

Spark Programming Guide

Spark编程指南

zhen-peng.yang@hpe.com | 2016-11-04

Contents

[Overview（概述） 3](#_Toc469484838)

[Linking with Spark（接入 Spark） 3](#_Toc469484839)

[Initializing Spark（初始化 Spark） 5](#_Toc469484840)

[Using the Shell（使用 SHELL） 6](#_Toc469484841)

[Resilient Distributed Datasets (RDDs) 8](#_Toc469484842)

[Parallelized Collections（并行集合） 8](#_Toc469484843)

[External Datasets（外部数据集） 9](#_Toc469484844)

[RDD Operations（RDD操作） 14](#_Toc469484845)

[Basics（基本操作） 14](#_Toc469484846)

[Passing Functions to Spark（将函数传递给Spark） 15](#_Toc469484847)

[Understanding closures（理解闭包） 18](#_Toc469484848)

[Working with Key-Value Pairs（使用键值对） 21](#_Toc469484849)

[Spark 的 rdd 的 action 操作 reducebykey 22](#_Toc469484850)

[Spark的RDD的action操作reduce 24](#_Toc469484851)

[Transformations（转换） 25](#_Toc469484852)

[Actions（动作） 39](#_Toc469484853)

[Shuffle operations（shuffle操作） 47](#_Toc469484854)

[RDD Persistence（RDD持久化） 49](#_Toc469484855)

[Which Storage Level to Choose?（存储级别的选择） 51](#_Toc469484856)

[Removing Data（移除数据） 52](#_Toc469484857)

[Shared Variables（共享变量） 52](#_Toc469484858)

[Broadcast Variables（广播变量） 53](#_Toc469484859)

[Accumulators（累加器） 54](#_Toc469484860)

[Deploying to a Cluster（发布到集群） 58](#_Toc469484861)

[Launching Spark jobs from Java / Scala 59](#_Toc469484862)

[Unit Testing 59](#_Toc469484863)

[Where to Go from Here 59](#_Toc469484864)

# Overview（概述）

At a high level, every Spark application consists of a driver program that runs the user’s main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

总的来说，每一个 Spark 应用程序都由一个*驱动程序* 组成，该程序运行用户的 main 函数，在集群中执行各种各样的 *并行操作* 。Spark 提出的最主要的抽象概念是弹性分布式数据集（RDD），它是元素的集合，划分到集群的各个节点上，并且可以被并行操作。RDD的创建可以从 HDFS( 或者其他任意支持 Hadoop 文件系统 ) 上的一个文件开始，或者通过转换驱动程序（driver program）中已经存在的Scala集合而来。用户也可以让 Spark 保留一个 RDD 在内存中，使其能在并行操作中被有效的重复使用。最后， RDD 能自动从节点故障中恢复。

A second abstraction in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only “added” to, such as counters and sums.

Spark 的第二个抽象概念是共享变量（shared variables），可以在并行操作中使用。在默认情况下，Spark 通过不同节点上的一系列任务来运行一个函数，它将每一个函数中用到的变量的拷贝传递到每一个任务中。有时候，一个变量需要在任务之间，或任务与驱动程序之间被共享。 Spark支持两种类型的共享变量：广播变量 （broadcast variables ），可以在内存的所有的结点上缓存变量；累加器（ accumulators ）：只能用于做加法的变量，例如计数或求和。

This guide shows each of these features in each of Spark’s supported languages. It is easiest to follow along with if you launch Spark’s interactive shell – either bin/spark-shell for the Scala shell or bin/pyspark for the Python one.

本指南将用 Spark 支持的每一种语言对这些特性做一一展示，并给出一些例子。你也可以通过打开Spark 交互式 shell（Scala 语言用 bin/spark-shell， Python 语言用 bin/pyspark ）来学习Spark。

# Linking with Spark（接入 Spark）

Spark 2.0.2 is built and distributed to work with Scala 2.11 by default. (Spark can be built to work with other versions of Scala, too.) To write applications in Scala, you will need to use a compatible Scala version (e.g. 2.11.X).

Spark2.0.1 默认情况下需要搭配使用 Scala2.11 （ Spark 也可以与其他版本的 Scala 搭配使用），如果用 Scala 来编写应用，你需要使用一个兼容版本的 Scala （比如 2.11.X 版本）。

To write a Spark application, you need to add a Maven dependency on Spark. Spark is available through Maven Central at:

要编写一个 Spark 应用，你需要给 Spark 添加一个 Maven 的依赖。Spark 可以通过 Maven 中心库来获得：

groupId = org.apache.spark

artifactId = spark-core\_2.11

version = 2.0.2

In addition, if you wish to access an HDFS cluster, you need to add a dependency on hadoop-client for your version of HDFS.

另外，如果你想访问一个HDFS集群，你需要根据你的 HDFS 版本，添加一个 hadoop-client的依赖：

groupId = org.apache.hadoop

artifactId = hadoop-client

version = <your-hdfs-version>

Finally, you need to import some Spark classes into your program. Add the following lines:

最后，你需要使用下面的语句在你的程序中引入一些 Spark 类。

**import** **org.apache.spark.SparkContext**

**import** **org.apache.spark.SparkConf**

(Before Spark 1.3.0, you need to explicitly import org.apache.spark.SparkContext.\_ to enable essential implicit conversions.)

以上为 Scala 语言接入 Spark 的方法，下面讲述 Python 语言：

Spark 2.0.1 works with Python 2.6+ or Python 3.4+. It can use the standard CPython

interpreter, so C libraries like NumPy can be used. It also works with PyPy 2.3+.

Spark2.0.1支持Python2.6 或者Python3.4及以上版本。它可以使用标准的CPython解析器，因此 C 的一些标准库如 NumPy 也可以使用。它也支持 PyPy2.3 及以上版本。

To run Spark applications in Python, use the bin/spark-submit script located in the Spark

directory. This script will load Spark’s Java/Scala libraries and allow you to submit

applications to a cluster. You can also use bin/pyspark to launch an interactive Python

shell.

可以调用Spark目录中的 bin/spark-submit脚本来运行Spark应用。这个脚本可以加载Spark的Java/Scala库来让你在集群中提交应用。你也可以使用bin/pyspark来运行Python交互式shell。

If you wish to access HDFS data, you need to use a build of PySpark linking to your version

of HDFS. Prebuilt packagesare also available on the Spark homepage for common HDFS

versions.

如果你想访问 HDFS 的数据，你需要根据你的 HDFS 版本来构建 PySpark 并实现接入。

Finally, you need to import some Spark classes into your program. Add the following line:

**from** **pyspark** **import** SparkContext, SparkConf

PySpark requires the same minor version of Python in both driver and workers. It uses the

default python version in PATH, you can specify which version of Python you want to use

by PYSPARK\_PYTHON, for example:

PySpark对驱动和代码都要求与Python保持相同的子版本。它使用PATH中Python的默认版本，你也可以使用PYSPARK\_PYTHON来指定你需要使用的Python版本，比如：

$ PYSPARK\_PYTHON=python3.4 bin/pyspark

$ PYSPARK\_PYTHON=/opt/pypy-2.5/bin/pypy bin/spark-submit examples/src/main/python/pi.py

# Initializing Spark（初始化 Spark）

The first thing a Spark program must do is to create a [SparkContext](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext) object, which tells Spark how to access a cluster. To create a SparkContextyou first need to build a [SparkConf](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkConf) object that contains information about your application.

Spark 程序需要做的第一件事情，就是创建一个SparkContext对象，它将告诉Spark如何访问一个集群。要创建一个SparkContext，你首先需要构建一个包含你应用信息的SparkConf对象。

Only one SparkContext may be active per JVM. You must stop() the active SparkContext before creating a new one.

每个JVM可能只有一个SparkContext是有效的，因此在创建新的SparkContext之前需要使用stop（）来结束有效的SparkContext对象。

Scala：

**val** conf **=** **new** **SparkConf**().setAppName(appName).setMaster(master)

**new** **SparkContext**(conf)

Python：

conf = SparkConf().setAppName(appName).setMaster(master)

sc = SparkContext(conf=conf)

The appName parameter is a name for your application to show on the cluster UI. master is a [Spark, Mesos or YARN cluster URL](http://spark.apache.org/docs/latest/submitting-applications.html#master-urls), or a special “local” string to run in local mode. In practice, when running on a cluster, you will not want to hardcode master in the program, but rather [launch the application with spark-submit](http://spark.apache.org/docs/latest/submitting-applications.html) and receive it there. However, for local testing and unit tests, you can pass “local” to run Spark in-process.

参数appName是在集群 UI 中显示的你应用的名字，master是一个[Spark,Mesos或YARN集群的 URL](https://spark.apache.org/docs/latest/submitting-applications.html#master-urls)，或者local模式运行的特殊字符串“local”。 实践中，当程序运行在集群中时，不需要在程序中硬编码master，而是使用spark-submit启动应用。然而对于本地测试和单元测试，你可以在进程内将”local”传给Spark。

## Using the Shell（使用 SHELL）

Scala 语言：

In the Spark shell, a special interpreter-aware SparkContext is already created for you, in the variable called sc. Making your own SparkContext will not work. You can set which master the context connects to using the --master argument, and you can add JARs to the classpath by passing a comma-separated list to the --jars argument. You can also add dependencies (e.g. Spark Packages) to your shell session by supplying a comma-separated list of maven coordinates to the --packages argument. Any additional repositories where dependencies might exist (e.g. SonaType) can be passed to the --repositories argument. For example, to run bin/spark-shell on exactly four cores, use:

使用Spark shell时 ,一个特殊的交互式的SparkContext已经为你创建好，叫做sc变量。不能自己创建SparkContext。你可以使用--master参数指定context连接的master。你可以通过把逗号分隔的jar列表赋给—jars参数来给classpath中增加外部JAR。你也可以使用--packages 参数给shell会话添加依赖包。任何包含依赖的资源库都可以传给--repositories参数。例如在四个core上运行bin/spark-shell:

$ ./bin/spark-shell --master local[4]

Or, to also add code.jar to its classpath, use:

也可以增加 code.jar:

$ ./bin/spark-shell --master local[4] --jars code.jar

To include a dependency using maven coordinates:

$ ./bin/spark-shell --master local[4] --packages "org.example:example:0.1"

For a complete list of options, run spark-shell --help. Behind the scenes, spark-shell invokes the more general [spark-submit script](http://spark.apache.org/docs/latest/submitting-applications.html).

运行 spark-shell --help 查看更多的参数。

Python 语言：

In the PySpark shell, a special interpreter-aware SparkContext is already created for you, in the variable called sc. Making your own SparkContext will not work. You can set which master the context connects to using the --master argument, and you can add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to --py-files. You can also add dependencies (e.g. Spark Packages) to your shell session by supplying a comma-separated list of maven coordinates to the --packages argument. Any additional repositories where dependencies might exist (e.g. SonaType) can be passed to the --repositories argument. Any python dependencies a Spark Package has (listed in the requirements.txt of that package) must be manually installed using pip when necessary. For example, to run bin/pyspark on exactly four cores, use:

使用PySpark shell时,一个特殊的交互式的SparkContext已经为你创建好，叫做sc变量。不能自己创建SparkContext。你可以使用--master参数指定context连接的master。你可以通过  
—py-files参数来给运行时路径中增加外部Python .zip，.egg和.py文件。你也可以使用  
--packages参数给shell会话添加依赖包。任何包含依赖的资源库都可以传给--repositories参数。一个Spark包的任何依赖在必要时都必须用pip手动安装好。例如在四个core上运行bin/pyspark：

$ ./bin/pyspark --master local[4]

Or, to also add code.py to the search path (in order to later be able to import code), use:

或者搜索路径中添加 code.py （以便后续引入代码）：

$ ./bin/pyspark --master local[4] --py-files code.py

For a complete list of options, run pyspark --help. Behind the scenes, pyspark invokes the more general [spark-submit script](http://spark.apache.org/docs/latest/submitting-applications.html).

