# Spark RDD API 详解

## RDD API

RDD 是弹性分布式数据集的缩写。RDD 是 Spark 系统的主力。作为用户，可以将 RDD 视为单个数据分区集合的句柄，这些分区是某些计算的结果。然而，RDD 实际上不止于此。在集群安装中，单独的数据分区可以位于单独的节点上。使用 RDD 作为句柄可以访问所有分区并使用包含的数据执行计算和转换。每当 RDD 的一部分或整个 RDD 丢失时，系统都能够使用 lineage 信息重建丢失分区的数据。Lineage 是指用于生成当前 RDD 的转换序列。因此，Spark 能够从大多数故障中自动恢复。

Spark 中可用的所有 RDD 都直接或间接地从类 RDD 派生而来。此类带有大量方法，可对关联分区内的数据执行操作。RDD 类是抽象的。每当使用 RDD 时，实际上是在使用 RDD 的协调实现。这些实现必须覆盖一些核心函数才能使 RDD 的行为符合预期。

Spark 最近成为处理大数据的一个非常流行的系统的一个原因是它没有对可以在 RDD 分区中存储哪些数据施加限制。RDD API 已经包含了许多有用的操作。但是，由于 Spark 的创建者必须保持 RDD 的核心 API 足够通用以处理任意数据类型，因此缺少许多便利功能。

基本的 RDD API 将每个数据项视为一个值。但是，用户通常希望使用键值对。因此，Spark 扩展了 RDD 的接口以提供额外的功能（PairRDDFunctions），这些功能显式地作用于键值 对。目前，spark 中有四个可用的 RDD API 扩展。它们如下：

**双RDDF函数：**此扩展包含许多用于聚合数值的有用方法。如果 RDD 的数据项可以隐式转换为 Scala数据类型 双精度值，则它们变得可用。

**对RDDF函数：**当数据项具有两个组件元组结构时，此接口扩展中定义的方法变得可用。Spark 会将第一个元组项（即 tuplename.1）解释为键，将第二项（即 tuplename.2）解释为关联值。

**有序RDDF函数：**如果数据项是键可隐式排序的双组件元组，则在此接口扩展中定义的方法变得可用。

**SequenceFileRDDF 函数：**该扩展包含多种方法，允许用户从 RDD 创建 Hadoop 序列。数据项必须是 PairRDDFunctions 所要求的两个组件键值元组。但是，考虑到元组组件到 Writable 类型的可转换性，还有其他要求。

由于当数据项满足上述要求时，Spark 将自动向用户提供具有扩展功能的方法，因此我们决定严格按字母顺序列出所有可能的可用功能。我们将在函数名后附加以下任一内容，以表明它属于要求数据项符合特定格式或类型的扩展。

**[Double] - 双 RDD 函数**

**[有序] -有序RDDFunctions**

**[Pair] - PairRDDFunctions**

**[SeqFile] - SequenceFileRDDFunctions**

## 聚合

**聚合**功能允许用户应用**2个** 不同的功能降低的RDD。在每个分区内应用第一个reduce 函数以将每个分区内的数据减少为单个结果。第二个reduce函数用于将所有partition的不同reduced结果组合在一起，得到一个最终结果。为分区内缩减和跨分区缩减提供两个单独的缩减功能的能力增加了很多灵活性。

例如，第一个 reduce 函数可以是 max 函数，第二个可以是 sum 函数。用户还指定初始值。

1.初始值应用于两个缩减级别。因此，无论是在分区内缩减还是跨分区缩减。

2.两个归约函数都必须是可交换的和结合的。

3.不要假设分区计算或组合分区的任何执行顺序。

**为什么要使用两种输入数据类型?**

假设使用金属探测器进行考古现场调查。在穿过现场时，根据金属探测器的输出获取重要发现的 GPS 坐标。稍后，打算绘制一张地图图像，使用**聚合**函数突出显示这些位置。

在这种情况下，**zeroValue** 可以是没有高光的区域地图。可能庞大的输入数据集存储为跨多个分区的 GPS 坐标。**seqOp（第一个减速器）**可以将 GPS 坐标转换为地图坐标，并在地图上的相应位置放置一个标记。**combOp（第二个减速器）**将接收这些高光作为部分贴图，并将它们组合成一个最终的输出贴图。

```
def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U
```

**案列1**

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
```

```
// 先用分区标签打印出 RDD 的内容
def myfunc(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}

z.mapPartitionsWithIndex(myfunc).collect
res28: Array[String] = Array([partID:0, val: 1], [partID:0, val: 2], [partID:0, val: 3],
[partID:1, val: 4], [partID:1, val: 5], [partID:1, val: 6])

z.aggregate(0)(math.max(_, _), _ + _)
res40: Int = 9

// 这个例子从初始返回 16 value is 5
// 分区 0 的缩减为 max(5, 1, 2, 3) = 5
// 分区 1 的缩减为 max(5, 4, 5, 6) = 6
// 跨分区的最终缩减为 5 + 5 + 6 = 16
// 注意最终缩减包括初始值
z.aggregate(5)(math.max(_, _), _ + _)
res29: Int = 16


val z = sc.parallelize(List("a","b","c","d","e","f"),2)

//先用分区标签打印出RDD的内容
def myfunc(index: Int, iter: Iterator[(String)]) : Iterator[String] = {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}

z.mapPartitionsWithIndex(myfunc).collect
res31: Array[String] = Array([partID:0, val: a], [partID:0, val: b], [partID:0, val: c],
[partID:1, val: d], [partID:1, val: e], [partID:1, val: f])

z.aggregate("")(_ + _, _+_)
res115: String = abcdef

// 在这里查看初始值“x”如何被应用三次。
// - 每个分区
// - 在第二个 reduce 函数中组合所有分区时一次。
z.aggregate("x")(_ + _, _+_)
res116: String = xxdefxabc

// 下面是一些更高级的例子。有些很难解决。

val z = sc.parallelize(List("12","23","345","4567"),2)
z.aggregate("")((x,y) => math.max(x.length, y.length).toString, (x,y) => x + y)
res141: String = 42


z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)
res142: String = 11


val z = sc.parallelize(List("12","23","345",""),2)
z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)
res143: String = 10
```

**案例2**

```
val z = sc.parallelize(List("12","23","","345"),2)
z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)
res144: String = 11
```

## *aggregateByKey*

除了聚合应用于具有相同键的值之外，它的工作方式与聚合函数类似。同样与聚合函数不同的是，初始值不应用于第二个reduce。

```
def aggregateByKey[U](zeroValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0:
ClassTag[U]): RDD[(K, U)]

def aggregateByKey[U](zeroValue: U, numPartitions: Int)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)
(implicit arg0: ClassTag[U]): RDD[(K, U)]

def aggregateByKey[U](zeroValue: U, partitioner: Partitioner)(seqOp: (U, V) ⇒ U, combOp: (U, U)
⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]
```

**案例3**

```
val pairRDD = sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),("cat", 12), ("dog", 12),
("mouse", 2)), 2)

// 看看分区中有什么
def myfunc(index: Int, iter: Iterator[(String, Int)]) : Iterator[String] = {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}
pairRDD.mapPartitionsWithIndex(myfunc).collect()

res2: Array[String] = Array([partID:0, val: (cat,2)], [partID:0, val: (cat,5)], [partID:0, val:
(mouse,4)], [partID:1, val: (cat,12)], [partID:1, val: (dog,12)], [partID:1, val: (mouse,2)])

pairRDD.aggregateByKey(0)(math.max(_, _), _ + _).collect()
res3: Array[(String, Int)] = Array((dog,12), (cat,17), (mouse,6))

pairRDD.aggregateByKey(100)(math.max(_, _), _ + _).collect
res4: Array[(String, Int)] = Array((dog,100), (cat,200), (mouse,200))
```

## *cartesian*

计算两个 RDD 之间的笛卡尔积（即第一个 RDD 的每个项目与第二个 RDD 的每个项目连接）并将它们作为新的 RDD 返回。（警告：使用此功能时要小心。内存消耗很快就会成为一个问题！）

```
def cartesian[U: ClassTag](other: RDD[U]): RDD[(T, U)]
```

**案例4**

```
val x = sc.parallelize(List(1,2,3,4,5))
val y = sc.parallelize(List(6,7,8,9,10))
x.cartesian(y).collect()
res0: Array[(Int, Int)] = Array(
    (1,6), (1,7), (1,8), (1,9), (1,10),
    (2,6), (2, 7), (2,8), (2,9), (2,10),
    (3,6), (3,7), (3,8), (3,9), (3,10) ,
    (4,6), (5,6), (4,7), (5,7), (4,8),
    (5,8), (4,9), (4,10), ( 5,9), (5,10)
)
```

## *checkpoint*

将在下一次计算 RDD 时创建一个检查点。检查点的 RDD 作为二进制文件存储在检查点目录中，可以使用 Spark 上下文指定。（警告：*Spark* 应用惰性求值。在调用操作之前不会发生检查点。）

重要说明：目录“my_directory_name”应该存在于所有从属设备中。作为替代方案，也可以使用 HDFS 目录 URL。

**定义检查点（）**

**案例5**

```
sc.setCheckpointDir("my_directory_name")
val a = sc.parallelize(1 to 4)
a.checkpoint
a.count
14/02/25 18:13:53 INFO SparkContext: Starting job: count at <console>:15
...
14/02/25 18:13:53 INFO MemoryStore: Block broadcast_5 stored as values to memory (estimated size
115.7 KB, free 296.3 MB)
14/02/25 18:13:53 INFO RDDCheckpointData: Done checkpointing RDD 11 to
file:/home/cloudera/Documents/spark-0.9.0-incubating-bin-cdh4/bin/my_directory_name/65407913-
fdc6-4ec1-82c9-48a1656b95d6/rdd-11, new parent is RDD 12

res23: Long = 4
```

## *coalesce, repartition*

将相关数据合并到给定数量的分区中。

*repartition(numPartitions)* 只是*coalesce(numPartitions, shuffle = true)*的缩写。

```
def coalesce ( numPartitions : Int , shuffle : Boolean = false ): RDD [T]
def repartition ( numPartitions : Int ): RDD [T]
```

**案例6**

