Spark

一、核心概念

1. 每个`spark`应用都由一个驱动器程序（`driver program`）来发起集群上的各种并行操作
   - `driver program`包含了应用的`main`函数，并且定义了集群上的分布式数据集，还对这些分布式数据集应用了相关操作
   - `driver program`通过一个`SparkContext`对象来访问`spark`
   - `driver program`一般要管理多个执行器(`executor`)节点
2. `SparkContext`：该对象代表了对计算集群的一个连接
   - 在`pyspark shell`中，当`shell`启动时，已经自动创建了一个`SparkContext`对象，它叫做`sc`。
   - 通常可以用它来创建`RDD`

二、安装和使用

1. 安装步骤：
   - 从`http://spark.apache.org/downloads.html`下载`Pre-built Apache Hadoop xx and later`的版本
   - 解压即可
2. 在`pycharm`中使用`pyspark`：
   - `File->Settings->Project->Project Structure`，选择右侧的`Add Content Root`。
   - 添加`spark`目录下的`python`目录
   - 注意，如果`pycharm`使用了`python3`，则需要在脚本中添加语句：

```
import os
os.environ["PYSPARK_PYTHON"]="python3"
```

三、 pyspark shell

1. `spark`带有交互式的`shell`，可以用于即时数据分析
   - `spark shell`可以与分布式存储在许多机器的内存或者硬盘上的数据进行交互，处理过程由`spark`自动控制
   - `pyspark shell`是`spark shell`的`python`版本
2. 使用`pyspark shell`：进入`spark`的安装目录，然后执行`bin/pyspark`。
   - `ubuntu16.04`中默认使用`python2.7`
   - 如果需要使用`python3`，则使用`export PYSPARK_PYTHON=python3`来导出环境变量
     - 或者在代码中使用`os.environ["PYSPARK_PYTHON"]="python3"`
   - 退出`pyspark shell`：`CTRL+D`
3. 修改`pyspark`日志：在`conf`目录下创建一个`log4j.proerties`的文件。

- 可以直接使用模板`log4j.properties.template`，将`log4j.rootCategory=INFO,console`修改为`log4j.rootCategory=WARN,console`

四、独立应用

1. 独立应用与`pyspark shell`的主要区别在于：你需要自行初始化`SparkContext`，除此之外二者使用的`API`完全相同。
2. 在`python`中，你需要把独立应用写成`python`脚本，然后使用`Spark`自带的`bin/spark-submit`脚本来运行：

   ```
   bin/spark-submit my_script.py
   ```

   `spark-submit`会帮助我们引入`python`程序的`spark`依赖
3. 在独立应用中，通常使用下面方法初始化`SparkContext`：

   ```
   from pyspark import SparkConf, SparkContext
   conf = SparkConf().setMaster('local').setAppName('My App')
   sc = SparkContext(conf = conf)
   ```

   首先创建一个`SparkConf`对象来配置应用，然后基于该`SparkConf`来创建一个`SparkContext`对象。
   - `.setMaster()`给出了集群的`URL`，告诉`spark`如何连接到集群上。这里`'local'`表示让`spark`运行在单机单线程上。
   - `.setAppName()`给出了应用的名字。当连接到一个集群上时，这个值可以帮助你在集群管理器的用户界面上找到你的应用。
4. 关闭`spark`可以调用`SparkContext`的`.stop()`方法，或者直接退出应用（如调用`System.exit(0)`或者`sys.exit()`）
5. 如果需要使用`python3`，则使用`export PYSPARK_PYTHON=python3`来导出环境变量。
   - 或者在代码中使用`os.environ["PYSPARK_PYTHON"]="python3"`

RDD

一、概述

1. `RDD`（弹性分布式数据集`Resilient Distributed Dataset`）：`Spark`中数据的核心抽象。
   - `RDD`是不可变的分布式对象集合
   - 每个`RDD`都被分为多个分区，这些分区运行在集群中的不同节点上
   - `RDD`可以包含`Python、Java、Scala`中任意类型的对象
2. 在`spark`中，`RDD`相关的函数分为三类：
   - 创建`RDD`
   - 转换(`transformation`)已有的`RDD`
   - 执行动作(`action`)来对`RDD`求值

   在这些背后，`spark`自动将`RDD`中的数据分发到集群上，并将`action`并行化执行。
3. `RDD`支持两类操作：
   - 转换操作(`transformation`)：它会从一个`RDD`生成一个新的`RDD`
   - 行动操作(`action`)：它会对`RDD`计算出一个结果，并将结果返回到`driver`程序中（或者把结果存储到外部存储系统，如`HDFS`中）
4. 如果你不知道一个函数时转换操作还是行动操作，你可以考察它的返回值：
   - 如果返回的是`RDD`，则是转换操作。如果返回的是其它数据类型，则是行动操作
5. 转换操作和行动操作的区别在于：行动操作会触发实际的计算。
   - 你可以在任意时候定义新的`RDD`，但是`Spark`只会惰性计算这些`RDD`：只有第一次在一个行动操作中用到时，才会真正计算
     - 所有返回`RDD`的操作都是惰性的（包括读取数据的`sc.textFile()`函数）
   - 在计算`RDD`时，它所有依赖的中间`RDD`也会被求值
     - 通过完整的转换链，`spark`只计算求值过程中需要的那些数据。
6. 默认情况下，`spark`的`RDD`会在你每次对它进行行动操作时重新计算。
   - 如果希望在多个行动操作中重用同一个`RDD`，则可以使用`RDD.persist()`让`spark`把这个`RDD`缓存起来
   - 在第一次对持久化的`RDD`计算之后，`Spark`会把`RDD`的内容保存到内存中（以分区的方式存储到集群的各个机器上）。然后在此后的行动操作中，可以重用这些数据

