亚太研究院

Spark 官方文档翻译

Spark 编程指南 (V1.2.0)

翻译者 Harli

Spark 官方文档翻译团成员

前 言

伴随着大数据相关技术和产业的逐步成熟，继 Hadoop 之后，Spark 技术以集大成的无可比拟的优势，发展迅速，将成为替代 Hadoop 的下一代云计算、大数据核心技术。

Spark 是当今大数据领域最活跃最热门的高效大数据通用计算平台，基于 RDD，Spark 成功的构建起了一体化、多元化的大数据处理体系，在“One Stack to rule them all”思想的引领下，Spark 成功的使用 Spark SQL、Spark Streaming、MLLib、GraphX 近乎完美的解决了大数据中 Batch Processing、Streaming Processing、Ad-hoc Query 等三大核心问题，更为美妙的是在 Spark 中 Spark SQL、Spark Streaming、MLLib、GraphX 四大子框架和库之间可以无缝的共享数据和操作，这是当今任何大数据平台都无可匹敌的优势。

在实际的生产环境中，世界上已经出现很多一千个以上节点的 Spark 集群，以 eBay 为例，eBay 的 Spark 集群节点已经超过 2000 个，Yahoo！等公司也在大规模的使用 Spark，国内的淘宝、腾讯、百度、网易、京东、华为、大众点评、优酷土豆等也在生产环境下深度使用 Spark。2014 Spark Summit 上的信息，Spark 已经获得世界 20 家顶级公司的支持，这些公司中包括 Intel、IBM 等，同时更重要的是包括了最大的四个 Hadoop 发行商，都提供了对 Spark 非常强有力的支持。

与 Spark 火爆程度形成鲜明对比的是 Spark 人才的严重稀缺，这一情况在中国尤其严重，这种人才的稀缺，一方面是由于 Spark 技术在 2013、2014 年才在国内的一些大型企业里面被逐步应用，另一方面是由于匮乏 Spark 相关的中文资料和系统化的培训。为此，Spark 亚太研究院和 51CTO 联合推出了“Spark 亚太研究院决胜大数据时代 100 期公益大讲堂”，来推动 Spark 技术在国内的普及及落地。

具体视频信息请参考 http://edu.51cto.com/course/course_id-1659.html

与此同时，为了向 Spark 学习者提供更为丰富的学习资料，Spark 亚太研究院去年 8 月发起并号召，结合网络社区的力量构建了 Spark 中文文档专家翻译团队，翻译了 Spark 中文文档 V1.1.0 版本。2014 年 12 月，Spark 官方团队发布了 Spark 1.2.0 版本，为了让学习者了解到最新的内容，Spark 中文文档专家翻译团队又对 Spark 1.2.0 版本进行了部分更新，在此，我谨代表 Spark 亚太研究院及广大 Spark 学习爱好者向专家翻译团队所有成员热情而专业的工作致以深刻的敬意！

当然，作为相对系统的 Spark 中文文档，不足之处在所难免，大家有任何建议或者意见都可以发邮件到 marketing@sparkinchina.com；同时如果您想加入 Spark 中文文档翻译团队，也请发邮件到 marketing@sparkinchina.com 进行申请；Spark 中文

文档的翻译是一个持续更新的、不断版本迭代的过程，我们会尽全力给大家提供更高质量的 Spark 中文文档翻译。

最后，也是最重要的，请允许我荣幸的介绍一下我们的 Spark 中文文档 1.2.0 版本翻译的专家团队成员，他们分别是（排名不分先后）：

- 傅智勇, 《快速开始(v1.2.0)》
- 王宇舟,《Spark 机器学习库 (v1.2.0)》
- 武扬，《在 Yarn 上运行 Spark (v1.2.0)》《Spark 调优(v1.2.0)》
- 徐骄，《Spark 配置(v1.2.0)》《Spark 作业调度(v1.2.0)》
- 蔡立宇，《Bagel 编程指南(v1.2.0)》
- harli，《Spark 编程指南 (v1.2.0)》
- 韩保礼，《Spark SQL 编程指南(v1.2.0)》
- 李丹丹，《文档首页(v1.2.0)》
- 李军，《Spark 实时流处理编程指南(v1.2.0)》
- 俞杭军，《使用 Maven 编译 Spark(v1.2.0)》
- 王之，《给 Spark 提交代码(v1.2.0)》
- Ernest ,《集群模式概览(v1.2.0)》《监控与相关工具(v1.2.0)》《提交应用程序(v1.2.0)》

Life is short, You need Spark!

Spark 亚太研究院院长 王家林
2015 年 2 月

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂

简 介

作为下一代云计算的核心技术 ,Spark 性能超 Hadoop 百倍 ,算法实现仅有其 1/10 或 1/100,是可以革命 Hadoop 的目前唯一替代者，能够做 Hadoop 做的一切事情，同时速度比 Hadoop 快了 100 倍以上。目前 Spark 已经构建了自己的整个大数据处理生态系统，国外一些大型互联网公司已经部署了 Spark。甚至连 Hadoop 的早期主要贡献者 Yahoo 现在也在多个项目中部署使用 Spark ;国内的淘宝、优酷土豆、网易、Baidu、腾讯、皮皮网等已经使用 Spark 技术用于自己的商业生产系统中，国内外的应用开始越来越广泛。Spark 正在逐渐走向成熟，并在这个领域扮演更加重要的角色, 刚刚结束的 2014 Spark Summit 上的信息，Spark 已经获得世界 20 家顶级公司的支持，这些公司中包括 Intel、IBM 等，同时更重要的是包括了最大的四个 Hadoop 发行商都提供了对非常强有力的支持 Spark 的支持.

鉴于 Spark 的巨大价值和潜力，同时由于国内极度缺乏 Spark 人才，Spark 亚太研究院在完成了对 Spark 源码的彻底研究的同时，不断在实际环境中使用 Spark 的各种特性的基础之上，推出了 Spark 亚太研究院决胜大数据时代 100 期公益大讲堂，希望能够帮助大家了解 Spark 的技术。同时，对 Spark 人才培养有近一步需求的企业和个人，我们将以公开课和企业内训的方式，来帮助大家进行 Spark 技能的提升。同样，我们也为企业提供一体化的顾问式服务及 Spark 一站式项目解决方案和实施方案。

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂是国内第一个 Spark 课程免费线上讲座，每周一期，从 7 月份起，每周四晚 20:00-21:30，与大家不见不散！老师将就 Spark 内核剖析、源码解读、性能优化及商业实战案例等精彩内容与大家分享，干货不容错过！

时间：从 7 月份起，每周一期，每周四晚 20:00-21:30
形式：腾讯课堂在线直播
学习条件：对云计算大数据感兴趣的技术人员
课程学习地址：http://edu.51cto.com/course/course_id-1659.html

Spark 编程指南（V1.2.0）

（翻译者：Harli）

Spark Programming Guide，原文档链接：

http://spark.apache.org/docs/latest/programming-guide.html

目录

第 1 章 概述

总的来说，每一个 Spark 应用程序都包含一个*驱动程序(driver program)*，它会运行用户的 main 函数，并在集群上执行各种*并行操作(parallel operations)*。Spark 提供的最主要的抽象概念是*弹性分布式数据集(resilient distributed dataset)* (RDD)， 它是一个元素集合，被分区地分布到集群的不同节点上，可以被并行操作。RDDs 可以从 HDFS(或者任意其他支持 Hadoop 的文件系统) 上的一个文件开始创建，或者通过转换驱动程序(driver program)中已经存在的 Scala 集合得到。 用户也可以让 Spark 将一个 RDD *持久化(persist)*到内存中，使其能在并行操作中被有效地重复使用。最后，RDD 能自动从节点故障中恢复。

Spark 的第二个抽象概念是*共享变量(shared variables)*，它可以在并行操作中使用。在默认情况下，当 Spark 将一个函数以任务集(a set of tasks)的形式在不同节点上并行运行时， 会将该函数所使用的每个变量的拷贝传递给每一个任务（task）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进行共享。 Spark 支持两种类型的共享变量： *广播变量(broadcast variables)*，它可以在所有节点的内存中缓存一个值；*累加器(accumulators)*：只能用于做加法的变量，例如计数器(counters)或求和器(sums)。

本指南将通过 Spark 支持的各种语言来展示这些特性。你可以通过运行 Spark 交互式 shell 的方式来轻松掌握 - 运行 bin/spark-shell 启动 Scala shell， 或 bin/pyspark 启动 Python shell。

