第3章

Spark 应用开发基础

3.1 Spark 的 Python 编程环境设置

Apache Spark 是 Scala 语言实现的一个计算框架。为了支持 Python 语言使用 Spark，Apache Spark 社区开发了一个工具 PySpark。利用 PySpark 中的 Py4j 库，可以通过 Python 语言操作 RDDs。

PySpark 提供了 PySpark Shell，它是一个结合了 Python API 和 Spark Core 的工具，同时能够初始化 Spark 环境。目前，由于 Python 具有丰富的扩展库，大量的数据科学家和数据分析从业人员都在使用 Python。因此，PySpark 将 Spark 支持 Python 是对两者的一次共同促进。

Spark 可以独立安装使用，也可以和 Hadoop 一起安装使用。在安装 Spark 之前，首先确保安装了 Java 8 或者更高的版本，并安装配置好 Scala。

1. Spark 安装

访问 Spark 官网 http://spark.apache.org/，并选择最新版本的 Spark 直接下载。截止到 2020 年 9 月 11 日，Spark 的最新版本是 3.0.1。下面以版本 2.4.5 为例，描述 Windows 10 下的环境配置过程。

(1) 将 spark-2.4.5-bin-hadoop2.7 解压缩到文件夹 D:\ProgramFiles 中。

(2) 配置环境变量，见表 3-1。

表 3-1 大数据平台的环境变量设置

变量名	变量值
JAVA_HOME	D:\ProgramFiles\Java\jdk1.8.0_202
HADOOP_HOME	D:\ProgramFiles\hadoop-3.1.3
HIVE_HOME	D:\ProgramFiles\hive-3.1.2
SCALA_HOME	D:\ProgramFiles\scala-2.12.11
SPARK_HOME	D:\ProgramFiles\spark-2.4.5-bin-hadoop2.7
PYSPARK_DRIVER_PYTHON	ipython
PYSPARK_DRIVER_PYTHON_OPTS	notebook

(3) 配置 PATH 路径，例如：

PATH＝%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;%HADOOP_HOME%\bin;%HADOOP_HOME%\sbin;%HIVE_HOME%\bin;%SCALA_

HOME%\bin;%SPARK_HOME%\bin

配置完成后，在 shell 中输入 spark-shell 或者 pyspark 就可以进入 Spark 的交互式编程环境中，前者是进入 Scala 交互式环境，后者是进入 Python 交互式环境，如图 3-1 所示。

```
C:\Users\zxm>spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://windows10.microdone.cn:4040
Spark context available as 'sc' (master = local[*], app id = local-1598944315462).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.5
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_202)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

图 3-1　Spark 的安装测试

2. 配置 Python 编程环境

介绍两种编程环境：Jupyter 和 Visual Studio Code。前者便于进行交互式编程，后者便于集成式开发。

1) PySpark in Jupyter Notebook

方法一：配置 PySpark 启动器的 2 个环境变量。

```
PYSPARK_DRIVER_PYTHON= jupyter
PYSPARK_DRIVER_PYTHON_OPTS= notebook
```

方法二：使用 findSpark 包，在代码中提供 Spark 上下文环境。该方法具有通用性，值得推荐。首先安装 findspark。

```
pip install findspark
```

然后打开一个 Jupyter notebook。在进行 Spark 编程时，先导入 findspark 包，示例如图 3-2 所示。

```
1  # 导入 findspark 并初始化
2  import findspark
3  findspark.init()
4  from pyspark import SparkConf, SparkContext
5  import random# 配置
6  conf = SparkConf().setMaster("local[*]").setAppName("Pi")# 利用上下文启动
7  sc = SparkContext(conf=conf)
8  num_samples = 100000000
9  def inside(p):
10     x, y = random.random(), random.random()
11     return x*x + y*y < 1
12 count = sc.parallelize(range(0, num_samples)).filter(inside).count()
13 pi = 4 * count / num_samples
14 print(pi)
15 sc.stop()
```

```
3.14168196
```

图 3-2　基于 findspark 包的 Spark 计算 pi 值

2) PySpark in VScode

在 VScode 上使用 Spark,不需要使用 findspark 包,可以直接进行编程。

