day4笔记

复习

第一梯队：mr/kafka/spark/flink

第二梯队：hdfs/zookeeper/hive/hbase/scala

第三梯队：flume/sqoop/azkaban/

1. 状态存储数据

   managed keyed state

   基于keyedStream可以管理的状态

   valueState/listState/mapState/reducingState/aggregatingState

   flink支持自定义计算函数

   > flink流计算官方教程：
   >
   > https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/try-flink/datastream/

   可以在计算函数中（RichMapFunction）完成状态的使用

   1. 重写open方法，在里面完成状态对象的创建
   2. 在map方法，实现状态的方法的使用《根据需求、根据状态提供的方法》

2. TTL

   状态可以设置剩余存活时间

   可以设置时间的更新机制

   过期数据的清理机制：snapshot、incremental、压实机制

3. checkpoint

   1. 由jobmanager发起：

      在数据流中添加barrier从而发起checkpoint

   2. 由taskmanager具体执行

      每一个算子遇到barrier就会做checkpoint的预提交处理。所有的算子都已经预提交处理，这一次checkpoint才算完成

   3. 一个新的checkpoint完成，原来的就会丢弃掉

4. state backend

   配置状态后端：状态最后要持久化到哪里，jobmanager、**filesystem**、rocksdb

   fileysystem的使用

   - 配置目录
   - hdfs，需要flink和hadoop的集成

状态管理

Broadcast State Pattern 广播状态

广播状态是Flink提供的第三种状态共享的场景。通常需要将一个吞吐量比较低的流中的状态数据进行广播

给下游的任务，另外一个流可以以只读的形式读取广播状态

non-keyed Stream connect BroadcastStream

需要继承**BroadcastProcessFunction**，实现里面的两个方法

> - processElement
>
>   可以获取到低吞吐量流广播过来的状态，处理高吞吐量流相关的业务逻辑
> - processBroadcastElement
>
>   用来处理广播流，即对低吞吐量流进行处理

案例需求：把符合过滤规则的内容过滤掉

业务需求：把评论中的某些内容过滤掉

- 评论内容--->数据量比较大，高吞吐量
- 需要过滤的内容-->数据量比较小的，需要广播的流。需要过滤的内容，应该广播到评论流里面

```
import org.apache.flink.api.common.state.MapStateDescriptor
import org.apache.flink.streaming.api.functions.co.BroadcastProcessFunction
import org.apache.flink.streaming.api.scala.OutputTag
import org.apache.flink.util.Collector

/**
  * 三个泛型分别表示
  * The input type of the non-broadcast side==》高吞吐量的流的类型，不需要广播的流
  * The input type of the broadcast side==》低吞吐量的流的类型，需要广播的流
  * The output type of the operator==》输出的流类型
  */
class NonKeyedStreamBroadcast(outputTag: OutputTag[String],mapStateDescriptor:
MapStateDescriptor[String,String]) extends BroadcastProcessFunction[String,String,String]{
  /**
    * 处理高吞吐量流
    * @param value 高吞吐量流对应的数据
    * @param ctx
    * @param out
    */
  override def processElement(value: String, ctx: BroadcastProcessFunction[String, String,
String]#ReadOnlyContext, out: Collector[String]): Unit = {

    //获取到只读broadcastState对象
    val readOnlyBroadcastState = ctx.getBroadcastState(mapStateDescriptor)
    if(readOnlyBroadcastState.contains("rule")){

      if(value.contains(readOnlyBroadcastState.get("rule"))){
        //non-broadcastStream中符合过滤规则
        out.collect("过滤规则是: "+readOnlyBroadcastState.get("rule")+", 符合过滤规则的数据是: "+value)
      }else{
        ctx.output(outputTag,value)
      }

    }else{
      println("rule 判断规则不存在")
      //通过side out将数据输出
      ctx.output(outputTag,value)
    }

  }

  /**
    * 处理低吞吐量流
    * @param value 低吞吐量流对应的数据
    * @param ctx
    * @param out
    */
  override def processBroadcastElement(value: String, ctx: BroadcastProcessFunction[String, String, String]#Context, out: Collector[String]): Unit = {

    //把broadcastStream中的数据放入到broadcastState中==》把过滤规则广播出去
    val broadcastState = ctx.getBroadcastState(mapStateDescriptor)
    broadcastState.put("rule",value)
  }
}
```

```
import org.apache.flink.api.common.state.MapStateDescriptor
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._

object NonKeyedStreamBroadcastCode {
  def main(args: Array[String]): Unit = {
    //1.执行环境
    val environment = StreamExecutionEnvironment.getExecutionEnvironment

    //2.dataSource
    //高吞吐量流: non-broadcasted stream
    val  highThroughputStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    //低吞吐量流: broadcastStream, 需要通过broadcast方法获取
    var mapStateDescriptor = new MapStateDescriptor("mapStateDescriptor",createTypeInformation[String],createTypeInformation[String])
    var lowThroughputStream = environment.socketTextStream("flink.baizhiedu.com",8888).broadcast(mapStateDescriptor)

    //non-broadcasted stream通过connect方法连接broadcastStream, 得到BroadcastConnectedStream

    val broadcastConnectedStream = highThroughputStream.connect(lowThroughputStream)
    var outputTag=new OutputTag[String]("non-match")

    //BroadcastConnectedStream对象提供的有process方法，可以完成业务逻辑处理
    val dataStream = broadcastConnectedStream.process(new
NonKeyedStreamBroadcast(outputTag,mapStateDescriptor))
    dataStream.print("匹配规则")
    dataStream.getSideOutput(outputTag).print("不匹配规则")

    environment.execute("nonKeyedStreamBroadcastJob")
  }

}
```

