Spark 编程指南__Python 版_2016/4/15

一、概述：

从高层次上来看，每一个 Spark 应用都包含一个驱动程序，用于执行用户的 main 函数以及在集群上运行各种并行操作。Spark 提供的主要抽象是弹性分布式数据集（RDD），这是一个包含诸多元素、被划分到不同节点上进行并行处理的数据集和。RDD 通过打开 HDFS（或其他 hadoop 支持的文件系统）上的一个文件、在驱动程序中打开一个已有的 Scala 集合或由其他 RDD 转换操作得到。用户可以要求 Spark 将 RDD 持久化到内存中，这样就可以有效地在并行操作中复用。另外，在节点发生错误时 RDD 可以自动恢复。

Spark 提供的另一个抽象是可以在并行操作中使用的共享变量。在默认情况下，当 Spark 将一个函数转化成许多任务在不同的节点上运行的时候，对于所有在函数中使用的变量，每一个任务都会得到一个副本。有时，某一个变量需要在任务之间或任务与驱动程序之间共享。Spark 支持两种共享变量：广播变量，用来将一个值缓存到所有节点的内存中；累加器，只能用与累加，比如计数器和求和。

这篇指南将展示这些特性在 Spark 支持的语言中是如何使用的（本文只翻译了 Python 部分）。如果你打开了 Spark 的交互命令行—— bin/spark-shell 的 Scala 命令行或 bin/pyspark 的 Python 命令行都可以——那么这篇文章你学习起来将是很容易的。

二、连接 Spark

Spark1.3.0 只支持 Python2.6 或更高的版本（但不支持 Python3）。它使用了标准的 CPython 解释器，所以诸如 NumPy 一类的 C 库也是可以使用的。

通过 Spark 目录下的 bin/spark-submit 脚本你可以在 Python 中运行 Spark 应用。这个脚本会载入 Spark 的 Java/Scala 库然后让你将应用提交到集群中。你可以执行 bin/pyspark 来打开 Python 的交互命令行。

如果你希望访问 HDFS 上的数据，你需要为你使用的 HDFS 版本建立一个 PySpark 连接。常见的 HDFS 版本标签都已经列在了这个第三方发行版页面。

最后，你需要将一些 Spark 的类 import 到你的程序中。加入如下这行：

- `from pyspark import SparkContext, SparkConf`

三、初始化 Spark

在一个 Spark 程序中要做的第一件事就是创建一个 SparkContext 对象来告诉 Spark 如何连接一个集群。为了创建 SparkContext，你首先需要创建一个 SparkConf 对象，这个对象会包含你的应用的一些相关信息。

- conf = SparkConf().setAppName("PythonPi").setMaster(Master)
- sc = SparkContext(conf=conf)

appName 参数是在集群 UI 上显示的你的应用的名称。master 是一个 Spark、Mesos 或 YARN 集群的 URL,如果你在本地运行那么这个参 数应该是特殊的"local"字符串。在实际使用中，当你在集群中运行你的程序，你一般不会把 master 参数写死在代码中，而是通过用 spark- submit 运行程序来获得这个参数。但是，在本地测试以及单元测试时，你仍需要自行传入"local"来运行 Spark 程序。

四、使用命令行

在 PySpark 命令行中，一个特殊的集成在解释器里的 SparkContext 变量已经建立好了，变量名叫做 sc。创建你自己的 SparkContext 不会起作用。你可以通过使–master 命令行参数来设置这个上下文连接的 master 主机，你也可以通--py-files 参数传递一个用逗号隔开的列表来将 Python 的.zip、.egg 或.py 文件添加到运行时路径中。你还可以通过–package 参数传递一个用逗号隔开的 maven 列表来给这个命令行会话添加依赖（比如 Spark 的包）。任何额外的包含依赖包的仓库（比如 SonaType）都可以通过传-- repositorys 参数来添加进去。Spark 包的所有 Python 依赖（列在这个包的 requirements.txt 文件中）在必要时都必须通过 pip 手动安装。

比如，使用四核来运行 bin/pyspark，应当输入这个命令：

- ./bin/pyspark –master local[4]