```
from pyspark import SparkContext, SparkConf
conf =SparkConf().setMaster("local[*]").setAppName("test")
sc =SparkContext(conf=conf)
logFile = "d:/ProgramFiles/spark-2.4.5-bin-hadoop2.7/README.md"
logData = sc.textFile(logFile, 2).cache()
numAs = logData.filter(lambda line: 'a'in line).count()
numBs = logData.filter(lambda line: 'b' in line).count()
print("Lines with a: {0}, Lines with b:{1}".format(numAs, numBs))
```

运行结果如图 3-3 所示。

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/09/01 15:55:38 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Lines with a: 61, Lines with b:30
```

图 3-3　在 VSCode 环境中运行 Spark 程序示例

3. SparkSession 介绍

SparkSession 是 Spark2.0 引入的新概念。SparkSession 为用户提供了统一的切入点,来让用户学习 Spark 的各项功能。

在 Spark 的早期版本中,SparkContext 是 Spark 的主要切入点,由于 RDD 是主要的 API,通过 SparkContext 来创建和操作 RDD。对于每个其他的 API,就需要使用不同的 Context。例如,对于 Streming,使用 StreamingContext;对于 SQL,使用 SQLContext;对于 Hive,使用 HiveContext。但是随着 DataSet 和 DataFrame 的 API 逐渐成为标准的 API,就需要为它们建立接入点。所以在 Spark2.0 中,引入 SparkSession 作为 DataSet 和 DataFrame API 的切入点,SparkSession 封装了 SparkConf、SparkContext 和 SQLContext。为了向后兼容,SQLContext 和 HiveContext 也被保存下来。

SparkSession 实质上是 SQLContext 和 HiveContext 的组合(未来可能还会加上 StreamingContext),所以在 SQLContext 和 HiveContext 上可用的 API 在 SparkSession 上同样是可以使用的。SparkSession 内部封装了 SparkContext,所以计算实际上是由 SparkContext 完成的。

在 PySpark 中,SparkContext 使用 Py4J 来启动一个 JVM 并创建一个 JavaSparkContext。默认情况下,PySpark 已经创建了一个名为 sc 的 SparkContext,并且在一个 JVM 进程中可以创建多个 SparkContext,但是只能有一个 active 级别的,因此,如果再创建一个新的 SparkContext 是不能正常使用的。

4. Spark 三种部署方式

Spark 支持以下三种不同类型的部署方式。

(1) Standalone(类似于 MapReduce1.0,slot 为资源分配单位)。

(2) Spark on Mesos(和 Spark 有“血缘”关系,能更好支持 Mesos)。

(3) Spark on YARN。

3.2 Spark 的工作机制

1. Spark 的基本架构

Spark 运行架构包括集群资源管理器(Cluster Manager)、运行作业任务的工作节点(Worker Node)、每个应用的任务控制节点(Driver)和每个工作节点上负责具体任务的执行进程(Executor),如图 3-4 所示。资源管理器可以自带或 Mesos 或 YARN。

与 Hadoop MapReduce 计算框架相比,Spark 所采用的 Executor 有两个优点。

(1) 利用多线程来执行具体的任务,减少任务的启动开销;

