在高级别，每个Spark应用程序都包含一个运行用户主要功能并在集群上执行各种并行操作的驱动程序。 Spark提供的主要抽象是一个弹性分布式数据集（RDD），它是在可以并行操作的集群节点之间划分的元素集合。 RDD通过从Hadoop文件系统（或任何其他Hadoop支持的文件系统）中的文件或驱动程序中的现有Scala集合开始进行创建，并进行转换。用户还可以要求Spark在内存中保留RDD，从而在并行操作中有效地重用RDD。最后，RDD自动从节点故障中恢复。

Spark中的第二个抽象是可以在并行操作中使用的共享变量。默认情况下，当Spark将并行功能作为不同节点上的一组任务并行运行时，它会将功能中使用的每个变量的副本发送给每个任务。有时，变量需要在任务之间或任务和驱动程序之间共享。 Spark支持两种类型的共享变量：广播变量，可用于缓存所有节点上的内存中的值，以及累加器，它们只是“添加”到诸如计数器和总和之间的变量。

本指南显示了Spark支持的每种语言中的每个功能。如果您启动Spark的交互式shell - Scala shell的bin / spark-shell或用于Python的bin / pyspark，那么最容易遵循。

# 与Spark连接

Spark 2.1.1与Python 2.6+或Python 3.4+配合使用。 它可以使用标准的CPython解释器，因此可以使用像NumPy这样的C库。 它也适用于PyPy 2.3+。

请注意，对于Spark 2.6.0，不支持对Python 2.6的支持，可能会在Spark 2.2.0中删除。

要在Python中运行Spark应用程序，请使用位于Spark目录中的bin / spark-submit脚本。 该脚本将加载Spark的Java / Scala库，并允许您将应用程序提交到集群。 您还可以使用bin / pyspark启动交互式Python shell。

如果您希望访问HDFS数据，则需要使用连接到您的HDFS版本的PySpark的构建。 Spark主页上还提供了预编译软件包，用于常见的HDFS版本。

最后，您需要将一些Spark类导入到程序中。 添加以下行：

**from pyspark import SparkContext, SparkConf**

PySpark在驱动程序和工作人员中都需要相同的小版本的Python。 它使用PATH中的默认python版本，您可以指定要由PYSPARK\_PYTHON使用的Python版本，例如：

**初始化Spark连接**

Spark程序必须做的第一件事是创建一个SparkContext对象，该对象告诉Spark如何访问集群。 要创建SparkContext，您首先需要构建一个SparkConf对象，其中包含有关应用程序的信息。

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

sc = SparkContext（conf = conf）

**appName**参数是应用程序在集群UI上显示的名称。 **master**是**Spark，Mesos或YARN**集群URL，或者**以本地模式运行的特殊“local”字符串**。 实际上，当在集群上运行时，您不需要在程序中硬编码主机，而是使用spark-submit启动应用程序并在其中接收。 但是，对于本地测试和单元测试，您可以通过“本地”运行Spark进程。

在PySpark shell中，已经为您创建了一个特殊的解释器感知**SparkContext**，在名为sc的变量中。 制作您自己的SparkContext将无法正常工作。 您**可以使用--master参数设置上下文连接的主机，并且可以将Python .zip，.egg或.py文件添加到运行时路径**，方法**是将逗号分隔的列表传递给--py-files。** 您还可以通过向-**-packages参数**提供逗号分隔的maven坐标列表，将依赖关系（例如Spark Packages）添加到shell会话。 可能存在依赖关系的任何其他存储库（例如Sonatype）都可以传递给--repositories参数。 必须使用pip手动安装Spark软件包具有的任何Python依赖关系（在该软件包的require.txt中列出）。 例如，要在**四个核心上运行bin / pyspark**，请使用：

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

或者，还要将**code.py添加到搜索路径（以便以后能够导入代码）**，请使用：

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

有关选项的完整列表，请运行pyspark --help。 在幕后，pyspark调用更一般的spark-submit脚本。

还可以在IPython中启动PySpark shell，这是增强型Python解释器。 PySpark与IPython 1.0.0及更高版本配合使用。 要使用IPython，在运行bin / pyspark时将PYSPARK\_DRIVER\_PYTHON变量设置为ipython：

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

使用jupyter notebook

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

您可以通过设置**PYSPARK\_DRIVER\_PYTHON\_OPTS来自定义ipython或jupyter命令**。

Jupyter Notebook服务器启动后，您可以从“文件”选项卡创建一个新的“Python 2”笔记本。 在笔记本电脑内部，您可以在笔记本电脑内部输入命令％pylab，然后再开始从Jupyter笔记本电脑上尝试Spark。

