# Off\_heap

分布式内存文件系统：Alluxio (原名Tachyon)

Alluxio 官网：<http://www.alluxio.org>

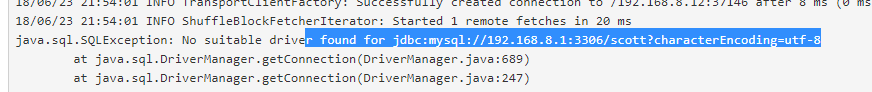
java unsafe API 直接操作操作系统的内存。

# 任务提交到集群运行

修改配置注意事项：

1. 本地模式的配置，必须要删除。
2. 把数据写入到windows上的mysql，必须把mysql的localhost换成具体的ip地址。

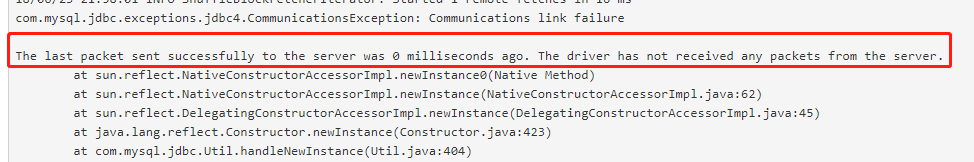
提交任务到集群运行时报错：



解决方案：

1. 直接把mysql的驱动jar包，放到spark\_home/jars目录下即可。
2. 直接通过配--jars mysql…jar 把驱动jar包导入到任务中

|  |
| --- |
| spark-submit --master spark://hdp-01:7077 **--jars** /root/mysql-connector-java-5.1.38.jar --class cn.huge.spark33.day06.IPLocalForStandalone /root/spark33-1.0-SNAPSHOT.jar hdfs://hdp-01:9000/ipaccess.log hdfs://hdp-01:9000/ip.txt |





如何定位端口是否是通的？

|  |
| --- |
| 方案1：  telnet 192.168.8.1 3306  yum -y install telnet  方案2：  mysql -h 192.168.8.1 -uroot -p123 |

解决方案： 把windows的防火墙关闭了



## 如何遇到问题，如何定位问题，并解决问题？？

1. 数据没有写入到msyql中
2. 去看任务运行的Driver端的日志，如果没有任何错误
3. 去任务运行的exector中查看错误日志。
4. 根据日志中的错误内容，来分析定位问题。

windows上的mysql禁止 root用户远程连接。

代码抽取：

把方法和写入mysql的函数，提取到一个IpUtils 对象中。

二分搜索用函数实现的注意事项：

|  |
| --- |
| **val** *binarySearch2* = (ip: Long, ipRules: Array[(Long, Long, String)]) => {  *// 两个索引* **var** low = 0  **var** high = ipRules.length - 1   *// 定义两个变量 进行判断* **var** result = **"unknown"  var** flag = **true  while** (low <= high && flag) {  *// 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) {  result = province  flag = **false** } **else if** (ip < start) { *// 在左区间* high = middle - 1  } **else** {  low = middle + 1  }  }  result } |

# spark中的共享变量：

spark中提供了2种共享变量

## 广播变量

### 总结：

1. val bc = sc.broadcast(广播的内容) 在driver端 ，把数据进行广播
2. 在函数中，使用bc.value获取到广播变量的值。 可以在任意的函数中使用，
3. 广播之后的数据，是只读的，不能修改。
4. RDD不能被广播。rdd不支持嵌套操作。
5. 广播之后的数据，保证了，每一个executor中的所有的task共用一份数据。

规则库，中间库，知识库数据：

稳定，长久的维护，数据分析被频繁的使用到。数据量不会太大。

这些数据，就优先考虑把这些数据进行广播。

## 累加器

全局的计数器。

### 需求分析：

|  |
| --- |
| *// 计数器* **var** counts = 0 *// Driver端定义的* **val** rdd1 = sc.makeRDD(*List*(1, 3, 4, 5, 6, 7, 9), 2)  *// 统计 数据的条数 executor端执行的* rdd1.foreach(t => {  counts += 1  *println*(**s"---+$**{counts}**"**) })  *// 期望：7 实际：0 // Driver端的数据没有变 println*(counts) |
| 这里不能再Driver端统计executor中的数据条件。  但是spark提供了累加器。 |

### 使用：

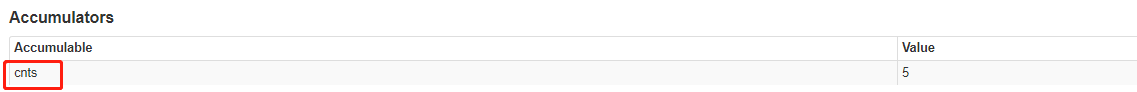
全局的计数器，只支持累加。

|  |
| --- |
| *// 定义一个累加器 用于统计数据的条数* **val** acc1: Accumulator[Int] = sc.accumulator(0, **"cnts"**) **val** acc2: LongAccumulator = sc.longAccumulator(**"cnts2"**) **val** rdd1 = sc.makeRDD(*List*(1, 3, 4, 5, 6, 7, 9), 2) *// 统计 数据的条数 executor端执行的* rdd1.foreach(t => {acc1.add(1)  acc2.add(1) }) *// 在driver端直接通过acc1.value来获取值 println*(acc1.value) *println*(acc2.value) |

