# pyspark package

## Subpackages

* [pyspark.sql module](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html)
* [pyspark.streaming module](http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html)
* [pyspark.ml package](http://spark.apache.org/docs/latest/api/python/pyspark.ml.html)
* [pyspark.mllib package](http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html)

## Contents

PySpark is the Python API for Spark.

Public classes:

* [SparkContext](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext):

Spark功能的主入口点。

* [RDD](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD):

弹性分布式数据集（RDD），Spark中的基本抽象。

* [Broadcast](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.Broadcast):

广播变量可以跨任务重用。

* [Accumulator](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.Accumulator):

一个“附加”共享变量，任务只能添加值。

* [SparkConf](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkConf):

装置pyspark

* [SparkFiles](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkFiles):

.访问附带作业的文件

* [StorageLevel](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.StorageLevel):

更细粒度的缓存持久性级别。

# SparkConf

class pyspark.SparkConf(loadDefaults=True, \_jvm=None, \_jconf=None)

Spark应用程序的配置。 用于将各种Spark参数设置为键值对。

大多数情况下，您将使用SparkConf（）创建一个SparkConf对象，它将从spark. \* Java系统属性中加载值。 在这种情况下，您直接在SparkConf对象上设置的任何参数都优先于系统属性。

对于单元测试，您也可以调用SparkConf（false）来跳过加载外部设置，并获得相同的配置，无论系统属性如何。

此类中的所有setter方法都支持链接。 例如，您可以编写conf.setMaster（“local”）。setAppName（“我的应用程序”）。

注意

一旦将**SparkConf对象**传递给Spark，就会**被克隆，不能再被用户修改**。

**contains(key)**

此配置是否包含给定的密钥？

**get(key, defaultValue=None)**

获取某些键的配置值，否则返回默认值。

**getAll()**

将所有值作为键值对列表获取。

**set(key, value)**

设置配置属性。

**setAll(pairs)**

设置多个参数，作为键值对列表传递。

|  |  |
| --- | --- |
| Parameters: | pairs – list of key-value pairs to set |

**setAppName(value)**

应用名称

**setExecutorEnv(key=None, value=None, pairs=None)**

设置要传递给执行程序的环境变量。

**setIfMissing(key, value)**

设置配置属性（如果尚未设置）。

**setMaster**(**value**)

设置要连接的主URL。

**setSparkHome(value)**

**设置Spark在工作节点上安装的路径**。

**toDebugString()**

返回可打印版本的配置，作为key =值对的列表，每行一个。

class **pyspark**.**SparkContext**(master=None, appName=None, sparkHome=None, pyFiles=None, environment=None, batchSize=0, serializer=PickleSerializer(), conf=None, gateway=None, jsc=None, profiler\_cls=<class 'pyspark.profiler.BasicProfiler'>)

Spark功能的主入口点。 **SparkContext表示与Spark集群的连接**，**可用于在该集群上创建RDD和广播变量。**

PACKAGE\_EXTENSIONS = ('.zip', '.egg', '.jar')

**accumulator**(value, accum\_param=None)

使用给定的初始值创建一个**累加器**，使用给定的AccumulatorParam辅助对象定义如何添加数据类型的值（如果提供）。 **如果您没有提供一个，则使用默认值AccumulatorParams进行整数和浮点数**。 对于其他类型，可以使用自定义的AccumulatorParam。

**addFile(path, recursive=False)**

在**每个节点上添加要使用此Spark作业下载的文件(可以作为依赖包)**。 传递的路径可以是本地文件，HDFS（或其他Hadoop支持的文件系统）中的文件，或HTTP，HTTPS或FTP URI。

要访问Spark作业中的文件，请使用文件名L {SparkFiles.get（fileName）<pyspark.files.SparkFiles.get>}来查找其下载位置。

如果递归选项设置为True，则可以给出目录。 **目前仅支持Hadoop支持的文件系统。**

>>> from pyspark import SparkFiles

>>> path = os.path.join(tempdir, "test.txt")

>>> with open(path, "w") as testFile:

... \_ = testFile.write("100")

>>> sc.addFile(path)

>>> def func(iterator):

... with open(SparkFiles.get("test.txt")) as testFile:

... fileVal = int(testFile.readline())

... return [x \* fileVal for x in iterator]

>>> sc.parallelize([1, 2, 3, 4]).mapPartitions(func).collect()

[100, 200, 300, 400]

**addPyFile(path)**

**为将来在此SparkContext上执行的所有任务添加.py或.zip依赖项**。 传递的路径可以是本地文件，HDFS（或其他Hadoop支持的文件系统）中的文件，或HTTP，HTTPS或FTP URI。

**applicationId**

**Spark应用程序的唯一标识符。** 其格式取决于调度程序的实现。

* in case of local spark app something like ‘local-1433865536131’
* in case of YARN something like ‘application\_1433865536131\_34483’

>>> sc.applicationId

u'local-...'

**binaryFiles**(path, minPartitions=None)

注意

试验

从HDFS，本地文件系统（在所有节点上可用）或作为字节数组的任何Hadoop支持的文件系统URI读取二进制文件目录。 每个文件作为单个记录读取并返回到键值对中，其中键是每个文件的路径，值是每个文件的内容。

注意

**小文件是首选，大文件也是允许的，但可能会导致性能不佳**。

binaryRecords(path, recordLength)

注意

试验

假设每个记录是一组具有指定数字格式的数字（参见ByteBuffer），并且每个记录的字节数是常数，从一个平坦的二进制文件加载数据。

|  |  |
| --- | --- |
| Parameters: | * path – Directory to the input data files * recordLength – The length at which to split the records |

**broadcast(value)**

**将一个只读变量广播到集群**，返回一个L {Broadcast <pyspark.broadcast.Broadcast>}对象，以便在分布式函数中读取它。 该变量将被发送到每个集群只有一次。

**cancelAllJobs()**

取消所有已安排或正在运行的作业。

cancelJobGroup(groupId)

取消指定组的活动作业。 有关更多信息，请参阅SparkContext.setJobGroup。

**defaultMinPartitions**

用户未给出Hadoop RDD的默认分区数

**defaultParallelism** (跟随计算机核数)

用户未给出的默认平行度（例如，为了减少任务）

**dump\_profiles(path)**

将配置文件统计数据转储到目录路径中

**emptyRDD()**

创建一个没有分区或元素的RDD。

getConf()

getLocalProperty(key)

Get a local property set in this thread, or null if it is missing. See [setLocalProperty](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext.setLocalProperty)

**classmethod getOrCreate(conf=None)** (非常有用，建议首选)

Get or instantiate a SparkContext and register it as a singleton object.

|  |  |
| --- | --- |
| Parameters: | conf – SparkConf (optional) |

hadoopFile(path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

从HDFS，本地文件系统（在所有节点上可用）或任何Hadoop支持的文件系统URI读取具有任意键和值类的“旧”Hadoop InputFormat。 机制与sc.sequenceFile相同。

Hadoop配置可以作为Python dict传递。 这将被转换为Java中的配置。

|  |  |
| --- | --- |
| Parameters: | * path – path to Hadoop file * inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapred.TextInputFormat”) * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”) * keyConverter – (None by default) * valueConverter – (None by default) * conf – Hadoop configuration, passed in as a dict (None by default) * batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically) |

**hadoopRDD(inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)**

从任意的Hadoop配置中读取具有任意键和值类的“旧”Hadoop InputFormat，该配置以Python语法传入。 这将被转换为Java中的配置。 机制与sc.sequenceFile相同。

|  |  |
| --- | --- |
| Parameters: | * inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapred.TextInputFormat”) * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”) * keyConverter – (None by default) * valueConverter – (None by default) * conf – Hadoop configuration, passed in as a dict (None by default) * batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically) |

newAPIHadoopFile(path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

从HDFS，本地文件系统（在所有节点上可用）或任何Hadoop支持的文件系统URI读取具有任意键和值类的“新API”Hadoop InputFormat。 机制与sc.sequenceFile相同。

