day3笔记

复习

1. 状态就是用来存储数据的，在内存中存储数据

   flink的状态远不止这个；状态的应用特别广泛

2. flink的状态分类

   1. 按照是否可管理：**manage state**+raw state
   2. 按照对应的数据流划分：**keyed state**+non-keyed state(opertor state)

   manage keyed state

3. flink提供了一些api可以完成对状态的数据存储的开发处理

状态	描述	方法
ValueState	存储一个值	value/update
ListState	存储多个值	get/add/update
MapState	存储多个key-value结构的数据	contains/put/get/keys
ReducingState	存储一个值；可以自动运算；要求输入类型和输出类型必须一致	add/get
AggregatingState	存储一个值；可以自动运算；输入类型和输出类型可以不一致	add/get

   所有的状态都有一个方法：clear-->清空、清除

4. 状态存储数据的开发
   - 需要获取到keyedStream：dataStream.keyBy(分组依据)
   - keyedStream.map(自定义MapFunction)
   - 自定义MapFunction
     - 写一个类，继承RichMapFunction
     - 在mapfunction里面重写map方法：就是对数据进行映射处理的方法

       方法参数就是接收到的数据；方法返回值就是计算完成之后的结果

       通过状态提供的方法完成状态的数据处理
     - 重写open方法：创建状态对象
       1. 需要创建状态描述者XxxStateDescriptor:唯一标记（名字）、类型信息、默认值
       2. 获取到RuntimeContext对象；getRuntimeContext
       3. 根据运行时上下文对象（运行时工厂）获取到对应的状态

          里面提供的有方法getState/getListState/getMapState..

在使用方法的是，需要一个状态描述者

5. 代码：

   刚开始写的时候，可以从下往上写

   写熟练了之后，就可以随便写

状态管理

参考`https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/stream/state/state.html`

Flink是基于状态的流计算引擎。

在Flink中有两种基本类型的state，分别是`Keyed State`和`Operator State`。Keyed State只能应用在KeyedStream上的操作。每一个keyed operator都会绑定一个或多个状态值。Operator State又被称为non-keyed state，每一个算子都会有对应的operator state。

Keyed State以及Operator State都会以两种方式存储：managed和raw。

managed state指的是由Flink控制state的数据结构，比如使用内部hash表、RocksDB等。正是基于此，Flink可以更好地在managed state基础上进行内存优化和故障恢复。

raw state指的是Flink只知道state是一些字节数组，其余一无所知。需要用户自己完成state的序列化以及反序列化。因此，Flink不能基于raw state进行内存优化以及故障恢复。所以在企业实战中，很少使用raw state

Managed Keyed State（必须掌握）☆

managed keyed state 接口提供了对不同数据类型的state的访问，这些state都是和key绑定的。这也就意味着managed keyed state只能应用在KeyedStream上。Flink内置的有以下几种managed keyed state

类型	使用场景	方法
ValueState	该状态用于存储单一状态值	update(T) T value() clear()
ListState	该状态用于存储集合状态值	add(T) addAll(List) Iterable get() update(List) clear()
MapState<UK, UV>	该状态用于存储Map集合状态值	put(UK, UV) putAll(Map<UK, UV>) get(UK) entries() keys() values() clear()
ReducingState	该状态用于存储单一状态值。该状态会通过调用用户提供的ReduceFunction，将添加的元素和历史状态自动做运算	add(T) T get() clear()
AggregatingState<IN, OUT>	该状态用于存储单一状态值。该状态会通过调用用户提供的AggregateFunction，将添加的元素和历史状态自动做运算。该状态和ReducingState不同点在于，输入数据类型和输出数据类型可以不同	add(IN) OUT get() clear()
~~FoldingState<T, ACC>~~	该状态用于存储单一状态值。该状态会通过调用用户提供的FoldFunction，将添加的元素和历史状态自动做运算。该状态和ReducingState不同点在于，输入数据类型和中间结果数据类型可以不同	add(T) T get() clear()

> It is important to keep in mind that these state objects are only used for interfacing with state. The state is not necessarily stored inside but might reside on disk or somewhere else. The second thing to keep in mind is that the value you get from the state depends on the key of the input element. So the value you get in one invocation of your user function can differ from the value in another invocation if the keys involved are different.
>
> To get a state handle, you have to create a `StateDescriptor`. This holds the name of the state (as we will see later, you can create several states, and they have to have unique names so that you can reference them), the type of the values that the state holds, and possibly a user-specified function, such as a `ReduceFunction`. Depending on what type of state you want to retrieve, you create either a `ValueStateDescriptor`, a `ListStateDescriptor`, a `ReducingStateDescriptor`, a `FoldingStateDescriptor` or a `MapStateDescriptor`.

> State is accessed using the `RuntimeContext`, so it is only possible in *rich functions*. Please see here for information about that, but we will also see an example shortly. The `RuntimeContext` that is available in a `RichFunction` has these methods for accessing state:
>
> - `ValueState<T> getState(ValueStateDescriptor<T>)`
> - `ReducingState<T> getReducingState(ReducingStateDescriptor<T>)`
> - `ListState<T> getListState(ListStateDescriptor<T>)`
> - `AggregatingState<IN, OUT> getAggregatingState(AggregatingStateDescriptor<IN, ACC, OUT>)`
> - `FoldingState<T, ACC> getFoldingState(FoldingStateDescriptor<T, ACC>)`
> - `MapState<UK, UV> getMapState(MapStateDescriptor<UK, UV>)`

代码实现整体思路☆☆☆

> 1. 写一个类，继承RichMapFunction类
> 2. 重写RichMapFunction里面的open方法
>
> 在open方法中，通过RuntimeContext对象的getXxxState(XxxStateDescriptor)方法获取到XxxState对象
>
> 3. 实现RichMapFunction里面的map方法
>
> 在map方法中，通过XxxState对象根据业务需要实现具体功能
>
> 4. 在代码中的KeyedStream上使用自定义的MapFunction