It is also possible to launch the PySpark shell in [IPython](http://ipython.org/), the enhanced Python interpreter. PySpark works with IPython 1.0.0 and later. To use IPython, set the PYSPARK\_DRIVER\_PYTHON variable to ipython when running bin/pyspark:

在IPython中也可以调用PySpark shell，IPython是增强的Python解析器。PySpark支持IPython 1.0.0及以上版本。如果要使用IPython，可以在运行bin/pyspark时设置PYSPARK\_DRIVER\_PYTHON的值为ipython。

$ PYSPARK\_DRIVER\_PYTHON=ipython ./bin/pyspark

To use the Jupyter notebook (previously known as the IPython notebook),

$ PYSPARK\_DRIVER\_PYTHON=jupyter ./bin/pyspark

You can customize the ipython or jupyter commands by setting PYSPARK\_DRIVER\_PYTHON\_OPTS.

After the Jupyter Notebook server is launched, you can create a new “Python 2” notebook from the “Files” tab. Inside the notebook, you can input the command %pylab inline as part of your notebook before you start to try Spark from the Jupyter notebook.

# Resilient Distributed Datasets (RDDs)

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

Spark围绕的概念是弹性分布式数据集（RDD），这是一个有容错机制并可以被并行操作的元素集合。可以通过两种方式创建RDD：

1.在驱动程序中并行操作一个已经存在的集合

2.引用一个外部存储系统的数据集，比如共享文件系统，HDFS，HBase或者任意支持Hadoop存储系统的数据源。

## Parallelized Collections（并行集合）

Parallelized collections are created by calling SparkContext’s parallelize method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:

并行集合是通过调用SparkContext的Parallelize方法，在一个已经存在的集合（对于Python而言，是迭代器或者集合）上创建的。集合的对象将会被拷贝，创建出一个可以被并行操作的分布式数据集。例如，下面的解释器输出，演示了如何从一个数组（1到5）创建一个并行集合：

Scala：

**val** data **=** **Array**(1, 2, 3, 4, 5)

**val** distData **=** sc.parallelize(data)

Python：

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

distData = sc.parallelize(data)

Once created, the distributed dataset (distData) can be operated on in parallel. For example, we might call distData.reduce((a, b) => a + b) to add up the elements of the array. We describe operations on distributed datasets later on.

一旦分布式数据集被创建好，它们将可以被并行操作，比如我们可以调用distData.reduce((a,b)=>a+b)来将数组的元素相加。我们后续将讲述分布式数据集操作。Python语言，调用语句为：distData.reduce(lambda a,b: a+b)。

One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to parallelize (e.g. sc.parallelize(data, 10)). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility.

并行集合的一个重要参数是数据集切分的分区数。Spark将会在集群上为每一份数据起一个任务。典型地，你可以在集群的每个CPU上分布2-4个分区。一般来说，Spark会尝试根据集群的状况，来自动设定分区的数目。当然，你也可以通过传递给parallelize的第二个参数（例如：sc.parallelize(data, 10)）来手动设置。需要注意的是：有些地方在代码中使用术语slices（分区的一个同义词）来保持向后兼容性。

## External Datasets（外部数据集）

Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, [Amazon S3](http://wiki.apache.org/hadoop/AmazonS3), etc. Spark supports text files, [SequenceFiles](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html), and any other Hadoop [InputFormat](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/InputFormat.html).

Spark（ PySpark）可以从Hadoop支持的存储源创建数据集（包括本地文件，HDFS，Cassandra，HBase，Amazon S3等）。Spark支持TextFile，SequenceFile以及任何其他的Hadoop输入格式。

Text file RDDs can be created using SparkContext’s textFile method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines. Here is an example invocation:

文本文件的RDD是可以通过SparkContext的TextFile方法创建，该方法接受一个文件的URI地址（或者机器上的一个本地路径，或者一个hdfs://， s3n://等其他URI）作为参数，读取行的数据集。下面是一个调用例子：

Scala：

scala> **val** distFile **=** sc.textFile("data.txt")

distFile**:** org.apache.spark.rdd.RDD[String] **=** data.txt **MapPartitionsRDD**[10] at textFile at <console**>:**26

Python：

>>> distFile = sc.textFile("data.txt")

Once created, distFile can be acted on by dataset operations. For example, we can add up the sizes of all the lines using the map and reduceoperations as follows: distFile.map(s => s.length).reduce((a, b) => a + b). For Python, it would be: distFile.map(lambda s:len(s)).reduce(lambda a, b: a + b).

一旦创建完成，distFile可以被用于数据集操作。例如，我们可以通过使用如下的map和reduce操作：

Scala: distFile.map(s => s.length).reduce((a, b) => a + b)

Python: distFile.map(lambda s:len(s)).reduce(lambda a, b: a + b)

来将所有数据行的长度相加。

Some notes on reading files with Spark:

读取文件时的一些注意点：

* If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system.
* 如果使用本地文件系统，worker节点上相同路径下的文件必须能够访问。可以将文件复制到所有的worker节点或者使用网络共享文件系统。
* All of Spark’s file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/\*.txt"), and textFile("/my/directory/\*.gz").
* 所有Spark的基于文件的输入方法，包括TextFile，支持在文件夹、压缩文件和通配符上运行。比如你可以使用TextFile(“/my/directory”)，textFile(“/my/directory/.TXT”)和textFile(“/my/directory/.gz”)。
* The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 64MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks.
* textFile方法也可以接受一个可选的第二个参数，来控制文件的分区数。默认情况下，Spark为每一块文件创建一个分区（HDFS默认块大小为64MB），不过你也可以通过传入一个更大的值来制定更高的分区数量。注意分区数不能比块数小。

Apart from text files, Spark’s Scala API also supports several other data formats:

除了文本文件，Spark的Scala API也支持其他的数据格式：

* SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file.
* SparkContext.wholeTextFile 允许你读取文件夹下多个小的文本文件，并按照文件名 / 内容对的方式返回。相比之下TextFile能够返回每个文件的每行的一个记录。
* For [SequenceFiles](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html), use SparkContext’s sequenceFile[K, V] method where K and V are the types of key and values in the file. These should be subclasses of Hadoop’s [Writable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Writable.html) interface, like [IntWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/IntWritable.html) and [Text](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Text.html). In addition, Spark allows you to specify native types for a few common Writables; for example, sequenceFile[Int, String] will automatically read IntWritables and Texts.
* 对于SequenceFile，使用SparkContext的SequenceFile[K,V]方法来创建，其中K和V是文件中的key和value的类型。像IntWritable和Text一样，它们必须是Hadoop的Writable接口的子类。另外，对于几种通用的Writable类型，Spark允许你指定原生类型来替代。例如：sequencFile[Int, String] 将会自动读取 IntWritable 和 Texts 。
* For other Hadoop InputFormats, you can use the SparkContext.hadoopRDD method, which takes an arbitrary JobConf and input format class, key class and value class. Set these the same way you would for a Hadoop job with your input source. You can also use SparkContext.newAPIHadoopRDD for InputFormats based on the “new” MapReduce API (org.apache.hadoop.mapreduce).
* 对于其他类型的 Hadoop 输入格式，你可以使用SparkContext.hadoopRDD方法，它接收任意类型的JobConf和输入格式类、键类型和值类型。按照像 Hadoop作业一样的方法，来设置输入源就可以了。你也可以使用SparkContext.newHadoopRDD作为输入格式， 它基于新的MapReduce API(org.apache.hadoop.mapreduce).
* RDD.saveAsObjectFile and SparkContext.objectFile support saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD.
* RDD.saveAsObjectFile and SparkContext.objectFile 支持保存 RDD 为包含序列化的 Java对象的一个简单格式。然而它不比像 Avro 这种专业格式高效，但是它提供了一个容易的方式来保存 RDD 。

除了文本文件，Spark的Python API也支持其他的数据格式：

* SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file.
* RDD.saveAsPickleFile and SparkContext.pickleFile support saving an RDD in a simple format consisting of pickled Python objects. Batching is used on pickle serialization, with default batch size 10.
* SequenceFile and Hadoop Input/Output Formats

**Note** this feature is currently marked Experimental and is intended for advanced users. It may be replaced in future with read/write support based on Spark SQL, in which case Spark SQL is the preferred approach.

**Writable Support**

PySpark SequenceFile support loads an RDD of key-value pairs within Java, converts Writables to base Java types, and pickles the resulting Java objects using [Pyrolite](https://github.com/irmen/Pyrolite/). When saving an RDD of key-value pairs to SequenceFile, PySpark does the reverse. It unpickles Python objects into Java objects and then converts them to Writables. The following Writables are automatically converted:

|  |  |
| --- | --- |
| **Writable Type** | **Python Type** |
| Text | unicode str |
| IntWritable | int |
| FloatWritable | float |
| DoubleWritable | float |
| BooleanWritable | bool |
| BytesWritable | bytearray |
| NullWritable | None |
| MapWritable | dict |

Arrays are not handled out-of-the-box. Users need to specify custom ArrayWritable subtypes when reading or writing. When writing, users also need to specify custom converters that convert arrays to custom ArrayWritable subtypes. When reading, the default converter will convert custom ArrayWritable subtypes to Java Object[], which then get pickled to Python tuples. To get Python array.array for arrays of primitive types, users need to specify custom converters.

**Saving and Loading SequenceFiles**

Similarly to text files, SequenceFiles can be saved and loaded by specifying the path. The key and value classes can be specified, but for standard Writables this is not required.

>>> rdd = sc.parallelize(range(1, 4)).map(**lambda** x: (x, "a" \* x ))

>>> rdd.saveAsSequenceFile("path/to/file")

>>> sorted(sc.sequenceFile("path/to/file").collect())

[(1, u'a'), (2, u'aa'), (3, u'aaa')]

**Saving and Loading Other Hadoop Input/Output Formats**

PySpark can also read any Hadoop InputFormat or write any Hadoop OutputFormat, for both ‘new’ and ‘old’ Hadoop MapReduce APIs. If required, a Hadoop configuration can be passed in as a Python dict. Here is an example using the Elasticsearch ESInputFormat:

$ SPARK\_CLASSPATH=/path/to/elasticsearch-hadoop.jar ./bin/pyspark

>>> conf = {"es.resource" : "index/type"} *# assume Elasticsearch is running on localhost defaults*

>>> rdd = sc.newAPIHadoopRDD("org.elasticsearch.hadoop.mr.EsInputFormat",\

"org.apache.hadoop.io.NullWritable", "org.elasticsearch.hadoop.mr.LinkedMapWritable", conf=conf)

>>> rdd.first() *# the result is a MapWritable that is converted to a Python dict*

(u'Elasticsearch ID',

{u'field1': True,

u'field2': u'Some Text',

u'field3': 12345})

Note that, if the InputFormat simply depends on a Hadoop configuration and/or input path, and the key and value classes can easily be converted according to the above table, then this approach should work well for such cases.

If you have custom serialized binary data (such as loading data from Cassandra / HBase), then you will first need to transform that data on the Scala/Java side to something which can be handled by Pyrolite’s pickler. A [Converter](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.python.Converter) trait is provided for this. Simply extend this trait and implement your transformation code in the convert method. Remember to ensure that this class, along with any dependencies required to access your InputFormat, are packaged into your Spark job jar and included on the PySpark classpath.

See the [Python examples](https://github.com/apache/spark/tree/master/examples/src/main/python) and the [Converter examples](https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples/pythonconverters) for examples of using Cassandra / HBase InputFormat and OutputFormat with custom converters.

## RDD Operations（RDD操作）

RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).

RDD支持两种操作：转换（transformation）：从现有的数据集创建一个新的数据集；动作（action）：在数据集上运行计算后，返回一个值给驱动程序。例如，map就是一种转换，它将数据集的每一个元素都传递给函数，并返回一个新的分布式数据集表示结果。另一方面，reduce是一种动作，通过一些函数将所有的元素叠加起来，并将最终结果返回给驱动程序。（不过还有一个并行的reduceByKey，能返回一个分布式数据集）

All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

Spark中的所有转换都是惰性的，也就是说，它们并不会直接计算结果。相反地，它们只是记住应用到基础数据集（例如一个文件）上的这些转换动作，只有当发生一个要求返回结果给驱动程序的动作时，这些转换才会真正运行。这个设计让Spark更加有效率的运行。例如，我们可以实现：通过map创建一个新的数据集，并在reduce中使用，最终只返回reduce的结果给驱动程序，而不是整个大的数据集。

By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

默认情况下，每一个转换过的RDD都会在你对它执行一个动作时被重新计算。不过，你也可以使用persist（或者cache）方法，持久化一个RDD在内存中。在这种情况下，Spark将会在集群中，保存相关元素，下次你查询这个RDD时，她将能更快速访问。在磁盘上持久化数据集，或在集群间复制数据集也是支持的。

### Basics（基本操作）

To illustrate RDD basics, consider the simple program below:

下面的代码演示了RDD的基本操作：

Scala**：**

**val** lines **=** sc.textFile("data.txt")

**val** lineLengths **=** lines.map(s **=>** s.length)

**val** totalLength **=** lineLengths.reduce((a, b) **=>** a + b)

Python：

lines = sc.textFile("data.txt")

lineLengths = lines.map(**lambda** s: len(s))

totalLength = lineLengths.reduce(**lambda** a, b: a + b)

The first line defines a base RDD from an external file. This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file. The second line defines lineLengths as the result of a map transformation.

