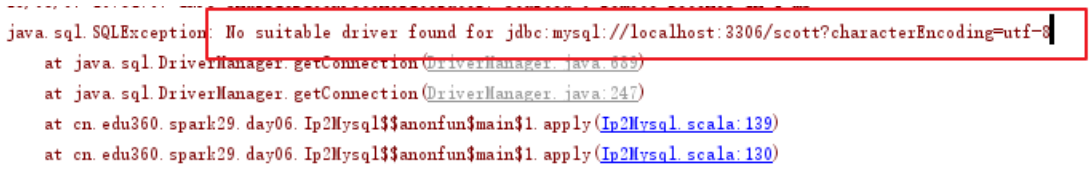
1. 综合案例：

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

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
}
```



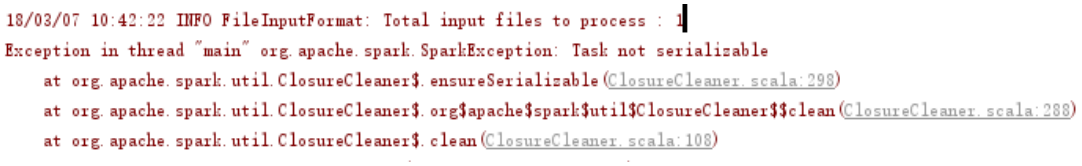
mysql 的 maven 依赖：

```
<!--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()
```

如果在 driver 端初始化了 conn 的对象，那么就会报序列化错误。



数据库连接方式一：

```
// 连接数据库方式
var conn: Connection = null
var pstm: PreparedStatement = null
try {
  conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/scott",
"root", "123")
  pstm = 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 =>
      pstm.setString(1, t._1)
      pstm.setInt(2, t._2)
      pstm.executeUpdate()
  })
} catch {
  case e: Exception => println(e.printStackTrace())
}
finally {
  // 关闭连接
  if (pstm != null) pstm.close()
  if (conn != null) conn.close()
}
```

如果把二分搜索的方法转换成函数，需要注意返回值：

```
// 有终止的条件
while (high >= low && !flag) {
  // 找中间值
  val middle = (low + high) / 2
  if (longIp >= ipRules(middle)._1 && longIp <=
    province = ipRules(middle)._3
    flag = true
  } else if (longIp < ipRules(middle)._1) { // 在
    // 修改最大值索引
    high = middle - 1
  } else { // 在右区间
    low = middle + 1
  }
```

如果把任务提交到集群中执行，报错如下：

```
18/03/07 23:20:42 INFO TransportClientFactory: Successfully created connection to /192.168.8.12:40501 after 5 ms (0 ms spent in bootstraps)
18/03/07 23:20:42 INFO ShuffleBlockFetcherIterator: Started 1 remote fetches in 27 ms
java.sql.SQLException: No suitable driver found for jdbc:mysql://172.16.1.191:3306/scott?characterEncoding=utf-8
        at java.sql.DriverManager.getConnection(DriverManager.java:689)
        at java.sql.DriverManager.getConnection(DriverManager.java:247)
        at cn.edu360.spark29.day06.IPUtils$$anonfun$4.apply(IPUtils.scala:105)
        at cn.edu360.spark29.day06.IPUtils$$anonfun$4.apply(IPUtils.scala:96)
        at org.apache.spark.rdd.RDD$$anonfun$foreachPartition$1$$anonfun$apply$29.apply(RDD.scala:926)
        at org.apache.spark.rdd.RDD$$anonfun$foreachPartition$1$$anonfun$apply$29.apply(RDD.scala:926)
        at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:2062)
        at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:2062)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
        at org.apache.spark.scheduler.Task.run(Task.scala:108)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:335)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)
```

解决方案：

```
[root@hdp-03 ~]# spark-submit  --master spark://hdp-01:7077 --jars mysql-connector-java-5.1.
38.jar --class cn.edu360.spark29.day06.Ip2MysqlBCHDFS spark29-1.0-SNAPSHOT.jar hdfs://hdp-01
:9000/access.log hdfs://hdp-01:9000/ip.txt
```

把程序执行的日志写入到一个文件中：

