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