

结果为：

```
[Row(_1=u'Alice', _2=1)] #自动分配列名
```

2. 指定列名：

```
l = [('Alice', 1)]
spark_session.createDataFrame(l, ['name', 'age']).collect()
```

结果为：

```
[Row(name=u'Alice', age=1)]
```

### 3. 通过字典指定列名:

```
d = [{'name': 'Alice', 'age': 1}]  
spark_session.createDataFrame(d).collect()
```

结果为:

```
[Row(age=1, name=u'Alice')]
```

## 3.2 从 RDD 创建

### 1. 未指定列名:

```
rdd = sc.parallelize(['Alice', 1])  
spark_session.createDataFrame(rdd).collect()
```

结果为:

```
[Row(_1=u'Alice', _2=1)] #自动分配列名
```

### 2. 指定列名:

```
rdd = sc.parallelize(['Alice', 1])  
spark_session.createDataFrame(rdd, ['name', 'age']).collect()
```

结果为:

```
[Row(name=u'Alice', age=1)]
```

### 3. 通过 Row 来创建:

```
from pyspark.sql import Row  
Person = Row('name', 'age')  
rdd = sc.parallelize(['Alice', 1]).map(lambda r: Person(*r))  
spark_session.createDataFrame(rdd, ['name', 'age']).collect()
```

结果为:

```
[Row(name=u'Alice', age=1)]
```

#### 4. 指定 schema :

```
from pyspark.sql.types import *
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)])
rdd = sc.parallelize([('Alice', 1)])
spark_session.createDataFrame(rdd, schema).collect()
```

结果为:

```
[Row(name=u'Alice', age=1)]
```

#### 5. 通过字符串指定 schema :

```
rdd = sc.parallelize([('Alice', 1)])
spark_session.createDataFrame(rdd, "a: string, b: int").collect()
```

结果为:

```
[Row(name=u'Alice', age=1)]
```

- 如果只有一列，则字符串 schema 为:

```
rdd = sc.parallelize([1])
spark_session.createDataFrame(rdd, "int").collect()
```

结果为:

```
[Row(value=1)]
```

## 3.3 从 pandas.DataFrame 创建

#### 1. 使用方式:

```
df = pd.DataFrame({'a':[1,3,5], 'b':[2,4,6]})
spark_session.createDataFrame(df).collect()
```

结果为:

```
[Row(a=1, b=2), Row(a=3, b=4), Row(a=5, b=6)]
```

## 3.4 从数据源创建

1. 从数据源创建的接口是 `DataFrameReader` :

```
reader = spark_session.read
```

2. 另外, 也可以不使用 `API` , 直接将文件加载到 `DataFrame` 并进行查询:

```
df = spark_session.sql("SELECT * FROM
parquet.`examples/src/main/resources/users.parquet`")
```

### 3.4.1 通用加载

1. 设置数据格式: `.format(source)`。

- 返回 `self`

```
df = spark_session.read.format('json').load('python/test_support/sql/people.json')
```

2. 设置数据 `schema` : `.schema(schema)`。

- 返回 `self`
- 某些数据源可以从输入数据中推断 `schema` 。一旦手动指定了 `schema` , 则不再需要推断。

3. 加载: `.load(path=None, format=None, schema=None, **options)`

- 参数:
  - `path` : 一个字符串, 或者字符串的列表。指出了文件的路径
  - `format` : 指出了文件类型。默认为 `parquet` (除非另有配置 `spark.sql.sources.default` )
  - `schema` : 输入数据的 `schema` , 一个 `StructType` 类型实例。
  - `options` : 其他的参数
- 返回值: 一个 `DataFrame` 实例
- 示例:

```
spark_session.read.format('json').load(['python/test_support/sql/people.json',
'python/test_support/sql/people1.json'])
```

### 3.4.2 专用加载

1. `.csv()` : 加载 `csv` 文件, 返回一个 `DataFrame` 实例

```
.csv(path, schema=None, sep=None, encoding=None, quote=None, escape=None, comment=None,
header=None, inferSchema=None, ignoreLeadingWhiteSpace=None,
ignoreTrailingWhiteSpace=None, nullValue=None, nanValue=None, positiveInf=None,
negativeInf=None, dateFormat=None, timestampFormat=None, maxColumns=None,
maxCharsPerColumn=None, maxMalformedLogPerPartition=None, mode=None,
columnNameOfCorruptRecord=None, multiLine=None)
```

2. `.jdbc()` : 加载数据库中的表

```
.jdbc(url, table, column=None, lowerBound=None, upperBound=None, numPartitions=None,
predicates=None, properties=None)
```

○ 参数:

- `url`: 一个 JDBC URL, 格式为: `jdbc:subprotocol:subname`
- `table`: 表名
- `column`: 列名。该列为整数列, 用于分区。如果该参数被设置, 那么 `numPartitions`、`lowerBound`、`upperBound` 将用于分区从而生成 `where` 表达式来拆分该列。
- `lowerBound`: `column` 的最小值, 用于决定分区的步长
- `upperBound`: `column` 的最大值 (不包含), 用于决定分区的步长
- `numPartitions`: 分区的数量
- `predicates`: 一系列的表达式, 用于 `where` 中。每一个表达式定义了 `DataFrame` 的一个分区
- `properties`: 一个字典, 用于定义 JDBC 连接参数。通常至少为: `{ 'user' : 'SYSTEM', 'password' : 'mypassword' }`

○ 返回: 一个 `DataFrame` 实例

3. `.json()`: 加载 `json` 文件, 返回一个 `DataFrame` 实例

```
.json(path, schema=None, primitivesAsString=None, prefersDecimal=None,
allowComments=None, allowUnquotedFieldNames=None, allowSingleQuotes=None,
allowNumericLeadingZero=None, allowBackslashEscapingAnyCharacter=None, mode=None,
columnNameOfCorruptRecord=None, dateFormat=None, timestampFormat=None, multiline=None)
```

示例:

```
spark_session.read.json('python/test_support/sql/people.json')
# 或者
rdd = sc.textFile('python/test_support/sql/people.json')
spark_session.read.json(rdd)
```

4. `.orc()`: 加载 `ORC` 文件, 返回一个 `DataFrame` 实例

```
.orc(path)
```

示例:

```
spark_session.read.orc('python/test_support/sql/orc_partitioned')
```

5. `.parquet()`: 加载 `Parquet` 文件, 返回一个 `DataFrame` 实例

```
.parquet(*paths)
```

示例:

```
spark_session.read.parquet('python/test_support/sql/parquet_partitioned')
```

6. `.table()`: 从 `table` 中创建一个 `DataFrame`

```
.table(tableName)
```

示例:

```
df = spark_session.read.parquet('python/test_support/sql/parquet_partitioned')
df.createOrReplaceTempView('tmpTable')
spark_session.read.table('tmpTable')
```

7. `.text()`: 从文本中创建一个 `DataFrame`

```
.text(paths)
```

它不同于 `.csv()`, 这里的 `DataFrame` 只有一列, 每行文本都是作为一个字符串。

示例:

```
spark_session.read.text('python/test_support/sql/text-test.txt').collect()
#结果为: [Row(value=u'hello'), Row(value=u'this')]
```

### 3.5 从 Hive 表创建

1. `spark SQL` 还支持读取和写入存储在 `Apache Hive` 中的数据。但是由于 `Hive` 具有大量依赖关系, 因此这些依赖关系不包含在默认 `spark` 版本中。
  - 如果在类路径中找到 `Hive` 依赖项, 则 `Spark` 将会自动加载它们
  - 这些 `Hive` 的依赖关系也必须存在于所有工作节点上
2. 配置: 将 `hive-site.xml`、`core-site.html` (用于安全配置)、`hdfs-site.xml` (用户 `HDFS` 配置) 文件放在 `conf/` 目录中完成配置。
3. 当使用 `Hive` 时, 必须使用启用 `Hive` 支持的 `SparkSession` 对象 (`enableHiveSupport`)
  - 如果未部署 `Hive`, 则开启 `Hive` 支持不会报错
4. 当 `hive-site.xml` 未配置时, 上下文会自动在当前目录中创建 `metastore_db`, 并创建由 `spark.sql.warehouse.dir` 指定的目录
5. 访问示例:



```

from pyspark.sql import SparkSession
spark_sess = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", '/home/xxx/yyy/') \
    .enableHiveSupport() \
    .getOrCreate()
spark_sess.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark_sess.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")
spark.sql("SELECT * FROM src").show()

```

6. 创建 `Hive` 表时，需要定义如何向/从文件系统读写数据，即：输入格式、输出格式。还需要定义该表的数据的序列化与反序列化。

可以通过在 `OPTIONS` 选项中指定这些属性：

```

spark_sess.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive
OPTIONS(fileFormat 'parquet')")