ValueState

实现wordcount

```
package com.baizhi.flink.state

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

/**
 * 通过wordcount功能，看ValueState的应用
 * 在这个代码中，会详细的聊一下state对象的创建思路
 */
object ValueStateJob {
  def main(args: Array[String]): Unit = {

    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment


    val dataStream: DataStream[String] = environment.socketTextStream("hadoop10", 9999)

    val keyedStream: KeyedStream[(String, Int), Tuple] = dataStream
      .flatMap(_.split("\\s+"))
      .map((_, 1))
      .keyBy(0)
```

```
    val result: DataStream[String] = keyedStream.map(new MyMapFunction)

    result.print()



    environment.execute("ValueStateJob")
  }

}
//两个类型参数，分别表示的是输入类型和输出类型
//输入类型：就是使用这个函数的keyedStream中的数据类型
//输出类型：是根据业务需要自己设置的类型
class MyMapFunction extends RichMapFunction[(String, Int),String]{

  //valueState中存储的是单词的个数
  var valueState:ValueState[Int]=_

  //open方法，用来做初始化的方法：只执行一次
  //在这个方法里面创建需要的状态对象
  override def open(parameters: Configuration): Unit = {

    //要创建状态对象，只需要通过RuntimeContext对象，提供的方法就可以把对象创建出来
    val runtimeContext: RuntimeContext = getRuntimeContext//通过RichMapFunction里面提供的方法
getRuntimeContext可以获取到一个RuntimeContext对象

    //valueStateDescriptor:就是valueState的一个描述者，就是在这个里面声明ValueState中存储的数据的类型
    //两个参数分别表示：唯一标记以及状态中需要存储的数据的类型信息
    var valueStateDescriptor:ValueStateDescriptor[Int]=new ValueStateDescriptor[Int]
("valueState",createTypeInformation[Int])
    valueState=runtimeContext.getState(valueStateDescriptor)//通过runtimeContext提供的getState方
法可以获取一个ValueState对象

  }

  //value:就是输入（流）进来的数据;每流入进来一个元素都会执行一次这个方法
  override def map(value: (String, Int)): String = {
    //在这个方法中完成word count的计算
    //思路：首先从状态中把word对应的count获取到，然后加1,加完之后，再把最新的结果存入到状态中

    //1.通过valueState的value方法，获取到状态中存储的数据
    val oldCount: Int = valueState.value()

    //让原来的数据加1
    val newCount: Int = oldCount + value._2//也可以这样写：oldCount+1

    //2.通过valueState的update方法，把新计算的结果存入到状态中
    valueState.update(newCount)

    value._1+"==的数量是==>"+newCount
  }
}
```

ListState

实现用户浏览商品类别统计

```
package com.baizhi.flink.state

import java.lang

import org.apache.flink.api.common.functions.RichMapFunction
import org.apache.flink.api.common.state.{ListState, ListStateDescriptor}
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._

import scala.collection.JavaConverters._

/**
 * 通过用户访问的类别
 * 业务系统发送过来的日志信息是这样的格式：用户编号  用户名  访问的类别名
 *
 * 通过状态完成统计处理
 * 应该根据用户做统计（keyBy(用户)）；一个用户有可能会访问很多类别：应该使用ListState存储用户访问过的类别
 */
object ListStateJob {
  def main(args: Array[String]): Unit = {


    val environment: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment

    //模拟采集业务系统的日志信息；接下来测试的时候，就应该按照这种格式输入数据：用户编号 用户名 访问的类别
    val dataStream: DataStream[String] = environment.socketTextStream("hadoop10", 9999)

    val keyedStream: KeyedStream[(String, String, String), Tuple] = dataStream
      .map(_.split("\\s+"))
      .map(array => (array(0), array(1), array(2)))
      .keyBy(0)

    val result: DataStream[(String, String)] = keyedStream.map(new MyListStateMapFunction)

    result.print()



    environment.execute("ListStateJob")

  }
```

```
}
class MyListStateMapFunction extends RichMapFunction[(String, String, String),(String,String)]{

  var listState:ListState[String]=_

  override def open(parameters: Configuration): Unit = {

    listState=getRuntimeContext.getListState(new ListStateDescriptor[String]
("lsd",createTypeInformation[String]))


  }

  override def map(value: (String, String, String)): (String, String) = {

    /*//根据业务需要，从状态中获取数据，然后处理数据，然后把数据在保存到状态中

    listState.add(value._3)//add方法就是往状态中添加一个数据


    //构建返回值
    //get方法，获取到状态中存储的数据
    val iter: lang.Iterable[String] = listState.get()


    val scalaIterable: Iterable[String] = iter.asScala//把java的Iterable转换成scala的Iterable

    val str: String = scalaIterable.mkString(",")//通过mkString方法，把iterable对象中的元素都通过逗
号连接起来*/

    //考虑到去重：存储的数据就是已经去重的数据
    //1.从状态中数据获取到，把新进来的数据添加上，然后去重；然后再存入状态中
    val oldIterable: lang.Iterable[String] = listState.get()
    val scalaList: List[String] = oldIterable.asScala.toList
//    println(scalaList)
    val list: List[String] = scalaList :+ value._3//追加:
//    println(scalaList+"=========================")
    val distinctList: List[String] = list.distinct//去重

    listState.update(distinctList.asJava)//更新状态中的数据;upate方法需要一个util.list;所以应该通过
asJava转换一下

    (value._1+":"+value._2,distinctList.mkString(" | "))
  }
}
```

MapState

统计用户浏览商品类别以及该类别的次数

```
var count = 1;
if(mapState.contains(value._2)){
count=mapState.get(value._2)+1
}

//把新的数据存储到mapState中
mapState.put(value._2,count)

//处理返回值
//1.从mapState中获取到现有数据
val nowData: List[String] = mapState.entries().asScala.map(entry=>entry.getKey+"-
>"+entry.getValue).toList

//2.把nowData转换成字符串，流入下游
(value._1,nowData.mkString(" | "))
```

```
package day2

import org.apache.flink.api.common.functions.RichMapFunction
import org.apache.flink.api.common.state.{ListState, ListStateDescriptor, MapState,
MapStateDescriptor}
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._

import scala.collection.JavaConverters._

/**
 * 通过MapState记录用户浏览的类别以及该类别对应的浏览次数
 */
object MapStateJob {
  def main(args: Array[String]): Unit = {
    /**
     * 1.2.3.4.5
     */
    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    //数据===》用户编号 用户名 所访问的类别
    val dataStream: DataStream[String] = environment.socketTextStream("hadoop10", 9999)

    //要处理，就应该根据用户分组===》根据用户做keyby
    val keyedStream: KeyedStream[(String, String), Tuple] = dataStream.map(_.split("\\s+"))
      .map(words => (words(0) + ":" + words(1), words(2)))
      .keyBy(0)
    val result: DataStream[String] = keyedStream.map(new MyMapMapFunction)

    result.print()

    environment.execute("MapStateJob")

  }
```

```
}
class MyMapMapFunction extends RichMapFunction[(String,String),String]{

  //通过MapState把用户访问的类别存储起来
  //mapState中的key是类别，value是该类别对应的访问次数
  var mapState:MapState[String,Int]=_


  override def open(parameters: Configuration): Unit = {

    mapState=getRuntimeContext.getMapState(new MapStateDescriptor[String,Int]
("MapStateDescriptor",createTypeInformation[String],createTypeInformation[Int]))
  }

  override def map(value: (String, String)): String = {
    var category:String = value._2
    //如果类别已经访问过，访问次数就在原有基础上加1；如果没有访问过，就标记为1
    var count:Int=0
    if(mapState.contains(category)){
      count=mapState.get(category)
    }

    //把类别以及对应的访问次数放入到状态中
    mapState.put(category,count+1)


    //构建返回值
    val list: List[String] = mapState.entries().asScala.map(entry => entry.getKey + ":" +
entry.getValue).toList

    val str: String = list.mkString(" | ")

    value._1+"--->"+str
  }
}
```