```
[root@hdp-03 ~]# nohup spark-submit  --master spark://hdp-01:7077 --jars mysql-connector-jav
a-5.1.38.jar --class cn.edu360.spark29.day06.Ip2MysqlBCHDFS spark29-1.0-SNAPSHOT.jar hdfs://
hdp-01:9000/access.log hdfs://hdp-01:9000/ip.txt  > sparkout.log 2>&1
```

输出重定向： 控制台 重定向 到文件中
1> sparklog2.log 2>&1 &
2>&1 错误的输出，写入到正确的输出文件中。
nohup & 把程序放在后台运行

2.Spark 共享变量

Spark 提供了两种有限类型的共享变量，广播变量和累加器。

2.1. 广播变量

2.1.1. 广播变量简述

使用广播变量，每个 Executor 的内存中，只驻留一份变量副本， 而不是对 每个 task 都传输一次大变量，省了很多的网络传输， 对性能提升具有很大帮助， 而且会通过高效的广播算法来减少传输代价。

使用广播变量的场景很多， spark 一种常见的优化方式就是小表广播， 使用 map join 来代替 reduce join，通过把小的数据集广播到各个节点上，节省了一次昂贵的 shuffle 操作。比如 driver 上有一张数据量很小的表， 其他节点上的 task 都需要 lookup 这张表， 那么 driver 可以先把这张表数据广播到各 executor 节点，这样 task 就可以在本地查表了。

2.1.2. 广播变量的原理

Spark 历史上采用了两种广播的方式，一种是通过 Http 协议传输数据（已废除）， 一种是通过 Torrent 协议来传输数据（`TorrentBroadcast`）。

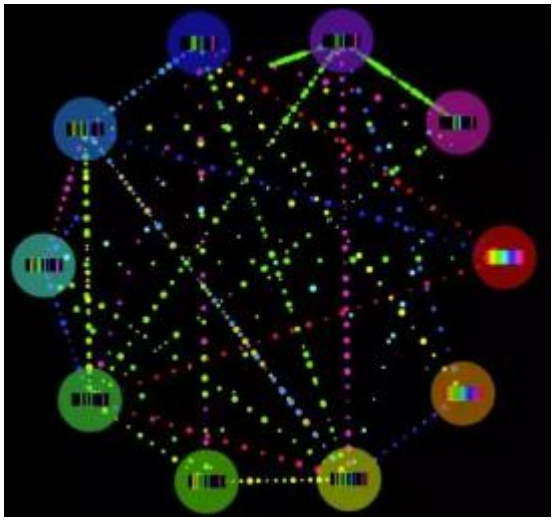
TorrentBroadcast 类似于比特洪流（BT 下载）

机制如下：

在 driver 端将序列化的对象分成小块，并将这些块存储在 driver 的 BlockManager 中。

在每个 executor 上，executor 首先尝试从 BlockManager 中获取对象。如果不存在，则使用远程网络传输从 driver 端 或其他 executor (如果可用) 中取出块。一旦它获得了块，它会将块放入其自己的 BlockManager 中，供其他执行者从中获取。

这可以防止 driver 程序成为发送广播数据的多个副本（每个 executor 一个）的瓶颈。



不同的数据块是来自不同的节点， 多个节点一起组成一个网络，在你下载的同时，你也在上传，所以说在享受别人提供的下载的同时，你也在贡献，最终所有人一起受益。

2.1.3. 广播变量的使用

通过在一个变量 v 上调用 SparkContext.broadcast(v)可以创建广播变量。广播变量是围绕着 v 的封装，可以通过 value 方法访问这个变量。

在 driver 端创建广播变量：

```
// 直接把 ipRules 广播出去
val ipbc: Broadcast[Array[(Long, Long, String)]] = sc.broadcast(ipRules)
```

使用的时候，通过代理对象的 value 方法获取广播的值：

