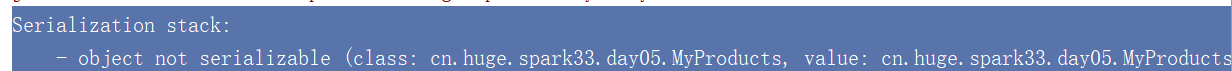
# 自定义排序

## 用类或者样例类来封装数据

在类或者样例类中实现比较器的特质，重写比较的方法。

类必须实现序列化特质。

样例类可以不需要实现序列化特质。



|  |
| --- |
| Serialization stack:  - object not serializable (class: cn.huge.spark33.day05.MyProducts, value: cn.huge.spark33.day05.MyProducts@69dc49b4) |

|  |
| --- |
| **object** SortDemo2 {   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   **val** products: RDD[String] = sc.makeRDD(*List*(**"pipian 99.9 1000"**, **"lazhu 3.5 10000"**, **"shoukao 299.9 10000"**, **"feizao 3.9 1000"**, **"shouji 4999.99 100"**))    *// 按照商品库存的降序   // 数据切分* **val** splitRdd: RDD[MyProducts] = products.map(t => {  **val** split = t.split(**" "**)  **val** pname = split(0)  **val** price = split(1).toDouble  **val** amount = split(2).toInt  **new** MyProducts(pname, price, amount)  })   **val** result: RDD[MyProducts] = splitRdd.sortBy(t => t)  result.foreach(*println*)    sc.stop()  } }  *// 类实现特质* **case class** MyProducts(**val** pname: String, **val** price: Double,**val** amount: Int) **extends** Ordered[MyProducts] */\*with Serializable\*/*{  *// 具有了比较的规则* **override def** compare(that: MyProducts): Int = {   **if** (**this**.price == that.price) {  *// 库存的升序* **this**.amount - that.amount  } **else** {  *// 按照价格的降序* **if** (that.price - **this**.price > 0) 1 **else** -1  }  }   **override def** toString = **s"MyProducts($**pname**, $**price**, $**amount**)"** } |

## 利用类的排序规则实现

数据还是元组，仅仅是利用类的排序规则

如果使用类： 类需要实现序列化特质。实现比较器

使用样例类，只需要实现比较器

利用类的排序规则进行排序之后，数据类型是不变的。之前是元组，现在还是元组。

|  |
| --- |
| **object** SortDemo3 {   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   **val** products: RDD[String] = sc.makeRDD(*List*(**"pipian 99.9 1000"**, **"lazhu 3.5 10000"**, **"shoukao 299.9 10000"**, **"feizao 3.9 1000"**, **"shouji 4999.99 100"**))    *// 按照商品库存的降序   // 数据切分 数据还是元组* **val** splitRdd = products.map(t => {  **val** split = t.split(**" "**)  **val** pname = split(0)  **val** price = split(1).toDouble  **val** amount = split(2).toInt  (pname, price, amount)  })   *// 仅仅是利用类的排序规则* **val** result:RDD[(String,Double,Int)] = splitRdd.sortBy(t => *MyProducts*(t.\_1, t.\_2, t.\_3))  result.foreach(*println*)   sc.stop()  } } |