(2) Executor 中有一个 BlockManager 存储模块,会将内存和磁盘共同作为存储设备,有效减少 IO 开销。

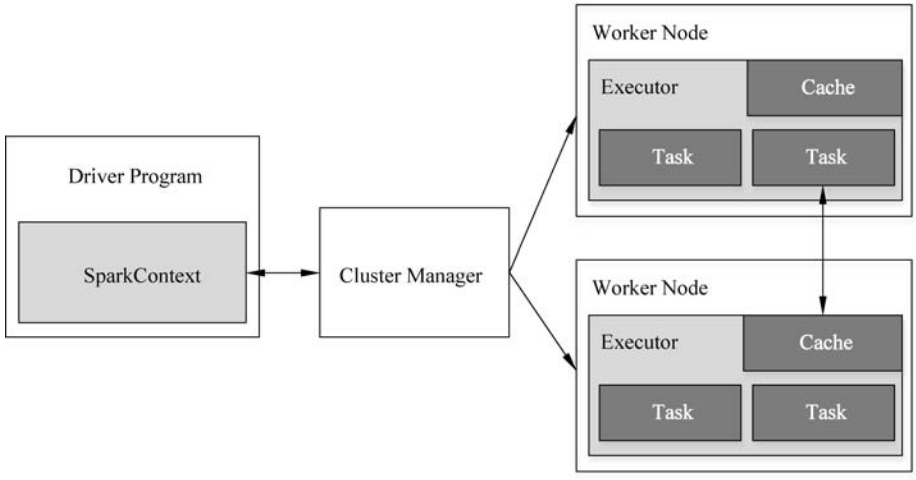

图 3-4 Spark 的运行架构图

一个 Application 由一个 Driver 和若干个 Job 构成,一个 Job 由多个 Stage 构成,一个 Stage 由多个没有 Shuffle 关系的 Task 组成,如图 3-5 所示。

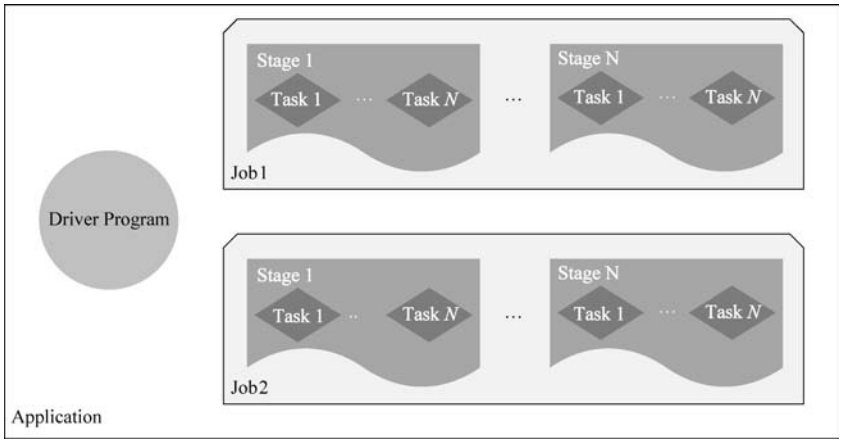

图 3-5 Application 的组成

当执行一个 Application 时，Driver 会向集群管理器申请资源，启动 Executor，并向 Executor 发送应用程序代码和文件。然后在 Executor 上执行 Task。运行结束后，执行结果则会返回给 Driver，或者写到 HDFS、其他数据库中。