第 2 章 接入 Spark

- Scala

Spark 1.2.0 需要搭配使用 Scala 2.10。编写 Scala 应用程序，你需要使用兼容的 Scala 版本(比如 2.10.X)。

编写一个 Scala 应用程序时，你需要在 Maven 中添加 Spark 依赖。Spark 依赖可以从 Maven 中心库获取：

```
groupId = org.apache.spark

artifactId = spark-core_2.10

version = 1.2.0
```

另外，如果你想访问一个 HDFS 集群，需要根据你的 HDFS 版本，添加 hadoop-client 依赖。 一些常见的 HDFS 版本标签已经在 第三方发行版(third party distributions) 页面中列出。

```
groupId = org.apache.hadoop

artifactId = hadoop-client

version = <your-hdfs-version>
```

最后，你需要将一些 Spark 的类和隐式转换导入到你的程序中。添加下列语句：

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import org.apache.spark.SparkConf
```

- Java

Spark 1.2.0 工作在 Java 6 或更高版本之上。如果你使用 Java 8，Spark 支持用 lambda 表达式 简化函数编写，除此之外你还可以使用 org.apache.spark.api.java.function 包中的类。

用 Java 编写 Spark 应用时，你需要添加 Spark 依赖，Spark 依赖可以从 Maven 中心库获取：

```
groupId = org.apache.spark

artifactId = spark-core_2.10

version = 1.2.0
```

另外，如果你想访问一个 HDFS 集群，需要根据你的 HDFS 版本，添加 hadoop-client 依赖。 一些常见的 HDFS 版本标签已经在 第三方发行版(third party distributions) 页面中列出。

```
groupId = org.apache.hadoop

artifactId = hadoop-client

version = <your-hdfs-version>
```

最后，你需要将一些 Spark 的类导入到你的程序中。添加下列语句：

```
import org.apache.spark.api.java.JavaSparkContext

import org.apache.spark.api.java.JavaRDD
```

```
import org.apache.spark.SparkConf
```

- Python

Spark 1.2.0 搭配使用 Python 2.6 或更高版本 (但 Python 3 除外)。它使用了标准的 CPython 解释器， 因此，可以使用像 NumPy 之类的 C 语言类库。

如果要在 Python 中运行 Spark 应用程序，可以使用位于 Spark 目录下的 bin/spark-submit 脚本。 该脚本将会加载 Spark 的 Java/Scala 类库，并允许你将应用程序提交到集群中。 你也可以使用 bin/pyspark 来启动一个交互式 Python shell。

另外 ,如果你想访问一个 HDFS 数据 ,需要连接你的 HDFS 版本来构建一个 PySpark。一些常见的 HDFS 版本标签已经在 第三方发行版(third party distributions) 页面中列出。 预构建包(Prebuilt packages) 在包含常用 HDFS 版本的 Spark 主页中也可以获取。

最后，你需要将一些 Spark 的类导入到你的程序中。添加下列语句：

```
from pyspark import SparkContext, SparkConf
```

第 3 章 初始化 Spark

- Scala

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。而要创建一个 SparkContext 对象，你首先要创建一个 SparkConf 对象， 该对象包含了你的应用程序的信息。

在每个 JVM 中只能有一个 SparkContext 处于 active 状态。 在创建一个新的 SparkContext 之前，你必须先 stop() 在 active 状态下的 SparkContext。

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

new SparkContext(conf)
```

其中， appName 参数是你的应用程序的名字，会在集群的 Web UI 界面上显示。master 参数用于指定 Spark, Mesos 或 YARN 集群 URL, 或一个特殊的字符串 “local” ，来指定在本地(local)模式下运行。 实际上，当在一个集群上运行时，一般不会在程序中对 master 参数进行硬编码，而是在 使用 spark-submit 脚本启动应用程序 时

传入 master 参数。 然而，在本地测试或者单元测试时，你可以传入“local” ，在进程中运行 Spark。

- Java

Spark 程序需要做的第一件事情，就是创建一个 JavaSparkContext 对象，它将告诉 Spark 如何访问一个集群。而要创建一个 SparkContext 对象你首先要创建一个 SparkConf 对象， 该对象包含了你的应用程序的信息。

```
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);

JavaSparkContext sc = new JavaSparkContext(conf);
```

其中， appName 参数是你的应用程序的名字，会在集群的 Web UI 界面上显示。master 参数用于指定 Spark, Mesos 或 YARN 集群 URL, 或一个特殊的字符串“local” ，来指定在本地(local)模式下运行。 实际上，当在一个集群上运行时，一般不会在程序中对 master 参数进行硬编码，而是在 使用 spark-submit 脚本启动应用程序 时传入 master 参数。 然而，在本地测试或者单元测试时，你可以传入“local” ，在进程中运行 Spark。

- Python

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。而要创建一个 SparkContext 对象你首先要创建一个 SparkConf 对象， 该对象包含了你的应用程序的信息。

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

sc = SparkContext(conf=conf)
```

其中， appName 参数是你的应用程序的名字，会在集群的 Web UI 界面上显示。master 参数用于指定 Spark, Mesos 或 YARN 集群 URL, 或一个特殊的字符串“local” ，来指定在本地(local)模式下运行。 实际上，当在一个集群上运行时，一般不会在程序中对 master 参数进行硬编码，而是在 使用 spark-submit 脚本启动应用程序 时传入 master 参数。 然而，在本地测试或者单元测试时，你可以传入“local” ，在进程中运行 Spark。

3.1 使用 Shell 进行交互式编程

- Scala

在 Spark shell 中，一个特殊的解释器感知(interpreter-aware) 的 SparkContext 已经为你创建好了，变量名是 sc。 创建你自己的 SparkContext 是不会生效的。你可以使用 --master 选项来设置 SparkContext 所连接的 master，也可以使用逗号分隔的文件列表的形式， 通过 --jars 选项将 JARs 添加到 classpath 中。例如，在四核的 CPU 上运行 bin/spark-shell ，使用：

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

或者，另外在 classpath 中添加 code.jar，使用：

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

运行 spark-shell --help 可获取命令选项的完整列表。 spark-shell 脚本运行的幕后，是调用了更通用的 spark-submit 脚本.

- Python

在 PySpark shell 中，一个特殊的解释器感知(interpreter-aware) 的 SparkContext 已经为你创建好了，变量名是 sc。 创建你自己的 SparkContext 是不会生效的。你可以使用 --master 选项来设置 SparkContext 所连接的 master，也可以使用逗号分隔的文件列表的形式， 通过 --py-files 选项将 Python 的.zip，.egg 或 .py 文件添加到运行路径下，例如，在四核的 CPU 上运行 bin/pyspark， 使用：

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

或者，另外将 code.py 添加到搜索路径下 (以便后续可以导入代码(import code))，使用：

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

运行 pyspark --help 可获取命令选项的完整列表。 pyspark 脚本的运行实际上是调用了更通用的 spark-submit 脚本。

在 IPython 中启动 PySpark shell 也是可以的，这是一个增强版 Python 解释器。PySpark 工作在 IPython 1.0.0 及更高版本下. 使用 IPython 时， 需要在运行 bin/pyspark 的时候，设置 PYSPARK_DRIVER_PYTHON 变量为 ipython：