又比如，把 code.py 文件添加到搜索路径中（为了能够 import 在程序中），应当使用这条命令：

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

想要了解命令行选项的完整信息请执行 pyspark –help 命令。在这些场景下，pyspark 会触发一个更通用的 spark-submit 脚本。

在 IPython 这个加强的 Python 解释器中运行 PySpark 也是可行的。PySpark 可以在 1.0.0 或更高版本的 IPython 上运行。为了 使用 IPython，必须在运行 bin/pyspark 时将 PYSPARK_DRIVER_PYTHON 变量设置为 ipython，就像这样：

- PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark

你还可以通过设置 PYSPARK_DRIVER_PYTHON_OPTS 来自省定制 ipython。比如，在运行 IPython Notebook 时开启 PyLab 图形支持应该使用这条命令：

- PYSPARK_DRIVER_PYTHON=ipython
  PYSPARK_DRIVER_PYTHON_OPTS="notebook –pylab
  inline" ./bin/pyspark

五、弹性分布式数据集（RDD）

Spark 是以 RDD 概念为中心运行的。RDD 是一个容错的、可以被并行操作的元素集合。创建一个 RDD 有两个方法：在你的驱动程序中**并行化**一个已经存在的集合；从外部存储系统中引用一个数据集，这个存储系统可以是一个共享文件系统，比如 HDFS、HBase 或任意提供了 Hadoop 输入格式的数据来源。

5.1、并行化集合

并行化集合是通过在驱动程序中一个现有的迭代器或集合上调用 SparkContext 的 parallelize 方法建立的。为了创建一个能够并行操作的分布数据集，集合中的元素都会被拷贝。比如，以下语句创建了一个包含 1 到 5 的并行化集合：

- `data = [1, 2, 3, 4, 5]`
- `distData = sc.parallelize(data)`

分布数据集（distData）被建立起来之后，就可以进行并行操作了。比如，我们可以调用 distData.reduce(lambda a,b: a+b)来对元素进行叠加。在后文中我们会描述分布数据集上支持的操作。

并行集合的一个重要参数是将数据集划分成分片的数量。对每一个分片，Spark 会在集群中运行一个对应的任务。典型情况下，集群中的每一个 CPU 将对应运行 2-4 个分片。一般情况下，Spark 会根据当前集群的情况自行设定分片数量。但是，你也可以通过将第二个参数传递给 parallelize 方法（比如 sc.parallelize(data, 10)）来手动确定分片数量。注意：有些代码中会使用切片（slice，分片的同义词）这个术语来保持向下兼容性。

5.2、外部数据集

PySpark 可以通过 Hadoop 支持的外部数据源（包括本地文件系统、HDFS、Cassandra、HBase、亚马逊 S3 等等）建立分布数据集。Spark 支持文本文件、序列文件以及其他任何 Hadoop 输入格式文件。

通过文本文件创建 RDD 要使用 SparkContext 的 textFile 方法。这个方法会使用一个文件的 URL（或本地文件路径，hdfs://，s3n://这样的 URL 等等）然后读入这个文件建立一个文本行的集合。以下是一个例子：

- `distFile = sc.textFile("data.txt")`

建立完成后 distFile 上就可以调用数据集操作了。比如，我们可以调用 map 和 reduce 操作来叠加所有文本行的长度，代码如下：

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

在 Spark 中读入文件时有几点要注意：

（1）如果使用了本地文件路径时，要保证在 worker 节点上这个文件也能够通过这个路径访问。这点可以通过将这个文件拷贝到所有 worker 上或者使用网络挂载的共享文件系统来解决。

（2）包括 textFile 在内的所有基于文件的 Spark 读入方法，都支持将文件夹、压缩文件、包含通配符的路径 howe 参数。比如，以下代码都是合法的：

- textFile(“/my/directory”)
- textFile(“/my/directory/*.txt”)
- textFile(“/my/directory/*.gz”)

（3）textFile 方法也可以传入第二个可选参数来控制文件的分片数量。默认情况下，Spark 会为文件的每一个块（在 HDFS 中块的大小默认是 64MB）创建一个分片。但是你也可以通过传入一个更大的值来要求 Spark 建立更多的分片。注意，分片的数量绝不能小于文件块的数量。