```
val y = sc.parallelize(1 to 10, 10)
val z = y.coalesce(2, false)
z.partitions.length()
res9: Int = 2
```

## ***cogroup***, *groupWith*

一组非常强大的函数，允许使用它们的键将最多 3 个键值 RDD 组合在一起。

```
def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]

def cogroup[W](other: RDD[(K, W) )], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]

def cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[ (K, (Iterable[V],
Iterable[W]))]

def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)]): RDD[( K, (Iterable[V],
Iterable[W1], Iterable[W2]))]

def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)], numPartitions: Int): RDD[(K,
(Iterable[V], Iterable[W1], Iterable[W2]))]

def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD [(K, W2)], partitioner: Partitioner):
RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]

def groupWith[W](other: RDD[(K, W) )]): RDD[(K, (Iterable[V], Iterable[W]))]

def groupWith [W1, W2] (other1: RDD [(K, W1)], other2: RDD [(K, W2)]): RDD [(K, (Iterable [V],
IterableW1], Iterable [W2]) )]
```

**案例7**

```
val a = sc.parallelize(List(1, 2, 1, 3), 1)
val b = a.map((_, "b"))
val c = a.map((_, "c"))
b.cogroup(c).collect()
res7: Array[(Int, (Iterable[String], Iterable[String]))] = Array(
    (2,(ArrayBuffer(b),ArrayBuffer(c))),
    (3, (ArrayBuffer(b),ArrayBuffer(c))),
    (1,(ArrayBuffer(b, b),ArrayBuffer(c, c))
)

val d = a.map((_, "d"))
b. cogroup(c, d).collect()
res9: Array[(Int, (Iterable[String], Iterable[String], Iterable[String]))] = Array(
    (2,(ArrayBuffer(b),ArrayBuffer(c), ArrayBuffer(d))),
    (3,(ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))),
    (1,(ArrayBuffer(b, b),ArrayBuffer(c, c),ArrayBuffer(d, d))
)

val x = sc.parallelize(List((1, "apple"), (2, "banana"), (3, "orange"), (4, "kiwi")), 2)
val y = sc.parallelize (List((5, "computer"), (1, "laptop"), (1, "desktop"), (4,"iPad")), 2)
x.cogroup(y).collect()
res23: Array[ (Int, (Iterable[String], Iterable[String]))] = Array(
    (4,(ArrayBuffer(kiwi),ArrayBuffer(iPad))),
    (2,(ArrayBuffer(banana),ArrayBuffer())),
    (3,(ArrayBuffer(orange),ArrayBuffer())),
    (1,(ArrayBuffer(apple),ArrayBuffer(laptop, desktop))),
    (5,(ArrayBuffer(),ArrayBuffer(computer)))
)
```

## *collect, toArray*

将 RDD 转换为 Scala 数组并返回它。如果您提供标准映射函数（即 f = T -> U），它将在将值插入结果数组之前应用。

```
def collect(): Array[T]
def collect[U: ClassTag](f: PartialFunction[T, U]): RDD[U]
def toArray(): Array[T]
```

**案例8**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.collect()
res29: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)
```

## *collectAsMap*

通过一个接一个地应用多个聚合器来组合由两个组件元组组成的 RDD 的值。

```
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) =>
C): RDD[(K, C)]

def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) =>
C, numPartitions: Int): RDD[(K, C)]

def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) =>
C, partitioner: Partitioner, mapSideCombine: Boolean = true, serializerClass: String = null):
RDD[(K, C)]
```

**案例9**

```
val a = sc.parallelize(List("dog","cat","gnu","salmon","rabbit",
                            "turkey","wolf","bear","bee"), 3)

val b = sc.parallelize(List(1,1,2,2,2,1,2,2,2), 3)

val c = b.zip(a)

val d = c.combineByKey(List(_), (x:List[String], y:String) => y :: x, (x:List[String],
    y:List[String]) => x ::: y)

d.collect()

res16: Array[(Int, List[String])] = Array((1,List(cat, dog, turkey)), (2,List(gnu, rabbit,
salmon, bee, bear, wolf)))
```

## *compute*

执行依赖项并计算 RDD 的实际表示。用户不应直接调用此函数。

```
def compute(split: Partition, context: TaskContext): Iterator[T]
```

## *context, sparkContext*

返回用于创建 RDD的*SparkContext*。

```
def compute(split: Partition, context: TaskContext): Iterator[T]
```

**案例10**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.context()
res8: org.apache.spark.SparkContext = org.apache.spark.SparkContext@58c1c2f1
```

## *count*

计算统计个数

```
def count(): Long
```

**案例11**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.count()
res2: Long = 4
```

## *countApprox*

计算不同值的近似数量。对于分布在许多节点上的大型 RDD，此功能可能比其他计数方法执行得更快。参数*relativeSD*控制计算的准确性。

```
def countApproxDistinct(relativeSD: Double = 0.05): Long
```

**案例12**

```
val a = sc.parallelize(1 to 10000, 20)
val b = a++a++a++a++a
b.countApproxDistinct(0.1)
res14: Long = 8224

b.countApproxDistinct(0.05)
res15: Long = 9750

b.countApproxDistinct(0.01)
res16: Long = 9947

b.countApproxDistinct(0.001)
res0: Long = 10000
```

## *countApproxDistinctByKey*

类似于*countApproxDistinct*，但计算每个不同键的不同值的近似数量。因此，RDD 必须由两部分元组组成。对于分布在许多节点上的大型 RDD，此功能可能比其他计数方法执行得更快。参数*relativeSD*控制计算的准确性。

```
def countApproxDistinctByKey(relativeSD: Double = 0.05): RDD[(K, Long)]

def countApproxDistinctByKey(relativeSD: Double, numPartitions: Int): RDD[(K, Long)]

def countApproxDistinctByKey(relativeSD: Double, partitioner: Partitioner): RDD[(K,Long)]
```

**案例13**

```
val a = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
val b = sc.parallelize(a.takeSample(true, 10000, 0), 20)
val c = sc.parallelize(1 to b.count().toInt, 20)
val d = b.zip(c)
d.countApproxDistinctByKey(0.1).collect()
res15: Array[(String, Long)] = Array((Rat, 2567), (Cat,3357), (Dog,2414), (Gnu,2494))

d.countApproxDistinctByKey(0.01).collect()
res16: Array[(String, Long)] = Array((Rat,2555), (Cat) ,2455), (Dog,2425), (Gnu,2513))

d.countApproxDistinctByKey(0.001).collect()
res0: Array[(String, Long)] = Array((Rat,2562), (Cat,2464), (狗,2451), (Gnu,2521))
```

## *countByKey*

与 count 非常相似，但分别为每个不同的键计算由两个组件元组组成的 RDD 的值。

```
def countByKey(): Map[K, Long]
```

**案例14**

```
val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"), (3, "Dog")), 2)
c.countByKey()
res3: scala .collection.Map[Int,Long] = Map(3 -> 3, 5 -> 1)
```

## *countByKeyApprox*

```
def countByKeyApprox(timeout: Long, confidence: Double = 0.95): PartialResult[Map[K,
BoundedDouble]]
```

## *countByValue*

返回一个映射，其中包含 RDD 的所有唯一值及其各自的出现次数。 （警告：此操作最终将在单个减速器中聚合信息。）

```
def countByValue(): Map[T, Long]
```

**案例15**

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))
b.countByValue()
res27: scala.collection .Map[Int,Long] = Map(5 -> 1, 8 -> 1, 3 -> 1,
                                             6 -> 1, 1 -> 6, 2 -> 3, 4 -> 2, 7 -> 1)
```

## *countByValueApprox*

```
def countByValueApprox(timeout: Long, confidence: Double = 0.95): PartialResult[Map[T,
BoundedDouble]]
```

## *dependencies*

返回此 RDD 所依赖的 RDD。

```
final def dependencies: Seq[Dependency[_]]
```

**案例16**

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[32] at parallelize at <console>:12
b.dependencies.length()
Int = 0

b.map(a => a).dependencies.length()
res40: Int = 1

b.cartesian(a).dependencies.length()
res41: Int = 2

b.cartesian(a).dependencies
res42: Seq[org.apache.spark.Dependency[_]] =
List(org.apache.spark.rdd.CartesianRDD$$anon$1@576ddaaa,
org.apache.spark.rdd.CartesianRDD$$anon$2@6d2efbbd)
```

## *distinct*

返回一个新的 RDD，其中每个唯一值只包含一次。

```
def distinct(): RDD[T]
def distinct(numPartitions: Int): RDD[T]
```

**案例17**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.distinct.collect()
res6: Array[String] = Array(Dog, Gnu, Cat, Rat)

val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
a.distinct(2).partitions.length()
res16: Int = 2

a.distinct(3).partitions.length()
res17: Int = 3
```

## *first*

查找 RDD 的第一个数据项并返回它。

```
def first(): T
```

**案例18**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.first()
res1: String = Gnu
```

## *filter*

为 RDD 的每个数据项计算一个布尔函数，并将函数返回*true* 的项 放入结果 RDD。

```
def filter(f: T => Boolean): RDD[T]
```

**案例19**

```
val a = sc.parallelize(1 to 10, 3)
val b = a.filter(_ % 2 == 0)
b.collect()
res3: Array[Int] = Array(2, 4, 6, 8, 10)
```

Collect 使用函数对象的*isDefinedAt*属性来确定测试函数是否与每个数据项兼容。只有通过此测试（=过滤器）的数据项才会使用函数对象进行映射。

**具有偏函数的混合数据**

```
val a = sc.parallelize(List("cat", "horse", 4.0, 3.5, 2, "dog"))
a.collect({case a: Int    => "is integer" |
           case b: String => "is string" }).collect
res17: Array[String] = Array(is string, is string, is integer, is string)

val myfunc: PartialFunction[Any, Any] = {
  case a: Int    => "is integer" |
  case b: String => "is string"
}

myfunc.isDefinedAt("")
res21: Boolean = true

myfunc.isDefinedAt(1)
res22: Boolean = true

myfunc.isDefinedAt(1.5)
res23: Boolean = false
```