     ```
     lines = sc.textFile("xxx.md")
     lines.persist()
     lines.count() #计算 lines,此时将 lines 缓存
     lines.first() # 使用缓存的 lines
     ```

   - 之所以默认不缓存`RDD`的计算结果，是因为：`spark`可以直接遍历一遍数据然后计算出结果，没必要浪费存储空间。

7. 每个 `Spark` 程序或者 `shell` 会话都按照如下流程工作：
   - 从外部数据创建输入的 `RDD`
   - 使用诸如 `filter()` 这样的转换操作对 `RDD` 进行转换，以定义新的 `RDD`
   - 对需要被重用的中间结果 `RDD` 执行 `persist()` 操作
   - 使用行动操作（如 `count()` 等）触发一次并行计算，`spark` 会对计算进行优化之后再执行
8. `spark` 中的大部分转化操作和一部分行动操作需要用户传入一个可调用对象。在 `python` 中，有三种方式：`lambda` 表达式、全局定义的函数、局部定义的函数
   - 注意：`python` 可能会把函数所在的对象也序列化之后向外传递。

     当传递的函数是某个对象的成员，或者包含了某个对象中一个字段的引用（如 `self.xxx` 时），`spark` 会把整个对象发送到工作节点上。
     - 如果 `python` 不知道如何序列化该对象，则程序运行失败
     - 如果该序列化对象太大，则传输的数据较多

     ```
     class XXX:
         def is_match(self,s):
           return xxx
         def get_xxx(self,rdd):
           return rdd.filter(self.is_match) # bad! 传递的函数 self.is_match 是对象的成员
         def get_yyy(self,rdd):
           return rdd.filter(lambda x:self._x in x) #bad! 传递的函数包含了对象的成员 self._x
     ```

   - 解决方案是：将你需要的字段从对象中取出，放到一个局部变量中：

     ```
     class XXX:
         def is_match(self,s):
           return xxx
         def get_xxx(self,rdd):
           _is_match = self.is_match
           return rdd.filter(_is_match) # OK
         def get_yyy(self,rdd):
           _x = self._x
           return rdd.filter(lambda x:_x in x) #OK
     ```

9. 在 `python` 中，如果操作对应的 `RDD` 数据类型不正确，则导致运行报错。

二、创建 RDD

2.1 通用RDD

1. 用户可以通过两种方式创建 `RDD`：
   - 读取一个外部数据集。

     ```
     lines = sc.textFile("xxx.md")
     ```

- 在 `driver` 程序中分发 `driver` 程序中的对象集合（如 `list` 或者 `set` ）

```
lines = sc.parallelize([1,3,55,1])
```

  - 这种方式通常仅仅用于开发和测试。在生产环境中，它很少用。因为这种方式需要将整个数据集先放到 `driver` 程序所在的机器的内存中。

2.2 Pair RDD

1. 键值对 `RDD` 的元素通常是一个二元元组（而不是单个值）
   - 键值对 `RDD` 也被称作 `Pair RDD`
   - 键值对 `RDD` 常常用于聚合计算
   - `spark` 为键值对 `RDD` 提供了并行操作各个键、跨节点重新进行数据分组的接口
2. `Pair RDD` 的创建：
   - 通过对常规 `RDD` 执行转化来创建 `Pair RDD`
     - 我们从常规 `RDD` 中抽取某些字段，将该字段作为 `Pair RDD` 的键
   - 对于很多存储键值对的数据格式，当读取该数据时，直接返回由其键值对数据组成的 `Pair RDD`
   - 当数据集已经在内存时，如果数据集由二元元组组成，那么直接调用 `sc.parallelize()` 方法就可以创建 `Pair RDD`

三、转换操作

1. 转换操作(`transformation`) 会从一个 `RDD` 生成一个新的 `RDD`
   - 在这个过程中并不会求值。求值发生在 `action` 操作中
   - 在这个过程中并不会改变输入的 `RDD` （ `RDD` 是不可变的），而是创建并返回一个新的 `RDD`
2. `spark` 会使用谱系图来记录各个 `RDD` 之间的依赖关系
   - 在对 `RDD` 行动操作中，需要这个依赖关系来按需计算每个中间 `RDD`
   - 当持久化的 `RDD` 丢失部分数据时，也需要这个依赖关系来恢复丢失的数据

3.1 通用转换操作

1. 基本转换操作：
   - `.map(f, preservesPartitioning=False)` ：将函数 `f` 作用于当前 `RDD` 的每个元素，返回值构成新的 `RDD` 。
     - `preservesPartitioning` ：如果为 `True` ，则新的 `RDD` 保留旧 `RDD` 的分区
   - `.flatMap(f, preservesPartitioning=False)` ：将函数 `f` 作用于当前 `RDD` 的每个元素，将返回的迭代器的内容构成了新的 `RDD` 。
     - `flatMap` 可以视作：将返回的迭代器扁平化

```
lines = sc.parallelize(['hello world','hi'])
lines.map(lambda line:line.split(" ")) #新的RDD元素为[['hello','world'],['hi',]]
lines.flatMap(lambda line:line.split(" ")) #新的RDD元素为 ['hello','word','hi']
```

- `.mapPartitions(f, preservesPartitioning=False)`：将函数 `f` 作用于当前 `RDD` 的每个分区，将返回的迭代器的内容构成了新的 `RDD`。

  > 这里 `f` 函数的参数是一个集合（表示一个分区的数据）

- `.mapPartitionsWithIndex(f, preservesPartitioning=False)`：将函数 `f` 作用于当前 `RDD` 的每个分区以及分区 `id`，将返回的迭代器的内容构成了新的 `RDD`。
  - 这里 `f` 函数的参数是f分区 `id` 以及一个集合（表示一个分区的数据）