除了文本文件之外，Spark 的 Python API 还支持多种其他数据格式：

（1）SparkContext.sholeTextFiles 能够读入包含多个小文本文件的目录，然后为每一个文件返回一个（文件名，内容）对。这是与 textFile 方法为每一个文本行返回一条记录相对应的。

（2）RDD.saveAsPickleFile 和 SparkContext.pickleFile 支持将 RDD 以串行化的 Python 对象格式存储起来。串行化的过程中会以默认 10 个一批的数量批量处理。

（3）序列文件和其他 Hadoop 输入输出格式。

注意：这个特性目前仍处于试验阶段，被标记为 Experimental，目前只适用于高级用户。这个特性在未来可能会被基于 Spark SQL 的读写支持所取代，因为 Spark SQL 是更好的方式。

5.3、可写类型支持

PySpark 序列文件支持利用 Java 作为中介载入一个键值对 RDD，将可写类型转化成 Java 的基本类型，然后使用 Pyrolite 将 java 结果对象串行化。当将一个键值对 RDD 储存到一个序列文件中时 PySpark 将会运行上述过程的相反过程。首先将 Python 对象反串行化成 Java 对象，然后转化成可写类型。以下可写类型会自动转换：

可写类型	Python 类型
Text	Unicode str
IntWritable	float

FloatWritable	float
DoubleWritable	float
BooleanWritable	bool
BytesWritable	bytearray
NullWritable	None
MapWritable	dict

数组是不能自动转换的。用户需要在读写时指定 ArrayWritable 的子类行。在读入的时候，默认的转换器会把自定义的 ArrayWritable 子类型转化成 Java 的 Object[]，之后串行化成 Python 的元组。为了获得 Python 的 array.array 类型来使用主要类型的数组，用户需要自行指定转换器。

5.3.1、保存和读取序列文件

和文本文件类似，序列文件可以通过指定路径来保存和读取。键值类型都可以自行指定，但是对于标准可写类型可以不指定。

- rdd = sc.paralelize(range(1, 4)).map(lambda x: (x, "a" * x))
- rdd.saveAsSequenceFile("path/to/file")
- sorted(sc.sequenceFile("path/to/file").collet())

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

注：SequenceFile 文件是 Hadoop 用来存储二进制形式的 key-value 对而设计的一种平面文件(Flat File)。

5.3.2、保存和读取其他 Hadoop 输入输出格式

PySpark 同样支持写入和读出其他 Hadoop 输入输出格式，包括“新”和“旧”两种 Hadoop MapReduce API。如果有必要，一个 Hadoop 配置可以以 Python 字典的形式传入。以下是一个例子，使用了 Elasticsearch ESInputFormat：

注：ElasticSearch 是一个基于 Lucene 的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎。

- SPARK_CLASSPATH=/path/to/elasticsearch-hadoop.jar ./bin/pyspark
- conf = { "es.resoutce": "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.my.LinkedMapWritable", conf = conf )
```

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

