第三节课

RDD是由不同的partition组成的，我们所进行的transformation和action是在partition上面进行的；
而在storage模块内部，RDD又被视为由不同的block组成，对于RDD的存取是以block为单位进行的，本质上partition和block是等价的，只是看待的角度不同。

在Spark storage模块中中存取数据的最小单位是block，所有的操作都是以block为单位进行的。 *

数据块（Block） 简单介绍Spark存储管理模块中所管理的几种主要数据块。

RDD数据块：用来标识所缓存的RDD数据。
Shuffle数据块：用来标识持久化的Shuffle数据。
广播变量数据块：用来标识所存储的广播变量数据。
任务返回结果数据块：用来标识存储在存储模块内部的任务返回结果。
流式数据块：只用在Spark Streaming中，用来标识所接收到的流式数据块。

RDD分区和数据块的关系

RDD上的所有运算都是基于分区的。
分区是一个逻辑上的概念，而数据块是物理上的数据实体。

- 在Spark中，分区和数据块是一一对应的，一个RDD钟的一个分区对应着存储管理模块中的一个数据块。
- spark为每一个RDD在其内部维护了独立的ID号，同时，对于每一个分区也有一个独立的索引号，“ID号+索引号”，就能唯一确定分区。

内存缓存

以默认或基于内存的持久化方式缓存RDD时，RDD中的每一个分区所对应的数据块是会被存储模块中的内存管理所管理的。内存缓存在其内部维护了一个以数据块名称为键，块内容为值得哈希表。
当内存已经达到所设置的阈值时应该如何处理（阈值通过spark.storage.memoryFraction设置）：

- 丢弃一些数据块，数据块持久化选项不包含磁盘缓存
- 将一些数据块存储到磁盘上，数据块持久化选项包含磁盘缓存

磁盘缓存

磁盘缓存的数据块会被放到磁盘中的特定目录下。
在磁盘缓存中，一个数据块对应着文件系统中的一个文件，文件名和块名称的映射关系是通过哈希算法计算所得的。建立了块和文件之间的对应关系，存取块的内容就变成了写入和读取相应的文件了。

持久化选项

当我们需要将某一个RDD持久化时，我们可以调用RDD所提供的persist()或者cache()函数进行操作。

- 当用户使用persist()函数进行持久化时，需要选择一种持久化方式，这种方式决定了RDD最后的持久化策略。
- 如果用户调用cache()函数进行持久化，则默认MEMORY_ONLY(内存)的方式持久化。

这些持久化策略都是由类org.apache.spark.StorageLevel决定的。

Spark的持久化级别

持久化级别	含义解释
MEMORY_ONLY	使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是策略，使用cache()方法时，实际就是使用的这种持久化策略。
MEMORY_AND_DISK	使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。
MEMORY_ONLY_SER	基本含义同MEMORY_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
MEMORY_AND_DISK_SER	基本含义同MEMORY_AND_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
DISK_ONLY	使用未序列化的Java对象格式，将数据全部写入磁盘文件中。
MEMORY_ONLY_2, MEMORY_AND_DISK_2, 等等.	对于上述任意一种持久化策略，如果加上后缀_2，代表的是将每个持久化的数据，都复制一份副本，并其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将头处重新计算一遍了。

如何选择不同的持久化选项

持久化方式选择的建议：

- 如果使用默认的持久化方式（MEMORY_ONLY）完全能够缓存RDD，那么久无需选择其他的持久化方式，因为这是CPU利用率最高的一种方式，能够确保程序跑得尽可能快。
- 如果默认的持久化方式无法完全缓存所需的RDD，可以尝试使用MEMEMORY_ONLY_SER这种持久化方式，并选用更快速的序列化方式，这能够搞笑的使用内存，同时确保程序跑得相对较快。
- 除非对RDD的重算带来的开销比缓存到磁盘来得到，一般情况下不要将RDD缓存到磁盘上，通常情况下，对于某一分区的重算会比从磁盘中读取要快。
- 如果想要确保快速的错误恢复机制，应尽可能选用带有备份的持久化方式。虽然前面提到过所有的持久化方式都可以通过重算丢失数据从错误中恢复，但是备份可以使得任务继续进行而无需等待丢失分区的重新计算。

Shuffle数据持久化

与RDD持久化的不同之处在于：

- 首先Shuffle数据块必须是在磁盘上进行缓存的，不能在内存中缓存
- 其次，在RDD基于磁盘的持久化中，每一个数据块对应着一个文件，而在Shuffle数据块的持久化中，Shuffle数据块表示的只是逻辑上的概念，不同的实现方式可以决定Shuffle数据块的不同存储方式。

在Spark存储管理模块中，Shuffle数据块有两种存储方式：

- 一种是将Shuffle数据块映射成文件，这是默认方式
- 另一种是将Shuffle数据块映射成文件中的一段，这种方式需要将spark.shuffle.consolidateFiles设置为true。

默认方式会产生大量文件，因此有了第二种实现方式，将分时运行的Map任务所产生的Shuffle数据块合并到同一个文件中，减少Shuffle文件的总数。

共享变量

在编写spark应用程序时，为了加速对一些小块数据的读取，我们往往希望这些数据在所有节点上都有一份拷贝，每个任务都能从本节点的拷贝中读取数据而无需通过远程传输获取数据——通过广播变量实现。