2. Spark 的执行过程

Spark 的基本运行流程如图 3-6 所示。

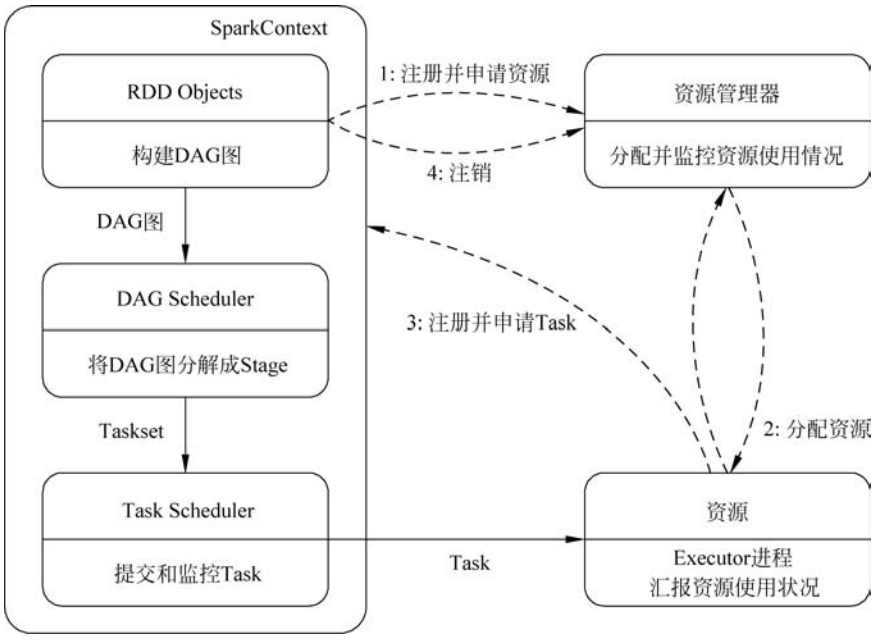

图 3-6　Spark 运行基本流程

(1) 首先为应用构建起基本的运行环境，即由 Driver 创建一个 SparkContext，进行资源的申请、任务的分配和监控。

(2) 资源管理器为 Executor 分配资源，并启动 Executor 进程。

(3) SparkContext 根据 RDD 的依赖关系构建 DAG 图；DAG 图提交给 DAGScheduler 解析成 Stage，然后把一个个 TaskSet 提交给底层调度器 TaskScheduler 处理；Executor 向 SparkContext 申请 Task；Task Scheduler 将 Task 发放给 Executor 运行，并提供应用程序代码。

(4) Task 在 Executor 上运行，把执行结果反馈给 TaskScheduler，然后反馈给 DAGScheduler，运行完毕后写入数据并释放所有资源。

总体而言，Spark 运行架构具有以下 3 个特点。

(1) 每个 Application 都有自己专属的 Executor 进程，并且该进程在 Application 运行期间一直驻留。Executor 进程以多线程的方式运行 Task。

(2) Spark 运行过程与资源管理器无关，只要能够获取 Executor 进程并保持通信即可。

(3) Task 采用了数据本地性和推测执行等优化机制。

3.3 弹性分布式数据集 RDD 基础

许多迭代式算法(如机器学习、图算法等)和交互式数据挖掘工具的共同之处是,不同计算阶段之间会重用中间结果。目前的 MapReduce 框架都是把中间结果写入到 HDFS 中,带来了大量的数据复制、磁盘 IO 和序列化开销。

RDD 就是为了解决这种问题而出现的,它提供了一个抽象的数据架构,从而不必担心底层数据的分布式特性,只需将具体的应用逻辑表达为一系列转换处理,不同 RDD 之间的转换操作形成依赖关系,可以实现管道化,避免中间数据存储。

Spark 的核心是 RDD(resilient distributed dataset),即弹性分布式数据集,是由 AMPLab 实验室提出的概念,属于一种分布式的内存系统数据集应用。

Spark 的主要优势来自 RDD 本身的特性,RDD 能与其他系统兼容,可以导入外部存储系统的数据集,例如 HDFS、HBase 或其他 Hadoop 数据源。

1. RDD 的概念

一个 RDD 就是一个分布式对象集合,本质上是一个只读的分区记录集合,每个 RDD 可分成多个分区,每个分区就是一个数据集片段,并且一个 RDD 的不同分区可以被保存到集群中不同的节点上,从而可以在集群中的不同节点上进行并行计算。

RDD 提供了一种高度受限的共享内存模型,即 RDD 是只读的记录分区的集合,不能直接修改,只能基于稳定的物理存储中的数据集创建 RDD,或者通过在其他 RDD 上执行确定的转换操作(如 map、join 和 group by)而创建得到新的 RDD。

RDD 是 Spark 的基石,也是 Spark 的灵魂。RDD 是弹性分布式数据集,是只读的分区记录集合。每个 RDD 有 5 个主要的属性。

(1) 一组分片(partition):数据集的最基本组成单位。

(2) 一个计算每个分片的函数:对于给定的数据集,需要做哪些计算。

(3) 依赖(Dependencies):RDD 的依赖关系,描述了 RDD 之间的 lineage。

(4) preferredLocations(可选):对于 data partition 的位置偏好。

(5) partitioner(可选):对于计算出来的数据结果如何分发。

2. RDD 的两种操作类型

RDD 提供了一组丰富的操作以支持常见的数据运算,有转换和动作两种操作方式。

(1) 转换(Transformations):返回 RDD,基于现有的数据集创建一个新的数据集,如 map、filter、groupBy、join 等。

(2) 动作(Actions):在数据集上进行运算,返回计算值,而不是一个 RDD,如 count、collect、save 等。RDD 通过“转换”运算得到新的 RDD,原 RDD 不受影响。但 Spark 会延迟这个转换的发生时间点,不会马上执行,而是等到执行了 Action 之后才会基于所有的 RDD 关系来执行转换。