```
>> (u'Elasticsearch ID',
    {u'field1': True,
     u'field2': u'Some Text',
     u'field3': 12345})
```

注意，如果这个读入格式仅仅依赖于一个 Hadoop 配置和/或输入路径，而且键值类型都可以根据前面的表格转换，那么刚才提到的这种方法非常合适。

如果你有一些自定义的序列化二进制数据（比如从 Cassandra/HBase）中读取数据，那么你需要首先在 Scala/Java 端将这些数据转化成可以被 Pyrolite 的串行化器处理的数据类型。一个转换器特质已经提供好了。简单地拓展这个特质同时在 convert 方法中实现你自己的转换代码即可。记住，要确保这个类以及访问你的输入格式所需的依赖都被打到了 Spark 作业包中，并且确保这个包已经包含到了 PySpark 的 classpath 中。

这里有一些通过自定义转换器来使用 Cassandra/HBase 输入输出格式的 Python 样例和转换器样例。

5.4、RDD 操作

RDD 支持两类操作：**转化操作**，用于从已有的数据集转化产生新的数据集；**启动操作**，用于在计算结束后向驱动程序返回结果。举个例子，map 是一个转化操作，可以将数据集中每一个元素传给一个函数，同时将计算结果作为一个新的 RDD 返回。另一方面，reduce 操作是一个启动操作，能够使用某些函数来聚集计算 RDD 中所有的元素，并且向驱动程序返回最终结果（同时还有一个并行的 reduceByKey 操作可以返回一个分布数据集）。

在 Spark 所有的转化操作都是**惰性求值**的，就是说它们并不会立刻真的计算出结果。相反，它们仅仅是记录下了转换操作的操作对象（比如：一个文件）。只有当一个启动操作被执行，要向驱动程序返回结果时，转化操作才会真的开始计算。这样的设计使得 Spark 运行更加高效——比如，我们会发觉由 map 操作产生的数据集将会在 reduce 操作中用到，之后仅仅是返回了 reduce 的最终的结果而不是 map 产生的庞大数据集。

在默认情况下，每一个由转化操作得到的 RDD 都会在每次执行启动操作时重新计算生成。但是，你也可以通过调用 persist（或 cache）方法来将 RDD 持久化到内存中，这样 Spark 就可以在下次使用这个数据集时快速获得。Spark 同样提供了对将 RDD 持久化到硬盘上或在多个节点间复制的支持。

5.5、基本操作

为了演示 RDD 的基本操作，请看以下的简单程序：

- lines = sc.textFile(“data.txt”)
- lineLengths = lines.map(lambda s: len(s))
- totalLength = lineLengths.reduce(lambda a, b: a + b)

第一行定义了一个由外部文件产生的基本 RDD。这个数据集不是从内存中载入的也不是由其他操作产生的；lines 仅仅是一个指向文件的指针。第二行将 lineLengths 定义为 map 操作的结果。再强调一次，由于惰性求值的缘故，lineLengths 并不会被立即计算得到。最后，我们运行了 reduce 操作，这是一个启动操作。从这个操作开始，Spark 将计算过程划分到许多任务并在多机上运行，每台机器运行自己部分的 map 操作和 reduce 操作，最终将自己部分的运算结果返回给驱动程序。

如果我们希望以后重复使用 lineLengths，只需在 reduce 前加入下面这行代码：

- lineLengths.persist()

这条代码将使得 lineLengths 在第一次计算生成之后保存在内存中。

5.6、向 Spark 传递函数

Spark 的 API 严重依赖于向驱动程序传递函数作为参数。有三种推荐的方法来传递函数作为参数。

（1）Lambda 表达式，简单的函数可以直接写成一个 lambda 表达式（lambda 表达式不支持多语句函数和无返回值的语句）。

（2）对于代码很长的函数，在 Spark 的函数调用中在本地用 def 定义。

（3）模块中的顶级函数。

比如，传递一个无法转化为 lambda 表达式长函数，可以像以下代码这样：

- “MyScript.py”””
- if __name__ == “__main__”:
- def myFunc(s):
- words = s.split(“ ”) #分词，以空格分开，将所有单词组成一个 list 返回
- return len(words) #返回 list 的长度，即词数
- sc= SparkContext(...)
- sc.textFile(“file.txt”).map(myFunc)

值得指出的是，也可以传递类实例中方法的引用（与单例对象相反），这种传递方法会将整个对象传递过去。比如，考虑一下代码：

```
class MyClass(object):
    def func(self, s):
        return s
    def doStuff(self, rdd):
        return rdd.map(self.func)
```

在这里，如果我们创建了一个新的 MyClass 对象，然后对它调用 doStuff 方法，map 会用到这个对象中 func 方法的引用，所以整个对象都需要传递到集群中。

还有另一种相似的写法，访问外层对象的数据域会传递整个对象的引用：

```
class MyClass(object):
    def __init__(self):
        self.field = “Hello”
    def doStuff(self, rdd):
        return rdd.map( lambda s: self.field + s )
```

此类问题最简单的避免方法就是，使用一个本地变量缓存一份这个数据域的拷贝，直接访问这个数据域：