```

可用的选项有：

- `fileFormat`：文件格式。目前支持6种文件格式：`'sequencefile'`、`'rcfile'`、`'orc'`、`'parquet'`、`'textfile'`、`'avro'`。
- `inputFormat, outputFormat`：这两个选项将相应的 `InputFormat` 和 `OutputFormat` 类的名称指定为字符串文字，如 `'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'`
  - 这两个选项必须成对出现
  - 如果已经制定了 `fileFormat`，则无法指定它们
- `serde`：该选项指定了 `serde` 类的名称
  - 如果给定的 `fileFormat` 已经包含了 `serde` 信息（如何序列化、反序列化的信息），则不要指定该选项
  - 目前的 `sequencefile`、`textfile`、`rcfile` 不包含 `serde` 信息，因此可以使用该选项
- `fieldDelim, escapeDelim, collectionDelim, mapkeyDelim, lineDelim`：这些选项只能与 `textfile` 文件格式一起使用，它们定义了如何将分隔的文件读入行。

## 四、DataFrame 保存

1. `DataFrame` 通过 `DataFrameWriter` 实例来保存到各种外部存储系统中。

- 你可以通过 `DataFrame.write` 来访问 `DataFrameWriter`

### 4.1 通用保存

1. `.format(source)`：设置数据格式

- 返回 `self`

```
df.write.format('json').save('./data.json')
```

2. `.mode(saveMode)`：当要保存的目标位置已经有数据时，设置该如何保存。

- 参数： `saveMode` 可以为：
  - `'append'`：追加写入
  - `'overwrite'`：覆写已有数据
  - `'ignore'`：忽略本次保存操作（不保存）
  - `'error'`：抛出异常（默认行为）
- 返回 `self`
- 示例：

```
df.write.mode('append').parquet('./data.dat')
```

3. `.partitionBy(*cols)`：按照指定的列名来将输出的 `DataFrame` 分区。

- 返回 `self`
- 示例：

```
df.write.partitionBy('year', 'month').parquet('./data.dat')
```

4. `.save(path=None, format=None, mode=None, partitionBy=None, **options)`：保存 `DataFrame`

## 4.2 专用保存

1. `.csv()`：将 `DataFrame` 保存为 `csv` 文件

```
.csv(path, mode=None, compression=None, sep=None, quote=None, escape=None, header=None,
nullValue=None, escapeQuotes=None, quoteAll=None, dateFormat=None, timestampFormat=None,
ignoreLeadingWhiteSpace=None, ignoreTrailingWhiteSpace=None)
```

示例：

```
df.write.csv('./data.csv')
```

2. `.insertInto()`：将 `DataFrame` 保存在 `table` 中

```
.insertInto(tableName, overwrite=False)
```

它要求当前的 `DataFrame` 与指定的 `table` 具有同样的 `schema`。其中 `overwrite` 参数指定是否覆盖 `table` 现有的数据。

3. `.jdbc()`：将 `DataFrame` 保存在数据库中

```
.jdbc(url, table, mode=None, properties=None)
```

○ 参数:

- `url`: 一个 JDBC URL, 格式为: `jdbc:subprotocol:subname`
- `table`: 表名
- `mode`: 指定当数据表中已经有数据时, 如何保存。可以为:
  - `'append'`: 追加写入
  - `'overwrite'`: 覆写已有数据
  - `'ignore'`: 忽略本次保存操作 (不保存)
  - `'error'`: 抛出异常 (默认行为)
- `properties`: 一个字典, 用于定义 JDBC 连接参数。通常至少为: `{ 'user' : 'SYSTEM', 'password' : 'mypassword' }`

4. `.json()`: 将 `DataFrame` 保存为 `json` 文件

```
.json(path, mode=None, compression=None, dateFormat=None, timestampFormat=None)
```

示例:

```
df.write.json('./data.json')
```

5. `.orc()`: 将 `DataFrame` 保存为 `ORC` 文件

```
.orc(path, mode=None, partitionBy=None, compression=None)
```

6. `.parquet()`: 将 `DataFrame` 保存为 `Parquet` 格式的文件

```
.parquet(path, mode=None, partitionBy=None, compression=None)
```

7. `.saveAsTable()`: 将 `DataFrame` 保存为 `table`

```
.saveAsTable(name, format=None, mode=None, partitionBy=None, **options)
```

8. `.text()`: 将 `DataFrame` 保存为文本文件

```
.text(path, compression=None)
```

该 `DataFrame` 必须只有一列, 该列必须为字符串。每一行数据将作为文本的一行。

## 五、DataFrame

1. 一个 `DataFrame` 实例代表了基于命名列的分布式数据集。

2. 为了访问 `DataFrame` 的列，有两种方式：

- 通过属性的方式：`df.key`
- 通过字典的方式：`df[key]`。推荐用这种方法，因为它更直观。

它并不支持 `pandas.DataFrame` 中其他的索引，以及各种切片方式

## 5.1 属性

1. `.columns`：以列表的形式返回所有的列名
2. `.dtypes`：以列表的形式返回所有的列的名字和数据类型。形式为：`[(col_name1,col_type1),...]`
3. `.isStreaming`：如果数据集的数据源包含一个或者多个数据流，则返回 `True`
4. `.na`：返回一个 `DataFrameNaFunctions` 对象，用于处理缺失值。
5. `.rdd`：返回 `DataFrame` 底层的 `RDD`（元素类型为 `Row`）
6. `.schema`：返回 `DataFrame` 的 `schema`
7. `.stat`：返回 `DataFrameStatFunctions` 对象，用于统计
8. `.storageLevel`：返回当前的缓存级别
9. `.write`：返回一个 `DataFrameWriter` 对象，它是 `no-streaming DataFrame` 的外部存储接口
10. `.writeStream`：返回一个 `DataStreamWriter` 对象，它是 `streaming DataFrame` 的外部存储接口

## 5.2 方法

### 5.2.1 转换操作

1. 聚合操作：

- `.agg(*exprs)`：在整个 `DataFrame` 开展聚合操作（是 `df.groupBy.agg()` 的快捷方式）

示例：

```
df.agg({"age": "max"}).collect() #在 age 列上聚合
# 结果为: [Row(max(age)=5)]
# 另一种方式:
from pyspark.sql import functions as F
df.agg(F.max(df.age)).collect()
```

- `.filter(condition)`：对行进行过滤。
  - 它是 `where()` 的别名
  - 参数：
    - `condition`：一个 `types.BooleanType` 的 `Column`，或者一个字符串形式的 `SQL` 的表达式
  - 示例：

```
df.filter(df.age > 3).collect()
df.filter("age > 3").collect()
df.where("age = 2").collect()
```

2. 分组：

- `.cube(*cols)`：根据当前 `DataFrame` 的指定列，创建一个多维的 `cube`，从而方便我们之后的聚合过程。
  - 参数：
    - `cols`：指定的列名或者 `Column` 的列表
  - 返回值：一个 `GroupedData` 对象
- `.groupBy(*cols)`：通过指定的列来将 `DataFrame` 分组，从而方便我们之后的聚合过程。
  - 参数：
    - `cols`：指定的列名或者 `Column` 的列表
  - 返回值：一个 `GroupedData` 对象
  - 它是 `groupby` 的别名
- `.rollup(*cols)`：创建一个多维的 `rollup`，从而方便我们之后的聚合过程。
  - 参数：
    - `cols`：指定的列名或者 `Column` 的列表
  - 返回值：一个 `GroupedData` 对象

### 3. 排序：

- `.orderBy(*cols, **kwargs)`：返回一个新的 `DataFrame`，它根据旧的 `DataFrame` 指定列排序
  - 参数：
    - `cols`：一个列名或者 `Column` 的列表，指定了排序列
    - `ascending`：一个布尔值，或者一个布尔值列表。指定了升序还是降序排序
      - 如果是列表，则必须和 `cols` 长度相同
- `.sort(*cols, **kwargs)`：返回一个新的 `DataFrame`，它根据旧的 `DataFrame` 指定列排序
  - 参数：
    - `cols`：一个列名或者 `Column` 的列表，指定了排序列
    - `ascending`：一个布尔值，或者一个布尔值列表。指定了升序还是降序排序
      - 如果是列表，则必须和 `cols` 长度相同
  - 示例：

```
from pyspark.sql.functions import *  
df.sort(df.age.desc())  
df.sort("age", ascending=False)  
df.sort(asc("age"))  
  
df.orderBy(df.age.desc())  
df.orderBy("age", ascending=False)  
df.orderBy(asc("age"))
```

- `.sortWithinPartitions(*cols, **kwargs)`：返回一个新的 `DataFrame`，它根据旧的 `DataFrame` 指定列在每个分区进行排序
  - 参数：
    - `cols`：一个列名或者 `Column` 的列表，指定了排序列

- `ascending`: 一个布尔值, 或者一个布尔值列表。指定了升序还是降序排序
  - 如果是列表, 则必须和 `cols` 长度相同

#### 4. 调整分区:

- `.coalesce(numPartitions)`: 返回一个新的 `DataFrame`, 拥有指定的 `numPartitions` 分区。
  - 只能缩小分区数量, 而无法扩张分区数量。如果 `numPartitions` 比当前的分区数量大, 则新的 `DataFrame` 的分区数与旧 `DataFrame` 相同
  - 它的效果是: 不会混洗数据
  - 参数:
    - `numPartitions`: 目标分区数量
- `.repartition(numPartitions, *cols)`: 返回一个新的 `DataFrame`, 拥有指定的 `numPartitions` 分区。
  - 结果 `DataFrame` 是通过 `hash` 来分区
  - 它可以增加分区数量, 也可以缩小分区数量