第一行从一个外部文件创建了一个基本的RDD对象。这个数据集并没有加载到内存中，行只不过是一个指向文件的指针，代码第二行定义行长度作为map的结果。

Again, lineLengths is not immediately computed, due to laziness. Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction, returning only its answer to the driver program.

此外，行长度由于惰性设计并没有立即计算。最后我们运行reduce，它是一个action。这时Spark将计算分解成运行在各个节点的任务，每个节点运行它的map部分以及一个本地的reduction，并仅将它的结果返回给驱动程序。

If we also wanted to use lineLengths again later, we could add:

如果你想再使用行长度，我们可以在reduce之前增加：

lineLengths.persist()

before the reduce, which would cause lineLengths to be saved in memory after the first time it is computed.

它可以在lineLengths第一次计算之前被保存在内存中。

### Passing Functions to Spark（将函数传递给Spark）

Spark’s API relies heavily on passing functions in the driver program to run on the cluster. There are two recommended ways to do this:

Spark API非常依赖于在集群中运行的驱动程序中传递函数，对于Scala来说有两种方式实现：

* [Anonymous function syntax](http://docs.scala-lang.org/tutorials/tour/anonymous-function-syntax.html), which can be used for short pieces of code.
* 匿名函数语法，可以用作简短的代码。
* Static methods in a global singleton object. For example, you can define object MyFunctions and then pass MyFunctions.func1, as follows:
* 全局单例对象的静态方法。例如，你可以定义MyFunctions对象，然后调用MyFunction.func1。

**object** **MyFunctions** {

**def** func1(s**:** String)**:** String = { ... }

}

myRdd.map(**MyFunctions**.func1)

对于Python来说有三种方式实现：

* [Lambda expressions](https://docs.python.org/2/tutorial/controlflow.html#lambda-expressions), for simple functions that can be written as an expression. (Lambdas do not support multi-statement functions or statements that do not return a value.)
* Lambda表达式，简单的函数可以写成表达式（lambda不支持多语句的函数或者不带返回值的语句）
* Local defs inside the function calling into Spark, for longer code.
* 对于长代码来说，将函数内的局部变量defs调用到Spark中。
* Top-level functions in a module.
* 模块内的高阶函数

For example, to pass a longer function than can be supported using a lambda, consider the code below:

比如，可以用下面的代码传递一个支持lambda的长函数：

*"""MyScript.py"""*

**if** \_\_name\_\_ == "\_\_main\_\_":

**def** myFunc(s):

words = s.split(" ")

**return** len(words)

sc = SparkContext(...)

sc.textFile("file.txt").map(myFunc)

Note that while it is also possible to pass a reference to a method in a class instance (as opposed to a singleton object), this requires sending the object that contains that class along with the method. For example, consider:

需要注意的是，虽然可以在一个类的实例中传递一个方法的引用（与单例对象相反），这里要求把包含那个类的对象连同方法一起发送，例如：

Scala：

**class** **MyClass** {

**def** func1(s**:** String)**:** String = { ... }

**def** doStuff(rdd**:** RDD[String])**:** RDD[String] **=** { rdd.map(func1) }

}

Python：

**class** **MyClass**(object):

**def** func(self, s):

**return** s

**def** doStuff(self, rdd):

**return** rdd.map(self.func)

Here, if we create a new MyClass instance and call doStuff on it, the map inside there references the func1 method of that*MyClass*instance, so the whole object needs to be sent to the cluster. It is similar to writing rdd.map(x => this.func1(x)).

这里，如果我们创建一个新的MyClass实例并对它调用doStuff方法，它内部的map将引用MyClass实例的func1方法，因此需要将整个对象发送给集群，这与用Scala写rdd.map(x=>this.func1(x))类似。

In a similar way, accessing fields of the outer object will reference the whole object:

同样地，访问外部对象的字段需要引用整个对象：

Scala：

**class** **MyClass** {

**val** field **=** "Hello"

**def** doStuff(rdd**:** RDD[String])**:** RDD[String] **=** { rdd.map(x **=>** field + x) }

}

is equivalent to writing rdd.map(x => this.field + x), which references all of this.

Python：

**class** **MyClass**(object):

**def** \_\_init\_\_(self):

self.field = "Hello"

**def** doStuff(self, rdd):

**return** rdd.map(**lambda** s: self.field + s)

To avoid this issue, the simplest way is to copy field into a local variable instead of accessing it externally:

为了解决这个问题，最简单的方式就是将字段赋值给本地变量，而不是外部访问它：

Scala：

**def** doStuff(rdd**:** RDD[String])**:** RDD[String] **=** {

**val** field\_ **=** **this**.field

rdd.map(x **=>** field\_ + x)

}

Python：

**def** doStuff(self, rdd):

field = self.field

**return** rdd.map(**lambda** s: field + s)

### Understanding closures（理解闭包）

One of the harder things about Spark is understanding the scope and life cycle of variables and methods when executing code across a cluster. RDD operations that modify variables outside of their scope can be a frequent source of confusion. In the example below we’ll look at code that uses foreach() to increment a counter, but similar issues can occur for other operations as well.

Spark的一个难点是理解在集群中执行代码时变量和方法的作用域和生命周期。RDD在变量作用域之外修改变量操作经常是一个容易产生困惑的地方。下面的例子中，我们将使用foreach()来增加一个计数器，但是其他的操作也会出现类似的问题。

#### Example（举例）

Consider the naive RDD element sum below, which may behave differently depending on whether execution is happening within the same JVM. A common example of this is when running Spark in local mode (--master = local[n]) versus deploying a Spark application to a cluster (e.g. via spark-submit to YARN):

想象一个RDD元素相加的例子，在同一JVM内不同的执行方式将导致不同的结果。最常见的一个例子，就是在本地模式下运行Spark （--master – local[n]）对比在集群中部署一个Spark应用。

Scala：

**var** counter **=** 0

**var** rdd **=** sc.parallelize(data)

*// Wrong: Don't do this!!*

rdd.foreach(x **=>** counter += x)

println("Counter value: " + counter)

Python：

counter = 0

rdd = sc.parallelize(data)

*# Wrong: Don't do this!!*

**def** increment\_counter(x):

**global** counter

counter += x

rdd.foreach(increment\_counter)

**print**("Counter value: ", counter)

#### Local vs. cluster modes（本地模式VS集群模式）

The behavior of the above code is undefined, and may not work as intended. To execute jobs, Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor. Prior to execution, Spark computes the task’s **closure**. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD (in this case foreach()). This closure is serialized and sent to each executor.

上述代码行为是未定义的，所以并不是得到预期的结果。Spark执行作业时，会将RDD操作拆分为任务（task），每个任务都通过一个执行器来执行。在执行之前，Spark会计算任务的闭包（闭包是一个能被调用的对象，它保存了创建它的作用域信息）。闭包是执行器在RDD上执行计算时必须可见的那些变量和方法（在本例中，是foreach()）。这个闭包序列化后发送到每个执行器的。

The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it’s no longer the **counter** on the driver node. There is still a **counter** in the memory of the driver node but this is no longer visible to the executors! The executors only see the copy from the serialized closure. Thus, the final value of **counter** will still be zero since all operations on **counter** were referencing the value within the serialized closure.

现在，发送到每个执行器的闭包中的变量都是副本，因此当foreach函数中引用计数器的时候，它不再是驱动节点上的计数器，此时在驱动节点的内存中仍然有一个计数器，但是它对执行器是不可见的，执行器只能够看到序列化闭包中的副本。因此，由于所有计数器的操作都引用序列化闭包中的值，计数器最终的值将仍然是0.

In local mode, in some circumstances the foreach function will actually execute within the same JVM as the driver and will reference the same original **counter**, and may actually update it.

本地模式中，在某些情况下，foreach函数在同一个JVM内部作为驱动来执行，且引用同一原始计数器，有时也会更新它。

To ensure well-defined behavior in these sorts of scenarios one should use an [Accumulator](http://spark.apache.org/docs/latest/programming-guide.html#accumulators). Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster. The Accumulators section of this guide discusses these in more detail.

为了确保这些场景中明确定义的操作，应该使用累加器。Spark中的累加器可专门用于为在集群的worker节点中划分执行时更新的变量提供一个安全机制。本指南的累加器章节会详细进行描述。

In general, closures - constructs like loops or locally defined methods, should not be used to mutate some global state. Spark does not define or guarantee the behavior of mutations to objects referenced from outside of closures. Some code that does this may work in local mode, but that’s just by accident and such code will not behave as expected in distributed mode. Use an Accumulator instead if some global aggregation is needed.

通常来说，闭包-构造函数类似于循环或者局部定义的方法，不应该用于修改某些全局的状态。Spark并未定义或者保证对从外部闭包引用的对象的行为。这样的代码应该在本地模式下进行调用，但是这种情况仅仅出于偶然并且在分布模式下这些代码不会产生预期的效果时才出现。如果需要全局聚合时，可以使用累加器。

#### Printing elements of an RDD（打印RDD的元素）

Another common idiom is attempting to print out the elements of an RDD using rdd.foreach(println) or rdd.map(println). On a single machine, this will generate the expected output and print all the RDD’s elements. However, in cluster mode, the output to stdout being called by the executors is now writing to the executor’s stdout instead, not the one on the driver, so stdout on the driver won’t show these! To print all elements on the driver, one can use the collect() method to first bring the RDD to the driver node thus: rdd.collect().foreach(println). This can cause the driver to run out of memory, though, because collect() fetches the entire RDD to a single machine; if you only need to print a few elements of the RDD, a safer approach is to use the take(): rdd.take(100).foreach(println).

另一个常见的术语就是使用rdd.foreach(println) 或者 rdd.map(println)打印RDD的元素。如果一个机器，这两句都能输出预期的结果也就是打印出RDD的所有元素。然而，在集群模式下，执行器调用的输出到标准输出的过程都将写入到执行器的标准输出，而不是驱动程序的标准输出，所以驱动程序上并不会输出。要在驱动程序上打印所有元素，我们可以使用collect()方法首先将RDD提供给驱动程序，从而使用rdd.collect().foreach(println)方法。但是这样的操作会引起驱动程序内存不足的问题，因为collect()方法会将整个RDD返回到一个机器上，如果你只需要打印RDD的几个元素，最安全的方式是使用take(): rdd.take(100).foreach(println)。

### Working with Key-Value Pairs（使用键值对）

While most Spark operations work on RDDs containing any type of objects, a few special operations are only available on RDDs of key-value pairs. The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.

大部分Spark的RDD操作可包含任意类型的对象，而一些特殊的操作只能包含键值对的RDD。最常见的是shuffle操作，比如根据键分组或者聚合元素。

In Scala, these operations are automatically available on RDDs containing [Tuple2](http://www.scala-lang.org/api/2.11.7/index.html#scala.Tuple2) objects (the built-in tuples in the language, created by simply writing (a, b)). The key-value pair operations are available in the [PairRDDFunctions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions) class, which automatically wraps around an RDD of tuples.

在Scala中，这些操作可以包含在Tuple2元素的RDD（Scala内置的元组类型，只需(a,b)就可创建此类型的对象）中使用。键值对操作可以在PairRDDFunctions类中应用，它可以自动包装一个RDD的元组。

In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2). Simply create such tuples and then call your desired operation.