```
$ PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark
```

你可以通过设置 PYSPARK_DRIVER_PYTHON_OPTS 环境变量来定制 ipython 命令 。 例如，启动支持 PyLab plot 的 IPython Notebook：

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

第 4 章 弹性分布式数据集（RDDs)

Spark 的核心概念是 *弹性分布式数据集(resilient distributed dataset)*，这是一个有容错机制并可以被并行操作的元素集合。目前有两种方式可以创建 RDDs：*并行化 (parallelizing)* 一个已经存在于驱动程序（driver program）中的集合(collection)， 或者引用外部存储系统上的一个数据集，比如一个共享文件系统，HDFS，HBase，或者任何提供了 Hadoop InputFormat 的数据源。

4.1 并行集合(Parallelized collections)

- Scala

并行集合，是通过对存在于驱动程序中的集合(Collection) (一个 Scala Seq)， 调用 SparkContext’ s parallelize 方法来构建的。 构建时会拷贝集合中的元素，创建一个可以被并行操作的分布式数据集。例如，这里演示了如何创建一个包含数字 1 到 5 的并行集合：

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

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

一旦创建了分布式数据集(distData)，就可以对其执行并行操作。例如，我们可以调用 distData.reduce((a, b) => a + b) 来累加数组的元素。后续我们会进一步地描述对分布式数据集的操作。

并行集合的一个重要参数是分片数(the number of *slices*)，表示数据集切分的份数。Spark 将在集群上为每一个分片数据起一个任务。 典型情况下，你希望集群的每个 CPU 分布 2-4 个分片（slices）。 通常，Spark 会尝试基于集群状况自动设置分片数。然而，你也可以进行手动设置，通过将分片数作为第二个参数传递给 parallelize 方法来实现。(例如：sc.parallelize(data, 10))。注意： 在一些代码中会使用术语 slices （分区的一个同义词）来提供向后兼容性。

- Java

并行集合，是通过对存在于驱动程序中的集合(Collection)， 调用 JavaSparkContext’ s parallelize 方法来构建的。 构建时会拷贝集合中的元素，创建一个可以被并行操作的分布式数据集。 例如，这里演示了如何创建一个包含数字 1 到 5 的并行集合：

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);

JavaRDD<Integer> distData = sc.parallelize(data);
```

一旦创建了分布式数据集(distData)，就可以对其执行并行操作。例如，我们可以调用 distData.reduce((a, b) -> a + b) 来累加列表的元素。后续我们会进一步地描述对分布式数据集的操作。

注意： *在本指南中，我们会经常使用简洁的 Java 8 的 lambda 语法来指明 Java 函数， 而在 Java 的旧版本中，你可以实现 org.apache.spark.api.java.function 包中的接口。下面我们将在 把函数传递到 Spark 中描述更多的细节.*

并行集合的一个重要参数是分片数(the number of *slices*)，表示数据集切分的份数。Spark 将在集群上为每一个分片数据起一个任务。 典型情况下，你希望集群的每个 CPU 分布 2-4 个分片（slices）。 通常，Spark 会尝试基于集群状况自动设置分片数。然而，你也可以进行手动设置 ,通过将分片数作为第二个参数传递给 parallelize 方法来实现。(例如：sc.parallelize(data, 10))。 注意： 在一些代码中会使用术语 slices （分区的一个同义词）来提供向后兼容性。

- Python

并行集合，是通过对存在于驱动程序中的迭代器(iterable)或集合(collection)，调用 SparkContext’ s parallelize 方法来构建的。 构建时会拷贝迭代器或集合中的元素，创建一个可以被并行操作的分布式数据集。例如，这里演示了如何创建一个包含数字 1 到 5 的并行集合：

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

distData = sc.parallelize(data)
```

一旦创建了分布式数据集(distData)，就可以对其执行并行操作。例如，我们可以调用 distData.reduce(lambda a, b: a + b) 来 累加列表的元素。后续我们会进一步地描述对分布式数据集的操作。

并行集合的一个重要参数是分区数(the number of *partitions*)，表示数据集切分的份数。Spark 将在集群上为每一个分区数据起一个任务。 典型情况下，你希望集群的每个 CPU 分布 2-4 个分区（partitions）。 通常，Spark 会尝试基于集群状况自动设置分区数。然而，你也可以进行手动设置，通过将分片数作为第二个参数传递给 parallelize 方法来实现。（例如：sc.parallelize(data, 10))。注意： 在一些代码中会使用术语 slices （分区的一个同义词）来提供向后兼容性。

4.2 外部数据集(External Datasets)

- Scala

Spark 可以从 Hadoop 支持的任何存储源中构建出分布式数据集，包括你的本地文件系统，HDFS，Cassandre，HBase， Amazon S3，等。Spark 支持 text files，SequenceFiles，以及其他任何一种 Hadoop InputFormat.

Text file RDDs 的创建可以使用 SparkContext’ s textFile 方法。该方法接受一个文件的 URI 地址（或者是机器上的一个本地路径，或者是一个 hdfs://， s3n://，等 URI）作为参数，并读取文件的每一行数据，放入集合中。下面是一个调用例子：

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

distFile: RDD[String] = MappedRDD@1d4cee08
```

一旦创建完成，就可以在 distFile 上执行数据集操作。例如，要想对所有行的长度进行求和，我们可以通过如下的 map 和 reduce 操作来完成： distFile.map(s => s.length).reduce((a, b) => a + b)。

Spark 读文件时的一些注意事项：

- 如果文件使用本地文件系统上的路径，那么该文件必须在工作节点的相同路径下也可以访问。可以将文件拷贝到所有的 workers 节点上 ,或者使用 network-mounted 共享文件系统。

- Spark 的所有基于文件的输入方法，包括 textFile，支持在目录上运行，压缩文件和通配符。例如 ,你可以使用 textFile("/my/directory")，textFile("/my/directory/*.txt")，和 textFile("/my/directory/*.gz").

- textFile 方法也带有可选的第二个参数，用于控制文件的分区数。默认情况下，Spark 会为文件的每一个 block(在 HDFS 中， 块的默认大小为 64MB) 创建一个分区，但是你也可以通过传入更大的值，来设置更高的分区数。注意，你不能得到比分块数更小的分区数。

除了 text files，Spark 的 Scala API 也支持其他几种数据格式：

- SparkContext.wholeTextFiles 可以让你读取包含多个小 text files 的目录，并且每个文件对应返回一个(filename, content) 对。而对应的 textFile 方法，文件的每一行对应返回一条记录(record)。

- 对于 SequenceFiles，使用 SparkContext’s sequenceFile[K, V] 方法，其中 K 和 V 分别对应文件中 key 和 values 的类型。这些类型必须是 Hadoop’s Writable 接口的子类，如 IntWritable 和 Text. 另外，Spark 允许你使用一些常见 Writables 的原生类型(native types)；例如， sequenceFile[Int, String] 会自动的转换为类型 IntWritables 和 Texts.

- 对于其他的 Hadoop InputFormats，你可以使用 SparkContext.hadoopRDD 方法，它可以接受 一个任意类型的 JobConf 和输入格式类，key 类 和 value 类。像 Hadoop Job 设置输入源那样去设置这些参数即可。 对基于"新"的 MapReduce API (org.apache.hadoop.mapreduce)的 InputFormats，你也可以使用 SparkContext.newHadoopRDD。

- RDD.saveAsObjectFile 和 SparkContext.objectFile 支持由序列化的 Java 对象组成的简单格式来保存 RDD。虽然这不是一种像 Avro 那样有效的序列化格式，但是它提供了一种可以存储任何 RDD 的简单方式。

- Java

Spark 可以从 Hadoop 支持的任何存储源中构建出分布式数据集，包括你的本地文件系统，HDFS，Cassandre，HBase， Amazon S3，等。Spark 支持 text files，SequenceFiles，以及其他任何 Hadoop InputFormat.

Text file RDDs 可以使用 SparkContext’s textFile 方法. 该方法接受一个文件的 URI 地址(或者是机器上的一个本地路径 ,或者是一个 hdfs://, s3n://, 等 URI)作为参数，并读取文件的每一行数据，放入集合中。下面是一个调用例子：

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

一旦创建完成，就可以在 distFile 上执行数据集操作。例如，要想对所有行的长度进行求和，我们可以通过如下的 map 和 reduce 操作来完成： distFile.map(s -> s.length()).reduce((a, b) -> a + b)。

Spark 读文件时的一些注意事项：

- 如果文件使用本地文件系统上的路径，那么该文件必须在工作节点的相同路径下也可以访问。可以将文件拷贝到所有的 workers 节点上 ,或者使用 network-mounted 共享文件系统。
- Spark 的所有基于文件的输入方法，包括 textFile，支持在目录上运行，压缩文件和通配符。例如 ,你可以使用 textFile("/my/directory"), textFile("/my/directory/*.txt")，和 textFile("/my/directory/*.gz").
- textFile 方法也带有可选的第二个参数，用于控制文件的分区数。默认情况下，Spark 会为文件的每一个 block(在 HDFS 中， 块的默认大小为 64MB) 创建一个分区，但是你也可以通过传入更大的值，来设置更高的分区数。注意，你不能得到比分块数更小的分区数。