# 弹性分布式数据集（RDD）

Spark围绕着弹性分布数据集（RDD）的概念，RDD是一种可以并行运行的元件的容错集合。 创建RDD有两种方法：并行化驱动程序中的现有集合，或者在外部存储系统（如共享文件系统，HDFS，HBase或提供Hadoop InputFormat的任何数据源）中引用数据集。

## 并行集合

通过在您的驱动程序中的现有迭代或集合上调用SparkContext的parallelize方法来创建并行集合。集合的元素被复制以形成可并行操作的分布式数据集。例如，这里是如何创建一个保存数字1到5的并行集合：

data = [1，2，3，4，5]

distData = sc.parallelize（data）

一旦创建，分布式数据集（distData）可以并行运行。例如，我们可以调用distData.reduce（lambda a，b：a + b）来添加列表的元素。我们稍后介绍分布式数据集的操作。并行集合的一个重要参数是将数据集切割到的分区数。 Spark将为集群的每个分区运行一个任务。通常，您需要为集群中的每个CPU分配2-4个分区。通常，Spark会根据您的群集自动设置分区数。但是，您也可以通过将其作为第二个参数进行手动设置来进行并行设置（例如sc.parallelize（data，10））。注意：代码中的某些地方使用术语片（分区的同义词）来保持向后兼容性。

## 外部数据集

PySpark可以从Hadoop支持的任何存储源创建分布式数据集，包括本地文件系统，HDFS，Cassandra，HBase，Amazon S3等。Spark支持文本文件，SequenceFiles和任何其他Hadoop InputFormat。

文本文件RDD可以使用SparkContext的textFile方法创建。 该方法为文件（机器上的本地路径或hdfs：//，s3n：//，etc URI）获取URI，并将其作为行的集合读取。 这是一个示例调用：

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

一旦创建，distFile可以通过数据集操作来执行。例如，我们可以使用map将所有行的大小相加，并减少操作如下：distFile.map（lambda s：len（s））。reduce（lambda a，b：a + b）。

**有关Spark的阅读文件的一些注意事项：**

如果在本地文件系统上使用路径，该文件也必须在工作节点上的相同路径上可访问。将文件复制到所有工作人员或使用网络安装的共享文件系统。

Spark的所有基于文件的输入法，包括textFile，支持在目录，压缩文件和通配符上运行。例如，您可以使用textFile（“/ my / directory”），textFile（“/ my / directory / \*。txt”）和**textFile（“/ my / directory / \*。Gz”）。**

**textFile**方法还需要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为文件的每个块创建一个分区（HDFS中默认为128MB），但是也可以通过传递较大的值来请求更高数量的分区。请注意，您不能有比块少的分区。

**除了文本文件，Spark的Python API还支持其他几种数据格式：**

**SparkContext.wholeTextFiles（）**允许您读取包含多个小文本文件的目录，并将它们作为（文件名，内容）对返回。这与textFile相反，textFile将在每个文件中每行返回一条记录。

**RDD.saveAsPickleFile和SparkContext.pickleFile**支持以包含腌制Python对象的简单格式保存RDD。批次用于pickle序列化，默认批量大小为10。

**SequenceFile和Hadoop输入/输出格式**

请注意，此功能当前标记为实验，适用于高级用户。将来可能会使用基于Spark SQL的读/写支持来替换它，在这种情况下，Spark SQL是首选方法。可行支持PySpark SequenceFile支持在Java中加载键值对的RDD，将Writables转换为基本Java类型，并使用Pyrolite腌制生成的Java对象。当将键值对的RDD保存到SequenceFile时，PySpark的反向。它将Python对象解析为Java对象，然后将其转换为**Writables**。以下**Writables**将自动转换：

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

数组不是开箱即用的。 **用户需要在阅读或写入时指定自定义的ArrayWritable子类型。** 写入时，用户还需要指定将数组转换为自定义ArrayWritable子类型的自定义转换器。 读取时，**默认转换器将自定义的ArrayWritable子类型转换为Java Object []，然后将其转换为Python元组**。 要获取原始类型数组的Python array.array，用户需要指定自定义转换器。

**保存和加载SequenceFiles**

与文本文件类似，可以通过指定路径来保存和加载**SequenceFiles**。 可以指定键值和值类，但对于标准**Writables**，这不是必需的。

>>> 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')]

**保存和加载其他Hadoop输入/输出格式**

PySpark还可以读取任何Hadoop InputFormat或为“新”和“旧”Hadoop MapReduce API编写任何Hadoop OutputFormat。 如果需要，Hadoop配置可以作为Python dict传递。 以下是使用Elasticsearch ESInputFormat的示例：