## 利用隐式转换来实现

类不需要实现比较器，

在上下文环境中，通过隐式转换把比较器的规则导入进行即可。

隐式转换，支持 隐式方法 ，隐式函数，隐式变量，隐式object

|  |
| --- |
| **object** SortDemo4 {   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   **val** products: RDD[String] = sc.makeRDD(*List*(**"pipian 99.9 1000"**, **"lazhu 3.5 10000"**, **"shoukao 299.9 10000"**, **"feizao 3.9 1000"**, **"shouji 4999.99 100"**))    *// 按照商品库存的降序   // 数据切分 数据还是元组* **val** splitRdd = products.map(t => {  **val** split = t.split(**" "**)  **val** pname = split(0)  **val** price = split(1).toDouble  **val** amount = split(2).toInt  (pname, price, amount)  })   *// 利用隐式转换 隐式方法* **implicit def** pro2Ordered(pro: MyProducts2): Ordered[MyProducts2] = {  **new** Ordered[MyProducts2] {  **override def** compare(that: MyProducts2): Int = {  **if** (pro.price == that.price) {  *// 库存的升序* pro.amount - that.amount  } **else** {  *// 按照价格的降序* **if** (that.price - pro.price > 0) 1 **else** -1  }  }  }  }   *// 仅仅是利用类的排序规则* **val** result: RDD[(String, Double, Int)] = splitRdd.sortBy(t => *MyProducts2*(t.\_1, t.\_2, t.\_3))  result.foreach(*println*)   sc.stop()  } }  **case class** MyProducts2(**val** pname: String, **val** price: Double, **val** amount: Int) {   **override def** toString = **s"MyProducts2($**pname**, $**price**, $**amount**)"** } |

更多的隐式相关的代码： <https://blog.csdn.net/qq_21439395/article/details/80200790>

## ordering的on方法

思考题： treeMap

|  |
| --- |
| **object** SortDemo5 {   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   **val** products: RDD[String] = sc.makeRDD(*List*(**"pipian 99.9 1000"**, **"lazhu 3.5 10000"**, **"shoukao 299.9 10000"**, **"feizao 3.9 1000"**, **"shouji 4999.99 100"**))    *// 按照商品库存的降序   // 数据切分 数据还是元组* **val** splitRdd = products.map(t => {  **val** split = t.split(**" "**)  **val** pname = split(0)  **val** price = split(1).toDouble  **val** amount = split(2).toInt  (pname, price, amount)  })    */\* t => (-t.\_2, t.\_3) 排序的条件  (String, Double, Int) 数据的类型  (Double, Int) 排序条件的类型  \*/* **implicit val** ord = *Ordering*[(Double, Int)].on[(String, Double, Int)](t => (-t.\_2, t.\_3))   *// 仅仅是利用类的排序规则* **val** result: RDD[(String, Double, Int)] = splitRdd.sortBy(t => t)  result.foreach(*println*)   sc.stop()  } |

## 直接利用元组封装多条件即可

|  |
| --- |
| **object** SortDemo6 {   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   **val** products: RDD[String] = sc.makeRDD(*List*(**"pipian 99.9 1000"**, **"lazhu 3.5 10000"**, **"shoukao 299.9 10000"**, **"feizao 3.9 1000"**, **"shouji 4999.99 100"**))    *// 数据切分 数据还是元组* **val** splitRdd = products.map(t => {  **val** split = t.split(**" "**)  **val** pname = split(0)  **val** price = split(1).toDouble  **val** amount = split(2).toInt  (pname, price, amount)  })   *// 仅仅是利用类的排序规则* **val** result: RDD[(String, Double, Int)] = splitRdd.sortBy(t => (-t.\_3, t.\_2))   *// 元组 和 样例类有何关系？* result.foreach(*println*)   sc.stop()  } } |