#### 5. 集合操作:

- `.crossJoin(other)`: 返回一个新的 `DataFrame`, 它是输入的两个 `DataFrame` 的笛卡儿积
 

可以理解为 `[row1, row2]`, 其中 `row1` 来自于第一个 `DataFrame`, `row2` 来自于第二个 `DataFrame`

  - 参数:
    - `other`: 另一个 `DataFrame` 对象
- `.intersect(other)`: 返回两个 `DataFrame` 的行的交集
  - 参数:
    - `other`: 另一个 `DataFrame` 对象
- `.join(other, on=None, how=None)`: 返回两个 `DataFrame` 的 `join`
  - 参数:
    - `other`: 另一个 `DataFrame` 对象
    - `on`: 指定了在哪些列上执行对齐。可以为字符串或者 `Column` (指定单个列)、也可以为字符串列表或者 `Column` 列表 (指定多个列)
 

注意: 要求两个 `DataFrame` 都存在这些列
    - `how`: 指定 `join` 的方式, 默认为 `'inner'`。可以为: `inner`、`cross`、`outer`、`full`、`full_outer`、`left`、`left_outer`、`right`、`right_outer`、`left_semi`、`left_anti`
- `.subtract(other)`: 返回一个新的 `DataFrame`, 它的行由位于 `self` 中、但是不在 `other` 中的 `Row` 组成。
  - 参数:
    - `other`: 另一个 `DataFrame` 对象
- `.union(other)`: 返回两个 `DataFrame` 的行的并集 (它并不会去重)
  - 它是 `unionAll` 的别名
  - 参数:
    - `other`: 另一个 `DataFrame` 对象

## 6. 统计:

- `.crosstab(col1, col2)`: 统计两列的成对频率。要求每一列的 `distinct` 值数量少于  $10^4$  个。最多返回  $10^6$  对频率。
  - 它是 `DataFrameStatFunctions.crosstab()` 的别名
  - 结果的第一列的列名为, `col1_col2`, 值就是第一列的元素值。后面的列的列名就是第二列元素值, 值就是对应的频率。
  - 参数:
    - `col1,col2`: 列名字符串 (或者 `Column`)
  - 示例:

```
df =pd.DataFrame({'a':[1,3,5], 'b':[2,4,6]})
s_df = spark_session.createDataFrame(df)
s_df.crosstab('a','b').collect()
#结果: [Row(a_b='5', 2=0, 4=0, 6=1), Row(a_b='1', 2=1, 4=0, 6=0), Row(a_b='3',
2=0, 4=1, 6=0)]
```

- `.describe(*cols)`: 计算指定的数值列、字符串列的统计值。
  - 统计结果包括: `count、mean、stddev、min、max`
  - 该函数仅仅用于探索数据规律
  - 参数:
    - `cols`: 列名或者多个列名字符串 (或者 `Column`)。如果未传入任何列名, 则计算所有的数值列、字符串列
- `.freqItems(cols,support=None)`: 寻找指定列中频繁出现的值 (可能有误报)
  - 它是 `DataFrameStatFunctions.freqItems()` 的别名
  - 参数:
    - `cols`: 字符串的列表或者元组, 指定了待考察的列
    - `support`: 指定所谓的频繁的标准 (默认是 1%)。该数值必须大于  $10^{-4}$

## 7. 移除数据:

- `.distinct()`: 返回一个新的 `DataFrame`, 它保留了旧 `DataFrame` 中的 `distinct` 行。  
 即: 根据行来去重
- `.drop(*cols)`: 返回一个新的 `DataFrame`, 它剔除了旧 `DataFrame` 中的指定列。
  - 参数:
    - `cols`: 列名字符串 (或者 `Column`)。如果它在旧 `DataFrame` 中不存在, 也不做任何操作 (也不报错)
- `.dropDuplicates(subset=None)`: 返回一个新的 `DataFrame`, 它剔除了旧 `DataFrame` 中的重复行。  
 它与 `.distinct()` 区别在于: 它仅仅考虑指定的列来判断是否重复行。
  - 参数:
    - `subset`: 列名集合 (或者 `Column` 的集合)。如果为 `None`, 则考虑所有的列。
  - `.drop_duplicates` 是 `.dropDuplicates` 的别名

- `.dropna(how='any', thresh=None, subset=None)`：返回一个新的 `DataFrame`，它剔除了旧 `DataFrame` 中的 `null` 行。
  - 它是 `DataFrameNaFunctions.drop()` 的别名
  - 参数：
    - `how`：指定如何判断 `null` 行的标准。'all'：所有字段都是 `na`，则是空行；'any'：任何字段存在 `na`，则是空行。
    - `thresh`：一个整数。当一行中，非 `null` 的字段数量小于 `thresh` 时，认为是空行。如果该参数设置，则不考虑 `how`
    - `subset`：列名集合，给出了要考察的列。如果为 `None`，则考察所有列。
- `.limit(num)`：返回一个新的 `DataFrame`，它只有旧 `DataFrame` 中的 `num` 行。

## 8. 采样、拆分：

- `.randomSplit(weights, seed=None)`：返回一组新的 `DataFrame`，它是旧 `DataFrame` 的随机拆分
  - 参数：
    - `weights`：一个 `double` 的列表。它给出了每个结果 `DataFrame` 的相对大小。如果列表的数值之和不等于 1.0，则它将被归一化为 1.0
    - `seed`：随机数种子
  - 示例：

```
splits = df.randomSplit([1.0, 2.0], 24)
splits[0].count()
```

- `.sample(withReplacement, fraction, seed=None)`：返回一个新的 `DataFrame`，它是旧 `DataFrame` 的采样
  - 参数：
    - `withReplacement`：如果为 `True`，则可以重复采样；否则是无放回采样
    - `fractions`：新的 `DataFrame` 的期望大小（占旧 `DataFrame` 的比例）。`spark` 并不保证结果刚好满足这个比例（只是一个期望值）
      - 如果 `withReplacement=True`：则表示每个元素期望被选择的次数
      - 如果 `withReplacement=False`：则表示每个元素期望被选择的概率
    - `seed`：随机数生成器的种子
- `.sampleBy(col, fractions, seed=None)`：返回一个新的 `DataFrame`，它是旧 `DataFrame` 的采样，它执行的是无放回的分层采样。分层由 `col` 列指定。
  - 参数：
    - `col`：列名或者 `Column`，它给出了分层的依据
    - `fractions`：一个字典，给出了每个分层抽样的比例。如果某层未指定，则其比例视作 0
  - 示例：

```
sampled = df.sampleBy("key", fractions={0: 0.1, 1: 0.2}, seed=0)
# df['key'] 这一列作为分层依据，0 抽取 10%，1 抽取 20%
```

## 9. 替换：



- `.replace(to_replace, value=None, subset=None)`：返回一组新的 `DataFrame`，它是旧 `DataFrame` 的数值替代结果
  - 它是 `DataFrameNaFunctions.replace()` 的别名
  - 当替换时，`value` 将被类型转换到目标列
  - 参数：
    - `to_replace`：可以为布尔、整数、浮点数、字符串、列表、字典，给出了被替代的值。
      - 如果是字典，则给出了每一列要被替代的值
    - `value`：一个整数、浮点数、字符串、列表。给出了替代值。
    - `subset`：列名的列表。指定要执行替代的列。
- `.fillna(value, subset=None)`：返回一个新的 `DataFrame`，它替换了旧 `DataFrame` 中的 `null` 值。
  - 它是 `DataFrameNaFunctions.fill()` 的别名
  - 参数：
    - `value`：一个整数、浮点数、字符串、或者字典，用于替换 `null` 值。如果是个字典，则忽略 `subset`，字典的键就是列名，指定了该列的 `null` 值被替换的值。
    - `subset`：列名集合，给出了要被替换的列

#### 10. 选取数据：

- `.select(*cols)`：执行一个表达式，将其结果返回为一个 `DataFrame`
  - 参数：
    - `cols`：一个列名的列表，或者 `Column` 表达式。如果列名为 `*`，则扩张到所有的列名
  - 示例：

```
df.select('*')
df.select('name', 'age')
df.select(df.name, (df.age + 10).alias('age'))
```

- `.selectExpr(*expr)`：执行一个 `SQL` 表达式，将其结果返回为一个 `DataFrame`
  - 参数：
    - `expr`：一组 `SQL` 的字符串描述
  - 示例：

```
df.selectExpr("age * 2", "abs(age)")
```

- `.toDF(*cols)`：选取指定的列组成一个新的 `DataFrame`
  - 参数：
    - `cols`：列名字符串的列表
- `.toJSON(use_unicode=True)`：返回一个新的 `DataFrame`，它将旧的 `DataFrame` 转换为 `RDD`（元素为字符串），其中每一行转换为 `json` 字符串。