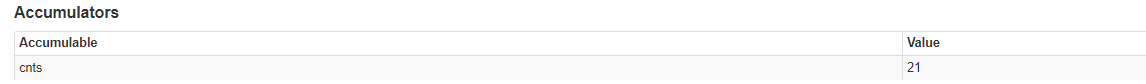
名称如何使用？

是我们在任务运行的监控界面查看的。

|  |
| --- |
| scala> val acc = sc.accumulator(0,"cnts")  warning: there were two deprecation warnings; re-run with -deprecation for details  acc: org.apache.spark.Accumulator[Int] = 0  scala> val rdd = sc.makeRDD(List(1,3,4,2,6),2)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> rdd.foreach(t=> acc.add(1))  [Stage 0:> (0 + 0) / [Stage 0:> (0 + 2) /  scala> acc.value |



|  |
| --- |
| scala> rdd.foreach(t => acc.add(t)) |



在一个application中，累加器的值，是累加的，是共用的。

### 总结：

1. 在Driver端定义累计器。可以为累加器取一个名称。
2. 在任何的函数中，对累加器进行赋值，只支持累加操作
3. 在**driver**端通过**累加器.value**获取到累加器的值。
4. 在一个application中，累加器是共用的，值是累加的。

统计任务运行的时长：

|  |
| --- |
| *// 定义两个累加器，用于通过时长* **val** ip2LongTime = sc.longAccumulator(**"ip2LongTime"**) **val** ipSearch = sc.longAccumulator(**"ipSearch"**) *// 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()  *// 把规则库的数据进行广播* **val** broadcast: Broadcast[Array[(Long, Long, String)]] = sc.broadcast(ipRules)  *// 数据切分* **val** longIp: RDD[Long] = logs.map(t => {  **val** t1 = System.*currentTimeMillis*()  **val** strIp = t.split(**"\\|"**)(1)  *// 把ip地址转换成10进制* **val** result = IpUtils.*ip2Long*(strIp)  **val** t2 = System.*currentTimeMillis*()  ip2LongTime.add(t2 - t1)  result })  *// 调用二分搜索来查询省份* **val** result: RDD[String] = longIp.map(ip => {  **val** t1 = System.*currentTimeMillis*()  *// 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  **val** reuslt = IpUtils.*binarySearch*(ip, iPRulesNews)  **val** t2 = System.*currentTimeMillis*()  ipSearch.add(t2 - t1)  reuslt })  *// 不再过滤非法值* **val** finalRes: RDD[(String, Int)] = result.map((\_, 1)).reduceByKey(\_ + \_)  finalRes.foreach(*println*)  *println*(**"ip2LongTime="**+ip2LongTime.value) *println*(**"ipSearch="**+ipSearch.value) |

# spark中的序列化-序列化的位置

## 回顾

序列化：内存中的对象，写到磁盘中。

反序列化： 把磁盘中的文件，反序列化到内存中。

|  |
| --- |
| **val** p1 = **new** Person(**"shuaishuai"**, 28)  **val** oos = **new** ObjectOutputStream(**new** FileOutputStream(**"f:/mrdata/person3.txt"**))   oos.writeInt(10) *// 不会触发数据刷新* oos.writeObject(p1)  *// oos.flush()* **val** s1 = **new** ObjectInputStream(**new** FileInputStream(**"f:/mrdata/person3.txt"**))   *println*(s1.readInt())  **val** p2o: Object = s1.readObject()  **val** p2 = p2o.asInstanceOf[Person]   *println*(p1 == p2)  *println*(p2.name) |

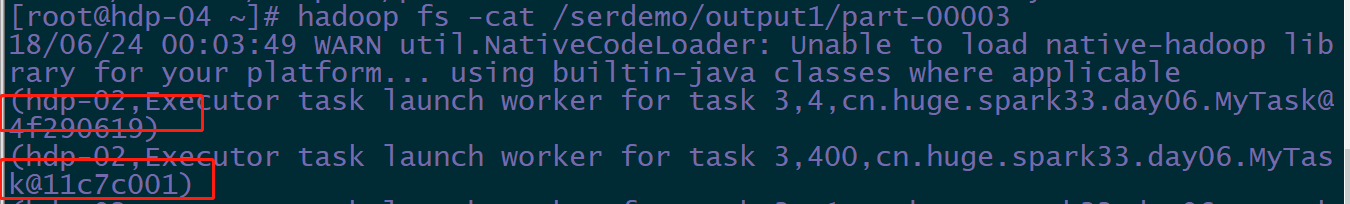
## 在函数中实例化类：

类不需要进行序列化

类会在处理的每一条数据的时候，都进行实例化。

|  |
| --- |
| **class** MyTask {  **val** *mp* = mutable.HashMap[String, Int](**"hadoop"** -> 4, **"spark"** -> 400) } |

|  |
| --- |
| **object** SerDemo1 {   **def** main(args: Array[String]): Unit = {  **val** sc = **new** SparkContext(**new** SparkConf())   *// hadoop spark tom cat* **val** file = sc.textFile(args(0))   **val** finalRes = file.map(t => {  *// 问题？ myTask 是否需要需要序列化 ？？ 不需要序列化  // MyTask会被创建几次？ 每次处理一条数据，就会创建一个实例* **val** mt = **new** MyTask()  *// host threadName* **val** host = InetAddress.*getLocalHost*.getHostName  **val** threadName = Thread.*currentThread*().getName   **val** result = mt.*mp*.getOrElse(t, -1)  *// 返回值* (host, threadName, result, mt.toString)  })   finalRes.saveAsTextFile(args(1))  sc.stop()  } } |