  示例：

  ```
  def f(splitIndex, iterator):
    xxx
  rdd.mapPartitionsWithIndex(f)
  ```

- `.filter(f)`：将函数 `f`（称作过滤器）作用于当前 `RDD` 的每个元素，通过 `f` 的那些元素构成新的 `RDD`
- `.distinct(numPartitions=None)`：返回一个由当前 `RDD` 元素去重之后的结果组成新的 `RDD`。
  - `numPartitions`：指定了新的 `RDD` 的分区数
- `sample(withReplacement, fraction, seed=None)` ：对当前 `RDD` 进行采样，采样结果组成新的 `RDD`
  - `withReplacement`：如果为 `True`，则可以重复采样；否则是无放回采样
  - `fractions`：新的 `RDD` 的期望大小（占旧 `RDD` 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
- `.sortBy(keyfunc, ascending=True, numPartitions=None)`：对当前 `RDD` 进行排序，排序结果组成新的 `RDD`
  - `keyfunc`：自定义的比较函数
  - `ascending`：如果为 `True`，则升序排列
- `.glom()`：返回一个 `RDD`，它将旧 `RDD` 每个分区的元素聚合成一个列表，作为新 `RDD` 的元素
- `.groupBy(f, numPartitions=None, partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：返回一个分组的 `RDD`

  示例：

  ```
  rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
  result = rdd.groupBy(lambda x: x % 2).collect()
  #结果为： [(0, [2, 8]), (1, [1, 1, 3, 5])]
  ```

2. 针对两个 `RDD` 的转换操作：

   尽管 `RDD` 不是集合，但是它也支持数学上的集合操作。注意：这些操作都要求被操作的 `RDD` 是相同数据类型的。

- `.union(other)`：合并两个`RDD`中所有元素，生成一个新的`RDD`。
  - `other`：另一个`RDD`

  该操作并不会检查两个输入`RDD`的重复元素，只是简单的将二者合并（并不会去重）。
- `.intersection(other)`：取两个`RDD`元素的交集，生成一个新的`RDD`。

  该操作会保证结果是去重的，因此它的性能很差。因为它需要通过网络混洗数据来发现重复的元素。
- `.subtract(other, numPartitions=None)`：存在于第一个`RDD`而不存在于第二个`RDD`中的所有元素组成的新的`RDD`。

  该操作也会保证结果是去重的，因此它的性能很差。因为它需要通过网络混洗数据来发现重复的元素。
- `.cartesian(other)`：两个`RDD`的笛卡尔积，生成一个新的`RDD`。

  新`RDD`中的元素是元组`(a,b)`，其中`a`来自于第一个`RDD`，`b`来自于第二个`RDD`
  - 注意：求大规模的`RDD`的笛卡尔积开销巨大
  - 该操作不会保证结果是去重的，它并不需要网络混洗数据。

3. `.keyBy(f)`：创建一个`RDD`，它的元素是元组`(f(x),x)`。

   示例：

   ```
   sc.parallelize(range(2,5)).keyBy(lambda x: x*x)
   # 结果为：[(4, 2), (9, 3), (16, 4)]
   ```

4. `.pipe(command, env=None, checkCode=False)`：返回一个`RDD`，它由外部进程的输出结果组成。
   - 参数：
     - `command`：外部进程命令
     - `env`：环境变量
     - `checkCode`：如果为`True`，则校验进程的返回值
5. `.randomSplit(weights, seed=None)`：返回一组新的`RDD`，它是旧`RDD`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
6. `.zip(other)`：返回一个`Pair RDD`，其中键来自于`self`，值来自于`other`
   - 它假设两个`RDD`拥有同样数量的分区，且每个分区拥有同样数量的元素
7. `.zipWithIndex()`：返回一个`Pair RDD`，其中键来自于`self`，值就是键的索引。
8. `.zipWithUniqueId()`：返回一个`Pair RDD`，其中键来自于`self`，值是一个独一无二的`id`。

   它不会触发一个`spark job`，这是它与`zipWithIndex`的重要区别。

3.2 Pair RDD转换操作

1. `Pair RDD`可以使用所有标准`RDD`上的可用的转换操作
   - 由于`Pair RDD`的元素是二元元组，因此传入的函数应当操作二元元组，而不是独立的元素。
2. 基本转换操作：

- `.keys()`：返回一个新的`RDD`，包含了旧`RDD`每个元素的键
- `.values()`：返回一个新的`RDD`，包含了旧`RDD`每个元素的值
- `.mapValues(f)`：返回一个新的`RDD`，元素为`[K,f(V)]`（保留原来的键不变，通过`f`改变值）。
- `.flatMapValues(f)`：返回一个新的`RDD`，元素为`[K,f(V)]`（保留原来的键不变，通过`f`改变值）。它与`.mapValues(f)`区别见下面的示例：

```
x=sc.parallelize([("a", ["x", "y", "z"]), ("b", ["p", "r"])])
x1=x.flatMapValues(lambda t:t).collect()
# x1: [('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
x2=x.mapValues(lambda t:t).collect()
# x2: [("a", ["x", "y", "z"]), ("b", ["p", "r"])]
```

- `.sortByKey(ascending=True, numPartitions=None, keyfunc=<function <lambda> at 0x7f51f1ab5050>)`：对当前`Pair RDD`进行排序，排序结果组成新的`RDD`
  - `keyfunc`：自定义的比较函数
  - `ascending`：如果为`True`，则升序排列
- `.sampleByKey(withReplacement, fractions, seed=None)`：基于键的采样（即：分层采样）
  - 参数：
    - `withReplacement`：如果为`True`，则是有放回的采样；否则是无放回的采样
    - `fractions`：一个字典，指定了键上的每个取值的采样比例（不同取值之间的采样比例无关，不需要加起来为1）
    - `seed`：随机数种子
- `.subtractByKey(other, numPartitions=None)`：基于键的差集。返回一个新的`RDD`，其中每个`(key,value)`都位于`self`中，且不在`other`中