ReducingState

(存储单一值，可以自动运算，要求输入类型和输出类型是一致的)

实现wordCount自动统计

```
override def open(parameters: Configuration): Unit = {

  val context: RuntimeContext = getRuntimeContext
  val name:String="ReducingStateDescriptor"
  val typeInfo:TypeInformation[Int]=createTypeInformation[Int]
  val reduceFunction: ReduceFunction[Int] = new ReduceFunction[Int] {
    override def reduce(value1: Int, value2: Int): Int = {
//        print(value1+"****"+value2)
      value1+value2
    }
  }
  var reducingStateDescriptor:ReducingStateDescriptor[Int]=new ReducingStateDescriptor[Int](name,reduceFunction,typeInfo)

  reducingState=context.getReducingState(reducingStateDescriptor)
}
```

通过reduceFunction完成自动计算

```
package day2

import org.apache.flink.api.common.functions.{ReduceFunction, RichMapFunction, RuntimeContext}
import org.apache.flink.api.common.state.{ReducingState, ReducingStateDescriptor, ValueState,
ValueStateDescriptor}
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._
/**
 * 通过ReducingState实现wordcount自动统计
 */
object ReducingStateJob {
  def main(args: Array[String]): Unit = {
    /**
     * 1.执行环境
     * 2.数据源：socket
     * 3.数据处理：
     *    3.1 flatmap
     *    3.2 map--->(word,1)
     *    3.3 keyby   ===>dataStream转换成了keyedStream
     *    3.4 map(new MyMapFunction)
     * 4.sink:print
     * 5.executeJob
     */
    /**
     * class MyMapFunction extends RichMapFunction
     * 通过valueState完成数据的统计处理
     * 1.在open方法中创建valueState对象
     *    a.需要RuntimeContext对象
     *
     *    b.RuntimeContext对象中提供的有方法，可以获取到ValueState
     * 2.在map方法中使用valueState对象
     */

    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    val dataStream: DataStream[String] = environment.socketTextStream("hadoop10", 9999)
```

```
    val keyedStream: KeyedStream[(String, Int), Tuple] = dataStream.flatMap(_.split("\\s+"))
      .map((_, 1))
      .keyBy(0)

    val result: DataStream[String] = keyedStream.map(new MyReducingMapFunction)


    result.print()

    environment.execute("ReducingStateJob")

  }

}

/**
 * In: 输入数据的类型;根据使用这个函数的数据流（keyedStream）类型决定
 * Out: 输出数据的类型；map方法的返回值类型。根据业务需要决定
 */
/*class MyMapFunction extends RichMapFunction[IN,Out]*/
class MyReducingMapFunction extends RichMapFunction[(String,Int),String]{

  //通过ReducingState完成wordcount 的自动统计
  var reducingState:ReducingState[Int]=_



  override def open(parameters: Configuration): Unit = {

    val context: RuntimeContext = getRuntimeContext
    val name:String="ReducingStateDescriptor"
    val typeInfo:TypeInformation[Int]=createTypeInformation[Int]
    val reduceFunction: ReduceFunction[Int] = new ReduceFunction[Int] {
      override def reduce(value1: Int, value2: Int): Int = {
//        print(value1+"****"+value2)
        value1+value2
      }
    }
    var reducingStateDescriptor:ReducingStateDescriptor[Int]=new ReducingStateDescriptor[Int]
(name,reduceFunction,typeInfo)

    reducingState=context.getReducingState(reducingStateDescriptor)
  }

  override def map(value: (String, Int)): String = {

    reducingState.add(value._2)//把需要计算的数据添加到reducingState里面

    value._1+":"+reducingState.get()
  }
}
```

AggeragetingState

(存储单一值，可以自动运算，输入类型和输出类型可以不一致的；还可以在运算过程中有中间类型)

实现用户订单平均金额

```
package day2

import org.apache.flink.api.common.functions.{AggregateFunction, RichMapFunction}
import org.apache.flink.api.common.state.{AggregatingState, AggregatingStateDescriptor}
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._

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

    //要求输入的数据:   用户编号  用户名  订单金额
    val dataStream: DataStream[String] = environment.socketTextStream("hadoop10", 9999)

    val keyedStream: KeyedStream[(String, Double), Tuple] = dataStream.map(_.split("\\s+"))
      .map(words => (words(0) + ":" + words(1), words(2).toDouble))
      .keyBy(0)


    val result: DataStream[String] = keyedStream.map(new MyAggregateMapFunction)

    result.print()

    environment.execute("AggregatingStateJob")
  }

}

//通过aggregatingState完成订单的平均金额的计算
class MyAggregateMapFunction extends RichMapFunction[(String,Double),String]{

  //第一个Double表示的是订单金额；第二个Double表示的是用户的订单平均金额
  var aggregatingState:AggregatingState[Double,Double]=_


  override def open(parameters: Configuration): Unit = {

    //第一个Double:输入类型，就是订单金额
    //第二个类型（Double,Int）:中间类型，计算过程中的类型，表示（订单总金额,订单个数）
    //第三个类型Double:输出类型，就是订单平均金额
    var name:String="aggregatingStateDescriptor"

    var aggFunction:AggregateFunction[Double,(Double,Int),Double]=new AggregateFunction[Double,
```

```
(Double,Int),Double] {
      override def createAccumulator(): (Double, Int) = (0,0)//初始值

      /**
       * 中间计算过程
       * @param value 输入数据，订单金额
       * @param accumulator 中间计算结果（订单总金额,订单个数）
       * @return
       */
      override def add(value: Double, accumulator: (Double, Int)): (Double, Int) =
(accumulator._1+value,accumulator._2+1)

      //计算结果
      override def getResult(accumulator: (Double, Int)): Double = accumulator._1/accumulator._2

      override def merge(a: (Double, Int), b: (Double, Int)): (Double, Int) =
(a._1+b._1,a._2+b._2)
    }
    var accType:TypeInformation[(Double,Int)]=createTypeInformation[(Double,Int)]
    var aggregatingStateDescriptor:AggregatingStateDescriptor[Double,(Double,Int),Double]= new
AggregatingStateDescriptor[Double,(Double,Int),Double](name,aggFunction,accType)
    aggregatingState=getRuntimeContext.getAggregatingState[Double,(Double,Int),Double]
(aggregatingStateDescriptor)
  }

  override def map(value: (String, Double)): String = {

    aggregatingState.add(value._2)//把这一次订单的金额放进去

    val avg: Double = aggregatingState.get()//获取到状态中计算完成之后的订单平均金额
    value._1+"的订单平均金额："+avg
  }
}
```