元组的本质，就使用样例类



# spark中的高级的特性-持久化

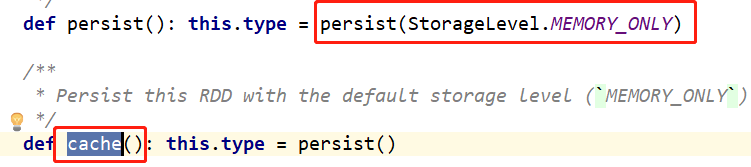
## 简介

**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。**

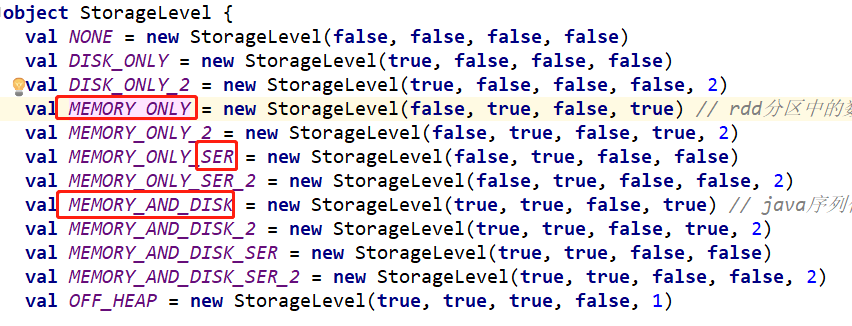
某一个rdd被使用了多次，每一次都被重新计算。

持久化的意义，把某些频繁使用的rdd进行持久化，然后以后基于该rdd的调用，都优先从持久化介质中获取数据。

## 持久化

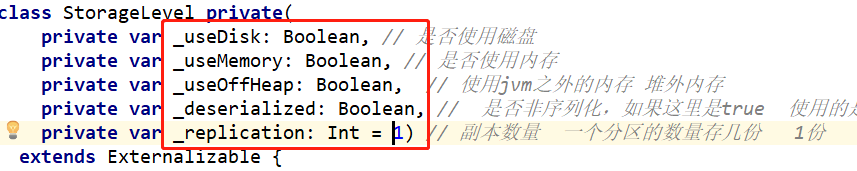


Persist方法中传递的参数，是一个存储的级别： StorageLevel



一共12个存储级别，通过不同的参数来实现的。

是否使用磁盘，是否使用内存，是否使用堆外内存，是否非序列化，副本数量



是否单独使用;是否有几个副本；是否有序列化； 堆外存储

## 如何使用：

常用的存储策略：

cache = persist(StorageLevel.MEMEORY\_ONLY)

MEMORY\_ONLY\_SER

MEMORY\_AND\_DISK : 优先使用内存，如果内存不足，再使用磁盘

使用方式： 直接在rdd后面调用persist或者cache方法即可。

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/storage")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/storage MapPartitionsRDD[1] at textFile at <console>:24  scala> val rdd2 = rdd1.map((\_,1))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[2] at map at <console>:26  scala> rdd2.cache()  res0: rdd2.type = MapPartitionsRDD[2] at map at <console>:26  scala> rdd2.collect |

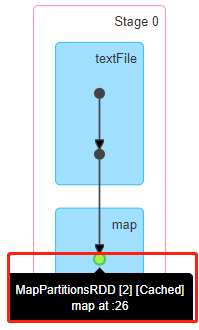


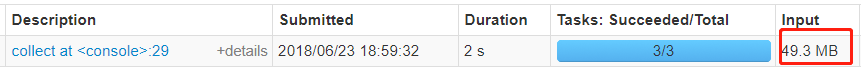
cache使用的是java的序列化机制，然后数据要比原始的数据大好几倍。

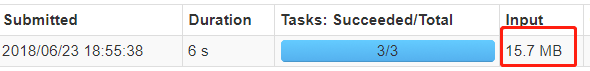
当在某一个rdd上调用cache或者persisit(xxx)之后，没有立即执行。

持久化算子，是lazy执行的，当触发action，才会执行。

被持久化的rdd：







在IDEA中使用：

|  |
| --- |
| *// 直接在rdd后面调用方法，参数传递具体的存储级别。* products.cache() products.persist(StorageLevel.*MEMORY\_ONLY\_SER*) |

在spark-shell中使用：

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/storage")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/storage MapPartitionsRDD[3] at textFile at <console>:24  scala> rdd1.persist(org.apache.spark.storage.StorageLevel.MEMORY\_AND\_DISK) |

缓存如何清除：

rdd1.unpersist()

实际中应该怎么用：

某一个rdd被使用了多次，持久化。

1. 优先使用cache。
2. StorageLevel.MEMORY\_AND\_DISK 或者 StorageLevel.MEMEORY\_ONLY\_SER