Hadoop配置可以作为Python dict传递。 这将被转换为Java中的配置

|  |  |
| --- | --- |
| Parameters: | * path – path to Hadoop file * inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapreduce.lib.input.TextInputFormat”) * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”) * keyConverter – (None by default) * valueConverter – (None by default) * conf – Hadoop configuration, passed in as a dict (None by default) * batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically) |

**newAPIHadoopRDD**(inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

从任意的Hadoop配置中读取具有任意键和值类的“新API”Hadoop InputFormat，该配置以Python dict的形式传递。 这将被转换为Java中的配置。 机制与sc.sequenceFile相同。

|  |  |
| --- | --- |
| Parameters: | * inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapreduce.lib.input.TextInputFormat”) * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”) * keyConverter – (None by default) * valueConverter – (None by default) * conf – Hadoop configuration, passed in as a dict (None by default) * batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically) |

**parallelize(c, numSlices=None)**

分发本地Python集合以形成RDD。 如果输入表示性能范围，则建议使用xrange。

>>> sc.parallelize([0, 2, 3, 4, 6], 5).glom().collect()

[[0], [2], [3], [4], [6]]

>>> sc.parallelize(xrange(0, 6, 2), 5).glom().collect()

[[], [0], [], [2], [4]]

**pickleFile(name, minPartitions=None)**

Load an RDD previously saved using [RDD.saveAsPickleFile](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.saveAsPickleFile) method.

>>> tmpFile = NamedTemporaryFile(delete=True)

>>> tmpFile.close()

>>> sc.parallelize(range(10)).saveAsPickleFile(tmpFile.name, 5)

>>> sorted(sc.pickleFile(tmpFile.name, 3).collect())

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**range(start, end=None, step=1, numSlices=None)**

创建一个新的包含元素的RDD，从头到尾（独占），逐个增加元素。 可以像python的内置range（）函数一样调用。 如果使用单个参数调用，则该参数被解释为end，start被设置为0。

|  |  |
| --- | --- |
| Parameters: | * start – the start value * end – the end value (exclusive) * step – the incremental step (default: 1) * numSlices – the number of partitions of the new RDD |
| Returns: | An RDD of int |

>>> sc.range(5).collect()

[0, 1, 2, 3, 4]

>>> sc.range(2, 4).collect()

[2, 3]

>>> sc.range(1, 7, 2).collect()

[1, 3, 5]

**runJob(rdd, partitionFunc, partitions=None, allowLocal=False)**

在指定的分区集上执行给定的partitionFunc，将结果作为元素数组返回。

如果未指定“分区”，则将在所有分区上运行.

>>> myRDD = sc.parallelize(range(6), 3)

>>> sc.runJob(myRDD, lambda part: [x \* x for x in part])

[0, 1, 4, 9, 16, 25]

>>> myRDD = sc.parallelize(range(6), 3)

>>> sc.runJob(myRDD, lambda part: [x \* x for x in part], [0, 2], True)

[0, 1, 16, 25]

**sequenceFile(path, keyClass=None, valueClass=None, keyConverter=None, valueConverter=None, minSplits=None, batchSize=0)**

读取具有任意键和值的Hadoop SequenceFile从HDFS，本地文件系统（在所有节点上可用）或任何Hadoop支持的文件系统URI的可写类。 机制如下：

1.一个Java RDD是从SequenceFile或其他InputFormat创建的，Key和Value可写类

2.通过Pyrolite酸洗尝试序列化

3.如果失败，则回退是在每个键和值上调用“toString”

4.PickleSerializer用于反序列化Python边的腌制对象

|  |  |
| --- | --- |
| Parameters: | * path – path to sequncefile * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”) * keyConverter – * valueConverter – * minSplits – minimum splits in dataset (default min(2, sc.defaultParallelism)) * batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically) |

**setCheckpointDir(dirName)**

设置RDD将要检查点的目录。 如果在集群上运行，该目录必须是HDFS路径。

**setJobGroup(groupId, description, interruptOnCancel=False)**

将组ID分配给此线程启动的所有作业，直到组ID设置为不同的值或清除为止。

通常，应用程序中的执行单元包含多个Spark操作或作业。 应用程序员可以使用此方法将所有这些作业分组在一起并给出组描述。 一旦设置，Spark Web UI 将这些作业与该组关联。

应用程序可以使用SparkContext.cancelJobGroup取消此组中的所有正在运行的作业。

>>> import threading

>>> from time import sleep

>>> result = "Not Set"

>>> lock = threading.Lock()

>>> def map\_func(x):

... sleep(100)

... raise Exception("Task should have been cancelled")

>>> def start\_job(x):

... global result

... try:

... sc.setJobGroup("job\_to\_cancel", "some description")

... result = sc.parallelize(range(x)).map(map\_func).collect()

... except Exception as e:

... result = "Cancelled"

... lock.release()

>>> def stop\_job():

... sleep(5)

... sc.cancelJobGroup("job\_to\_cancel")

>>> supress = lock.acquire()

>>> supress = threading.Thread(target=start\_job, args=(10,)).start()

>>> supress = threading.Thread(target=stop\_job).start()

>>> supress = lock.acquire()

>>> print(result)

Cancelled

**如果作业组的interruptOnCancel设置为true**，则作业取消将导致在作业执行器线程上调用Thread.interrupt（）。 这有助于确保任务实际上能够及时停止，但由于HDFS-1208，HDFS可以通过将节点标记为死亡来响应Thread.interrupt（），因此默认关闭。

**setLocalProperty(key, value)**

**设置影响从此线程提交的作业的本地属性，例如Spark Fair调度程序池。**

**setLogLevel(logLevel)**

**控制我们的logLevel。 这将覆盖任何用户定义的日志设置。 有效的日志级别包括：ALL，DEBUG，ERROR，致命，INFO，OFF，TRACE，WARN**

**classmethod setSystemProperty(key, value)**

设置Java系统属性，如spark.executor.memory。 必须在实例化SparkContext之前调用这一点。

**show\_profiles**

将配置文件统计信息打印到stdout

**sparkUser**

为运行SparkContext的用户获取SPARK\_USER。

**startTime**

当Spark上下文启动时返回时代

**statusTracker()**

Return [StatusTracker](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.StatusTracker) object

**stop()**

关闭SparkContext。

**textFile(name, minPartitions=None, use\_unicode=True)**

**从HDFS，本地文件系统（在所有节点上可用）或任何Hadoop支持的文件系统URI读取文本文件，并将其作为字符串的RDD返回。**

如果**use\_unicode为False，**则字**符串将保持为str（编码为utf-8**），它比unicode更快更小。 （在Spark 1.2中添加）

>>> path = os.path.join(tempdir, "sample-text.txt")

>>> with open(path, "w") as testFile:

... \_ = testFile.write("Hello world!")

>>> textFile = sc.textFile(path)

>>> textFile.collect()

[u'Hello world!']

**uiWebUrl**

**返回SparkContext启动的SparkUI实例的URL**

**union(rdds) (算子操作)**

构建合并的RDD列表。

这支持使用不同序列化格式的RDD的（），尽管这迫使它们使用默认序列化程序进行预留：

>>> path = os.path.join(tempdir, "union-text.txt")

>>> with open(path, "w") as testFile:

... \_ = testFile.write("Hello")

>>> textFile = sc.textFile(path)

>>> textFile.collect()

[u'Hello']

>>> parallelized = sc.parallelize(["World!"])

>>> sorted(sc.union([textFile, parallelized]).collect())

[u'Hello', 'World!']

version

The version of Spark on which this application is running.