因为它只检查类型本身！如果在这个类型上使用操作，必须明确声明你想要的类型而不是任何类型。否则编译器（显然）不知道它应该生成什么字节码：

```
val myfunc2: PartialFunction[Any, Any] = {case x if (x < 4) => "x"}
<console>:10: error: value < is not a member of Any

val myfunc2: PartialFunction[Int, Any] = {case x if (x < 4) => "x"}
myfunc2: PartialFunction[Int,Any] = <function1>
```

## *filterByRange*

返回仅包含指定键范围内的项目的 RDD。从我们的测试来看，这似乎仅在您的数据在键值对中并且已经按键排序时才有效。

```
def filterByRange (lower: K, upper: K): RDD [P]
```

**案例20**

```
val randRDD = sc.parallelize(List( (2,"cat"), (6, "mouse"),(7, "cup"), (3, "book"), (4, "tv"),
(1, "screen"), (5, "heater")), 3)
val sortedRDD = randRDD.sortByKey()

sortedRDD.filterByRange(1, 3).collect()
res66: Array[(Int, String)] = Array((1,screen), (2,cat), (3,book))
```

## *filterWith*

这是*filter*的扩展版本。它需要两个函数参数。第一个参数必须符合*Int -> T*并且每个分区执行一次。它将分区索引转换为类型*T*。第二个函数看起来像*(U, T) -> Boolean*。*T*是转换后的分区索引，*U*是来自 RDD 的数据项。最后，该函数必须返回 true 或 false（即应用过滤器）。

```
def filterWith[A: ClassTag](constructA: Int => A)(p: (T, A) => Boolean): RDD[T]
```

**案例21**

```
val a = sc.parallelize(1 to 9, 3)
val b = a.filterWith(i => i)((x,i) => x % 2 == 0 || i % 2 == 0)
b.collect()
res37: Array[Int] = Array(1, 2, 3, 4, 6, 7, 8, 9)

val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 5)
a.filterWith(x=> x)((a, b) =>  b == 0).collect()
res30: Array[Int] = Array(1, 2)

a.filterWith(x=> x)((a, b) =>  a % (b+1) == 0).collect()
res33: Array[Int] = Array(1, 2, 4, 6, 8, 10)

a.filterWith(x=> x.toString)((a, b) =>  b == "2").collect()
res34: Array[Int] = Array(5, 6)
```

## *flatMap*

类似于*map*，但允许在 map 函数中发出多个项目。

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]
```

**案例22**

```
val a = sc.parallelize(1 to 10, 5)
a.flatMap(1 to _).collect()
res47: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2,
3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

sc.parallelize(List(1, 2, 3), 2).flatMap(x => List(x, x, x)).collect()
res85: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3 )

// 下面的程序生成列表中项目的随机数量的副本（最多 10 个）。
val x = sc.parallelize(1 to 10, 3)
x.flatMap(List.fill(scala.util.Random.nextInt(10))(_)).collect()

res1: Array[Int] = Array(1, 2, 3, 3, 3,
                         4, 4, 4, 4, 4, 4, 4, 4, 4,
                         5, 5,
                         6, 6, 6, 6, 6 , 6, 6, 6,
                         7, 7, 7,
                         8, 8, 8, 8, 8, 8, 8, 8,
                         9, 9, 9, 9, 9,
                         10, 10, 10, 10, 10, 10 , 10, 10
                        )
```

## *flatMapValues*

非常相似*mapValues*，但折叠映射期间的值的固有结构。

```
def flatMapValues[U](f: V => TraversableOnce[U]): RDD[(K, U)]
```

**案例23**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length , x))
b.flatMapValues("x" + _ + "x").collect
res6: Array[(Int, Char)] = Array((3,x), (3,d), (3,o) , (3,g), (3,x),
                                 (5,x), (5,t), (5,i), (5,g), (5,e), (5,r), ( 5,x),
                                 (4,x), (4,l), (4,i), (4,o), (4,n), (4,x),
                                 (3,x), (3, c), (3,a), (3,t), (3,x),
                                 (7,x), (7,p), (7,a), (7,n), (7,t) , (7,h),
                                 (7,e), (7,r), (7,x),
                                 (5,x), (5,e), (5,a), (5,g), ( 5,l), (5,e), (5,x)
                                )
```

## *flatMapWith*

与*flatMap*类似，但允许从 flatMap 函数内部访问分区索引或分区索引的派生词。

```
def flatMapWith[A: ClassTag, U: ClassTag](constructA: Int => A, preservesPartitioning: Boolean =
false)(f: (T, A) => Seq[U]): RDD[U]
```

**案例24**

```
val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9), 3)
a.flatMapWith(x => x, true)((x, y) => List (y, x)).collect()
res58: Array[Int] = Array(0, 1, 0, 2,
                          0, 3, 1, 4,
                          1, 5, 1, 6,
                          2, 7, 2, 8, 2 , 9
                         )
```

## *fold*

聚合每个分区的值。每个分区内的聚合变量用*zeroValue*初始化。

```
def fold(zeroValue: T)(op: (T, T) => T): T
```

**案例25**

```
val a = sc.parallelize(List(1,2,3), 3)
a.fold(0)(_ + _)
res59: Int = 6
```

## *foldByKey*

与*fold*非常相似，但分别为 RDD 的每个键执行折叠。此函数仅在 RDD 由两部分元组组成时才可用。

```
def foldByKey (zeroValue: V) (func: (V, V) => V): RDD [(K, V)]
def foldByKey (zeroValue: V, numPartitions: Int) (func: (V, V) => V ): RDD [(K, V)]
def foldByKey (zeroValue: V, partitioner: Partitioner) (func: (V, V) => V): RDD [(K, V)]
```

**案例26**

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = a.map(x => (x.length, x))
b.foldByKey("")(_ + _).collect()
res84: Array[(Int, String)] = Array((3,dogcatowlgnuant)

val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.foldByKey("")(_ + _).collect()
res85: Array[(Int, String)] = Array((4,lion), (3,dogcat), (7,panther), (5,tigereagle))
```

## *foreach*

为每个分区执行一个无参数函数。通过迭代器参数提供对分区中包含的数据项的访问。

```
def foreachPartition(f: Iterator[T] => Unit)
```

**案例27**

```
val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3)
b.foreachPartition(x => println(x.reduce(_ + _)))
6
15
24
```

## *foreachWith*

为每个分区执行一个无参数函数。通过迭代器参数提供对分区中包含的数据项的访问。

```
def foreachWith[A: ClassTag](constructA: Int => A)(f: (T, A) => Unit)
```

**案例28**

```
val a = sc.parallelize(1 to 9, 3)
a.foreachWith(i => i)((x,i) => if (x % 2 == 1 && i % 2 == 0) println(x) )
1
3
7
9
```

## *fullOuterJoin*

在两个配对的 RDD 之间执行全外连接。

```
def fullOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], Option[W]))]

def fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))]

def fullOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Option[V],
Option[W]))]
```

**案例29**

```
val pairRDD1 = sc.parallelize(List( ("cat",2), ("cat", 5), ("book", 4),("cat", 12)))
val pairRDD2 = sc.parallelize(List( ("cat",2), ("cup", 5), ("mouse", 4),("cat", 12)))
pairRDD1.fullOuterJoin(pairRDD2).collect()

res5: Array[(String, (Option[Int], Option[Int]))] = Array((book,(Some(4),None)),
           (mouse,(None,Some(4))),
           (cup,(None,Some(5))),
           (cat,(Some(2),Some(2))),
           (cat,(Some(2),Some(12))),
           (cat,(Some(5),Some(2))),
           (cat,(Some(5),Some(12))),
           (cat,(Some(12),Some(2))),
           (cat,(Some(12),Some(12)))
      )
```

## *generator, setGenerator*

允许在打印依赖图时设置附加到 RDD 名称末尾的字符串。

```
@transient
var generator
def setGenerator(_generator: String)
```

## *getCheckpointFile*

返回检查点文件的路径，如果 RDD 尚未被检查点，则返回 null。

```
def getCheckpointFile: Option[String]
```

**案例30**

```
sc.setCheckpointDir("/home/cloudera/Documents")
val a = sc.parallelize(1 to 500, 5)
val b = a++a++a++a++a
b.getCheckpointFile()
res49: Option[String] = None

b.checkpoint()
b.getCheckpointFile
res54: Option[String] = None

b.collect()
b.getCheckpointFile()
res57: Option[String] = Some(file:/home/cloudera/Documents/cb978ffb-a346-4820-b3ba-
d56580787b20/rdd-40)
```

## *preferredLocations*

返回此 RDD**首选**的主机。特定主机的实际偏好取决于各种假设。

```
final def preferredLocations(split: Partition): Seq[String]
```

## *getStorageLevel*

检索当前设置的 RDD 存储级别。如果 RDD 还没有设置存储级别，这只能用于分配新的存储级别。下面的示例显示了当您尝试重新分配存储级别时将遇到的错误。

```
def getStorageLevel
```

**案例31**

```
val a = sc.parallelize(1 to 100000, 2)
a.persist(org.apache.spark.storage.StorageLevel.DISK_ONLY)
a.getStorageLevel.description()
String = Disk Serialized 1x Replicated

a.cache()
java.lang.UnsupportedOperationException: Cannot change storage level of an RDD after it was
already assigned a level
```

## *glom*

组装一个包含分区所有元素的数组并将其嵌入到 RDD 中。每个返回的数组包含一个分区的内容。

```
def glom(): RDD[Array[T]]
```

**案例32**

```
val a = sc.parallelize(1 to 100, 3)
a.glom.collect
res8: Array[Array[Int]] = Array(Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                                      11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                                      21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                                      31, 32, 33),
                                Array(34, 35, 36, 37, 38, 39, 40,
                                      41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                                      51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
                                      61, 62, 63, 64, 65, 66),
                                Array(67, 68, 69, 70, 71, 72, 73, 74, 75,
                                      76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
                                      86, 87, 88, 89, 90, 91, 92, 93, 94,
                                      95, 96, 97, 98, 99, 100)
                               )
```

## *groupBy*

```
def groupBy[K: ClassTag](f: T => K): RDD[(K, Iterable[T])]
def groupBy[K: ClassTag](f: T => K, numPartitions: Int): RDD[( K, Iterable[T])]
def groupBy[K: ClassTag](f: T => K, p: Partitioner): RDD[(K, Iterable[T])]
```

**案例33**