总结持久化：

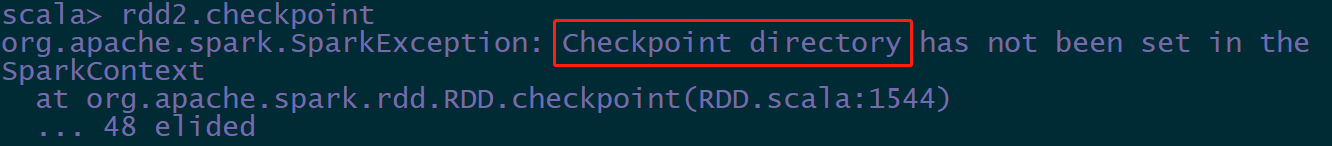
1. 持久化算子，是lazy执行的，只有当触发action算子，就把rdd相应的数据存储到相关的持久化介质中。
2. 持久化之后的算子，rdd的依赖关系是没有变的，以后基于该rdd的所有操作，都是优先从存储介质中获取；如果存储介质中没有数据，根据rdd的依赖关系重新计算。
3. 仅仅使用cache的时候，可能由于内存不足，而导致cache了一部分的分区数据，也有可能没有cache任何的数据。

# checkpoint

把rdd中的数据以文件方式写入到分布式的文件hdfs中。

checkpoint 检查点。

1，想要做checkpoint，必须在SparkContex上，设置checkpoint的目录，而且这个目录必须是分布式的文件系统。



scala> sc.setCheckpointDir("hdfs://hdp-01:9000/ckpoint-2018")

最终的目录结构为：

/ckpoint-2018/19a2aee5-fc27-4d87-9177-a56863e90511/rdd-3/part-00000

目录结构：/设置的checkpointDir/application-id/rdd-id/分区的数据

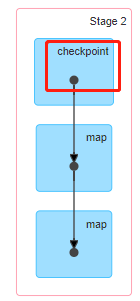
多个application，可以共用同一个checkpointDir。

实际使用：

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/wordcount/input")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/wordcount/input MapPartitionsRDD[1] at textFile at <console>:24  scala> val rdd2 = rdd1.flatMap(\_.split(" ")).map((\_,1))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:26  scala> sc.setCheckpointDir("hdfs://hdp-01:9000/ckpoint-2018")  scala> rdd2.checkpoint  scala> rdd2.top(10) |

## checkpoint总结：

1. 要想对rdd做checkpoint，必须先对SparkContext设置checkpointDir
2. 是lazy执行的，当触发action才会进行checkpoint。
3. checkpoint会产生两个job。第一执行业务逻辑。第二个job把rdd中的数据写入到hdfs中。
4. 当对某一个rdd执行checkpoint之后，这个rdd的父依赖关系不存在了，取而代之是CheckpointRDD。对该rdd的所有的操作，都从hdfs的目录下读取数据。



怎么用？

业务逻辑特别复杂，机器学习中的迭代的数据，数据经过非常复杂的处理之后得到的结果数据。

## cache和checkpoint的比较：

都是lazy执行的。

cache，存储在内存中，checkpoint，分布式的文件系统中。

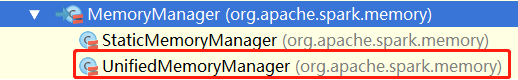
cache 产生一个job，checkpoint，会产生2个job。

cache不会改变rdd的依赖关系，checkpoint会删除之前的依赖关系，生成新的依赖（CheckpointRDD）

# Spark的内存管理机制

spark1.6之前 静态管理机制

spark1.6开始，统一内存管理机制。



## 内存分为3部分：

storage: 缓存 60% \* 50%

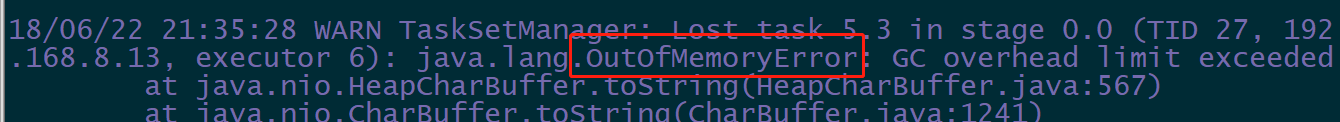
execution： shuffle ,join等运行 60% \* 50%

other: spark内部的数据运行； 保护oom 40%

## 动态占用机制

1. 如果双方的内存都是要完了，直接溢出磁盘。
2. Storage占用的execution的内存，可以被Execution剔除。
3. execution占用了Storage的内存,不能被剔除，直到exection占用的内存释放掉。