**wholeTextFiles(path, minPartitions=None, use\_unicode=True)**

**从HDFS，本地文件系统（在所有节点上可用）或任何Hadoop支持的文件系统URI读取文本文件的目录。 每个文件作为单个记录读取并返回到键值对中，其中键是每个文件的路径，值是每个文件的内容。**

如果use\_unicode为False，则字符串将保持为str（编码为utf-8），它比unicode更快更小。 （在Spark 1.2中添加）

例如，如果您有以下文件：

hdfs://a-hdfs-path/part-00000

hdfs://a-hdfs-path/part-00001

...

hdfs://a-hdfs-path/part-nnnnn

Do rdd = sparkContext.wholeTextFiles(“hdfs://a-hdfs-path”), then rdd contains:

(a-hdfs-path/part-00000, its content)

(a-hdfs-path/part-00001, its content)

...

(a-hdfs-path/part-nnnnn, its content)

Note

**小文件是首选的，因为每个文件将被完全加载到内存中。**

>>> dirPath = os.path.join(tempdir, "files")

>>> os.mkdir(dirPath)

>>> with open(os.path.join(dirPath, "1.txt"), "w") as file1:

... \_ = file1.write("1")

>>> with open(os.path.join(dirPath, "2.txt"), "w") as file2:

... \_ = file2.write("2")

>>> textFiles = sc.wholeTextFiles(dirPath)

>>> sorted(textFiles.collect())

[(u'.../1.txt', u'1'), (u'.../2.txt', u'2')]

# SparkFiles

class pyspark.**SparkFiles** (主要针对程序运行添加的依赖文件)

**解析通过L {SparkContext.addFile（）<pyspark.context.SparkContext.addFile>}添加的文件的路径。**

SparkFiles只包含类方法; 用户不应该创建SparkFiles实例。

classmethod get(filename)

**获取通过SparkContext.addFile（）添加的文件的绝对路径。**

**classmethod getRootDirectory()**

获取包含通过SparkContext.addFile（）添加的文件的根目录。

# RDD（针对RDD可执行的操作）

class pyspark.**RDD(jrdd, ctx, jrdd\_deserializer=AutoBatchedSerializer(PickleSerializer()))**

弹性分布式数据集（RDD），Spark中的基本抽象。 表示可以并行操作的元素的不可变分区集合。

**aggregate(zeroValue, seqOp, combOp)**

**聚合每个分区的元素，然后对所有分区的结果，使用给定的组合函数和中性“零值”。**

允**许函数op（t1，t2）修改t1并将其作为结果值返回，以避免对象分配**; 但是，它不应该修改t2。

第一个函数（seqOp）可以返回与该RDD类型不同的结果类型U。 因此，我们需要一个操作来将T合并成一个U，一个操作用于合并两个U

>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))

>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))

>>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)

(10, 4)

>>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)

(0, 0)

**aggregateByKey(zeroValue, seqFunc, combFunc, numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

**聚合**

使用给定的组合函数和中性“零值”来聚合每个键的值。 该函数可以返回不同于该RDD，V中的值的类型的不同的结果类型U。因此，我们需要一个操作来将V合并成一个U，一个操作用于合并两个U，前者的操作用于 合并分区中的值，后者用于在分区之间合并值。 为了避免内存分配，这两个功能都允许修改并返回其第一个参数，而不是创建一个新的U。

**cache()**

使用默认存储级别（MEMORY\_ONLY）保持此RDD。

**cartesian(other)**

返回此RDD的**笛卡尔乘积，另**一个，即**所有成对元素（a，b）的RDD，其中a为自身，b为其他元素。**

>>> rdd = sc.parallelize([1, 2])

>>> sorted(rdd.cartesian(rdd).collect())

[(1, 1), (1, 2), (2, 1), (2, 2)]

**checkpoint() 检查点**

**将此RDD标记为检查点。 它将被保存到使用SparkContext.setCheckpointDir（）设置的检查点目录中的文件中，并且对其父RDD的所有引用将被删除。 在此RDD上执行任何作业之前，必须调用此函数。 强烈建议将此RDD保留在内存中，否则将其保存在文件中将需要重新计算。**

**coalesce(numPartitions, shuffle=False)**

返回一个减少到numPartitions分区的新RDD。

>>> sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()

[[1], [2, 3], [4, 5]]

>>> sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()

[[1, 2, 3, 4, 5]]

**cogroup(other, numPartitions=None)**

对于**自身或其他的每个关键字k，返回包含元组的结果RDD，该元组具有该键的值列表以及其他值。**

>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2)])

>>> [(x, tuple(map(list, y))) for x, y in sorted(list(x.cogroup(y).collect()))]

[('a', ([1], [2])), ('b', ([4], []))]

**collect()**

返回包含此RDD中所有元素的列表。

注意

只有在生成的数组预计为小的时候才应该使用这个方法，因为所有的数据被加载到驱动程序的内存中。

**collectAsMap()**

将该RDD中的键值对作为字典返回给主机。

注意

因为所有的数据都被加载到驱动程序的内存中，所以这个方法只应该被使用，如果所得到的数据是小的。

>>> m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()

>>> m[1]

2

>>> m[3]

4

**combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

使用一组自定义聚合函数组合每个键的元素的通用功能。

对于“组合类型”C，将RDD [（K，V）]转换为类型RDD [（K，C）]的结果。

用户提供三个功能：

* createCombiner, 将V转换为C（例如，创建单元素列表）
* mergeValue, 将V合并成C（例如，将其添加到列表的末尾）
* mergeCombiners, 将两个C组合成一个。

此外，用户可以控制输出RDD的分区。

注意

V和C可以不同 - 例如，可以将类型（Int，Int）的RDD组合成类型为（Int，List [Int]）的RDD。

>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

>>> def add(a, b): return a + str(b)

>>> sorted(x.combineByKey(str, add, add).collect())

[('a', '11'), ('b', '1')]

**context**

The [SparkContext](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext) that this RDD was created on.

**count()**

返回此RDD中的元素数。

>>> sc.parallelize([2, 3, 4]).count()

3

**countApprox(timeout, confidence=0.95)**

注意

试验

即使不是所有任务都完成，count（）的近似版本在超时期间返回可能不完整的结果。

>>> rdd = sc.parallelize(range(1000), 10)

>>> rdd.countApprox(1000, 1.0)

1000

**countApproxDistinct(relativeSD=0.05)**

注意

试验

返回RDD中不同元素的大致数量。

所使用的算法是基于streamlib实现的“HyperLogLog in Practice：Algorithmic Engineering of the Art of Art Artinality Estimation Algorithm”，可以在这里获得。

|  |  |
| --- | --- |
| Parameters: | relativeSD – Relative accuracy. Smaller values create counters that require more space. It must be greater than 0.000017. |

>>> n = sc.parallelize(range(1000)).map(str).countApproxDistinct()

>>> 900 < n < 1100

True

>>> n = sc.parallelize([i % 20 for i in range(1000)]).countApproxDistinct()

>>> 16 < n < 24

True

**countByKey()**

计算每个键的元素数，并将结果作为字典返回到主。

>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

>>> sorted(rdd.countByKey().items())

[('a', 2), ('b', 1)]

**countByValue()**

将此RDD中的每个唯一值的计数作为（值，计数）对的字典返回。

>>> sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items())

[(1, 2), (2, 3)]

**distinct(numPartitions=None)**

返回一个包含此RDD中不同元素的新RDD。

>>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())

[1, 2, 3]

**filter(f)**

返回一个仅包含满足谓词的元素的新RDD。

>>> rdd = sc.parallelize([1, 2, 3, 4, 5])

>>> rdd.filter(lambda x: x % 2 == 0).collect()

[2, 4]

**first()**

Return the first element in this RDD.

>>> sc.parallelize([2, 3, 4]).first()

2

>>> sc.parallelize([]).first()

Traceback (most recent call last):

...