```
def doStuff(self, rdd):
    field = self.field
    return rdd.map(lambda s: field + s)
```

5.7、使用键值对

虽然大部分 Spark 的 RDD 操作都支持所有种类的对象，但是有少部分特殊的操作只能作用于键值对类型的 RDD。这类操作中最常见的就是分布的 suhffle 操作，比如将元素通过键来分组或聚集计算。

在 Python 中，这类操作一般都会使用 Python 内建的元组（tuple）类型，比如（1,2）。它们会先简单地创建类似这样的元组，让那后调用你想要的操作。

比如，以下代码对键值对调用了 reduceByKey 操作，来统计每一文本行在文本文件中出现的次数：

```
lines = sc.textFile(“data.txt”)
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```

我们还可以使用 counts.sortByKey()，比如，当我们想将这些键值对按照字母表顺序排序，然后调用 counts.collect()方法来将结果以对象列表的形式返回。

5.8、转化操作

下面的表格列出了 Spark 支持的常用转化操作。

转化操作	作用
map(func)	返回一个新的分布数据集，由原数据集元素经 func 处理后的结果组成
filter(func)	返回一个新的数据集，由传给 func 返回 True 的原数据集元素组成，即过滤器
flatMap(func)	与 map 类似，但是每个传入元素可能有 0 或多个返回值，func 可以返回一个序列而不是一个值
mapParitions(func)	类似 map，但是 RDD 的每个分片都会分开独立运行，所以 func 的参数和返回值都必须是迭代器
mapParitionsWithIndex(func)	类似 mapParitions，但是 func 有两个参数，第一个是分片的序号，第二个是迭代器。返回值还是迭代器。
sample(withReplacement,fraction,seed)	使用提供的随机数种子取样，然后替换或不替换
union(otherDataset)	返回新的数据集，包括原数据集和参数数据集的所有元素
intersection(otherDataset)	返回新数据集，是两个集的交集
distinct([numTasks])	返回新的集，包括原集中的不重复元素
groupByKey([numTasks])	当用于键值对 RDD 时，返回（键，值迭代器）对的数据集
aggregateByKey(zeroValue)(seqOp,combOp,[numTasks])	用于键值对 RDD 时，返回（K，U）对集，对每一个 Key 的 value 进行聚集计算
sortByKey([ascending],[numTasks])	用于键值对 RDD 时，返回 RDD 按键的顺序排序，升降序由第一个参数决定
join(otherDataset,[numTasks])	用于键值对（K，V）和（K，W）RDD 时返回（K，（V，W））对 RDD

cogroup(otherDataset,[numTasks])	用于两个键值对 RDD 时，返回（K，（V 迭代器，W 迭代器））RDD
cartesian(otherDataset)	用于 T 和 U 类型 RDD 时，返回（T，U）对类型键值对 RDD
pipe(command,[envVars])	通过 shell 命令管道处理每个 RDD 分片
coalesce(numPartitions)	把 RDD 的分片数量降低到参数大小
repartition(numPartitions)	重新打乱 RDD 中元素顺序并重新分片，数量由参数决定
repartitionAndSortWithinPartitions(partitioner)	按照参数给定的分片器重新分片，同时每个分片内部按照键排序

5.9、启动操作

下面的表格列出了 Spark 支持的部分常用启动操作。

启动操作	作用
reduce(func)	使用 func 进行聚集计算，func 的参数是两个，返回值一个，两次 func 运行应当是完全解耦的，这样才能正确地并行运算
collect()	向驱动程序返回数据集的元素组成的数组
count()	返回数据集元素的数量
first()	返回数据集的第一个元素
take(n)	返回前 n 个元素组成的数组
takeSample(withReplacement,num,[seed])	返回一个由原数据集中任意 num 个元素的数组，并且替换之
takeOrder(n,[ordering])	返回排序后的前 n 个元素
saveAsTextFile(path)	将数据集的元素写成文本文件
saveAsSequenceFile(path)	将数据集的元素写成序列文件，这个 API 只能用与 Java 和 Scala 程序

saveAsObjectFile(path)	将数据集的元素使用 Java 的序列化特性写到文件中，这个 API 只能用与 Java 和 Scala 程序
countByCount()	只能用于键值对 RDD，返回一个(K,int)hashmap，返回每个 key 的出现次数
foreach(func)	对数据集的每个元素执行 func，通常用于完成一些带有副作用的函数，比如更新累加器（见下文）或与外部存储交互等