collect方法，如果数据量太大，直接报错OOM。



spark2.2.0中关于内存分配的参数：

<http://spark.apachecn.org/docs/cn/2.2.0/configuration.html#memory-management-内存管理>

spark1.6 0.75 spark2.x 0.6



假定executor： 1024mb的内存

系统预留内存： 300Mb

留给storage+ execution： （1024-300）\*0.6 = 434.4M

单独给到Storage： 434.4M \* 50% =

单独给到Exection 434.4M \* 50% =

分配给exector的最低的内存要求是： 300 \* 1.5 = 450M

# 把数据结果写入到mysql中

根据IP地址求归属地，然后按次数统计 ，把结果数据写入到mysql中

访问日志数据 🡪 ip地址

规则库中比较 🡪 归属地 --》 wordcount ---》 结果数据

## 需求分析：

ipaccess.log 🡪 ip地址

ip.txt 🡪 规则数据，中间库，知识库数据 稳定，长期的维护；使用频繁。

ipaccess.log ip 🡪 longIp Array[(start,end,province)] 🡪 wordcount

* 写入到mysql中

RDD不支持嵌套：

|  |
| --- |
| 18/06/23 16:05:45 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)  org.apache.spark.SparkException: This RDD lacks a SparkContext. It could happen in the following cases:  (1) RDD transformations and actions are NOT invoked by the driver, but inside of other transformations; for example, rdd1.map(x => rdd2.values.count() \* x) is invalid because the values transformation and count action cannot be performed inside of the rdd1.map transformation. For more information, see SPARK-5063.  (2) When a Spark Streaming job recovers from checkpoint, this exception will be hit if a reference to an RDD not defined by the streaming job is used in DStream operations. For more information, See SPARK-13758. |

## 代码实现：

|  |
| --- |
| *// 根据ip地址获取longIp* **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 }  *// 定义一个二分搜索的方法* **def** binarySearch(ip: Long, ipRules: Array[(Long, Long, String)]): String = {  *// 两个索引* **var** low = 0  **var** high = ipRules.length - 1  **while** (low <= high) {  *// 取中间索引* **val** middle = (low + high) / 2  *// 获取中间索引位置的值* **val** (start, end, province) = ipRules(middle)  *// 正好找到位置* **if** (ip >= start && ip <= end) {  **return** province  } **else if** (ip < start) { *// 在左区间* high = middle - 1  } **else** {  low = middle + 1  }  }  *// 程序走到这里，没有找到对应的province* **"unknown"** }  **def** main(args: Array[String]): Unit = {   **val** sc = *MySpark*(**this**.getClass.getSimpleName)   *// 读取数据* **val** logs: RDD[String] = sc.textFile(**"f:/mrdata/ipdata/ipaccess.log"**)  **val** ipData: RDD[String] = sc.textFile(**"f:/mrdata/ipdata/ip.txt"**)   **val** ipRuleRDD: RDD[(Long, Long, String)] = ipData.map(t => {  **val** split = t.split(**"\\|"**)  **val** start = split(2).toLong  **val** end = split(3).toLong  **val** province = split(6)  (start, end, province)  })   *// RDD不能嵌套操作* **val** ipRules: Array[(Long, Long, String)] = ipRuleRDD.collect()   *// 数据切分* **val** longIp: RDD[Long] = logs.map(t => {  **val** strIp = t.split(**"\\|"**)(1)  *// 把ip地址转换成10进制  ip2Long*(strIp)  })    *// 调用二分搜索来查询省份* **val** result:RDD[String] = longIp.map(ip => {  *binarySearch*(ip, ipRules)  })   *// 不再过滤非法值* **val** finalRes: RDD[(String, Int)] = result.map((\_,1)).reduceByKey(\_+\_)    *// 对结果数据写入到mysql中* |