> 可以应用在舆情监控上

扩展：敏感词可以加减操作

Keyed Stream connect BroadcastStream

需要继承**KeyedBroadcastProcessFunction**

案例需求： 某电商平台，用户在某一类别下消费总金额达到一定数量，会有奖励

分析：

1. 不同类别会有对应的奖励机制，需要把这个奖励机制广播给用户消费对应的流
2. 用户的消费应该是一个高吞吐量流
3. 通过用户消费流连接奖励机制流，然后通过process处理
4. 用户消费流应该根据用户标记以以及类别分组===》流是KeyedStream
   ProcessFunction应该选中KeyedBroadcastProcessFunction
5. 在KeyedBroadcastProcessFunction中完成奖励机制以及用户消费统计、分析、处理

```
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.state.{MapStateDescriptor, ReducingState,
ReducingStateDescriptor}
import org.apache.flink.streaming.api.functions.co.KeyedBroadcastProcessFunction
import org.apache.flink.streaming.api.scala.OutputTag
import org.apache.flink.util.Collector
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._

class KeyedStreamBroadcast(outputTag: OutputTag[String], mapStateDescriptor:
MapStateDescriptor[String,Double]) extends
KeyedBroadcastProcessFunction[String,OrderItem,Rule,User]{

  var orderTotalAmountState:ReducingState[Double]=_
  override def open(parameters: Configuration): Unit = {


    orderTotalAmountState=getRuntimeContext.getReducingState(new ReducingStateDescriptor[Double]
("userTotalAmount",new ReduceFunction[Double]() {
      override def reduce(value1: Double, value2: Double): Double = value1+value2
    },createTypeInformation[Double]))

  }

  //处理nonBroadcastStream
  override def processElement(value: OrderItem, ctx: KeyedBroadcastProcessFunction[String, 
OrderItem, Rule, User]#ReadOnlyContext, out: Collector[User]): Unit = {

    val broadcastState = ctx.getBroadcastState(mapStateDescriptor)

    //将本次订单金额累计到历史订单总金额
    var thisorderTotalAmount = value.count*value.price
    orderTotalAmountState.add(thisorderTotalAmount)


    if(broadcastState!=null&broadcastState.contains(value.category)){
      //类别下对应的threshold
      val threshold = broadcastState.get(value.category)

      var orderTotalAmount=orderTotalAmountState.get()
      if(orderTotalAmount>=threshold){
        //符合奖励规则
        //将符合奖励规则的用户输出到下游
        out.collect(new User(value.userId,value.username))
      }else{
        //不符合奖励规则
        ctx.output(outputTag,"您还差"+(threshold-orderTotalAmount)+"就可以获得奖励")
      }

    }else{
      //value.category分类下还没有设置奖励规则
      ctx.output(outputTag,"奖励规则制定中，会有很多丰厚礼品，请抓紧时间购买")
    }


  }

  //处理broadcastStream
  override def processBroadcastElement(value: Rule, ctx: KeyedBroadcastProcessFunction[String, 
OrderItem, Rule, User]#Context, out: Collector[User]): Unit = {

    val broadcastState = ctx.getBroadcastState(mapStateDescriptor)
    broadcastState.put(value.category,value.threshold)
  }
}
```

