Apache Flink进阶之状态流

本课目标

- ✓ 深入理解Flink State原理
- ✓ 掌握常见的DataStream算子（source，transform，Sink）
- ✓ 掌握场景的DataStream的Sink操作
- ✓ 掌握Fink State核心概念

1 Flink数据源

1.1 Flink之数据源

- source简介：source是程序的数据源输入，你可以通过StreamExecutionEnvironment.addSource(sourceFunction)来为你的程序添加一个source。flink提供了大量的已经实现好的source方法，你也可以自定义source：
  - 通过实现sourceFunction接口来自定义无并行度的source
  - 通过实现ParallelSourceFunction 接口 or 继承RichParallelSourceFunction 来自定义有并行度的source
  - 不过大多数情况下，我们使用自带的source即可。
- 获取source的方式：基于文件readTextFile(path)
  - 读取文本文件，文件遵循TextInputFormat 读取规则，逐行读取并返回。可以用于简单的测试
- 基于socket：socketTextStream
  - 从socker中读取数据
- 基于集合：fromCollection(Collection)
  - 通过java 的collection集合创建一个数据流，集合中的所有元素必须是相同类型的。可以用于简单的测试
- 自定义输入：addSource 可以实现读取第三方数据源的数据
  - 系统内置提供了一批connectors，连接器会提供对应的source支持
- 扩展的connectors
  - Apache Kafka (source/sink) **重点**
  - Apache Cassandra (sink)
  - Amazon Kinesis Streams (source/sink)
  - Elasticsearch (sink)
  - Hadoop FileSystem (sink)
  - RabbitMQ (source/sink)
  - Apache NiFi (source/sink)

- Twitter Streaming API (source)

1.2 数据源之collection

```
#案例1，数据源之集合
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        //创建数据源
        ArrayList<String> data = new ArrayList<String>();
        data.add("flink1");
        data.add("flink2");
        data.add("flink3");

        //etl
        DataStreamSource<String> stringDataStreamSource =
env.fromCollection(data);
        SingleOutputStreamOperator<String> mapResult =
stringDataStreamSource.map(new MapFunction<String, String>() {
            @Override
            public String map(String value) {
                return value;
            }
        });

        mapResult.print().setParallelism(1);
        env.execute("FlinkSourceFromCollection");
    }
```

1.2.1 自定义单并行度数据源

```
#案例2，数据源之自定义并行度
/**
 * 自定义数据源，不支持并行
 */
public class NotParallelSource implements SourceFunction<Long> {
    private long number = 1L;
    private boolean isRunning = true;

    @Override
    public void run(SourceContext<Long> sourceContext) throws Exception {
        while (isRunning) {
            sourceContext.collect(number);
            number++;
            Thread.sleep(1000);
        }

    }
    @Override
    public void cancel() {
        isRunning = false;

    }
}

//程序入口
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();


        //数据源
        DataStreamSource<Long> longDataStreamSource = env.addSource(new
NotParallelSource()).setParallelism(1);

        SingleOutputStreamOperator<Long> filterResult =
longDataStreamSource.filter(new FilterFunction<Long>() {
            @Override
            public boolean filter(Long value) throws Exception {
                System.out.println("接受到的数据: "+value);
                return value % 2 == 0;
            }
        });


        filterResult.print().setParallelism(1);
        env.execute("FlinkWithNotParallelSource");
    }
```

1.2.2 自定义多并行度数据源

```
/*
* 案例3,自定义数据源，支持并行度
*/
public class ParallelSource implements ParallelSourceFunction<Long> {
    private long number = 1L;
    private boolean isRunning = true;

    @Override
    public void run(SourceContext<Long> sourceContext) throws Exception {
        while (isRunning) {
            sourceContext.collect(number);
            number++;
            Thread.sleep(1000);
        }
    }

    @Override
    public void cancel() {
        isRunning = false;

    }
}

//程序入口
public class FlinkWithParallelSource {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();



        //数据源
        DataStreamSource<Long> longDataStreamSource = env.addSource(new
ParallelSource()).setParallelism(2);
        SingleOutputStreamOperator<Long> mapOprator =
longDataStreamSource.map(new MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) {
                System.out.println("接受到的数据: " + value);
                return value;
            }
        });
        SingleOutputStreamOperator<Long> filterOprator = mapOprator.filter(new
FilterFunction<Long>() {
            @Override
            public boolean filter(Long value) {
                return value % 2 == 0;
            }
        });


        filterOprator.print().setParallelism(1);
        env.execute("FlinkWithParallelSource");
    }
}
```