3. 基于键的聚合操作：

   在常规`RDD`上，`fold()、aggregate()、reduce()`等都是行动操作。在`Pair RDD`上，有类似的一组操作，用于针对相同键的元素进行聚合。这些操作返回`RDD`，因此是转化操作而不是行动操作。

   返回的新`RDD`的键为原来的键，值为针对键的元素聚合的结果。

   - `.reduceByKey(f,numPartitions=None,partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：合并具有相同键的元素。`f`作用于同一个键的那些元素的值。
     - 它为每个键进行并行的规约操作，每个规约操作将键相同的值合并起来
     - 因为数据集中可能有大量的键，因此该操作返回的是一个新的`RDD`：由键，以及对应的规约结果组成
   - `.foldByKey(zeroValue,f,numPartitions=None,partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：通过`f`聚合具有相同键的元素。其中`zeroValue`为零值。参见`.fold()`
   - `.aggregateByKey(zeroValue,seqFunc,combFunc,numPartitions=None,partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：通过`f`聚合具有相同键的元素。其中`zeroValue`为零值。参见`.aggregate()`
   - `.combineByKey(createCombiner,mergeValue,mergeCombiners, numPartitions=None,partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：它是最为常用的基于键的聚合函数，大多数基于键的聚合函数都是用它实现的。

和`aggregate()`一样，`combineByKey()`可以让用户返回与输入数据类型不同的返回值。

你需要提供三个函数：

- `createCombiner(v)`：`v`表示键对应的值。返回一个`C`类型的值（表示累加器）
- `mergeValue(c,v)`：`c`表示当前累加器，`v`表示键对应的值。返回一个`C`类型的值（表示更新后的累加器）
- `mergeCombiners(c1,c2)`：`c1`表示某个分区某个键的累加器，`c2`表示同一个键另一个分区的累加器。返回一个`C`类型的值（表示合并后的累加器）

其工作流程是：遍历分区中的所有元素。考察该元素的键：

- 如果键从未在该分区中出现过，表明这是分区中的一个新的键。则使用`createCombiner()`函数来创建该键对应的累加器的初始值。

  注意：这一过程发生在每个分区中，第一次出现各个键的时候发生。而不仅仅是整个`RDD`中第一次出现一个键时发生。
- 如果键已经在该分区中出现过，则使用`mergeValue()`函数将该键的累加器对应的当前值与这个新的值合并
- 由于每个分区是独立处理的，因此同一个键可以有多个累加器。如果有两个或者更多的分区都有同一个键的累加器，则使用`mergeCombiners()`函数将各个分区的结果合并。

4. 数据分组：

   - `.groupByKey(numPartitions=None, partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：根据键来进行分组。
     - 返回一个新的`RDD`，类型为`[K,Iterable[V]]`，其中`K`为原来`RDD`的键的类型，`V`为原来`RDD`的值的类型。
     - 如果你分组的目的是为了聚合，那么直接使用`reduceByKey、aggregateByKey`性能更好。
   - `.cogroup(other,numPartitions=None)`：它基于`self`和`other`两个`TDD`中的所有的键来进行分组，它提供了为多个`RDD`进行数据分组的方法。
     - 返回一个新的`RDD`，类型为`[K,(Iterable[V],Iterable[W])]`。其中`K`为两个输入`RDD`的键的类型，`V`为原来`self`的值的类型，`W`为`other`的值的类型。
     - 如果某个键只存在于一个输入`RDD`中，另一个输入`RDD`中不存在，则对应的迭代器为空。
     - 它是`groupWith`的别名，但是`groupWith`支持更多的`TDD`来分组。

5. 数据连接：

   数据连接操作的输出`RDD`会包含来自两个输入`RDD`的每一组相对应的记录。输出`RDD`的类型为`[K,(V,W)]`，其中`K`为两个输入`RDD`的键的类型，`V`为原来`self`的值的类型，`W`为`other`的值的类型。

   - `.join(other,numPartitions=None)`：返回一个新的`RDD`，它是两个输入`RDD`的内连接。
   - `.leftOuterJoin(other,numPartitions=None)`：返回一个新的`RDD`，它是两个输入`RDD`的左外连接。
   - `.rightOuterJoin(other,numPartitions=None)`：返回一个新的`RDD`，它是两个输入`RDD`的右外连接。
   - `.fullOuterJoin(other, numPartitions=None)`：执行`right outer join`

四、行动操作

1. 行动操作(`action`)会对`RDD`计算出一个结果，并将结果返回到`driver`程序中（或者把结果存储到外部存储系统，如`HDFS`中）
    - 行动操作会强制执行依赖的中间`RDD`的求值
2. 每当调用一个新的行动操作时，整个`RDD`都会从头开始计算
    - 要避免这种低效的行为，用户可以将中间`RDD`持久化
3. 在调用`sc.textFile()`时，数据并没有读取进来，而是在必要的时候读取。
    - 如果未对读取的结果`RDD`缓存，则该读取操作可能被多次执行
4. `spark`采取惰性求值的原因：通过惰性求值，可以把一些操作合并起来从而简化计算过程。

4.1 通用行动操作

1. `.reduce(f)`：通过`f`来聚合当前`RDD`。
    - `f`操作两个相同元素类型的`RDD`数据，并且返回一个同样类型的新元素。
    - 该行动操作的结果得到一个值（类型与`RDD`中的元素类型相同）
2. `.fold(zeroValue,op)`：通过`op`聚合当前`RDD`