```
// iprules 直接从广播变量中获取到  通过 value 获取到广播变量的值
val ipRulesfBC: Array[(Long, Long, String)] = ipbc.value
```

2.1.4. 广播变量总结

广播变量缓存到各个 executor 的内存中，而不是每个 Task，每个 executor 中的 task 共用一份广播数据。

广播变量被创建后，能在集群中运行的任何函数调用，通过广播变量.value 获取值

广播变量是只读的，只能在 executor 中读取，不能对该广播的变量进行修改

rdd 不能广播　不能在一个 RDD 中操作另外一个 RDD

> 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.

广播一个 rdd 时，报错如下：

```
Exception in thread "main" java.lang.IllegalArgumentException: requirement failed: Can not directly broadcast RDDs; instead, call collect() and broadcast the result.
    at scala.Predef$.require(Predef.scala:224)
    at org.apache.spark.SparkContext.broadcast(SparkContext.scala:1486)
    at cn.edu360.spark29.day06.Ip2Mysql2$.main(Ip2Mysql2.scala:99)
    at cn.edu360.spark29.day06.Ip2Mysql2.main(Ip2Mysql2.scala)
18/03/07 11:33:53 INFO SparkContext: Invoking stop() from shutdown hook
```

什么时候用广播变量：

中间数据，应用数据，需要广播。知识库的数据

数据特点：数据稳定，数据量不大，数据会被频繁使用

```
// 获取日志数据中的 ip 地址字段
val logData = logFile.map(_.split("\\|")(1))
```

2.2. 累加器

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。

累加器通过对一个初始化了的变量 v 调用 SparkContext.accumulator(v)来创建。在集群上运行的任务可以通过 add 或者"+="方法在累加器上进行累加操作。但是，它们不能读取它的值。只有 Driver 能够读取它的值，通过累加器的 value 方法。

累加器只支持**加法**操作，可以高效地并行，用于实现计数器和变量求和。

spark-shell 中操作：

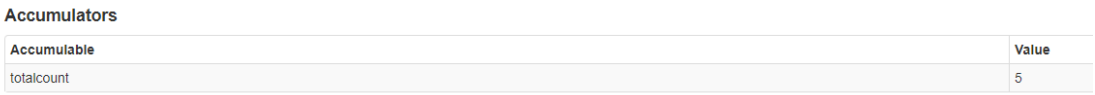
```
// 第一步： 定义一个累加器

val accum = sc.accumulator(0,"totalCounts")

// 第二步： 使用 添加 add（） +=

sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += 1)

查看值：

accum.value
```

idea 中定义：

```
// 定义一个累加器
val ac  = sc.accumulator(0L,"totaltime")

val ac2  = sc.accumulator(0L,"totalSum")

// 2.0API
val lac = sc.longAccumulator("name")
val ipRule: RDD[(Long, Long, String)] = data.map(t=>
  {
```

```
    val t1 = System.currentTimeMillis()
    val ipdatas = t.split("[|]")
    val start = ipdatas(2).toLong
    val end = ipdatas(3).toLong
    val province = ipdatas(6)
    val t2 = System.currentTimeMillis()
    // 进行统计
    ac.add(t2-t1)
    (start, end, province)
})

 println(ac.value)
```

可以在监控页面显示累加器的值：



Accumulators

Accumulable	Value
totalcount	5

累加器在 spark1.x 和 spark2.x 的版本中使用方式有细微差别：

```
Spark1.x:
scala> val accum1 = sc.accumulator(0, "My Accumulator")
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x) // add()
scala> accum1.value
Spark2.x:
scala> val accum2 = sc.longAccumulator("MyAccumulator2")
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum2.add(x)
scala> accum2.value
```

想要统计次数，就需要使用累加器。

```
val rdd = sc.makeRDD(List(1, 3, 4, 65, 7))
var i = 0
rdd.foreach(t => i += 1)
println(i)
```

1，在 driver 端查看累加器的值，用.value，再 executor 中，直接查看 通过累加器的 name。

2，累加器在一个 application 中，是被共享的。

3. 序列化：

使用 kryo 序列化机制：