2 常见Transformation操作

2.1 map和filter

```
public class MapAndFilterDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> numberStream = env.addSource(new
NotParallelSource()).setParallelism(1);
        /*
```

```
         * MapFunction<T, O>
         * @param <T> Type of the input elements. (输入元素的类型)
         * @param <O> Type of the returned elements. (输出元素的类型)
         *
         * */
        SingleOutputStreamOperator<Long> dataStream = numberStream.map(new
MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接受到了数据: "+value);
                return value;
            }
        });

        /**
         * FilterFunction<T>
         * @param <T> The type of the filtered elements.
         */
        SingleOutputStreamOperator<Long> filterDataStream = dataStream.filter(new
FilterFunction<Long>() {
            @Override
            public boolean filter(Long number) throws Exception {
                return number % 2 == 0;//true
            }
        });

        filterDataStream.print().setParallelism(1);
        env.execute("MapAndFilterDemo");
    }
}
```

2.2 flatMap

```
public class FlatMapAndTimeWindowDemo {
    public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env=
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> textStream =
env.socketTextStream("localhost",8888);
        //执行transformation操作
        SingleOutputStreamOperator<WordCount> wordCountStream =
textStream.flatMap(new FlatMapFunction<String, WordCount>() {
            public void flatMap(String line, Collector<WordCount> out) throws
Exception {
                String[] fields = line.split(",");
                for (String word : fields) {

                    out.collect(new WordCount(word, 1L));
                }
            }
```

```java
        }).keyBy("word")
                .timeWindow(Time.seconds(2), Time.seconds(1))//每隔1秒计算最近2秒
                .sum("count");

        wordCountStream.print().setParallelism(1);

        env.execute("FlatMapAndTimeWindowDemo");
    }

    public static class WordCount{
        public String word;
        public long count;
        public WordCount(){

        }
        public WordCount(String word,long count){
            this.word=word;
            this.count=count;
        }

        @Override
        public String toString() {
            return "WordCount{" +
                    "word='" + word + '\'' +
                    ", count=" + count +
                    '}';
        }
    }
}
```

2.3 union

```java
public class UnionAndTimeWindowAllDemo {
    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源
        DataStreamSource<Long> text1 = env.addSource(new
NotParallelSource()).setParallelism(1);//注意：针对此source，并行度只能设置为1

        DataStreamSource<Long> text2 = env.addSource(new
NotParallelSource()).setParallelism(1);

        //把text1和text2组装到一起
        DataStream<Long> text = text1.union(text2);

        DataStream<Long> num = text.map(new MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接收到数据: " + value);
                return value;
            }
        });
        //每2秒钟处理一次数据
        DataStream<Long> sum = num.timeWindowAll(Time.seconds(2)).sum(0);
        //打印结果
        sum.print().setParallelism(1);
        env.execute("UnionDemo");
    }
}
```

2.4 connect， coMap， coFlatMap

```
public class ConnectionDemo {
    public static void main(String[] args) throws Exception {
        //获取Flink的运行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        //获取数据源
        DataStreamSource<Long> text1 = env.addSource(new
NotParallelSource()).setParallelism(1);//注意：针对此source，并行度只能设置为1

        DataStreamSource<Long> text2 = env.addSource(new
NotParallelSource()).setParallelism(1);


        SingleOutputStreamOperator<String> text2_str = text2.map(new
MapFunction<Long, String>() {
            @Override
            public String map(Long value) throws Exception {
                return "str_" + value;
            }
        });

        //connect
        ConnectedStreams<Long, String> connectStream = text1.connect(text2_str);

        /**
         * Long：数据源1的类型
         * String：数据源2的类型
         * Object：输出的类型
         */
        SingleOutputStreamOperator<Object> result = connectStream.map(new
CoMapFunction<Long, String, Object>() {
            //这个方法处理的是数据源 1
            @Override
            public Object map1(Long value) throws Exception {
                return value;
```

```java
            }
            //这个方法处理的就是数据源 2
            @Override
            public Object map2(String value) throws Exception {
                return value;
            }
        });

        //打印结果
        result.print().setParallelism(1);
        String jobName = ConnectionDemo.class.getSimpleName();
        env.execute(jobName);
    }
}
```