    该操作首先对每个分区中的元素进行聚合（聚合的第一个数由`zeroValue`提供）。然后将分区聚合结果与`zeroValue`再进行聚合。
    - `f`操作两个相同元素类型的`RDD`数据，并且返回一个同样类型的新元素。
    - 该行动操作的结果得到一个值（类型与`RDD`中的元素类型相同）

    > `zeroValue`参与了分区元素聚合过程，也参与了分区聚合结果的再聚合过程。

3. `.aggregate(zeroValue,seqOp,combOp)`：该操作也是聚合当前`RDD`。聚合的步骤为：

    首先以分区为单位，对当前`RDD`执行`seqOp`来进行聚合。聚合的结果不一定与当前`TDD`元素相同类型。

    然后以`zeroValue`为初始值，将分区聚合结果按照`combOp`来聚合（聚合的第一个数由`zeroValue`提供），得到最终的聚合结果。
    - `zeroValue`：`combOp`聚合函数的初始值。类型与最终结果类型相同
    - `seqOp`：分区内的聚合函数，返回类型与`zeroValue`相同
    - `combOp`：分区之间的聚合函数。

    > `zeroValue`参与了分区元素聚合过程，也参与了分区聚合结果的再聚合过程。

    示例：取均值：

    ```
    sum_count = nums.aggregate((0,0),
             (lambda acc,value:(acc[0]+value,acc[1]+1),
             (lambda acc1,acc2:(acc1[0]+acc2[0],acc1[1]+acc2[1]))
    )
    return  sum_count[0]/float(sum_count[1])
    ```

4. 获取`RDD`中的全部或者部分元素：
    - `.collect()`：它将整个`RDD`的内容以列表的形式返回到`driver`程序中。
        - 通常在测试中使用，且当`RDD`内容不大时使用，要求所有结果都能存入单台机器的内存中
        - 它返回元素的顺序可能与你预期的不一致

- `.take(n)`：以列表的形式返回`RDD`中的`n`个元素到`driver`程序中。
  - 它会尽可能的使用尽量少的分区
  - 它返回元素的顺序可能与你预期的不一致
- `.takeOrderd(n,key=None)`：以列表的形式按照指定的顺序返回`RDD`中的`n`个元素到`driver`程序中。
  - 默认情况下，使用数据的降序。你也可以提供`key`参数来指定比较函数
- `.takeSample(withReplacement,num,seed=None)`：以列表的形式返回对`RDD`随机采样的结果。
  - `withReplacement`：如果为`True`，则可以重复采样；否则是无放回采样
  - `num`：预期采样结果的数量。如果是重复采样，则最终采样结果就是`num`。如果是无放回采样，则最终采样结果不能超过`RDD`的大小。
  - `seed`：随机数生成器的种子
- `top(n,key=None)`：获取`RDD`的前`n`个元素。
  - 默认情况下，它使用数据降序的`top n`。你也可以提供`key`参数来指定比较函数。
- `.first()`：获取`RDD`中的第一个元素。

5. 计数：
   - `.count()`：返回`RDD`的元素总数量（不考虑去重）
   - `.countByValue()`：以字典的形式返回`RDD`中，各元素出现的次数。
   - `.histogram(buckets)`：计算分桶
     - 参数：
       - `buckets`：指定如何分桶。
         - 如果是一个序列，则它指定了桶的区间。如`[1,10,20,50]`代表了区间`[1,10) [10,20) [20,50]`（最后一个桶是闭区间）。该序列必须是排序好的，且不能包含重复数字，且至少包含两个数字。
         - 如果是一个数字，则指定了桶的数量。它会自动将数据划分到`min~max`之间的、均匀分布的桶中。它必须大于等于1.
     - 返回值：一个元组`(桶区间序列，桶内元素个数序列)`
     - 示例：

```
rdd = sc.parallelize(range(51))
rdd.histogram(2)
# 结果为 ([0, 25, 50], [25, 26])
rdd.histogram([0, 5, 25, 50])
#结果为 ([0, 5, 25, 50], [5, 20, 26])
```

6. `.foreach(f)`：对当前`RDD`的每个元素执行函数`f`。
   - 它与`.map(f)`不同。`.map`是转换操作，而`.foreach`是行动操作。
   - 通常`.foreach`用于将`RDD`的数据以`json`格式发送到网络服务器上，或者写入到数据库中。
7. `.foreachPartition(f)`：对当前`RDD`的每个分区应用`f`

   示例：

```
def f1(x):
    print(x)
def f2(iterator):
    for x in iterator:
        print(x)
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.foreach(f1)
rdd.foreachPartition(f2)
```

8. 统计方法：
   - `.max(key=None)`：返回`RDD`的最大值。
     - 参数：
       - `key`：对`RDD`中的值进行映射，比较的是`key(x)`之后的结果（但是返回的是`x`而不是映射之后的结果）
   - `.mean()`：返回`RDD`的均值
   - `.min(key=None)`：返回`RDD`的最小值。
     - 参数：
       - `key`：对`RDD`中的值进行映射，比较的是`key(x)`之后的结果（但是返回的是`x`而不是映射之后的结果）
   - `.sampleStdev()`：计算样本标准差
   - `.sampleVariance()`：计算样本方差
   - `.stdev()`：计算标准差。它与样本标准差的区别在于：分母不同
   - `.variance()`：计算方差。它与样本方差的区别在于：分母不同
   - `.sum()`：计算总和

4.2 Pair RDD 行动操作

1. `Pair RDD`可以使用所有标准`RDD`上的可用的行动操作
   - 由于`Pair RDD`的元素是二元元组，因此传入的函数应当操作二元元组，而不是独立的元素。
2. `.countByKey()`：以字典的形式返回每个键的元素数量。
3. `.collectAsMap()`：以字典的形式返回所有的键值对。
4. `.lookup(key)`：以列表的形式返回指定键的所有的值。

