Flink-day2笔记

复习

1. Flink是一个基于状态的流计算引擎
    - 流计算：dataSource /数据处理 / sink
    - **基于状态的计算**
    - 处理有界数据流以及无界数据流、支持很多资源管理系统（standalone、yarn、k8s、mesos）、支持一系列的存储（hdfs/mysql/redis/hbase）
2. Flink的开发
    - 开发环境：直接通过API的getExecutionEnvironment方法获取到开发环境（支持本地、支持远程、获取到资源）
    - 读取数据源
    - 通过flink算子完成对应的操作处理map/keyBy
    - sink
3. flink程序的部署
    - 本地测试：开发工具中完成测试
    - 命令行：
        - 程序打包：jar
        - 上传到服务器
        - 通过flink脚本执行jar中job
    - webUI完成
    - 跨平台部署：直接在代码中，把jar、服务器地址写好
4. Flink运行架构的理解
    1. 整个flink程序是一个job；根据operator chain对job进行划分，划分成了task；根据并行度把task划分为subtask
    2. jobmanager、taskmanager、client
    3. task slot：taskmanager计算能力的一种表现
        - 一个job不同task的subtask共享task slot
        - 在程序开发中，需要设置并行度（task slot的最小个数是并行度最大值）
    4. checkpoint

        自动的，用来基于状态的持久化机制
    5. savepoint

        手动版checkpoint

Stream(DataStreamAPI)

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

DataSource:数据源

数据源是程序读取数据的来源。用户可以通过`StreamExecutionEnvironment.addSource(sourceFunction)`将数据源添加到程序中。Flink提供了很多的sourceFunction，用户也可以自定义sourceFunction。可以通过实现`SourceFunction`接口实现非并行化，也可以通过实现`ParallelSourceFunction`或者继承`RichParallelSourceFunction`实现并行化。

一系列数据源的应用

File-based（了解）

- ***readTextFile(path)*** - Reads text files, i.e. files that respect the `TextInputFormat` specification, line-by-line and returns them as Strings.

  读取本地文件

  ```
  val dataStream: DataStream[String] = environment.readTextFile( filePath = "d:/a.txt")
  ```

  读取分布式文件系统HDFS

  > 1. 添加Hadoop依赖
  > 2. 读取hdfs中的文件

  ```
  <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-client -->
   <dependency>
     <groupId>org.apache.hadoop</groupId>
     <artifactId>hadoop-client</artifactId>
     <version>2.10.0</version>
   </dependency>

   <!-- 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>
  ```

  ```
  //1.创建执行环境
   val environment = StreamExecutionEnvironment.getExecutionEnvironment


   //2.获取数据源

   val text = environment.readTextFile("hdfs://flink.baizhiedu.com:8020/flink/flink-
  words")

   //3.对获取到的数据进行转换
   val result = text.flatMap(line => line.split("\\s+"))
     .map(word => (word, 1))
  ```

```
    .keyBy(0)
    .sum(1)

  //4.打印结果
  result.print()

  //5.执行job
  environment.execute("myFlinkJob")
```

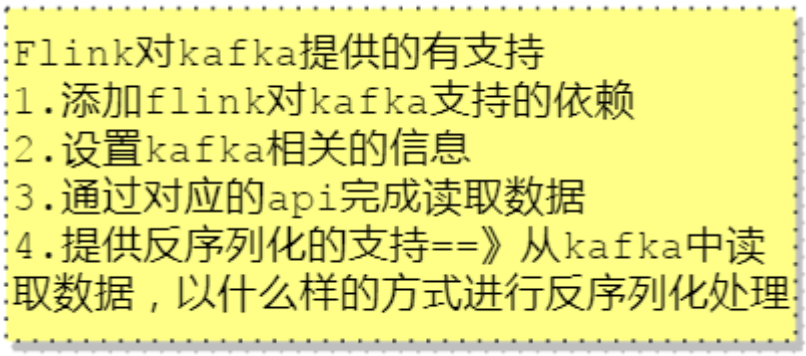
> Note:
>
> - 必须保证hdfs中有代码中对应的文件
>
> `hdfs://flink.baizhiedu.com:8020/flink/flink-words`

Socket-based

- ***socketTextStream*** - Reads from a socket. Elements can be separated by a delimiter.