在Python中，这些RDD操作包含了Python内置的元组类型，比如(1,2)。可以简单的创建这样的元素然后调用你需要的操作。

For example, the following code uses the reduceByKey operation on key-value pairs to count how many times each line of text occurs in a file:

比如，下面的代码使用键值对的reduceByKey操作来计算每个文本行在文件中出现了多少次：

Scala：

**val** lines **=** sc.textFile("data.txt")

**val** pairs **=** lines.map(s **=>** (s, 1))

**val** counts **=** pairs.reduceByKey((a, b) **=>** a + b)

Python：

lines = sc.textFile("data.txt")

pairs = lines.map(**lambda** s: (s, 1))

counts = pairs.reduceByKey(**lambda** a, b: a + b)

We could also use counts.sortByKey(), for example, to sort the pairs alphabetically, and finally counts.collect() to bring them back to the driver program as an array of objects.

我们也可以使用counts.sortByKey()，例如按照字母顺序排序然后使用counts.collect()继续将它们作为驱动程序的一个数组对象。

**Note:** when using custom objects as the key in key-value pair operations, you must be sure that a custom equals() method is accompanied with a matching hashCode() method. For full details, see the contract outlined in the [Object.hashCode() documentation](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()).

注意：在键值对操作中使用自定义对象作为key时，必须保证自定义的equals()方法有一个匹配的hashcode()方法。

### Spark 的 rdd 的 action 操作 reducebykey

顾名思义，reduceByKey就是对元素KV对的RDD中Key相同的元素的Value进行reduce，因此，Key相同的多个元素的值被reduce为一个值，然后与原RDD中的Key组成一个新的KV对。被reduce的过程调用的是reduceByKey中的方法，比如下面例子中，相同Key的Value执行minus操作：

from pyspark import SparkConf, SparkContext

from operator import add

conf = SparkConf().setMaster("local").setAppName("My App")

sc = SparkContext(conf = conf)

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

print "----output----------------------"

#pythonLines = lines.reduceByKey(add)

def minus(x, y):

return x - y

pythonLines = lines.reduceByKey(minus)

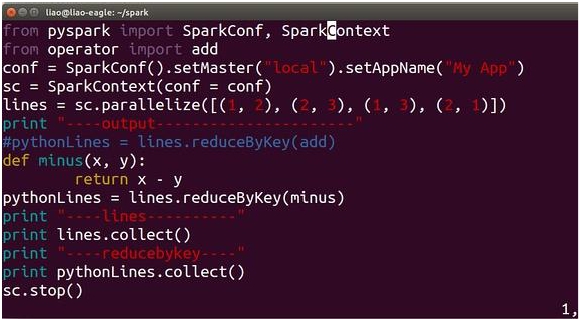
print "----lines----------"

print lines.collect()

print "----reducebykey----"

print pythonLines.collect()

sc.stop()



~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
----output----------------------  
----lines----------  
[(1, 2), (2, 3), (1, 3), (2, 1)]  
----reducebykey----  
[(1, -1), (2, 2)]

|  |
| --- |
| https://img5.doubanio.com/view/note/large/public/p27776646.jpg |
|  |

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
相同key累减  
例如key为1的有(1, 2), (1, 3)， 2 - 3 结果为-1 =-》(1, -1)  
key为2的有(2, 3), (2, 1)，value 为3 和1 ， 3- 1为2 => (2, 2)  
最后得到[(1, -1), (2, 2)]

另外一个例子：

val a = sc.parallelize(List((1,2),(1,3),(3,4),(3,6)))

a.reduceByKey((x,y) => x + y).collect

// 结果 Array((1,5), (3,10))

### Spark的RDD的action操作reduce

reduce(binary\_function)

reduce 将 RDD 中元素前两个传给输入函数，产生一个新的 return 值，新产生的 return 值与RDD 中下一个元素（第三个元素）组成两个元素，再被传给输入函数，直到最后只有一个值为止。

val c = sc.parallelize(1 to 10)

c.reduce((x, y) => x + y)//结果 55

具体过程， RDD 有 1 2 3 4 5 6 7 8 9 10 个元素:

1+2=3

3+3=6

6+4=10

10+5=15

15+6=21

21+7=28

28+8=36

36+9=45

45+10=55

### Transformations（转换）

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc ([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD), [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaRDD.html), [Python](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD), [R](http://spark.apache.org/docs/latest/api/R/index.html)) and pair RDD functions doc ([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions), [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaPairRDD.html)) for details.

下表列出了一些Spark支持的常见的转换。请参考RDD API doc （Scala，Java，Python）和pair RDD functions doc（Scala，Java）了解更多细节。