五、其他方法和属性

1. 属性：
   - `.context`：返回创建该`RDD`的`SparkContext`
2. `.id()`：返回`RDD`的`ID`
3. `.isEmpty()`：当且仅当`RDD`为空时，它返回`True`；否则返回`False`
4. `.name()`：返回`RDD`的名字

5. `.setName(name)`：设置`RDD`的名字
6. `.stats()`：返回一个`StatCounter`对象，用于计算`RDD`的统计值
7. `.toDebugString()`：返回一个`RDD`的描述字符串，用于调试
8. `.toLocalIterator()`：返回一个迭代器，对它迭代的结果就是对`RDD`的遍历。

六、持久化

1. 如果简单的对`RDD`调用行动操作，则`Spark`每次都会重新计算`RDD`以及它所有的依赖`RDD`。
   - 在迭代算法中，消耗会格外大。因为迭代算法通常会使用同一组数据。
2. 当我们让`spark`持久化存储一个`RDD`时，计算出`RDD`的节点会分别保存它们所求出的分区数据。
   - 如果一个拥有持久化数据的节点发生故障，则`spark`会在需要用到该缓存数据时，重新计算丢失的分区数据。
   - 我们也可以将数据备份到多个节点上，从而增加对数据丢失的鲁棒性。
3. 我们可以为`RDD`选择不同的持久化级别：在`pyspark.StorageLevel`中：
   - `MEMORY_ONLY`：数据缓存在内存中。
     - 内存占用：高；`CPU`时间：低；是否在内存：是；是否在磁盘中：否。
   - `MEMORY_ONLY_SER`：数据经过序列化之后缓存在内存中。
     - 内存占用：低；`CPU`时间：高；是否在内存：是；是否在磁盘中：否。
   - `MEMORY_AND_DISK`：数据缓存在内存和硬盘中。
     - 内存占用：高；`CPU`时间：中等；是否在内存：部分；是否在磁盘中：部分。
     - 如果数据在内存中放不下，则溢写到磁盘上。如果数据在内存中放得下，则缓存到内存中
   - `MEMORY_AND_DISK_SER`：数据经过序列化之后缓存在内存和硬盘中。
     - 内存占用：低；`CPU`时间：高；是否在内存：部分；是否在磁盘中：部分。
     - 如果数据在内存中放不下，则溢写到磁盘上。如果数据在内存中放得下，则缓存到内存中
   - `DISK_ONLY`：数据缓存在磁盘中 。
     - 内存占用：低；`CPU`时间：高；是否在内存：否；是否在磁盘中：是。

   如果在存储级别末尾加上数字`N`，则表示将持久化数据存储为`N`份。如：

   ```
   MEMORY_ONLY_2 #表示对持久化数据存储为 2 份
   ```

   > 在`python`中，总是使用`pickle library`来序列化对象，因此在`python`中可用的存储级别有：
   >
   > `MEMORY_ONLY、MEMORY_ONLY_2、MEMORY_AND_DISK、MEMORY_AND_DISK_2、DISK_ONLY、DISK_ONLY_2`

4. `.persist(storageLevel=StorageLevel(False, True, False, False, 1))`：对当前`RDD`进行持久化
   - 该方法调用时，并不会立即执行持久化，它并不会触发求值，而仅仅是对当前`RDD`做个持久化标记。一旦该`RDD`第一次求值时，才会发生持久化。
   - `.persist()`的默认行为是：将数据以序列化的形式缓存在`JVM`的堆空间中
5. `.cache()`：它也是一种持久化调用。
   - 它等价于`.persist(MEMORY_ONLY)`
   - 它不可设定缓存级别

6. `.unpersist()`：标记当前`RDD`是未缓存的，并且将所有该`RDD`已经缓存的数据从内存、硬盘中清除。
7. 当要缓存的数据太多，内存放不下时，`spark`会利用最近最少使用(`LRU`)的缓存策略，把最老的分区从内存中移除。
   - 对于`MEMORY_ONLY、MEMORY_ONLY_SER`级别：下一次要用到已经被移除的分区时，这些分区就要重新计算
   - 对于`MEMORY_AND_DISK、MEMORY_AND_DISK_SER`级别：被移除的分区都会被写入磁盘。
8. `.getStorageLevel()`：返回当前的缓存级别

七、分区

7.1 基本概念

1. 如果使用可控的分区方式，将经常被一起访问的数据放在同一个节点上，那么可以大大减少应用的通信开销。
   - 通过正确的分区，可以带来明显的性能提升
   - 为分布式数据集选择正确的分区，类似于为传统的数据集选择合适的数据结构
2. 分区并不是对所有应用都是有好处的：如果给定的`RDD`只需要被扫描一次，则我们完全没有必要对其预先进行分区处理。
   - 只有当数据集多次在诸如连接这种基于键的操作中使用时，分区才会有帮助。
3. `Spark`中所有的键值对`RDD`都可以进行分区。系统会根据一个针对键的函数对元素进行分组。
   - `spark`可以确保同一个组的键出现在同一个节点上
4. 许多`spark`操作会自动为结果`RDD`设定分区
   - `sortByKey()`会自动生成范围分区的`RDD`
   - `groupByKey()`会自动生成哈希分区的`RDD`

   其它还有`join()、leftOuterJoin()、rightOuterJoin()、cogroup()、groupWith()、groupByKey()、reduceByKey()、combineByKey()、partitionBy()`，以及`mapValues()`（如果输入`RDD`有分区方式）、`flatMapValues()`（如果输入`RDD`有分区方式）