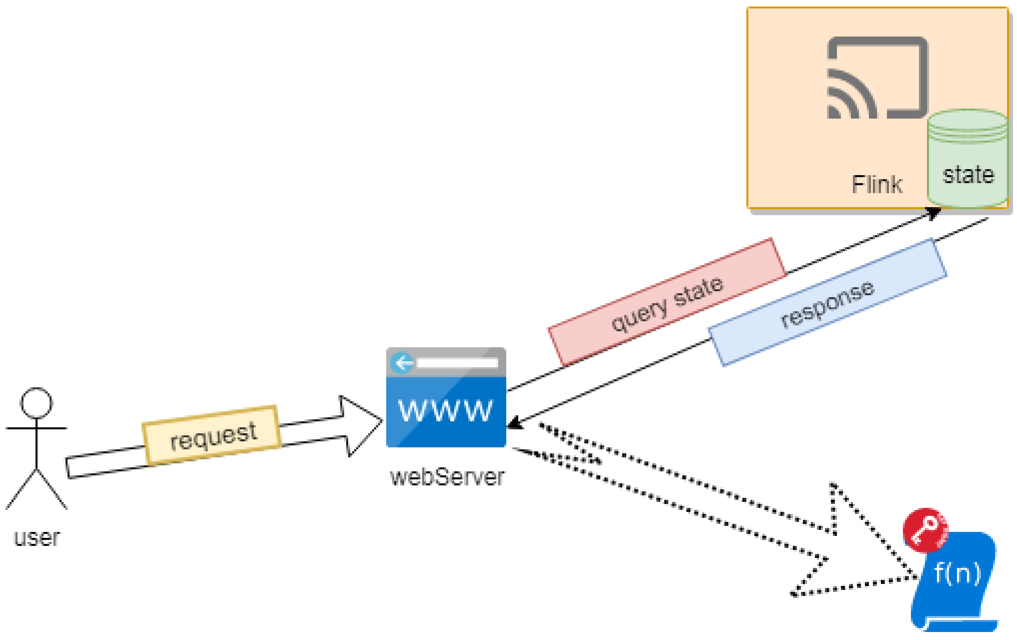
```
import org.apache.flink.api.common.state.MapStateDescriptor
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._
```

```
/**
  * 用户类
  *
  * @param id
  * @param name
  */
case class User(id:String,name:String)

/**
  * 规则类，也就是奖励类
  * @param category 类别
  * @param threshol 对应类别下的阈值
  */
case class Rule(category:String,threshold:Double)

/**
  * 订单详细类
  * @param userId
  * @param username 用户名
  * @param category 类别
  * @param productName 商品名
  * @param count 商品数量
  * @param price 单价
  */
case class
OrderItem(userId:String,username:String,category:String,productName:String,count:Int,price:Doubl
e)

object KeyedStreamBroadcastCode {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment

    //高吞吐量流
    //数据输入要求： 按照订单详情类中的属性顺序输入
    //例如==》 101 zhangsan 电子类 手机 1 2300
    val highThroughputStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val nonBroadcastStream = highThroughputStream.map(line => line.split("\\s+"))
      .map(words => OrderItem(words(0), words(1), words(2), words(3), words(4).toInt,
words(5).toDouble))
      .keyBy(orderItem => orderItem.userId + ":" + orderItem.category)

    //MapStateDescriptor
    var mapStateDescriptor = new MapStateDescriptor[String,Double]
("broadcastStreamMapStateDescriptor",createTypeInformation[String],createTypeInformation[Double]
)

    //低吞吐量流
    //数据输入要求：按照Rule类的属性顺序输入
    //例如==》 电子类 5000
    val lowThroughputStream = environment.socketTextStream("flink.baizhiedu.com",8888)


    val broadcastStream = lowThroughputStream.map(line => line.split("\\s+"))
```

```
        .map(words => Rule(words(0), words(1).toDouble))
        .broadcast(mapStateDescriptor)

    //连接
    val broadcastConnectedStream = nonBroadcastStream.connect(broadcastStream)

    var outputTag = new OutputTag[String]("没有奖励")

    //process
    val dataStream = broadcastConnectedStream.process(new
KeyedStreamBroadcast(outputTag,mapStateDescriptor))

    dataStream.print("奖励: ");
    dataStream.getSideOutput(outputTag).print("没有奖励")

    environment.execute("keyedStreamBroadcast")

  }

}
```