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

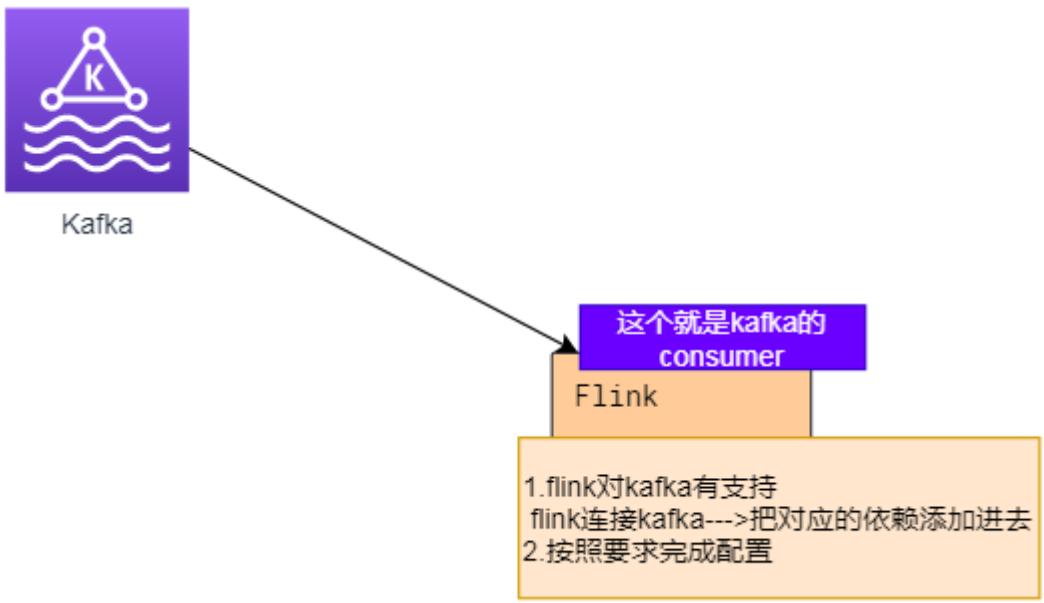
Read from Apache Kafka☆



> *previously on Zookeeper&Kafka*
>
> - Zookeeper （standalone）
>
> `https://zookeeper.apache.org/doc/current/zookeeperStarted.html`
>
> ```
> 1.启动zookeeper
> [root@flink apache-zookeeper-3.5.7-bin]# bin/zkServer.sh start
> ZooKeeper JMX enabled by default
> Using config: /opt/install/apache-zookeeper-3.5.7-bin/bin/../conf/zoo.cfg
> Starting zookeeper ... STARTED
>
> 2.连接zookeeper
> [root@flink apache-zookeeper-3.5.7-bin]# bin/zkCli.sh -server 127.0.0.1:2181
> ```
>
> - Kafka （standalone）
>
> `http://kafka.apache.org/quickstart`

```
1.启动kafka
[root@flink kafka_2.11-2.2.0]# bin/kafka-server-start.sh config/server.properties

2.创建主题
[root@flink kafka_2.11-2.2.0]# bin/kafka-topics.sh --create --bootstrap-server
localhost:9092 --replication-factor 1 --partitions 1 --topic test

3.查看主题
[root@flink kafka_2.11-2.2.0]#  bin/kafka-topics.sh --list --bootstrap-server
localhost:9092
test

4.发送消息
[root@flink kafka_2.11-2.2.0]# bin/kafka-console-producer.sh --broker-list localhost:9092 -
-topic topica
  >

5.消费消息
[root@flink kafka_2.11-2.2.0]# bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic topic01 --from-beginning
```

Flink集成Kafka



`https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/connectors/kafka.html`

