

1. 自定义分区器:

```
// 自定义分区器
/**
 * 目的: 根据学科的名称 来进行分区  php  bigdata
 */
class SubPartitioner(val totalSub: Array[String]) extends Partitioner {
  // 学科名称和分区编号的映射关系
  val mp = new mutable.HashMap[String, Int]()

  // 分区编号从0 开始
  // 需要对map 赋值
  var index = 0
  // mp.put("php",1)
  for (sub <- totalSub) {
    mp(sub) = index // mp(php) = 0
    index += 1
  }

  // 把学科和学科对应的分区添加进集合中
  // for(i <- 0 until totalSub.length){
  //   mp(totalSub(i))=i
  // }

  // 分区的数量  分区的数量, 学科的名称
  override def numPartitions: Int = totalSub.length

  // 根据key 来获取相应的分区编号  传递一个学科的名称  返回一个分区编号
  override def getPartition(key: Any): Int = {
    // 对 获取到的key 数据进行转换
    val keys = key.asInstanceOf[(String, String)]

    // 学科的名称
    val subName = keys._1
    // 根据学科的名称, 去对应的映射关系中找到该学科对应的分区编号
    mp(subName)
  }
}
```

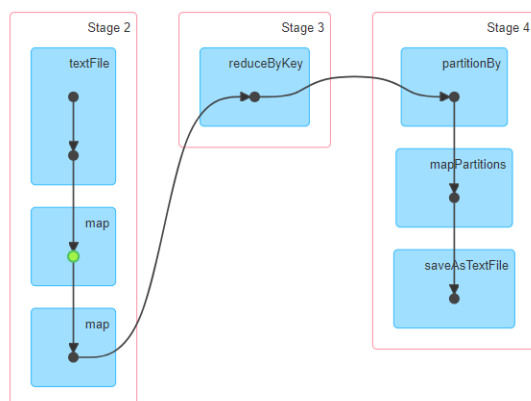
自定义分区器调用:

```
// 按照key 分组聚合
val reduceData: RDD[((String, String), Int)] = sbWithOne.reduceByKey(_ + _)

/**
 * 自定义的分区器
 * hashPartitioner
 * php 0
 * javaee 1
 * bigdata 2
 */

// 调用partitionBy 方法, 进行重分区
val result: RDD[((String, String), Int)] = reduceData.partitionBy(new
SubPartitioner(totalSub))
```

运行 DAG 图:

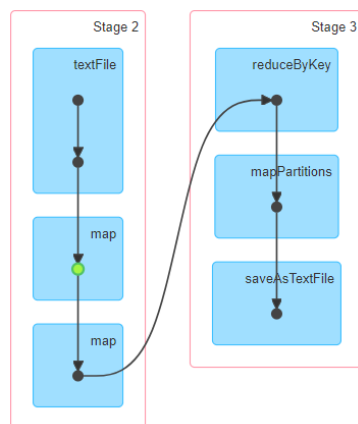


优化的分区器:

直接在分区的时候调用, 省略了一个分区的步骤, stage 的数量从 3 个下降到 2 个

```
// 按照key 分组聚合
val result: RDD[((String, String), Int)] = sbWithOne.reduceByKey(new
SubPartitioner(totalSub),_+_)
```

优化后的运行 DAG 图:



2. 自定义排序:

针对多种排序条件:

1, 把数据封装成类或者 case class, 然后类继承 Ordered[类型] , 然后可以进行排序了。

类中指定了排序的规则。

如果是 class, 需要实现序列化特质, Serializable, 如果是 case class, 可以不实现该序列化特质。

这种处理方式, 返回值类型是类的实例对象。

```
java.io.NotSerializableException: cn.edu360.spark29.day05.Person
Serialization stack:
- object not serializable (class: cn.edu360.spark29.day05.Person, value: cn.edu360.spark29.day05.Person@702992b7)
  at org.apache.spark.serializer.SerializationDebugger$.improveException(SerializationDebugger.scala:40)
  at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala:46)
  at org.apache.spark.serializer.SerializationStream.writeValue(Serializer.scala:134)
```