除了 text files，Spark 的 Java API 也支持其他几种数据格式：

- JavaSparkContext.wholeTextFiles 可以让你读取包含多个小 text files 的目录，并且每个文件对应返回一个(filename, content) 对。 而对应的 textFile 方法，文件的每一行对应返回一条记录(record)。
- 对于 SequenceFiles，使用 SparkContext’ s sequenceFile[K, V] 方法，其中 K 和 V 分别对应文件中 key 和 values 的类型。 这些类型必须是 Hadoop’ s Writable 接口的子类，如 IntWritable 和 Text.
- 对于其他的 Hadoop InputFormats，你可以使用 JavaSparkContext.hadoopRDD 方法 ,它可以接受一个任意类型的 JobConf 和输入格式类，key 类 和 value 类。像 Hadoop Job 设置输入源那样去设置这些参数即可。 对基于"新"的 MapReduce API (org.apache.hadoop.mapreduce)的 InputFormats，你也可以使用 JavaSparkContext.newHadoopRDD。
- JavaRDD.saveAsObjectFile 和 JavaSparkContext.objectFile 支持用序列化的 Java 对象组成的简单格式来保存 RDD。虽然这不是一种像 Avro 那样有效的序列化格式，但是它提供了一种可以存储任何 RDD 的简单方式。

- Python

PySpark 可以从 Hadoop 支持的任何存储源中构建出分布式数据集，包括你的本地文件系统，HDFS，Cassandre，HBase， Amazon S3，等。Spark 支持 text files，SequenceFiles，以及其他任何 Hadoop InputFormat.

Text file RDDs 可以使用 SparkContext’ s textFile 方法. 该方法接受一个文件的 URI 地址(或者是机器上的一个本地路径 ,或者是一个 hdfs://, s3n://, 等 URI)作为参数，并读取文件，得到行的集合。下面是一个调用例子：

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

一旦创建完成，就可以在 distFile 上执行数据集操作。例如，要想对所有行的长度进行求和，我们可以通过如下的 map 和 reduce 操作来完成：distFile.map(lambda s: len(s)).reduce(lambda a, b: a + b)。

Spark 读文件时的一些注意事项：

- 如果文件使用本地文件系统上的路径，那么该文件必须在工作节点的相同路径下也可以访问。可以将文件拷贝到所有的 workers 节点上 ,或者使用 network-mounted 共享文件系统。
- Spark 的所有基于文件的输入方法，包括 textFile，支持在目录上运行，压缩文件和通配符。例如 ,你可以使用 textFile("/my/directory")，textFile("/my/directory/*.txt")，和 textFile("/my/directory/*.gz")。
- textFile 方法也带有可选的第二个参数，用于控制文件的分区数。默认情况下，Spark 会为文件的每一个 block(在 HDFS 中， 块的默认大小为 64MB) 创建一个分区，但是你也可以通过传入更大的值，来设置更高的分区数。注意，你不能得到比分块数更小的分区数。

除了 text files，Spark 的 Python API 也支持其他几种数据格式：

- JavaSparkContext.wholeTextFiles 可以让你读取包含多个小 text files 的目录，并且每个文件对应返回一个(filename, content) 对。 而对应的 textFile 方法，文件的每一行对应返回一条记录(record)。
- RDD.saveAsPickleFile 和 SparkContext.pickleFile 支持由 pickled Python 对象组成的简单格式保存 RDD。使用批量的方式处理 pickle 模块的对象序列化，默认批处理大小为 10。
- SequenceFile 和 Hadoop 输入/输出的格式。
- 注意 此功能当前标识为试验性的(Experimental)， 是为高级用户而提供的。在将来的版本中 ,可能会因为支持基于 SparkSQL 的读写而被取代 ,在这种情况下， SparkSQL 是首选的方法。

Writable 支持(Writable Support)

PySpark 的 SequenceFile 支持加载 Java 中的键值(key-value)对 RDD，可以将 Writables 转换为基本的 Java 类型， 并且通过 Pyrolite ，在结果 Java 对象上执行 pickles 序列化操作。当将一个键值对的 RDD 保存为 SequenceFile 时， PySpark 会对其进行反操作。 它会 unpickles Python 的对象为 Java 对象，然后再将它们转换为 Writables。下表中的 Writables 会被自动地转换：

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

数组不支持开箱(out-of-the-box)处理。 当读或写数组时，用户需要指定自定义的 ArrayWritable 子类。 当写数组时，用户也需要指定自定义的转换器(converters)，将数组转换为自定义的 ArrayWritable 子类。 当读数组时，默认的转换器(converter)会将自定义的 ArrayWritable 子类 转换为 Java 的 Object[]，然后被 pickled 成 Python 的元组。如果要获取包含基本数据类型(primitive types)的数组，Python 的 array.array 的话， 用户需要为该数组指定自定义的转换器(converters)。

保存和加载 SequenceFiles

类似于 text files，SequenceFiles 可以被保存和加载到指定的路径下。可以指定 key 和 value 的类型，但对标准的 Writables 类型则不需要指定。

```
>>> 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')]
```

保存和加载其他的 Hadoop 输入/输出 格式(Input/Output Formats)

PySpark 也可以读任何 Hadoop InputFormat 或者写任何 Hadoop OutputFormat，包括“新”和“旧”两个 Hadoop MapReduce APIs。 如果需要的话，可以将传递进来的一个 Hadoop 配置当成 一个 Python 字典(Python dict)。这里有一个使用了 Elasticsearch ESInputFormat 的样例：

```
$ 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})
```

注意，如果这个 InputFormat 只是简单地依赖于 Hadoop 配置和/或输入路径，以及 key 和 value 的类型，它就可以很容易地根据上面的表格进行转换， 那么这种方法应该可以很好地处理这些情况。

如果你有一个定制序列化(serialized)的二进制数据(比如加载自 Cassandra/HBase 的数据)， 那么你首先要做的，是在 Scala/Java 侧将数据转换为可以用 Pyrolite 的 pickler 处理的东西。 Converter 特质(trait) 提供了这一转换功能。 简单地 extend 该特质，然后在 convert 方法中实现你自己的转换代码。记住要确保该类，以及访问你的 InputFormat 所需的依赖，都需要被打包到你的 Spark 作业(Job) 的 jar，并且包含在 PySpark 的类路径(classpath)中。

在 Python 样例 和 Converter 样例 上给出了带自定义转换器(converters)的 Cassandra / HBase 的 InputFormat 和 OutputFormat 的使用样例。

4.3 RDD 操作

RDDs 支持两种操作：*转换(transformations)*，可以从已有的数据集创建一个新的数据集；而 *动作(actions)*，在数据集上运行计算后，会向驱动程序返回一个值。 例如， map 就是一种转换，它将数据集每一个元素都传递给函数，并返回一个新的分布数据集来表示结果。 另一方面， reduce 是一种动作， 通过一些函数将所有的元素聚合(aggregates)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).

Spark 中的所有转换都是 *惰性的(lazy)*, 也就是说，它们并不会马上计算结果。相反的，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(driver program)的动作时，这些转换才会真正运行。 这种设计让 Spark 更加有效率的运行。例如，我们对 map 操作创建的数据集进行 reduce 操作时，只会向驱动(driver)返回 reduce 操作的结果，而不是返回更大的 map 操作创建的数据集。

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(action)时被重新计算。不过，你也可以使用 *持久化(persist)*(或者 缓存(cache))方法，把一个 RDD 持久化(persist)到内存中。 在这种情况下，Spark 会在集群中保存相关元素，以便你下次查询这个 RDD 时，能更快速地访问。对于把 RDDs 持久化到磁盘上，或在集群中复制到多个节点也是支持的。

4.3.1 基础操作

- Scala

为了描述 RDD 的基础操作，可以考虑下面的简单程序：

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

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

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

第一行通过一个外部文件定义了一个基本的 RDD。这个数据集未被加载到内存，也未在上面执行动作： lines 仅仅指向这个文件。第二行定义了 lineLengths 作为 map 转换的结果。此外，由于惰性，*不会* 立即计算 lineLengths。 最后，我们运行 reduce，这是一个动作(action)。这时候，Spark 才会将这个计算拆分成不同的 task， 并运行在独立的机器上，并且每台机器运行它自己的 map 部分和本地的 reducation，仅仅返回它的结果给驱动程序。

如果我们希望以后可以复用 lineLengths，可以添加：

```
lineLengths.persist()
```

在 reduce 之前，这将导致 lineLengths 在第一次被计算之后，被保存在内存中。

- Java

为了描述 RDD 的基础操作，可以考虑下面的简单程序：

```
JavaRDD<String> lines = sc.textFile("data.txt");

JavaRDD<Integer> lineLengths = lines.map(s -> s.length());

int totalLength = lineLengths.reduce((a, b) -> a + b);
```

