Spark 计算模型

1. 课程目标

1.1. 熟练使用 RDD 的算子完成计算

1.2. 掌握 RDD 的原理

2. 弹性分布式数据集 RDD

2.1. RDD 概述

2.1.1. 什么是 RDD

RDD（Resilient Distributed Dataset）叫做分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD 具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

2.1.2. RDD 的属性

```
* Internally, each RDD is characterized by five main properties:
*
*  - A list of partitions
*  - A function for computing each split
*  - A list of dependencies on other RDDs
*  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
*  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
*    an HDFS file)
```

1）一组分片（Partition），即数据集的基本组成单位。对于 RDD 来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建 RDD 时指定 RDD 的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的 CPU Core 的数目。

2）一个计算每个分区的函数。Spark 中 RDD 的计算是以分片为单位的，每个 RDD 都会实现 compute 函数以达到这个目的。compute 函数会对迭代器进行复合，不需要保存每次计算的结果。

3）RDD 之间的依赖关系。RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。

4）一个 Partitioner，即 RDD 的分片函数。当前 Spark 中实现了两种类型的分片函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于于 key-value 的 RDD，才会有 Partitioner，非 key-value 的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分片数量，也决定了 parent RDD Shuffle 输出时的分片数量。

5）一个列表，存储存取每个 Partition 的优先位置（preferred location）。对于一个 HDFS 文件来说，这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

2.2. 创建 RDD

1）由一个已经存在的 Scala 集合创建。

```
val rdd1 = sc.parallelize(Array(1,2,3,4,5,6,7,8))
```

2）由外部存储系统的数据集创建，包括本地的文件系统，还有所有 Hadoop 支持的数据集，比如 HDFS、Cassandra、HBase 等

```
val rdd2 = sc.textFile("hdfs://node1.itcast.cn:9000/words.txt")
```

2.3. RDD 编程 API

2.3.1. Transformation

RDD 中的所有转换都是延迟加载的，也就是说，它们并不会直接计算结果。相反的，它们只是记住这些应用到基础数据集（例如一个文件）上的转换动作。只有当发生一个要求返回结果给 Driver 的动作时，这些转换才会真正运行。这种设计让 Spark 更加有效率地运行。

常用的 Transformation：

转换	含义
map(*func*)	返回一个新的 RDD，该 RDD 由每一个输入元素经过 func 函数转换后组成
filter(*func*)	返回一个新的 RDD，该 RDD 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（所以 func 应该返回一个序列，而不是单一元素）

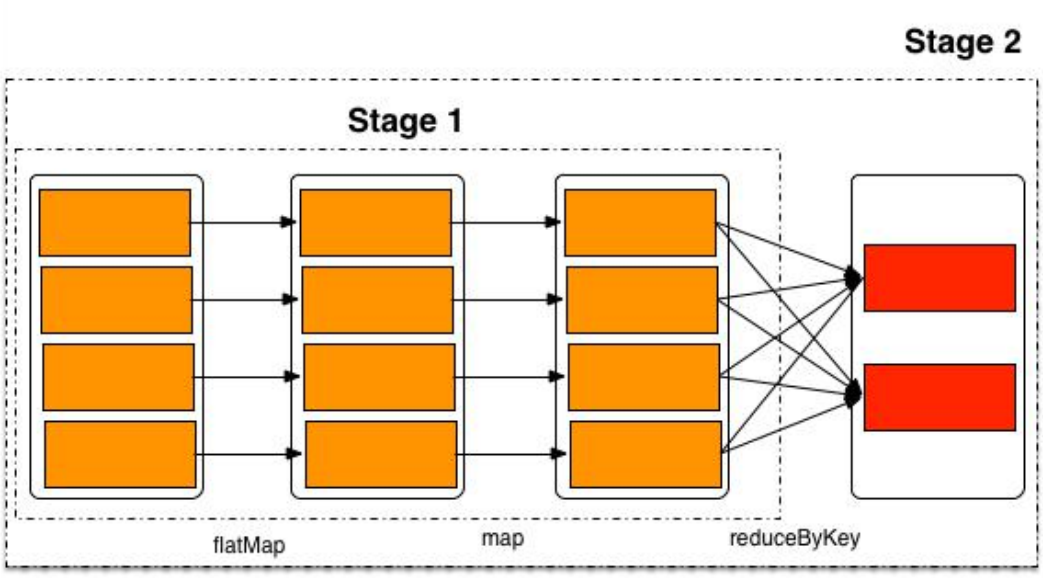
mapPartitions(*func*)	类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]
mapPartitionsWithIndex(*func*)	类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 (Int, Interator[T]) => Iterator[U]
sample(*withReplacement, fraction, seed*)	根据 fraction 指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed 用于指定随机数生成器种子
union(*otherDataset*)	对源 RDD 和参数 RDD 求并集后返回一个新的 RDD
intersection(*otherDataset*)	对源 RDD 和参数 RDD 求交集后返回一个新的 RDD
distinct([*numTasks*]))	对源 RDD 进行去重后返回一个新的 RDD
groupByKey([*numTasks*])	在一个(K,V)的 RDD 上调用，返回一个(K, Iterator[V])的 RDD
reduceByKey(*func*, [*numTasks*])	在一个(K,V)的 RDD 上调用，返回一个(K,V)的 RDD，使用指定的 reduce 函数，将相同 key 的值聚合到一起，与 groupByKey 类似，reduce 任务的个数可以通过第二个可选的参数来设置
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	
sortByKey([*ascending*], [*numTasks*])	在一个(K,V)的 RDD 上调用，K 必须实现 Ordered 接口，返回一个按照 key 进行排序的(K,V)的 RDD
sortBy(*func*,[*ascending*], [*numTasks*])	与 sortByKey 类似，但是更灵活
join(*otherDataset*, [*numTasks*])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素对在一起的(K,(V,W))的 RDD
cogroup(*otherDataset*, [*numTasks*])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD
cartesian(*otherDataset*)	笛卡尔积
pipe(*command*, [*envVars*])	
coalesce(*numPartitions*)	
repartition(*numPartitions*)	
repartitionAndSortWithinPartitions(*partitioner*)	