1. 引入maven依赖

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.11</artifactId>
  <version>1.10.0</version>
</dependency>
```

2. 各种DataSource的应用

- ***SimpleStringSchema***

  SimpleStringSchema只会反序列化value

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

    val properties = new Properties()
    properties.setProperty("bootstrap.servers", "flink.baizhiedu.com:9092")

    var text = environment
      .addSource(new FlinkKafkaConsumer[String]("topic01", new SimpleStringSchema(),
properties));


    //3.对获取到的数据进行转换
    val result = text.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)
      .sum(1)

    //4.打印结果
    result.print()

    //5.执行job
    environment.execute("myFlinkJob")
//    println(environment.getExecutionPlan)

  }

}
```

- **KafkaDeserializationSchema**

  通过实现这个接口，可以反序列化key、value、partition、offset等

```
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.streaming.connectors.kafka.KafkaDeserializationSchema
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.flink.api.scala._

/**
  * 泛型分别是key/value/partition/offset的类型
  */
class MyKafkaDeserializationSchema extends
KafkaDeserializationSchema[(String,String,Int,Long)]{
  override def isEndOfStream(t: (String, String, Int, Long)): Boolean = false;
```

```
  override def deserialize(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]]):
(String, String, Int, Long) = {

    if(consumerRecord.key()!=null){

      (new String(consumerRecord.key()),new
String(consumerRecord.value()),consumerRecord.partition(),consumerRecord.offset())
    }else{
      (null,new
String(consumerRecord.value()),consumerRecord.partition(),consumerRecord.offset())



    }
  }

  override def getProducedType: TypeInformation[(String, String, Int, Long)] = {
    createTypeInformation[(String, String, Int, Long)];
  }
}
```

```
import java.util.Properties

import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer

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

    val properties = new Properties()
    properties.setProperty("bootstrap.servers", "flink.baizhiedu.com:9092")

    /*var text = environment
      .addSource(new FlinkKafkaConsumer[String]("topic01", new SimpleStringSchema(),
properties));*/
    var text = environment
      .addSource(new FlinkKafkaConsumer[(String,String,Int,Long)]("topic01", new
MyKafkaDeserializationSchema(), properties));


    //3.对获取到的数据进行转换
    val result = text.flatMap(line =>line._2.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)
      .sum(1)

    //4.打印结果
    result.print()


    //5.执行job
```

```
    environment.execute("myFlinkJob")
//    println(environment.getExecutionPlan)

  }

}
```

- ***JSONKeyValueDeserializationSchema***

  这个是flink-kafka提供的类，可以直接使用，在使用的时候要求kafka中topic的key、value都必须是json。也可以在使用的过程中，指定是否读取元数据（topic、partition、offset等）

```
import java.util.Properties

import org.apache.flink.shaded.jackson2.com.fasterxml.jackson.databind.node.ObjectNode
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.flink.streaming.util.serialization.JSONKeyValueDeserializationSchema
import org.apache.flink.api.scala._

object JSONKeyValueDeserializationSchema {


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

    //设置并行度，通过打印执行计划查看并行度是否起作用
    var properties: Properties = new Properties()
    properties.setProperty("bootstrap.servers", "flink.baizhiedu.com:9092")

    //2.获取数据源
//    val result = environment.addSource(new FlinkKafkaConsumer[ObjectNode]("topic01",
new JSONKeyValueDeserializationSchema(true), properties));


    val text = environment
        .addSource(new FlinkKafkaConsumer[ObjectNode]("topic05",new
JSONKeyValueDeserializationSchema(false),properties));

    //先查看一下内容整体
    //text.map(t=>t.toString).print()

    text.map(t=>
(t.get("value").get("id").asInt(),t.get("value").get("name").asText())).print()


    //5.执行job
    environment.execute("myFlinkJob")


  }
```

```
}
```

```
[root@flink kafka_2.11-2.2.0]# bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic topic05
>{"id":101,"name":"xiaohei"}
```

> Note
>
> 1. 注意导包`import org.apache.flink.api.scala._`
> 2. Kafka中的数据需要是JSON格式

算子

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

算子	描述
map	映射
flatmap	映射（压平）
filter	过滤操作
keyby	分组操作；执行完成之后得到的是keyedStream；keyby算子可以把dataStream转换成keyedStream