#### 11. 列操作：

- `.withColumn(colName, col)`：返回一个新的 `DataFrame`，它将旧的 `DataFrame` 增加一列（或者替换现有的列）

- 参数:
  - `colName`: 一个列名, 表示新增的列 (如果是已有的列名, 则是替换的列)
  - `col`: 一个 `Column` 表达式, 表示新的列
- 示例:

```
df.withColumn('age2', df.age + 2)
```

- `.withColumnRenamed(existing, new)`: 返回一个新的 `DataFrame`, 它将旧的 `DataFrame` 的列重命名
  - 参数:
    - `existing`: 一个字符串, 表示现有的列的列名
    - `col`: 一个字符串, 表示新的列名

## 5.2.2 行动操作

### 1. 查看数据:

- `.collect()`: 以 `Row` 的列表的形式返回所有的数据
- `.first()`: 返回第一行 (一个 `Row` 对象)
- `.head(n=None)`: 返回前面的 `n` 行
  - 参数:
    - `n`: 返回行的数量。默认为1
  - 返回值:
    - 如果返回1行, 则是一个 `Row` 对象
    - 如果返回多行, 则是一个 `Row` 的列表
- `.show(n=20, truncate=True)`: 在终端中打印前 `n` 行。
  - 它并不返回结果, 而是 `print` 结果
  - 参数:
    - `n`: 打印的行数
    - `truncate`: 如果为 `True`, 则超过20个字符的字符串被截断。如果为一个数字, 则长度超过它的字符串将被截断。
- `.take(num)`: 以 `Row` 的列表的形式返回开始的 `num` 行数据。
  - 参数:
    - `num`: 返回行的数量
- `.toLocalIterator()`: 返回一个迭代器, 对它迭代的结果就是 `DataFrame` 的每一行数据 (`Row` 对象)

### 2. 统计:

- `.corr(col1, col2, method=None)`: 计算两列的相关系数, 返回一个浮点数。当前仅支持皮尔逊相关系数
  - `DataFrame.corr()` 是 `DataFrameStatFunctions.corr()` 的别名
  - 参数:
    - `col, col2`: 为列的名字字符串 (或者 `Column`) 。
    - `method`: 当前只支持 `'pearson'`

- `.cov(col1,col2)`：计算两列的协方差。
  - `DataFrame.cov()` 是 `DataFrameStatFunctions.cov()` 的别名
  - 参数：
    - `col,col2`：为列的名字字符串（或者 `Column`）
- `.count()`：返回当前 `DataFrame` 有多少行

### 3. 遍历：

- `.foreach(f)`：对 `DataFrame` 中的每一行应用 `f`
  - 它是 `df.rdd.foreach()` 的快捷方式
- `.foreachPartition(f)`：对 `DataFrame` 的每个分区应用 `f`
  - 它是 `df.rdd.foreachPartition()` 的快捷方式
  - 示例：

```
def f(person):
    print(person.name)
df.foreach(f)

def f(people):
    for person in people:
        print(person.name)
df.foreachPartition(f)
```

- `.toPandas()`：将 `DataFrame` 作为 `pandas.DataFrame` 返回
  - 只有当数据较小，可以在驱动器程序中放得下时，才可以用该方法

## 5.2.3 其它方法

### 1. 缓存：

- `.cache()`：使用默认的 `storage level` 缓存 `DataFrame`（缓存级别为：`MEMORY_AND_DISK`）
- `.persist(storageLevel=StorageLevel(True, True, False, False, 1))`：缓存 `DataFrame`
  - 参数：
    - `storageLevel`：缓存级别。默认为 `MEMORY_AND_DISK`
- `.unpersist(blocking=False)`：标记该 `DataFrame` 为未缓存的，并且从内存和磁盘冲移除它的缓存块。
- 2. `.isLocal()`：如果 `collect()` 和 `take()` 方法能本地运行（不需要任何 `executor` 节点），则返回 `True`。否则返回 `False`
- 3. `.printSchema()`：打印 `DataFrame` 的 `schema`
- 4. `.createTempView(name)`：创建一个临时视图，`name` 为视图名字。

临时视图是 `session` 级别的，会随着 `session` 的消失而消失。

- 如果指定的临时视图已存在，则抛出 `TempTableAlreadyExistsException` 异常。
- 参数：
  - `name`：视图名字

- 示例:

```
df.createTempView("people")
df2 = spark_session.sql("select * from people")
```

5. `.createOrReplaceTempView(name)`: 创建一个临时视图, `name` 为视图名字。如果该视图已存在, 则替换它。

- 参数:

- `name`: 视图名字

6. `.createGlobalTempView(name)`: 创建一个全局临时视图, `name` 为视图名字

`spark sql` 中的临时视图是 `session` 级别的, 会随着 `session` 的消失而消失。如果希望一个临时视图跨 `session` 而存在, 则可以建立一个全局临时视图。

- 如果指定的全局临时视图已存在, 则抛出 `TempTableAlreadyExistsException` 异常。
- 全局临时视图存在于系统数据库 `global_temp` 中, 必须加上库名取引用它
- 参数:

- `name`: 视图名字

- 示例:

```
df.createGlobalTempView("people")
spark_session.sql("SELECT * FROM global_temp.people").show()
```

7. `.createOrReplaceGlobalTempView(name)`: 创建一个全局临时视图, `name` 为视图名字。如果该视图已存在, 则替换它。

- 参数:

- `name`: 视图名字

8. `.registerTempTable(name)`: 创建一个临时表, `name` 为表的名字。

在 `spark 2.0` 中被废弃, 推荐使用 `createOrReplaceTempView`

9. `.explain(extended=False)`: 打印 `logical plan` 和 `physical plan`, 用于调试模式

- 参数:

- `extended`: 如果为 `False`, 则仅仅打印 `physical plan`

## 六、Row

1. 一个 `Row` 对象代表了 `DataFrame` 的一行
2. 你可以通过两种方式访问一个 `Row` 对象:
  - 通过属性的方式: `row.key`
  - 通过字典的方式: `row[key]`
3. `key in row` 将在 `Row` 的键上遍历 (而不是值上遍历)
4. 创建 `Row`: 通过关键字参数来创建:

```
row = Row(name="Alice", age=11)
```

- 如果某个参数为 `None`，则必须显式指定，而不能忽略
5. 你可以创建一个 `Row` 作为一个类来使用，它的作用随后用于创建具体的 `Row`

```
Person = Row("name", "age")
p1 = Person("Alice", 11)
```

6. 方法:

- `.asDict(recursive=False)`: 以字典的方式返回该 `Row` 实例。如果 `recursive=True`，则递归的处理元素中包含的 `Row`

## 七、Column

1. `Column` 代表了 `DataFrame` 的一列
2. 有两种创建 `Column` 的方式:
  - 通过 `DataFrame` 的列名来创建:

```
df.colName
df['colName']
```

- 通过 `Column` 表达式来创建:

```
df.colName+1
1/df['colName']
```

### 7.1 方法

1. `.alias(*alias, **kwargs)`: 创建一个新列，它给旧列一个新的名字（或者一组名字，如 `explode` 表达式会返回多列）
  - 它是 `name()` 的别名
- 参数:
  - `alias`: 列的别名
  - `metadata`: 一个字符串，存储在列的 `metadata` 属性中
- 示例:

```
df.select(df.age.alias("age2"))
# 结果为: [Row(age2=2), Row(age2=5)]
df.select(df.age.alias("age3", metadata={'max': 99}))
      .schema['age3'].metadata['max']
# 结果为: 99
```

## 2. 排序:

- `.asc()`: 创建一个新列, 它是旧列的升序排序的结果
- `.desc()`: 创建一个新列, 它是旧列的降序排序的结果

3. `.astype(dataType)`: 创建一个新列, 它是旧列的数值转换的结果

- 它是 `.cast()` 的别名

4. `.between(lowerBound, upperBound)`: 创建一个新列, 它是一个布尔值。如果旧列的数值在 `[lowerBound, upperBound]` (闭区间) 之内, 则为 `True`5. 逻辑操作: 返回一个新列, 是布尔值。 `other` 为另一 `Column`

- `.bitwiseAND(other)`: 二进制逻辑与
- `.bitwiseOR(other)`: 二进制逻辑或
- `.bitwiseXOR(other)`: 二进制逻辑异或

## 6. 元素抽取:

- `.getField(name)`: 返回一个新列, 是旧列的指定字段组成。

此时要求旧列的数据是一个 `StructField` (如 `Row`)

## ■ 参数:

- `name`: 一个字符串, 是字段名

## ■ 示例:

```
df = sc.parallelize([Row(r=Row(a=1, b="b"))]).toDF()
df.select(df.r.getField("b"))
#或者
df.select(df.r.a)
```

- `.getItem(key)`: 返回一个新列, 是旧列的指定位置 (列表), 或者指定键 (字典) 组成。

## ■ 参数:

- `key`: 一个整数或者一个字符串

## ■ 示例:

```
df = sc.parallelize([(1, 2), {"key": "value"}]).toDF(["l", "d"])
df.select(df.l.getItem(0), df.d.getItem("key"))
#或者
df.select(df.l[0], df.d["key"])
```