RDD 提供的转换接口都非常简单,都是类似 map、filter、groupBy、join 等粗粒度的数据转换操作,而不是针对某个数据项的细粒度修改(不适合网页爬虫)。

1）RDD 转换操作

下面以 RDD 包含{1，2，3，3}为例，说明基本的转换结果。基本的 RDD 转换函数见表 3-2。

表 3-2　基本的 RDD 转换函数

函数名	功　能	例　子	结　果
map()	通过自定义函数进行映射。对每个元素应用函数	rdd. map(x=> x+1)	{2,3,4,4}
flatMap()	先映射(map)，再把元素合并为一个集合。常用来抽取单词	rdd. flatMap(x=> x. to(3))	{1,2,3,2,3,3,3}
filter()	对元素过滤，保留符合条件的元素	rdd. filter(x=> x!=1)	{2,3,3}
distinct()	去重	rdd. distinct()	{1,2,3}
sample (withReplacement, fraction,[seed])	根据给定的随机种子 seed，随机抽样出比例为 fraction 的数据	list = numpy. arange(1,100,2) listRDD = sc. parallelize(list) sampleRDD = listRDD. sample(0,0. 2). collect()	[25, 33, 41, 53, 63, 65, 69, 77, 87, 95]
groupByKey()	应用于(K,V)键值对的数据集时，返回一个新的(K,Iterable < V >)形式的数据集。Key 相同的值被分为一组	kvd. groupByKey ( ). map (lambda x: (x [0], list (x [1]))). collect()	[(1, [2]), (3, [4, 6]), (5, [6])]
reduceByKey (func)	应用于(K,V)键值对的数据集时，返回一个新的(K,V)形式的数据集，其中的每个值是将每个 key 传递到函数 func 中进行聚合	kvd. reduceByKey(lambda x,y: x+y). collect()	[(1, 2), (3, 10), (5, 6)]
sortByKey	通过 Key 值对 KV 对数据集排序	kvd. sortByKey (ascending = False). collect()	[(5, 6), (3, 4), (3, 6), (1, 2)]
join	对两个 RDD 进行 cogroup、笛卡尔积、展平操作，(K,V)和(K,W)转换为(K,(V,W))	x = sc. parallelize([("a", 1), ("b", 4)]) y = sc. parallelize([("a", 2), ("a", 3)]) sorted(x. join(y). collect())	[('a', (1, 2)), ('a', (1, 3))]

其他转换操作还包括 mapPartitions、mapPartitionsWithIndex、intersection、aggregateByKey、cartesian、pipe、coalesce、repartition、repartitionAndSortWithinPartitions 等。

2）比较 map()和 flatMap()