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func*returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |
| **sample**(*withReplacement*, *fraction*, *seed*) | Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Return a new dataset that contains the union of the elements in the source dataset and the argument. |
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numTasks*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.  **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.  **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks. |
| **reduceByKey**(*func*, [*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numTasks*]) | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument. |
| **join**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. |
| **cogroup**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith. |
| **cartesian**(*otherDataset*) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). |
| **pipe**(*command*, *[envVars]*) | Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. |
| **coalesce**(*numPartitions*) | Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. |
| **repartition**(*numPartitions*) | Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |
| **repartitionAndSortWithinPartitions**(*partitioner*) | Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. |
| **转换** | **含义** |
| **map(func)** | 返回一个新的分布式数据集，该数据集由func函数转换每个输入元素后形成。  scala> val a = sc.parallelize(1 to 9, 3)  scala> val b = a.map(x => x\*2)  scala> a.collect  res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)  scala> b.collect  res11: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18) |
| **filter(func)** | 返回一个新的数据集，数据集由经过 func 函数计算后返回值为 true 的输入元素组成。  scala> val filterRdd =  sc.parallelize(List(1,2,3,4,5)).map(\_\*2).filter(\_>5)  scala> filterRdd.collect  res5: Array[Int] = Array(6, 8, 10) |
| **flatMap(func)** | 与 map 类似，但是每个输入元素可以被映射为 0 或多个输出元素（因此 func 应该返回一个序列，而非单一元素）  scala> val a = sc.parallelize(1 to 4, 2)  scala> val b = a.flatMap(x => 1 to x)  scala> b.collect  res12: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4) |
| **mapPartitions(func)** | 与 map 类似，但是独立运行在 RDD 的每个分区（或块）上，因此在类型为 T 的 RDD 上运行时， func 的函数类型必须是 Iterator[T] => Iterator[U]  该函数和 map 函数类似，只不过映射函数的参数由 RDD中的每一个元素变成了 RDD 中每一个分区的迭代器。如果在映射的过程中需要频繁创建额外的对象，使用mapPartitions 要比 map 高效的多。  比如，将 RDD 中的所有数据通过 JDBC 连接写入数据库，如果使用 map 函数，可能要为每一个元素都创建一个connection，这样开销很大，如果使用 mapPartitions，那么只需要针对每一个分区建立一个 connection。  var rdd1 = sc.makeRDD(1 to 5,2)  //rdd1 有两个分区  scala> var rdd3 = rdd1.**mapPartitions**{ x => {  | var result = List[Int]()  | var i = 0  | while(x.hasNext){  | i += x.next()  | }  | result.::(i).iterator  | }}  rdd3: org.apache.spark.rdd.RDD[Int] =  MapPartitionsRDD[84] at mapPartitions at :23  //rdd3 将 rdd1 中每个分区中的数值累加  scala> rdd3.collect  res65: Array[Int] = Array(3, 12)  scala> rdd3.partitions.size  res66: Int = 2  scala> val a = sc.parallelize(1 to 9, 3)  scala> 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  }  scala> a.mapPartitions(myfunc).collect  res0: Array[(Int, Int)] = Array((2,3), (1,2),(5,6), (4,5), (8,9), (7,8)) |
| **mapPartitionsWithIndex(func)** | 类似于 mapPartitions, 但是为 func 提供了一个整型参数表示分区的索引值，因此在类型为 T 的 RDD 上运行时， func 的函数类型必须是 (Int, Iterator[T]) => Iterator[U]  var rdd1 = sc.makeRDD(1 to 5,2)  //rdd1 有两个分区  var rdd2 = rdd1.**mapPartitionsWithIndex**{  (x,iter) => {  var result = List[String]()  var i = 0  while(iter.hasNext){  i += iter.next()  }  result.::(x + "|" + i).iterator  }  }  //rdd2 将 rdd1 中每个分区的数字累加，并在每个分区的累加结果前面加了分区索引  scala> rdd2.collect  res13: Array[String] = Array(0|3, 1|12) |
| **sample(withReplacement, fraction, seed)** | 根据给定的随机数生成器种子，对数据进行采样，可以选择是否用随机数进行替换。 |
| **union(otherDataset)** | 返回一个新的数据集，新数据集由源数据集和参数数据集联合而成。  scala> var rdd = sc.parallelize(List(('a',1),('a',2)))  rdd: org.apache.spark.rdd.RDD[(Char, Int)] = ParallelColl  ectionRDD[10] at parallelize at <console>:12  scala> var rdd2 = sc.parallelize(List(('b',1),('b',2)))  rdd2: org.apache.spark.rdd.RDD[(Char, Int)] = ParallelCol  lectionRDD[11] at parallelize at <console>:12  scala> rdd union rdd2  res3: org.apache.spark.rdd.RDD[(Char, Int)] = UnionRDD[  12] at union at <console>:17  scala> res3.collect  res4: Array[(Char, Int)] = Array((a,1), (a,2), (b,1), (b,2)) |
| **intersection(otherDataset)** | 返回一个包含源数据集和参数数据集交集的新数据集该函数返回两个 RDD 的交集，并且去重。  scala> var rdd1 = sc.makeRDD(1 to 2,1)  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[45] at makeRDD at :21  scala> rdd1.collect  res42: Array[Int] = Array(1, 2)  scala> var rdd2 = sc.makeRDD(2 to 3,1)  rdd2: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[46] at makeRDD at :21  scala> rdd2.collect  res43: Array[Int] = Array(2, 3)  scala> **rdd1.intersection(rdd2).collect**  res45: Array[Int] = Array(2)  scala> **var rdd3 = rdd1.intersection(rdd2)**  rdd3: org.apache.spark.rdd.RDD[Int] =  MapPartitionsRDD[59] at intersection at :25  scala> rdd3.partitions.size  res46: Int = 1  scala> **var rdd3 = rdd1.intersection(rdd2,2)**  rdd3: org.apache.spark.rdd.RDD[Int] =  MapPartitionsRDD[65] at intersection at :25  scala> rdd3.partitions.size  res47: Int = 2 |
| **distinct([numTasks]))** | 返回一个包含源数据集中所有不重复元素的新数据集  hadoop fs -cat /tmp/lxw1234/1.txt  hello world  hello spark  hello hive  hi spark  //读取 HDFS 文件到 RDD  scala> var data =  sc.textFile("/tmp/lxw1234/1.txt")  data: org.apache.spark.rdd.RDD[String] =  MapPartitionsRDD[1] at textFile at :21  scala> data.map(\_.toUpperCase).collect  res32: **Array[String]** = Array(HELLO WORLD, HELLO SPARK, HELLO HIVE, HI SPARK)  scala> data.flatMap(\_.toUpperCase).collect  res33: **Array[Char]** = Array(H, E, L, L, O, , W,  O, R, L, D, H, E, L, L, O, , S, P, A, R, K, H,  E, L, L, O, , H, I, V, E, H, I, , S, P, A, R,  K)  scala> data.flatMap(line =>  line.split("\\s+")).collect  res61: Array[String] = Array(hello, world,  hello, spark, hello, hive, hi, spark)  scala> data.flatMap(line =>  line.split("\\s+")).distinct.collect  res62: Array[String] = Array(hive, hello,  world, spark, hi) |
| **groupByKey([numTasks])** | 在一个（ K,V ）对的数据集上调用，返回一个（ K ，Seq[V]) 对的数据集。  注意：如果你要对每个 key 做分组以实现聚合（比如求和或取平均值），使用reduceByKey 或 aggregateByKey 会有更好的性能。  注意：输出的默认并行性级别取决于父 RDD 的分区数量，你可以传入一个可选的numTasks 参数来设置不同的任务数量。  val wc = sc.textFile("/home/scipio/README.md").fl atMap(\_.split('')).map((\_,1)).groupByKey  wc.collect  res0: Array[(String, Iterable[Int])] =Array((means,ArrayBuffer(1)), (under,ArrayBuffer(1, 1)), (this,ArrayBuffer(1, 1, 1, 1)), (Because,ArrayBuffer(1)), (Python,ArrayBuffer(1, 1)), (agree,ArrayBuffer(1)), (cluster.,ArrayBuffer(1)), (its,ArrayBuffer(1)), (YARN,,ArrayBuffer(1, 1, 1)), (have,Ar… |
| **reduceByKey(func, [numTasks])** | 在一个（ K ，V) 对的数据集上调用时，返回一个（ K ， V）对的数据集，使用指定的 reduce 函数 func ，将相同 key 的值聚合到一起。类似 groupByKey ， reduce 任务个数是可以通过第二个可选参数来配置的。  scala> val a =sc.parallelize(List((1,2),(3,4),(3,6)))  scala> a.reduceByKey((x,y) => x + y).collect  res7: Array[(Int, Int)] = Array((1,2), (3,10)) |
| **aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])** | 在一个（ K ，V) 对的数据集上调用时，返回一个（ K ， V）对的数据集，使用给定的联结函数和一个中立的初始值”，将相同key 的值聚合到一起。允许聚合值类型与输入值类型不同，以避免不必要的内存分配。类似 groupByKey ，reduce 任务个数是可以通过第二个可选参数来配置的。  def seq(a:Int, b:Int) : Int ={  math.max(a,b)  }  def comb(a:Int, b:Int) : Int ={  a + b  }  val data =  sc.parallelize(List((1,3),(1,2),(1, 4),(2,3)))  data.aggregateByKey(3,4)(seq, comb).collect  输出结果是：  Array((1,10), (2,3))  参数3代表做比较的初始值，参数4代表并行化分区的数量。  参数seq代表与初始值比较的函数，参数comb是进行合并的方法。  将这个测试程序拿文字做一下描述就是：在data数据集中，按key将value进行分组合并，合并时在seq函数与指定的初始值3进行比较，保留大的值；然后在comb中来处理合并的方式。 |
| **sortByKey([ascending], [numTasks])** | 在一个 K 实现排序的（K ， V) 对的数据集上调用，返回一个按照 Key 进行排序的（ K ， V ）对数据集。升序或降序由ascending 布尔参数决定。  val rdd = sc.textFile("/home/scipio/README.md")  val wordcount = rdd.flatMap(\_.split('')).map((\_,1)).reduceByKey(\_+\_)  val wcsort = wordcount.map(x=>(x.2,x.1)).sortByKey(false).map(x=>(x.2,x.1))  wcsort.saveAsTextFile(“/home/scipio/sort/txt”)  升序的话，sortByKey(true) |
| **join(otherDataset, [numTasks])** | 在类型为（ K,V) 和（K,W) 类型的数据集上调用时，返回一个相同 key 对应的所有元素在一起的 (K, (V, W)) 数据集。支持leftOuterJoin 、rightOuterJoin, 和 fullOuterJoin 形式的外连接。  scala> val rdd1 = sc.parallelize(List(('a',1),('a',2),('b',3),('b',4)))  rdd1: org.apache.spark.rdd.RDD[(Char, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:12  scala> val rdd2 = sc.parallelize(List(('a',5),('a',6),('b',7),('b',8)))  rdd2: org.apache.spark.rdd.RDD[(Char, Int)] = ParallelCollectionRDD[11] at parallelize at <console>:12  scala> rdd1 join rdd2  res1: org.apache.spark.rdd.RDD[(Char, (Int, Int))] = FlatMappedValuesRDD[14] at join at <console>:17  res1.collect  res2: Array[(Char, (Int, Int))] = Array((b,(3,7)), (b,(3,8)), (b,(4,7)), (b,(4,8)), (a,(1,5)), (a,(1,6)), (a,(2,5)), (a,(2,6))) |
| **cogroup(otherDataset, [numTasks])** | 在类型为（ K,V) 和（K,W) 的数据集上调用，返回一个 (K,Seq[V], Seq[W]) 元组的数据集。这个操作也可以称之为groupwithcogroup 相当于 SQL 中的全外关联 full outer join，返回左右 RDD 中的记录，关联不上的为空。  参数 numPartitions 用于指定结果的分区数。  参数 partitioner 用于指定分区函数。  ## 参数为 1 个 RDD 的例子  var rdd1 =  sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  var rdd2 =  sc.makeRDD(Array(("A","a"),("C","c"),("D","d")),2)  scala> var rdd3 = **rdd1.cogroup(rdd2)**  rdd3: org.apache.spark.rdd.RDD[(String,  (Iterable[String], Iterable[String]))] =  MapPartitionsRDD[12] at cogroup at :25  scala> rdd3.partitions.size  res3: Int = 2  scala> rdd3.collect  res1: Array[(String, (Iterable[String],  Iterable[String]))] = Array(  (B,(CompactBuffer(2),CompactBuffer())),  (D,(CompactBuffer(),CompactBuffer(d))),  (A,(CompactBuffer(1),CompactBuffer(a))),  (C,(CompactBuffer(3),CompactBuffer(c)))  )  scala> var rdd4 = **rdd1.cogroup(rdd2,3)**  rdd4: org.apache.spark.rdd.RDD[(String,  (Iterable[String], Iterable[String]))] =  MapPartitionsRDD[14] at cogroup at :25  scala> rdd4.partitions.size  res5: Int = 3  scala> rdd4.collect  res6: Array[(String, (Iterable[String],  Iterable[String]))] = Array(  (B,(CompactBuffer(2),CompactBuffer())),  (C,(CompactBuffer(3),CompactBuffer(c))),  (A,(CompactBuffer(1),CompactBuffer(a))),  (D,(CompactBuffer(),CompactBuffer(d))))  ## 参数为 2 个 RDD 的例子  var rdd1 = sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  var rdd2 = sc.makeRDD(Array(("A","a"),("C","c"),("D","d")),2)  var rdd3 = sc.makeRDD(Array(("A","A"),("E","E")),2)  scala> var rdd4 = **rdd1.cogroup(rdd2,rdd3)**  rdd4: org.apache.spark.rdd.RDD[(String, (Iterable[String], Iterable[String], Iterable[String]))] =  MapPartitionsRDD[17] at cogroup at :27  scala> rdd4.partitions.size  res7: Int = 2  scala> rdd4.collect  res9: Array[(String, (Iterable[String], Iterable[String], Iterable[String]))] = Array(  (B,(CompactBuffer(2),CompactBuffer(),CompactBuffer())),  (D,(CompactBuffer(),CompactBuffer(d),CompactBuffer())),  (A,(CompactBuffer(1),CompactBuffer(a),CompactBuffer(A))),  (C,(CompactBuffer(3),CompactBuffer(c),CompactBuffer())),  (E,(CompactBuffer(),CompactBuffer(),CompactBuffer(E)))) |
| **cartesian(otherDataset)** | 笛卡尔积，在类型为 T 和 U 类型的数据集上调用时，返回一个 (T, U) 对数据集( 两两的元素对 )（就是对给定的两个 RDD 进行笛卡尔计算）  scala> val a = sc.parallelize(List(1,2,3))  a: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[62] at parallelize at <console>:12  scala> val b = sc.parallelize(List(4,5,6))  b: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[63] at parallelize at <console>:12  scala> val result = a.cartesian(b)  result: org.apache.spark.rdd.RDD[(Int, Int)] =  CartesianRDD[64] at cartesian at <console>:16  scala> result.collect  res78: Array[(Int, Int)] = Array((1,4), (1,5), (1,6), (2,4),  (2,5), (2,6), (3,4), (3,5), (3,6))  注意：笛卡尔计算会迅速消耗大量的内存，所以使用这个函数的时候要小心。 |
| **pipe(command, [envVars])** | 通过一个 shell 命令来对 RDD 的每个分区使用管道，比如一个 Perl 或批处理脚本。RDD 的元素被写入进程的标准输入，输出行写到标准输出，并返回为一个字符串 RDD。 |
| **coalesce(numPartitions)** | 减少 RDD 的分区数到 numPartitions 值，这在运行操作中非常有用，在过滤掉大数据集之后执行更高效。  在 Spark 的 Rdd 中，Rdd 是分区的。  有时候需要重新设置 Rdd 的分区数量，比如 Rdd 的分区中，Rdd 分区比较多，但是每个 Rdd 的数据量比较小，需要设置一个比较合理的分区。或者需要把 Rdd 的分区数量调大。还有就是通过设置一个 Rdd 的分区来达到设置生成的文件的数量。  有两种方法是可以重设 Rdd 的分区：分别是 coalesce()方法和 repartition()。  def coalesce(numPartitions: Int, shuffle: Boolean =  false)(implicit ord: Ordering[T] = null): RDD[T]  该函数用于将 RDD 进行重分区，使用 HashPartitioner 。  第一个参数为重分区的数目，第二个为是否进行 shuffle ，默认为 false;  scala> var data =  sc.textFile("/tmp/lxw1234/1.txt")  data: org.apache.spark.rdd.RDD[String] =  MapPartitionsRDD[53] at textFile at :21  scala> data.collect  res37: Array[String] = Array(hello world, hello spark, hello hive, hi spark)  scala> data.partitions.size  res38: Int = 2 //RDD data 默认有两个分区  scala> **var rdd1 = data.coalesce(1)**  rdd1: org.apache.spark.rdd.RDD[String] =  CoalescedRDD[2] at coalesce at :23  scala> rdd1.partitions.size  res1: Int = 1 //rdd1 的分区数为 1  scala> **var rdd1 = data.coalesce(4)**  rdd1: org.apache.spark.rdd.RDD[String] =  CoalescedRDD[3] at coalesce at :23  scala> rdd1.partitions.size  res2: Int = 2 //如果重分区的数目大于原来的分区数，那么必须指定shuffle的参数为true，否则，分区数不变  scala> **var rdd1 = data.coalesce(4,true)**  rdd1: org.apache.spark.rdd.RDD[String] =  MapPartitionsRDD[7] at coalesce at :23  scala> rdd1.partitions.size  res3: Int = 4 |
| **repartition(numPartitions)** | 对 RDD 中的数据进行重组，来创建更多或更少的分区并平衡分区中的数据。这个操作在网络中将对所有的数据重组。  该函数其实就是 coalesce 函数第二个参数为 true 的  实现。  scala> **var rdd2 = data.repartition(1)**  rdd2: org.apache.spark.rdd.RDD[String] =  MapPartitionsRDD[11] at repartition at :23  scala> rdd2.partitions.size  res4: Int = 1  scala> **var rdd2 = data.repartition(4)**  rdd2: org.apache.spark.rdd.RDD[String] =  MapPartitionsRDD[15] at repartition at :23  scala> rdd2.partitions.size  res5: Int = 4 |
| **repartitionAndSortWithinPartitions(partitioner)** | 根据给定的partitioner对RDD重新分区，在每个结果分区中通过它们的key对所有数据进行排序。它比重新分区然后在分区内部执行排序更有效，因为它可以将排序操作融入到shuffle阶段。  scala> val data =  sc.parallelize(List((1,3),(1,2),(5,4),(1,4) ),(2,3),(2,4)),3)  data: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[3] at parallelize at <console>:21  scala>data.repartitionAndSortWithinPartitions(new HashPartitioner(3)).collect  res3: Array[(Int, Int)] = Array((1,4),(1,3),(1,2),(2,3),(2,4),(5,4)) |

### Actions（动作）

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc ([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD), [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaRDD.html), [Python](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD), [R](http://spark.apache.org/docs/latest/api/R/index.html))

and pair RDD functions doc ([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions), [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaPairRDD.html)) for details.