## 7. 判断:

- `.isNotNull()`: 返回一个新列, 是布尔值。表示旧列的值是否非 `null`
- `.isNull()`: 返回一个新列, 是布尔值。表示旧列的值是否 `null`
- `.isin(*cols)`: 返回一个新列, 是布尔值。表示旧列的值是否在 `cols` 中

## ■ 参数:

- `cols`: 一个列表或者元组

## ■ 示例:

```
df[df.name.isin("Bob", "Mike")]
df[df.age.isin([1, 2, 3])]
```

- `like(other)`: 返回一个新列，是布尔值。表示旧列的值是否 `like other`。它执行的是 SQL 的 `like` 语义

- 参数:
  - `other`: 一个字符串，是 SQL `like` 表达式
- 示例:

```
df.filter(df.name.like('A1%'))
```

- `rlike(other)`: 返回一个新列，是布尔值。表示旧列的值是否 `rrlike other`。它执行的是 SQL 的 `rlike` 语义

- 参数:
  - `other`: 一个字符串，是 SQL `rlike` 表达式

#### 8. 字符串操作: `other` 为一个字符串。

- `.contains(other)`: 返回一个新列，是布尔值。表示是否包含 `other`。
- `.endswith(other)`: 返回一个新列，是布尔值。表示是否以 `other` 结尾。

示例:

```
df.filter(df.name.endswith('ice'))
```

- `.startswith(other)`: 返回一个新列，是布尔值。表示是否以 `other` 开头。
- `.substr(startPos, length)`: 返回一个新列，它是旧列的子串

- 参数:
  - `startPos`: 子串开始位置 (整数或者 `Column`)
  - `length`: 子串长度 (整数或者 `Column`)

#### 9. `.when(condition, value)`: 返回一个新列。

- 对条件进行求值，如果满足条件则返回 `value`，如果不满足:
  - 如果有 `.otherwise()` 调用，则返回 `otherwise` 的结果
  - 如果没有 `.otherwise()` 调用，则返回 `None`
- 参数:
  - `condition`: 一个布尔型的 `Column` 表达式
  - `value`: 一个字面量值，或者一个 `Column` 表达式
- 示例:

```
from pyspark.sql import functions as F
df.select(df.name, F.when(df.age > 4, 1).when(df.age < 3, -1).otherwise(0))
```

- `.otherwise(value)`: `value` 为一个字面量值，或者一个 `Column` 表达式

## 八、GroupedData

1. `GroupedData` 通常由 `DataFrame.groupBy()` 创建，用于分组聚合

### 8.1 方法

1. `.agg(*exprs)`：聚合并以 `DataFrame` 的形式返回聚合的结果
  - 可用的聚合函数包括：`avg`、`max`、`min`、`sum`、`count`
  - 参数：
    - `exprs`：一个字典，键为列名，值为聚合函数字符串。也可以是一个 `Column` 的列表
  - 示例：

```
df.groupBy(df.name).agg({"*": "count"}) #字典
# 或者
from pyspark.sql import functions as F
df.groupBy(df.name).agg(F.min(df.age)) #字典
```

2. 统计：

- `.avg(*cols)`：统计数值列每一组的均值，以 `DataFrame` 的形式返回
  - 它是 `mean()` 的别名
  - 参数：
    - `cols`：列名或者列名的列表
  - 示例：

```
df.groupBy().avg('age')
df.groupBy().avg('age', 'height')
```

- `.count()`：统计每一组的记录数量，以 `DataFrame` 的形式返回
  - `.max(*cols)`：统计数值列每一组的最大值，以 `DataFrame` 的形式返回
    - 参数：
      - `cols`：列名或者列名的列表
  - `.min(*cols)`：统计数值列每一组的最小值，以 `DataFrame` 的形式返回
    - 参数：
      - `cols`：列名或者列名的列表
  - `.sum(*cols)`：统计数值列每一组的和，以 `DataFrame` 的形式返回
    - 参数：
      - `cols`：列名或者列名的列表
3. `.pivot(pivot_col, values=None)`：对指定列进行透视。
    - 参数：



- `pivot_col`：待分析的列的列名
- `values`：待分析的列上，待考察的值的列表。如果为空，则 `spark` 会首先计算 `pivot_col` 的 `distinct` 值
- 示例：

```
df4.groupBy("year").pivot("course", ["dotNET", "Java"]).sum("earnings")
#结果为: [Row(year=2012, dotNET=15000, Java=20000), Row(year=2013, dotNET=48000,
Java=30000)]
# "dotNET", "Java" 是 course 字段的值
```

## 九、functions

1. `pyspark.sql.functions` 模块提供了一些内建的函数，它们用于创建 `Column`
  - 它们通常多有公共的参数 `col`，表示列名或者 `Column`。
  - 它们的返回结果通常都是 `Column`

### 9.1 数学函数

这里的 `col` 都是数值列。

1. `abs(col)`：计算绝对值
2. `acos(col)`：计算 `acos`
3. `cos(col)`：计算 `cos` 值
4. `cosh(col)`：计算 `cosh` 值
5. `asin(col)`：计算 `asin`
6. `atan(col)`：计算 `atan`
7. `atan2(col1,col2)`：计算从直角坐标  $(x,y)$  到极坐标  $(r,\theta)$  的角度  $\theta$
8. `bround(col,scale=0)`：计算四舍五入的结果。如果 `scale>=0`，则使用 `HALF_EVEN` 舍入模式；如果 `scale<0`，则将其舍入到整数部分。
9. `cbrt(col)`：计算立方根
10. `ceil(col)`：计算 `ceiling` 值
11. `floor(col)`：计算 `floor` 值
12. `corr(col1,col2)`：计算两列的皮尔逊相关系数
13. `covar_pop(col1,col2)`：计算两列的总体协方差 (公式中的除数是 `N`)
14. `covar_samp(col1,col2)`：计算两列的样本协方差 (公式中的除数是 `N-1`)
15. `degrees(col)`：将弧度制转换为角度制
16. `radians(col)`：将角度制转换为弧度制
17. `exp(col)`：计算指数： $e^x$
18. `expm1(col)`：计算指数减一： $e^x-1$
19. `fractorial(col)`：计算阶乘

20. `pow(col1,col2)` : 返回幂级数 `col1col2`
21. `hash(*cols)` : 计算指定的一些列的 `hash code` , 返回一个整数列
  - 参数:
    - `cols` : 一组列名或者 `Columns`
22. `hypot(col1,col2)` : 计算  $\sqrt{a^2 + b^2}$  (没有中间产出的上溢出、下溢出) , 返回一个数值列
23. `log(arg1,arg2=None)` : 计算对数。其中第一个参数为底数。如果只有一个参数, 则使用自然底数。
  - 参数:
    - `arg1` : 如果有两个参数, 则它给出了底数。否则就是对它求自然底数。
    - `arg2` : 如果有两个参数, 则对它求对数。
24. `log10(col)` : 计算基于10的对数
25. `log1p(col)` : 计算  $\ln(x + 1)$
26. `log2(col)` : 计算基于2的对数
27. `rand(seed=None)` : 从均匀分布 `U~[0.0,1.0]` 生成一个独立同分布(`i.i.d`) 的随机列
  - 参数:
    - `seed` : 一个整数, 表示随机数种子。
28. `randn(seed=None)` : 从标准正态分布 `N~(0.0,1.0)` 生成一个独立同分布(`i.i.d`) 的随机列
  - 参数:
    - `seed` : 一个整数, 表示随机数种子。
29. `rint(col)` : 返回最接近参数值的整数的 `double` 形式。
30. `round(col,scale=0)` : 返回指定参数的四舍五入形式。  
 如果 `scale>=0` , 则使用 `HALF_UP` 的舍入模式; 否则直接取参数的整数部分。
31. `signum(col)` : 计算正负号
32. `sin(col)` : 计算 `sin`
33. `sinh(col)` : 计算 `sinh`
34. `sqrt(col)` : 计算平方根
35. `tan(col)` : 计算 `tan`
36. `tanh(col)` : 计算 `tanh`
37. `toDegrees(col)` : 废弃。使用 `degrees()` 代替
38. `toRadians(col)` : 废弃, 使用 `radians()` 代替

## 9.2 字符串函数

1. `ascii(col)` : 返回一个数值列, 它是旧列的字符串中的首个字母的 `ascii` 值。其中 `col` 必须是字符串列。
2. `base64(col)` : 返回一个字符串列, 它是旧列 (二进制值) 的 `BASE64` 编码得到的字符串。其中 `col` 必须是二进制列。
3. `bin(col)` : 返回一个字符串列, 它是旧列 (二进制值) 的字符串表示 (如二进制 `1101` 的字符串表示为 `'1101'` ) 其中 `col` 必须是二进制列。

4. `cov(col,fromBase,toBase)`：返回一个字符串列，它是一个数字的字符串表达从 `fromBase` 转换到 `toBase`。

○ 参数：

- `col`：一个字符串列，它是数字的表达。如 `1028`。它的基数由 `fromBase` 给出
- `fromBase`：一个整数，`col` 中字符串的数值的基数。
- `toBase`：一个整数，要转换的数值的基数。

○ 示例：

```
df = spark_session.createDataFrame([("010101",)], ['n'])
df.select(conv(df.n, 2, 16).alias('hex')).collect()
# 结果: [Row(hex=u'15')]
```