**$ 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})

请注意，如果InputFormat仅仅依赖于Hadoop配置和/或输入路径，并且可以根据上表容易地转换密钥和值类，那么这种方法对于这种情况应该是很好的。

如果您有自定义的序列化二进制数据（例如从Cassandra / HBase加载数据），则首先需要将Scala / Java端的数据转换为可由Pyrolite的pickler处理的内容。 为此提供了转换器特性。 简单地扩展这个特性，并在convert方法中实现你的转换代码。 请记住，确保这个类以及访问您的InputFormat所需的任何依赖项被打包到您的Spark工作jar中，并包含在PySpark类路径中。

**有关使用Cassandra / HBase InputFormat和OutputFormat与自定义转换器的示例，请参阅Python示例和转换器示例。**

## RDD操作

RDD支持两种类型的操作：从现有操作创建新数据集的转换以及在数据集上运行计算后将值返回给驱动程序的操作。例如，**map是通过函数传递每个数据集元素并返回表示结果的新RDD的转换**。另一方面，r**educe是使用一些函数聚合RDD的所有元素并将最终结果返回给驱动程序的动作**（尽管还有一个并行的reduceByKey返回分布式数据集）。

**Spark中的所有转换都是懒惰的**，因为它们**不会马上计算它们的结果。**相反，他们只记住应用于某些基本数据集（例如文件）的转换。只有当一个动作需要将结果返回给驱动程序时，才会计算转换。这种设计使Spark能够更高效地运行。例如，我们可以意识到，通过map创建的数据集将被用于减少并仅将reduce的结果返回给驱动程序，而不是较大的映射数据集。

**默认情况下，每次转换的RDD可能会在每次对其进行操作时重新计算。**但是，**您也可以使用persist（或cache）方法在内存中保留RDD**，在这种情况下，Spark将在下次查询时将集群上的元素保留在更快的访问中。还支持在磁盘上持久的RDD，或跨多个节点进行复制。

为了说明RDD的基础知识，请考虑以下简单的程序：

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

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

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

第一行定义了一个外部文件的基础RDD。 此数据集未加载到内存中或以其他方式处理：行只是指向文件的指针。 第二行将lineLength定义为map变换的结果。 再次，由于懒惰，lineLengths不会立即计算。 最后，我们运行reduce，这是一个动作。 此时，Spark将计算分解为在单独的机器上运行的任务，每台机器都运行其map的一部分和本地缩减，仅返回其对驱动程序的答案。

如果我们以后也想再次使用lineLengths，我们可以添加：

**lineLengths.persist()**

在reduce之前，这将导致lineLength在第一次计算之后被保存在内存中。

## 将函数传递给Spark

Spark的API很大程度上依赖于驱动程序中的传递函数来运行在集群上。 推荐的方法有三种：Lambda表达式，用于可以写为表达式的简单函数。 （Lambdas不支持多语句函数或不返回值的语句。）本地def内部函数调用到Spark，更长的代码。模块中的顶级功能。

例如，要传递比使用lambda支持的更长的函数，请考虑以下代码：

**"""MyScript.py"""**

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

**def myFunc(s):**

**words = s.split(" ")**

**return len(words)**

**sc = SparkContext(...)**

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

请注意，虽然也可以传递对类实例（而不是单例对象）中的方法的引用，但这需要发送包含该类的对象以及该方法。 例如，考虑：

class MyClass(object):

def func(self, s):

return s

def doStuff(self, rdd):

return rdd.map(self.func)

在这里，如果我们创建一个新的MyClass并调用doStuff，那么其中的Map将引用该MyClass实例的func方法，因**此需要将整个对象发送到集群。**

**以类似的方式，访问外部对象的字段将引用整个对象：**

**class MyClass(object):**

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

**self.field = "Hello"**

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

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

为**了避免这个问题，最简单的方法是将字段复制到局部变量中，而不是外部访问它。**

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

**field = self.field**

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

## 了解并行化

Spark的一个较难的事情是在跨群集执行代码时了解变量和方法的范围和生命周期。 在其范围之外修改变量的RDD操作可能是一个常见的混乱来源。 在下面的示例中，我们将看看使用foreach（）来增加计数器的代码，但是对于其他操作也可能会出现类似的问题。

**例**

考虑下面的天真的RDD元素总和，其可能的行为不同，取决于执行是否在同一个JVM中发生。 一个常见的例子是在本地模式（--master = local [n]）运行Spark时，而不是将Spark应用程序部署到集群（例如，通过spark-submit to YARN）：

**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)

## 本地与群集模式

上述代码的行为未定义，可能无法正常工作。为了执行任务，Spark将RDD操作的处理分解为任务，每个任务由执行者执行。在执行之前，Spark会计算任务的关闭。封闭是对于执行器在RDD（在这种情况下为foreach（））执行其计算必须可见的变量和方法。该关闭序列化并发送给每个执行器。