ValueError: RDD is empty

**flatMap(f, preservesPartitioning=False)**

**通过首先对这个RDD的所有元素应用一个函数，然后使结果变平，来返回一个新的RDD。**

>>> rdd = sc.parallelize([2, 3, 4])

>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())

[1, 1, 1, 2, 2, 3]

>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())

[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]

**flatMapValues(f)**

**通过flatMap函数传递键值对RDD中的每个值，而无需更改键; 这也保留了原始RDD的分区。**

>>> x = sc.parallelize([("a", ["x", "y", "z"]), ("b", ["p", "r"])])

>>> def f(x): return x

>>> x.flatMapValues(f).collect()

[('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]

**fold(zeroValue, op)**

**聚合每个分区的元素**，然后使用给定的关联函数和中性“零值”对所有分区的结果进行聚合。

允许函数op（t1，t2）修改t1并将其作为其结果值返回，以避免对象分配; 但是，它不应该修改t2。

这与Scala等功能语言中的非分布式集合实现的折叠操作有所不同。 该折叠操作可以单独应用于分区，然后将这些结果折叠成最终结果，而不是按照一些定义的顺序将折叠应用于每个元素。 对于不可交换的函数，结果可能与应用于非分布式集合的折叠的结果不同。

>>> from operator import add

>>> sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)

15

**foldByKey(zeroValue, func, numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

使用关联函数“func”和中性“零值”合并每个键的值，该值可以任意次数添加到结果中，并且不得更改结果（例如，0为加法，或1为乘法。）。

>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

>>> from operator import add

>>> sorted(rdd.foldByKey(0, add).collect())

[('a', 2), ('b', 1)]

**foreach(f)**

对这个RDD的所有元素应用一个函数。

>>> def f(x): print(x)

>>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)

**foreachPartition(f) 针对分区**

将函数应用于此RDD的每个分区。

>>> def f(iterator):

... for x in iterator:

... print(x)

>>> sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)

**fullOuterJoin(other, numPartitions=None)**

执行自我和其他的正确的外部联合。

对于自身中的每个元素（k，v），如果其他元素（k，（v，无））中没有元素，则所得到的RDD将包含其他的w或对（k，（v，None））中的所有对（k，（v，w） 有关键k。

类似地，对于其他每个元素（k，w），得到的RDD将包含对于自身中的v的所有对（k，（v，w）），或者对（k，（None，w）），如果没有元素 在自己有钥匙k。

哈希将生成的RDD划分成给定数量的分区。

>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2), ("c", 8)])

>>> sorted(x.fullOuterJoin(y).collect())

[('a', (1, 2)), ('b', (4, None)), ('c', (None, 8))]

**getCheckpointFile()**

获取此RDD被检查点的文件的名称

RDD在本地检查点未定义。

**getNumPartitions()**

返回RDD中的分区数

>>> rdd = sc.parallelize([1, 2, 3, 4], 2)

>>> rdd.getNumPartitions()

2

**getStorageLevel()**

获取RDD当前的存储级别。

>>> rdd1 = sc.parallelize([1,2])

>>> rdd1.getStorageLevel()

StorageLevel(False, False, False, False, 1)

>>> print(rdd1.getStorageLevel())

Serialized 1x Replicated

**glom() 合并分区的所有元素到新的列表**

返回通过将每个分区中的所有元素合并到列表中创建的RDD。

>>> rdd = sc.parallelize([1, 2, 3, 4], 2)

>>> sorted(rdd.glom().collect())

[[1, 2], [3, 4]]

**groupBy(f, numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

返回分组项目的RDD。

>>> rdd = sc.parallelize([1, 1, 2, 3, 5, 8])

>>> result = rdd.groupBy(lambda x: x % 2).collect()

>>> sorted([(x, sorted(y)) for (x, y) in result])

[(0, [2, 8]), (1, [1, 1, 3, 5])]

**groupByKey(numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

将RDD中每个键的值分组为单个序列。 哈希用numPartition分区分区生成的RDD。

注意

如果要分组以便在每个键上执行聚合（如总和或平均值），则使用reduceByKey或aggregateByKey将提供更好的性能。

>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

>>> sorted(rdd.groupByKey().mapValues(len).collect())

[('a', 2), ('b', 1)]

>>> sorted(rdd.groupByKey().mapValues(list).collect())

[('a', [1, 1]), ('b', [1])]

**groupWith(other, \*others)**

别名为cogroup，但支持多个RDD。

>>> w = sc.parallelize([("a", 5), ("b", 6)])

>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2)])

>>> z = sc.parallelize([("b", 42)])

>>> [(x, tuple(map(list, y))) for x, y in sorted(list(w.groupWith(x, y, z).collect()))]

[('a', ([5], [1], [2], [])), ('b', ([6], [4], [], [42]))]

**histogram(buckets)**

使用提供的桶计算直方图。 水桶都是向右开放的，除了最后一个是关闭的。 例如 [1,10,20,50]表示桶是[1,10] [10,20] [20,50]，这意味着1 <= x <10，10 <= x <20，20 <= x <= 50。 在1和50的输入上，我们将有一个1,0,1的直方图。

如果直方图均匀间隔（例如[0,10,20,30]），则可以从O（log n）插入切换到每个元素O（1）（其中n是桶的数量）。

桶必须排序，不包含任何重复项，并且至少有两个元素。

如果桶是一个数字，它将生成在RDD的最小值和最大值之间均匀间隔的桶。 例如，如果最小值为0且最大值为100，则给定桶为2，则所得到的桶将为[0,50] [50,100]。 桶必须至少为1.如果RDD包含无穷大，则会出现异常。 如果RDD中的元素不变（max == min），则将使用单个存储桶。

返回值是桶和直方图的元组。

>>> rdd = sc.parallelize(range(51))

>>> rdd.histogram(2)

([0, 25, 50], [25, 26])

>>> rdd.histogram([0, 5, 25, 50])

([0, 5, 25, 50], [5, 20, 26])

>>> rdd.histogram([0, 15, 30, 45, 60]) # evenly spaced buckets

([0, 15, 30, 45, 60], [15, 15, 15, 6])

>>> rdd = sc.parallelize(["ab", "ac", "b", "bd", "ef"])

>>> rdd.histogram(("a", "b", "c"))

(('a', 'b', 'c'), [2, 2])

id()

此RDD的唯一ID（在其SparkContext中）。

intersection(other)

返回这个RDD与另一个RDD。 输出不会包含任何重复的元素，即使输入RDD。

注意

该方法在内部执行随机播放。

>>> rdd1 = sc.parallelize([1, 10, 2, 3, 4, 5])

>>> rdd2 = sc.parallelize([1, 6, 2, 3, 7, 8])

>>> rdd1.intersection(rdd2).collect()

[1, 2, 3]

**isCheckpointed()**

返回此RDD是否可靠地或本地检查点和物化。

**isEmpty()**

当且仅当RDD根本不包含元素时才返回true。

Note

即使RDD至少有1个分区，RDD也可能为空。

>>> sc.parallelize([]).isEmpty()

True

>>> sc.parallelize([1]).isEmpty()

False

**isLocallyCheckpointed()**

返回是否将此RDD标记为本地检查点。

暴露于测试。

**join(other, numPartitions=None)**

返回一个RDD，其中包含所有成对元素，具有自我匹配和其他匹配键。

每对元素将作为（k，（v1，v2））元组返回，其中（k，v1）在自身中，（k，v2）在另一个元组中。

在集群中执行散列连接。

>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2), ("a", 3)])

>>> sorted(x.join(y).collect())

[('a', (1, 2)), ('a', (1, 3))]

**keyBy(f)**

通过应用f来创建RDD中元素的元组。

>>> x = sc.parallelize(range(0,3)).keyBy(lambda x: x\*x)

>>> y = sc.parallelize(zip(range(0,5), range(0,5)))

>>> [(x, list(map(list, y))) for x, y in sorted(x.cogroup(y).collect())]

[(0, [[0], [0]]), (1, [[1], [1]]), (2, [[], [2]]), (3, [[], [3]]), (4, [[2], [4]])]

**keys() （返回键）**

**用每个元组的键返回一个RDD。**

>>> m = sc.parallelize([(1, 2), (3, 4)]).keys()

>>> m.collect()

[1, 3]

**leftOuterJoin(other, numPartitions=None)**

执行自我和其他的左外连接。

对于自身中的每个元素（k，v），如果其他元素（k，（v，无））中没有元素，则所得到的RDD将包含其他的w或对（k，（v，None））中的所有对（k，（v，w） 有关键k。