2.5 Split和Select（了解）

```java
public class SplitAndSelectDemo {
    public static void main(String[] args) throws  Exception {
        //获取Flink的运行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源
        DataStreamSource<Long> text = env.addSource(new
NotParallelSource()).setParallelism(1);//注意：针对此source，并行度只能设置为1

        //对流进行切分，按照数据的奇偶性进行区分
        SplitStream<Long> splitStream = text.split(new OutputSelector<Long>() {
            @Override
            public Iterable<String> select(Long value) {
                ArrayList<String> outPut = new ArrayList<>();
                if (value % 2 == 0) {
                    outPut.add("even");//偶数
                } else {
                    outPut.add("odd");//奇数
                }
                return outPut;
            }
        });

        //选择一个或者多个切分后的流
        DataStream<Long> evenStream = splitStream.select("even");

        DataStream<Long> oddStream = splitStream.select("odd");
        DataStream<Long> moreStream = splitStream.select("odd","even");

        //打印结果
        moreStream.print().setParallelism(1);
        env.execute("SplitAndSelectDemo");
    }
}
```

3 常见sink操作

3.1 print() / printToErr()

打印每个元素的toString()方法的值到标准输出或者标准错误输出流中

3.2 writeAsText()

```
public class WriteTextDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> numberStream = env.addSource(new
NotParallelSource()).setParallelism(1);
        SingleOutputStreamOperator<Long> dataStream = numberStream.map(new
MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接受到的数据: "+value);
                return value;
            }
        });
        SingleOutputStreamOperator<Long> filterDataStream = dataStream.filter(new
FilterFunction<Long>() {
            @Override
            public boolean filter(Long number) throws Exception {
                return number % 2 == 0;
            }
        });

        //保存到本地

 filterDataStream.writeAsText("/flink_stage/WriteTextDemo").setParallelism(1);
        filterDataStream.print();
        env.execute("WriteTextDemo");
    }
}
```

3.3 Flink提供的sink

- Apache Kafka (source/sink)
- Apache Cassandra (sink)
- Amazon Kinesis Streams (source/sink)
- ElasticSearch (sink)
- Hadoop FileSystem (sink)
- RabbitMQ (source/sink)

- Apache NiFi (source/sink)
- Twitter Streaming API (source)
- Google PubSub (source/sink)

4. State

4.1 state概述

Apache Flink® — Stateful Computations over Data Streams

回顾单词计数的例子

```
/**
 * 单词计数
 */
public class WordCount {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> data = env.socketTextStream("localhost", 8888);
        SingleOutputStreamOperator<Tuple2<String, Integer>> result =
data.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
collector) throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(new Tuple2<>(word, 1));
                }
            }
        }).keyBy("0").sum(1);
        result.print();
        env.execute("WordCount");
    }
}
```

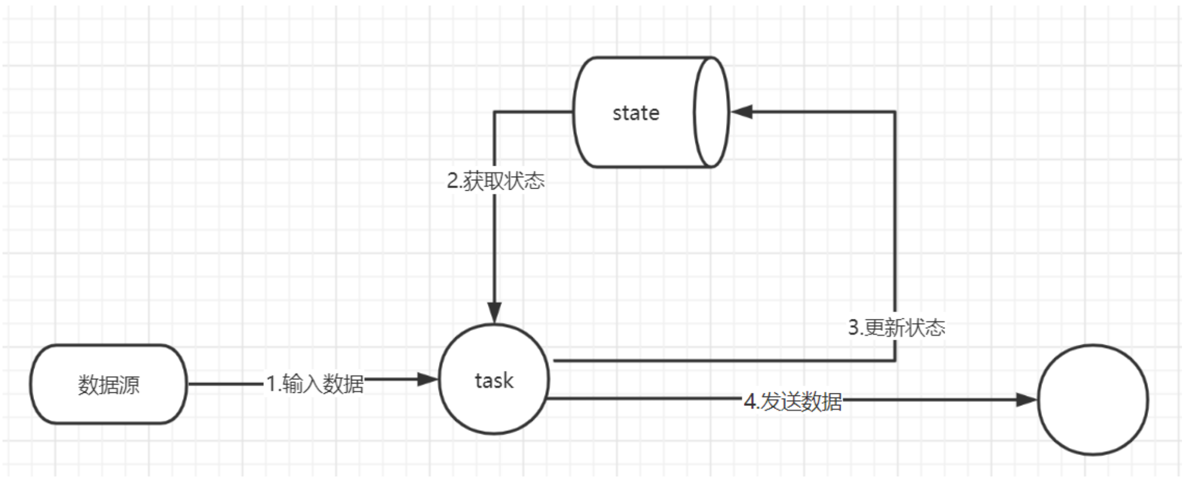
输入

```
hadoop,hadoop
hadoop
hive,hadoop
```