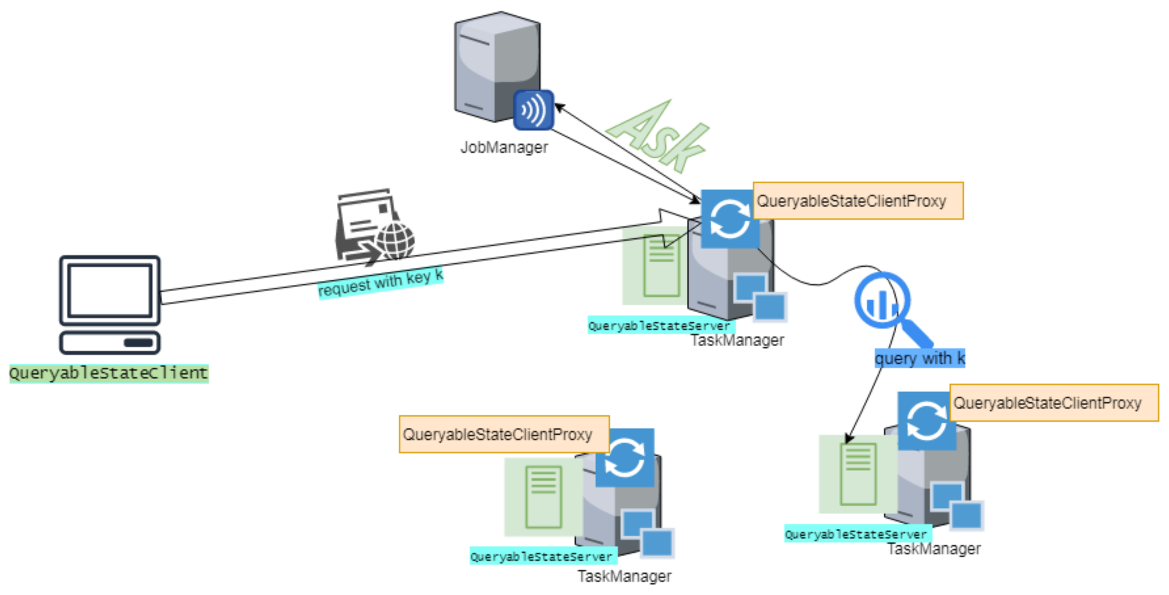
Queryable State 状态可查询☆

> Flink提供的有状态可查询机制，可以通过第三方程序读取到flink内部的状态数据



Queryable State简单讲，就是flink技术之后的结果（state），允许通过第三方应用程序查询到

Architecture（架构）

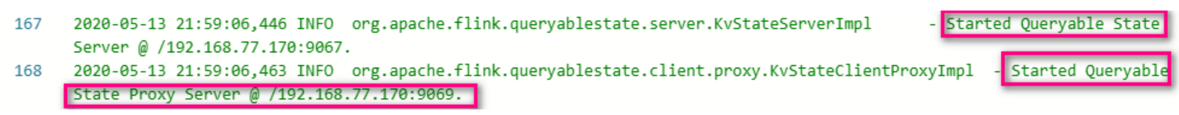


> 以上图对应的说明
>
> 1. 在Flink的状态可查询的架构中，存在三个基本概念
>
>   QueryableStateClient:第三方程序，不是flink架构中的内容
>
>   QueryableStateClientProxy:flink架构中的一部分，用来处理客户端的请求
>
>   QueryableStateServer:flink架构中的一部分，查询状态服务端（可查询的状态都在这里面）
> 2. flink状态可查询的执行
>   - 客户端发送状态可查询请求给taskManager中的QueryableStateClientProxy
>
>   通过key查询对应的状态数据
>   - queryableStateClientProxy根据key到jobManager中获取到这个key对应的状态存储在哪个taskmanager上面
>   - 根据key到指定的taskmanager上面的queryableStateServer中获取到这个key对应的状态
>   - 之后将获取的状态返回给客户端

Activating Queryable State

要激活Queryable State，需要做以下几步操作：

> 1. 把Flink的opt目录下的flink-queryable-state-runtime_2.11-1.10.0.jar文件复制到Flink的lib目录下
>
> ```
> [root@flink flink-1.10.0]# pwd
> /opt/install/flink-1.10.0
> [root@flink flink-1.10.0]# cp opt/flink-queryable-state-runtime_2.11-1.10.0.jar lib
> ```
>
> 2. 在Flink的配置文件conf/flink-conf.yaml中添加以下配置
>
> ```
> queryable-state.enable: true
> ```
>
> 3. 重新启动Flink
>
> 如果能在taskManager的日志文件中看到以下信息，就说明激活了Queryable State

> `Started the Queryable State Proxy Server @ ...`
>
> 
>
> ```