```
val a = sc.parallelize(1 to 9, 3)
a.groupBy(x => { if (x % 2 == 0) "even" else "odd" }).collect()
res42: Array[(String, Seq[Int])] = Array((even,ArrayBuffer(2, 4, 6, 8)), (odd,ArrayBuffer(1, 3,
5, 7, 9)))

val a = sc.parallelize(1 to 9, 3)
def myfunc(a: Int) : Int =
{
  a % 2
}
```

```
a.groupBy(myfunc).collect()
res3: Array[(Int, Seq[Int])] = Array((0,ArrayBuffer(2, 4, 6, 8)), (1,ArrayBuffer(1, 3, 5, 7,
9)))

val a = sc.parallelize(1 to 9, 3)
def myfunc(a: Int) : Int =
{
  a % 2
}
a.groupBy(x => myfunc(x), 3).collect
a.groupBy(myfunc(_), 1).collect
res7: Array[(Int, Seq[Int])] = Array((0,ArrayBuffer(2, 4, 6, 8)), (1,ArrayBuffer(1, 3, 5, 7,
9)))

import org.apache.spark.Partitioner
class MyPartitioner extends Partitioner {
def numPartitions: Int = 2
def getPartition(key: Any): Int =
{
    key match
    {
      case null     => 0
      case key: Int => key          % numPartitions
      case _        => key.hashCode % numPartitions
    }
  }
  override def equals(other: Any): Boolean =
  {
    other match
    {
      case h: MyPartitioner => true
      case _                => false
    }
  }
}
val a = sc.parallelize(1 to 9, 3)
val p = new MyPartitioner()
val b = a.groupBy((x:Int) => { x }, p)
val c = b.mapWith(i => i)((a, b) => (b, a))
c.collect()
res42: Array[(Int, (Int, Seq[Int]))] = Array((0,(4,ArrayBuffer(4))), (0,(2,ArrayBuffer(2))), (0,
(6,ArrayBuffer(6))), (0,(8,ArrayBuffer(8))), (1,(9,ArrayBuffer(9))), (1,(3,ArrayBuffer(3))), (1,
(1,ArrayBuffer(1))), (1,(7,ArrayBuffer(7))), (1,(5,ArrayBuffer(5))))
```

## *groupByKey*

与*groupBy*非常相似，但不是提供函数，而是每对的关键组件将自动呈现给分区器。

```
def groupByKey(): RDD[(K, Iterable[V])]
def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]
def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]
```

**案例34**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
val b = a.keyBy(_.length)
b.groupByKey.collect()
res11: Array[(Int, Seq[String])] = Array((4,ArrayBuffer(lion)), (6,ArrayBuffer(spider)),
(3,ArrayBuffer(dog, cat)), (5,ArrayBuffer(tiger, eagle)))
```

## *histogram*

这些函数采用双精度的 RDD，并根据用户通过双精度值数组提供的自定义桶边界创建具有均匀间距（桶数等于 *bucketCount*）或任意间距的直方图。两种变体的结果类型略有不同，第一个函数将返回由两个数组组成的元组。第一个数组包含计算的桶边界值，第二个数组包含相应的值计数（即直方图）。该函数的第二个变体将仅以整数数组的形式返回直方图。

```
def histogram(bucketCount: Int): Pair[Array[Double], Array[Long]]
def histogram(buckets:Array[Double],evenBuckets:Boolean = false):Array[Long]
```

**案例34**

```
val a = sc.parallelize(List(1.1, 1.2, 1.3, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 9.0), 3)
a.histogram(5)
res11: (Array[Double], Array[Long]) = (Array(1.1, 2.68, 4.26, 5.84, 7.42, 9.0),Array(5, 0, 0, 1,
4))

val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0,
8.9, 5.5), 3)
a.histogram(6)
res18: (Array[Double], Array[Long]) = (Array(1.0, 2.5, 4.0, 5.5, 7.0, 8.5, 10.0),Array(6, 0, 1,
1, 3, 4))
```

**自定义间距示例**

```
val a = sc.parallelize(List(1.1, 1.2, 1.3, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 9.0), 3)
a.histogram(Array(0.0, 3.0, 8.0))
res14: Array[Long ] = Array(5, 3)

val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0,
5.9, 5) .
a.histogram(Array(0.0, 5.0, 10.0))
res1: Array[Long] = Array(6, 9)

a.histogram(Array(0.0, 5.0, 10.0, 15.0))
res1: Array[Long] = Array( 6, 8, 1)
```

## *id*

检索由其设备上下文分配给 RDD 的 ID。

```
val id: Int
```

**案例35**

```
val y = sc.parallelize(1 to 10, 10)
y.id
res16: Int = 19
```

## *intersection*

返回两个 RDD 中相同的元素。

```
def intersection(other: RDD[T], numPartitions: Int): RDD[T]

def intersection(other: RDD[T], partitioner: Partitioner)(implicit ord: Ordering[T] = null):
RDD[T]

def intersection(other: RDD[T]): RDD[T]
```

**案例36**

```
val x = sc.parallelize(1 to 20)
val y = sc.parallelize(10 to 30)
val z = x.intersection(y)

z.collect()
res74: Array[Int] = Array(16, 12, 20, 13, 17, 14, 18, 10, 19, 15, 11)
```

## *isCheckpointed*

指示 RDD 是否已被检查点。只有在真正创建检查点后，该标志才会升起。

```
def isCheckpointed: Boolean
```

**案例37**

```
sc.setCheckpointDir("/home/cloudera/Documents")
c.isCheckpointed()
res6: Boolean = false

c.checkpoint()
c.isCheckpointed()
res8: Boolean = false

c.collect()
c.isCheckpointed()
res9: Boolean = true
```

## *iterator*

返回此 RDD 分区的兼容迭代器对象。永远不应直接调用此函数。

```
final def iterator(split: Partition, context: TaskContext): Iterator[T]
```

## *join*

使用两个键值 RDD 执行内部连接。

```
def join [W] (other: RDD [(K, W)]): RDD [(K, (V, W))]
def join [W] (other: RDD [(K, W)], numPartitions: Int ): RDD [(K, (V, W))]
def join [W] (other: RDD [(K, W)], partitioner: Partitioner): RDD [(K, (V, W))]
```

**案例38**

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"),
3)
val d = c.keyBy(_.length)
b.join(d).collect()

res0: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)), (6,
(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)),
(3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,
(rat,bee)))
```

## *keyBy*

通过对每个数据项应用一个函数来构造二元元组（键值对）。函数的结果成为键，原始数据项成为新创建的元组的值。

```
def keyBy[K](f: T => K): RDD[(K, T)]
```

**案列39**

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
b.collect()
res26: Array[(Int, String)] = Array((3,dog), (6,salmon), (6,salmon), (3,rat), (8,elephant))
```

## *keys*

从所有包含的元组中提取键并在新的 RDD 中返回它们。

```
def keys: RDD[K]
```

**案列40**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.keys.collect()
res2: Array[Int] = Array(3, 5, 4, 3, 7, 5)
```

## *leftOuterJoin*

使用两个键值 RDD 执行左外连接。

```
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]

def leftOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, Option[W]))]

def leftOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, Option[W]))]
```

**案列41**

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"),
3)
val d = c.keyBy(_.length)
b.leftOuterJoin(d).collect()

res1: Array[(Int, (String, Option[String]))] = Array((6,(salmon,Some(salmon))), (6,
(salmon,Some(rabbit))), (6,(salmon,Some(turkey))), (6,(salmon,Some(salmon))), (6,
(salmon,Some(rabbit))), (6,(salmon,Some(turkey))), (3,(dog,Some(dog))), (3,(dog,Some(cat))), (3,
(dog,Some(gnu))), (3,(dog,Some(bee))), (3,(rat,Some(dog))), (3,(rat,Some(cat))), (3,
(rat,Some(gnu))), (3,(rat,Some(bee))), (8,(elephant,None)))
```

## *lookup*

RDD 扫描所有与提供的值匹配的键，并将它们的值作为 Scala 序列返回。

```
def lookup(key: K): Seq[V]
```

**案列42**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.lookup(5)
res0: Seq[String] = WrappedArray(tiger, eagle)
```

## *map*

在 RDD 的每一项上应用一个转换函数，并将结果作为一个新的 RDD 返回。

```
def map[U: ClassTag](f: T => U): RDD[U]
```

**案列43**

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.map(_.length)
val c = a.zip( b)
c.collect()
res0: Array[(String, Int)] = Array((dog,3), (salmon,6), (salmon,6), (rat,3), (elephant,8))
```

## *mapPartitions*

这是一个专门的映射，每个分区只调用一次。各个分区的全部内容可通过输入参数 (*Iterarator[T]*)作为连续值流提供。自定义函数必须返回另一个*Iterator[U]*。组合的结果迭代器会自动转换为新的 RDD。请注意，由于我们选择了分区，以下结果中缺少元组 (3,4) 和 (6,7)。

```
def mapPartitions[U: ClassTag](f: Iterator[T] => Iterator[U], preservesPartitioning: Boolean =
false): RDD[U]
```

**案列44**

```
val a = sc.parallelize(1 to 9, 3)
def myfunc[T](iter: Iterator[T]) : Iterator[(T, T)] = {
  var res = List[(T, T)]()
  var pre = iter.next()
  while (iter.hasNext)
  {
    val cur = iter.next;
    res .::= (pre, cur)
    pre = cur;
  }
  res.iterator()
}
a.mapPartitions(myfunc).collect()
res0: Array[(Int, Int)] = Array((2,3), (1,2), (5,6), (4,5), (8,9), (7,8))
```

**案列45**

```
val x = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9,10), 3)
def myfunc(iter: Iterator[Int]) : Iterator[Int] = {
  var res = List[Int]()
  while (iter.hasNext) {
    val cur = iter.next;
    res = res ::: List.fill(scala.util.Random.nextInt(10))(cur)
  }
  res.iterator
}
x.mapPartitions(myfunc).collect
// 有些数字根本没有输出。这是因为为其生成的随机数为零。
res8: Array[Int] = Array(1,
                         2, 2, 2, 2,
                         3, 3, 3, 3, 3, 3, 3, 3, 3,
                         4, 4, 4, 4, 4, 4, 4 ,
                         5,
                         7, 7, 7,
                         9, 9,
                         10
                        )
```