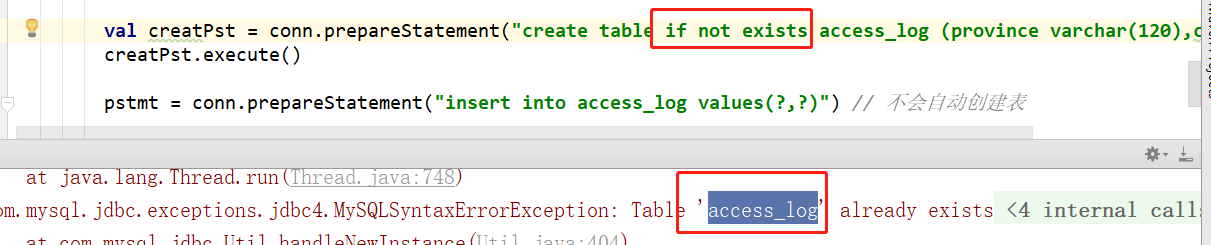
## 数据入库



原因： 缺少mysql的驱动jar包。

|  |
| --- |
| *<!--导入mysql的驱动jar包-->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.38</**version**> </**dependency**> |

搜索jar包的pom配置<http://search.maven.org/>



|  |
| --- |
| finalRes.foreach(tp => {   **var** conn: Connection = **null  var** pstmt: PreparedStatement = **null  try** {  *// URL* **val** url = **"jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8"  val** user = **"root"  val** passwd = **"123"** conn = DriverManager.*getConnection*(url, user, passwd)   **val** creatPst = conn.prepareStatement(**"create table if not exists access\_log (province varchar(120),cnts int)"**)  creatPst.execute()   pstmt = conn.prepareStatement(**"insert into access\_log values(?,?)"**) *// 不会自动创建表   // 赋值* pstmt.setString(1, tp.\_1)  pstmt.setInt(2, tp.\_2)   pstmt.execute()   } **catch** {  **case** e: Exception => e.printStackTrace()  } **finally** {  **if** (pstmt != **null**) pstmt.close()  **if** (conn != **null**) conn.close()  } }) |

## try catch需要的注意事项：

不能再driver端catch executor端的错误，属于不同的机器。

错误的代码， 2/0 这个错误不能被捕获。

|  |
| --- |
| **var** conn: Connection = **null  var** pstmt: PreparedStatement = **null  try** {  *// URL* **val** url = **"jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8"  val** user = **"root"  val** passwd = **"123"** *// 3 / 0 // driver端 的错误可以被捕获  // 对结果数据写入到mysql中* finalRes.foreach(tp => {  2 / 0 *// executor中* conn = DriverManager.*getConnection*(url, user, passwd)  pstmt = conn.prepareStatement(**"insert into access\_log values(?,?)"**) *// 不会自动创建表  // 赋值* pstmt.setString(1, tp.\_1)  pstmt.setInt(2, tp.\_2)  pstmt.execute()   })  } **catch** {  **case** e: Exception => *// e.printStackTrace()* } **finally** {  **if** (pstmt != **null**) pstmt.close()  **if** (conn != **null**) conn.close()  } |

## 闭包

|  |
| --- |
| 在函数内部引用了一个外部的变量：  闭包：  conn = DriverManager.*getConnection*(url, user, passwd)finalRes.foreach(tp => {pstmt = conn.prepareStatement(**"insert into access\_log values(?,?)"**) *// 不会自* }) |