State Time-To-Live(TTL)

基本使用（必须掌握）

在Flink中，支持对所有的keyed state设置存活时间。该特性默认是关闭的，一旦开启并且状态值已经过期，Flink将会尽最大努力清楚所存储的状态值。

TTL支持单一值失效特性，也就意味着ListState中的每一个元素和MapState中的每一个entry都会有单独的失效时间。

要使用stateTTL,首先需要构建一个StateTtlConfig 配置对象。然后通过调用StateDescriptor对象中的enableTimeToLive方法并且将配置对象传递过去来开启TTL机制

```
import org.apache.flink.api.common.state.StateTtlConfig
import org.apache.flink.api.common.state.ValueStateDescriptor
import org.apache.flink.api.common.time.Time

val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))//①
    .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)//②
    .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)//③
    .build

val stateDescriptor = new ValueStateDescriptor[String]("text state", classOf[String])
stateDescriptor.enableTimeToLive(ttlConfig)
```

对于以上代码，需要说明的有以下几点

1. ①处是Time-To-Live的值，是必须要设置的。可以根据需要设置对应的时间值
2. ②处是TTL的更新机制，默认是OnCreateAndWrite

   可用值有两个

   ```
   StateTtlConfig.UpdateType.OnCreateAndWrite - 创建和写入更新时间
   StateTtlConfig.UpdateType.OnReadAndWrite - 读取和写入更新时间
   ```

3. ③处是state的可见性配置，过期的但是还没有被清理掉的数据是否可以读取到，默认值NeverReturnExpired

   可用值有两个

   ```
   StateTtlConfig.StateVisibility.NeverReturnExpired - 过期数据永不返回
   StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp - 过期数据如果还没有被清理就返回
   ```

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

class ValueStateMapFunctionTTL  extends RichMapFunction[(String,Int),(String,Int)]{

  var valueState:ValueState[Int]=_;

  override def open(parameters: Configuration): Unit = {

    //1.创建valueStateDescriptor对象
    var valueStateDescriptor = new ValueStateDescriptor[Int]
("myValueStateDescriptor",createTypeInformation[Int]);

    //2.获取到RuntimeContext
    var runtimeContext = getRuntimeContext;


    //ttl配置对象
```

```
    val ttlConfig = StateTtlConfig
      .newBuilder(Time.seconds(5))
      .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
      .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
      .build

    //开启ttl
    valueStateDescriptor.enableTimeToLive(ttlConfig)

    //3.通过RuntimeContext对象的getState方法获取到ValueState对象
    valueState = runtimeContext.getState(valueStateDescriptor)
  }

  override def map(value: (String,Int)): (String,Int) = {
    //1.通过valueState对象的value方法获取到历史数据
    var historyData = valueState.value()

    //2.通过valueState对象的update方法更新数据
    valueState.update(historyData+value._2)

    //3.返回值
    (value._1,valueState.value())
  }
}
```

> Note
>
> 1. 一旦开启了TTL机制，系统为每个存储的状态数据额外开辟8个字节的空间，用来存储state的时间戳
> 2. TTL目前仅支持processing time
> 3. 如果程序一开始没有启用TTL，重启服务开启了TTL，则服务在故障恢复时StateMigrationException

Cleanup of Expired State（过期状态的清理机制-垃圾回收）

> 1.9以及之前版本：
>
> 这就意味着，在默认情况下，如果过期数据没有被读取，就不会被删除。很有可能导致过期数据越来越大而占用太多内存。可以通过调用StateTtlConfig.Builder的.cleanupInBackground方法开启后台清理

> 1.10版本：
>
> 如果配置的state backend，则在后台定期进行垃圾回收。可以通过以下API禁用后台清理
>
> ```
> import org.apache.flink.api.common.state.StateTtlConfig
> val ttlConfig = StateTtlConfig
> .newBuilder(Time.seconds(1))
> .disableCleanupInBackground
> .build
> ```

Cleanup in full snapshot（全本快照）

可以通过配置Cleanup in full snapshot机制，在系统恢复或者启动的时候， 加载状态数据，此时会将过期的数据删除

```
import org.apache.flink.api.common.state.StateTtlConfig
import org.apache.flink.api.common.time.Time

val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupFullSnapshot
    .build
```

> 也就是只有Flink服务重启的时候才会清理过期数据

Incremental cleanup（增量处理）

增量清理策略，在用户每一次读取或者写入状态数据的时候，该清理策略就会运行一次。系统的state backend会保存所有状态的一个全局迭代器。每一次访问状态或者/和记录处理时，该迭代器就会增量迭代一个批次的数据，检查是否存在过期的数据，如果存在就删除