**使用 flatmap**

```
val x = sc.parallelize(1 to 10, 3)
x.flatMap(List.fill(scala.util.Random.nextInt(10))(_)).collect

res1: Array[Int] = Array(1, 2) ,
    3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4,
    5, 5,
    6, 6, 6, 6, 6, 6, 6, 6,
    7, 7, 7 ,
    8, 8, 8, 8, 8, 8, 8, 8,
    9, 9, 9, 9, 9,
    10, 10, 10, 10, 10, 10, 10, 10
)
```

## *mapPartitionsWithContext* （已弃用和开发者 *API*）

*mapPartitions*类似，但允许访问有关映射器内处理状态的信息。

```
def mapPartitionsWithContext[U: ClassTag](f: (TaskContext, Iterator[T]) => Iterator[U],
preservesPartitioning: Boolean = false): RDD[U]
```

**案列46**

```
val a = sc.parallelize(1 to 9, 3)
import org.apache.spark.TaskContext
def myfunc(tc: TaskContext, iter: Iterator[Int]) : Iterator[Int] = {
  tc.addOnCompleteCallback(() => println(
    "Partition: "     + tc.partitionId +
    ", AttemptID: "   + tc.attemptId ))

  iter.toList.filter(_ % 2 == 0).iterator()
}
a.mapPartitionsWithContext(myfunc).collect()

14/04/01 23:05:48 INFO SparkContext: Starting job: collect at <console>:20
```

```
...
14/04/01 23:05:48 INFO Executor: Running task ID 0
Partition: 0, AttemptID: 0, Interrupted: false
...
14/04/01 23:05:48 INFO Executor: Running task ID 1
14/04/01 23:05:48 INFO TaskSetManager: Finished TID 0 in 470 ms on localhost (progress: 0/3)
...
14/04/01 23:05:48 INFO Executor: Running task ID 2
14/04/01 23:05:48 INFO TaskSetManager: Finished TID 1 in 23 ms on localhost (progress: 1/3)
14/04/01 23:05:48 INFO DAGScheduler: Completed ResultTask(0, 1)

?
res0: Array[Int] = Array(2, 6, 4, 8)
```

## *mapPartitionsWithIndex*

类似于*mapPartitions*，但需要两个参数。第一个参数是分区的索引，第二个参数是遍历该分区内所有项目的迭代器。输出是一个迭代器，其中包含应用函数编码的任何转换后的项目列表。

```
def mapPartitionsWithIndex[U: ClassTag](f: (Int, Iterator[T]) => Iterator[U],
preservesPartitioning: Boolean = false): RDD[U]
```

**案例47**

```
val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 3)
def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String ] = {
  iter.map(x => index + "," + x)
}
x.mapPartitionsWithIndex(myfunc).collect()
res10: Array[String] = Array(0,1, 0,2, 0,3, 1,4, 1,5, 1,6, 2,7, 2,8, 2,9, 2,10)
```

## *mapPartitionsWithSplit*

此方法已在 API 中标记为已弃用。

```
def mapPartitionsWithSplit[U: ClassTag](f: (Int, Iterator[T]) => Iterator[U],
preservesPartitioning: Boolean = false): RDD[U]
```

## *mapValues*

获取由两分量元组组成的 RDD 的值，并应用提供的函数来转换每个值。然后，它使用键和转换后的值形成新的二元元组，并将它们存储在新的 RDD 中。

```
def mapValues[U](f: V => U): RDD[(K, U)]
```

**案例48**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length , x))
b.mapValues("x" + _ + "x").collect()
res5: Array[(Int, String)] = Array((3,xdogx), (5,xtigerx), (4,xlionx) , (3,xcatx),
(7,xpantherx), (5,xeaglex))
```

## *mapWith*

这是*map*的扩展版本。它需要两个函数参数。第一个参数必须符合*Int -> T*并且每个分区执行一次。它将分区索引映射到一些转换后的*T*类型的分区索引。这是每个分区执行一次某种初始化代码的好地方。就像创建一个随机数生成器对象。第二个函数必须符合*(U, T) -> U*。*T*是转换后的分区索引，*U* 是 RDD 的数据项。最后，该函数必须返回*U*类型的转换数据项。

```
def mapWith[A: ClassTag, U: ClassTag](constructA: Int => A, preservesPartitioning: Boolean =
false)(f: (T, A) => U): RDD[U]
```

**案例49**

```
// 生成 9个小于 1000 的随机数。
val x = sc.parallelize(1 to 9, 3)
x.mapWith(a => new scala.util.Random)((x, r) => r.nextInt(1000) )).collect
res0: Array[Int] = Array(940, 51, 779, 742, 757, 982, 35, 800, 15)

val a = sc.parallelize(1 to 9, 3)
val b = a.mapWith ("Index:" + _)((a, b) => ("Value:" + a, b))
b.collect
res0: Array[(String, String)] = Array((Value:1,Index: 0), (Value:2,Index:0), (Value:3,Index:0),
(Value:4,Index:1), (Value:5,Index:1), (Value:6,Index: 1), (Value:7,Index:2), (Value:8,Index:2),
(Value:9,Index:2)
```

## *max*

返回 RDD**列表变体中**最大的元素

```
def max()(implicit ord: Ordering[T]): T
```

**案例50**

```
val y = sc.parallelize(10 to 30)
y.max()
res75: Int = 30

val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion"), (18, "cat")))
a.max()
res6: (Int, String) = (18,cat)
```

## ***mean*** *, meanApprox*

调用*stats*并提取均值分量。在某些情况下，函数的近似版本可以更快地完成。但是，它以准确性换取速度。

```
def mean(): Double
def meanApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]
```

**案例51**

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0,
8.9, 5.5), 3)
a.mean()
res0: Double = 5.3
```

## ***min***

返回 RDD**列表变体**中的最小元素

```
def min()(implicit ord: Ordering[T]): T
```

**案例52**

```
val y = sc.parallelize(10 to 30)
y.min()
res75: Int = 10


val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion") ), (8, "cat")))
a.min()
res4: (Int, String) = (3,tiger)
```

## ***name, setName***

允许使用自定义名称标记 RDD。

```
@transient var name: String
def setName(_name: String)
```

**案例53**

```
val y = sc.parallelize(1 to 10, 10)
y.name()
res13: String = null
y.setName("Fancy RDD Name")
y.name()
res15: String = Fancy RDD Name
```

## *partitionBy*

使用其键重新分区为键值 RDD。分区器实现可以作为第一个参数提供。

```
def partitionBy (partitions: Partitions): RDD [(K, V)]
```

## *partitioner*

指定一个指向默认分区器的函数指针，该函数指针将用于 *groupBy*、*subtract*、*reduceByKey*（来自 *PairedRDDFunctions*）等函数。

```
@transient val partitioner: Option[Partitioner]
```

## *partitions*

返回与此 RDD 关联的分区对象的数组。

```
final def partitions: Array[Partition]
```

**案例54**

```
val b = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
b.partitions()
res48: Array[org.apache.spark.Partition] =
Array(org.apache.spark.rdd.ParallelCollectionPartition@18aa,
org.apache.spark.rdd.ParallelCollectionPartition@18ab)
```

## *persist, cache*

这些功能可以用来调整一个RDD的存储级别。在释放内存时，Spark 将使用存储级别标识符来决定应该保留哪些分区。无参数变体*persist()*和*cache()*只是*persist(StorageLevel.MEMORY_ONLY)* 的缩写。（警告：一旦存储级别已经改变，不能再改了！）

```
def cache():RDD[T]
def persist():RDD[T]
def persist(newLevel:StorageLevel):RDD[T]
```

**案例55**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.getStorageLevel()
res0: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1)
c.cache()
c.getStorageLevel()
res2: org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true, 1)
```

## *pipe*

获取每个分区的 RDD 数据并通过 stdin 将其发送到 shell 命令。命令的结果输出被捕获并作为字符串值的 RDD 返回。

```
def pipe(command: String): RDD[String]

def pipe(command: String, env: Map[String, String]): RDD[String]

def pipe(command: Seq[String], env: Map[String, String] ] = Map(), printPipeContext: (String =>
Unit) => Unit = null, printRDDElement: (T, String => Unit) => Unit = null): RDD[String]
```

**案例56**

```
val a = sc.parallelize(1 to 9, 3)
a.pipe("head -n 1").collect()
res2: Array[String] = Array(1, 4, 7)
```

## *randomSplit*

根据权重数组将 RDD 随机拆分为多个较小的 RDD，权重数组指定分配给每个较小 RDD 的总数据元素的百分比。请注意，每个较小 RDD 的实际大小仅大约等于权重数组指定的百分比。下面的第二个示例显示每个较小 RDD 中的项目数与权重数组不完全匹配。可以指定随机可选种子。此函数可用于将数据拆分为机器学习的训练集和测试集。

```
def randomSplit(weights: Array[Double], seed: Long = Utils.random.nextLong): Array[RDD[T]]
```

**案例57**

```
val y = sc.parallelize(1 to 10)
val splits = y.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)
training.collect()
res:85 Array [Int] = Array(1, 4, 5, 6, 8, 10)
test.collect()
res86: Array[Int] = Array(2, 3, 7, 9)

val y = sc.parallelize(1 to 10)
val splits = y.randomSplit(Array(0.1, 0.3, 0.6))

val rdd1 = splits(0)
val rdd2 = splits(1)
val rdd3 = splits(2)

rdd1.collect()
res87: Array[Int] = Array(4, 10)
rdd2.collect()
res88: Array[Int] = Array(1, 3, 5, 8)
rdd3.collect()
res91: Array[Int] = Array(2, 6, 7, 9)
```

## *reduce*

该函数提供了 Spark 中众所周知的*reduce*功能。请注意，任何功能 *°F*您提供的，应该是为了生成可重复的结果交换。