哈希将生成的RDD划分成给定数量的分区。

>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2)])

>>> sorted(x.leftOuterJoin(y).collect())

[('a', (1, 2)), ('b', (4, None))]

**localCheckpoint()**

使用Spark的现有缓存层将此RDD标记为本地检查点。

该方法适用于希望截断RDD谱系的用户，同时跳过在可靠的分布式文件系统中复制实际数据的昂贵步骤。这对于需要定期截断的长谱系的RDD（例如GraphX）很有用。

本地检查点牺牲性能的容错性。特别地，检查点数据被写入执行器中的临时本地存储器，而不是可靠的容错存储器。效果是，如果执行程序在计算过程中失败，则检查点数据可能不再可访问，导致无法恢复的作业失败。

这与使用动态分配是不安全的，它会将执行程序与其缓存的块一起删除。如果您必须同时使用这两个功能，建议您将spark.dynamicAllocation.cachedExecutorIdleTimeout设置为较高的值。

通过SparkContext.setCheckpointDir（）设置的检查点目录不被使用。

**lookup(key)**

返回键盘RDD中的值列表。 如果RDD只有搜索密钥映射到的分区，才能有效地进行此操作。

>>> l = range(1000)

>>> rdd = sc.parallelize(zip(l, l), 10)

>>> rdd.lookup(42) # slow

[42]

>>> sorted = rdd.sortByKey()

>>> sorted.lookup(42) # fast

[42]

>>> sorted.lookup(1024)

[]

>>> rdd2 = sc.parallelize([(('a', 'b'), 'c')]).groupByKey()

>>> list(rdd2.lookup(('a', 'b'))[0])

['c']

**map(f, preservesPartitioning=False)**

通过对这个RDD的每个元素应用一个函数来返回一个新的RDD。

>>> rdd = sc.parallelize(["b", "a", "c"])

>>> sorted(rdd.map(lambda x: (x, 1)).collect())

[('a', 1), ('b', 1), ('c', 1)]

**mapPartitions(f, preservesPartitioning=False)**

通过对这个RDD的每个分区应用一个函数来返回一个新的RDD。

>>> rdd = sc.parallelize([1, 2, 3, 4], 2)

>>> def f(iterator): yield sum(iterator)

>>> rdd.mapPartitions(f).collect()

[3, 7]

**mapPartitionsWithIndex(f, preservesPartitioning=False)**

通过对此RDD的每个分区应用函数，同时跟踪原始分区的索引来返回新的RDD。

>>> rdd = sc.parallelize([1, 2, 3, 4], 4)

>>> def f(splitIndex, iterator): yield splitIndex

>>> rdd.mapPartitionsWithIndex(f).sum()

6

**mapPartitionsWithSplit(f, preservesPartitioning=False)**

**不推荐使用：使用mapPartitionsWithIndex代替。**

通过对此RDD的每个分区应用函数，同时跟踪原始分区的索引来返回新的RDD。

>>> rdd = sc.parallelize([1, 2, 3, 4], 4)

>>> def f(splitIndex, iterator): yield splitIndex

>>> rdd.mapPartitionsWithSplit(f).sum()

6

**mapValues(f) （不改键，仅改变该键的值）**

**通过地图函数传递键值对RDD中的每个值，而无需更改键; 这也保留了原始RDD的分区。**

>>> x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])

>>> def f(x): return len(x)

>>> x.mapValues(f).collect()

[('a', 3), ('b', 1)]

**max(key=None)**

查找此RDD中的最大项目。

|  |  |
| --- | --- |
| Parameters: | key – A function used to generate key for comparing |

>>> rdd = sc.parallelize([1.0, 5.0, 43.0, 10.0])

>>> rdd.max()

43.0

>>> rdd.max(key=str)

5.0

**mean()**

计算这个RDD元素的平均值。

>>> sc.parallelize([1, 2, 3]).mean()

2.0

**meanApprox(timeout, confidence=0.95)**

注意

试验

在超时期间返回平均值的近似操作或满足置信度。

>>> rdd = sc.parallelize(range(1000), 10)

>>> r = sum(range(1000)) / 1000.0

>>> abs(rdd.meanApprox(1000) - r) / r < 0.05

True

**min(key=None)**

找到此RDD中的最小项目。

|  |  |
| --- | --- |
| Parameters: | key – A function used to generate key for comparing |

>>> rdd = sc.parallelize([2.0, 5.0, 43.0, 10.0])

>>> rdd.min()

2.0

>>> rdd.min(key=str)

10.0

**name()**

返回此RDD的名称。

**partitionBy(numPartitions, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

使用指定的分区器返回分区的RDD副本。

>>> pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1]).map(lambda x: (x, x))

>>> sets = pairs.partitionBy(2).glom().collect()

>>> len(set(sets[0]).intersection(set(sets[1])))

0

**persist(storageLevel=StorageLevel(False, True, False, False, 1))**

设置此RDD的存储级别，以便在第一次计算时将其值跨操作持久化。 如果RDD尚未设置存储级别，则这只能用于分配新的存储级别。 如果没有指定存储级别默认值为（MEMORY\_ONLY）。

>>> rdd = sc.parallelize(["b", "a", "c"])

>>> rdd.persist().is\_cached

True

**pipe(command, env=None, checkCode=False)**

**将由管道元素创建的RDD返回到分叉的外部过程。**

>>> sc.parallelize(['1', '2', '', '3']).pipe('cat').collect()

[u'1', u'2', u'', u'3']

|  |  |
| --- | --- |
| Parameters: | checkCode – whether or not to check the return value of the shell command. |

**randomSplit(weights, seed=None)**

**用提供的权重随机分割此RDD。**

|  |  |
| --- | --- |
| Parameters: | * weights – weights for splits, will be normalized if they don’t sum to 1 * seed – random seed |
| Returns: | split RDDs in a list |

>>> rdd = sc.parallelize(range(500), 1)

>>> rdd1, rdd2 = rdd.randomSplit([2, 3], 17)

>>> len(rdd1.collect() + rdd2.collect())

500

>>> 150 < rdd1.count() < 250

True

>>> 250 < rdd2.count() < 350

True

**reduce(f)**

使用指定的交换和关联二进制运算符减少该RDD的元素。 目前在本地减少分区。

>>> from operator import add

>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add)

15

>>> sc.parallelize((2 for \_ in range(10))).map(lambda x: 1).cache().reduce(add)

10

>>> sc.parallelize([]).reduce(add)

Traceback (most recent call last):

...

ValueError: Can not reduce() empty RDD

**reduceByKey(func, numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>)**

使用关联和交换减少函数合并每个键的值。

在将结果发送到reducer之前，这也将在每个映射器上执行本地合并，类似于MapReduce中的“组合器”。

如果没有指定numPartitions，则输出将使用numPartition分区进行分区，或默认并行级别。 默认分区器是哈希分区。

>>> from operator import add

>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

>>> sorted(rdd.reduceByKey(add).collect())

[('a', 2), ('b', 1)]

**reduceByKeyLocally(func)**

使用关联和交换缩减功能合并每个键的值，但将结果立即作为字典返回主。

在将结果发送到reducer之前，这也将在每个映射器上执行本地合并，类似于MapReduce中的“组合器”。

>>> from operator import add

>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

>>> sorted(rdd.reduceByKeyLocally(add).items())

[('a', 2), ('b', 1)]

**repartition(numPartitions)**

**返回一个正好具有numPartition分区的新RDD。**

可以增加或减少此RDD中的并行级别。 在内部，它使用随机播放来重新分配数据。 如果您减少此RDD中的分区数，请考虑使用coalesce，这可以避免执行shuffle。

>>> rdd = sc.parallelize([1,2,3,4,5,6,7], 4)

>>> sorted(rdd.glom().collect())

[[1], [2, 3], [4, 5], [6, 7]]

>>> len(rdd.repartition(2).glom().collect())

2

>>> len(rdd.repartition(10).glom().collect())