下表列出了一些常见的action操作。请参考RDD API doc （Scala，Java，Python）和pair RDD functions doc（Scala，Java）了解更多的细节。

|  |  |
| --- | --- |
| **Action** | **Meaning** |
| **reduce**(*func*) | Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| **count**() | Return the number of elements in the dataset. |
| **first**() | Return the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Return an array with the first *n* elements of the dataset. |
| **takeSample**(*withReplacement*, *num*, [*seed*]) | Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |
| **takeOrdered**(*n*, *[ordering]*) | Return the first *n* elements of the RDD using either their natural order or a custom comparator. |
| **saveAsTextFile**(*path*) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*)  (Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| **saveAsObjectFile**(*path*)  (Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile(). |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| **foreach**(*func*) | Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an [Accumulator](http://spark.apache.org/docs/latest/programming-guide.html#accumulators) or interacting with external storage systems.  **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See [Understanding closures](http://spark.apache.org/docs/latest/programming-guide.html#understanding-closures-a-nameclosureslinka)for more details. |
| **动作** | **含义** |
| **reduce(func)** | 通过函数 func（接受两个参数，返回一个参数）聚集数据集中的所有元素。这个功能必须可交换且可关联的，从而可以正确的被并行执行。  scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[36] at makeRDD at :21  scala> rdd1.reduce(\_ + \_)  res18: Int = 55  scala> var rdd2 =  sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("C"  ,1)  rdd2: org.apache.spark.rdd.RDD[(String, Int)] =  ParallelCollectionRDD[38] at makeRDD at :21  scala> rdd2.reduce((x,y) => {  | (x.\_1 + y.\_1,x.\_2 + y.\_2)  | })  res21: (String, Int) = (CBBAA,6) |
| **collect()** | 在驱动程序中，以数组的形式返回数据集的所有元素。这通常会在使用 filter 或者其它操作并返回一个足够小的数据子集后再使用会比较有用。  scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[36] at makeRDD at :21  scala> rdd1.collect  res23: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9,  10) |
| **count()** | 返回数据集的元素的个数。  scala> var rdd1 =  sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  rdd1: org.apache.spark.rdd.RDD[(String, String)] =  ParallelCollectionRDD[34] at makeRDD at :21  scala> rdd1.count  res15: Long = 3 |
| **first()** | 返回数据集的第一个元素（类似于 take （ 1 ））（不排序）  scala> var rdd1 =  sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  rdd1: org.apache.spark.rdd.RDD[(String, String)] =  ParallelCollectionRDD[33] at makeRDD at :21  scala> rdd1.first  res14: (String, String) = (A,1)  scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[0] at makeRDD at :21  scala> rdd1.first  res8: Int = 10 |
| **take(n)** | 返回一个由数据集的前 n 个元素组成的数组。注意，这个操作目前并非并行执行，而是由驱动程序计算所有的元素。take 用于获取 RDD 中从 0 到 num-1 下标的元素，不排序。  scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[40] at makeRDD at :21  scala> rdd1.take(1)  res0: Array[Int] = Array(10)  scala> rdd1.take(2)  res1: Array[Int] = Array(10, 4)  def top(num: Int)(implicit ord: Ordering[T]): Array[T]  top 函数用于从 RDD 中，按照默认（降序）或者指定的排序规则，返回前 num 个元素。  scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[40] at makeRDD at :21  scala> rdd1.top(1)  res2: Array[Int] = Array(12)  scala> rdd1.top(2)  res3: Array[Int] = Array(12, 10)  //指定排序规则  scala> implicit val myOrd = implicitly[Ordering[Int]].reverse  myOrd: scala.math.Ordering[Int] =[scala.math.Ordering$$anon$4@767499ef](mailto:scala.math.Ordering$$anon$4@767499ef)  scala> rdd1.top(1)  res4: Array[Int] = Array(2)  scala> rdd1.top(2)  res5: Array[Int] = Array(2, 3) |
| **takeSample(withReplacement, num, [seed])** | 返回一个数组，在数据集中随机采样 num 个元素组成，可以选择是否用随机数替换不足的部分，Seed 用于指定的随机数生成器种子 |
| **takeOrdered(n, [ordering])** | 使用 RDD 的自然排序方法或者自定义比较方法返回前 n 个元素。  与 top 类似，只不过以和 top 相反的顺序返回元素。  scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[40] at makeRDD at :21  scala> rdd1.top(1)  res4: Array[Int] = Array(2)  scala> rdd1.top(2)  res5: Array[Int] = Array(2, 3)  scala> rdd1.takeOrdered(1)  res6: Array[Int] = Array(12)  scala> rdd1.takeOrdered(2)  res7: Array[Int] = Array(12, 10) |
| **saveAsTextFile(path)** | 将数据集的元素，以 textfile 的形式，保存到本地文件系统、HDFS 或者任何其它 hadoop 支持的文件系统的指定文件夹中。对于每个元素，Spark 将会调用 toString 方法，将它转换为文件中的文本行。  var rdd1 = sc.makeRDD(1 to 10,2)  scala>  rdd1.saveAsTextFile("hdfs://cdh5/tmp/lxw1234.com/")  //保存到 HDFS  hadoop fs -ls /tmp/lxw1234.com  Found 2 items  -rw-r--r-- 2 lxw1234 supergroup 0 2015-07-10 09:15  /tmp/lxw1234.com/\_SUCCESS  -rw-r--r-- 2 lxw1234 supergroup 21 2015-07-10 09:15  /tmp/lxw1234.com/part-00000  hadoop fs -cat /tmp/lxw1234.com/part-00000  1  2  3  4  5  6  7  8  9  10  注意：如果使用rdd1.saveAsTextFile(“file:///tmp/lxw1234.com”) 将文件保存到本地文件系统，那么只会保存在 Executor 所在机器的本地目录。  // 指定压缩格式保存：  rdd1.saveAsTextFile("hdfs://cdh5/tmp/lxw1234.com/",cl  assOf[com.hadoop.compression.lzo.LzopCodec])  hadoop fs -ls /tmp/lxw1234.com  -rw-r--r-- 2 lxw1234 supergroup 0 2015-07-10 09:20  /tmp/lxw1234.com/\_SUCCESS  -rw-r--r-- 2 lxw1234 supergroup 71 2015-07-10 09:20  /tmp/lxw1234.com/part-00000.lzo  hadoop fs -text /tmp/lxw1234.com/part-00000.lzo  1  2  3  4  5  6  7  8  9  10 |
| **saveAsSequenceFile(path)  (Java and Scala)** | 将数据集的元素，以 Hadoop sequencefile 的格式，保存到本地文件系统、HDFS 或者任何其它 hadoop 支持的文件系统的指定文件夹中。这个只限于由 key-value 对组成，并实现了 Hadoop 的Writable 接口，或者 Scala 中隐式的可以转换为 Writable 的 RDD。（ Spark 包括了基本类型的转换，例如 Int ，Double ， String，等等） |
| **saveAsObjectFile(path)  (Java and Scala)** | 使用 Java 序列化将数据集中的元素存储为一个可以使用SparkContext.objectFile() 加载的文件格式。  var rdd1 = sc.makeRDD(1 to 10,2)  rdd1.saveAsObjectFile("hdfs://cdh5/tmp/lxw1234.com/")  hadoop fs -cat /tmp/lxw1234.com/part-00000  SEQ !org.apache.hadoop.io.NullWritable"org.apache.had  oop.io.BytesWritableT |
| **countByKey()** | 对 (K,V) 类型的 RDD 有效，返回一个 (K ， Int) 对的哈希 Map，表示每个 key 对应的元素个数。  scala> var rdd1 =  sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("B"  ,3)))  rdd1: org.apache.spark.rdd.RDD[(String, Int)] =  ParallelCollectionRDD[7] at makeRDD at :21  scala> rdd1.countByKey  res5: scala.collection.Map[String,Long] = Map(A -> 2,  B -> 3) |
| **foreach(func)** | 对数据集的每一个元素运行函数 func 。这通常针对更新一个累加器，或者和外部存储系统进行交互等进行操作。  注意：在 foreach 外除了累加器，修改变量可能导致未定义的错误。  Foreach 一般用于遍历 RDD，将函数 func 应用于每个元素。  scala> var cnt = sc.accumulator(0)  cnt: org.apache.spark.Accumulator[Int] = 0  scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] =  ParallelCollectionRDD[5] at makeRDD at :21  scala> rdd1.foreach(x => cnt += x)  scala> cnt.value  res51: Int = 55  scala> rdd1.collect.foreach(println)  1  2  3  4  5  6  7  8  9  10 |

其他函数：

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

sortBy 根据给定的排序 k 函数将 RDD 中的元素进行排序。

scala> var rdd1 = sc.makeRDD(Seq(3,6,7,1,2,0),2)

scala> rdd1.sortBy(x => x).collect

res1: Array[Int] = Array(0, 1, 2, 3, 6, 7) //默认升序

scala> rdd1.sortBy(x => x,false).collect

res2: Array[Int] = Array(7, 6, 3, 2, 1, 0) //降序

//RDD[K,V]类型

scala>var rdd1 = sc.makeRDD(Array(("A",2),("A",1),("B",6),("B",3),("B",7)))

scala> rdd1.sortBy(x => x).collect

res3: Array[(String, Int)] = Array((A,1), (A,2), (B,3), (B,6), (B,7))

//按照V进行降序排序

scala> rdd1.sortBy(x => x.\_2,false).collect

res4: Array[(String, Int)] = Array((B,7), (B,6), (B,3), (A,2), (A,1))

The Spark RDD API also exposes asynchronous versions of some actions, like foreachAsync for foreach, which immediately return a FutureAction to the caller instead of blocking on completion of the action. This can be used to manage or wait for the asynchronous execution of the action.

### Shuffle operations（shuffle操作）

Certain operations within Spark trigger an event known as the shuffle. The shuffle is Spark’s mechanism for re-distributing data so that it’s grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

Spark内部特定的操作会触发shuffle事件。Shuffle是Spark的一种机制，用来将数据重新分配，使它的不同分组跨分区。这通常涉及到跨executor和机器复制数据，使得shuffle操作成为一个复杂并且消耗资源的操作。

#### Background（背景）

To understand what happens during the shuffle we can consider the example of the [reduceByKey](http://spark.apache.org/docs/latest/programming-guide.html#ReduceByLink) operation. The reduceByKey operation generates a new RDD where all values for a single key are combined into a tuple - the key and the result of executing a reduce function against all values associated with that key. The challenge is that not all values for a single key necessarily reside on the same partition, or even the same machine, but they must be co-located to compute the result.

为了了解 shuffle 过程中发生了什么，我们用 reduceByKey 操作来说明。reduceByKey 操作产生了一个新的 RDD，原 RDD 中所有对应同一个键的值组成了一个元祖——也就是键和该键对应的所有元素上执行了一个 reduce 函数的结果。该操作的挑战是同一个键所对应的所有元素不一定在同一个分区，甚至不在同一个机器上，但是它们必须被统一定位以计算结果。

In Spark, data is generally not distributed across partitions to be in the necessary place for a specific operation. During computations, a single task will operate on a single partition - thus, to organize all the data for a single reduceByKey reduce task to execute, Spark needs to perform an all-to-all operation. It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key - this is called the **shuffle**.

在 Spark 里，数据通常并不是跨分区分布，而是分布在便于进行特定操作的地方。在计算的过程

中，单一任务将会在单一的分区上进行操作——这样，为了给 reduceByKey 的一个单一 reduce

任务的执行组织所有的数据， Spark 需要执行一个所有对所有的操作，它必须从所有的分区中读取所有键值对应的所有值，然后跨分区汇集值来计算每个键对应的最终结果——这就是 shuffle 。

Although the set of elements in each partition of newly shuffled data will be deterministic, and so is the ordering of partitions themselves, the ordering of these elements is not. If one desires predictably ordered data following shuffle then it’s possible to use:

尽管 shuffle 操作后每个分区上的元素集合和分区的顺序是确定的，但是分区中元素的顺序是不确

定的。如果你希望预测 shuffle 操作后数据的顺序，可以使用如下操作：

* mapPartitions to sort each partition using, for example, .sorted
* mapPartitions用来对使用的每个分区排序，比如.sorted.
* repartitionAndSortWithinPartitions to efficiently sort partitions while simultaneously repartitioning
* repartitionAndSortWithinPartitions 高效的对分区排序并重新分区
* sortBy to make a globally ordered RDD
* sortBy来创建一个全局有序的ＲＤＤ

Operations which can cause a shuffle include **repartition** operations like [repartition](http://spark.apache.org/docs/latest/programming-guide.html#RepartitionLink) and [coalesce](http://spark.apache.org/docs/latest/programming-guide.html#CoalesceLink), **‘ByKey** operations (except for counting) like [groupByKey](http://spark.apache.org/docs/latest/programming-guide.html#GroupByLink) and [reduceByKey](http://spark.apache.org/docs/latest/programming-guide.html#ReduceByLink), and **join** operations like [cogroup](http://spark.apache.org/docs/latest/programming-guide.html#CogroupLink) and [join](http://spark.apache.org/docs/latest/programming-guide.html#JoinLink).