- map以及flatmap

```
 //输入数据为a b  c
//    val value: DataStream[String] = dataStream.flatMap(_.split("\\s+"))
//    value.print() //打印出来就是a b c三个元素
    val dataStream2: DataStream[Array[String]] = dataStream.map(_.split("\\s+"))//
    dataStream2.print()//打印出来就是一个数组地址
    val value: DataStream[String] = dataStream2.map(e => e(0) + "***" + e(1))//从数组中获取对应位置的元素，然后拼接成字符串
    value.print()//a***b
```

- keyedStream的理解

> 在flink中，数据是有状态的；数据的状态很多时候是和keyedStream结合在一起使用的；keyedState（同一个key对应的是同一块状态区域）

datasink：数据输出

支持多种输出方式：打印、文件(**HDFS**)、redis、**kafka**....

`https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/datastream_api.html#data-sinks`

生产环境，通常使用`flink-connector-filesystem`把结果写入到外部文件系统

1. 添加`flink-connector-filesystem`依赖

```
<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-connector-filesystem -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-filesystem_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

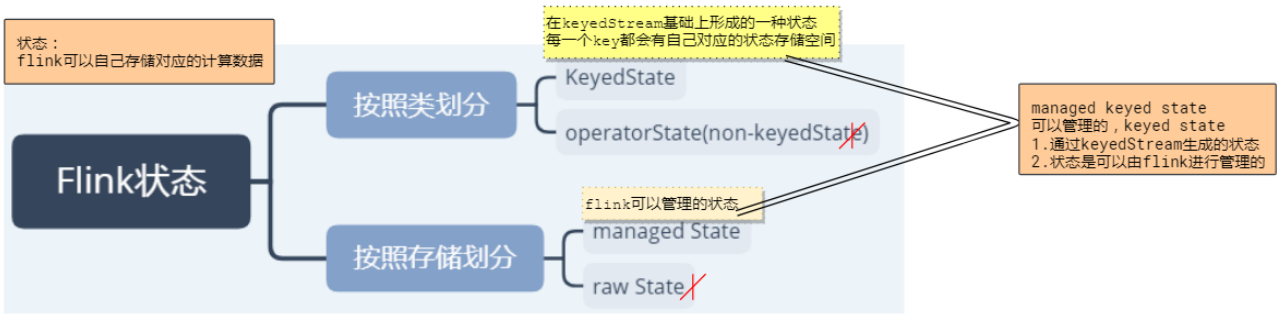
2. 代码实现

```
import org.apache.flink.api.common.serialization.SimpleStringEncoder
import org.apache.flink.core.fs.Path
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink
import
org.apache.flink.streaming.api.functions.sink.filesystem.bucketassigners.DateTimeBucketAssigner
import org.apache.flink.streaming.api.scala._

object FileDataSinkFlinkConnectorFileSystem {


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


    //2.获取数据源
    val text = environment.socketTextStream("flink.baizhiedu.com",9999)

    var streamFileSink = StreamingFileSink.forRowFormat(new
Path("hdfs://flink.baizhiedu.com:8020/flink-result"),
      new SimpleStringEncoder[(String,Int)]())
      //按照指定格式生成写入路径;如果没有这个，系统flink会按照其内置的路径yyyy-MM-dd--HH
      .withBucketAssigner(new DateTimeBucketAssigner[(String, Int)]("yyyy-MM-dd"))
      .build();

    //3.对获取到的数据进行转换
    val result = text.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)
      .sum(1)

    //4.把数据写入到文件系统
```

```
    result.addSink(streamFileSink)

    //5.执行job
    environment.execute("myFlinkJob")