输出

```
4> (hadoop,1)
4> (hadoop,2)
4> (hadoop,3)
1> (hive,1)
4> (hadoop,4)
```

- 我们会发现，单词出现的次数有累计的效果。如果没有状态的管理，是不会有累计的效果的，所以Flink里面还有state的概念。

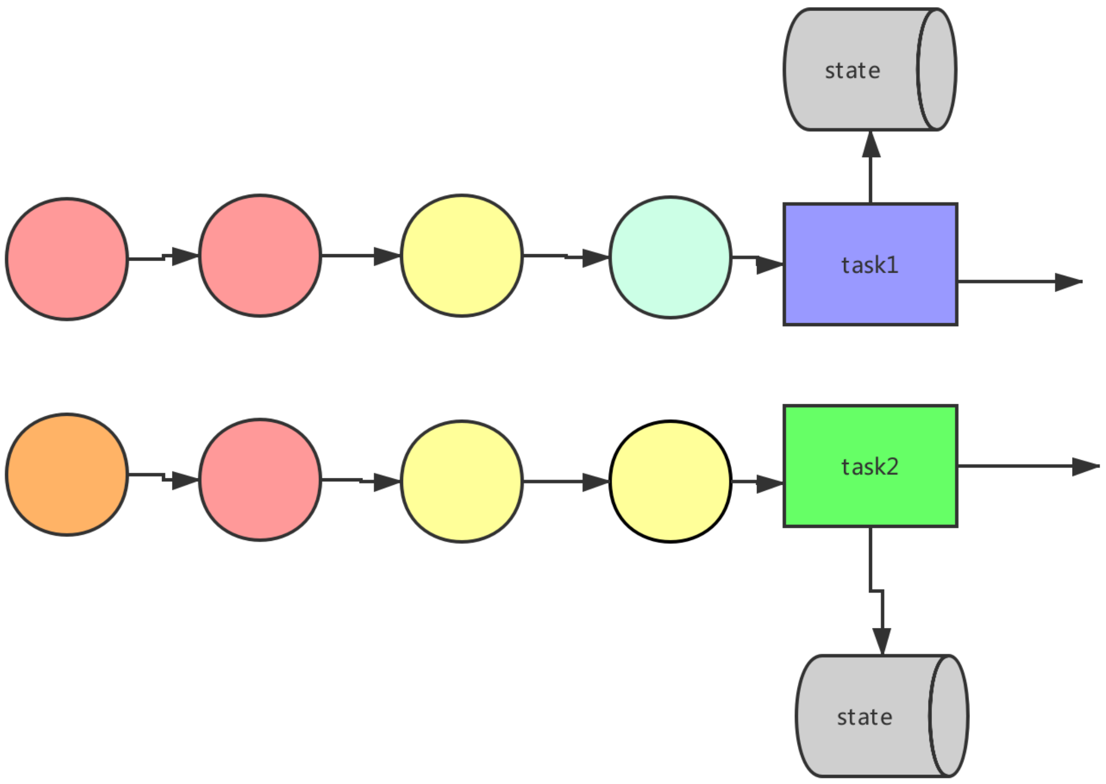


State：一般指一个具体的task/operator的状态。State可以被记录，在失败的情况下数据还可以恢复，Flink中有两种基本类型的State：Keyed State，Operator State，他们两种都可以以两种形式存在：原始状态(raw state)和托管状态(managed state)
托管状态：由Flink框架管理的状态，我们通常使用的就是这种。
原始状态：由用户自行管理状态具体的数据结构，框架在做checkpoint的时候，使用byte[]来读写状态内容，对其内部数据结构一无所知。通常在DataStream上的状态推荐使用托管的状态，当实现一个用户自定义的operator时，**会使用到原始状态**。但是我们工作中一般不常用，所以我们不考虑他。