会引起 shuffle 的操作包括“重分区”操作，如 repartition 和 coalesce ；“ ByKey “操作（除了counting ），比如 groupByKey 和 reduceByKey ；以及“join “操作比如 cogroup 和join 。

#### Performance Impact（性能影响）

The **Shuffle** is an expensive operation since it involves disk I/O, data serialization, and network I/O. To organize data for the shuffle, Spark generates sets of tasks - map tasks to organize the data, and a set of reduce tasks to aggregate it. This nomenclature comes from MapReduce and does not directly relate to Spark’s map and reduce operations.

Shuffle 操作属于昂贵的操作，因为它涉及磁盘 I/O 、数据序列化和网络 I/O 。为是组织 Shuffle 的数据，Spark 生成一个 map 任务集合来组织数据，生成一个 reduce 任务集合来聚合数据。这个概念来自于 MapReduce ，与 Spark 自身的 map 和 reduce 操作并没有直接关系。

Internally, results from individual map tasks are kept in memory until they can’t fit. Then, these are sorted based on the target partition and written to a single file. On the reduce side, tasks read the relevant sorted blocks.

在内部，每个单独的 map 任务将结果存储在内存中，直到无法分配内存。然后，这些结果基于目

标分区排序并存储在单一的文件中。在 reduce 端，reduce 任务读取已排序的相关块。

Certain shuffle operations can consume significant amounts of heap memory since they employ in-memory data structures to organize records before or after transferring them. Specifically, reduceByKey and aggregateByKey create these structures on the map side, and 'ByKey operations generate these on the reduce side. When data does not fit in memory Spark will spill these tables to disk, incurring the additional overhead of disk I/O and increased garbage collection.

某些 shuffle 操作会消耗大量的堆内存，因为它们使用基于内存的数据结构来在传递数据之前或之

后组织它们。特别地， reduceByKey 和 aggregateByKey 在 map 端创建这些结构然后“ByKey ”操作在 reduce 端生成这些结构。当没有足够的内存时， Spark 将这些数据写到磁盘

上， 引起额外的磁盘 I/O 开销和垃圾回收处理。

Shuffle also generates a large number of intermediate files on disk. As of Spark 1.3, these files are preserved until the corresponding RDDs are no longer used and are garbage collected. This is done so the shuffle files don’t need to be re-created if the lineage is re-computed. Garbage collection may happen only after a long period of time, if the application retains references to these RDDs or if GC does not kick in frequently. This means that long-running Spark jobs may consume a large amount of disk space. The temporary storage directory is specified by the spark.local.dirconfiguration parameter when configuring the Spark context.

shuffle 也会产生大量的中间文件。在 Spark1.3 中，这些文件在相应的 RDD 不再使用并且被垃圾回收之前不会清除，这样做是为了使 shuffle 操作如果需要使用上次的结果时不用重新计算。垃圾回收可能会很久之后才会发生，这意味着长时间运行的 Spark 会消耗可用的磁盘空间。临时文件目录在配置 sparkContext 时 spark.local.dir 配置指定。

Shuffle behavior can be tuned by adjusting a variety of configuration parameters. See the ‘Shuffle Behavior’ section within the [Spark Configuration Guide](http://spark.apache.org/docs/latest/configuration.html).

shuffle 操作可以被一些配置参数调整，这些将在 spark 的配置指导中详细说明。

## RDD Persistence（RDD持久化）

One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

Spark 的重要功能之一是通过操作将数据持久化（或缓存）到内存中。当你持久化一个 RDD，每

个节点都将它的计算分区结果保存在内存中，并在对此数据集（或衍生出得数据集）进行的其他动

作中重用。这样可以使后续的动作变得更加迅速（通常可以提高 10 倍）。缓存是构建迭代算法的

关键，而且对更快的交互体验至关重要。

You can mark an RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark’s cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

你可以用 persist() 或 cache() 方法来标记一个要被持久化的 RDD，该 RDD 首次在一个动作

（Action ）中被计算时，它将会被保留在计算节点的内存中并重用。 Spark 的 Cache 有容错机制——如果 RDD 的任一分区丢失了，通过使用原先创建它的转换操作，它将会被自动重算（不需要全部重算，只计算丢失的部分）。

In addition, each persisted RDD can be stored using a different storage level, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes. These levels are set by passing a StorageLevel object ([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.storage.StorageLevel),[Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/storage/StorageLevel.html), [Python](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.StorageLevel)) to persist(). The cache() method is a shorthand for using the default storage level, which is StorageLevel.MEMORY\_ONLY (store deserialized objects in memory). The full set of storage levels is:

此外，每一个持久化的 RDD 都可以用不同的保存级别进行保存，从而允许你持久化数据集在硬

盘，或者作为序列化的 Java 对象持久化在内存中（节省空间），甚至于跨结点复制。这些级别是

通过将一个 org.apache.spark.storage.StorageLevel 对象传递给 persist() 方法进行设置。

cache() 方法是使用默认存储级别的快捷方法，也就是 StorageLevel.MEMORY\_ONLY( 将反序列化的对象存入内存）。完整的可选存储级别如下：

|  |  |
| --- | --- |
| **Storage Level** | **Meaning** |
| MEMORY\_ONLY | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.  将 RDD 作为反序列化的对象储存在 JVM 中。如果 RDD 不能存储到内存中，一些分区将不会被缓存，并且在需要的时候被重新计算。这个是默认的级别。 |
| MEMORY\_AND\_DISK | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.  将 RDD 作为反序列化的的对象存储在 JVM 中。如果 RDD 不能被与内存装下，超出的分区将被保存在硬盘上，并且在需要时被读取。 |
| MEMORY\_ONLY\_SER  (Java and Scala) | Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a [fast serializer](http://spark.apache.org/docs/latest/tuning.html), but more CPU-intensive to read.  将 RDD 作为序列化的的对象进行存储（每一分区占用一个字节数组）。通常来说，这比将对象反序列化的空间利用率更高，尤其当使用 [fast](http://spark.incubator.apache.org/docs/latest/tuning.html) [serializer,](http://spark.incubator.apache.org/docs/latest/tuning.html)但在读取时会比较占用 CPU |
| MEMORY\_AND\_DISK\_SER  (Java and Scala) | Similar to MEMORY\_ONLY\_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.  与 MEMORY\_ONLY\_SER 相似，但是把超出内存的分区将存储在硬盘上而不是在每次需要的时候重新计算。 |
| DISK\_ONLY | Store the RDD partitions only on disk.  只将 RDD 分区存储在硬盘上。 |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, etc. | Same as the levels above, but replicate each partition on two cluster nodes.  与上述的存储级别一样，但是将每一个分区都复制到两个集群结点上 |
| OFF\_HEAP (experimental) | Similar to MEMORY\_ONLY\_SER, but store the data in [off-heap memory](http://spark.apache.org/docs/latest/configuration.html#memory-management). This requires off-heap memory to be enabled.  与 MEMORY\_ONLY\_SER 类似，但是将数据存储在外堆内存中，这需要外堆内存启用。 |

**Note:** In Python, stored objects will always be serialized with the[*Pickle*](https://docs.python.org/2/library/pickle.html)library, so it does not matter whether you choose a serialized level. The available storage levels in Python include*MEMORY\_ONLY*,*MEMORY\_ONLY\_2*,*MEMORY\_AND\_DISK*,*MEMORY\_AND\_DISK\_2*,*DISK\_ONLY*, and*DISK\_ONLY\_2*.

**注意：**在 python 里，总是使用 Pickle 库对存储对象进行序列化，所以它不关心你选择何种serialized 级别。

Spark also automatically persists some intermediate data in shuffle operations (e.g. reduceByKey), even without users calling persist. This is done to avoid recomputing the entire input if a node fails during the shuffle. We still recommend users call persist on the resulting RDD if they plan to reuse it.

Spark 也会在 shuffle 操作 ( 比如 reduceByKey) 中存储一些中间对象，甚至用户在不需要调用

persist 的情况下。这样做的目的是避免 shuffle 过程中在一个节点失败后重新计算整个输入。我们建议用户持久化它们的 RDD， 如果需要重用的话。

### Which Storage Level to Choose?（存储级别的选择）

Spark’s storage levels are meant to provide different trade-offs between memory usage and CPU efficiency. We recommend going through the following process to select one:

Spark 的不同存储级别，旨在满足内存使用和 CPU 效率权衡上的不同需求。我们建议通过以下的

步骤来进行选择：

* If your RDDs fit comfortably with the default storage level (MEMORY\_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
* 如果你的RDDs可以很好地与默认的存储级别（MEMORY\_ONLY）契合，就不需要做任何修改了。这已经是CPU使用效率最高的选项，它使得RDDs的操作尽可能地快。
* If not, try using MEMORY\_ONLY\_SER and [selecting a fast serialization library](http://spark.apache.org/docs/latest/tuning.html) to make the objects much more space-efficient, but still reasonably fast to access. (Java and Scala)
* 如果不行，试着使用 MEMORY\_ONLY\_SER 并且选择一个快速序列化的库使得对象在有比较高的空间使用率的情况下，依然可以较快被访问。
* Don’t spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.
* 尽可能不要存储到硬盘上，除非计算数据集的函数，计算量特别大，或者它们过滤了大量的数据。否则，重新计算一个分区的速度，和与从硬盘中读取基本差不多快。
* Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). All the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.
* 如果你想有快速故障恢复能力，使用复制存储级别（例如：用Spark来响应web应用的请求）。所有的存储级别都有通过重新计算丢失数据恢复错误的容错机制，但是复制存储级别可以让你在RDD上持续的运行任务，而不需要等待丢失的分区被重新计算。

### Removing Data（移除数据）

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. If you would like to manually remove an RDD instead of waiting for it to fall out of the cache, use the RDD.unpersist() method.

Spark 自动地监控每个节点上的缓存使用率。并通过 LRU 算法删除过时的数据。如果你想手动删除一个 RDD 而不是等待它从缓存中过时，使用 RDD.unpersist 方法。

# Shared Variables（共享变量）

Normally, when a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all the variables used in the function. These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program. Supporting general, read-write shared variables across tasks would be inefficient. However, Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators.

一般来说，当一个函数被传递给 Spark 操作（比如 map 或 reduce ）并在一个远程集群节点上运行时，它实际上操作的是这个函数中用到的所有变量的独立副本。这些变量被复制到每一个机器上，远程机器上对变量的所有更新都不会回传到驱动程序。在任务之间使用通用的，支持读写的共享变量是低效的。尽管如此， Spark 还是为两种常见的使用模式提供了两种有限类型的共享变量，广播变量和累加器。

## Broadcast Variables（广播变量）

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

广播变量允许程序员将一个只读的变量缓存在每台机器上，而不用在任务之间传递变量。他们可以这样被使用，例如，以一种高效的方式给每个节点一个大的输入数据集。 Spark 还尝试使用高效地广播算法来分发变量，进而减少通信的开销。

Spark actions are executed through a set of stages, separated by distributed “shuffle” operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Spark 的动作通过一系列的阶段执行，这些阶段由分布式的 shuffle 操作区分。Spark 在每个阶段内自动地广播每个任务需要的通用数据。这些广播数据被序列化地缓存，在运行任务之前被反序列化出来。这意味着当我们需要在多个阶段的任务之间使用相同的数据，或者以反序列化形式缓存数据非常重要的时候，显式地创建广播变量才有用。

Broadcast variables are created from a variable v by calling SparkContext.broadcast(v). The broadcast variable is a wrapper around v, and its value can be accessed by calling the value method. The code below shows this:

广播变量是通过在一个变量 v 上调用 SparkContext.broadcast(v) 创建的。广播变量是围绕着 v 的封装，可以通过 value 方法访问这个变量。举例如下：

Scala：

scala> **val** broadcastVar **=** sc.broadcast(**Array**(1, 2, 3))

broadcastVar**:** org.apache.spark.broadcast.Broadcast[Array[Int]] **=** **Broadcast**(0)

scala> broadcastVar.value

res0**:** Array[Int] **=** **Array**(1, 2, 3)

Python：

>>> broadcastVar = sc.broadcast([1, 2, 3])

<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value

[1, 2, 3]

After the broadcast variable is created, it should be used instead of the value v in any functions run on the cluster so that v is not shipped to the nodes more than once. In addition, the object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