2.3.2. Action

动作	含义
reduce(*func*)	通过 func 函数聚集 RDD 中的所有元素，这个功能必须是课交换且可并联的
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
count()	返回 RDD 的元素个数
first()	返回 RDD 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
takeSample(*withReplacement,num*, [*seed*])	返回一个数组，该数组由从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，seed 用于指定随机数生成器种子
takeOrdered(*n*, *[ordering]*)	
saveAsTextFile(*path*)	将数据集的元素以 textfile 的形式保存到 HDFS 文件系统或者其他支

	持的文件系统，对于每个元素，Spark 将会调用 toString 方法，将它装换为文件中的文本
saveAsSequenceFile(*path*)	将数据集中的元素以 Hadoop sequencefile 的格式保存到指定的目录下，可以使 HDFS 或者其他 Hadoop 支持的文件系统。
saveAsObjectFile(*path*)	
countByKey()	针对(K,V)类型的 RDD，返回一个(K,Int)的 map，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，运行函数 func 进行更新。

2.3.3. WordCount 中的 RDD



2.3.4. 练习

启动 spark-shell

```
/usr/local/spark-1.5.2-bin-hadoop2.6/bin/spark-shell --master spark://node1.itcast.cn:7077
```

练习 1：

```
//通过并行化生成 rdd
val rdd1 = sc.parallelize(List(5, 6, 4, 7, 3, 8, 2, 9, 1, 10))
//对 rdd1 里的每一个元素乘 2 然后排序
val rdd2 = rdd1.map(_ * 2).sortBy(x => x, true)
//过滤出大于等于十的元素
val rdd3 = rdd2.filter(_ >= 10)
//将元素以数组的方式在客户端显示
rdd3.collect
```

练习 2：

```
val rdd1 = sc.parallelize(Array("a b c", "d e f", "h i j"))
//将 rdd1 里面的每一个元素先切分在压平
val rdd2 = rdd1.flatMap(_.split(' '))
rdd2.collect
```

练习 3：

```
val rdd1 = sc.parallelize(List(5, 6, 4, 3))
val rdd2 = sc.parallelize(List(1, 2, 3, 4))
//求并集
val rdd3 = rdd1.union(rdd2)
//求交集
val rdd4 = rdd1.intersection(rdd2)
//去重
rdd3.distinct.collect
rdd4.collect
```

练习 4：

```
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)))
//求 jion
val rdd3 = rdd1.join(rdd2)
rdd3.collect
//求并集
val rdd4 = rdd1 union rdd2
//按 key 进行分组
rdd4.groupByKey
rdd4.collect
```

练习 5：

```
val rdd1 = sc.parallelize(List(("tom", 1), ("tom", 2), ("jerry", 3), ("kitty", 2)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)))
//cogroup
val rdd3 = rdd1.cogroup(rdd2)
//注意 cogroup 与 groupByKey 的区别
rdd3.collect
```

练习 6：

```
val rdd1 = sc.parallelize(List(1, 2, 3, 4, 5))
//reduce 聚合
val rdd2 = rdd1.reduce(_ + _)
rdd2.collect
```