   对于`map()`操作，由于理论上它可能改变元素的键，因此其结果不会有固定的分区方式。

   对于二元操作，输出数据的分区方式取决于输入`RDD`的分区方式
   - 默认情况下，结果采用哈希分区
   - 若其中一个输入`RDD`已经设置过分区方式，则结果就使用该分区方式
   - 如果两个输入`RDD`都设置过分区方式，则使用第一个输入的分区方式
5. 许多`spark`操作会利用已有的分区信息，如`join()、leftOuterJoin()、rightOuterJoin()、cogroup()、groupWith()、groupByKey()、reduceByKey()、combineByKey()、lookup()`。 这些操作都能从分区中获得收益。
   - 任何需要将数据根据键跨节点进行混洗的操作，都能够从分区中获得好处

7.2 查看分区

1. `.getNumPartitions`属性可以查看`RDD`的分区数

7.3 指定分区

1. 在执行聚合或者分组操作时，可以要求`Spark`使用指定的分区数（即`numPartitions`参数）
    - 如果未指定该参数，则`spark`根据集群的大小会自动推断出一个有意义的默认值
2. 如果我们希望在除了聚合/分组操作之外，也能改变`RDD`的分区。那么`Spark`提供了`.repartition()`方法
    - 它会把数据通过忘了进行混洗，并创建出新的分区集合
    - 该方法是代价比较大的操作，你可以通过`.coalesce()`方法将`RDD`的分区数减少。它是一个代价相对较小的操作。
3. `.repartition(numPartitions)`：返回一个拥有指定分区数量的新的`RDD`
    - 新的分区数量可能比旧分区数增大，也可能减小。
4. `.coalesce(numPartitions,shuffle=False)`：返回一个拥有指定分区数量的新的`RDD`
    - 新的分区数量必须比旧分区数减小
5. `.partitionBy(numPartitions, partitionFunc=<function portable_hash at 0x7f51f1ac0668>)`：返回一个使用指定分区器和分区数量的新的`RDD`
    - 新的分区数量可能比旧分区数增大，也可能减小。
    - 这里`partitionFunc`是分区函数。注意：如果你想让多个`RDD`使用同一个分区方式，则应该使用同一个分区函数对象（如全局函数），而不要给每个`RDD`创建一个新的函数对象。
6. 对于重新调整分区的操作结果，建议对其持久化。
    - 如果未持久化，那么每次用到这个`RDD`时，都会重复地对数据进行分区操作，性能太差

八、混洗

1. `spark`中的某些操作会触发`shuffle`
2. `shuffle`是`spark`重新分配数据的一种机制，它使得这些数据可以跨不同区域进行分组
    - 这通常涉及到在`executor`和驱动器程序之间拷贝数据，使得`shuffle`成为一个复杂的、代价高昂的操作
3. 在`spark`里，特定的操作需要数据不会跨分区分布。如果跨分区分别，则需要混洗。

    以`reduceByKey`操作的过程为例。一个面临的挑战是：一个`key`的所有值不一定在同一个分区里，甚至不一定在同一台机器里。但是它们必须共同被计算。

    为了所有数据都在单个`reduceByKey`的`reduce`任务上运行，我们需要执行一个`all-to-all`操作：它必须从所有分区读取所有的`key`和`key`对应的所有的值，并且跨分区聚集取计算每个`key`的结果。 这个过程叫做`shuffle`。

4. 触发混洗的操作包括：
    - 重分区操作，如`repartition、coalesce`等等
    - `ByKey`操作（除了`countint`之外），如`groupByKey、reduceByKey`等等
    - `join`操作，如`cogroup、join`等等
5. 混洗是一个代价比较高的操作，它涉及到磁盘`IO`，数据序列化，网络`IO`
    - 为了准备混洗操作的数据，`spark`启动了一系列的任务：`map`任务组织数据，`reduce`完成数据的聚合。

        > 这些术语来自于`MapReduce`，与`spark`的`map,reduce`操作没有关系

        - `map`任务的所有结果数据会保存在内存，直到内存不能完全存储为止。然后这些数据将基于目标分区进行排序，并写入到一个单独的文件中

        - `reduce` 任务将读取相关的已排序的数据块
    - 某些混洗操作会大量消耗堆内存空间，因为混洗操作在数据转换前后，需要使用内存中的数据结构对数据进行组织
    - 混洗操作还会在磁盘上生成大量的中间文件。
        - 这么做的好处是：如果`spark` 需要重新计算`RDD` 的血统关系时，混洗操作产生的这些中间文件不需要重新创建

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`pyspqrk.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`pyspqrk.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. `.pqrquet()`：将`DataFrame`保存为`Pqrquet`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，切该列必须为字符串。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 agg 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来访问一个 `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('Al%'))
```

   - `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,θ) 的角度 θ
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. `expml(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 $\mathrm{col1}^{col2}$
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. `toDegreees(col)`：废弃。使用`degrees()`代替
38. `toRadias(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，格式化成表示`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` ：它一一对应于 `matching` 中要替换的字符
    - 示例：

```
df = spark.createDataFrame([('translate',)], ['a'])
df.select(translate('a', "rnlt", "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)`：计算一列或者一组列中的`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。
    - 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`。
    - 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。
    - 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`.
    - 参数：
      - `col`：包含`json`格式的字符串的列。
      - `path`：`json`的字段的路径。
    - 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`Arrytype`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`pyspqrk.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)]
```

累加器、广播变量

1. `spark` 中的累加器(`accumulator`) 和广播变量(`broadcast variable`) 都是共享变量（ 所谓共享，就是在驱动器程序和工作节点之间共享 ）
   - 累加器用于对信息进行聚合
   - 广播变量用于高效的分发较大的对象

一、累加器

1. 在集群中执行代码时，一个难点是：理解变量和方法的范围、生命周期。下面是一个闭包的例子：