```
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
```

3.1. 在函数中调用

如果直接在函数中调用类的实例

该类不需要进行序列化。

原因：数据并没有 driver 和 executor 之间传输，类是在 executor 中实例化的

task 在处理每一条数据的时候，都需要对类进行实例化。

```
val f1 = sc.textFile(args(0))
val result = f1.map(t => {
  // 创建一个实例 该实例在哪里创建？ executor driver task？
  val mt = new MyTask()
  // 打印当前的对象
  println(s"@@@@@@@@@@@@@@+++${mt.toString}+++@@@@@@@@@@@@@@")
  // 通过该对象的集合获取值
  mt.mp.getOrElse(t, 0)
  // 获取当前的主机名
  val hostName = InetAddress.getLocalHost.getHostName
  // 获取当前的线程名
  val threadName = Thread.currentThread().getName
  (hostName, threadName ,mt.mp.getOrElse(t,0),mt.toString)
```

```
)})
result.saveAsTextFile(args(1))
sc.stop()

// 类的定义
class MyTask {
  val mp = Map("hadoop" -> 10, "spark" -> 1000)
}
```

3.2. 闭包引用

```
Caused by: java.io.NotSerializableException: cn.edu360.spark30.day06.MyTask
Serialization stack:
        - object not serializable (class: cn.edu360.spark30.day06.MyTask, value:
cn.edu360.spark30.day06.MyTask@72fd8a3c)
        - field (class: cn.edu360.spark30.day06.SerTest3$$anonfun$2, name: task$1
, type: class cn.edu360.spark30.day06.MyTask)
```

当我们在函数内部使用了一个外部的引用，该引用对象对应的类必须要进行序列化。 extends Serializable

在 driver 端的对象和 executor 中的对象，不是同一个，执行 task 的时候，进行反序列化，获取到了新的对象。

类是在 driver 端实例化了，但是在 executor 中，每一个 task 对应着一个新的实例对象。

（同一个 task 中的数据，共用一个实例）

```
val f1 = sc.textFile(args(0))

// 在 driver 端创建一个类的实例对象 该类必须实现序列化
val mt = new MyTask()
// 打印当前的实例对象
println(s"@@@@@@@@@@@@@@+++${mt.toString}+++@@@@@@@@@@@@@@")
val result = f1.map(t => {
  // 获取当前的主机名
  val hostName = InetAddress.getLocalHost.getHostName
  // 获取当前的线程名
  val threadName = Thread.currentThread().getName
  (hostName, threadName, mt.mp.getOrElse(t, 0), mt.toString)
```

```
})
result.saveAsTextFile(args(1))
sc.stop()


// 序列化
class MyTask extends Serializable{
  val mp = Map("hadoop" -> 10, "spark" -> 1000)
}
```

当在函数内部使用了一个外部的引用，就生成一个闭包。

闭包是可以包含自由（未绑定到特定对象）变量的代码块；这些变量不是在这个代码块内

或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。比如说，

在函数中使用定义在其外的局部变量，这就形成了一个闭包。

var sum = 0

List(1,2,3,4,5).foreach(x => sum += x)

在 scala 中，闭包捕获了变量本身，而不是变量的值。

闭包的本质：代码块+上下文

3.3.广播变量

同一个 executor 中的所有 task 共享一份数据。

```
val f1 = sc.textFile(args(0))

// 在driver 端创建一个类的实例对象
val mt = new MyTask()
val bc = sc.broadcast(mt)
// 打印当前的实例对象
println(s"@@@@@@@@@@@@@@+++${mt.toString}+++@@@@@@@@@@@@@@")
val result = f1.map(t => {
```

```
  val newTask = bc.value
  // 获取当前的主机名
  val hostName = InetAddress.getLocalHost.getHostName
  // 获取当前的线程名
  val threadName = Thread.currentThread().getName
  (hostName, threadName, newTask.mp.getOrElse(t, 0), newTask.toString)
})
result.saveAsTextFile(args(1))
sc.stop()
```