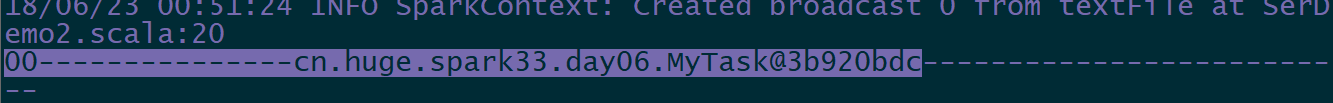


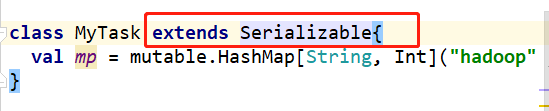
## 闭包引用：

在driver端实例化类，在executor端引用。

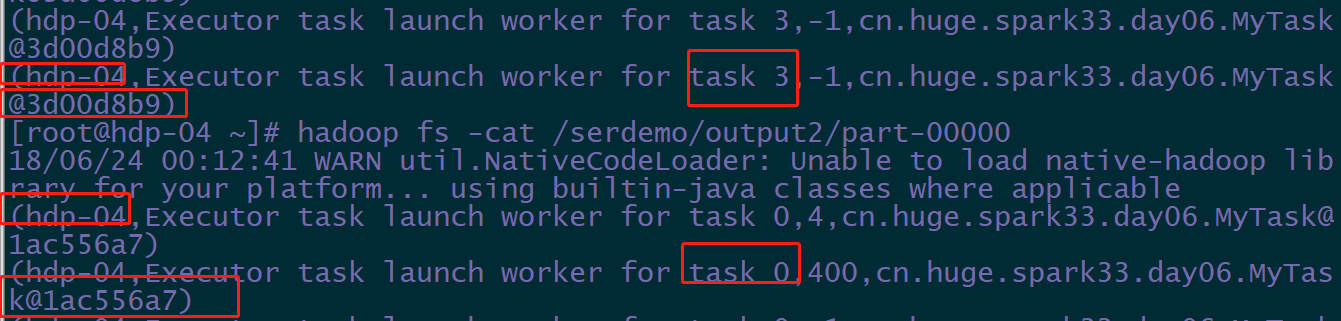
结论：

1，Driver端的类，确实进行了实例化，

2，闭包引用的类，必须进行序列化，然后发送给executor端去执行。



3，每一个task中的所有的数据，共用一个实例，不同的task之间，使用的是不同的实例。

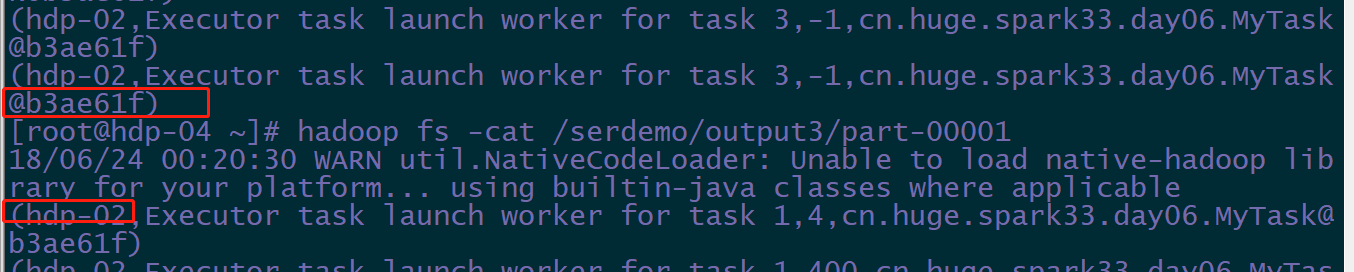


|  |
| --- |
| **val** sc = **new** SparkContext(**new** SparkConf())  *// hadoop spark tom cat* **val** file = sc.textFile(args(0))  *// 这个类就是在Driver端实例化的 类是否需要实现序列化特质？？ 必须要 // 一条数据一个实例？？？ 一个task中的所有的数据，共用一个实例呢？？* **val** mt = **new** MyTask() *println*(**s"00---------------$**{mt.toString}**--------------------------"**)  **val** finalRes = file.map(t => {  *// host threadName* **val** host = InetAddress.*getLocalHost*.getHostName  **val** threadName = Thread.*currentThread*().getName  **val** result = mt.*mp*.getOrElse(t, -1)  *// 返回值* (host, threadName, result, mt.toString) })  finalRes.saveAsTextFile(args(1)) sc.stop() |

## 广播变量

结论： 在同一个executor中的所有的task，共用一份数据。

|  |
| --- |
| spark-submit --master spark://hdp-01:7077 --class cn.huge.spark33.day06.SerDemo3 /root/spark33-1.0-SNAPSHOT.jar hdfs://hdp-01:9000/serdemo/input hdfs://hdp-01:9000/serdemo/output3 |



|  |
| --- |
| **def** main(args: Array[String]): Unit = {  **val** sc = **new** SparkContext(**new** SparkConf())   *// hadoop spark tom cat* **val** file = sc.textFile(args(0))   *// 这个类就是在Driver端实例化的 类是否需要实现序列化特质？？ 必须要  // 一个executor中的所有task中的所有的数据，共用一个实例呢？？* **val** mt = **new** MyTask()   **val** bc = sc.broadcast(mt)  *println*(**s"00---------------$**{mt.toString}**--------------------------"**)   **val** finalRes = file.map(t => {  *// host threadName* **val** host = InetAddress.*getLocalHost*.getHostName  **val** threadName = Thread.*currentThread*().getName  *// 获取广播变量的值* **val** newMT = bc.value  **val** result = newMT.*mp*.getOrElse(t, -1)  *// 返回值* (host, threadName, result, newMT.toString)  })   finalRes.saveAsTextFile(args(1))  sc.stop() } |

