学习Spark框架，就要先熟悉其编程核心RDD（弹性分布式数据集），如同学习Hadoop就要熟悉MapReduce编程模式一样。最近几年好多声音在说Spark将要代替Hadoop，其实经过对Spark长时间学习之后感觉，Spark要替换的不是Hadoop而是MapReduce。

下面针对Spark的核心RDD进行入门级的介绍，Spark源码RDD.scala中描述RDD的特性为：

分布式环境中分区的集合

一个函数计算于每个分区上

一组对于其他RDD的依赖

（可选）Key-value型的RDD是根据哈希分区

（可选）每一分片的优先计算位置

但我觉得还应该增加两点：

提供高效的容错机制，失败可以自动恢复重建

提供内存式计算

针对RDD的特性，该博文分为三部分，分别介绍“什么是RDD”、“RDD之间的依赖”、“RDD的高可靠性”。

RDD学习之一 ——“什么是RDD”

## 1. 什么是RDD？

RDD是Resilient Distributed Datasets， 弹性 分布式 数据集。首先从概念上来理解RDD。

### 1.1 数据集

RDD是不可变的、只读的分区的集合，一个RDD包括了多个分区，每个分区就是一个数据片段。对于分区与RDD的关系如图：

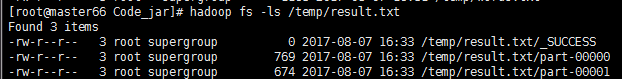


为了更形象的理解，这里用一个实际数据RDD来介绍。

**val** file = sc.textFile("hdfs://master66:8020/temp/words.txt")

file.saveAsTextFile（"hdfs://master66:8020/temp/words.txt"）

这里的file是一个RDD，是从本地磁盘或者HDFS存储的txt文本得来的，该文本中存储了一篇英文短文。 如果我直接对file这个RDD进行存储，得到的结果是一个文件夹，包括了两个分区。如图：

Spark内部会有相应的分区机制对结果RDD进行分区。

**为什么要进行分区？**

如果针对一项任务进行并行操作，需要该任务同时作用于我们的数据集，那么就要采用分区来实现并行化。举个简单的例子：一个人数棋子个数是串行化任务，我们想并行化该任务，需要让多个人来一起数棋子，这就需要把棋子分成N份，每人一份分别数各自的一堆棋子，最终把各自数得的结果进行汇总。对应的RDD就是原先的棋子，一个个的分区就是被分开的一堆堆棋子。

其实分区的数量会对并行计算的粒度有一定的决定性影响。（其他因素也会有决定性影响，如task数量、executor数量等）

默认的对RDD的分区的数量是由参数spark.default.parallelism决定的，该参数也决定着task的个数。如果不设定该参数，则系统根据slave worker的数量来决定分区数量。

### 1.2 如何做到分布式

所谓的分布式数据集是指数据分布在一个集群的多台机器上，Spark的RDD实际上是对分布式内存的一种抽象，意思是指： 一个RDD的多个分区存储于集群内存中，用户在编写Spark程序时可以像操作本地数据集一样来操作分布式环境下的RDD，从而感觉不到RDD是被分配在不同节点上。RDD可以存储于HDFS等分布式文件系统中来实现分布式存储。

下图是RDD在集群中的存储分布：



在Spark集群中，每个RDD以Block的形式存储于多台机器中，每个Executor进程会自动一个BlockManagerSlave来管理部分Block。

而Block元数据有Driver上的RDD BlockManagerMaster来管理。

BlockManagerSlave每生成一个block需要向BlockManagerMaster注册该Block，BlockManagerMaster管理RDD与Block的关系，当RDD不需要存储的时候则由BlockManagerMaster向BlockManagerSlave发送指令来删除相应的Block。（类似于HDFS的NameNode和DataNodes）

### 1.3那么何为弹性？

在关于弹性方面，官方论文中也没有提及弹性是指哪方面的特性，查看资料发现更多的声音是指RDD的容错弹性，即分区损坏了可以自动恢复。不过个人感觉在弹性方面不仅指其容错弹性，还包括分片以及存储方面的弹性，总体可以归结为一下三点：

**一是指RDD数据分区的自由弹性。**

Spark提供了Partitioner接口，内部实现了HashPartitioner和RangePartitioner两种分区方式，但同时也允许用户定义自己的partitioner，只需要定义一个分区类来继承Spark的Partitioner抽象类，并实现其中的numPartitions、getPartition、equals方法即可。如下：

