Spark Task序列化代码分析

Spark的作业会通过DAGScheduler的处理生产许多的Task并构建成DAG图，而分割出的Task最终是需要经过网络分发到不同的Executor。在分发的时候，Task一般都会依赖一些文件和Jar包，这些依赖的文件和Jar会对增加分发的时间，所以Spark在分发Task的时候会将Task进行序列化，包括对依赖文件和Jar包的序列化。这个是通过spark.closure.serializer参数设置的。我们可以看源码的实现：

```
// Serializer for closures and tasks.
val env = SparkEnv.get
val ser = env.closureSerializer.newInstance()
```

这就是获取序列化类的代码，而closureSerializer就是SparkEnv中的，我们来看它是如何初始化closureSerializer的：

```
val closureSerializer = instantiateClassFromConf[Serializer](
  "spark.closure.serializer", "org.apache.spark.serializer.JavaSerializer")
```

这就是通过获取spark.closure.serializer参数的设置，目前Task的序列号只支持Java序列化，在文档http://spark.apache.org/docs/latest/streaming-programming-guide.html#task-launching-overheads里面描述有误，我已经提交issue进行修正：https://github.com/apache/spark/pull/9734。

在初始化好closureSerializer 之后，我们就可以对Task进行序列化了：

```
val serializedTask: ByteBuffer = try {
      Task.serializeWithDependencies(task, sched.sc.addedFiles, sched.sc.addedJars, ser)
```

这就是序列化Task，我们再来看看serializeWithDependencies的实现：

```
//////////////////////////////////////////////////////////////////////
User: 过往记忆
Date: 2015-11-16
Time: 23:59
```

```
 bolg:
 本文地址：/archives/1531
 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 过往记忆博客微信公共帐号：iteblog_hadoop
/////////////////////////////////////////////////////////////////////////
def serializeWithDependencies(
    task: Task[_],
    currentFiles: HashMap[String, Long],
    currentJars: HashMap[String, Long],
    serializer: SerializerInstance)
  : ByteBuffer = {

  val out = new ByteArrayOutputStream(4096)
  val dataOut = new DataOutputStream(out)

  // Write currentFiles
  dataOut.writeInt(currentFiles.size)
  for ((name, timestamp) <- currentFiles) {
    dataOut.writeUTF(name)
    dataOut.writeLong(timestamp)
  }

  // Write currentJars
  dataOut.writeInt(currentJars.size)
  for ((name, timestamp) <- currentJars) {
    dataOut.writeUTF(name)
    dataOut.writeLong(timestamp)
  }

  // Write the task itself and finish
  dataOut.flush()
  val taskBytes = serializer.serialize(task).array()
  out.write(taskBytes)
  ByteBuffer.wrap(out.toByteArray)
}
```

没错，这个就是Java序列化的经典代码，这段代码将依赖文件和jar进行了序列化。而序列化的Task最终将在Exectuor中进行反序列化，如下：

```
val (taskFiles, taskJars, taskBytes) = Task.deserializeWithDependencies(serializedTask)
```

deserializeWithDependencies就是反序列化的核心代码：

```
//////////////////////////////////////////////////////////////////////
 User: 过往记忆
 Date: 2015-11-16
 Time: 23:59
 bolg:
 本文地址：/archives/1531
 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 过往记忆博客微信公共帐号：iteblog_hadoop
//////////////////////////////////////////////////////////////////////
def deserializeWithDependencies(serializedTask: ByteBuffer)
  : (HashMap[String, Long], HashMap[String, Long], ByteBuffer) = {

  val in = new ByteBufferInputStream(serializedTask)
  val dataIn = new DataInputStream(in)

  // Read task's files
  val taskFiles = new HashMap[String, Long]()
  val numFiles = dataIn.readInt()
  for (i <- 0 until numFiles) {
    taskFiles(dataIn.readUTF()) = dataIn.readLong()
  }

  // Read task's JARs
  val taskJars = new HashMap[String, Long]()
  val numJars = dataIn.readInt()
  for (i <- 0 until numJars) {
    taskJars(dataIn.readUTF()) = dataIn.readLong()
  }

  // Create a sub-buffer for the rest of the data, which is the serialized Task object
  val subBuffer = serializedTask.slice()  // ByteBufferInputStream will have read just up to task
  (taskFiles, taskJars, subBuffer)
}
```

反序列化之后，Exectuor将会通过网络把这些依赖的文件和Jar包下载下来，最终启动Task。