5. `concat(*cols)`：创建一个新列，它是指定列的字符串拼接的结果（没有分隔符）。

○ 参数

- `cols`：列名字符串列表，或者 `Column` 列表。要求这些列具有同样的数据类型

6. `concat_ws(sep,*cols)`：创建一个新列，它是指定列的字符串使用指定的分隔符拼接的结果。

○ 参数

- `sep`：一个字符串，表示分隔符
- `cols`：列名字符串列表，或者 `Column` 列表。要求这些列具有同样的数据类型

7. `decode(col,charset)`：从二进制列根据指定字符集来解码成字符串。

○ 参数：

- `col`：一个字符串或者 `Column`，为二进制列
- `charset`：一个字符串，表示字符集。

8. `encode(col,charset)`：把字符串编码成二进制格式。

○ 参数：

- `col`：一个字符串或者 `Column`，为字符串列
- `charset`：一个字符串，表示字符集。

9. `format_number(col,d)`：格式化数值成字符串，根据 `HALF_EVEN` 来四舍五入成 `d` 位的小数。

○ 参数：

- `col`：一个字符串或者 `Column`，为数值列
- `d`：一个整数，格式化表示 `d` 位小数。

10. `format_string(format,*cols)`：返回 `print` 风格的格式化字符串。

○ 参数：

- `format`：`print` 风格的格式化字符串。如 `%s%d`
- `cols`：一组列名或者 `Columns`，用于填充 `format`

11. `hex(col)`：计算指定列的十六进制值（以字符串表示）。

○ 参数：

- `col`：一个字符串或者 `Column`，为字符串列、二进制列、或者整数列

12. `initcap(col)`：将句子中每个单词的首字母大写。

○ 参数：

- `col`：一个字符串或者 `Column`，为字符串列
- 13. `input_file_name()`：为当前的 `spark task` 的文件名创建一个字符串列
- 14. `instr(str,substr)`：给出 `substr` 在 `str` 的首次出现的位置。位置不是从0开始，而是从1开始的。  
 如果 `substr` 不在 `str` 中，则返回 0。  
 如果 `str` 或者 `substr` 为 `null`，则返回 `null`。
  - 参数：
    - `str`：一个字符串或者 `Column`，为字符串列
    - `substr`：一个字符串
- 15. `locate(substr,str,pos=1)`：给出 `substr` 在 `str` 的首次出现的位置（在 `pos` 之后）。位置不是从0开始，而是从1开始的。  
 如果 `substr` 不在 `str` 中，则返回 0。  
 如果 `str` 或者 `substr` 为 `null`，则返回 `null`。
  - 参数：
    - `str`：一个字符串或者 `Column`，为字符串列
    - `substr`：一个字符串
    - `pos`：起始位置（基于0开始）
- 16. `length(col)`：计算字符串或者字节的长度。
  - 参数：
    - `col`：一个字符串或者 `Column`，为字符串列，或者为字节列。
- 17. `levenshtein(left,right)`：计算两个字符串之间的 `Levenshtein` 距离。  
`Levenshtein` 距离：刻画两个字符串之间的差异度。它是从一个字符串修改到另一个字符串时，其中编辑单个字符串（修改、插入、删除）所需要的最少次数。
- 18. `lower(col)`：转换字符串到小写
- 19. `lpad(col,len,pad)`：对字符串，向左填充。
  - 参数：
    - `col`：一个字符串或者 `Column`，为字符串列
    - `len`：预期填充后的字符串长度
    - `pad`：填充字符串
- 20. `ltrim(col)`：裁剪字符串的左侧空格
- 21. `md5(col)`：计算指定列的 `MD5` 值（一个32字符的十六进制字符串）
- 22. `regexp_extract(str,pattern,idx)`：通过正则表达式抽取字符串中指定的子串。
  - 参数：
    - `str`：一个字符串或者 `Column`，为字符串列，表示被抽取的字符串。
    - `pattern`：一个 `Java` 正则表达式子串。
    - `idx`：表示抽取第几个匹配的结果。
  - 返回值：如果未匹配到，则返回空字符串。
- 23. `.regexp_replace(str,pattern,replacement)`：通过正则表达式替换字符串中指定的子串。
  - 参数：
    - `str`：一个字符串或者 `Column`，为字符串列，表示被替换的字符串。

- `pattern`：一个 Java 正则表达式子串。
  - `replacement`：表示替换的子串
  - 返回值：如果未匹配到，则返回空字符串。
24. `repeat(col,n)`：重复一个字符串列 `n` 次，结果返回一个新的字符串列。
- 参数：
    - `col`：一个字符串或者 `Column`，为字符串列
    - `n`：一个整数，表示重复次数
25. `reverse(col)`：翻转一个字符串列，结果返回一个新的字符串列
26. `rpad(col,len,pad)`：向右填充字符串到指定长度。
- 参数：
    - `col`：一个字符串或者 `Column`，为字符串列
    - `len`：指定的长度
    - `pad`：填充字符串
27. `rtrim(col)`：剔除字符串右侧的空格符
28. `sha1(col)`：以16进制字符串的形式返回 SHA-1 的结果
29. `sha2(col,numBites)`：以16进制字符串的形式返回 SHA-2 的结果。
- `numBites` 指定了结果的位数（可以为 244,256,384,512，或者 0 表示 256）
30. `soundex(col)`：返回字符串的 SoundEx 编码
31. `split(str,pattern)`：利用正则表达式拆分字符串。产生一个 `array` 列
- 参数：
    - `str`：一个字符串或者 `Column`，为字符串列
    - `pattern`：一个字符串，表示正则表达式
32. `substring(str,pos,len)`：抽取子串。
- 参数：
    - `str`：一个字符串或者 `Column`，为字符串列，或者字节串列
    - `pos`：抽取的起始位置
    - `len`：抽取的子串长度
  - 返回值：如果 `str` 表示字符串列，则返回的是子字符串。如果 `str` 是字节串列，则返回的是字节子串。
33. `substring_index(str,delim,count)`：抽取子串
- 参数：
    - `str`：一个字符串或者 `Column`，为字符串列
    - `delim`：一个字符串，表示分隔符
    - `count`：指定子串的下标。如果为正数，则从左开始，遇到第 `count` 个 `delim` 时，返回其左侧的内容；如果为负数，则从右开始，遇到第 `abs(count)` 个 `delim` 时，返回其右侧的内容；
  - 示例：

```
df = spark.createDataFrame([('a.b.c.d',)], ['s'])
df.select(substring_index(df.s, '.', 2).alias('s')).collect()
# [Row(s=u'a.b')]
df.select(substring_index(df.s, '.', -3).alias('s')).collect()
# [Row(s=u'b.c.d')]
```

34. `translate(srcCol,matching,replace)`：将 `srcCol` 中指定的字符替换成另外的字符。

○ 参数：

- `srcCol`：一个字符串或者 `Column`，为字符串列
- `matching`：一个字符串。只要 `srcCol` 中的字符串，有任何字符匹配了它，则执行替换
- `replace`：它——对应于 `matching` 中要替换的字符

○ 示例：

```
df = spark.createDataFrame([('translate',)], ['a'])
df.select(translate('a', "rnl", "123").alias('r')).collect()
# [Row(r=u'1a2s3ae')]
# r->1, n->2, l->3, t->空字符
```

35. `trim(col)`：剔除字符串两侧的空格符

36. `unbase64(col)`：对字符串列执行 `BASE64` 编码，并且返回一个二进制列

37. `unhex(col)`：对字符串列执行 `hex` 的逆运算。给定一个十进制数字字符串，将其逆转换为十六进制数字字符串。

38. `upper(col)`：将字符串列转换为大写格式

## 9.3 日期函数

1. `add_months(start, months)`：增加月份

○ 参数：

- `start`：列名或者 `Column` 表达式，指定起始时间
- `months`：指定增加的月份

○ 示例：

```
df = spark_session.createDataFrame([('2015-04-08',)], ['d'])
df.select(add_months(df.d, 1).alias('d'))
# 结果为: [Row(d=datetime.date(2015, 5, 8))]
```