flatMap()对每个输入元素输出多个输出元素。flat 是压扁的意思，即将 RDD 中元素压扁后返回一个新的 RDD。其工作原理如图 3-7 所示。

3）RDD 的动作

主要的动作见表 3-3。

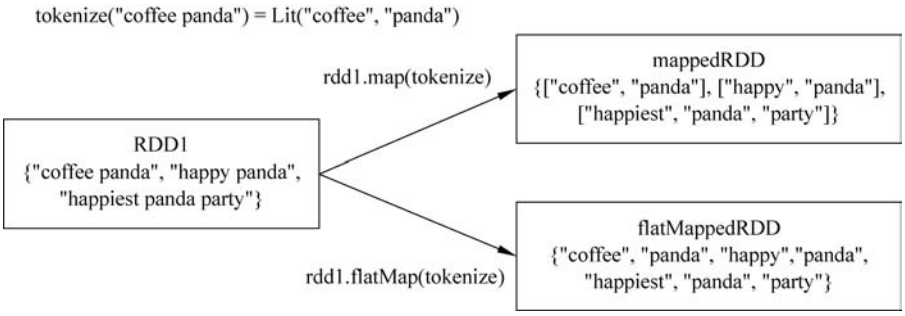

图 3-7　map()和 flatMap()函数的比较

表 3-3　RDD 的动作操作示例

函数名	功　能	例　子	结　果
collect()	相当于 toArray，将分布式的 RDD 返回为一个单机的 Array 数组	rdd. collect()	{1,2,3,3}
count()	返回 RDD 中元素的个数	rdd. count()	4
countByValue()	返回一个 map，表示唯一元素出现的个数	rdd. countByValue()	{(1, 1), (2, 1), (3,2)}
take(num)	返回数据集中前 num 个元素形成的数组	rdd. take(2)	{1,2}
top(num)	返回前几个元素	rdd. top(2)	{3,3}
takeOrdered (num)(ordering)	返回基于提供的排序算法的前几个元素	rdd. takeOrdered(2)(myOrdering)	{3,3}
takeSample (withReplacement, num,[seed])	取样例	rdd. takeSample(false,1)	不确定
reduce(func)	合并 RDD 中元素	rdd. reduce((x,y)=> x+y)	9
fold(zero)(func)	与 reduce()相似提供 zero value	rdd. fold(0)((x,y)=> x+y)	9
aggregate (zeroValue)(seqOp, combOp)	与 fold()相似，返回不同类型	rdd. aggregate((0,0))((x,y)=> (x. _1+y, x. _2+1),(x,y)=> (x. _1+y. _1,x. _2+y. _2))	(9,4)
foreach(func)	对数据集中每个元素使用 func()函数，无返回	rdd. foreach(func)	什么也没有
first	返回数据集中第一个元素，与 take (1)类似	rdd. first()	1
countByKey	只用于 KV 类型 RDD，返回每个 Key 的个数	kvd. countByKey()	defaultdict(int, {3: 2, 5: 1, 1: 1})
saveAsTextFile	保存数据到文本文件	rdd. saveAsTextFile ("file:///db/spark")将文件保存到本地文件系统；saveAsTextFile (" hdfs://db/spark/") //保存到 HDFS	文本文件
saveAsSequenceFile	保存数据为序列化文件	用法同 saveAsTextFile	序列化文件
saveAsObjectFile	保存数据为对象文件	用法同 saveAsTextFile	对象文件

3. 集合运算

RDDs 支持数学集合的计算，如并集、交集计算。注意：进行计算的 RDDs 应该是相同类型。

下面以两个 RDD 的 Transformations 为例：一个 RDD 包含{1，2，3}，另一个 RDD 包含{3，4，5}，见表 3-4。

表 3-4　集合运算示例

函数名	功　能	例　子	结　果
union()	并集	rdd. union(other)	{1，2，3，3，4，5}
intersection()	交集	rdd. intersection(other)	{3}
subtract()	取存在第一个 RDD、而不存在第二个 RDD 的元素（使用场景，机器学习中，移除训练集）	rdd. subtract(other)	{1，2}
cartesian()	笛卡尔积	rdd. cartesian(other)	{(1，3)，(1，4)，…(3，5)}

cartesian()操作非常耗时，其技术原理如图 3-8 所示。

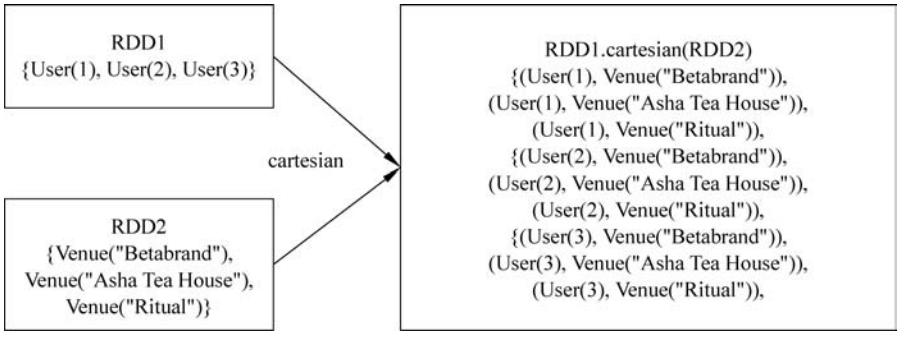

图 3-8　RDD 的 catesian 运算

4. RDD 典型的执行过程

表面上 RDD 的功能很受限、不够强大，实际上 RDD 已经被实践证明可以高效地表达许多框架的编程模型（如 MapReduce、SQL、Pregel）。Spark 用 Scala 语言实现了 RDD 的 API，程序员可以通过调用 API 实现对 RDD 的各种操作。其基本执行过程如图 3-9 所示。