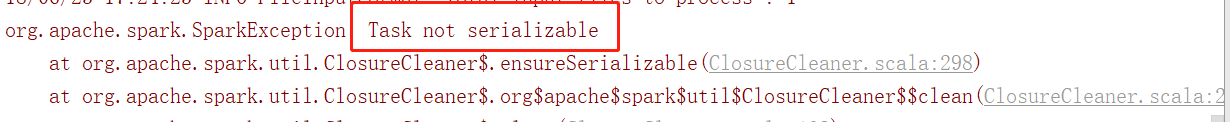
闭包引用：

在函数内部引用了一个外部的变量。

代码块 + 上下文

task在序列化的时候，发现引用了一个没有被序列化的类，所以就会报错。

DriverManager没有实现序列化特质。



## 利用foreachPartition来实现数据入库

|  |
| --- |
| finalRes.foreachPartition(it => {  **var** conn: Connection = **null  var** pstmt: PreparedStatement = **null  try** {  *// URL* **val** url = **"jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8"  val** user = **"root"  val** passwd = **"123"** *// 在生成task的时候，被引用的对象，必须也被序列化发送到executor端。* conn = DriverManager.*getConnection*(url, user, passwd)  *// 闭包引用* pstmt = conn.prepareStatement(**"insert into access\_log values(?,?)"**) *// 不会自动创建表  // 赋值* it.foreach(tp => {  pstmt.setString(1, tp.\_1)  pstmt.setInt(2, tp.\_2)  pstmt.execute()  })  } **catch** {  **case** e: Exception => e.printStackTrace()  } **finally** {  **if** (pstmt != **null**) pstmt.close()  **if** (conn != **null**) conn.close()  } }) |

# 利用广播变量

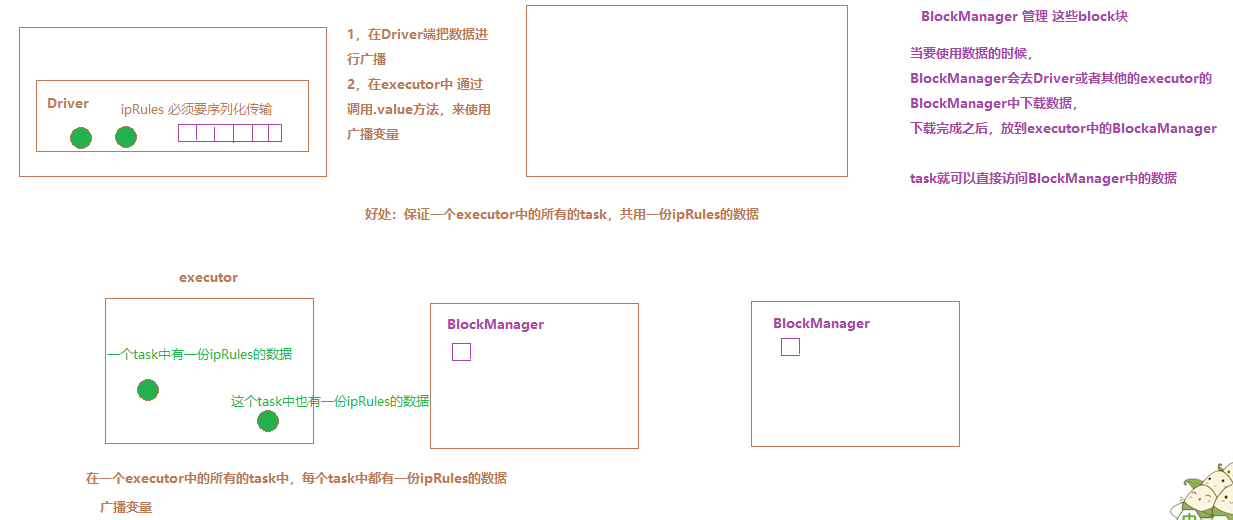
## 理论

广播变量是使用TorrentBroadCast实现的：



比特洪流技术

快播： 快



## 广播变量的使用：

在Driver端把数据进行广播：

不能广播rdd

|  |
| --- |
| *// 把规则库的数据进行广播* **val** broadcast: Broadcast[Array[(Long, Long, String)]] = sc.broadcast(ipRules) |

在executor中使用：

|  |
| --- |
| **val** result: RDD[String] = longIp.map(ip => {  *// 只能保证一个task中共用一份反序列化的数据* **val** iPRulesNews:Array[(Long,Long,String)] = broadcast.value  *binarySearch*(ip, iPRulesNews) }) |

对于有知识库的规则数据，优先使用广播变量实现。

# 今日重点：

1，自定义排序 元组来封装规则数据

1. 持久化和checkpoint cache的使用

持久化和checkpoint的区别和联系

1. spark的内存管理 --- 0.6% 0.5% executor 1g
2. ip求归属地，把结果数据写入到mysql中

二分查找

foreachPartition

广播变量

## 作业题： url的匹配 用广播变量

PVUV的统计