发送给每个执行器的闭包中的变量现在是副本，因此当在foreach函数中引用计数器时，它不再是驱动程序节点上的计数器。在驱动程序节点的内存中仍然有一个计数器，但这对执行程序来说已经不再可见了！执行者只看到序列化关闭的副本。因此，计数器的最终值仍然为零，因为计数器上的所有操作都引用了序列化闭包中的值。

在本地模式下，在某些情况下，foreach函数实际上将在与驱动程序相同的JVM内执行，并引用相同的原始计数器，并可能实际更新它。

为了确保在这些场景中明确定义的行为，应该使用累加器。 Spark中的累加器专门用于提供一种机制，用于在群集中的工作者节点上执行分割时安全更新变量。本指南的“累加器”部分更详细地讨论了这些。

一般来说，闭包 - 构造如循环或本地定义的方法不应该用于突变某些全局状态。 Spark不定义或保证从关闭外部引用的对象的突变行为。一些代码可以在本地模式下工作，但这只是偶然的，这样的代码在分布式模式下不会像预期那样运行。如果需要一些全局聚合，则使用累加器。

# 打印RDD元素

另一个常见的成语是使用rdd.foreach（println）或rdd.map（println）打印RDD的元素。 在单个机器上，这将产生预期的输出并打印所有RDD的元素。 但是，在群集模式下，执行程序调用的stdout的输出现在正在写入执行程序的stdout，而不是驱动程序上的stdout，因此驱动程序上的stdout不会显示这些！ 要打印驱动程序中的所有元素，可以使用collect（）方法首先将RDD带到驱动程序节点：rdd.collect（）。foreach（println）。 这可能会导致驱动程序内存不足，因为collect（）将整个RDD提取到一台机器上; 如果只需要打印RDD的几个元素，一个更安全的方法是使用take（）：rdd.take（100）.foreach（println）。

## 使用键值对

虽然大多数Spark操作适用于包含任何类型对象的RDD，但是几个特殊操作只能在键值对的RDD上使用。 最常见的是分布式“随机播放”操作，例如按键分组或聚合元素。

在Python中，这些操作适用于包含内置Python元组的RDD，例如（1,2）。 只需创建这样的元组，然后调用所需的操作。

例如，以下代码使用键值对上的reduceByKey操作来计算每行文本在文件中的发生次数：

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

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

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

例如，我们也可以使用counts.sortByKey（）来按字母顺序排列对，最后将counts.collect（）作为对象列表返回到驱动程序。

转换

下表列出了Spark支持的一些常见转换。 有关详细信息，请参阅RDD API文档（Scala，Java，Python，R）和RDD函数doc（Scala，Java）。

|  |  |
| --- | --- |
| Transformation | Meaning |
| **map**(*func*) | 返回通过函数func传递源的每个元素形成的新的分布式数据集。 |
| **filter**(*func*) | 返回通过选择func返回true的源的元素形成的新数据集。 |
| **flatMap**(*func*) | 类似于map，但是每个输入项可以映射到0个或更多的输出项（所以func应该返回一个Seq而不是单个项）。 |
| **mapPartitions**(*func*) | **与map类似，但在RDD的各自分区（块）上单独运行，所以当在类型T的RDD上运行时，func必须为Iterator <T> => Iterator <U>。** |
| **mapPartitionsWithIndex**(*func*) | 与mapPartitions类似**，但也提供了一个表示分区索引的整数值的func**，所以当在类型为T的RDD上运行时，func必须是类型**（Int，Iterator <T>）=> Iterator <U>。** |
| **sample**(*withReplacement*, *fraction*, *seed*) | **使用给定的随机数发生器种子对有数据的一部分数据进行采样或不进行替换。** |
| **union**(*otherDataset*) | **返回包含源数据集和参数中元素的并集的新数据集。** |
| **intersection**(*otherDataset*) | 返回一个包含源数据集和参数中的元素的新RDD。 |
| **distinct**([*numTasks*])) | **返回一个包含源数据集的不同元素的新数据集。** |
| **groupByKey**([*numTasks*]) | 当对（K，V）对的数据集进行调用时，返回（K，Iterable <V>）对的数据集。  **注意：如果要分组以便在每个键上执行聚合（如总和或平均值），则使用reduceByKey或aggregateByKey将产生更好的性能。**  **注意：默认情况下，输出中的并行级别取决于父RDD的分区数。 您可以传递可选的numTasks参数来设置不同数量的任务。** |
| **reduceByKey**(*func*, [*numTasks*]) | 当（K，V）对的数据集被调用时，返回一个数据集（K，V）**对，其中使用给定的reduce函数func聚合每个键的值，该函数必须是类型（V，V）=> V.像groupByKey一样，可以通过可选的第二个参数来配置reduce任务的数量。** |
| **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*]) | 当对（K，V）对的数据集进行调用时，**返回（K，U）对的数据集**，其中使**用给定的组合函数和中性“零”值对每个键的值进行聚合。** 允许与输入值类型不同的聚合值类型，同时避免不必要的分配。 像groupByKey一样，可以通过可选的第二个参数来配置reduce任务的数量。 |
| **sortByKey([*ascending*], [*numTasks*])** | 在**K实现有序的（K，V）对的数据集上被调用时，按照布尔升序参数中指定的按升序或降序的顺序返回按键排列的（K，V）对的数据集。** |
| **join**(*otherDataset*, [*numTasks*]) | 当对类型（K，V）和（K，W）的数据集进行调用时，返回一个（K，（V，W））对的数据集与每个键的所有元素对。 通过leftOuterJoin，rightOuterJoin和fullOuterJoin支持外连接。 |
| **cogroup**(*otherDataset*, [*numTasks*]) | 当调**用类型（K，V）和（K，W）的数据集时**，返回一个数据集**（K，（Iterable <V>，Iterable W）））**元组。 此操作也称为**GroupWith**。 |
| **cartesian**(*otherDataset*) | 当对**类型T和U的数据集进行调用时，返回（T，U）对（所有元素对）的数据集**。 |
| **pipe**(*command*, *[envVars]*) | 通过shell命令管道RDD的每个分区，例如 一个Perl或bash脚本。 **RDD元素被写入进程的stdin，并且将其stdout的行输出作为字符串的RDD返回。** |
| **coalesce**(*numPartitions*) | 将RDD中的分区数减少到numPartition。 用于过滤大型数据集后更有效地运行操作。 |
| **repartition(*numPartitions*)** | 随机重新清理RDD中的数**据以创建更多或更少的分区，并在其间平衡。** 这总是通过网络洗牌所有的数据。 |
| **repartitionAndSortWithinPartitions**(*partitioner*) | 根据给定的分区器重新分配RDD，并且在每个生成的分区中，通**过它们的键对记录进行排序**。 这比调用重新分区，然后在每个分区中排序更有效，因为它可以将排序推入洗牌机器。 |