2, 数据不进行封装, 在排序的时候, 利用 class 或者 case class 指定排序的规则。

类需要继承 Ordered[类型], 实现序列化特质, case 不需实现序列化特质。

返回值的类型是: 还是原来的数据类型。和类无关。

3, 利用隐式转换, 类可以不实现 Ordered 的特质,

隐式转换支持，隐式的 object 和隐式的变量，**优先使用隐式的 object**。

(隐式转换可以写在任意地方，但是导入到当前的对象中)

```
// ordering object 变量 优先使用隐式的 object
implicit object order extends Ordering[Person3] {
  override def compare(x: Person3, y: Person3): Int = {
    // 升序
    x.fv - y.fv
  }
}

// 隐式的变量 隐式值
implicit val ord = new Ordering[Person3] {
  override def compare(x: Person3, y: Person3): Int = {
    -(x.fv - y.fv)
  }
}

// 在排序的时候，指定排序的规则 通过类和 case class 来传递规则
val by: RDD[(String, Int, Int)] = map.sortBy(t => new Person3(t._1, t._2, t._3))
```

4, 最简单的方式，直接使用元组，封装要排序的规则：

```
// 直接在元组中指定排序的规则
val by: RDD[(String, Int, Int)] = map.sortBy(t => (-t._3, t._2))
```

详情可参考：https://blog.csdn.net/qq_21439395/article/details/80200790

3.RDD 的缓存/持久化

Spark 速度非常快的原因之一，就是不同操作中可以在内存中持久化或缓存数据集。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。

默认情况下，每一个转换过的 RDD 都会在它之上执行一个动作时被重新计算。

3.1. 缓存意义

RDD 被缓存后，Spark 将会在集群中，保存相关元数据，下次使用这个 RDD 时，它将能更快速访问，不需要计算。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。

如果 rdd 只被使用一次或者很少次，不需要持久化，如果持久化无谓的 RDD，会浪费内存（或硬盘）空间，反而降低系统整体性能。如果 rdd 被重复使用或者计算其代价很高，才考虑持久化。另外，shuffle 后生成的 rdd 尽量持久化，因为 shuffle 代价太高。

3.2. 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。根据 useDisk、useMemory、off_heap、deserialized、replication 五个参数的组合提供了 12 种存储级别。

分布式内存文件系统: Alluxio (原名 Tachyon)

Alluxio 官网: <http://www.alluxio.org>

五个参数:

```
class StorageLevel private(  
  private var _useDisk: Boolean, // 是否使用磁盘  
  private var _useMemory: Boolean, // 是否使用内存  
  private var _useOffHeap: Boolean, // 使用使用jvm内存  
  private var _deserialized: Boolean, // 是否序列化, 如果是false 使用的是java的序列化 如果是true, 用的是spark.  
  private var _replication: Int = 1) // 副本数量 一个分区的数量存几份 1份
```

```
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)
```

比较常用的:

cache

memory_and_ser

memory_and_disk

cache()

persist(StorageLevel.MEMORY_AND_DISK)

如果不使用 cache (), 那么就必须指定缓存的级别:

rdd.persist(StorageLevel.MEMORY_AND_DISK)