```
import org.apache.flink.api.common.state.StateTtlConfig
val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupIncrementally(10, true)
    .build
```

该策略需要两个参数

> 1. cleanupSize - max number of keys pulled from queue for clean up upon state touch for any key
>
> 一次检查的key的数量
>
> 2. runCleanupForEveryRecord - run incremental cleanup per each processed record
>
> 是否每一次record processing都会触发incremental cleanup。如果为false，就表示只有访问状态时才触发incremental cleanup；true则表示访问状态以及记录处理都会触发incremental cleanup

> Note
>
> - 如果没有状态访问或者记录处理，过期的数据就不会删除，会被持久化
> - incremental cleanup需要花费时间，从而增加了record processing的延迟
> - 目前，incremental cleanup仅支持 Heap state backend。如果是RocksDB，该机制不起作用

Cleanup during RocksDB compaction(压实机制)

如果使用的是RocksDB作为state backend，Flink将会通过Compaction filter实现后台清理。Compaction（压实机制） filter会检查状态的时间戳以获取剩余存活时间并把过期数据清除掉

```
import org.apache.flink.api.common.state.StateTtlConfig

val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupInRocksdbCompactFilter(1000)
    .build
```

参数queryTimeAfterNumEntries表示处理了多少个key之后去获取时间。以对比存储的时间戳，将过期的数据删除掉

频繁的更新时间戳会提高清理速度。但是由于采用JNI调用本地代码，会降低压实性能。默认情况，每处理1000个key，RocksDB backend会查询一次当前时间戳从而清理过期数据

> 扩展
>
> RocksDB是一个基于内存+磁盘的嵌入式的轻量级的NoSQL产品，底层维护一张HashTable。所有的记录都是顺序追加到磁盘，最新状态存储在内存中。RocksDB不支持更新磁盘。但是RocksDB底层有一套Compaction机制（压实机制），用于合并磁盘文件，以防止文件过大

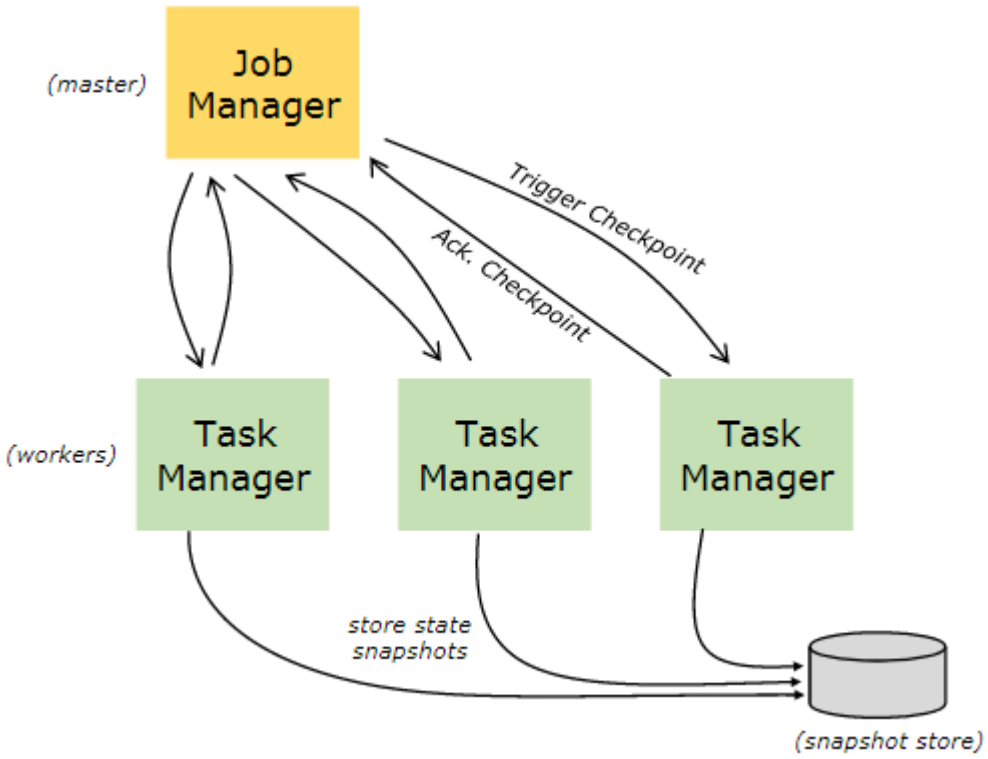
> Note
>
> 在Flink1.10之前，RocksDB的CompactionFilter特性是默认关闭的，需要使用，应该在flin-conf.yaml配置文件中开启
>
> ```
> state.backend.rocksdb.ttl.compaction.filter.enabled: true
> ```
>
> ```
> This feature is disabled by default. It has to be firstly activated for the RocksDB backend by setting Flink configuration option state.backend.rocksdb.ttl.compaction.filter.enabled or by calling RocksDBStateBackend::enableTtlCompactionFilter if a custom RocksDB state backend is created for a job
> ```

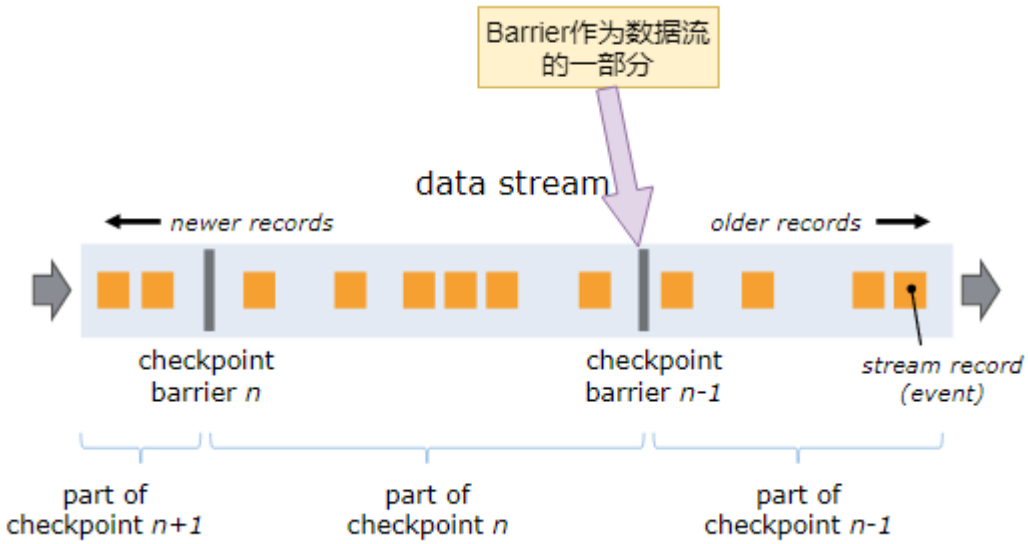
Checkpoint&Savepoint（必须掌握）

Flink是一个有状态的流计算引擎，因此状态的管理和容错是非常重要的。为了程序的健壮性，Flink提出了Checkpoint机制，该机制用于持久化计算节点的状态数据，从而实现Flink故障恢复。

Checkpoint机制指的是Flink会定期将状态数据持久化到远程文件系统，比如HDFS（这取决于state backend）。

JobManager负责checkpoint的发起以及协调。JobManager节点会定期向TaskManager节点发送Barrier（实际上是JobManager创建的CheckpointCoordinator），TaskManager接收到Barrier信号，会把Barrier信号作为数据流的一部分传递给所有算子。每一个算子接收到Barrier信号后会预先提交自己的状态信息并且给JobManger应答，同时会将Barrier信号传递给下游算子。JobManager接收到所有算子的应答后才认定此次Checkpoint是成功的，并且会自动删除上一次Checkpoint数据。否则，如果在规定的时间内没有收到所有算子的应答，则认为本次Checkpoint快照制作失败 。





Savepoint是手动触发的checkpoint，它获取程序的快照并将其写入state backend。Checkpoint依赖于常规的检查点机制：在执行过程中个，程序会定期在TaskManager上快照并且生成checkpoint。为了恢复，只需要最后生成的checkpoint。旧的checkpoint可以在新的checkpoint完成后安全地丢弃。

Savepoint与上述的定期checkpoint类似，只是他们由用户触发，并且在新的checkpoint完成时不会自动过期。Savepoint可以通过命令行创建，也可以通过REST API在取消Job时创建。