5.10、RDD 持久化

5.10.1、RDD 持久化

Spark 的一个重要功能就是将数据集**持久化**（或**缓存**）到内存中以便在多个操作中重复使用。当我们持久化一个 RDD 时，每一个节点将这个 RDD 的每一个分片计算并保存到内存中以便在下次对这个数据集（或者这个数据集衍生的数据集）的计算中可以复用。这使得接下来的计算过程速度能够加快（经常能加快超过十倍的速度）。缓存是加快迭代算法和快速交互过程速度的关键工具。

你可以通过调用 persist 或 cache 方法来标记一个想要持久化的 RDD。在第一次被计算产生之后，它就会始终停留在节点的内存中。Spark 的缓存是具有容错性的——如果 RDD 的任意一个分片丢失了，Spark 就会依照这个 RDD 产生的转化过程自动重算一遍。

另外，每一个持久化的 RDD 都有一个可变的**存储级别**，这个级别使得用户可以改变 RDD 持久化的储存位置。比如，你可以将数据集持久化到硬盘上，也可以将它以序列化的 Java 对象形式（节省空间）持久化到内存中，还可以将这个数据集在节点之间复制，或者使用 Tachyon 将它储存到堆外。这些存储级别都是通过向 persist()传递一个 StorageLevel 对象（Scala, Java, Python）来设置的。

注意：在 Python 中，储存的对象永远是通过 Pickle 库序列化过的，所以设不设置序列化级别不会产生影响。

Spark 还会在 shuffle 操作（比如 reduceByKey）中自动储存中间数据，即使用户没有调用 persist。这是为了防止在 shuffle 过程中某个节点出错而导致的全盘重算。不过如果用户打算复用某些结果 RDD，我们仍然建议用户对结果 RDD 手动调用 persist，而不是依赖自动持久化机制。

5.10.2、应该选择哪个存储级别？

Spark 的存储级别是为了提供内存使用与 CPU 效率之间的不同取舍平衡程度。我们建议用户通过考虑以下流程来选择合适的存储级别：

（1）如果你的 RDD 很适合默认的级别（MEMORY_ONLY），那么就使用默认级别吧。这是 CPU 最高效运行的选择，能够让 RDD 上的操作以最快速度运行。

（2）否则，试试 MEMORY_ONLY_SER 选项并且选择一个块的序列化库来使对象的空间利用率更高，同时尽量保证访问速度足够快。

（3）不要往硬盘上持久化，除非重算数据集的过程代价确实很昂贵，或者这个过程过滤了巨量的数据。否则，重新计算分片有可能跟读硬盘速度一样快。

（4）如果你希望快速的错误恢复（比如用 Spark 来处理 web 应用的请求），使用复制级别。所有的存储级别都提供了重算丢失数据的完整容错机制，但是复制一份副本能省去等待重算的时间。

（5）在大内存或多应用的环境中，处于实验中的 OFF_HEAP 模式有诸多优点：

（5.1）这个模式允许多个执行者共享 Tachyon 中的同一个内存池

（5.2）这个模式显著降低了垃圾回收的花销。

（5.3）在某一个执行者个体崩溃之后缓存的数据不会丢失。

5.10.3、删除数据

Spark 会自动监视每个节点的缓存使用同时使用 LRU 算法丢弃旧数据分片。如果你想手动删除某个 RDD 而不是等待它被自动删除，调用 RDD.unpersist()方法。

5.11、共享变量

通常情况下，当一个函数传递给一个在远程集群节点上运行的 Spark 操作（比如 map 和 reduce）时，Spark 会对涉及到的变量的所有副本执行这个函数。这些变量会被复制到每个机器上，而且这个过程不会被反馈给驱动程序。通常情况下，在任务之间读写共享变量是很低效的。但是，Spark 仍然提供了有限的两种共享变量类型用于常见的使用场景：广播变量和累加器。