# 操作

下表列出了Spark支持的一些常见操作。 参考RDD API文档（Scala，Java，Python，R）和RDD函数doc（Scala，Java）的详细信息。

|  |  |
| --- | --- |
| Action | Meaning |
| **reduce**(*func*) | 使用函数func（它需要两个参数并返回一个）**来聚合数据集的元素。** 该函数应该是交换和关联的，以便它可以并行计算。 |
| **collect**() | 在驱动程序中**将数据集的所有元素作为数组返回**。 这通常在返回足够小的数据子集的过滤器或其他操作之后很有用。 |
| **count**() | 返回数据集中的元素数。 |
| **first**() | **返回数据集的第一个元素（类似于（1））** |
| **take**(*n*) | 返回**数组与数据集的前n个元素。** |
| **takeSample**(*withReplacement*, *num*, [*seed*]) | **返回具有数据集的num个元素的随机样本的数组，可以带或不带替换，可选地预先指定随机数生成器种子。** |
| **takeOrdered**(*n*, *[ordering]*) | 使用**自然顺序或自定义比较器**返回RDD的前n个元素。 |
| **saveAsTextFile**(*path*) | 在**本地文件系统，HDFS或任何其他Hadoop支持的文件系统**的给定目录中，将数据集的元素作为文本文件（或一组文本文件）编写。 Spark将在每个元素上调用toString将其转换为文件中的一行文本。 |
| **saveAsSequenceFile**(*path*)  (Java and Scala) | 在本地文件系统，HDFS或任何其他Hadoop支持的文件系统的给定路径中，**将数据集的元素编写为Hadoop SequenceFile。** 这可用于实现Hadoop可写接口的键值对的RDD。 在Scala中，它也可以隐式转换为Writable（Spark包括基本类型的转换，如Int，Double，String等）。 |
| **saveAsObjectFile**(*path*)  (Java and Scala) | 使用Java序列化以简单的格式写入数据集的元素，然后可以使用**SparkContext.objectFile（）加载数据集。** |
| **countByKey**() | 仅适用于类型（K，V）的RDD。 返回与每个键的计数的（K，Int）对的hashmap。 |
| **foreach**(*func*) | 在**数据集的每个元素上运行函数func**。 这通常用于副作用，例如更新累加器或与外部存储系统交互。 **注意：除了foreach（）之外的累加器以外的其他变量可能会导致未定义的行为。** 有关详细信息，请参阅了解闭包。 |

Spark RDD API还暴露了一些操作的异步版本，例如foreachAsync for foreach，它立即将FutureAction返回给调用者，而不是在完成操作时阻止。 这可以用于管理或等待异步执行操作。