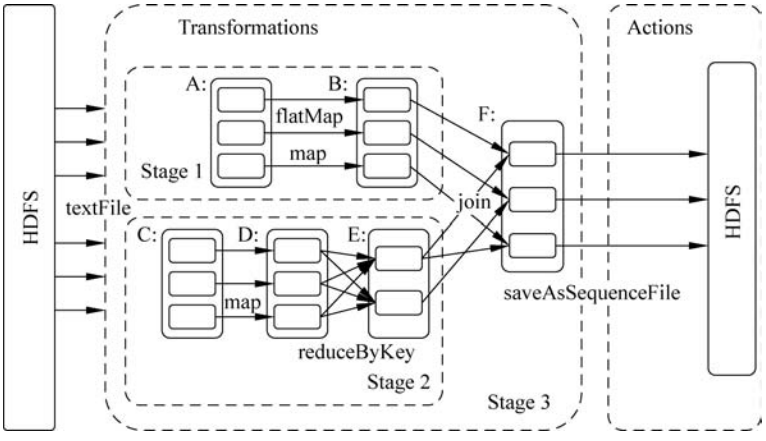

图 3-9　RDD 的执行过程

(1) RDD 读入外部数据源进行创建。

(2) RDD 经过一系列的转换(Transformation)操作,每一次都会产生不同的 RDD,供给下一个转换操作使用。

(3) 最后一个 RDD 经过“动作”操作进行转换,并输出到外部数据源。

这一系列处理称为一个 lineage(血缘关系),即 DAG 拓扑排序的结果。这样操作的优点是惰性调用、管道化、避免同步等待、不需要保存中间结果、每次操作变得简单。

5. RDD 之间的依赖关系

RDD 只能基于稳定物理存储中的数据集和其他已有的 RDD 上执行确定性操作来创建。能从其他 RDD 通过确定操作创建新的 RDD 的原因是 RDD 含有从其他 RDD 衍生(即计算)出本 RDD 的相关信息(即 lineage)。Dependency 代表了 RDD 之间的依赖关系,即血缘(Lineage),分为窄依赖和宽依赖。

(1) 窄依赖:一个父 RDD 最多被一个子 RDD 用在一个集群节点上管道式执行。例如 map、filter、union 等。

(2) 宽依赖:指子 RDD 的分区依赖于父 RDD 的所有分区,这是因为 shuffle 类操作要求所有父分区可用。例如 groupByKey、reduceByKey、sort、partitionBy 等。

注意:一个 RDD 对不同的父节点可能有不同的依赖方式,可能对父节点 1 是宽依赖,对父节点 2 是窄依赖。

窄依赖和宽依赖的比较如图 3-10 所示。

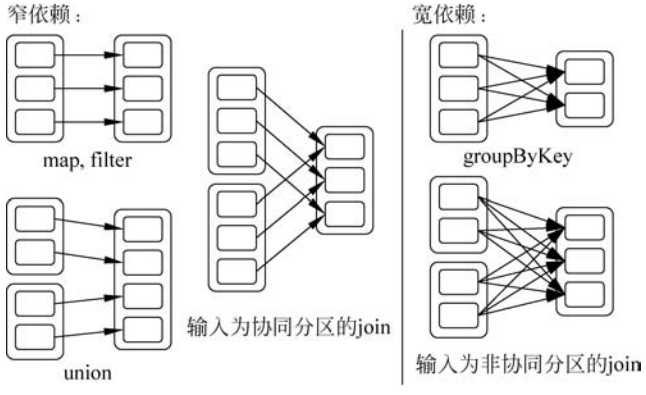

图 3-10 窄依赖和宽依赖的比较

3.4 RDD 的 Python 程序设计

本节利用 Python 语言,采用 Jupyter Notebook 环境开展 RDD 的操作。

1. 基本 RDD 的转换运算

1) 创建 intRDD

创建 intRDD 最简单的方式就是使用 SparkContext 的 parallelize 方法,命令如下:

```
intRDD=sc.parallelize([3,1,2,5,5])
```

这是一个转换运算,不会立即执行。通过执行 collect()的动作运算,就能够立即完成。