> 2020-05-13 21:59:06,446 INFO  org.apache.flink.queryablestate.server.KvStateServerImpl      - Started Queryable State
> Server @ /192.168.77.170:9067.
> 2020-05-13 21:59:06,463 INFO  org.apache.flink.queryablestate.client.proxy.KvStateClientProxyImpl  - Started Queryable
> State Proxy Server @ /192.168.77.170:9069.
> ```

Making State Queryable

可以通过以下两种方式让state在外部系统中可见：

- 创建QueryableStateStream，该Stream只是充当一个sink，将数据存储到queryablestate中
- 通过stateDescriptor.setQueryable(String queryableStateName)方法，将state可查

Queryable State Stream（了解）

通过KeyedStream对象的asQueryableState(stateName, stateDescriptor)方法，可以得到一个QueryableStateStream对象，这个对象提供的状态值是可查询的

```
// ValueState
QueryableStateStream asQueryableState(
    String queryableStateName,
    ValueStateDescriptor stateDescriptor)

// Shortcut for explicit ValueStateDescriptor variant
QueryableStateStream asQueryableState(String queryableStateName)

// FoldingState
QueryableStateStream asQueryableState(
    String queryableStateName,
    FoldingStateDescriptor stateDescriptor)

// ReducingState
QueryableStateStream asQueryableState(
    String queryableStateName,
    ReducingStateDescriptor stateDescriptor)
```

> **Note:** There is no queryable `ListState` sink as it would result in an ever-growing list which may not be cleaned up and thus will eventually consume too much memory.