```
savepoint:
savepoint就是手动版的checkpoint
可以随时通过命令行完成持久化
跟checkpoint不一样的地方就一点；一次savepoint不会把上一次的savepoint替换掉
```

默认情况下，Flink的Checkpoint机制是禁用的，如果需要开启，可以通过以下API完成

```
StreamExecutionEnvironment.enableCheckpointing(n)
```

n表示每间隔多少毫秒执行一次checkpoint

可以通过以下参数更精准地控制Checkpoint

```
 //5000:每间隔5000毫秒执行一次checkpoint。

    // CheckpointingMode.EXACTLY_ONCE:checkpointing模式是精准一次
    /**
      * This mode means that the system will
      * checkpoint the operator and user function state in such a way that, upon recovery,
      * every record will be reflected exactly once in the operator state.
      */

    //checkpointing模式还有一个值是CheckpointingMode.AT_LEAST_ONCE
    /**
      * Sets the checkpointing mode to "at least once". This mode means that the system will
      * checkpoint the operator and user function state in a simpler way. Upon failure and
recovery,
      * some records may be reflected multiple times in the operator state.
      */

    environment.enableCheckpointing(5000,CheckpointingMode.EXACTLY_ONCE)

    //Sets the maximum time that a checkpoint may take before being discarded.
    // in milliseconds
    environment.getCheckpointConfig.setCheckpointTimeout(4000)

    //两次检查点间隔不得小于2秒，优先级高于checkpoint interval
    environment.getCheckpointConfig.setMinPauseBetweenCheckpoints(2000)

    //允许checkpoint失败的参数，默认值是0。取代了setFailOnCheckpointingErrors(boolean)
    environment.getCheckpointConfig.setTolerableCheckpointFailureNumber(2)

    //当任务取消时，检查点数据该如何处理
    //RETAIN_ON_CANCELLATION:任务取消时，没有加savepoint,检查点数据保留
    //DELETE_ON_CANCELLATION：任务取消时，检查点数据删除（不建议使用）

environment.getCheckpointConfig.enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETA
IN_ON_CANCELLATION)
```

State backend（必须掌握）

State backend指定了状态数据（检查点数据）存储的位置以及如何存储。Flink提供了多种State backend的实现。state backend有两种配置方式

- 每一个job单独配置state backend

```
val env = StreamExecutionEnvironment.getExecutionEnvironment()
env.setStateBackend(...)//只针对这一个job
```

- 在flink-conf.yaml中配置所有job使用的state backend

```
#==============================================================================
# Fault tolerance and checkpointing
#==============================================================================

# The backend that will be used to store operator state checkpoints if
# checkpointing is enabled.
#
# Supported backends are 'jobmanager', 'filesystem', 'rocksdb', or the
# <class-name-of-factory>.
#
 state.backend: filesystem

# Directory for checkpoints filesystem, when using any of the default bundled
# state backends.
#
# state.checkpoints.dir: hdfs://namenode-host:port/flink-checkpoints

 state.checkpoints.dir: hdfs:///flink-checkpoints

# Default target directory for savepoints, optional.
#
# state.savepoints.dir: hdfs://namenode-host:port/flink-checkpoints

 state.savepoints.dir: hdfs:///flink-checkpoints

# Flag to enable/disable incremental checkpoints for backends that
# support incremental checkpoints (like the RocksDB state backend).
```

配置文件配置完成之后，重新启动Flink，检查全局state backend配置是否成功

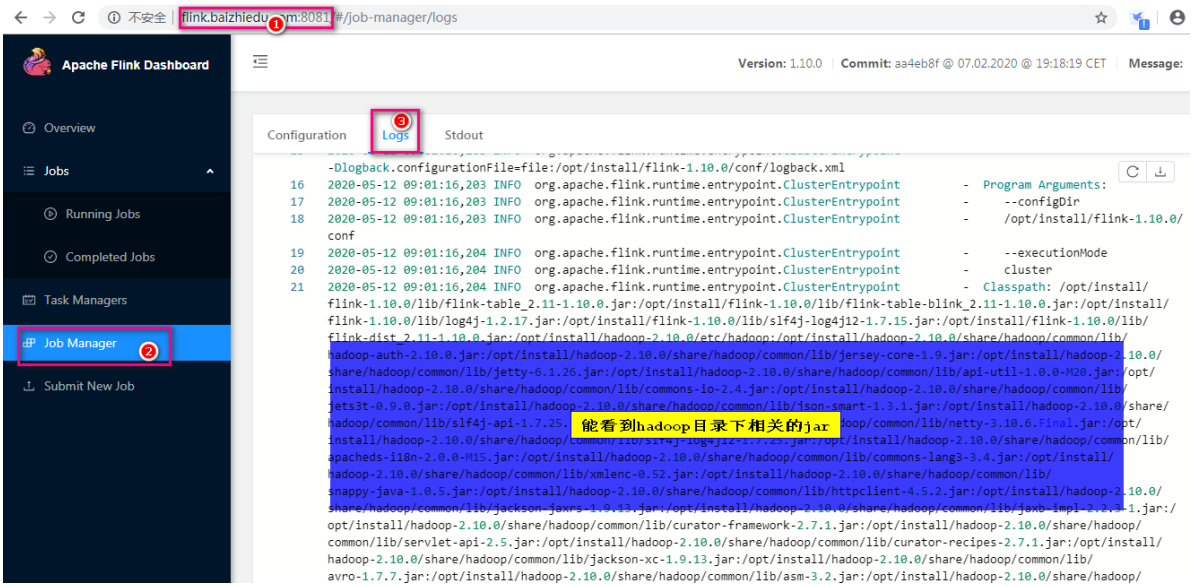
1. 停止flink