第一行通过一个外部文件定义了一个基本的 RDD。这个数据集未被加载到内存，也未在上面执行动作： lines 仅仅指向这个文件。第二行定义了 lineLengths 作为 map 转换的结果。此外，由于惰性，*不会* 立即计算 lineLengths。 最后，我们运行 reduce，这是一个动作(action)。这时候，Spark 才会将这个计算拆分成不同的 task， 并运行在独立的机器上，并且每台机器运行它自己的 map 部分和本地的 reducation，仅仅返回它的结果给驱动程序。

如果我们希望以后可以复用 lineLengths，可以添加：

```
lineLengths.persist();
```

在 reduce 之前，这将导致 lineLengths 在第一次被计算之后，被保存在内存中。

- Python

为了描述 RDD 的基础操作，可以考虑下面的简单程序：

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

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

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

第一行通过一个外部文件定义了一个基本的 RDD。这个数据集未被加载到内存，也未在上面执行动作：lines 仅仅指向这个文件。第二行定义了 lineLengths 作为 map 转换的结果。此外， 由于惰性，*不会* 立即计算 lineLengths。 最后，我们运行 reduce，这是一个动作(action)。这时候，Spark 才会将这个计算拆分成不同的 task， 并运行在独立的机器上，并且每台机器运行它自己的 map 部分和本地的 reducation，仅仅返回它的结果给驱动程序。

如果我们希望以后可以复用 lineLengths，可以添加：

```
lineLengths.persist()
```

在 reduce 之前，这将导致 lineLengths 在第一次被计算之后，被保存在内存中。

4.3.2 把函数传递到 Spark

- Scala

Spark 的 API ,在很大程度上依赖于把驱动程序(driver program)中的函数传递到集群上运行。这有两种推荐的实现方式:

- 使用匿名函数的语法(Anonymous function syntax) ,这可以用来替换简短的代码。

- 使用全局单例对象(global singleton object)的静态方法。比如，你可以定义函数对象 object MyFunctions，然后传递该对象的方法 MyFunctions.func1， 如下所示：

```
object MyFunctions {

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

}



myRdd.map(MyFunctions.func1)
```

注意：由于可能传递的是一个类实例的方法的引用(而不是一个单例对象(singleton object))，在传递方法的时候，应该同时传递包含该方法的对象。比如，考虑：

```
class MyClass {

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

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

}
```

这里，如果我们创建了一个类实例 new MyClass，并且调用了实例的 doStuff 方法，该方法中的 map 处调用了这个 MyClass 实例 的 func1 方法 ，所以需要将整个对象传递到集群中。类似于写成： rdd.map(x => this.func1(x)).

类似地，访问外部对象的字段时将引用整个对象：

```
class MyClass {

  val field = "Hello"

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

}
```

等同于写成 rdd.map(x => this.field + x)，引用了整个 this。为了避免这种问题，最简单的方式是把 field 拷贝到本地变量，而不是去外部访问它：

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

  val field_ = this.field

  rdd.map(x => field_ + x)

}
```

- Java

Spark 的 API，在很大程度上依赖于把驱动程序(driver program)中的函数传递到集群上运行。 在 Java 中，函数由那些实现了 org.apache.spark.api.java.function 包中的接口的类表示。有两种创建这样的函数的方式：

- 在你自己的类中实现 Function 接口，可以是匿名内部类，或者命名类，并且传递类的一个实例到 Spark。
- 在 Java 8 中，使用 lambda 表达式 来简明地定义函数的实现。

为了保持简洁性，本指南中大量使用了 lambda 语法，这在长格式(long-form)中很容易使用所有相同的 APIs。比如，我们可以把上面的代码写成：

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new Function<String, Integer>() {
  public Integer call(String s) { return s.length(); }
});
int totalLength = lineLengths.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer a, Integer b) { return a + b; }
});
```

或者，如果不方便编写内联函数的话，可以写成：

```
class GetLength implements Function<String, Integer> {
  public Integer call(String s) { return s.length(); }
}
class Sum implements Function2<Integer, Integer, Integer> {
  public Integer call(Integer a, Integer b) { return a + b; }
}

JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new GetLength());
int totalLength = lineLengths.reduce(new Sum());
```

注意，Java 中的匿名内部类也可以访问封闭域(the enclosing scope)中的变量，只要这些变量标识为 final 即可。 Spark 会像处理其他语言一样，将这些变量拷贝到每个工作(worker)节点上。

- Python

Spark 的 API，在很大程度上依赖于把驱动程序(driver program)中的函数传递到集群上运行。有三种推荐方法可以使用：

- 使用 Lambda 表达式 来编写可以写成一个表达式的简单函数。 (Lambdas 不支持没有返回值的多语句函数(multi-statement functions) 或表达式。)
- Spark 调用的函数中的 Local defs， 可以用来代替更长的代码。
- 模块(module)中的顶级(Top-level)函数。

例如，如果想传递一个支持使用 lambda 表达式的更长的函数，可以考虑以下代码：

```
"""MyScript.py"""
if __name__ == "__main__":
    def myFunc(s):
        words = s.split(" ")
        return len(words)

    sc = SparkContext(...)
    sc.textFile("file.txt").map(myFunc)
```

注意：由于可能传递的是一个类实例的方法的引用(而不是一个单例对象(singleton object))，在传递方法的时候，应该同时传递包含该方法的对象。比如，考虑：

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

这里，如果我们创建了一个类实例 new MyClass，并且调用了实例的 doStuff 方法，该方法中的 map 处调用了这个 MyClass 实例的 func 方法，所以需要将整个对象传递到集群中。

类似地，访问外部对象的字段时将引用整个对象：

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

为了避免这种问题，最简单的方式是把 field 拷贝到本地变量，而不是去外部访问它:

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

4.3.3 键值对(Key-Value Pairs)的使用

- Scala

虽然，在包含任意类型的对象的 RDDs 中，可以使用大部分的 Spark 操作，但也有一些特殊的操作只能在键值(key-value) 对的 RDDs 上使用。最常见的一个就是分布式的洗牌("shuffle")操作，诸如基于 key 值对元素进行分组或聚合的操作。

在 Scala 中，包含 二元组(Tuple2) 对象（可以通过简单地(a, b)代码，来构建内置(built-in)于语言中的元组(tuples)）的 RDDs 支持这些操作，只要你在程序中导入了 org.apache.spark.SparkContext._，就能进行隐式转换(implicit conversions)。PairRDDFunctions 类支持键值(key-value)对的操作， 如果你导入了隐式转换，该类型就能自动地对元组(tuples) RDD 的元素进行转换。

比如，下列代码在键值(key-value)对上使用了 reduceByKey 操作，来计算在一个文件中每行文本出现的总次数：

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

我们也可以使用 counts.sortByKey()， 比如，将键值对以字典序(alphabetically)进行排序。最后使用 counts.collect() 转换成对象的数组形式，返回给驱动程序(driver program)。

注意： 在键值(key-value)对操作中，如果使用了自定义对象作为键，你必须确保该对象实现了自定义的 equals() 和对应的 hashCode() 方法。 更多详情请查看 Object.hashCode() 文档 大纲中列出的规定。

- Java

虽然在包含任意类型的对象的 RDDs 中，可以使用大部分的 Spark 操作，但也有一些特殊的操作只能在键值(key-value)对的 RDDs 上使用。最常见的一个是分布式的洗牌("shuffle")操作，诸如基于 key 值对元素进行分组或聚合的操作。

在 Java 中,可以使用 Scala 标准库中的 scala.Tuple2 类来表示键值(key-value)对，你可以简单地调用 new Tuple2(a, b) 来创建一个元组(tuple)，然后使用 tuple._1() 和 tuple._2()方法来访问元组的字段。

使用 JavaPairRDD 来表示键值(key-value)对 RDDs。你可以使用指定版本的 map 操作，从 JavaRDDs 构建 JavaPairRDDs，比如 mapToPair 和 flatMapToPair。JavaPairRDD 支持标准的 RDD 函数，也支持特殊的键值(key-value)函数。