返回的QueryableStateStream可视为sink，无法进一步转换。在内部，将QueryableStateStream转换为一个operator，这个operator将所有传入记录用来更新queryable state实例。更新逻辑在调用asQueryableState方法时传递的StateDescriptor参数对象中完成。在如下程序中，Keyed Stream的所有记录在底层都是通过value state.update（value）更新状态实例：

```
stream.keyBy(0).asQueryableState("query-name")
```

```
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.state.ReducingStateDescriptor
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._


object WordCountQueryableState {
```

```
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment

    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    var reducingStateDescriptor=new ReducingStateDescriptor[(String,Int)]
("reducingStateDescriptor",new ReduceFunction[(String,Int)] {
      override def reduce(value1: (String, Int), value2: (String, Int)): (String, Int) = {
        (value1._1,(value1._2+value2._2))
      }
    },createTypeInformation[(String,Int)])

    dataStream.flatMap(line=>line.split("\\s+"))
      .map(word=>(word,1))
      .keyBy(0)
      .asQueryableState("wordCountqueryableStateName",reducingStateDescriptor)

    environment.execute("wordCountQueryableStateJob")
  }

}
```

Managed Keyed State

可以通过StateDescriptor.setQueryable(String queryableStateName)方法实现managed keyed State状态可查

```
import org.apache.flink.api.common.functions.RichMapFunction
import org.apache.flink.api.common.state.{ValueState, ValueStateDescriptor}
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._

class MyMapFunction extends  RichMapFunction[(String,Int),(String,Int)]{
  var valueState:ValueState[Int]=_


  override def open(parameters: Configuration): Unit = {
    val runtimeContext = getRuntimeContext

    var valueStateDescriptor=new ValueStateDescriptor[Int]
("valueStateDescriptor",createTypeInformation[Int])

    valueStateDescriptor.setQueryable("WordCountQueryableStateManagedKeyedStateName")

    valueState=runtimeContext.getState(valueStateDescriptor)
  }

  override def map(value: (String, Int)): (String, Int) = {
    val oldValue = valueState.value()

    var newValue = valueState.update(oldValue+value._2)

    (value._1,valueState.value())
  }
}
```

```
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._

object WordCountQueryableStateManagedKeyedState {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment

    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    dataStream.flatMap(line=>line.split("\\s+"))
        .map(word=>(word,1))
        .keyBy(0)
        .map(new MyMapFunction)
        .print()

    environment.execute("WordCountQueryableStateManagedKeyedState")
  }
}
```

Querying State

- 引入依赖

  ```
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-core</artifactId>
    <version>1.10.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-queryable-state-client-java</artifactId>
    <version>1.10.0</version>
  </dependency>
  ```

- 代码实现

  ```
  import java.util.concurrent.CompletableFuture
  import java.util.function.Consumer

  import org.apache.flink.api.common.JobID
  import org.apache.flink.api.common.functions.ReduceFunction
  import org.apache.flink.api.common.state.{ReducingState, ReducingStateDescriptor}
  import org.apache.flink.streaming.api.scala._
  object QueryableClient {
    def main(args: Array[String]): Unit = {
      import org.apache.flink.queryablestate.client.QueryableStateClient

      val client = new QueryableStateClient("flink.baizhiedu.com", 9069)
  ```

```
    var reducingStateDescriptor=new ReducingStateDescriptor[(String,Int)]
("reducingStateDescriptor",new ReduceFunction[(String,Int)] {
      override def reduce(value1: (String, Int), value2: (String, Int)): (String, Int) = {
        (value1._1,(value1._2+value2._2))
      }
    },createTypeInformation[(String,Int)])

    var jobId =JobID.fromHexString("1f8ade8cf2d956bf553f0348a79c3f6e")
    val completableFuture: CompletableFuture[ReducingState[(String, Int)]] =
client.getKvState(jobId,"wordCountqueryableStateName","this",createTypeInformation[String],
reducingStateDescriptor)

    //同步获取数据
    /*val reducingState: ReducingState[(String, Int)] = completableFuture.get()
    print(reducingState.get())
    client.shutdownAndWait();*/


    //异步获取数据
    completableFuture.thenAccept(new Consumer[ReducingState[(String,Int)]] {
      override def accept(t: ReducingState[(String, Int)]): Unit = {
        print(t.get())
      }
    })

    Thread.sleep(1000)
    client.shutdownAndWait()
  }

}
```

如果创建了单独的module，还需要引入以下依赖才可以正常运行客户端程序

```
<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-streaming-scala -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```