# JdbcRDD

|  |
| --- |
| **val** sc = *MySpark*(**this**.getClass.getSimpleName) */\*\*  \* sc: SparkContext,  \* getConnection: () => Connection,  \* sql: String,  \* lowerBound: Long,  \* upperBound: Long,  \* numPartitions: Int,  \* mapRow: (ResultSet)  \*/* **val** getConnection = () => {  **val** url = **"jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8"** DriverManager.*getConnection*(url, **"root"**, **"123"**) }  *// select \* from salgrade where grade > 1 and grade < 2 // select \* from salgrade where grade > 3 and grade < 5* **val** result: JdbcRDD[(Int, Double)] = **new** JdbcRDD(sc,  getConnection,  **"select \* from salgrade where grade >= ? and grade <= ?"**,  1,  5,  2, *// 分区数量* m => {  **val** grade = m.getInt(1)  **val** low = m.getDouble(2)  (grade, low)  } ) result.foreach(*println*) |

# URL的匹配

|  |
| --- |
| **object** UrlMatchDemo {   **def** main(args: Array[String]): Unit = {   **val** sc = *MySpark*(**this**.getClass.getSimpleName)  *// 读取数据* **val** file = sc.textFile(**"f:/mrdata/sparkurldata/40690.txt"**)   **val** url: RDD[String] = sc.textFile(**"f:/mrdata/sparkurldata/url.db"**)   *// url.db 切分 url type* **val** urlSplitRdd: RDD[(String, String)] = url.map(t => {  **val** url = t.substring(0, 14)  **val** types = t.substring(14, 16)  (url, types)  })   *// 明文数据，切分 加密 MD5Url type* **val** md5Url: RDD[(String, String)] = file.map(str => {  **val** split = str.split(**"\t#\t"**)   *// 切分获取去掉http:// 的url* **val** url = split(0).replace(**"http://"**, **""**)  *// md5加密* **val** md5Url = UrlUtils.*md5Encoding*(url).substring(0, 14)  (md5Url, split(1))  })  */\*  val leftJoinRdd: RDD[(String, (String, Option[String]))] = urlSplitRdd.leftOuterJoin(md5Url)   val finalRes = leftJoinRdd.map {  case (key, (urlType, md5Type)) => {   // 则将url.db中的type更新为40690中的type, 不同的话不做任何修改；  // md5Type.getOrElse(urlType)  val finalType = md5Type match {  case Some(v) => if (v.size == 1) "0".concat(v) else v  case None => urlType  }  key.concat(finalType)  }  }\*/   // --------------- 广播变量 -----------  // 聚合匹配* **val** urlMd5Map: collection.Map[String, String] = md5Url.collectAsMap()  **val** bc = sc.broadcast(urlMd5Map)   **val** result: RDD[String] = urlSplitRdd.map {  **case** (key, types) => {  **val** newUrlMd5Map: collection.Map[String, String] = bc.value  key.concat(newUrlMd5Map.getOrElse(key, types))  }  }   result.foreach(*println*)    sc.stop()  } } |