```
[root@flink flink-1.10.0]# pwd
/opt/install/flink-1.10.0
[root@flink flink-1.10.0]# bin/stop-cluster.sh
Stopping taskexecutor daemon (pid: 3972) on host flink.
Stopping standalonesession daemon (pid: 3627) on host flink.
```

2. 启动flink

```
[root@flink flink-1.10.0]# pwd
/opt/install/flink-1.10.0
[root@flink flink-1.10.0]# bin/start-cluster.sh
Starting cluster.
Starting standalonesession daemon on host flink.
Starting taskexecutor daemon on host flink.
```

3. web-UI界面查看相关日志信息



Note

Adding Hadoop Classpaths

Flink will use the environment variable `HADOOP_CLASSPATH` to augment the classpath that is used when starting Flink components such as the Client, JobManager, or TaskManager. Most Hadoop distributions and cloud environments will not set this variable by default so if the Hadoop classpath should be picked up by Flink the environment variable must be exported on all machines that are running Flink components.

When running on YARN, this is usually not a problem because the components running inside YARN will be started with the Hadoop classpaths, but it can happen that the Hadoop dependencies must be in the classpath when submitting a job to YARN. For this, it's usually enough to do a

```
export HADOOP_CLASSPATH=`hadoop classpath`
```

in the shell. Note that `hadoop` is the hadoop binary and that `classpath` is an argument that will make it print the configured Hadoop classpath.

Putting the Hadoop configuration in the same class path as the Hadoop libraries makes Flink pick up that configuration.

因为state backend需要将数据同步到HDFS，所以Flink需要和Hadoop集成。需要在环境变量中配置HADOOP_CLASSPATH

vi /etc/profile，然后最下面添加以下内容

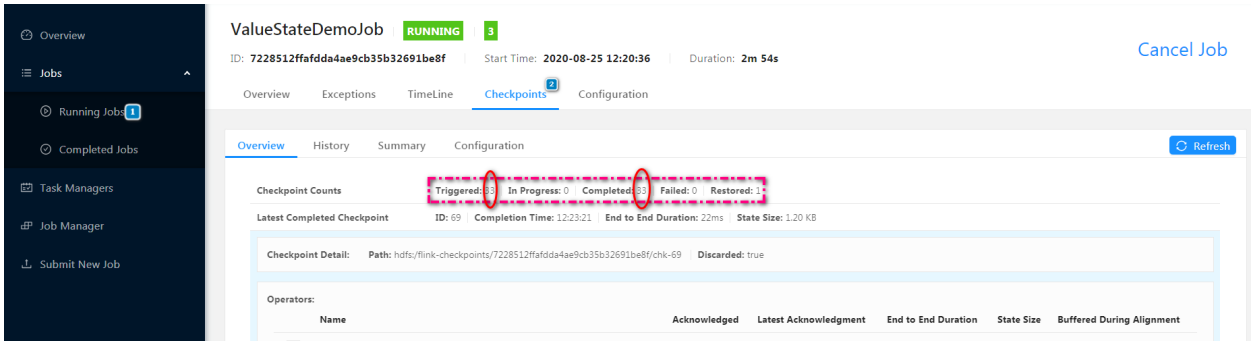
```
export HADOOP_CLASSPATH=`hadoop classpath`
```

source /etc/profile

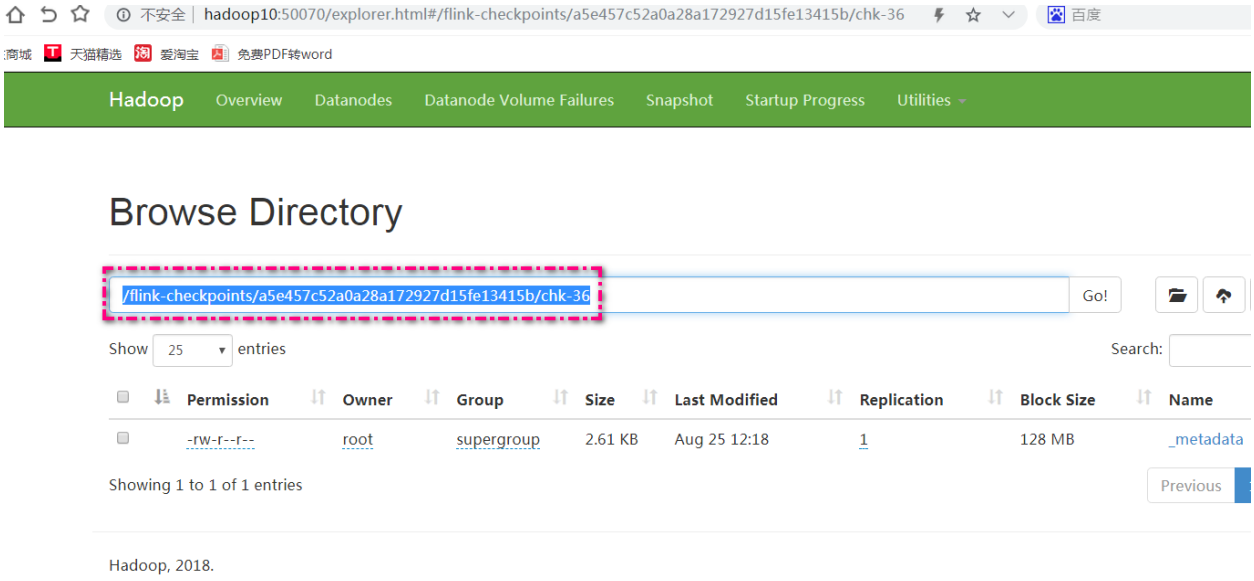
结合着检查点以及状态完成数据的故障恢复整体思路

1. 在flink配置文件中配置state backend
2. 在flink代码中开启checkpoint

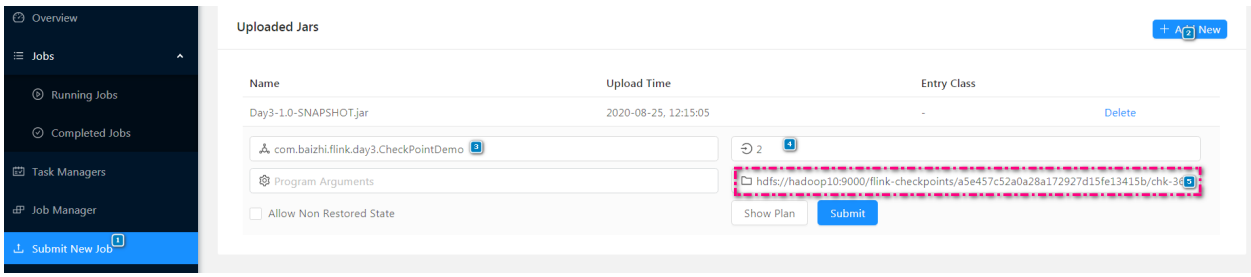
3. 把flink代码打包，通过flink UI界面传输到flink环境中执行
4. 确认checkpoint的配置是否生效



5. 让程序执行计算结果，在taskManager中查看
6. 取消掉job
7. 到hdfs中复制生成的checkpoint路径



8. 重新启动flink程序，在checkpoint的位置，输入恢复数据需要的hdfs路径



Broadcast State Pattern

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[Stri
ng])
    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

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)
  }
}
```