在广播变量被创建后，它应该在集群运行的任何函数中，代替 v 值被调用，从而 v 值不需要被再次传递到这些结点上。另外，为了确保所有的节点获得相同的变量，对象 v 在被广播之后就不应该再修改。

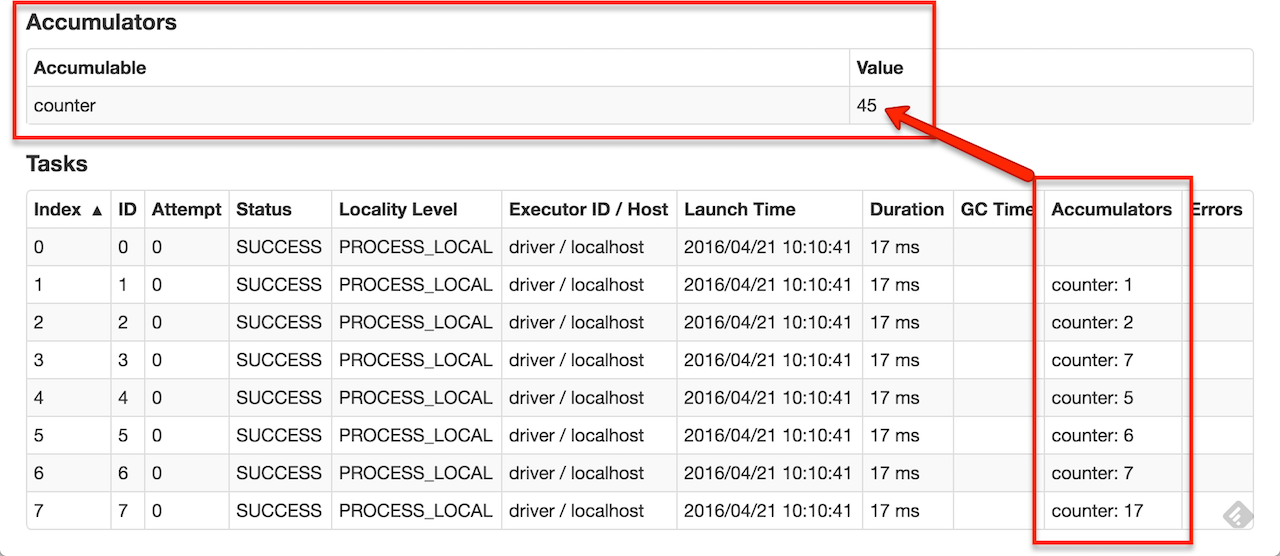
## Accumulators（累加器）

Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

累加器是一种只能通过关联和交换操作进行“加”的变量，因此可以高效被并行支持。它可以被用来实现计数器和求和。Spark 原生地只支持数字类型的累加器，开发者可以自己添加新的支持类型。

If accumulators are created with a name, they will be displayed in Spark’s UI. This can be useful for understanding the progress of running stages (NOTE: this is not yet supported in Python).

如果累加器是通过命名来创建的，那么它将在 Spark 用户界面中会被显示出来。这对了解运行阶段的过程非常有用。（注意：这个在 Python 中暂未支持）



For Scala: A numeric accumulator can be created by calling SparkContext.longAccumulator() or SparkContext.doubleAccumulator() to accumulate values of type Long or Double, respectively. Tasks running on a cluster can then add to it using the add method. However, they cannot read its value. Only the driver program can read the accumulator’s value, using its value method.

Scala ：一个累加器可以通过调用 SparkContext.longAccumulator() 或者SparkContext.doubleAccumulator() 方法来累加 Long 或者 Double 类型的值。运行在集群上的任务，可以通过使用 add 方法来给它加值。然而，他们不能读取这个值。只有驱动程序可以使用value 的方法来读取累加器的值。

For Python: An accumulator is created from an initial value v by calling SparkContext.accumulator(v). Tasks running on a cluster can then add to it using the add method or the += operator. However, they cannot read its value. Only the driver program can read the accumulator’s value, using its valuemethod.

Python：一个累加器可以通过调用 SparkContext.accumulator(v) 方法从一个初始值 v 中创建。运行在集群上的任务，可以通过 add 或者"+="方法在累加器上进行累加操作。然而，它们不能读取这个值。只有驱动程序可以使用 value 的方法来读取累加器的值。

The code below shows an accumulator being used to add up the elements of an array:

下面的代码展示了如何把一个数组中的所有元素累加到累加器上 :

Scala：

scala> **val** accum **=** sc.longAccumulator("My Accumulator")

accum**:** org.apache.spark.util.LongAccumulator = **LongAccumulator**(id**:** 0, name**:** Some(My Accumulator), value**:** 0)

scala> sc.parallelize(**Array**(1, 2, 3, 4)).foreach(x **=>** accum.add(x))

...

10/09/29 18**:**41:08 INFO SparkContext: Tasks finished in 0.317106 s

scala> accum.value

res2**:** Long = 10

Python：

>>> accum = sc.accumulator(0)

Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(**lambda** x: accum.add(x))

...

10/09/29 18:41:08 INFO SparkContext: Tasks finished **in** 0.317106 s

scala> accum.value

10

Scala: While this code used the built-in support for accumulators of type Long, programmers can also create their own types by subclassing [AccumulatorV2](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.util.AccumulatorV2). The AccumulatorV2 abstract class has several methods which need to override: reset for resetting the accumulator to zero, and add for add anothor value into the accumulator, merge for merging another same-type accumulator into this one. Other methods need to override can refer to scala API document. For example, supposing we had a MyVector class representing mathematical vectors, we could write:

Scala ：尽管上面的例子使用了内置支持的累加器类型 Long ，但是开发人员也可以通过继承AccumulatorV2 类来创建它们自己的累加器类型。 AccumulatorV2 抽象类有几个方法需要复写：reset 方法将累加器重置于 0 ，add 方法将另一个值加到累加器中， merge 方法将另一个同类型的累加器与当前累加器合并。其他需要复写的方法可以参见 Scala API 文档。假设我们有一个代表数学 vector 的 MyVector 类。我们可以像下面这样实现：

注意：当开发人员自定义 AccumulatorV2 类型的时候，结果集类型可以与添加的元素的类型不

同。

Python: While this code used the built-in support for accumulators of type Int, programmers can also create their own types by subclassing [AccumulatorParam](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.AccumulatorParam). The AccumulatorParam interface has two methods: zero for providing a “zero value” for your data type, and addInPlace for adding two values together. For example, supposing we had a Vector class representing mathematical vectors, we could write:

Python ：尽管上面的例子使用了内置支持的累加器类型 Int，但是开发人员也可以通过继承AccumulatorParam 类来创建它们自己的累加器类型。 AccumulatorParam 接口有两个方法：zero 方法为你的类型提供一个 0 值。

addInPlace 方法将两个值相加。

假设我们有一个代表数学 vector 的 Vector 类。我们可以像下面这样实现：

Scala：

**object** **VectorAccumulatorV2** **extends** **AccumulatorV2**[MyVector, MyVector] {

**val** vec\_ **:** MyVector = **MyVector**.createZeroVector

**def** reset()**:** MyVector = {

vec\_.reset()

}

**def** add(v1**:** MyVector, v2**:** MyVector)**:** MyVector = {

vec\_.add(v2)

}

...

}

*// Then, create an Accumulator of this type:*

**val** myVectorAcc **=** **new** **VectorAccumulatorV2**

*// Then, register it into spark context:*

sc.register(myVectorAcc, "MyVectorAcc1")

Python:

**class** **VectorAccumulatorParam**(AccumulatorParam):

**def** zero(self, initialValue):

**return** Vector.zeros(initialValue.size)

**def** addInPlace(self, v1, v2):

v1 += v2

**return** v1

*# Then, create an Accumulator of this type:*

vecAccum = sc.accumulator(Vector(...), VectorAccumulatorParam())

Note that, when programmers define their own type of AccumulatorV2, the resulting type can be different than that of the elements added.

For accumulator updates performed inside **actions only**, Spark guarantees that each task’s update to the accumulator will only be applied once, i.e. restarted tasks will not update the value. In transformations, users should be aware of that each task’s update may be applied more than once if tasks or job stages are re-executed.

对累加器来说，更新操作仅仅发生在动作内部。Spark 确保每个任务对累加器的更新只能应用一

次，也就是说，重启的任务将不会对值进行更新。在转换动作中，用户应该注意到，如果任务或者

作业的步骤被重新执行，每个任务的更新操作可能不止应用一次。

Accumulators do not change the lazy evaluation model of Spark. If they are being updated within an operation on an RDD, their value is only updated once that RDD is computed as part of an action. Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map(). The below code fragment demonstrates this property:

累加器并没有改变 Spark 的惰性求值模型。如果它们被 RDD 上的操作更新，它们的值只有当

RDD 因为动作操作被计算时才被更新。因此，当执行一个惰性的转换操作，比如 map 时，不能保证对累加器值的更新被实际执行了。下面的代码片段演示了此特性：

Scala：

**val** accum **=** sc.longAccumulator

data.map { x **=>** accum.add(x); x }

*// Here, accum is still 0 because no actions have caused the map operation to be computed.*

//在这里,accum 的值仍然是 0，因为没有动作操作引起 map 被实际的计算.

Python:

accum = sc.accumulator(0)

**def** g(x):

accum.add(x)

**return** f(x)

data.map(g)

*# Here, accum is still 0 because no actions have caused the `map` to be computed.*

# Deploying to a Cluster（发布到集群）

The [application submission guide](http://spark.apache.org/docs/latest/submitting-applications.html) describes how to submit applications to a cluster. In short, once you package your application into a JAR (for Java/Scala) or a set of .py or .zip files (for Python), the bin/spark-submit script lets you submit it to any supported cluster manager.

[应用提交指南](https://spark.apache.org/docs/latest/submitting-applications.html) 描述了如何提交应用到一个 Spark 集群中。简而言之，一旦你打包好你的应用（JAR for Java/Scala ，或者一堆 .py / .zip files for Python), bin/spark-submit 脚本允许你提交

它们到任意的 cluster manager 。

# Launching Spark jobs from Java / Scala

The [org.apache.spark.launcher](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/launcher/package-summary.html) package provides classes for launching Spark jobs as child processes using a simple Java API.

# Unit Testing

Spark is friendly to unit testing with any popular unit test framework. Simply create a SparkContext in your test with the master URL set to local, run your operations, and then call SparkContext.stop() to tear it down. Make sure you stop the context within a finally block or the test framework’s tearDown method, as Spark does not support two contexts running concurrently in the same program.

Spark 很好的支持流行的单元测试框架。 通过将 master URL 设置为 local ，可以简单的在你的测试中创建一个 SparkContext ，它可以执行你的操作然后调用 SparkContext.stop() 停止。确保在finally 块中停止 context 或者在单元测试框架的 tearDown 停止 context ，因为 Spark 不支持在同一个程序的同时拥有两个 context 。

# Where to Go from Here

You can see some [example Spark programs](http://spark.apache.org/examples.html) on the Spark website. In addition, Spark includes several samples in the examples directory ([Scala](https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples),[Java](https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples), [Python](https://github.com/apache/spark/tree/master/examples/src/main/python), [R](https://github.com/apache/spark/tree/master/examples/src/main/r)). You can run Java and Scala examples by passing the class name to Spark’s bin/run-example script; for instance:

./bin/run-example SparkPi

For Python examples, use spark-submit instead:

./bin/spark-submit examples/src/main/python/pi.py

For R examples, use spark-submit instead:

./bin/spark-submit examples/src/main/r/dataframe.R

For help on optimizing your programs, the [configuration](http://spark.apache.org/docs/latest/configuration.html) and [tuning](http://spark.apache.org/docs/latest/tuning.html) guides provide information on best practices. They are especially important for making sure that your data is stored in memory in an efficient format. For help on deploying, the [cluster mode overview](http://spark.apache.org/docs/latest/cluster-overview.html) describes the components involved in distributed operation and supported cluster managers.

Finally, full API documentation is available in [Scala](http://spark.apache.org/docs/latest/api/scala/#org.apache.spark.package), [Java](http://spark.apache.org/docs/latest/api/java/), [Python](http://spark.apache.org/docs/latest/api/python/) and [R](http://spark.apache.org/docs/latest/api/R/).