# Spark on yarn

## 简介

yarn: 统一的资源调度平台

spark: 大数据计算引擎

spark任务 提交到yarn 集群上去运行。

和standalone集群是否有关系？

master ,worker是否配置，是否启动，没有任何的关系。

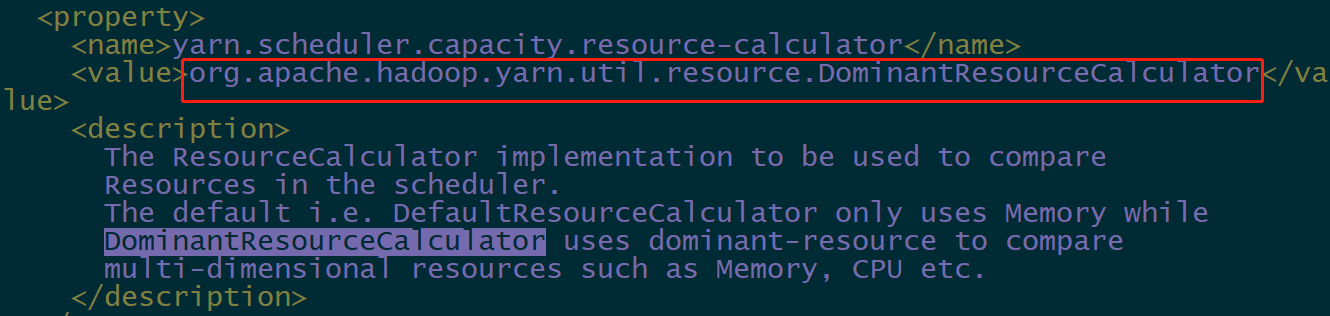
spark on yarn 中的spark是否要配置？ yarn的目录结构。

spark的需要配置多少台呢？

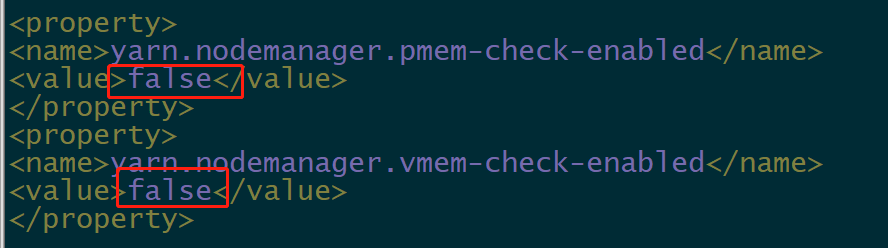
提交任务的地方，客户端。客户端只需要一台即可。

## yarn集群的补充配置

1. hadoop的配置文件目录下，capacity-scheduler.xml



2，设置内存检测为false yarn-site.xml



yarn.nodemanager.pmem-check-enabled

是否启动一个线程检查每个任务正使用的物理内存量，如果任务超出分配值，则直接将其杀掉，默认是true。

yarn.nodemanager.vmem-check-enabled

是否启动一个线程检查每个任务正使用的虚拟内存量，如果任务超出分配值，则直接将其杀掉，默认是true。

在 yarn client模式下，报内存溢出的错误。

资源的分发：

# cd /root/apps/hadoop/etc/hadoop

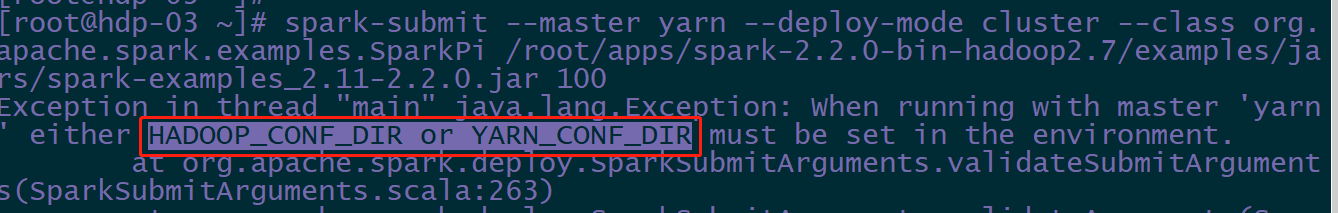
# for i in 2 3 ;do scp capacity-scheduler.xml yarn-site.xml hdp-0$i:$PWD ; done

启动hdfs 和yarn集群：

start-all-hdp.sh

## 配置spark：

必须设置 HADOOP\_CONF\_DIR or YARN\_CONF\_DIR ，指定yarn在哪里。

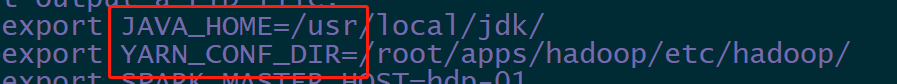


下载spark的安装包，上传，解压，修改配置文件。一台机器即可。就是客户端

spark-env.sh:

1,jdk

2, YARN\_CONF\_DIR



|  |
| --- |
| spark-submit --master yarn --deploy-mode cluster --class org.apache.spark.examples.SparkPi /root/apps/spark-2.2.0-bin-hadoop2.7/examples/jars/spark-examples\_2.11-2.2.0.jar 100 |

## 提交spark on yarn的方式：

spark 2.x 版本的任务提交 spark1.6

集群模式： --master **yarn** --deploy-mode cluster --master yarn-cluster

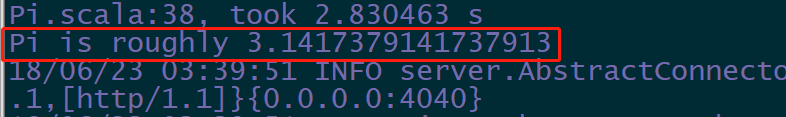
客户端模式：--master yarn --deploy-mode **client** --master yarn-client