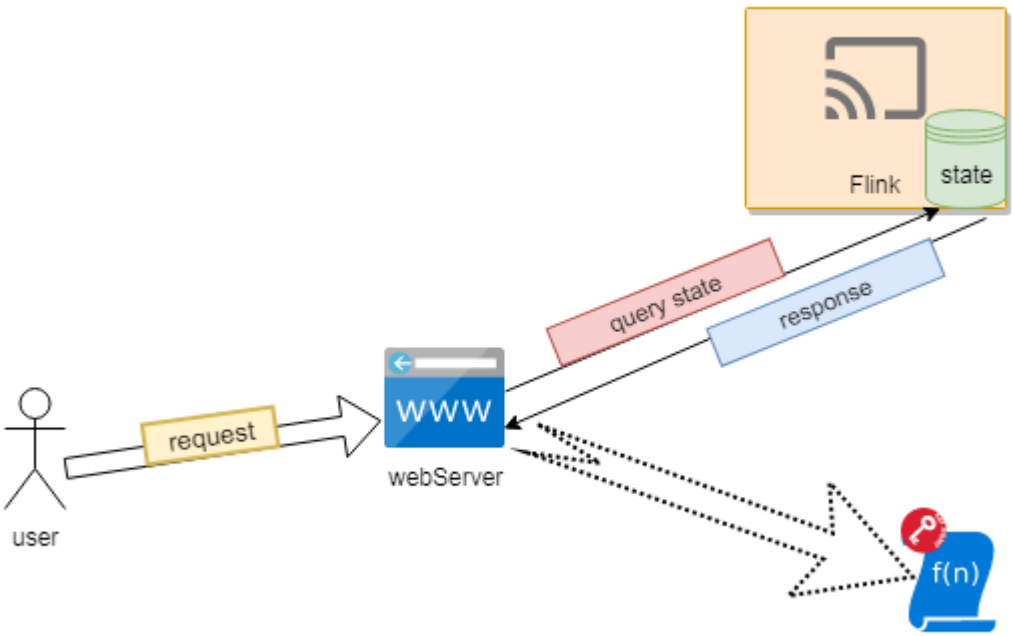
```
import org.apache.flink.api.common.state.MapStateDescriptor
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._

/**
  * 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")
```

```
    }
}
```

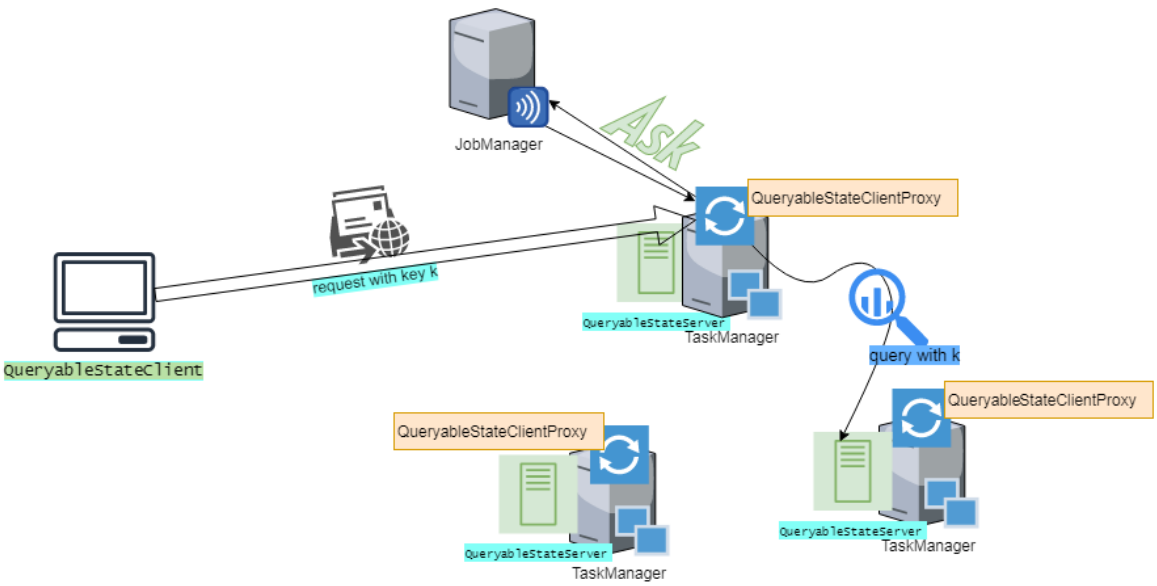
Queryable State☆

> 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

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 @ ...
```

```
167   2020-05-13 21:59:06,446 INFO  org.apache.flink.queryablestate.server.KvStateServerImpl      - Started Queryable State
      Server @ /192.168.77.170:9067.
168   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],r
educingStateDescriptor)

    //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>
```