例如，下面的代码在键值(key-value)对上使用 reduceByKey 操作来计算在一个文件中每行文本出现的总次数：

```
JavaRDD<String> lines = sc.textFile("data.txt");

JavaPairRDD<String, Integer> pairs = lines.mapToPair(s -> new Tuple2(s, 1));

JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b);
```

我们也可以使用 counts.sortByKey()， 例如，将键值对以字典序(alphabetically)进行排序。后调用 counts.collect() 转换成对象的数组形式，返回给驱动程序(driver program)。

注意： 在键值(key-value)对操作中，如果使用了自定义对象作为键，你必须确保该对象实现了自定义的 equals() 和对应的 hashCode() 方法。 更多详情请查看 Object.hashCode() documentation 文档大纲中列出的规定。

- Python

虽然在包含任意类型的对象的 RDDs 中，可以使用大部分的 Spark 操作，但也有一些特殊的操作只能在键值(key-value)对的 RDDs 上使用。最常见的一个就是分布式的洗牌("shuffle")操作，诸如基于 key 值对元素进行分组或聚合的操作。

在 Python 中，RDDs 支持的操作包含 Python 内置的元组(tuples)操作，比如 (1, 2)。你可以简单地创建这样的元组，然后调用期望的操作。

例如，下面的代码在键值(key-value)对上使用 reduceByKey 操作来计算在一个文件中每行文本出现的总次数：

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

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

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

我们也可以使用 counts.sortByKey()， 例如，按照字典序(alphabetically)排序键值对。最后调用 counts.collect() 转换成对象的数组形式，返回给驱动程序(driver program)。

4.3.4 转换(Transformations)

下表中列出了 Spark 支持的一些常见的转换 (Transformations)。详情请参考 RDD API 文档 (Scala, Java, Python) 和 pair RDD 函数文档 (Scala, Java)。

转换(Transformation)	含义
map(*func*)	返回一个新分布式数据集，由每一个输入元素经过 *func* 函数转换后组成。
filter(*func*)	返回一个新数据集，由经过 *func* 函数计算后返回值为 true 的输入元素组成。
flatMap(*func*)	类似于 map，但是每一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(Seq)，而不是单一元素)。
mapPartitions(*func*)	类似于 map，但独立地在 RDD 的每一个分区(partition，对应块(block))上运行，当在类型为 T 的 RDD 上运行时，*func* 的函数类型必须是 Iterator<T> => Iterator<U>。
mapPartitionsWithIndex(*func*)	类似于 mapPartitions，但 *func* 带有一个整数参数表示分区(partition)的索引值。当在类型为 T 的 RDD 上运行时，*func* 的函数类型必须是(Int, Iterator<T>) => Iterator<U>。
sample(*withReplacement*, *fraction*, *seed*)	根据 *fraction* 指定的比例，对数据进行采样，可以选择是否用随机数进行替换，seed 用于指定随机数生成器种子。
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的元素联合(union)而成。
intersection(*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的元素的交集(intersection)组成。
distinct([*numTasks*]))	返回一个新的数据集，新数据集由源数据集过滤掉多余的

	重复元素只保留一个而成。
groupByKey([*numTasks*])	在一个 (K, V) 对的数据集上调用，返回一个 (K, Iterable<V>) 对的数据集。 **注意：**如果你想在每个 key 上分组执行聚合（如总和或平均值）操作，使用 reduceByKey 或 combineByKey 会产生更好的性能。 **注意：** 默认情况下，输出的并行数依赖于父 RDD(parent RDD)的分区数(number of partitions)。你可以通过传递可选的第二个参数 numTasks 来设置不同的任务数。
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，该函数的类型必须是 (V,V) => V。类似 groupByKey，reduce 的任务个数是可以通过第二个可选参数来配置的。
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	在一个 (K, V) 对的数据集上调用时，返回一个 (K, U) 对的数据集，对每个键的值使用给定的组合函数(combine functions)和一个中性的"零"值进行聚合。允许聚合后的值类型不同于输入的值类型，从而避免了不必要的内存分配。如同 groupByKey， 可以通过设置第二个可选参数来配置 reduce 任务的个数。
sortByKey([*ascending*], [*numTasks*])	在一个 (K, V) 对的数据集上调用，其中，K 必须实现 Ordered，返回一个按照 Key 进行排序的 (K, V) 对数据集，升序或降序由布尔参数 ascending 决定。
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 对应的所有元素对在一起的 (K, (V, W)) 对的数据集。外联(Outer joins)操作由 leftOuterJoin，rightOuterJoin 和 fullOuterJoin 来支持。
cogroup(*otherDataset*, [*numTasks*])	在类型为 (K, V) 和 (K, W)的数据集上调用，返回一个 (K, Iterable<V>, Iterable<W>)元组(tuples)的数据集。这个操作也可以称之为 groupWith。
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)对的数据集(所有元素交互进行笛卡尔积)。
pipe(*command*, *[envVars]*)	以管道(Pipe)方式将 RDD 的各个分区(partition) 传递到 shell 命令，比如 一个 Perl 或 bash 脚本中。 RDD 的元素会被写入进程的标准输入(stdin)，并且将作为字符串 的 RDD(RDD of strings) ，在进程的标准输出(stdout)上输出一行行数据。
coalesce(*numPartitions*)	把 RDD 的分区数降低到指定的 numPartitions。过滤掉一

	个大数据集之后再执行操作会更加有效。
repartition(*numPartitions*)	随机地对 RDD 的数据重新洗牌(Reshuffle)，以便创建更多或更少的分区，对它们进行平衡。总是在网络上对所有数据进行洗牌(shuffles)。
repartitionAndSortWithinPartitions(partitioner)	根据指定的 partitioner 对 RDD 重新分区， 并且在每个结果分区中，记录会根据其键值进行排序。 这比先调用 repartition，然后在每个分区中调用 sorting 的效率高很多，因为它可以把 sorting 放到 shuffle 机制中执行。

4.3.5 动作(Actions)

下表中列出了 Spark 支持的一些常见的动作 (actions)。详情请参考 RDD API 文档 (Scala, Java, Python) 和 pair RDD 函数文档 (Scala, Java)。

动作(Action)	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。该函数应该是可交换和可结合的，以便它可以正确地并行计算。
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。
count()	返回数据集的元素的个数。
first()	返回数据集的第一个元素。 (类似于 take(1)).
take(*n*)	返回一个由数据集的前 *n* 个元素组成的数组。注意，这个操作目前不能并行执行，而是由驱动程序(driver program)计算所有的元素。
takeSample(*withReplacement*, *num*, [*seed*])	返回一个数组，由数据集中随机采样的 num 个元素组成，可以选择是否用随机数替换不足的部分，可以指定可选参数 seed，预先指定一个随机数生成器的种子。
takeOrdered(*n*, *[ordering]*)	返回一个由数据集的前 *n* 个元素，并使用自然顺序或定制顺序对这些元素进行排序。
saveAsTextFile(*path*)	将数据集的元素，以 text file (或 text file 的集合) 的形式，保存到本地文件系统的指定目录，Spark 会对每个元素调用 toString 方法，然后转换为文件中的文本行。
saveAsSequenceFile(*path*) (Java and Scala)	将数据集的元素，以 Hadoop sequencefile 的格式，保存到各种文件系统的指定路径下，包括本地系统，HDFS 或者任何其它 hadoo 支持的文件系统。 该方法只能用于 键值(key-value)对的 RDDs，或者实现了 Hadoop 的 Writable 接口的情况下。在 Scala 中，也可以用于支持隐式转换为 Writable 的类型。(Spark 包括了基本类型的转换，例如 Int，Double，String，等等)。
saveAsObjectFile(*path*)	以简单地 Java 序列化方式将数据集的元素写入指定的路径， 对应的可以

(Java and Scala)	用 SparkContext.objectFile() 加载该文件。
countByKey()	只对 (K,V) 类型的 RDD 有效。返回一个 (K，Int) 对的 hashmap，其中 (K, Int) 对表示每一个 key 对应的元素个数。
foreach(*func*)	在数据集的每一个元素上，运行 *func* 函数。这通常用于副作用(side effects)，例如更新一个累加器变量(accumulator variable)(参见下文)，或者和外部存储系统进行交互.