```
counter = 0
rdd = sc.parallelize(data)

def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)
print("Counter value: ", counter)
```

   上述代码的行为是不确定的，并且无法按照预期正常工作。

2. 在执行作业时， `spark` 会分解 `RDD` 操作到每个 `executor` 的 `task` 中。在执行之前， `spark` 计算任务的闭包
   - 所谓闭包：指的是 `executor` 要在 `RDD` 上进行计算时，必须对执行节点可见的那些变量和方法
   - 闭包被序列化，并被发送到每个 `executor`
3. 在上述代码中，闭包的变量的副本被发送给每个 `executor` ，当 `counter` 被 `foreach` 函数引用时，它已经不再是驱动器节点的 `counter` 了
   - 虽然驱动器程序中，仍然有一个 `counter` 在内存中；但是对于 `executors` ，它是不可见的。
   - `executor` 看到的只是序列化的闭包的一个副本。所有对 `counter` 的操作都是在 `executor` 的本地进行。
   - 要想正确实现预期目标，则需要使用累加器

1.1 Accumulator

1. 一个累加器(`Accumulator`)变量只支持累加操作
   - 工作节点和驱动器程序对它都可以执行 `+=` 操作，但是只有驱动器程序可以访问它的值。

     > 在工作节点上，累加器对象看起来就像是一个只写的变量

   - 工作节点对它执行的任何累加，都将自动的传播到驱动器程序中。
2. `SparkContext` 的累加器变量只支持基本的数据类型，如 `int、float` 等。
   - 你可以通过 `AccumulatorParam` 来实现自定义的累加器
3. `Accumulator` 的方法：

   - `.add(term)`：向累加器中增加值`term`
4. `Accumulator`的属性：
   - `.value`：获取累加器的值。只可以在驱动器程序中使用
5. 通常使用累加器的流程为：
   - 在驱动器程序中调用`SparkContext.accumulator(init_value)`来创建出带有初始值的累加器
   - 在执行器的代码中使用累加器的`+=`方法或者`.add(term)`方法来增加累加器的值
   - 在驱动器程序中使用累加器的`.value`属性来访问累加器的值

   示例：

```
file=sc.textFile('xxx.txt')
acc=sc.accumulator(0)
def xxx(line):
  global acc #访问全局变量
  if yyy:
    acc+=1
  return zzz
rdd=file.map(xxx)
```

1.2 累加器与容错性

1. `spark`中同一个任务可能被运行多次：
   - 如果工作节点失败了，则`spark`会在另一个节点上重新运行该任务
   - 如果工作节点处理速度比别的节点慢很多，则`spark`也会抢占式的在另一个节点上启动一个投机性的任务副本
   - 甚至有时候`spark`需要重新运行任务来获取缓存中被移出内存的数据
2. 当`spark`同一个任务被运行多次时，任务中的累加器的处理规则：
   - 在行动操作中使用的累加器，`spark`确保每个任务对各累加器修改应用一次
     - 因此：如果想要一个无论在失败还是重新计算时，都绝对可靠的累加器，我们必须将它放在`foreach()`这样的行动操作中
   - 在转化操作中使用的累加器，无法保证只修改应用一次。
     - 转化操作中累加器可能发生不止一次更新
     - 在转化操作中，累加器通常只用于调试目的

二、广播变量

1. 广播变量可以让程序高效的向所有工作节点发送一个较大的只读值
2. `spark`会自动的把闭包中所有引用到的变量都发送到工作节点上。虽然这很方便，但是也很低效。原因有二：
   - 默认的任务发射机制是专门为小任务进行优化的
   - 事实上，你很可能在多个并行操作中使用同一个变量。但是`spark`会为每个操作分别发送。

2.1 Broadcast

1. `Broadcast`变量的`value`中存放着广播的值，该值只会被发送到各节点一次

2. `Broadcast` 的方法：
    - `.destroy()` ：销毁当前 `Broadcast` 变量的所有数据和所有 `metadata` 。
        - 注意：一旦一个 `Boradcast` 变量被销毁，那么它就再也不能被使用
        - 该方法将阻塞直到销毁完成
    - `.dump(value,f)` ：保存 `Broadcast` 变量
    - `.load(path)` ：加载 `Broadcast` 变量
    - `.unpersist(blocking=False)` ：删除 `Broadcast` 变量在 `executor` 上的缓存备份。
        - 如果在此之后，该 `Broadcast` 被使用，则需要从驱动器程序重新发送 `Broadcast` 变量到 `executor`
        - 参数：
            - `blocking` ：如果为 `True` ，则阻塞直到 `unpersist` 完成
3. 属性：
    - `.value` ：返回 `Broadcast` 变量的值
4. 使用 `Broadcast` 的流程：
    - 通过 `SparkContext.broadcast(xx)` 创建一个 `Broadcast` 变量
    - 通过 `.value` 属性访问该对象的值
    - 该变量只会被发送到各节点一次，应该作为只读值来处理（修改这个值并不会影响到其他的节点）

   示例：

```
bd=sc.broadcast(tuple('name','json'))
def xxx(row):
  s=bd.value[0]+row
  return s
rdd=rdd.map(xxx)
```

2.2 广播的优化

1. 当广播一个较大的值时，选择既快又好的序列化格式非常重要
    - 如果序列化对象的时间较长，或者传送花费的时间太久，则这个时间很容易成为性能瓶颈
2. `spark` 中的 `Java API` 和 `Scala API` 默认使用的序列化库为 `Java` 序列化库，它对于除了基本类型的数组以外的任何对象都比较低效。
    - 你可以使用 `spark.serializer` 属性来选择另一个序列化库来优化序列化过程