10

**repartitionAndSortWithinPartitions(numPartitions=None, partitionFunc=<function portable\_hash at 0x7fdfde9fb578>, ascending=True, keyfunc=<function <lambda> at 0x7fdfde9f9de8>)**

根据给定的分区器重新分配RDD，并且在每个生成的分区中，通过它们的键对记录进行排序。

>>> rdd = sc.parallelize([(0, 5), (3, 8), (2, 6), (0, 8), (3, 8), (1, 3)])

>>> rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x: x % 2, 2)

>>> rdd2.glom().collect()

[[(0, 5), (0, 8), (2, 6)], [(1, 3), (3, 8), (3, 8)]]

**rightOuterJoin(other, numPartitions=None)**

**执行自我和其他的正确的外部联合。**

对于其他每个元素（k，w），如果在自身中没有元素，则所得到的RDD将包含v中的所有对（k，（v，w）），或者对（k，（无，w）） 有关键k。

哈希将生成的RDD划分成给定数量的分区。

>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2)])

>>> sorted(y.rightOuterJoin(x).collect())

[('a', (2, 1)), ('b', (None, 4))]

**sample(withReplacement, fraction, seed=None)**

返回此RDD的采样子集。

|  |  |
| --- | --- |
| Parameters: | * withReplacement – 可以对多个元素进行多次采样（取样时更换） * fraction – 样本的预期大小作为不包含替换的RDD大小的一部分：每个元素被选择的概率; 分数必须为[0，1]替换：每个元素选择的预期次数; 分数必须> = 0 * seed – 种子为随机数发生器 |

Note

这并不能保证提供给定DataFrame的总计数所指定的分数。

>>> rdd = sc.parallelize(range(100), 4)

>>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14

True

**sampleByKey(withReplacement, fractions, seed=None)**

返回通过键（通过分层采样）采样的RDD的子集。 使用分数指定的不同键的可变采样率创建此RDD的样本，采样率图的关键。

>>> fractions = {"a": 0.2, "b": 0.1}

>>> rdd = sc.parallelize(fractions.keys()).cartesian(sc.parallelize(range(0, 1000)))

>>> sample = dict(rdd.sampleByKey(False, fractions, 2).groupByKey().collect())

>>> 100 < len(sample["a"]) < 300 and 50 < len(sample["b"]) < 150

True

>>> max(sample["a"]) <= 999 and min(sample["a"]) >= 0

True

>>> max(sample["b"]) <= 999 and min(sample["b"]) >= 0

True

**sampleStdev()**

计算该RDD元素的样本标准偏差（通过除以N-1而不是N来校正标准差的偏差）。

>>> sc.parallelize([1, 2, 3]).sampleStdev()

1.0

**sampleVariance()**

计算该RDD元素的样本方差（通过除以N-1而不是N来校正方差估计中的偏差）。

>>> sc.parallelize([1, 2, 3]).sampleVariance()

1.0

**saveAsHadoopDataset(conf, keyConverter=None, valueConverter=None)**

使用旧的Hadoop OutputFormat API（映射的包），将任何Hadoop文件系统的键值对（RDD [（K，V）]）的Python RDD输出。 键/值使用用户指定的转换器转换为输出，或者默认情况下，

org.apache.spark.api.python.JavaToWritableConverter.

|  |  |
| --- | --- |
| Parameters: | * conf – Hadoop job configuration, passed in as a dict * keyConverter – (None by default) * valueConverter – (None by default) |

**saveAsHadoopFile(path, outputFormatClass, keyClass=None, valueClass=None, keyConverter=None, valueConverter=None, conf=None, compressionCodecClass=None)**

使用旧的Hadoop OutputFormat API（映射的包），将任何Hadoop文件系统的键值对（RDD [（K，V）]）的Python RDD输出。 如果未指定键值和值类型将被推断。 使用用户指定的转换器或org.apache.spark.api.python.JavaToWritableConverter将键和值转换为输出。 conf被应用在与该RDD的SparkContext相关联的基本Hadoop conf的顶部上，以创建用于保存数据的合并的Hadoop MapReduce作业配置。

|  |  |
| --- | --- |
| Parameters: | * path – path to Hadoop file * outputFormatClass – fully qualified classname of Hadoop OutputFormat (e.g. “org.apache.hadoop.mapred.SequenceFileOutputFormat”) * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.IntWritable”, None by default) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.Text”, None by default) * keyConverter – (None by default) * valueConverter – (None by default) * conf – (None by default) * compressionCodecClass – (None by default) |

**saveAsNewAPIHadoopDataset(conf, keyConverter=None, valueConverter=None)**

使用新的Hadoop OutputFormat API（mapreduce包），将任意一个键值对的RDD（RDD [（K，V）]）输出到任何Hadoop文件系统。 键/值使用用户指定的转换器转换为输出，或者默认情况下，

org.apache.spark.api.python.JavaToWritableConverter.

|  |  |
| --- | --- |
| Parameters: | * conf – Hadoop job configuration, passed in as a dict * keyConverter – (None by default) * valueConverter – (None by default) |

**saveAsNewAPIHadoopFile(path, outputFormatClass, keyClass=None, valueClass=None, keyConverter=None, valueConverter=None, conf=None)**

使用新的Hadoop OutputFormat API（mapreduce包），将任意一个键值对的RDD（RDD [（K，V）]）输出到任何Hadoop文件系统。 如果未指定键值和值类型将被推断。 使用用户指定的转换器或org.apache.spark.api.python.JavaToWritableConverter将键和值转换为输出。 conf被应用在与该RDD的SparkContext相关联的基本Hadoop conf的顶部上，以创建用于保存数据的合并的Hadoop MapReduce作业配置。

|  |  |
| --- | --- |
| Parameters: | * path – path to Hadoop file * outputFormatClass – fully qualified classname of Hadoop OutputFormat (e.g. “org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat”) * keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.IntWritable”, None by default) * valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.Text”, None by default) * keyConverter – (None by default) * valueConverter – (None by default) * conf – Hadoop job configuration, passed in as a dict (None by default) |

**saveAsPickleFile(path, batchSize=10)**

**将此RDD保存为序列化对象的SequenceFile。 使用的串行器是**

pyspark.serializers.PickleSerializer, default batch size is 10.

>>> tmpFile = NamedTemporaryFile(delete=True)

>>> tmpFile.close()

>>> sc.parallelize([1, 2, 'spark', 'rdd']).saveAsPickleFile(tmpFile.name, 3)

>>> sorted(sc.pickleFile(tmpFile.name, 5).map(str).collect())

['1', '2', 'rdd', 'spark']

**saveAsSequenceFile(path, compressionCodecClass=None)**

使用从RDD的键和值类型转换的org.apache.hadoop.io.Writable类型，将任意一个键值对的RDD（RDD [（K，V）]）输出到任何Hadoop文件系统。 机制如下：

1. Pyrolite用于将腌制的Python RDD转换为Java对象的RDD。
2. 此Java RDD的键和值将转换为Writables并写出。

|  |  |
| --- | --- |
| Parameters: | * path – path to sequence file * compressionCodecClass – (None by default) |

**saveAsTextFile(path, compressionCodecClass=None)**

将此RDD保存为文本文件，使用元素的字符串表示形式。

|  |  |
| --- | --- |
| Parameters: | * path – path to text file * compressionCodecClass – (None by default) string i.e. “org.apache.hadoop.io.compress.GzipCodec” |

>>> tempFile = NamedTemporaryFile(delete=True)

>>> tempFile.close()

>>> sc.parallelize(range(10)).saveAsTextFile(tempFile.name)

>>> from fileinput import input

>>> from glob import glob

>>> ''.join(sorted(input(glob(tempFile.name + "/part-0000\*"))))

'0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'

Empty lines are tolerated when saving to text files.