2. `current_data()`：返回当前日期作为一列

3. `current_timestamp()`：返回当前的时间戳作为一列

4. `date_add(start,days)`：增加天数

○ 参数：

- `start`：列名或者 `Column` 表达式，指定起始时间
- `days`：指定增加的天数

5. `date_sub(start,days)` : 减去天数
  - 参数:
    - `start` : 列名或者 `Column` 表达式, 指定起始时间
    - `days` : 指定减去的天数
6. `date_diff(end,start)` : 返回两个日期之间的天数差值
  - 参数:
    - `end` : 列名或者 `Column` 表达式, 指定结束时间。为 `date/timestamp/string`
    - `start` : 列名或者 `Column` 表达式, 指定起始时间。为 `date/timestamp/string`
7. `date_format(date,format)` : 转换 `date/timestamp/string` 到指定格式的字符串。
  - 参数:
    - `date` : 一个 `date/timestamp/string` 列的列名或者 `Column`
    - `format` : 一个字符串, 指定了日期的格式化形式。支持 `java.text.SimpleDateFormat` 的所有格式。
8. `dayofmonth(col)` : 返回日期是当月的第几天 (一个整数)。其中 `col` 为 `date/timestamp/string`
9. `dayofyear(col)` : 返回日期是当年的第几天 (一个整数)。其中 `col` 为 `date/timestamp/string`
10. `from_unixtime(timestamp, format='yyyy-MM-dd HH:mm:ss')` : 转换 `unix` 时间戳到指定格式的字符串。
  - 参数:
    - `timestamp` : 时间戳的列
    - `format` : 时间格式化字符串
11. `from_utc_timestamp(timestamp, tz)` : 转换 `unix` 时间戳到指定时区的日期。
12. `hour(col)` : 从指定时间中抽取小时, 返回一个整数列
  - 参数:
    - `col` : 一个字符串或者 `Column`。是表示时间的字符串列, 或者 `datetime` 列
13. `minute(col)` : 从指定时间中抽取分钟, 返回一个整数列
  - 参数:
    - `col` : 一个字符串或者 `Column`。是表示时间的字符串列, 或者 `datetime` 列
14. `second(col)` : 从指定的日期中抽取秒, 返回一个整数列。
  - 参数:
    - `col` : 一个字符串或者 `Column`。是表示时间的字符串列, 或者 `datetime` 列
15. `month(col)` : 从指定时间中抽取月份, 返回一个整数列
  - 参数:
    - `col` : 一个字符串或者 `Column`。是表示时间的字符串列, 或者 `datetime` 列
16. `quarter(col)` : 从指定时间中抽取季度, 返回一个整数列
  - 参数:
    - `col` : 一个字符串或者 `Column`。是表示时间的字符串列, 或者 `datetime` 列
17. `last_day(date)` : 返回指定日期的当月最后一天 (一个 `datetime.date`)
  - 参数:
    - `date` : 一个字符串或者 `Column`。是表示时间的字符串列, 或者 `datetime` 列
18. `months_between(date1,date2)` : 返回 `date1` 到 `date2` 之间的月份 (一个浮点数)。

也就是 `date1-date2` 的天数的月份数量。如果为正数，表明 `date1 > date2`。

○ 参数：

- `date1`：一个字符串或者 `Column`。是表示时间的字符串列，或者 `datetime` 列
- `date2`：一个字符串或者 `Column`。是表示时间的字符串列，或者 `datetime` 列

19. `next_day(date, dayOfWeek)`：返回指定天数之后的、且匹配 `dayOfWeek` 的那一天。

○ 参数：

- `date1`：一个字符串或者 `Column`。是表示时间的字符串列，或者 `datetime` 列
- `dayOfWeek`：指定星期几。是大小写敏感的，可以为：'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'

20. `to_date(col, format=None)`：转换 `pyspark.sql.types.StringType` 或者 `pyspark.sql.types.TimestampType` 到 `pyspark.pysql.types.DateType`

○ 参数：

- `col`：一个字符串或者 `Column`。是表示时间的字符串列
- `format`：指定的格式。默认为 `yyyy-MM-dd`

21. `to_timestamp(col, format=None)`：将 `StringType, TimestampType` 转换为 `DataType`。

○ 参数：

- `col`：一个字符串或者 `Column`。是表示时间的字符串列
- `format`：指定的格式。默认为 `yyyy-MM-dd HH:mm:ss`

22. `to_utc_timestamp(timestamp, tz)`：根据给定的时区，将 `StringType, TimestampType` 转换为 `DataType`。

○ 参数：

- `col`：一个字符串或者 `Column`。是表示时间的字符串列
- `tz`：一个字符串，表示时区

23. `trunc(date, format)`：裁剪日期到指定的格式。

○ 参数：

- `date`：一个字符串或者 `Column`。是表示时间的字符串列
- `format`：指定的格式。如：'year', 'YYYY', 'yy', 'month', 'mon', 'mm', 'd'

24. `unix_timestamp(timestamp=None, format='yyyy-MM-dd HH:mm:ss')`：给定一个 `unix timestamp`（单位为秒），将其转换为指定格式的字符串。使用默认的时区和默认的 `locale`。

如果转换失败，返回 `null`。

如果 `timestamp=None`，则返回当前的 `timestamp`。

○ 参数：

- `timestamp`：一个 `unix` 时间戳列。
- `format`：指定转换的格式

25. `weekofyear(col)`：返回给定时间是当年的第几周。返回一个整数。

26. `year(col)`：从日期中抽取年份，返回一个整数。

## 9.4 聚合函数

1. `count(col)`：计算每一组的元素的个数。

2. `avg(col)`：计算指定列的均值



3. `approx_count_distinct(col, rsd=None)` : 统计指定列有多少个 `distinct` 值
4. `countDistinct(col,*cols)` : 计算一行或者一组列中的 `distinct value` 的数量。
5. `collect_list(col)` : 返回指定列的元素组成的列表 (不会去重)
6. `collect_set(col)` : 返回指定列的元素组成的集合 (去重)
7. `first(col,ignorenulls=False)` : 返回组内的第一个元素。  
如果 `ignorenulls=True` , 则忽略 `null` 值, 直到第一个非 `null` 值。如果都是 `null` , 则返回 `null` 。  
如果 `ignorenulls=False` , 则返回组内第一个元素(不管是不是 `null` )
8. `last(col,ignorenulls=False)` : 返回组内的最后一个元素。  
如果 `ignorenulls=True` , 则忽略 `null` 值, 直到最后一个非 `null` 值。如果都是 `null` , 则返回 `null` 。  
如果 `ignorenulls=False` , 则返回组内最后一个元素(不管是不是 `null` )
9. `grouping(col)` : 判断 `group by list` 中的指定列是否被聚合。如果被聚合则返回1, 否则返回 0。
10. `grouping_id(*cols)` : 返回 `grouping` 的级别。  
`cols` 必须严格匹配 `grouping columns` , 或者为空 (表示所有的 `grouping columns` )
11. `kurtosis(col)` : 返回一组元素的峰度
12. `max(col)` : 返回组内的最大值。
13. `mean(col)` : 返回组内的均值
14. `min(col)` : 返回组内的最小值
15. `skewness(col)` : 返回组内的偏度
16. `stddev(col)` : 返回组内的样本标准差 (分母除以 `N-1` )
17. `stddev_pop(col)` : 返回组内的总体标准差 (分母除以 `N` )
18. `stddev_samp(col)` : 返回组内的标准差, 与 `stddev` 相同
19. `sum(col)` : 返回组内的和
20. `sumDistinct(col)` : 返回组内 `distinct` 值的和
21. `var_pop(col)` : 返回组内的总体方差。 (分母除以 `N` )
22. `var_samp(col)` : 返回组内的样本方差。 (分母除以 `N-1` )
23. `variance(col)` : 返回组内的总体方差, 与 `var_pop` 相同

## 9.5 逻辑与按位函数

1. `.bitwiseNot(col)` : 返回一个字符串列, 它是旧列的比特级的取反。
2. `isnan(col)` : 返回指定的列是否是 `NaN`
3. `isnull(col)` : 返回指定的列是否为 `null`
4. `shiftLeft(col,numBites)` : 按位左移指定的比特位数。
5. `shiftRight(col,numBites)` : 按位右移指定的比特位数。
6. `shiftRightUnsigned(col,numBites)` : 按位右移指定的比特位数。但是无符号移动。

## 9.6 排序、拷贝

1. `asc(col)` : 返回一个升序排列的 `Column`

2. `desc(col)`：返回一个降序排列的 `Column`
3. `col(col)`：返回值指定列组成的 `Column`
4. `column(col)`：返回值指定列组成的 `Column`

## 9.7 窗口函数

1. `window(timeColumn,windowDuration,slideDuration=None,startTime=None)`：将 `rows` 划分到一个或者多个窗口中（通过 `timestamp` 列）
  - 参数：
    - `timeColumn`：一个时间列，用于划分 `window`。它必须是 `pyspark.sql.types.TimestampType`
    - `windowDuration`：表示时间窗口间隔的字符串。如 `'1 second','1 day 12 hours','2 minutes'`。单位字符串可以为 `'week','day','hour','minute','second','millisecond','microsecond'`。
    - `slideDuration`：表示窗口滑动的间隔，即：下一个窗口移动多少。如果未提供，则窗口为 `tumbling windows`。单位字符串可以为 `'week','day','hour','minute','second','millisecond','microsecond'`。
    - `startTime`：起始时间。它是 1970-01-01 00:00:00 以来的相对偏移时刻。如，你需要在每个小时的 15 分钟开启小时窗口，则它为 `15 minutes`：`12:15-13:15,13:15-14:15,...`
  - 返回值：返回一个称作 `window` 的 `struct`，它包含 `start,end`（一个半开半闭区间）
2. `cume_dist()`：返回一个窗口中的累计分布概率。
3. `dense_rank()`：返回窗口内的排名。（`1,2,...` 表示排名为 `1,2,...`）
 