# 洗牌操作

Spark内的某些操作会触发称为随机播放的事件。 shuffle是Spark的重新分布数据的机制，因此它在分区之间的分组不同。这通常涉及将数据复制到执行器和机器上，从而使洗牌成为复杂而昂贵的操作。

## 背景

要了解shuffle中会发生什么，我们可以考虑reduceByKey操作的示例。 reduceByKey操作生成一个新的RDD，其中单个密钥的所有值都被组合成一个元组 - 关键字和执行reduce函数的结果与所有与该关键字关联的值。挑战在于，**并不是单个密钥的所有值都必须驻留在同一个分区上，甚至是同一个机器上，但它们必须位于同一位置才能计算结果。**

在Spark中**，数据通常不会跨分区分布**，**以便在特定操作的必要位置。**在计算过程中，单个任务将在单个分区上运行 - 因此，为**了执行单个reduceByKey还原任务来组织所有数据，Spark需要执行一对一的操作。**它必须从所有分区中读取以查找所有键的所有值，然后将分区中的值汇集在一起​​，以计算每个键的最终结果 - 这称为**随机播放**。

虽然**新混洗数据的每个分区中的元素集将是确定性的**，分区本身的排序也是如此，但是这些元素的排序不是。如果一个人想要随机播放之后可预测的有序数据，那么可以使用：

1. **mapPartitions以使用例如.sorted对每个分区进行排序**
2. **repartitionAndSortWithinPartitions有效地对分区进行分类，同时重新分区**
3. **sortBy来制作一个全局排序的RDD**

可能导致洗牌的操作包括重新分区操作，如重新分区和合并，“ByKey操作（除了计数），如groupByKey和reduceByKey），并加入像cogroup和join这样的操作。

## 性能影响

随机播**放是一项昂贵的操作，**因为它涉及**磁盘I / O，数据串行化和网络I / O**。要组织随机播放的数据，Spark会生成一组**任务 - map任务以组织数据**，以及一**组reduce任务以进行汇总。**这个命名法来自于**MapReduce**，并不直接与Spark的map和reduce操作有关。

在内部，单独的map任务的结果将保存在内存中，直到它们不适合为止。然后，这些根据目标分区进行排序并写入单个文件。在减少的一面，任务读取相关的排序块。

某些随机播放操作可能会占用大量的**堆内存**，因为它们在传输之前或之后使用内存中的数据结构来组织记录。具体来说，reduceByKey和aggregateByKey在map上创建这些结构，而ByKey操作会在reduce方面生成这些结构。当数据不适合内存时，Spark会将这些表溢出到磁盘，导致磁盘I / O的额外开销和增加的垃圾回收。

随机播放还会在磁盘上生成大量的中间文件。从Spark 1.3开始，这些文件将被保留，直到相应的RDD不再使用并被垃圾回收。这样做，所以如果重新计算谱系，则不需要重新创建洗牌文件。如果应用程序保留对这些RDD的引用或GC不频繁启动，垃圾收集可能仅在长时间之后才会发生。这意味着长时间运行的Spark作业可能会消耗大量的磁盘空间。当配置Spark上下文时，临时存储目录由spark.local.dir配置参数指定。

可以通过调整各种配置参数来调整洗牌行为。请参阅“Spark配置指南”中的“随机行为”部分。

## RDD持久性

Spark中最重要的功能之一是在操作中持久（或缓存）内存中的数据集。当您持有RDD时，每个节点都会将其计算的任何分区存储在内存中，并将其重用于该数据集（或从其导出的数据集）的其他操作。这样可以使未来的动作快得多（通常超过10倍）。缓存是迭代算法和快速交互使用的关键工具。

您可以使用其上的persist（）或cache（）方法标记要持久化的RDD。第一次在动作中计算，它将被保存在节点的内存中。 Spark的缓存是容错的 - 如果RDD的任何分区丢失，它将使用最初创建的转换自动重新计算。

此外，可以使用不同的存储级别存储每个持久化的RDD，从而允许您将数据集保留在磁盘上，将其保存在内存中，但将其作为序列化Java对象（以节省空间），将其复制到节点上。通过将StorageLevel对象（Scala，Java，Python）**传递给persist（）**来设置这些级别。**存储级别**

|  |  |
| --- | --- |
| Storage Level | Meaning |
| MEMORY\_ONLY | 将RDD作为反序列化的Java对象存储在JVM中。 如果RDD不适合内存，某些分区将不会被缓存，每次需要时都会重新计算。 这是默认级别。 |
| MEMORY\_AND\_DISK | 将RDD作为反序列化的Java对象存储在JVM中。 如果RDD不适合内存，请存储不适合磁盘的分区，并在需要时从那里读取。 |
| MEMORY\_ONLY\_SER  (Java and Scala) | 将RDD存储为序列化的Java对象（每个分区一个字节数组）。 这通常比反序列化对象更节省空间，特别是在使用快速串行器时，但读取CPU耗费更多。 |
| MEMORY\_AND\_DISK\_SER  (Java and Scala) | 与MEMORY\_ONLY\_SER类似，但将不适合内存的分区溢出到磁盘，而不是每次需要重新计算它们。 |
| DISK\_ONLY | 将RDD分区存储在磁盘上。 |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, etc. | 与上述级别相同，但复制两个群集节点上的每个分区。 |
| OFF\_HEAP (experimental) | 与MEMORY\_ONLY\_SER类似，但将数据存储在堆内存中。 这需要启用脱堆内存。 |