cache 需要注意的事项

1.cache 是一个 lazy 的算子, 必须有 Actions 类型的算子 (比如: count) 触发执行。

2.如果内存足够的大，那么所有的分区都能被缓存，如果内存不足，那看情况，最差的情况就是一个分区的数据都没有被缓存。

3.缓存有可能丢失，或者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。

4, persist(StorageLevel.MEMORY_AND_DISK) 优先使用内存，内存不足，再使用磁盘

IDEA 中写代码使用：

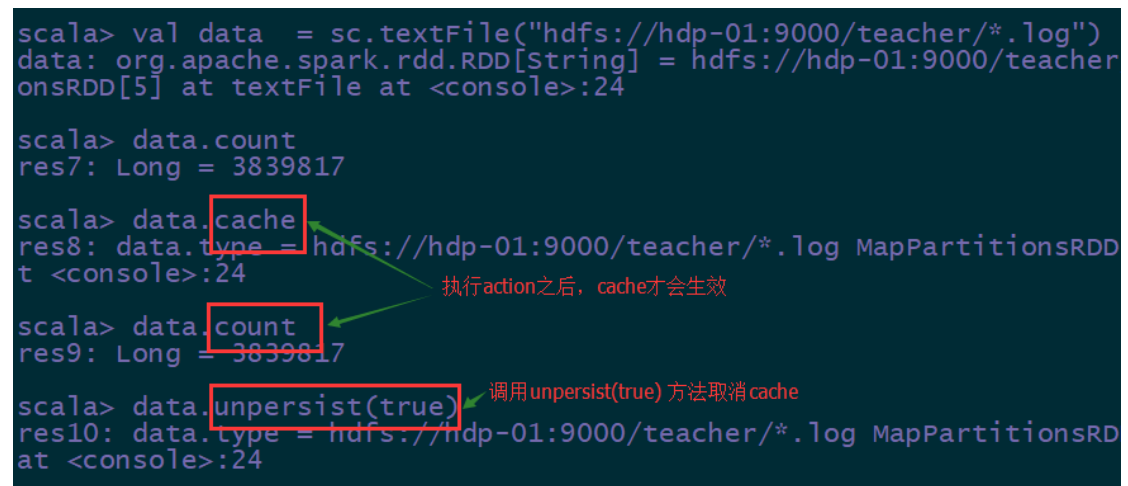
cache ()

persist (StorageLevel.MEMORY_AND_DISK)

如果在 spark-shell 中使用：

rdd2.persist(org.apache.spark.storage.StorageLevel.MEMORY_AND_DISK)

cache 执行方式：



```
scala> val data = sc.textFile("hdfs://hdp-01:9000/teacher/*.log")
data: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/teacher
onsRDD[5] at textFile at <console>:24

scala> data.count
res7: Long = 3839817

scala> data.cache
res8: data.type = hdfs://hdp-01:9000/teacher/*.log MapPartitionsRDD
t <console>:24

scala> data.count
res9: Long = 3839817

scala> data.unpersist(true)
res10: data.type = hdfs://hdp-01:9000/teacher/*.log MapPartitionsRD
at <console>:24
```

```
val rdd = sc.textFile("hdfs://hdp-01:9000/teacher.log")

import org.apache.spark.storage.StorageLevel

val rdd2 = rdd.persist(StorageLevel.MEMORY_AND_DISK)

rdd2.top(10)
```

RDDs					
RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
hdfs://hdp-01:9000/teacher.log	Disk Serialized 1x Replicated	2	100%	0.0 B	132.0 MB

查看任务监控界面：

<div><div>APACHE Spark 2.2.0</div><div>Jobs</div><div>Stages</div><div>Storage</div><div>Environment</div><div>Executors</div><div>SQL</div></div>					
Storage					
RDDs					
RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	
hdfs://hdp-01:9000/teacher/*.log	Memory Deserialized 1x Replicated	3	100%	401.0 MB	

实际中：

如果某一个 RDD 被使用了超过一次，就应该进行 cache 或者 persist。提升效率。

非常复杂的计算结果（shuffle，得到的结果后续被使用，应当进行 cache）

如果 cache 和 persist 还不能满足需求，就使用 checkpoint

4.Checkpoint

4.1. 背景

既然可以缓存，为何需要检查点

如果缓存丢失了，则需要重新计算。

cache 机制保证了需要访问重复数据的应用（如迭代型算法和交互式应用）可以运行的更快。与 Hadoop MapReduce job 不同的是 Spark 的逻辑/物理执行图可能很庞大，task 中 computing chain 可能会很长，计算某些 RDD 也可能很耗时。这时，如果 task 中

途运行出错，那么 task 的整个 computing chain 需要重算，代价太高。因此，有必要将计算代价较大的 RDD checkpoint 一下，这样，当下游 RDD 计算出错时，可以直接从 checkpoint 过的 RDD 那里里读取数据继续算。