它和 `rank()` 的区别在于：`dense_rank()` 的排名没有跳跃（比如有3个排名为1，那么下一个排名是2，而不是下一个排名为4）
4. `rank()`：返回窗口内的排名。（`1,2,...` 表示排名为 `1,2,...`）。
 

如有3个排名为1，则下一个排名是 4。
5. `percent_rank()`：返回窗口的相对排名（如：百分比）
6. `lag(col,count=1,default=None)`：返回当前行之前偏移行的值。如果当前行之前的行数小于 `count`，则返回 `default` 值。
  - 参数：
    - `col`：一个字符串或者 `Column`。开窗的列
    - `count`：偏移行
    - `default`：默认值
7. `lead(col,count=1,default=None)`：返回当前行之后偏移行的值。如果当前行之后的行数小于 `count`，则返回 `default` 值。
  - 参数：
    - `col`：一个字符串或者 `Column`。开窗的列
    - `count`：偏移行
    - `default`：默认值
8. `ntile(n)`：返回有序窗口分区中的 `ntile group id`（从 1 到 `n`）
9. `row_number()`：返回一个序列，从 1 开始，到窗口的长度。

## 9.8 其它

1. `array(*cols)`: 创新一个新的 `array` 列。

◦ 参数:

- `cols`: 列名字符串列表, 或者 `Column` 列表。要求这些列具有同样的数据类型

◦ 示例:

```
df.select(array('age', 'age').alias("arr"))
df.select(array([df.age, df.age]).alias("arr"))
```

2. `array_contains(col, value)`: 创建一个新列, 指示 `value` 是否在 `array` 中 (由 `col` 给定)

其中 `col` 必须是 `array` 类型。而 `value` 是一个值, 或者一个 `Column` 或者列名。

◦ 判断逻辑:

- 如果 `array` 为 `null`, 则返回 `null`;
- 如果 `value` 位于 `array` 中, 则返回 `True`;
- 如果 `value` 不在 `array` 中, 则返回 `False`

◦ 示例:

```
df = spark_session.createDataFrame([(["a", "b", "c"],), ([],)], ['data'])
df.select(array_contains(df.data, "a"))
```

3. `create_map(*cols)`: 创建一个 `map` 列。

◦ 参数:

- `cols`: 列名字符串列表, 或者 `Column` 列表。这些列组成了键值对。如  
(`key1,value1,key2,value2,...`)

◦ 示例:

```
df.select(create_map('name', 'age').alias("map")).collect()
#[Row(map={u'Alice': 2}), Row(map={u'Bob': 5})]
```

4. `broadcast(df)`: 标记 `df` 这个 `Dataframe` 足够小, 从而应用于 `broadcast join`

◦ 参数:

- `df`: 一个 `Dataframe` 对象

7. `coalesce(*cols)`: 返回第一个非 `null` 的列组成的 `Column`。如果都为 `null`, 则返回 `null`

◦ 参数:

- `cols`: 列名字符串列表, 或者 `Column` 列表。

8. `crc32(col)`: 计算二进制列的 `CRC32` 校验值。要求 `col` 是二进制列。

9. `explode(col)`: 将一个 `array` 或者 `map` 列拆成多行。要求 `col` 是一个 `array` 或者 `map` 列。

示例:

```
eDF = spark_session.createDataFrame([Row(a=1, intlist=[1,2,3], mapfield={"a": "b"})])
eDF.select(explode(eDF.intlist).alias("anInt")).collect()
# 结果为: [Row(anInt=1), Row(anInt=2), Row(anInt=3)]
eDF.select(explode(eDF.mapfield).alias("key", "value")).show()
#结果为:
# +---+-----+
# |key|value|
# +---+-----+
# | a |    b|
# +---+-----+
```

10. `posexplode(col)`: 对指定 `array` 或者 `map` 中的每个元素, 依据每个位置返回新的一行。

要求 `col` 是一个 `array` 或者 `map` 列。

示例:

```
eDF = spark_session.createDataFrame([Row(a=1, intlist=[1,2,3], mapfield={"a": "b"})])
eDF.select(posexplode(eDF.intlist)).collect()
#结果为: [Row(pos=0, col=1), Row(pos=1, col=2), Row(pos=2, col=3)]
```

11. `expr(str)`: 计算表达式。

o 参数:

- `str`: 一个表达式。如 `length(name)`

12. `from_json(col,schema,options={})`: 解析一个包含 `JSON` 字符串的列。如果遇到无法解析的字符串, 则返回 `null`。

o 参数:

- `col`: 一个字符串列, 字符串是 `json` 格式
- `schema`: 一个 `StructType` (表示解析一个元素), 或者 `StructType` 的 `ArrayType` (表示解析一组元素)
- `options`: 用于控制解析过程。

o 示例:

```
from pyspark.sql.types import *
schema = StructType([StructField("a", IntegerType())])
df = spark_session.createDataFrame([(1, '{"a": 1}')] , ("key", "value"))
df.select(from_json(df.value, schema).alias("json")).collect()
#结果为: [Row(json=Row(a=1))]
```

13. `get_json_object(col,path)`: 从 `json` 字符串中提取指定的字段。如果 `json` 字符串无效, 则返回 `null`。

o 参数:

- `col`: 包含 `json` 格式的字符串的列。
- `path`: `json` 的字段的路径。

o 示例:

```
data = [("1", '{"f1": "value1", "f2": "value2"}'), ("2", '{"f1": "value12"}')]
df = spark_session.createDataFrame(data, ("key", "jstring"))
df.select(df.key, get_json_object(df.jstring, '$.f1').alias("c0"),
          get_json_object(df.jstring, '$.f2').alias("c1")).collect()
# 结果为: [Row(key=u'1', c0=u'value1', c1=u'value2'), Row(key=u'2', c0=u'value12', c1=None)]
```

14. `greatest(*cols)`：返回指定的一堆列中的最大值。要求至少包含2列。

它会跳过 `null` 值。如果都是 `null` 值，则返回 `null`。

15. `least(*cols)`：返回指定的一堆列中的最小值。要求至少包含2列。

它会跳过 `null` 值。如果都是 `null` 值，则返回 `null`。

16. `json_tuple(col,*fields)`：从 `json` 列中抽取字段组成新列（抽取 `n` 个字段，则生成 `n` 列）

◦ 参数：

- `col`：一个 `json` 字符串列
- `fields`：一组字符串，给出了 `json` 中待抽取的字段

17. `lit(col)`：创建一个字面量值的列

18. `monotonically_increasing_id()`：创建一个单调递增的 `id` 列（64位整数）。

它可以确保结果是单调递增的，并且是 `unique` 的，但是不保证是连续的。

它隐含两个假设：

- 假设 `dataframe` 分区数量少于 1 billion
- 假设每个分区的记录数量少于 8 billion

19. `nanvl(col1,col2)`：如果 `col1` 不是 `NaN`，则返回 `col1`；否则返回 `col2`。

要求 `col1` 和 `col2` 都是浮点列（`DoubleType` 或者 `FloatType`）

20. `size(col)`：计算 `array/map` 列的长度（元素个数）。

21. `sort_array(col,asc=True)`：对 `array` 列中的 `array` 进行排序（排序的方式是自然的顺序）

◦ 参数：

- `col`：一个字符串或者 `Column`，指定一个 `array` 列
- `asc`：如果为 `True`，则是升序；否则是降序

22. `spark_partition_id()`：返回一个 `partition ID` 列

该方法产生的结果依赖于数据划分和任务调度，因此是未确定结果的。

23. `struct(*cols)`：创建一个新的 `struct` 列。

◦ 参数：

- `cols`：一个字符串列表（指定了列名），或者一个 `Column` 列表

◦ 示例：

```
df.select(struct('age', 'name').alias("struct")).collect()
# [Row(struct=Row(age=2, name=u'Alice')), Row(struct=Row(age=5, name=u'Bob'))]
```

24. `to_json(col,options={})`: 将包含 `StructType` 或者 `ArrayType` 的 `StructType` 转换为 `json` 字符串。如果遇到不支持的类型, 则抛出异常。

○ 参数:

- `col`: 一个字符串或者 `Column`, 表示待转换的列
- `options`: 转换选项。它支持和 `json datasource` 同样的选项

25. `udf(f=None,returnType=StringType)`: 根据用户定义函数(UDF)来创建一列。

○ 参数:

- `f`: 一个 `python` 函数, 它接受一个参数
- `returnType`: 一个 `pyspark.sql.types.DataType` 类型, 表示 `udf` 的返回类型

○ 示例:

```
from pyspark.sql.types import IntegerType
slen = udf(lambda s: len(s), IntegerType())
df.select(slen("name").alias("slen_name"))
```

26. `when(condition,value)`: 对一系列条件求值, 返回其中匹配的哪个结果。

如果 `Column.otherwise()` 未被调用, 则当未匹配时, 返回 `None`; 如果 `Column.otherwise()` 被调用, 则当未匹配时, 返回 `otherwise()` 的结果。

○ 参数:

- `condition`: 一个布尔列
- `value`: 一个字面量值, 或者一个 `Column`

○ 示例:

```
df.select(when(df['age'] == 2, 3).otherwise(4).alias("age")).collect()
# [Row(age=3), Row(age=4)]
```