广播变量允许程序员将一个只读的变量缓存在每台机器上，而不用在任务之间传递变量。

只有driver端可以读取广播变量。

广播变量可被用于有效地给每个节点一个大输入数据集的副本。Spark还尝试使用高效地广播算法来分发变量，进而减少通信的开销。

广播变量是由存储管理模块进行管理的。

广播数据块是以MEMORY_AND_DISK的持久化方式存入本节点的存储管理模块中的。

通过设置过期清理机制，spark内部会清理过期的广播变量。

累加器是仅仅被相关操作累加的变量，因此可以在并行中被有效地支持。它可以被用来实现计数器和总和。Spark原生地只支持数字类型的累加器，编程者可以添加新类型的支持。如果创建累加器时指定了名字，可以在Spark的UI界面看到。这有利于理解每个执行阶段的进程。

聚合调优
join操作会发生shuffle，在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。
两个RDD发生join时，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。
如果数据比例不均衡的情况下，会发生任务倾斜导致任务长期等待，甚至程序崩溃退出。
可以利用广播变量将小数据集广播后发送到每个work结点进行join。

UI监控

应用监控UI以网页形式提供给最终用户，spark提供了两类形式的UI监控，分为实时UI管理和历史UI管理。
实时UI管理
实时UI管理是指在spark程序运行过程中，可以通过UI界面刷新查看即时的作业信息。
Spark程序的启动，意味着需要创建一个SparkContext，同时也启动了一个Web服务。默认情况下监听4040端口，浏览这个Spark程序的UI可以简单打开http://:4040链接。如果在一台机器上有多个SparkContext启动，那么监听的端口顺延。
Spark程序的实时监控界面大致包含下表中的四部分内容，从各个维度监控spark的作业运行情况。
UI监控信息分类

|Stage |
从Spark作业调度的角度，展示了调度阶段和任务的完成状态|
|Storage|
从Spark存储系统的角度，展示了Spark中RDD的存储状态|
|Environment|
从Spark运行环境的角度，展示了Spark及系统环境参数|
|Executor|
从Spark执行的角度，展示了每个执行体运行任务的状况|

代码

共享变量

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

import scala.collection.{Iterator, Map}
import scala.collection.mutable.ArrayBuffer

/**
  * Created by ding on 2017/12/23.
  */
object SharedVariables {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("july_online").setMaster("local[2]")
    val sc = new SparkContext(conf)
```

```
    sc.setLogLevel("WARN")
    val testRDD = sc.parallelize(Array(1,2,3))
    val broadCast = sc.broadcast(1)
    testRDD.map(_+broadCast.value).foreach(println)
    val accumulator = sc.accumulator(4, "july_online")
    testRDD.foreach(accumulator += _)
    println(accumulator.value)
    // wrong method
    // testRDD.foreach( l => println(l+accumulator.value))


    val bigRDD: RDD[(String, Long)] = sc.makeRDD(Array("spark", "hadoop", "hive", "yarn", "hbase", "flink", "flume", "kafka"),
4).zipWithIndex()
    val smallRDD = sc.makeRDD(Array("spark", "hadoop"), 4).zipWithIndex()
    val resultRDD1 = bigRDD.join(smallRDD)
    resultRDD1.foreach(println)
    println()



    val smallValue = smallRDD.collectAsMap()
    val smallBD = sc.broadcast(smallValue)
    val resultRDD2 = bigRDD.mapPartitions { iter =>
      val small = smallBD.value
      val arrayBuffer = ArrayBuffer[(String,Long,Long)]()
      iter.foreach{case(className,number) =>{
        if(small.contains(className)){
          arrayBuffer .+= ((className, small.getOrElse(className,0), number))
        }
        arrayBuffer.iterator
      }
      }
      arrayBuffer.iterator
    }
    resultRDD1.foreach(println)




    sc.stop()
  }
}
```

持久化

```
import org.apache.spark.storage.StorageLevel
import org.apache.spark.{SparkConf, SparkContext}

/**
  * Created by ding on 2017/12/23.
  */
object Test {
  def main(args: Array[String]) {
```

```
    val conf = new SparkConf().setAppName("july_online").setMaster("local[2]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("WARN")
    /**

      */
    val testRdd1 = sc.textFile("C:\\Users\\ding\\Desktop\\Spark1.txt", 4)
    val testRdd2 = sc.textFile("C:\\Users\\ding\\Desktop\\Spark2.txt", 4)
    val wordRdd1 = testRdd1.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_)
    wordRdd1.persist(StorageLevel.MEMORY_AND_DISK)
    val wordRdd2 = testRdd2.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_)
    wordRdd1.join(wordRdd2).foreach(println)
    val sparkCount = wordRdd1.filter(_._1.equals("Spark")).map(_._2).first()
    println(sparkCount)
    wordRdd1.unpersist()
    sc.wait(1000)
  }
}
```

预聚合

```
val rawClassRDD = sc.makeRDD(Array("spark", "hadoop", "hive", "yarn", "hbase", "flink", "flink", "flink", "kafka"), 4).zipWithIndex()
    rawClassRDD.groupByKey().mapValues{case iter: Iterable[Long] =>
      var number:Long = 0
      iter.foreach(l => number += l)
      number
    }.foreach(println)
    println()
    rawClassRDD.reduceByKey(_+_).foreach(println)
```