注意：**在Python中，存储的对象将始终使用Pickle库进行序列化，所以选择序列化级别并不重要。** Python中的可用存储级别包括MEMORY\_ONLY，MEMORY\_ONLY\_2，MEMORY\_AND\_DISK，MEMORY\_AND\_DISK\_2，DISK\_ONLY和DISK\_ONLY\_2。

即使没有用户显示声明持久性，Spark也会以随机播放操作（例如reduceByKey）自动保留一些中间数据。 这样做是为了避免重新计算整个输入，如果在随机播放期间节点出现故障。 如果计划重用它，我们仍然建议用户在生成的RDD上调用持久性。

**哪个存储级别可供选择？**

Spark的存储级别旨在提供内存使用和CPU效率之间的不同权衡。我们建议您通过以下过程来选择一个：

1. 如果您的RDD适合于默认存储级别（MEMORY\_ONLY），请以这种方式离开。这是CPU效率最高的选项，允许RDD上的操作尽可能快地运行。
2. 如果没有，请尝试使用MEMORY\_ONLY\_SER并选择一个快速序列化库来使对象更加节省空间，但仍然能够快速访问。 （Java和Scala）
3. 不要溢出到磁盘，除非计算您的数据集的功能是昂贵的，或者它们过滤大量的数据。否则，重新计算分区可能与从磁盘读取分区一样快。
4. 如果您希望快速故障恢复（例如，如果使用Spark来服务来自Web应用程序的请求），请使用复制的存储级别。所有存储级别通过重新计算丢失的数据来提供完整的容错能力，但复制的数据可让您继续在RDD上运行任务，而无需重新计算丢失的分区。

## 删除数据

Spark会自动监视每个节点的缓存使用情况，并以最近最少使用（LRU）方式丢弃旧数据分区。如果要手动删除RDD，而不是等待它脱离缓存，请使用RDD.unpersist（）方法。

## 共享变量

通常，当传递到Spark操作（例如map或reduce）的函数在远程集群节点上执行时，它可以在函数中使用的所有变量的单独副本上工作。这些变量被复制到每个机器，并且远程机器上的变量的更新都不会被传播回到驱动程序。在任务之间支持一般的，读写共享变量将是低效的。然而，Spark为两种常用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。

## 广播变量

广播变量允许程序员在每个机器上保留只读变量，而不是使用任务运送它的副本。例如，可以使用它们以有效的方式为每个节点提供大型输入数据集的副本。 Spark还尝试使用高效的广播算法分发广播变量，以降低通信成本。

Spark动作通过一组分阶段执行，由分散的“随机播放”操作隔开。 Spark自动广播每个阶段任务所需的公共数据。以这种方式广播的数据以序列化形式缓存，并在运行每个任务之前进行反序列化。这意味着，显式创建广播变量只有在跨多个阶段的任务需要相同数据或者以反序列化格式缓存数据很重要时才有用。