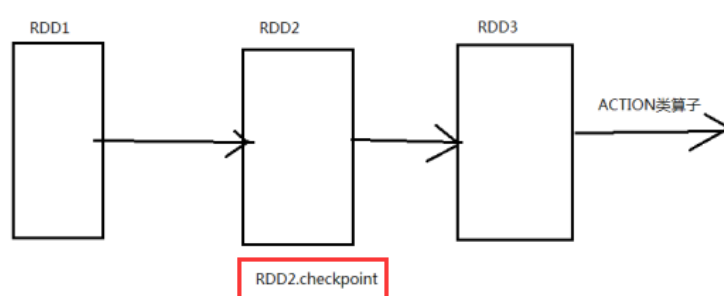
把计算代价特别大的rdd，进行checkpoint

4.2. 原理说明

为当前 RDD 设置检查点。该函数将会创建一个二进制的文件，并存储到 checkpoint 目录中，该目录是用 `SparkContext.setCheckpointDir(path)` 设置的。在 checkpoint 的过程中，该 RDD 的所有依赖于父 RDD 中的信息将全部被移出。对 RDD 进行 checkpoint 操作并不会马上被执行，必须执行 Action 操作才能触发。

Checkpoint操作比较适用于迭代计算非常复杂的情况，也就是恢复计算代价非常高的情况，适当进行 checkpoint 会有很大的好处。

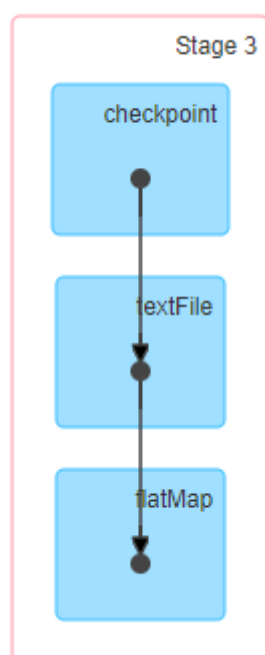
checkpoint 流程：



1.在 RDD 的 job 执行完成之后，会自动的从 finalRDD(RDD3)从后往前进行回溯(为什么能够回溯？因为 RDD 的第三大特性，RDD 之间是有一系列的依赖关系)，遇到哪一个 RDD (这里是 RDD2) 调用了 checkpoint 这个方法，就会对这个 RDD 做一个标记 made for checkpoint

- 2.另外重新启动一个新的 job,重新计算被标记的 RDD, 将 RDD 的结果写入到 HDFS 中
- 3.如何对第二步进行优化: 重新计算被标记的 RDD,这样的话这个 RDD 就被计算了两次, 最好调用 checkpoint 之前进行 cache 一下, 这样的话, 重新启动这个 job 只需要将内存中的数据拷贝到 HDFS 上就可以 (省去了计算的过程)
- 4.checkpoint 的 job 执行完成之后, 会**将这个 RDD 的依赖关系切断** (RDD2 不需要再依赖 RDD1, 因为 RDD2 已经持久化, 以后需要数据的时候直接从持久化的路径取就可以了), 并统一更名为 **CheckpointRDD** (RDD3 的父 RDD 更名为 checkpointRDD)

checkpoint 之后, 操作该 rdd 的数据, 都从 checkpoint 目录来读取:



checkpoint 的目录:

```
drwxr-xr-x - root supergroup 0 2018-01-26 08:00 /ck2018888/0ca4292f-b6d6-48d3-aaeb-e8b52d842ebc/rdd-3
drwxr-xr-x - root supergroup 0 2018-01-26 08:05 /ck2018888/0ca4292f-b6d6-48d3-aaeb-e8b52d842ebc/rdd-6
```