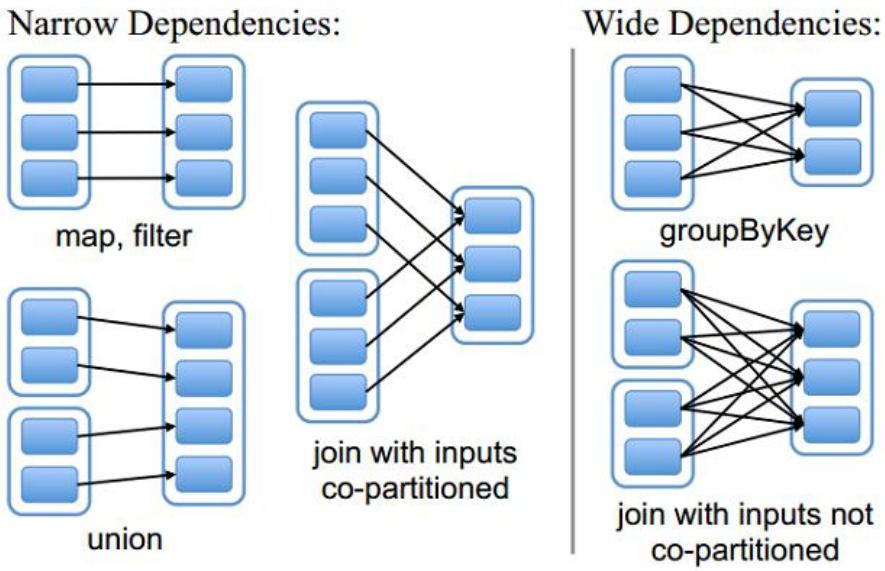
练习 7：

```
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2),  ("shuke", 1)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 3), ("shuke", 2), ("kitty", 5)))
val rdd3 = rdd1.union(rdd2)
//按 key 进行聚合
val rdd4 = rdd3.reduceByKey(_ + _)
rdd4.collect
//按 value 的降序排序
val rdd5 = rdd4.map(t => (t._2, t._1)).sortByKey(false).map(t => (t._2, t._1))
rdd5.collect

//想要了解更多，访问下面的地址
http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html
```

2.4. RDD 的依赖关系

RDD 和它依赖的父 RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。



2.4.1. 窄依赖

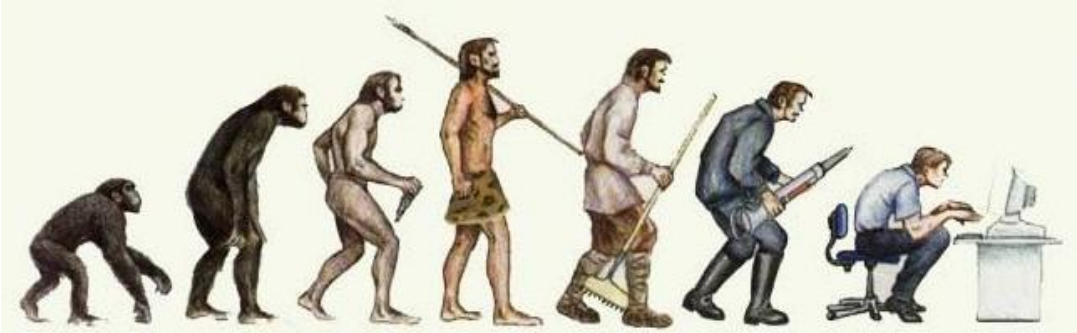
窄依赖指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用
总结：窄依赖我们形象的比喻为**独生子女**

2.4.2. 宽依赖

宽依赖指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition
总结：窄依赖我们形象的比喻为**超生**

2.4.3. Lineage

RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列 Lineage（即血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。



2.5. RDD 的缓存

Spark 速度非常快的原因之一，就是在不同操作中可以在内存中持久化或缓存个数据集。当持久化某个 RDD 后，每一个节点都将把计算的分片结果保存在内存中，并在对此 RDD 或衍生出的 RDD 进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD 相关的持久化和缓存，是 Spark 最重要的特征之一。可以说，缓存是 Spark 构建迭代式算法和快速交互式查询的关键。

2.5.1. RDD 缓存方式

RDD 通过 persist 方法或 cache 方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。

```
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)

/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def cache(): this.type = persist()
```

通过查看源码发现 cache 最终也是调用了 persist 方法，默认的存储级别都是仅在内存存储一份，Spark 的存储级别还有好多种，存储级别在 object StorageLevel 中定义的。

```
object StorageLevel {
  val NONE = new StorageLevel(false, false, false, false)
  val DISK_ONLY = new StorageLevel(true, false, false, false)
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)
  val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
  val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
  val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
  val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
  val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
  val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
  val OFF_HEAP = new StorageLevel(false, false, true, false)
```

缓存有可能丢失，或者存储存储于内存的数据由于内存不足而被删除，RDD 的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于 RDD 的一系列转换，丢失的数据会被重算，由于 RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

2.6. DAG 的生成

DAG(Directed Acyclic Graph)叫做有向无环图，原始的 RDD 通过一系列的转换就就形成了 DAG，根据 RDD 之间的依赖关系的不同将 DAG 划分成不同的 Stage，对于窄依赖，partition 的转换处理在 Stage 中完成计算。对于宽依赖，由于有 Shuffle 的存在，只能在 parent RDD 处理完成后，才能开始接下来的计算，因此宽依赖是划分 Stage 的依据。