5.11.1、广播变量

广播变量允许程序员在每台机器上保持一个只读变量的缓存而不是将一个变量的拷贝传递给各个任务。它们可以被使用，比如，给每一个节点传递一份大输入数据集的拷贝是很低效的。Spark 试图使用高效的广播算法来分布广播变量，以此来降低通信花销。

可以通过 SparkContext.broadcast(v)来从变量 v 创建一个广播变量。这个广播变量是 v 的一个包装，同时它的值可以通过调用 value 方法来获得。以下的代码展示了这一点：

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

>>> <pyspark.broadcast.Broadcast opject at 0x102789f10>

- broadcastVar.value

>>> [1, 2, 3]

在广播变量被创建之后，在所有函数中都应当使用它来代替原来的变量 v，这样就可以保证 v 在节点之间只被传递一次。另外，v 变量在被广播之后不应该再被修改了，这样可以确保每一个节点上储存的广播变量的一致性（如果这个变量后来又被传输给一个新的节点）。

5.11.2、累加器

累加器是在一个相关过程中只能被”累加“的变量，对这个变量的操作可以有效地被并行化。它们可以被用于实现计数器（就像在 MapReduce 过程中）或求和运算。Spark 原生支持对数字类型的累加器，程序员也可以为其他新的类型添加支持。累加器被以一个名字创建之后，会在 Spark 的 UI 中显示出来。这有助于了解计算的累计过程（注意：目前 Python 中不支持这个特性）。

可以通过 SparkContext.accumulator(v)来从变量 v 创建一个累加器。在集群中运行的任务随后可以使用 add 方法或+=操作符（在 Scala 和 Python 中）来向这个累加器中累加值。但是，他们不能读取累加器中的值。只有驱动程序可以读取累加器中的值，通过累加器的 value 方法。

以下的代码展示了向一个累加器中累加数组元素的过程：

- 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

这段代码用了累加器对 int 类型的内建支持，程序员可以通过继承 AccumulatorParam 类来创建自己想要的类型支持。AccumultorParam 的接口提供了来嗯个方法：zero’用于为你的数据类型提供零值；'addInPlace'用于计算两个值的和。比如，假设我们有一个 Vector‘类表示数学中的向量，我们可以这样写：

- class VectorAccumultorParam(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())

累加器的更新操作只会被运行一次，Spark 提供了保证，每个任务中对累加器的更新操作都只会被运行一次。比如，重启一个任务不会再次更新累加器。在转化过程中，用户应该留意每个任务的更新操作在任务或作业重新运算时是否被执行了超过一次。

累加器不会改变 Spark 的惰性求值模型。如果累加器在对 RDD 的操作中被更新了，它们的值只会在启动操作中作为 RDD 计算过程中的一部分被更新。所以，在一个懒惰的转化操作中调用累加器的更新，并没法保证会被即使运行。下面的代码段展示了这一点：

- accum = sc.accumulator(0)
- data.map(lambda x => accum.add(x); f(x))
- # Here, accum is still 0 because no actions have cause the 'map' to be computed.

5.12、在集群上部署

这个应用提交指南描述了一个应用被提交到集群上的过程。简而言之，只要你把你的应用打成了 JAR 包（Java/Scala 应用）或.py 文件的集合或.zip 压缩包（Python 应用），bin/spark-submit 脚本会将应用提交到任意支持的集群管理器上。

5.13、单元测试

Spark 对单元测试是友好的，可以与任何流行的单元测试框架相容。你只需要在测试中创建一个 SparkContext，并如前文所述将 master 的 URL 设为 local，执行你的程序，最后调用 SparkContext.stop()来终止运行。请确保你在 finally 块或测试框架的 tearDown 方法中终止了上下文，因为 Spark 不支持两个上下文在一个程序中同时运行。

6、还有什么要做的

你可以在 Spark 的网站上看到更多的 Spark 样例程序。另外，在 examples 目录下还有许多样例代码（Scala, Java, Python）。你可以通过将类名称传给 Spark 的 bin/run-example 脚本来运行 Java 和 Scala 语言样例，举例说明：