第一级目录, 指定的 `sc.setCheckpointDir(Path)`

第二级目录: 对应的是为我们的 application

4.3. 操作步骤:

```
# 读取数据

val data = sc.textFile("hdfs://hdp-01:9000/teacher/*.log")

# 设置 sparkContext 的 checkpoint 目录

sc.setCheckpointDir("hdfs://hdp-01:9000/ck2017")

# 对数据进行一些复杂的算子操作

val data2 = data.filter(_.contains("bigdata"))

# 对得到的结果数据设置检查点

data2.checkpoint

# 触发执行

data2.count

# 此处会有两次执行，一次把数据写入 checkpoint 目录，一次执行 count 计算结果

还可以对 data2 这个 rdd 执行缓存，然后执行时，会从缓存中写入数据到 hdfs 中的

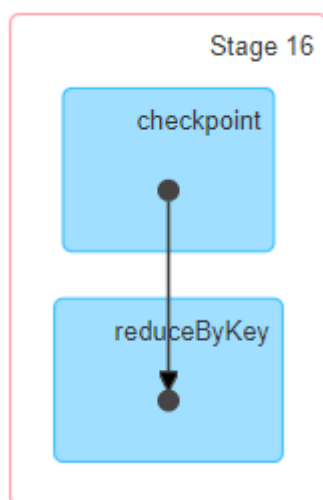
checkpoint 目录。

data2.cache
```

缓存和 checkPoint 的比较:

- 1, 缓存 (持久化) , 是在触发 action 之后, 把数据写入到内存或者磁盘中。
- 2, checkpoint 也是在触发 action 之后, 执行任务。单独再启动一个 job, 负责写入数据到 hdfs 中。(把 rdd 分区中的数据, 以二进制文本的方式写入到 hdfs 中, 有几个分区, 就有几个二进制文件)
- 3, 某一个 RDD 被 checkpoint 之后, 他的父依赖关系, 会被删除, 该 RDD 转换成了

CheckpointRDD, 以后再对该 rdd 的所有操作, 都是从 hdfs 中的 checkpoint 的具体目录来读取数据。缓存之后, rdd 的依赖关系还是存在的。



checkpoint:

业务逻辑特别复杂, 或者经历多次 shuffle, 或者经过多次迭代之后得到的数据,

机器学习中, 迭代次数非常多的时候, 优先使用 checkpoint

机器学习, 迭代信息, 从历史的数据中, 分析出来的一些规律, 然后基于这些规律进行预测。

当使用 checkpoint 时, 优先使用 **cache+checkpoint**

5. 综合案例:

需求: 根据访问日志中的 IP 地址字段, 来计算 IP 归属地 (省份), 并按照省份, 计算出访问次数, 并将计算好的结果写入到 mysql 中

需求分析:

2 个文件,

access.log 日志文件 ip 地址字段

ip.txt

数据特点： 不会频繁改变； 在处理每一条数据的时候，都会被用到。

知识库，应用库 中间库 mysql hdfs 中， hbase

collect

知识库数据 + collect + 广播变量使用

服务器上的日志，写入到 hdfs 中，给大数据部门进行分析使用。

思路：

- 1，把日志数据进行切分，获取 ip 地址字段，把 ip 地址转换成 10 进制数据
- 2，把 ip 规则库数据，切分，获取 10 进制的起始值，终止值，省份

IP 地址转换为 10 进制数据的方法：

```
// 定义一个方法，把 ip 地址转换为 10 进制
def ip2Long(ip: String): Long = {
  val fragments = ip.split("[.]")
  var ipNum = 0L
  for (i <- 0 until fragments.length){
    ipNum = fragments(i).toLong | ipNum << 8L
  }
  ipNum
}
```

mysql 的 maven 依赖：

```
<!--mysql 依赖 pom-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.32</version>
</dependency>
```

如果需要创建表:

```
val pst1 = conn.prepareStatement("create table IF NOT EXISTS access_log3(province  
varchar(20),count int)")  
pst1.execute()
```

数据库连接方式一:

```
// 连接数据库方式  
  
var conn: Connection = null  
var pstmt: PreparedStatement = null  
try {  
    conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/scott",  
    "root", "123")  
    pstmt = conn.prepareStatement("insert into access_log values (?,?)")  
    // 因为拿到的每一个元素是一个迭代器，所以这里还需要循环迭代  
    tp.foreach({  
        t =>  
            pstmt.setString(1, t._1)  
            pstmt.setInt(2, t._2)  
            pstmt.executeUpdate()  
    })  
} catch {  
    case e: Exception => println(e.printStackTrace())  
}  
finally {  
    // 关闭连接  
    if (pstmt != null) pstmt.close()  
    if (conn != null) conn.close()  
}
```