//Spark的抽象Partitioner类

|  |
| --- |
| abstract class Partitioner extends Serializable { |

|  |  |
| --- | --- |
|  | def numPartitions: Int |

|  |  |
| --- | --- |
|  | def getPartition(key: Any): Int |

|  |  |
| --- | --- |
|  | } |

这样在对RDD分区时调用partitionBy方法并传入自定义的分区类的实例对象就可以实现自由分区了。如下：

rdd.partitionBy(new PartitionClass())

**二是指RDD在存储方面可以自动地在内存和磁盘上进行数据存储的转换。**

RDD操作的中间结果默认存内存，内存不足则部分分区不会被缓存，当需要这些分区时需要重新计算。

Spark提供了两种RDD存储方式的API：persist和cache。下面是两种方式的源码：

def cache(): this.type = persist()

def persist(): this.type = persist(StorageLevel.MEMORY\_ONLY)

// newLevel 表示RDD缓存的级别

def persist（newLevel： StorageLevel）: this.type = {}

阅读源码可以看出cache（）和persist（）的区别。 cache()实际上是调用的了无参数的persist方法，而无参数的persist调用了MEMORY\_ONLY级别的persist。

**三是指RDD的高效容错机制（下文“RDD提供的高可靠性”有详细介绍）**

RDD提供两种容错机制：Lineage和checkPoint， 当有分区出错时可以通过这两种方式从其他分区中恢复，并进行重新计算。

## 2. RDD之间的依赖

介绍RDD之间的依赖前需要明确RDD之间的操作。

2.1 RDD的生成

RDD作为只读的分区记录集合，只能由持久化存储的数据或者由其他RDD转换而来。

最常见的从持久化存储数据中创建RDD过程如下：

rdd = sc.textFile(“path/file”) //从本地或HDFS等文件系统中输入创建

rdd = sc.parallelize( array ) //从集合创建

其他的针对RDD的操作主要分为两类：Transformation 和 action。

transformation是一种惰性操作，只记录了从一个RDD到另一个RDD的转换操作，并没有实际的去计算（我借用虚拟内存的延时分配来理解，程序申请内存时只是为其分配虚拟内存空间，只有当该程序真正去使用内存时才会为其分配物理内存），只有当action操作发生时，才会真正地执行前面的转换操作。这样对于RDD的一系列的transformation和action操作的记录会构成一个操作图，叫linage， 如图：

**val** result = file.flatMap(\_.split(" ")).map((\_, 1)).reduceByKey(\_+\_)

该程序的linage为：



这是简单的wordcount程序（神奇的scala，一行代码就搞定），这一行代码首先对RDD进行flatMap转换，通过对每一行的数据进行根据空格切分，得到一行一个单词；之后对每一行转换为（word，1）将每一个单词标记为出现1次； 最后reduceByKey将（k，v）对中具有相同key的value进行相加。

前面的两次map操作是转换操作，只会记录，后面的reduceByKey是action操作，会触发RDD的操作执行，此时前面的两次map才会真正执行。

RDD的transformation和action操作举例：

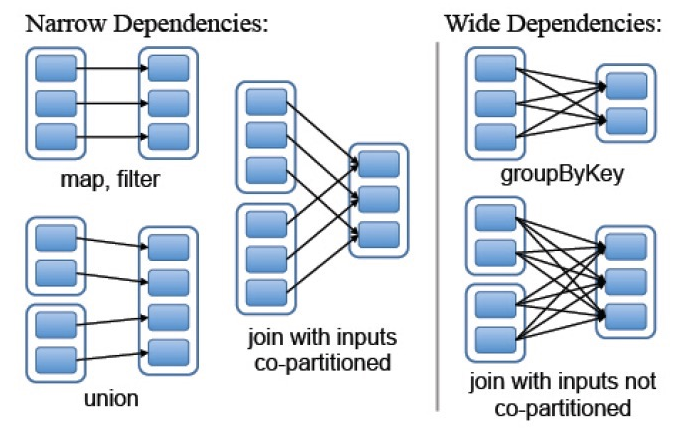
|  |  |
| --- | --- |
| **transformation（惰性操作）** | **action** |
| map | collect |
| flatMap | count |
| filter | take（获取前N条记录） |
| distinct | top（获取排序后的前N条记录） |
| sample | takeOrdered |
| union | reduce |
| intersection | reduceByKey |
| subtract | foreach |
| groupBy | save |
| join |  |
| Cartesian |  |

map与flatMap区别：两者都是对每个元素进行操作，只是传递给flatMap的函数的返回类型必须是可迭代的，如list。

2.2 RDD之间的依赖

RDD的分区之间存在两种依赖关系：宽依赖和窄依赖。

下面是RDDs经典的依赖图：



## 3.RDD提供的高可靠性