>>> tempFile2 = NamedTemporaryFile(delete=True)

>>> tempFile2.close()

>>> sc.parallelize(['', 'foo', '', 'bar', '']).saveAsTextFile(tempFile2.name)

>>> ''.join(sorted(input(glob(tempFile2.name + "/part-0000\*"))))

'\n\n\nbar\nfoo\n'

使用压缩代码

>>> tempFile3 = NamedTemporaryFile(delete=True)

>>> tempFile3.close()

>>> codec = "org.apache.hadoop.io.compress.GzipCodec"

>>> sc.parallelize(['foo', 'bar']).saveAsTextFile(tempFile3.name, codec)

>>> from fileinput import input, hook\_compressed

>>> result = sorted(input(glob(tempFile3.name + "/part\*.gz"), openhook=hook\_compressed))

>>> b''.join(result).decode('utf-8')

u'bar\nfoo\n'

**setName(name)**

为RDD命名

>>> rdd1 = sc.parallelize([1, 2])

>>> rdd1.setName('RDD1').name()

u'RDD1'

**sortBy(keyfunc, ascending=True, numPartitions=None)**

通过给定的keyfunc对这个RDD进行排序

>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]

>>> sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

>>> sc.parallelize(tmp).sortBy(lambda x: x[1]).collect()

[('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]

**sortByKey(ascending=True, numPartitions=None, keyfunc=<function <lambda> at 0x7fdfde9f9ed8>)**

对此RDD进行排序，该RDD被假定为由（键，值）对组成

>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]

>>> sc.parallelize(tmp).sortByKey().first()

('1', 3)

>>> sc.parallelize(tmp).sortByKey(True, 1).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

>>> sc.parallelize(tmp).sortByKey(True, 2).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

>>> tmp2 = [('Mary', 1), ('had', 2), ('a', 3), ('little', 4), ('lamb', 5)]

>>> tmp2.extend([('whose', 6), ('fleece', 7), ('was', 8), ('white', 9)])

>>> sc.parallelize(tmp2).sortByKey(True, 3, keyfunc=lambda k: k.lower()).collect()

[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9), ('whose', 6)]

**stats()**

返回一个StatCounter对象，在一个操作中捕获RDD元素的均值，方差和计数。

**stdev()**

计算该RDD元素的标准偏差。

>>> sc.parallelize([1, 2, 3]).stdev()

0.816...

**subtract(other, numPartitions=None)**

返回自身中不包含在其他的每个值。

>>> x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 3)])

>>> y = sc.parallelize([("a", 3), ("c", None)])

>>> sorted(x.subtract(y).collect())

[('a', 1), ('b', 4), ('b', 5)]

subtractByKey(other, numPartitions=None)

在自己中返回每个（键，值）对，其中没有配对的匹配键。

>>> x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 2)])

>>> y = sc.parallelize([("a", 3), ("c", None)])

>>> sorted(x.subtractByKey(y).collect())

[('b', 4), ('b', 5)]

**sum()**

添加此RDD中的元素。

>>> sc.parallelize([1.0, 2.0, 3.0]).sum()

6.0

**sumApprox(timeout, confidence=0.95)**

注意

试验

大概操作在超时时间内返回总和或满足置信度。

>>> rdd = sc.parallelize(range(1000), 10)

>>> r = sum(range(1000))

>>> abs(rdd.sumApprox(1000) - r) / r < 0.05

True

**take(num)**

拿起RDD的第一个num元素。

它首先扫描一个分区，并使用该分区的结果来估计满足限制所需的其他分区数。

从RDD＃take（）中的Scala实现翻译。

注意

只有当所产生的数组被预期为小时才会使用此方法，因为所有的数据都被加载到驱动程序的内存中。

>>> sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)

[2, 3]

>>> sc.parallelize([2, 3, 4, 5, 6]).take(10)

[2, 3, 4, 5, 6]

>>> sc.parallelize(range(100), 100).filter(lambda x: x > 90).take(3)

[91, 92, 93]

**takeOrdered(num, key=None)**

从按顺序排列或由可选键功能指定的RDD获取N个元素。

注意

只有当所产生的数组被预期为小时才会使用此方法，因为所有的数据都被加载到驱动程序的内存中。

>>> sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6)

[1, 2, 3, 4, 5, 6]

>>> sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7], 2).takeOrdered(6, key=lambda x: -x)

[10, 9, 7, 6, 5, 4]

**takeSample(withReplacement, num, seed=None)**

返回此RDD的固定大小采样子集。

注意

只有在生成的数组预计为小的时候才应该使用这个方法，因为所有的数据被加载到驱动程序的内存中。

>>> rdd = sc.parallelize(range(0, 10))

>>> len(rdd.takeSample(True, 20, 1))

20

>>> len(rdd.takeSample(False, 5, 2))

5

>>> len(rdd.takeSample(False, 15, 3))

10

**toDebugString()**

**此RDD的描述及其调试的递归依赖关系。**

**toLocalIterator()**

返回一个包含此RDD中所有元素的迭代器。 迭代器将消耗与RDD中最大分区一样多的内存。

>>> rdd = sc.parallelize(range(10))

>>> [x for x in rdd.toLocalIterator()]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**top(num, key=None) （take）**

**从RDD获取前N个元素。**

注意

只有在生成的数组预计为小的时候才应该使用这个方法，因为所有的数据被加载到驱动程序的内存中。

注意

它返回以降序排列的列表。

>>> sc.parallelize([10, 4, 2, 12, 3]).top(1)

[12]

>>> sc.parallelize([2, 3, 4, 5, 6], 2).top(2)

[6, 5]

>>> sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)

[4, 3, 2]

**treeAggregate(zeroValue, seqOp, combOp, depth=2)**

**以多级树模式聚合此RDD的元素。**

|  |  |
| --- | --- |
| Parameters: | depth – suggested depth of the tree (default: 2) |

>>> add = lambda x, y: x + y

>>> rdd = sc.parallelize([-5, -4, -3, -2, -1, 1, 2, 3, 4], 10)

>>> rdd.treeAggregate(0, add, add)

-5

>>> rdd.treeAggregate(0, add, add, 1)

-5

>>> rdd.treeAggregate(0, add, add, 2)

-5

>>> rdd.treeAggregate(0, add, add, 5)

-5

>>> rdd.treeAggregate(0, add, add, 10)

-5

**treeReduce(f, depth=2)**

**以多级树型模式减少此RDD的元素。**

|  |  |
| --- | --- |
| Parameters: | depth – suggested depth of the tree (default: 2) |

>>> add = lambda x, y: x + y

>>> rdd = sc.parallelize([-5, -4, -3, -2, -1, 1, 2, 3, 4], 10)

>>> rdd.treeReduce(add)

-5

>>> rdd.treeReduce(add, 1)

-5

>>> rdd.treeReduce(add, 2)

-5

>>> rdd.treeReduce(add, 5)

-5

>>> rdd.treeReduce(add, 10)

-5

**union(other)**

返回这个RDD的联合和另一个。

>>> rdd = sc.parallelize([1, 1, 2, 3])

>>> rdd.union(rdd).collect()

[1, 1, 2, 3, 1, 1, 2, 3]

**unpersist()**

将RDD标记为非持久性，并将其从内存和磁盘中删除。

**values()**

用每个元组的值返回一个RDD。

>>> m = sc.parallelize([(1, 2), (3, 4)]).values()

>>> m.collect()

[2, 4]

**variance()**

计算RDD的方差

>>> sc.parallelize([1, 2, 3]).variance()

0.666...