  }

}
```

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。

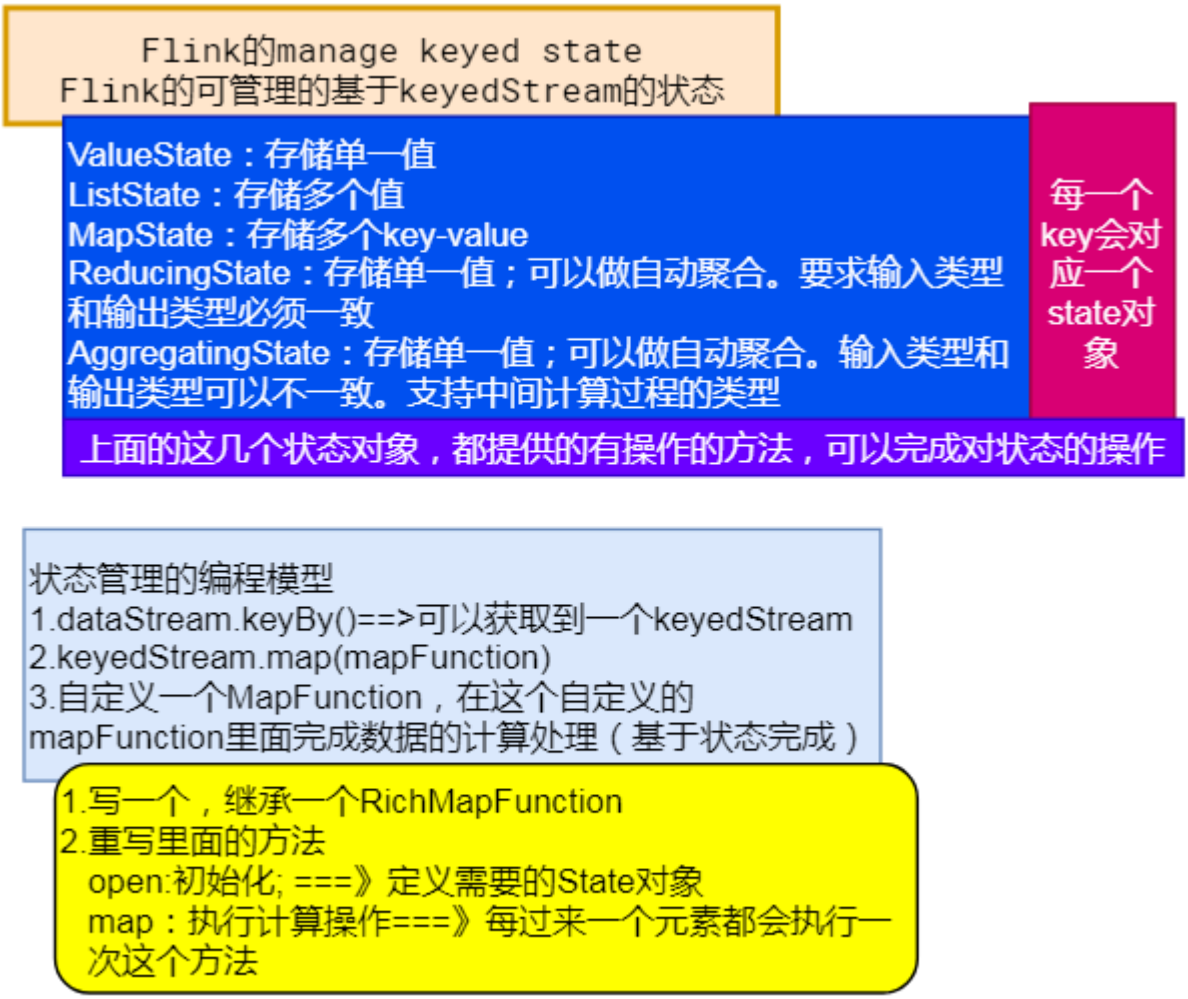
在Flink中有两种基本类型的state，分别是`Keyed State`和`Operator State`。Keyed State只能应用在KeyedStream上的操作。每一个keyed operator都会绑定一个或多个状态值。Operator State又被称为non-keyed state，每一个算子都会有对应的operator state。

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）☆



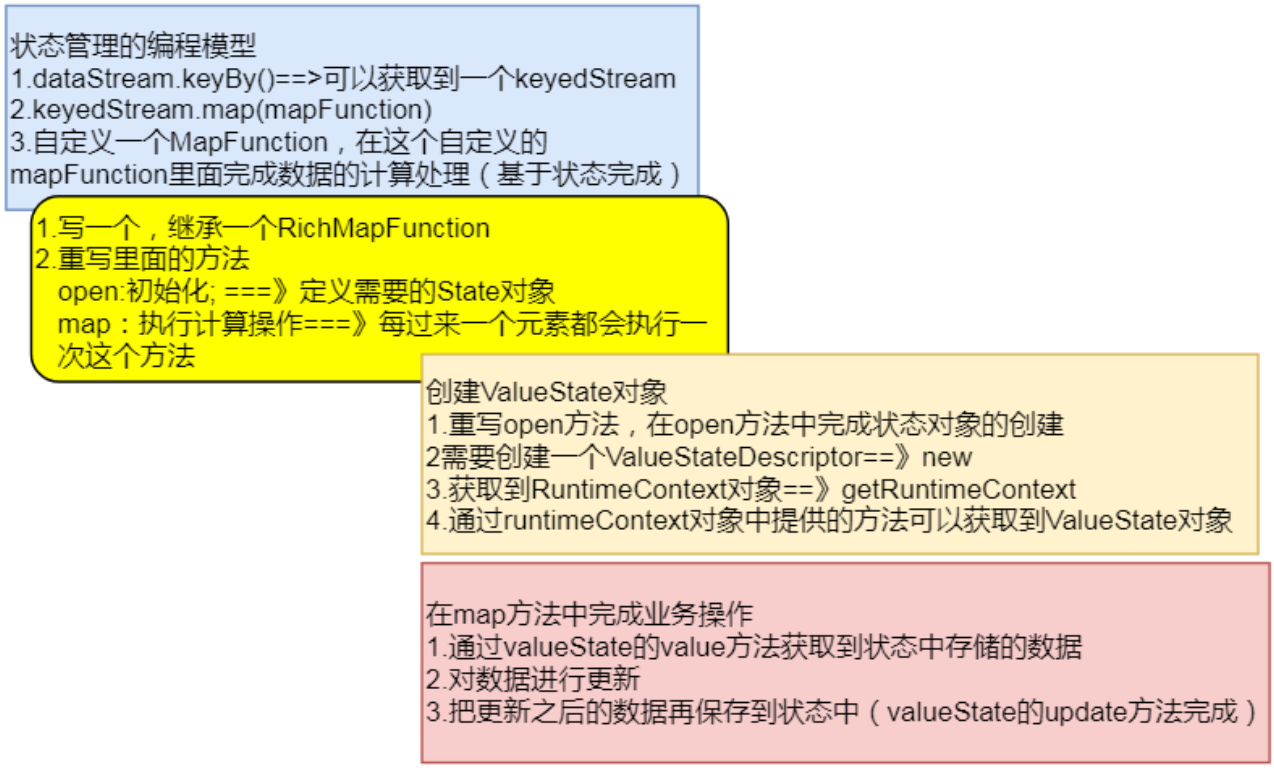
managed keyed state 接口提供了对不同数据类型的state的访问，这些state都是和key绑定的。这也就意味着managed keyed state只能应用在KeyedStream上。Flink内置的有以下几种managed keyed state

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}
```

```scala
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
  }
}
```

总结

1. 必须有keyedStream才可以使用keyed state(ValueState/ListState/MapState/ReducingState/AggregatintState)

   keyedState就是和key绑定在一起的状态（每一个key对应一个状态；不同的key对应的是不同的状态）

2. 要使用state，就需要创建

   通过RuntimeContext对象提供的方法完成state的创建

   - 通过flink提供的方法，获取到runtimContext对象
   - 创建state对象的方法需要stateDescriptor（描述者，用来描述创建出来的state可以存储什么类型的数据）---》通过new关键字创建Descriptor

3. 在map方法中使用state完成数据的存取处理

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._
```

```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(" | "))
```

```scala
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
  }
}
```

作业

1. 上课的代码自己敲
2. 复习
3. 面试题

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
```

```scala
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
```

```scala
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
  }
}
```