最主要的区别： Driver程序 运行在哪里。

集群模式下，Driver运行在yarn的集群中；客户端提交任务成功之后，客户端就可以退出了。



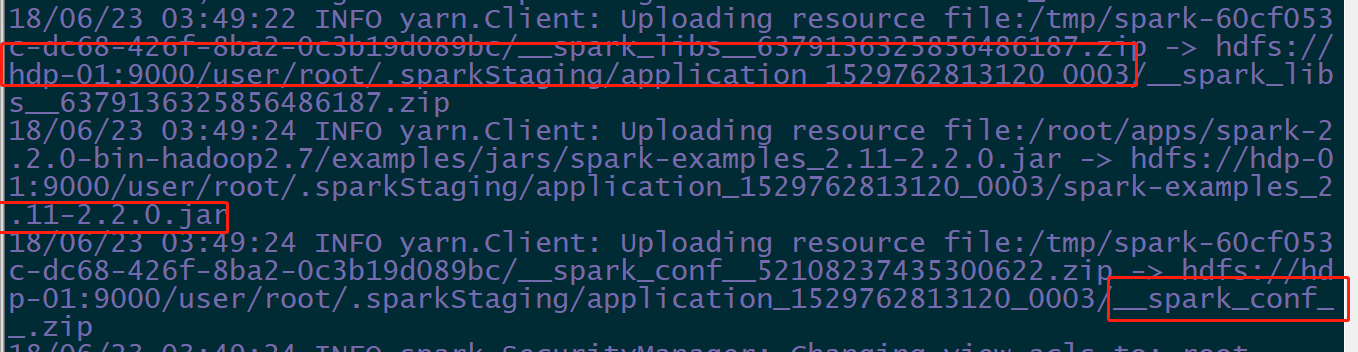
客户端模式，Driver就运行在客户端。 客户端不能退出。



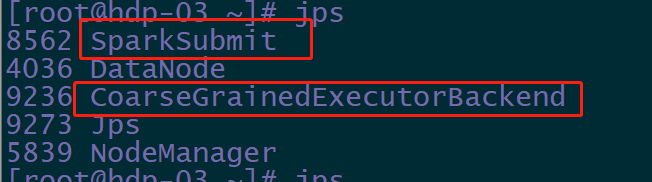
## cluster集群模式

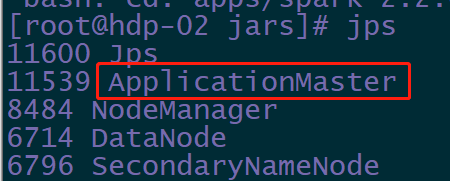
Driver运行在集群中，和ApplicationMaster运行在一起。Driver挂掉可以重新启动。

MrAppMaster



产生哪些进程？





ApplicationMaster ： 负责申请容器启动executor

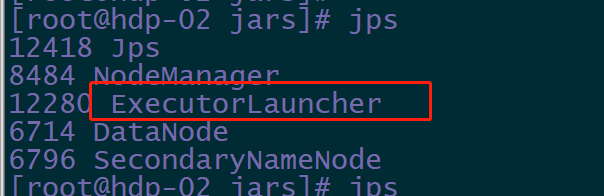
Driver运行在这里，

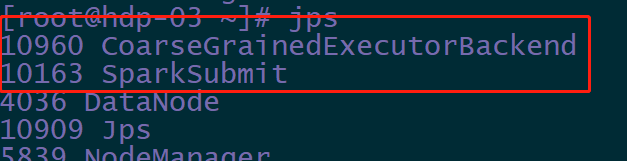
Executor: 执行task的地方

SparkSubmit: 哪里执行spark-submit命令，就在哪里产生该进程。任务提交完成，就可以执行退出。然后程序运行在集群中。

客户端是看不到程序运行的结果。

## Client模式



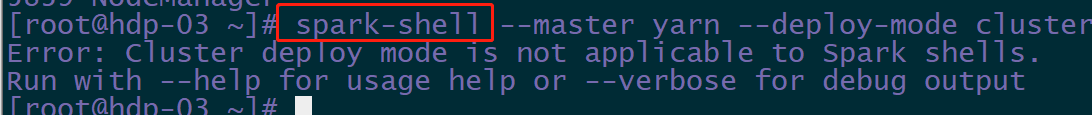


ExecutorLauncher:

就是阉割版的ApplicationMaster。

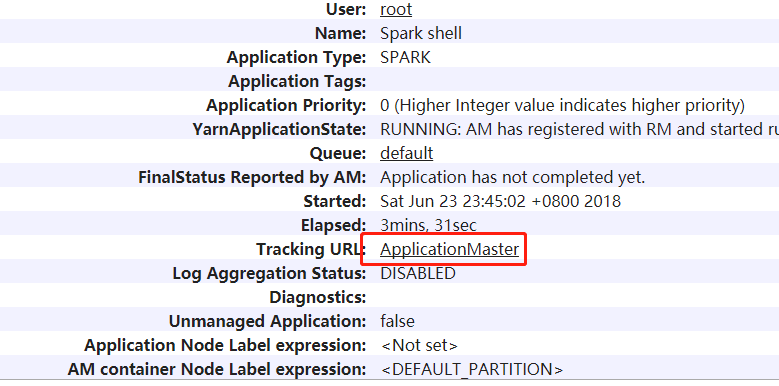
仅仅是负责申请资源，启动executor。

Driver进程，在客户端。



# spark-shell --master yarn --deploy-mode client

可以和standalone模式一样，查看任务运行的DAG，job，task等可视化信息。



## 多种模式下的比较

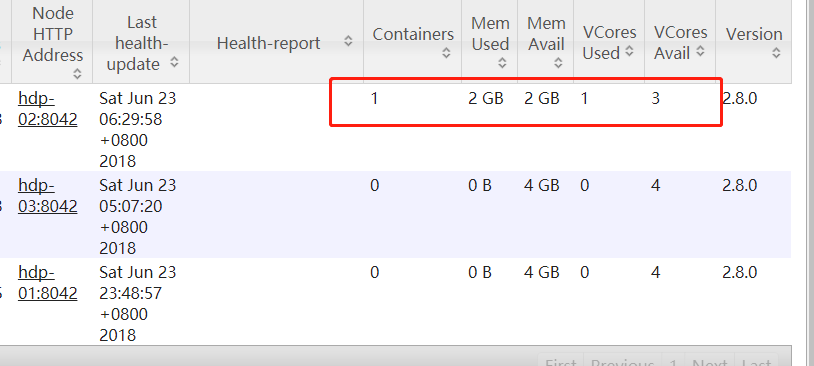


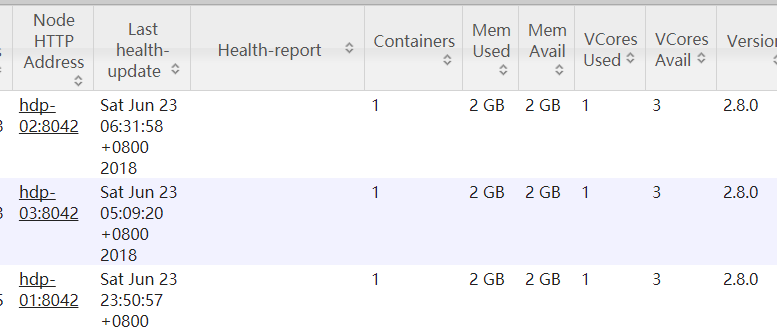
在yarn cluster模式下，Driver挂掉之后，可以在集群中重新启动。

yarn client,standalone， Driver不能挂掉的。

## yarncluster的资源配置：

### yarn-cluster模式下的默认分配：





ApplicationMaster ： 默认占用了 2g，1个cores

executor: 默认也占用了2g ,1 个cores

### yarn cluster模式下的自定义：

--executor-cores

--executor-memory

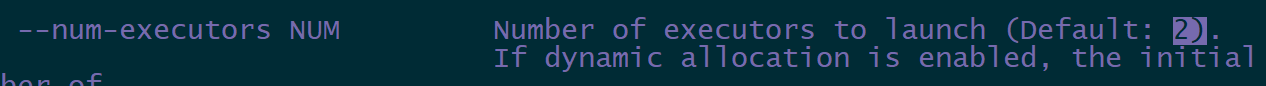
--num-executors 相当于standalone模式下的—total-executor-cores

--driver-memory

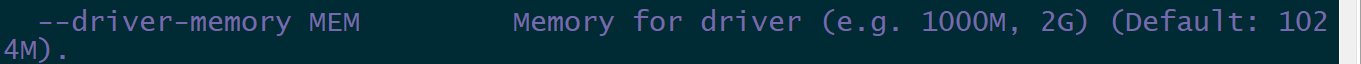
--driver-cores

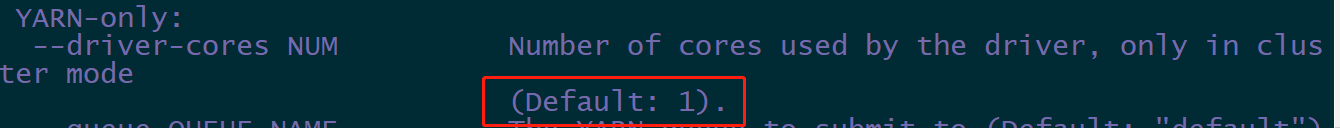
driver的配置，就是ApplicationMaster的配置。

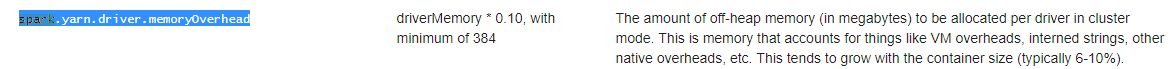




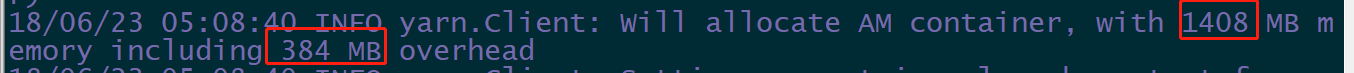






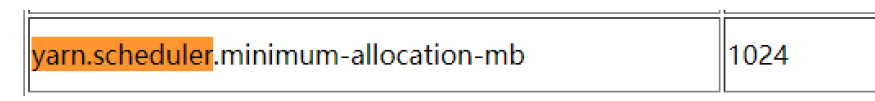


--driver-memory =1024 + 384M = 1408M

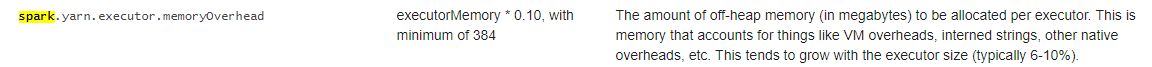


在yarn中，为每一个容器分配的最小内存是1g。 yarn中的资源分配，必须是整数倍。

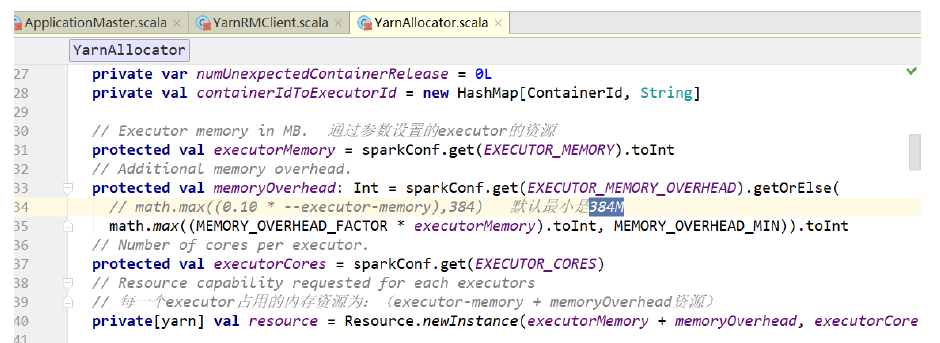
<http://hadoop.apache.org/docs/r2.8.4/hadoop-yarn/hadoop-yarn-common/yarn-default.xml>