4.4RDD 持久化(Persistence)

Spark 最重要的一个功能 ,就是在不同操作间 ,将一个数据集*持久化(persisting)* (或 *缓存(caching)*) 到内存中。当你持久化(persist)一个 RDD，每一个节点都会把它计算的所有分区(partitions)存储在内存中，并在对数据集 (或者衍生出的数据集)执行其他动作(actioins)时重用。这将使得后续动作(actions)的执行变得更加迅速(通常快 10 倍)。 缓存(Caching)是用 Spark 构建迭代算法和快速地交互使用的关键。

你可以使用 persist() 或 cache() 方法来持久化一个 RDD。在首次被一个动作(action)触发计算后， 它将会被保存到节点的内存中。 Spark 的缓存是带有容错机制的，如果 RDD 丢失任何一个分区的话，会自动地用原先构建它的转换(transformations)操作来重新进行计算。

此外，每一个被持久化的 RDD 都可以用不同的 *存储级别(storage level)* 进行存储，比如，允许你持久化数据集到硬盘，序列化 Java 对象(节省空间)后存储到内存， 跨节点复制，或者以 off-heap 的方式存储在 Tachyon。这些级别的选择，是通过将一个 StorageLevel 对象 (Scala Java, Python) 传递到 persist() 方法中进行设置的。 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 (deserialized) 的对象存入内存 ）。 完整的可选存储级别如下：

存储级别(Storage Level)	**含义**
MEMORY_ONLY	将 RDD 以反序列化(deserialized) 的 Java 对象存储到 JVM。如果 RDD 不能被内存装下，一些分区将不会被缓存，并且在需要的时候被重新计算。这是默认的级别。
MEMORY_AND_DISK	将 RDD 以反序列化(deserialized) 的 Java 对象存储到 JVM。如果 RDD 不能被内存装下，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	将 RDD 以 *序列化(serialized)* 的 Java 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(deserialized)的空间利用率更高，尤其当使用 快速序列化器(fast serializer)，但在读取时会比较耗 CPU。
MEMORY_AND_DISK_SER	类似于 MEMORY_ONLY_SER，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	只将 RDD 分区存储在硬盘上。
MEMORY_ONLY_2,	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节点上。

MEMORY_AND_DISK_2,

等。

OFF_HEAP (experimental)	以序列化的格式(serialized format)将 RDD 存储到 Tachyon。 相比于 MEMORY_ONLY_SER，OFF_HEAP 降低了垃圾收集(garbage collection)的开销，并使 executors 变得更小而且共享内存池，这在大堆(heaps)和多应用并行的环境下是非常吸引人的。 而且，由于 RDDs 驻留于 Tachyon 中，executor 的崩溃不会导致内存中的缓存丢失。 在这种模式下，Tachyon 中的内存是可丢弃的。 因此， Tachyon 不会尝试重建一个在内存中被清除的分块。

注意: *在 Python 中，存储对象时总是使用 Pickle 库来序列化(serialized)，而不管你是否选择了一个序列化的级别。*

即使用户没有调用 persist，Spark 也会自动地持久化一些洗牌(shuffle)操作(比如，reduceByKey)的中间数据。这是为了避免在一个节点上的洗牌(shuffle)过程失败时，重新计算整个输入。我们仍然建议用户在结果 RDD 上调用 persist， 如果希望重用它的话。

4.4.1 如何选择存储级别?(Which Storage Level to Choose?)

Spark 的存储级别旨在满足内存使用和 CPU 效率权衡上的不同需求。我们建议通过以下方法进行选择：

- 如果你的 RDDs 可以很好的与默认的存储级别(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 的操作尽可能的快。
- 如果不行，试着使用 MEMORY_ONLY_SER， 并且 选择一个快速序列化库 使对象在有比较高的空间使用率(space-efficient)的情况下，依然可以较快被访问。
- 尽可能不要存储到硬盘上，除非计算数据集的函数的计算量特别大，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，可能和从硬盘中读取差不多快。
- 如果你想有快速的故障恢复能力，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的数据来恢复错误的容错机制，但是复制的存储级别可以让你在 RDD 上持续地运行任务，而不需要等待丢失的分区被重新计算。
- 在大量的内存或多个应用程序的环境下，试验性的 OFF_HEAP 模式具有以下几个优点：
  - 允许多个 executors 共享 Tachyon 中相同的内存池。
  - 极大地降低了垃圾收集器(garbage collection)的开销。
  - 即使个别的 executors 崩溃了，缓存的数据也不会丢失。

4.4.2 移除数据

Spark 会自动监控各个节点上的缓存使用情况，并使用最近最少使用算法(least-recently-used (LRU)) 删除老的数据分区。 如果你想手动移除一个 RDD，而不是等它自动从缓存中清除，可以使用 RDD.unpersist() 方法。

第 5 章 共享变量(Shared Variables)

一般来说，当一个函数被传递给一个在远程集群节点上运行的 Spark 操作(例如 map 或 reduce) 时，它操作的是这个函数用到的所有变量的独立拷贝。 这些变量会被拷贝到每一台机器，而且在远程机器上对变量的所有更新都不会被传播回驱动程序。通常看来，读-写任务间的共享变量显然不够高效。然而，Spark 还是为两种常见的使用模式，提供了两种有限的共享变量：广播变量(broadcast variables)和累加器(accumulators)。

5.1 广播变量(Broadcast Variables)

广播变量允许程序员保留一个只读的变量，缓存在每一台机器上，而不是每个任务保存一份拷贝。它们可以这样被使用，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 会尝试使用一种高效的广播算法来传播广播变量，从而减少通信的代价。

广播变量是通过调用 SparkContext.broadcast(v) 方法从变量 v 创建的。广播变量是一个 v 的封装器，它的值可以通过调用 value 方法获得。如下代码展示了这个：

- 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)
```

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节点上。另外，对象 v 不能在广播后修改， 这样可以保证所有节点具有相同的广播变量的值(比如，后续如果变量被传递到一个新的节点)。

- Java

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});

broadcastVar.value();
// returns [1, 2, 3]
```

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节点上。另外，对象 v 不能在广播后修改， 这样可以保证所有节点具有相同的广播变量的值(比如，后续如果变量被传递到一个新的节点)。

- Python

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value
[1, 2, 3]
```

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节点上。另外，对象 v 不能在广播后修改， 这样可以保证所有节点具有相同的广播变量的值(比如，后续如果变量被传递到一个新的节点)。

5.2 累加器(Accumulators)

累加器是一种只能通过具有结合性的操作(associative operation)进行“加(added)”的变量，因此可以高效地支持并行。它们可以用来实现计数器(如 MapReduce 中)和求和器。Spark 原生就支持数值类型的累加器，开发者可以自己添加新的支持类型。 如果创建了一个命名的累加器(accumulators)， 这些累加器将会显示在 Spark UI 界面上。 这对于了解当前运行阶段(stages)的进展情况是非常有用的(注意：这在 Python 中尚未支持)。

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，可以通过使用 add 方法或 += 操作符(在 Scala 和 Python)来给它加值。然而，它们不能读取这个值。只有驱动程序可以使用 value 方法来读取累加器的值。

以下代码展示了如何利用一个累加器，将一个数组里面的所有元素相加：

- Scala

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

scala> accum.value
res2: Int = 10
```

虽然代码可以使用内置支持的 Int 类型的累加器，但程序员也可以通过子类化(subclassing) AccumulatorParam 来创建自己的类型。AccumulatorParam 接口有两个方法： zero ，为你的数据类型提供了一个“零值(zero value)” ， 以及 addInPlace 提供了两个值相加的方法。 比如， 假设我们有一个表示数学上向量的 Vector 类，我们可以这么写：

```
object VectorAccumulatorParam extends AccumulatorParam[Vector] {
  def zero(initialValue: Vector): Vector = {
    Vector.zeros(initialValue.size)
  }
  def addInPlace(v1: Vector, v2: Vector): Vector = {
    v1 += v2
  }
}

// 然后，创建一个该类型的累加器(Accumulator):
val vecAccum = sc.accumulator(new Vector(...))(VectorAccumulatorParam)
```

在 Scala 中， Spark 也支持更通用的 Accumulable 接口去累加数据，其结果类型和累加的元素不同(比如，构建一个包含所有元素的列表)，并且 SparkContext.accumulableCollection 方法可以累加普通的 Scala 集合(collection)类型。

对于累加器的更新，只有在 actions 中执行时，Spark 才保证每个 task 对累加器的更新仅执行一次，也就是说，重启的 tasks 不会再更新该累加器。在转换操作中，用户应该意识到，如果 tasks 或 job 的 stages 被重新执行的话，每个 task 中的累加器就可能不止一次被更新了。