```
def reduce(f: (T, T) => T): T
```

**案例58**

```
val a = sc.parallelize(1 to 100, 3)
a.reduce(_ + _)
res41: Int = 5050
```

## *reduceByKey, reduceByKeyLocally, reduceByKeyToDriver*

该函数提供了 Spark中众所周知的*reduce*功能。请注意，任何功能 *°F*您提供的，应该是为了生成可重复的结果交换。

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]
def reduceByKeyLocally(func: (V, V) => V): Map[K, V]
def reduceByKeyToDriver (func: (V, V) => V): Map[K, V]
```

**案例59**

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = a.map(x => (x.length, x))
b.reduceByKey(_ + _).collect()
res86: Array[(Int, String)] = Array((3,dogcatowlgnuant))

val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.reduceByKey(_ + _).collect()
res87: Array[(Int, String)] = Array((4,lion), (3,dogcat), (7,panther), (5,tigereagle))
```

## *repartition*

此函数将分区数更改为 numPartitions 参数**Listing Variants**指定的数

```
def repartition(numPartitions: Int)(隐式ord: Ordering[T] = null): RDD[T]
```

**案例60**

```
val rdd = sc.parallelize (List (1, 2, 10, 4, 5, 2, 1, 1, 1), 3)
rdd.partitions.length()
res2: Int = 3
val rdd2 = rdd.repartition (5)
rdd2 .partitions.length()
res6: Int = 5
```

## *repartitionAndSortWithinPartitions*

根据给定的分区器对 RDD 进行重新分区，并在每个结果分区内，按键对记录进行排序。

```
def repartitionAndSortWithinPartitions (partitioner: Partitioner): RDD [(K, V)]
```

**案例61**

```
// 首先我们将进行未排序的范围分区
val randRDD = sc.parallelize(List( (2,"cat"), (6, "mouse"),(7, "cup"), (3, "book" "), (4, "tv"),
(1, "screen"), (5, "heater")), 3)
val rPartitioner = new org.apache.spark.RangePartitioner(3, randRDD)
val partitioned = randRDD .partitionBy(rPartitioner)
def myfunc(index: Int, iter: Iterator[(Int, String)]) : Iterator[String] = {
  iter.map(x => "[partID:" + index + ", val: " + x + "]")
}
partitioned.mapPartitionsWithIndex(myfunc).collect

res0: Array[String] = Array([partID:0, val: (2,cat)], [partID:0, val: (3,book) )], [partID:0,
val: (1,screen)], [partID:1, val: (4,tv)], [partID:1, val: (5,heater)],[partID: 2, val: (6,
mouse)], [partID: 2, val: (7, cup)])


// 现在让重新分区，但这次让它排序
val partitioned = randRDD.repartitionAndSortWithinPartitions(rPartitioner)
def myfunc(index: Int, iter: Iterator[(Int, String)]) : Iterator[String] = {
  iter.map(x => "[partID:" + index + ", val: " + x + "]")
}
partitioned.mapPartitionsWithIndex(myfunc).collect

res1: Array[String] = Array([partID:0, val: (1, screen)], [partID:0, val: (2,cat)], [partID:0,
val: (3,book)], [partID:1, val: (4,tv)], [partID:1 , val: (5,heater)], [partID:2, val:
(6,mouse)], [partID:2, val: (7,cup)])
```

## *rightOuterJoin*

使用两个键值 RDD 执行右外连接。请注意，密钥必须通常具有可比性才能使其正常工作。

```
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]

def rightOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], W))]

def rightOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Option[ V], W))]
```

**案例62**

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"),
3)
val d = c.keyBy(_.length)
b.rightOuterJoin(d).collect()

res2: Array[(Int, (Option[String], String))] = Array((6,(Some(salmon),salmon)), (6,
(Some(salmon),rabbit)), (6,(Some(salmon),turkey)), (6,(Some(salmon),salmon)), (6,
(Some(salmon),rabbit)), (6,(Some(salmon),turkey)), (3,(Some(dog),dog)), (3,(Some(dog),cat)), (3,
(Some(dog),gnu)), (3,(Some(dog),bee)), (3,(Some(rat),dog)), (3,(Some(rat),cat)), (3,
(Some(rat),gnu)), (3,(Some(rat),bee)), (4,(None,wolf)), (4,(None,bear)))
```

## *sample*

随机选择一个 RDD 的一部分项目，并在一个新的 RDD 中返回它们。

```
def sample(withReplacement: Boolean, fraction: Double, seed: Int): RDD[T]
```

**案例63**

```
val a = sc.parallelize(1 to 10000, 3)
a.sample(false, 0.1, 0).count()
res24: Long = 960

a.sample(true, 0.3, 0).count()
res25: Long = 2888

a.sample(true, 0.3, 13).count()
res26: Long = 2985
```

## *sampleByKey*

根据希望出现在最终RDD中的每个key的分数，随机对key值对RDD进行采样。

```
def sampleByKey(withReplacement: Boolean, fractions: Map[K, Double], seed: Long =
Utils.random.nextLong): RDD[(K, V)]
```

**案例64**

```
val randRDD = sc.parallelize(List( (7,"cat"), (6, "mouse"),(7, "cup"), (6, "book"), (7, "tv"),
(6, "screen"), (7, "heater")))
val sampleMap = List((7, 0.4), (6, 0.6)).toMap()
randRDD.sampleByKey(false, sampleMap,42).collect()

res6: Array[(Int, String)] = Array((7,cat), (6,mouse), (6,book), (6,screen), (7,heater))
```

## *sampleByKeyExact*

这被标记为实验性的，所以不记录它。

```
def sampleByKeyExact(withReplacement: Boolean, fractions: Map[K, Double], seed: Long =
Utils.random.nextLong): RDD[(K, V)]
```

## *saveAsHadoopFile, saveAsHadoopDataset, saveAsNewAPIHadoopFile*

使用用户指定的任何 Hadoop outputFormat 类以 Hadoop 兼容格式保存 RDD。

```
def saveAsHadoopDataset(conf: JobConf)
def saveAsHadoopFile[F <: OutputFormat[K, V]](path: String)(implicit fm: ClassTag[F])

def saveAsHadoopFile[F <: OutputFormat[K, V]](path: String, codec: Class[_ <: CompressionCodec])
(implicit fm: ClassTag[F])

def saveAsHadoopFile(path: String, keyClass: Class[_], valueClass: Class[_], outputFormatClass:
Class[_ <: OutputFormat[_, _]], codec: Class[_ <: CompressionCodec])

def saveAsHadoopFile(path: String, keyClass: Class[_], valueClass: Class[_], outputFormatClass:
Class[_ <: OutputFormat[_, _]], conf: JobConf = new JobConf(self.context.hadoopConfiguration),
codec: Option[Class[_ <: CompressionCodec]] = None)

def saveAsNewAPIHadoopFile[F <: NewOutputFormat[K, V]](path: String)(implicit fm: ClassTag[F])

def saveAsNewAPIHadoopFile(path: String, keyClass: Class[_], valueClass: Class[_],
outputFormatClass: Class[_ <: NewOutputFormat[_, _]], conf: Configuration =
self.context.hadoopConfiguration)
```

## *saveAsObjectFile*

以二进制格式保存 RDD。

```
def saveAsObjectFile(path: String)
```

**案例65**

```
val x = sc.parallelize(1 to 100, 3)
x.saveAsObjectFile("objFile")
val y = sc.objectFile[Int]("objFile")
y.collect()
res52: Array[Int] =  Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                                        11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                                        21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                                        31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
                                        41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                                        51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
                                        61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
                                        71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
                                        81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
                                          91, 92, 93, 94, 95, 96, 97, 98, 99, 100
                                         )
```

## *saveAsSequenceFile*

将 RDD 保存为 Hadoop 序列文件。

```
def saveAsSequenceFile(path: String, codec: Option[Class[_ <: CompressionCodec]] = None)
```

**案例66**

```
val v = sc.parallelize(Array(("owl",3), ("gnu",4), ("dog",1), ("cat",2), ("ant",5)), 2)
v.saveAsSequenceFile("hd_seq_file")
14/04/19 05:45:43 INFO FileOutputCommitter: Saved output of task
'attempt_201404190545_0000_m_000001_191' to file:/home/cloudera/hd_seq_file

[cloudera@localhost ~]$ ll ~/hd_seq_file
total 8
-rwxr-xr-x 1 cloudera cloudera 117 Apr 19 05:45 part-00000
-rwxr-xr-x 1 cloudera cloudera 133 Apr 19 05:45 part-00001
-rwxr-xr-x 1 cloudera cloudera   0 Apr 19 05:45 _SUCCESS
```

## *saveAsTextFile*

将 RDD 保存为文本文件。一次一行。

```
def saveAsTextFile(path: String)
def saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec])
```

**案例67**

```
val a = sc.parallelize(1 to 10000, 3)
a.saveAsTextFile("mydata_a")
14/04/03 21:11:36 INFO FileOutputCommitter: Saved output of task
'attempt_201404032111_0000_m_000002_71' to file:/home/cloudera/Documents/spark-0.9.0-incubating-
bin-cdh4/bin/mydata_a


[cloudera@localhost ~]$ head -n 5 ~/Documents/spark-0.9.0-incubating-bin-cdh4/bin/mydata_a/part-
00000
1
2
3
4
5

// Produces 3 output files since we have created the a RDD with 3 partitions
[cloudera@localhost ~]$ ll ~/Documents/spark-0.9.0-incubating-bin-cdh4/bin/mydata_a/
-rwxr-xr-x 1 cloudera cloudera 15558 Apr  3 21:11 part-00000
-rwxr-xr-x 1 cloudera cloudera 16665 Apr  3 21:11 part-00001
-rwxr-xr-x 1 cloudera cloudera 16671 Apr  3 21:11 part-00002
```

**压缩示例**

```
import org.apache.hadoop.io.compress.GzipCodec
a.saveAsTextFile("mydata_b", classOf[GzipCodec])

[cloudera@localhost ~]$ ll ~/Documents/spark-0.9.0-incubating-bin-cdh4/bin/mydata_b/
total 24
-rwxr-xr-x 1 cloudera cloudera 7276 Apr  3 21:29 part-00000.gz
-rwxr-xr-x 1 cloudera cloudera 6517 Apr  3 21:29 part-00001.gz
-rwxr-xr-x 1 cloudera cloudera 6525 Apr  3 21:29 part-00002.gz

val x = sc.textFile("mydata_b")
x.count()
res2: Long = 10000
```