**zip(other)**

将此RDD与另一个RDD一起使用，返回与每个RDD中每个RDD第二个元素中的第一个元素的键值对等。假设两个RDD在每个分区中具有相同数量的分区和相同数量的元素（例如一个 是通过地图在另一个）。

>>> x = sc.parallelize(range(0,5))

>>> y = sc.parallelize(range(1000, 1005))

>>> x.zip(y).collect()

[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]

**zipWithIndex()**

使用其元素索引来拉伸此RDD。

排序首先基于分区索引，然后根据每个分区中的项目顺序。 因此，第一个分区中的第一个项目获取索引0，最后一个分区中的最后一个项目将收到最大的索引。

当此RDD包含多个分区时，此方法需要触发Spark作业。

>>> sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()

[('a', 0), ('b', 1), ('c', 2), ('d', 3)]

**zipWithUniqueId()**

使用生成的唯一长ids来压缩此RDD。

第k个分区中的项目将获得ids k，n + k，2 \* n + k，...，其中n是分区数。 所以可能存在差距，但是这种方法不会引发火花作业，这与zipWithIndex不同

>>> sc.parallelize(["a", "b", "c", "d", "e"], 3).zipWithUniqueId().collect()

[('a', 0), ('b', 1), ('c', 4), ('d', 2), ('e', 5)]

# StorageLevel

class pyspark.**StorageLevel**(useDisk, useMemory, useOffHeap, deserialized, replication=1)

用于控制RDD存储的标志。 每个StorageLevel记录是否使用内存，是否将RDD丢失到磁盘，如果它不在内存中，是否将数据保留在特定于JAVA的序列化格式的内存中，以及是否复制多个节点上的RDD分区。 还包含一些常用存储级别MEMORY\_ONLY的静态常量。 由于数据始终在Python端进行序列化，所有常量都使用序列化格式。

DISK\_ONLY = StorageLevel(True, False, False, False, 1)

DISK\_ONLY\_2 = StorageLevel(True, False, False, False, 2)

MEMORY\_AND\_DISK = StorageLevel(True, True, False, False, 1)

MEMORY\_AND\_DISK\_2 = StorageLevel(True, True, False, False, 2)

MEMORY\_AND\_DISK\_SER = StorageLevel(True, True, False, False, 1)

MEMORY\_AND\_DISK\_SER\_2 = StorageLevel(True, True, False, False, 2)

MEMORY\_ONLY = StorageLevel(False, True, False, False, 1)

MEMORY\_ONLY\_2 = StorageLevel(False, True, False, False, 2)

MEMORY\_ONLY\_SER = StorageLevel(False, True, False, False, 1)

MEMORY\_ONLY\_SER\_2 = StorageLevel(False, True, False, False, 2)

OFF\_HEAP = StorageLevel(True, True, True, False, 1)

# Broadcast

class pyspark.Broadcast(sc=None, value=None, pickle\_registry=None, path=None)

A broadcast variable created with [SparkContext.broadcast()](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext.broadcast). Access its value through value.、

使用SparkContext.broadcast（）创建的广播变量。 通过价值获取其价值。

Examples:

>>> from pyspark.context import SparkContext

>>> sc = SparkContext('local', 'test')

>>> b = sc.broadcast([1, 2, 3, 4, 5])

>>> b.value

[1, 2, 3, 4, 5]

>>> sc.parallelize([0, 0]).flatMap(lambda x: b.value).collect()

[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

>>> b.unpersist()

>>> large\_broadcast = sc.broadcast(range(10000))

**destroy()**

销毁与此广播变量相关的所有数据和元数据。 谨慎使用; 一旦广播变量被破坏，就不能再次使用。 该方法阻止，直到销毁完成。

**dump(value, f)**

**load(path)**

**unpersist(blocking=False)**

在执行器上删除此广播的缓存副本。 如果在调用该广播后使用广播，则需要将其重新发送给每个执行者。

|  |  |
| --- | --- |
| Parameters: | blocking – Whether to block until unpersisting has completed |

**value**

Return the broadcasted value

# Accumulator

class pyspark.Accumulator(aid, value, accum\_param)

可以累积的共享变量，即具有交换和关联的“添加”操作。 Spark群集上的工作任务可以使用+ =运算符将值添加到累加器，但只允许驱动程序使用值访问其值。 工作人员的更新将自动传播到驱动程序。

虽然SparkContext支持基本数据类型（如int和float）的累加器，但用户还可以通过提供自定义的AccumulatorParam对象来定义自定义类型的累加器。 请参阅本模块的doctest作为示例。

add(term)

将一个项添加到此累加器的值

value

获得累积值 仅在驱动程序中可用

# AccumulatorParam

class pyspark.AccumulatorParam

帮助对象定义如何累加给定类型的值。

**addInPlace**(value1, value2)

添加累加器数据类型的两个值，返回一个新值; 为了效率，还可以更新value1到位并返回。

zero(value)

为类型提供“零值”，在尺寸上与提供的值兼容（例如，零向量）

# MarshalSerializer

class pyspark.MarshalSerializer

使用Python的Marshal序列化器对对象进行序列化：

http://docs.python.org/2/library/marshal.html

这个串行器比PickleSerializer快，但支持更少的数据类型。

dumps(obj)

loads(obj)

# PickleSerializer

class pyspark.PickleSerializer

使用Python的pickle serializer对对象进行序列化：

http://docs.python.org/2/library/pickle.html

这个串行器支持几乎任何Python对象，但可能不如更专业的序列化器那么快。

dumps(obj)

loads(obj, encoding=None)

# StatusTracker

class pyspark.StatusTracker(jtracker)

用于监控作业和阶段进度的低级状态报告API。

这些API有意提供非常弱的一致性语义; 这些API的消费者应该准备处理空/丢失的信息。 例如，作业的阶段ID可能是已知的，但是状态API可能没有关于这些阶段的细节的任何信息，因此getStageInfo可能会为有效的阶段id返回None。

为了限制内存使用，这些API仅提供有关最近的作业/阶段的信息。 这些API将为最后一个spark.ui.retainedStages阶段和spark.ui.retainedJobs作业提供信息。

getActiveJobsIds()

返回包含所有活动作业的ids的数组。

**getActiveStageIds**()

返回一个包含所有活动阶段的ID的数组。

**getJobIdsForGroup(jobGroup=None)**

返回特定作业组中所有已知作业的列表。 如果jobGroup为None，则返回与作业组关联的所有已知作业。

返回的列表可能包含运行，失败和已完成的作业，并且可能会因此方法的调用而异。 该方法不保证其结果中元素的顺序。

**getJobInfo(jobId)**

返回SparkJobInfo对象，如果无法找到作业信息或垃圾回收，则返回None。

**getStageInfo(stageId)**

返回SparkStageInfo对象，如果无法找到舞台信息或垃圾回收，则返回None。

# SparkJobInfo

class pyspark.SparkJobInfo

展示关于Spark工作的信息。

class pyspark.SparkStageInfo

展示有关Spark Stages的信息。

# Profiler

class pyspark.Profiler(ctx)

注意

DeveloperApi类

PySpark支持自定义分析器，这样可以使用不同的分析器，并输出不同于BasicProfiler中提供的格式。

自定义分析器必须定义或继承以下方法：

配置文件 - 将生成某种系统配置文件。 统计 - 返回收集的统计数据。 转储 - 将配置文件转储到路径添加 - 将配置文件添加到现有累积配置文件

在创建SparkContext时选择profiler类

>>> from pyspark import SparkConf, SparkContext

>>> from pyspark import BasicProfiler

>>> class MyCustomProfiler(BasicProfiler):

... def show(self, id):

... print("My custom profiles for RDD:%s" % id)

...

>>> conf = SparkConf().set("spark.python.profile", "true")

>>> sc = SparkContext('local', 'test', conf=conf, profiler\_cls=MyCustomProfiler)

>>> sc.parallelize(range(1000)).map(lambda x: 2 \* x).take(10)

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> sc.parallelize(range(1000)).count()

1000

>>> sc.show\_profiles()

My custom profiles for RDD:1

My custom profiles for RDD:3

>>> sc.stop()

**dump(id, path)**

将配置文件转储到路径中，id是RDD ID

**profile(func)**

对功能func进行分析

**show(id)**

将配置文件统计信息打印到stdout，id是RDD ID

**stats()**

返回收集的分析统计信息（pstats.Stats）

# BasicProfiler

class pyspark.BasicProfiler(ctx)

BasicProfiler是默认的分析器，它是基于cProfile和累加器实现的

**profile(func)**

运行和配置传入的方法to\_profile。返回一个配置文件对象。

**stats()**