- Java

```
Accumulator<Integer> accum = sc.accumulator(0);


sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x));

// ...

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


accum.value();

// returns 10
```

虽然代码可以使用内置支持的 Integer 类型的累加器，但程序员也可以通过子类化(subclassing) AccumulatorParam 来创建自己的类型。AccumulatorParam 接口有两个方法：zero ，为你的数据类型提供了一个“零值(zero value)”，以及 addInPlace 提供了两个值相加的方法。比如，假设我们有一个表示数学上向量的 Vector 类，我们可以这么写：

```
class VectorAccumulatorParam implements AccumulatorParam<Vector> {

  public Vector zero(Vector initialValue) {

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

  }

  public Vector addInPlace(Vector v1, Vector v2) {

    v1.addInPlace(v2); return v1;

  }

}


// 然后，创建一个该类型的累加器(Accumulator):

Accumulator<Vector> vecAccum = sc.accumulator(new Vector(...), new VectorAccumulatorParam());
```

在 Java 这， Spark 也支持更通用的 Accumulable 接口去累加数据，其结果类型和累加的元素不同(比如，构建一个包含所有元素的列表)。

对于累加器的更新， 只有在 actions 中执行时， Spark 才保证每个 task 对累加器的更新仅执行一次， 也就是说，重启的 tasks 不会再更新该累加器。 在转换操作中，用户应该意识到， 如果 tasks 或 job 的 stages 被重新执行的话， 每个 task 中的累加器就可能不止一次被更新了。

- 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
```

虽然代码可以使用内置支持的 Int 类型的累加器，但程序员也可以通过子类化(subclassing) AccumulatorParam 来创建自己的类型. AccumulatorParam 接口有两个方法： zero ，为你的数据类型提供了一个“零值(zero value)” ， 以及 addInPlace 提供了两个值相加的方法。 比如， 假设我们有一个表示数学上向量的 Vector 类，我们可以这么写：

```
class VectorAccumulatorParam(AccumulatorParam):
    def zero(self, initialValue):
        return Vector.zeros(initialValue.size)

    def addInPlace(self, v1, v2):
        v1 += v2
        return v1

# 然后，创建一个该类型的累加器(Accumulator)：
vecAccum = sc.accumulator(Vector(...), VectorAccumulatorParam())
```

对于累加器的更新，只有在 actions 中执行时，Spark 才保证每个 task 对累加器的更新仅执行一次，也就是说，重启的 tasks 不会再更新该累加器。在转换操作中，用户应该意识到，如果 tasks 或 job 的 stages 被重新执行的话，每个 task 中的累加器就可能不止一次被更新了。

第 6 章 把代码部署到集群上(Deploying to a Cluster)

应用程序提交指南(application submission guide) 描述了如何将应用程序提交到一个集群，简单地说，一旦你将你的应用程序打包成一个 JAR(对于 Java/Scala) 或者一组的 .py 或 .zip 文件 (对于 Python)，bin/spark-submit 脚本可以让你将它提交到支持的任何集群管理器中。

第 7 章 单元测试(Unit Testing)

Spark 对单元测试非常友好，可以使用任何流行的单元测试框架。在你的测试中简单地创建一个 SparkContext ，并将 master URL 设置成 local，运行你的各种操作，然后调用 SparkContext.stop() 结束测试。确保在 finally 块或测试框架的 tearDown 方法中调用 context 的 stop 方法，因为 Spark 不支持在一个程序中同时运行两个 contexts。

第 8 章 Spark 1.0 之前版本的迁移(Migrating from pre-1.0 Versions of Spark)

- Scala

Spark 1.0 冻结了 1.X 系列的 Spark 核心(Core) API，现在，其中的 API，除了标识为 “试验性(experimental)” 或 “开发者的(developer) API” 的，在将来的版本中都会被支持。对 Scala 用户而言，唯一的改变在于组操作(grouping operations)，比如，groupByKey，cogroup 和 join，其返回值已经从 (Key, Seq[Value])对修改为(Key, Iterable[Value])。

迁移指南也可以从 Spark Streaming，MLlib 和 GraphX 获取。

- Java

Spark 1.0 冻结了 1.X 系列的 Spark 核心(Core) API，现在，其中的 API，只要不是标识为 “试验性(experimental)” 或 “开发者的(developer) API” 的，在将来的版本中都会被支持。 其中对 Java API 做了一些修改：

- 对于 org.apache.spark.api.java.function 中的类函数(Function classes)， 在 1.0 版本中变成了接口，这意味着旧的代码中 extends Function 应该需要为 implement Function。
- 增加了 map 转换(transformations)的新变体，如 mapToPair 和 mapToDouble， 用于创建指定数据类型的 RDDs。
- 组操作(grouping operations)， 如 groupByKey， cogroup 和 join 的返回值也被修改了，从原先返回 (Key, List<Value>) 对改为(Key, Iterable<Value>)。

迁移指南也可以从 Spark Streaming， MLlib 和 GraphX 获取。

- Python

Spark 1.0 冻结了 1.X 系列的 Spark 核心(Core) API，现在，其中的 API，只要不是标识为 “试验性(experimental)” 或 “开发者的(developer) API” 的，在将来的版本中都会被支持。对 Python 用户而言，唯一的修改在于分组操作(grouping operations)，比如 groupByKey， cogroup 和 join， 其返回值从 (key, list of values) 对 修改为 (key, iterable of values)。

迁移指南也可以从 Spark Streaming， MLlib 和 GraphX 获取。

下一步

你可以在 Spark 的网站上看到 spark 程序的样例。另外， Spark 在 examples 目录 (Scala, Java, Python) 中也包含了一些样例 。 你可以通过将类名传递给 spark 的 bin/run-example 脚本来运行 Java 和 Scala 的样例，例如：

```
./bin/run-example SparkPi
```

对于 Python 样例, 要使用 spark-submit：

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

为了帮助优化你的程序，在 配置(configuration) 和 调优(tuning) 的指南上提供了最佳实践信息。它们在确保将你的数据用一个有效的格式存储在内存上，是非常重要的。 对于部署的帮助信息，可以查看 集群模式概述(cluster mode overview)， 描述了分布式操作以及支持集群管理器所涉及的组件。

最后，完整的 API 文档可以查看 Scala, Java and Python 。

Spark 亚太研究院

Spark 亚太研究院是中国最专业的一站式大数据 Spark 解决方案供应商和高品质大数据企业级完整培训与服务供应商，以帮助企业规划、架构、部署、开发、培训和使用 Spark 为核心，同时提供 Spark 源码研究和应用技术训练。针对具体 Spark 项目，提供完整而彻底的解决方案。包括 Spark 一站式项目解决方案、Spark 一站式项目实施方案及 Spark 一体化顾问服务。

官网： www.sparkinchina.com

视频课程：

《大数据 Spark 实战高手之路》 国内第一个 Spark 视频系列课程

从零起步，分阶段无任何障碍逐步掌握大数据统一计算平台 Spark，从 Spark 框架编写和开发语言 Scala 开始，到 Spark 企业级开发，再到 Spark 框架源码解析、Spark 与 Hadoop 的融合、商业案例和企业面试，一次性彻底掌握 Spark，成为云计算大数据时代的幸运儿和弄潮儿，笑傲大数据职场和人生！

- 第一阶段：熟练的掌握 Scala 语言
  课程学习地址：http://edu.51cto.com/pack/view/id-124.html
- 第二阶段：精通 Spark 平台本身提供给开发者 API
  课程学习地址：http://edu.51cto.com/pack/view/id-146.html
- 第三阶段：精通 Spark 内核
  课程学习地址：http://edu.51cto.com/pack/view/id-148.html
- 第四阶段：掌握基于 Spark 上的核心框架的使用
  课程学习地址：http://edu.51cto.com/pack/view/id-149.html
- 第五阶段：商业级别大数据中心黄金组合：Hadoop+ Spark
  课程学习地址：http://edu.51cto.com/pack/view/id-150.html
- 第六阶段：Spark 源码完整解析和系统定制
  课程学习地址：http://edu.51cto.com/pack/view/id-151.html

图书：

《大数据 spark 企业级实战》

京东购买官网：http://item.jd.com/11622851.html

当当购买官网：http://product.dangdang.com/23631792.html

亚马逊购买官网：http://www.amazon.cn/dp/B00RMD8KI2/