- ./bin/run-example SparkPi

对于 Python 例子，使用 spark-submit 脚本代替：

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

为了给你优化代码提供帮助，配置指南和调优指南提供了关于最佳实践的一些信息。确保你的数据储存以高效的格式储存在内存中，这很重要。为了给你部署应用提供帮助，集群模式概览描述了许多呢荣，包括分布式操作和支持的集群管理器。

最后，完整的 API 文档在这里。Python 版本

Spark 编程指南__Python 版_2016/4/15

附：

0、关于 hadoop、spark 的启动

通常，当我们需要使用 hadoop 的 hdfs、yarn 和 spark 时，需要进行以下启动：首先启动 hadoop 的 all，然后启动 spark 的 all，具体命令如下：

进入 hadoop 安装路径

- cd /usr/local/hadoop

启动 hadoop 的 all

- sbin/start-all.sh

退出 hadoop 安装路径

- cd

进入 spark 安装路径

- cd /usr/local/spark/spark-1.6.0-bin-hadoop2.6

启动 spark 的 all

- sbin/start-all.sh

查看启动情况，启动成功的话：

在 Master 上会有如下启动：SecondaryNameNode、Jps、NodeManager、NameNode、Master、DataNode、ResourceManager、Worker；

在 Slave 上会有如下启动：Jps、Worker、DataNode、NodeManager

- jps

查看 dfs 情况，即 datanode 的存活情况（live or dead），要先进入 hadoop 安装路径

- bin/hdfs dfsadmin -report

1、命令行参数获取

- import sys
- sys.argv[1]

注：sys.argv 用来获取命令行的参数。

sys.argv[0]表示代码本身文件路径。

如：

- python test.py -test

  那么 sys.argv[0]就是 test.py，sys.argv[1]是-test。

2、本地路径写法

- sc.textFile("file:///usr/local/spark/spark-1.6.0-bin-hadoop2.6/mytest/mydata.txt")

  注：file:///一定是三个/，使用本地路径时要将 mydata.txt 在集群上放在统一的路径下。

  读取本地文件时，读取的是 Master 上的本地文件，要注意。

3、代码里有中文时（即使只是注释），在最开始加上这句

- #-*- coding:utf-8 -*-

4、exit(0)：无错误退出
exit(1)：有错误退出
退出代码是告诉解释器的（或操作系统）

5、hdfs 操作

进入 hadoop 安装路径

- cd /usr/local/hadoop

创建一个 mytest 目录，当仅写/test 时，默认建于 hdfs://Master:9000/test/；所以若想建在默认路径下，需要写全 hdfs://Master:9000/user/nathychen/test/

- bin/hdfs dfs -mkdir /mytest

上传本地文件 test.txt 到 mytest 目录，当简写/mytest/时，默认表示为 hdfs://Master:9000/user/nathychen/mytest/

- bin/hdfs dfs -put test.txt /mytest/

上传本地目录/mylocaltest 到 mytest 目录

- bin/hdfs dfs -put mylocaltest/ /mytest/

查看/mytest 目录

- bin/hdfs dfs -ls /mytest

创建一个空文件 nulltest.txt

- bin/hdfs dfs -touchz nulltest.txt

删除文件 nulltest.txt

- bin/hdfs dfs -rm nulltest.txt

删除目录/mytest

- bin/hdfs dfs -rm -r /mytest

重命名，将 mytest1 重命名为 mytest2，文件或目录均可

- bin/hdfs dfs -mv /mytest1 /mytest2

查看文件

- bin/hdfs dfs -cat test.txt

将指定目录下的所有内容融合（merge）成一个文件，并下载到本地，默认路径为 hadoop/安装路径

- bin/hdfs dfs -getmerge /mytest mylocaltest.txt

查看文件和目录的大小

- bin/hdfs dfs -du test.txt

将/mytest 目录拷贝到本地/mylocaldir

- bin/hdfs dfs -copyToLocal /mytest /mylocaldir

查看 dfs 的情况

- `bin/hdfs dfsadmin -report`

  查看正在跑的 `java` 程序

- `jps`