**广播变量通过调用SparkContext.broadcast（v）从变量v创建**。广播变量是围绕v的包装器，其值可以通过调用value方法来访问。下面的代码显示：

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

<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value

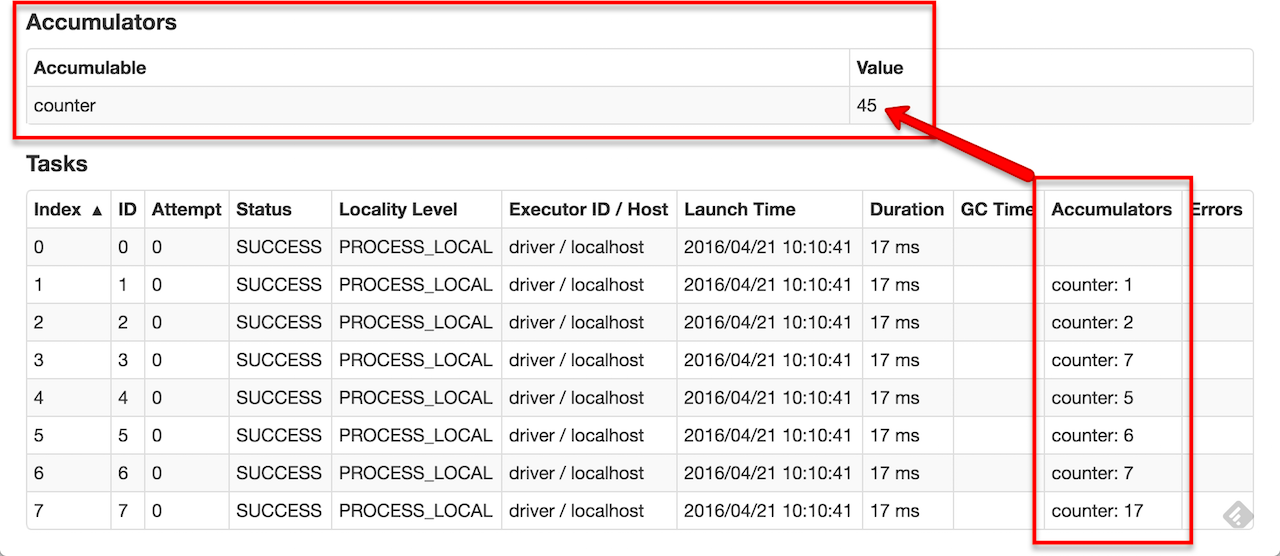
[1, 2, 3]

**创建广播变量后，应在群集中运行的任何函数中使用它而不是值v，以便v不会多次发送到节点。** 此外，对象v在广播之后不应被修改，以便确保所有节点获得与广播变量相同的值（例如，如果变量稍后发送到新节点）。

## 累加器

累加器是仅通过关联和交换操作“添加”的变量，因此可以并行有效地支持。 它们可用于实现计数器（如MapReduce）或总和。 Spark本身支持数字类型的累加器，程序员可以添加对新类型的支持。

作为用户，您可以创建命名或未命名的累加器。 如下图所示，一个命名的累加器（在本例中为计数器）将显示在web UI中，用于修改该累加器的阶段。 Spark在“任务”表中显示任务修改的每个累加器的值。

在UI中跟踪累加器对于了解运行阶段的进度可能是有用的（注意：这在Python中尚不支持）。

通过调用SparkContext.accumulator（v）从初始值v创建一个累加器。 然后可以使用add方法或+ =运算符来添加在集群上运行的任务。 但是，他们看不到它的价值。 只有驱动程序可以使用其值方法读取累加器的值。

下面的代码显示了一个累加器用于将数组的元素相加：

>>> accum = sc.accumulator(0)

>>> accum

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

>>> accum.value

10

虽然此代码使用内置的Int类型的累加器支持，但程序员也可以通过对AccumulatorParam进行子类化来创建自己的类型。 AccumulatorParam接口有两种方法：零为您的数据类型提供“零值”，而addInPlace用于将两个值一起添加。 例如，假设我们有一个代表数学向量的Vector类，我们可以写：

**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())**

对于仅在动作内部执行的累加器更新，Spark保证每个任务对累加器的更新只会被应用一次，即重新启动的任务将不会更新该值。 在转换中，用户应该意识到，如果重新执行任务或作业阶段，则每个任务的更新可能会被多次应用。

累加器不改变Spark的懒惰评估模型。 如果在RDD的操作中更新它们，则只有在RDD作为操作的一部分计算时，才会更新其值。 因此，累加器更新不能保证在像map（）这样的懒惰变换中执行。 以下代码片段演示此属性：

**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.**

# 部署到集群

应用程序提交指南介绍如何将应用程序提交到集群。简而言之，一旦将应用程序打包到JAR（用于Java / Scala）或一组.py或.zip文件（对于Python）中，bin / spark-submit脚本可以将其提交给任何支持的集群管理器。

## 从Java / Scala启动Spark作业

org.apache.spark.launcher包提供了使用简单Java API将Spark作为子进程启动的类。

## 单元测试

Spark可以与任何流行的单元测试框架进行单元测试。只需在测试中创建一个SparkContext，将主URL设置为本地，运行您的操作，然后调用SparkContext.stop（）将其拆下来。确保在finally块或测试框架的tearDown方法中停止上下文，因为Spark不支持在同一程序中同时运行的两个上下文。

从哪里走

您可以在Spark网站上看到一些示例Spark程序。此外，Spark包含了examples目录中的几个示例（Scala，Java，Python，R）。您可以通过将类名传递给Spark的bin / run-example脚本来运行Java和Scala示例;例如：

**./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**

有关优化程序的帮助，配置和调整指南提供有关最佳做法的信息。 它们对于确保您的数据以有效的格式存储在内存中尤其重要。 有关部署的帮助，集群模式概述描述了分布式操作涉及的组件和支持的集群管理器。

最后，Scala，Java，Python和R中提供了完整的API文档。