**写入HDFS 的示例**

```
val x = sc.parallelize(List(1,2,3,4,5,6,6,7,9,8,10,21), 3)
x.saveAsTextFile("hdfs://localhost:8020/user /cloudera/test");

val sp = sc.textFile("hdfs://localhost:8020/user/cloudera/sp_data")
sp.flatMap(_.split(" ")).saveAsTextFile("hdfs://localhost:8020/user/cloudera /sp_x")
```

## *stats*

同时计算RDD中所有值的均值、方差和标准差。

```
def stats(): StatCounter
```

**案例68**

```
val x = sc.parallelize(List(1.0, 2.0, 3.0, 5.0, 20.0, 19.02, 19.29, 11.09, 21.0), 2)
x.stats()
res16: org.apache.spark.util.StatCounter = (count: 9,
                                             mean: 11.266667,
                                             stdev: 8.126859
                                            )
```

## *sortBy*

该函数对输入的 RDD 的数据进行排序，并将其存储在一个新的 RDD 中。第一个参数要求指定一个函数，该函数将输入数据映射到想要 sortBy 的键。第二个参数（可选）指定您希望数据按升序还是降序排序。**列出变体**

```
def sortBy[K](f: (T) ⇒ K, ascending: Boolean = true, numPartitions: Int = this.partitions.size)
(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]
```

**案例69**

```
val y = sc.parallelize(Array(5, 7, 1, 3, 2, 1))
y.sortBy(c => c, true).collect()
res101: Array[Int] = Array(1, 1, 2, 3, 5, 7)

y.sortBy(c => c, false).collect()
res102: Array[Int] = Array(7, 5, 3, 2, 1, 1)

val z = sc.parallelize(Array(("H", 10), ("A", 26), ("Z", 1), ("L", 5)))
z.sortBy(c => c._1, true).collect()
res109: Array[(String, Int)] = Array((A,26), (H,10), (L,5), (Z,1))

z.sortBy(c => c._2, true).collect()
res108: Array[(String, Int)] = Array((Z,1), (L,5), (H,10), (A,26))
```

## *sortByKey*

该函数对输入的 RDD 的数据进行排序并将其存储在一个新的 RDD 中。输出 RDD 是一个经过混洗的 RDD，因为它存储了由减速器输出的经过混洗的数据。这个功能的实现其实很巧妙。首先，它使用范围分区器对混洗后的 RDD 范围内的数据进行分区。然后它使用标准排序机制使用 mapPartitions 对这些范围进行单独排序。

```
def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.size): RDD[P]
```

**案例70**

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = sc.parallelize(1 to a.count.toInt, 2)
val c = a.zip(b)
c.sortByKey(true).collect()
res74: Array[(String, Int)] = Array((ant,5), (cat,2), (dog,1), (gnu, 4), (owl,3))
c.sortByKey(false).collect
res75: Array[(String, Int)] = Array((owl,3), (gnu,4), (dog,1), (cat ,2), (ant,5))

val a = sc.parallelize(1 to 100, 5)
val b = a.cartesian(a)
val c = sc.parallelize(b.takeSample(true, 5, 13), 2)
val d = c.sortByKey(false)
res56: Array[(Int, Int)] = Array((96,9), (84,76), (59,59), (53,65), (52) ,4))
```

## *STDEV*，*sampleStdev*

统计信息和提取物或者*STDEV*-component或校正*sampleStdev*-component

```
def stdev(): Double
def sampleStdev(): Double
```

**案例71**

```
val d = sc.parallelize(List(0.0, 0.0, 0.0), 3)
d.stdev()
res10: Double = 0.0
d.sampleStdev()
```

```
res11: Double = 0.0

val d = sc.parallelize(List(0.0, 1.0), 3)
d.stdev()
d.sampleStdev()
res18: Double = 0.5
res19: Double = 0.7071067811865476

val d = sc.parallelize(List(0.0, 0.0, 1.0), 3)
d.stdev()
res14: Double = 0.4714045207910317
d.sampleStdev()
res15: Double = 0.5773502691896257
```

## *subtract*

执行公知的标准集减法操作： A - B

```
def subtract(other: RDD[T]): RDD[T]
def subtract(other: RDD[T], numPartitions: Int): RDD[T]
def subtract(other: RDD[T], p: Partitioner): RDD[T]
```

**案例73**

```
val a = sc.parallelize(1 to 9, 3)
val b = sc.parallelize(1 to 3, 3)
val c = a.subtract(b)
c.collect()
res3: Array[Int] = Array(6, 9, 4, 7, 5, 8)
```

## *subtractByKey*

与*subtract*非常相似，但不是提供函数，而是每对的key-component 将自动用作从第一个RDD 中删除项目的标准。

```
def subtractByKey[W: ClassTag](other: RDD[(K, W)]): RDD[(K, V)]
def subtractByKey[W: ClassTag](other: RDD[(K, W)], numPartitions: Int): RDD[(K, V)]
def subtractByKey[W: ClassTag](other: RDD[(K, W)], p: Partitioner): RDD[(K, V)]
```

**案例74**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
val b = a.keyBy(_.length)
val c = sc.parallelize(List("ant", "falcon", "squid"), 2)
val d = c.keyBy(_.length)
b.subtractByKey(d).collect()
res15: Array[(Int, String)] = Array((4,lion))
```

## *sum, sumApprox*

计算包含在 RDD 中的所有值的总和。在某些情况下，函数的近似版本可以更快地完成。但是，它以准确性换取速度。

```
def sum(): Double
def sumApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]
```

**案例75**

```
val x = sc.parallelize(List(1.0, 2.0, 3.0, 5.0, 20.0, 19.02, 19.29, 11.09, 21.0), 2)
x.sum()
res17: Double = 101.39999999999999
```

## *take*

提取RDD的前*n*项并将它们作为数组返回。（注意：这听起来很简单，但实际上对于 *Spark* 的实现者来说是一个相当棘手的问题，因为有问题的项目可以在许多不同的分区中。）

```
def take(num: Int): Array[T]
```

**案例76**

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.take(2)
res18: Array[String] = Array(dog, cat )

val b = sc.parallelize(1 to 10000, 5000)
b.take(100)
res6: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                         11 , 12, 13, 14, 15, 16, 17, 18, 19, 20,
                         21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                         31, 32, 33, 34, 635 , 37, 38, 39, 40,
                         41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                         51, 52, 53, 54, 55, 56, 57, 58, 59, 60 ,
                         61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
                         71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
                         81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
                         91, 92, 93, 94, 95, 96, 97, 98, 99, 100
                        )
```

## *takeOrdered*

使用其固有的隐式排序函数对 RDD 的数据项进行排序，并将前*n* 项作为数组返回。

```
def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]
```

**案例77**

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.takeOrdered(2)
res19: Array[String] = Array(ape, cat )
```

## *takeSample*

在以下方面表现

与*sample*不同：

- 它将返回确切数量的样本（提示：第二个参数）
- 它返回一个数组而不是 RDD。
- 它在内部随机化返回项目的顺序。

```
def takeSample(withReplacement: Boolean, num: Int, seed: Int): Array[T]
```

**案例78**

```
val x = sc.parallelize(1 to 1000, 3)
x.takeSample(true, 100, 1)
res3: Array[Int] = Array(339, 718, 810, 105, 71, 268, 333, 360,
                         341, 300, 68, 848, 431, 449, 773, 172,
                         802, 339, 431, 285, 937, 301, 167, 69,
                         330, 864, 40, 645, 4, 4, 3, 6 3 3 160,
                         675, 232, 794, 577, 571, 805, 317, 136,
                         860, 522, 45, 628, 178, 321, 482, 657,
                         114, 332, 9, 20, 7 7 130, 863, 773, 559,
                         301, 694, 460, 839, 952, 664, 851, 260,
                         729, 823, 880, 792, 964, 614, 83, 8, 8,
                         6, 1, 8, 8, 3, 8 663, 344, 546, 918, 436,
                         451, 397, 670, 756, 512, 391, 70, 213, 896, 123, 858
                        )
```

## *toDebugString*

返回一个字符串，其中包含有关 RDD 及其依赖项的调试信息。

```
def toDebugString: String
```

**案例79**

```
val a = sc.parallelize(1 to 9, 3)
val b = sc.parallelize(1 to 3, 3)
val c = a.subtract(b)
c.toDebugString
res6: String =
MappedRDD[15] at subtract at <console>:16 (3 partitions)
  SubtractedRDD[14] at subtract at <console>:16 (3 partitions)
    MappedRDD[12] at subtract at <console>:16 (3 partitions)
      ParallelCollectionRDD[10] at parallelize at <console>:12 (3 partitions)
    MappedRDD[13] at subtract at <console>:16 (3 partitions)
      ParallelCollectionRDD[11] at parallelize at <console>:12 (3 partitions)
```

## *toJavaRDD*

这个 RDD 对象嵌入到一个 JavaRDD 对象中并返回它。

```
def toJavaRDD() : JavaRDD[T]
```

**案例81**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.toJavaRDD()
res3: org.apache.spark.api.java.JavaRDD[String] = ParallelCollectionRDD[6] at parallelize at
<console>:12
```

## *toLocalIterator*

在主节点将 RDD 转换为 scala 迭代器。

```
def toLocalIterator: Iterator[T]
```

**案例82**

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
val iter = z.toLocalIterator()

iter.next()
res51: Int = 1

iter.next()
res52: Int = 2
```

## *top*

利用 $T$ 的隐式排序来确定顶部的 $k$ 值并将它们作为数组返回。

```
def top(num: Int)(implicit ord: Ordering[T]): Array[T]
```