--executor-memory =1024 + 384M = 1408M

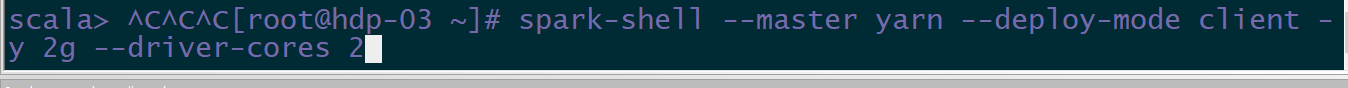


向上取整， 2g 内存。

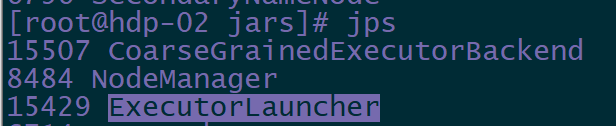


## yarn-client模式下的资源分配

--driver 就不能用了。





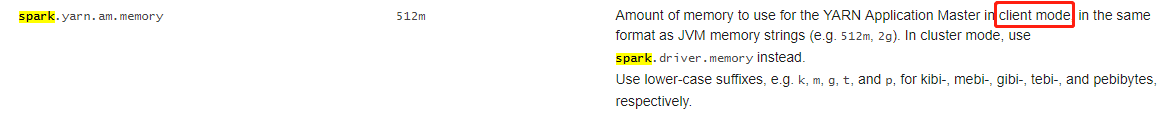


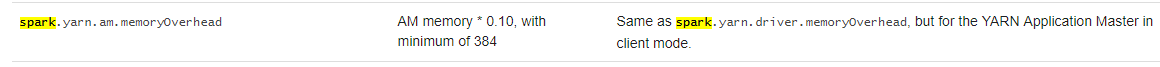
executor: 2g 1cores

ExecutorLauncher: 1g 1cores

这里通过--driver--xx设置的ExecutorLauncher的资源，是没有生效的。

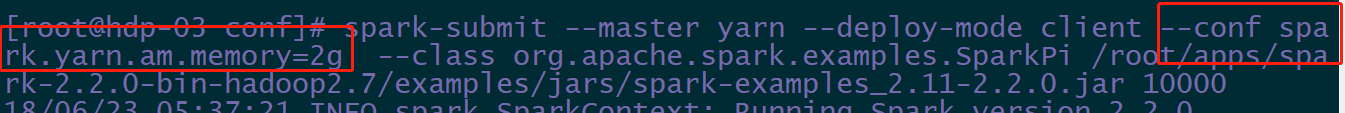
<http://spark.apache.org/docs/2.2.0/running-on-yarn.html>



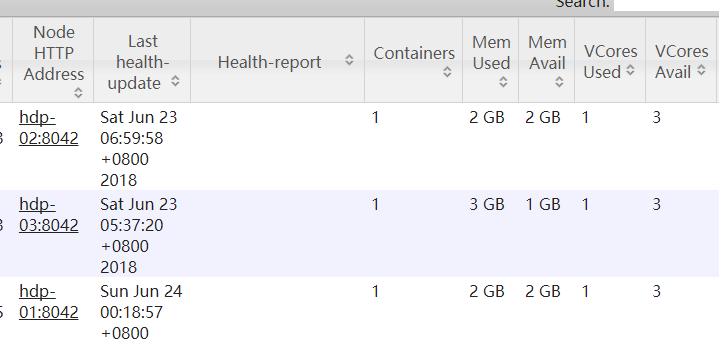


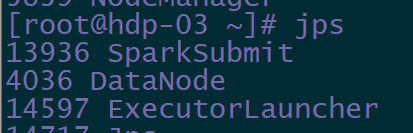
512m + 384m = 896M

--conf spark.yarn.am.memory=2g



2g + 384m = 向上取整 3g

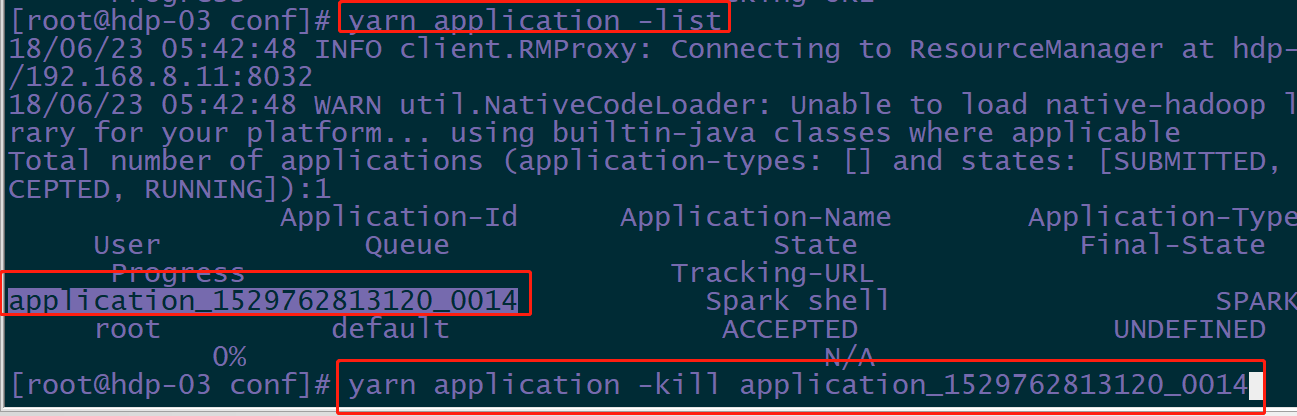




## yarn模式下，如何杀死程序

yarn application -list 查看 任务

yarn application -kill appId 删除程序



yarn-cluster模式 ，更常用。

需要客户端监控任务，可以选择yarn-client模式。

spark-shell 只能作用于yarn - client模式。