**案例83**

```
val c = sc.parallelize(Array(6, 9, 4, 7, 5, 8), 2)
c.top(2)
res28: Array[Int] = Array(9, 8)
```

## *toString*

组装 RDD 的可读的文本描述。

```
override def toString: String
```

**案例84**

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.toString()
res61: String = ParallelCollectionRDD[80] at parallelize at <console>:21

val randRDD = sc.parallelize(List( (7,"cat"), (6, "mouse"),(7, "cup"), (6, "book"), (7, "tv"),
(6, "screen"), (7, "heater")))
val sortedRDD = randRDD.sortByKey()
sortedRDD.toString()
res64: String = ShuffledRDD[88] at sortByKey at <console>:23
```

## *treeAggregate*

计算与聚合相同的东西，除了它以多级树模式聚合 RDD 的元素。另一个区别是它不使用第二个缩减函数 (combOp) 的初始值。默认情况下使用深度为 2 的树，但这可以通过 depth 参数进行更改。

```
def treeAggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U, depth: Int = 2)
(implicit arg0: ClassTag[U]): U
```

**案例85**

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)

// 让我们先用分区标签打印出 RDD 的内容
def myfunc(index: Int, iter: Iterator[( Int)]) : Iterator[String] = {
  iter.map(x => "[partID:" + index + ", val: " + x + "]")
}

z.mapPartitionsWithIndex(myfunc).collect()
```

```
res28: Array [String] = Array([partID:0, val: 1], [partID:0, val: 2], [partID:0, val: 3],
[partID:1, val: 4], [partID:1 , val: 5], [partID:1, val: 6])

z.treeAggregate(0)(math.max(_, _), _ + _)
res40: Int = 9

// 注意与普通聚合不同。树聚合不应用第二次缩减的初始值
// 此示例返回 11，因为初始值是 5
// 分区 0 的缩减将是 max(5, 1, 2,
// 分区 1 的缩减将是 max(4, 5, 6) = 6
// 分区之间的最终缩减将是 5 + 6 = 11
// 注意最终的缩减不包括初始值
z.treeAggregate(5)( math.max(_, _), _ + _)
res42: Int = 11
```

## *treeReduce*

工作方式与 reduce 类似，只是在多级树模式中减少 RDD 的元素。

```
def  treeReduce(f: (T, T) ⇒ T, depth: Int = 2): T
```

**案例86**

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.treeReduce(_+_)
res49: Int = 21
```

## *union*, ++

执行标准集合操作：A union B

```
def ++ (other: RDD [T]): RDD [T]
def union (other: RDD [T]): RDD [T]
```

**案例87**

```
val a = sc.parallelize(1 to 3, 1)
val b = sc.parallelize(5 to 7, 1)
(a ++ b).collect()
res0: Array[Int] = Array(1, 2, 3, 5 , 6, 7)
```

## *unpersist*

使 RDD 非物质化（即擦除硬盘和内存中的所有数据项）。但是，RDD 对象仍然存在。如果它在计算中被引用，Spark 将使用存储的依赖图自动重新生成它。

```
def unpersist(blocking: Boolean = true): RDD[T]
```

**案例88**

```
val y = sc.parallelize(1 to 10, 10)
val z = (y++y)
z.collect()
z.unpersist(true)
14/04/19 03:04:57 INFO UnionRDD: Removing RDD 22 from persistence list
14/04/19 03:04:57 INFO BlockManager: Removing RDD 22
```

## *values*

从所有包含的元组中提取值并在新的 RDD 中返回它们。

```
def values: RDD[V]
```

**案例89**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.values.collect()
res3: Array[String] = Array(dog, tiger, lion, cat, panther, eagle)
```

## ***variance***, *sampleVariance*

调用 stats 并提取方差分量或校正后的样本**方差分量。

```
def variance(): Double
def sampleVariance(): Double
```

**案例90**

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0,
8.9, 5.5), 3)
a.variance()
res70: Double = 10.605333333333332

val x = sc.parallelize(List(1.0, 2.0, 3.0, 5.0, 20.0, 19.02, 19.29, 11.09, 21.0), 2)
x.variance()
res14: Double = 66.04584444444443

x.sampleVariance()
res13: Double = 74.30157499999999
```

## *zip*

通过将任一分区的第 i 个相互组合来连接两个 RDD。生成的 RDD 将由两个组件的元组组成，这些元组被 PairRDDFunctions 扩展提供的方法解释为键值对。

```
def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]
```

**案例91**

```
val a = sc.parallelize(1 to 100, 3)
val b = sc.parallelize(101 to 200, 3)
a.zip(b).collect()
res1: Array[(Int, Int)] = Array((1,101), (2,102), (3,103), (4,104), (5,105), (6,106), (7,107),
(8,108), (9,109), ( 10,110), (11,111), (12,112), (13,113), (14,114), (15,115), (16,116),
(17,117), (18,118), (19,119), (20,2,12,120) , (23,123), (24,124), (25,125), (26,126), (27,127),
(28,128), (29,129), (30,130), (31,131), (32,132), (31,3,31) 35,135), (36,136), (37,137),
(38,138), (39,139), (40,140), (41,141), (42,142), (43,143), (44,144), (45,4,145) (45,6,145) ,
(48,148), (49,149), (50,150), (51,151), (52,152), (53,153), (54,154), (55,155), (56,156),
(57,157), (51,58,51) 60,160)、(61,161)、(62,162)、(63,163)、(64,164)、(65,165)、(66,166)、
(67,167)、(68,168)、(69,169)、(70,7,17••0) , (73,173), (74,174), (75,175), (76,176), (77,177),
(78,...

val a = sc.parallelize(1 to 100, 3)
val b = sc.parallelize(101 to 200, 3)
val c = sc.parallelize(201 to 300, 3)
a.zip(b).zip(c) .map((x) => (x._1._1, x._1._2, x._2 )).collect()
res12: Array[(Int, Int, Int)] = Array((1,101,201), (2,102,202), (3,103,203), (4,104,204),
(5,105,205), (6,106,206), (2,901), (2,901), (2,901), (7,901) , （10110210）, （11111211）,
（12112212）, （13113213）, （14114214）, （15115215）, （16116216）, （17117217）, （18118218）,
（19119219）, （20120220）, （21121221）, （ 22122222）, （23123223）, （24124224）, （25125225）,
（26126226）, （27127227）, （28128228）, （29129229）, （30130230）, （31131231）, （32132232）,
（33133233）, （34134234） , （35135235）, （36136236）, （37137237）, （38138238）, （39139239）,
（40140240）, （41141241）, （42142242）, （43143243）, （44144244）, （45145245）, （46146246）, （
47,147,247), (48,148,248), (49,149,249), (50,150,250), (51,151,251), (52,152,252), (53,153,253),
(525,45), (525,45)...
```

## *zipParitions*

类似于*zip*。但提供了对压缩过程的更多控制。

```
def zipPartitions[B: ClassTag, V: ClassTag](rdd2: RDD[B])(f: (Iterator[T], Iterator[B]) =>
Iterator[V]): RDD[V]

def zipPartitions[B: ClassTag, V: ClassTag](rdd2: RDD[B], preservesPartitioning: Boolean)(f:
(Iterator[T], Iterator[B]) => Iterator[V]): RDD[V]

def zipPartitions[B: ClassTag , C: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD[C])(f:
(Iterator[T], Iterator[B], Iterator[C]) => Iterator[V]) : RDD[V]

def zipPartitions[B: ClassTag, C: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD[C],
preservesPartitioning: Boolean)(f: (Iterator[T], Iterator[ B], Iterator[C]) => Iterator[V]):
RDD[V]

def zipPartitions[B: ClassTag, C: ClassTag, D: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD
[C], rdd4: RDD[D])(f: (Iterator[T], Iterator[B], Iterator[C], Iterator[D]) => Iterator[V]):
RDD[V]

def zipPartitions[B: ClassTag, C: ClassTag, D: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3:
RDD[C], rdd4: RDD[D], preservesPartitioning: Boolean)(f: (Iterator [T], Iterator[B],
Iterator[C], Iterator[D]) => Iterator[V]): RDD[V]
```

**案例92**

```
val a = sc.parallelize(0 to 9, 3)
val b = sc.parallelize(10 to 19, 3)
val c = sc.parallelize(100 to 109, 3)
def myfunc(aiter: Iterator[Int], biter : Iterator[Int], citer: Iterator[Int]): Iterator[String]
=
{
  var res = List[String]()
  while (aiter.hasNext && biter.hasNext && citer.hasNext)
  {
    val x = aiter.next + " " + biter.next + " " + citer.next
    res ::= x
  }
  res.iterator()
}
a.zipPartitions(b, c)(myfunc).collect()
res50: Array[String] = Array(2 12 102, 1 11 101, 0 10 100, 5 15 105, 4 14 104, 3 13 103, 9 19
109, 8 18 108, 7 17 107, 6 16 106)
```

## *zipWithIndex*

使用元素索引**压缩**RDD 的元素。索引从 0 开始。如果 RDD 分布在多个分区中，则启动一个 spark Job 来执行此操作。

```
def zipWithIndex(): RDD[(T, Long)]
```

**案例93**

```
val z = sc.parallelize(Array("A", "B", "C", "D"))
val r = z.zipWithIndex()
res110: Array[(String, Long)] = Array((A,0) , (B,1), (C,2), (D,3))

val z = sc.parallelize(100 to 120, 5)
val r = z.zipWithIndex()
r.collect()
res11: Array[(Int, Long) ] = Array((100,0), (101,1), (102,2), (103,3), (104,4), (105,5),
(106,6), (107,7) ), (108,8), (109,9), (110,10), (111,11), (112,12), (113,13), (114,14),
(115,15), (116,16), (117,17), (118,18), (119,19), (120,20))
```

## *zipWithUniqueId*

这与 zipWithIndex 不同，因为它只是为每个数据元素提供唯一的 id，但 id 可能与数据元素的索引号不匹配。即使 RDD 分布在多个分区中，此操作也不会启动 spark 作业。
将以下示例的结果与 zipWithIndex 的第二个示例的结果进行比较。能够看到差异。

```
def zipWithUniqueId(): RDD[(T, Long)]
```

**案例95**

```
val z = sc.parallelize(100 to 120, 5)
val r = z.zipWithUniqueId()
r.collect()

res12: Array[(Int, Long)] = Array((100,0), (101,5), (102, 10), (103,15), (104,1), (105,6),
(106,11), (107,16), (108,2), (109,7), (110,12) , (111,17), (112,3), (113,8), (114,13), (115,18),
(116,4), (117,9), (118,14), ( 119,19), (120,24))
```