4.2 State类型

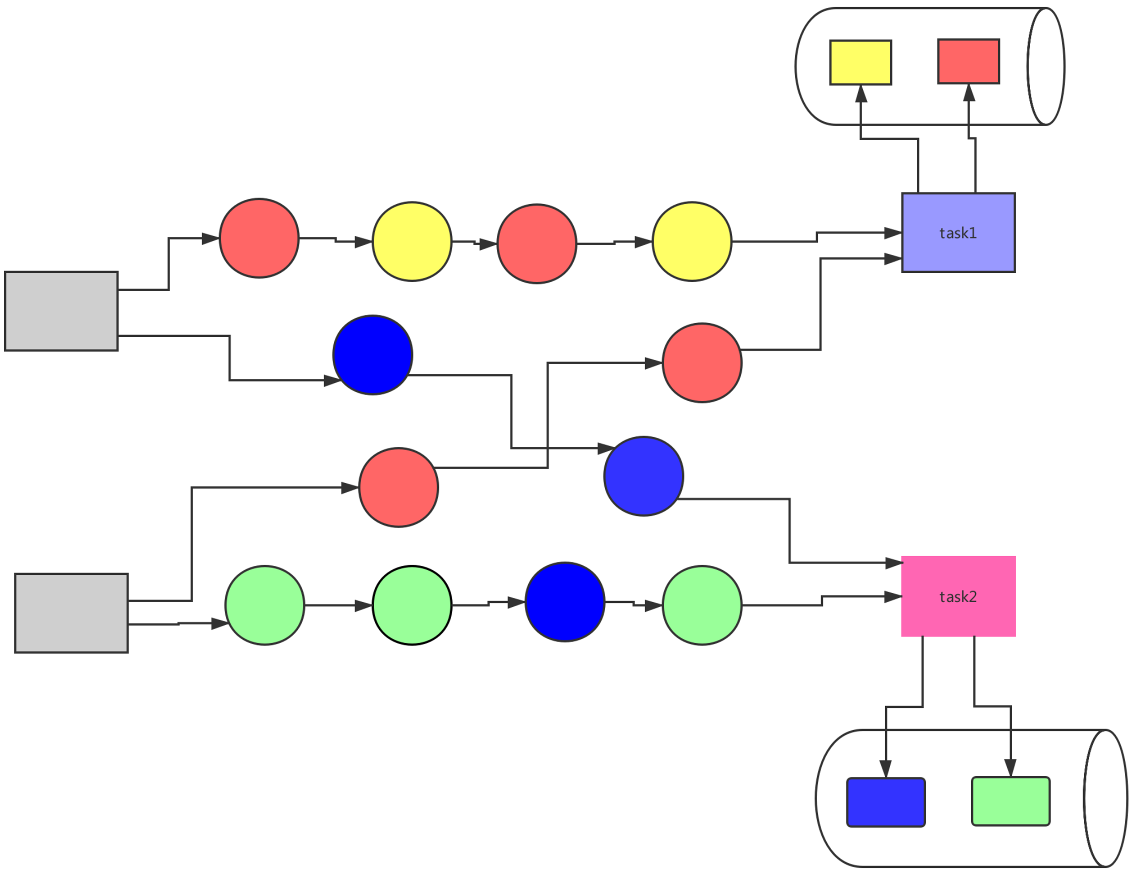
Operator State（task级别的）

- 里面没有shuffle的操作，或者说里面没有key by 的操作
- operator state是task级别的state，说白了就是每个task对应一个state
- Kafka Connector source中的每个分区（task）都需要记录消费的topic的partition和offset等信息。
- operator state 只有一种托管状态：`ValueState`



Keyed State（最常见的）

- 只要做了key by 以后后面的算子如果说有算子的话，都是KeyedState
- 针对的是Key级别的，每个Key都有自己的状态（针对的是每一个Key）

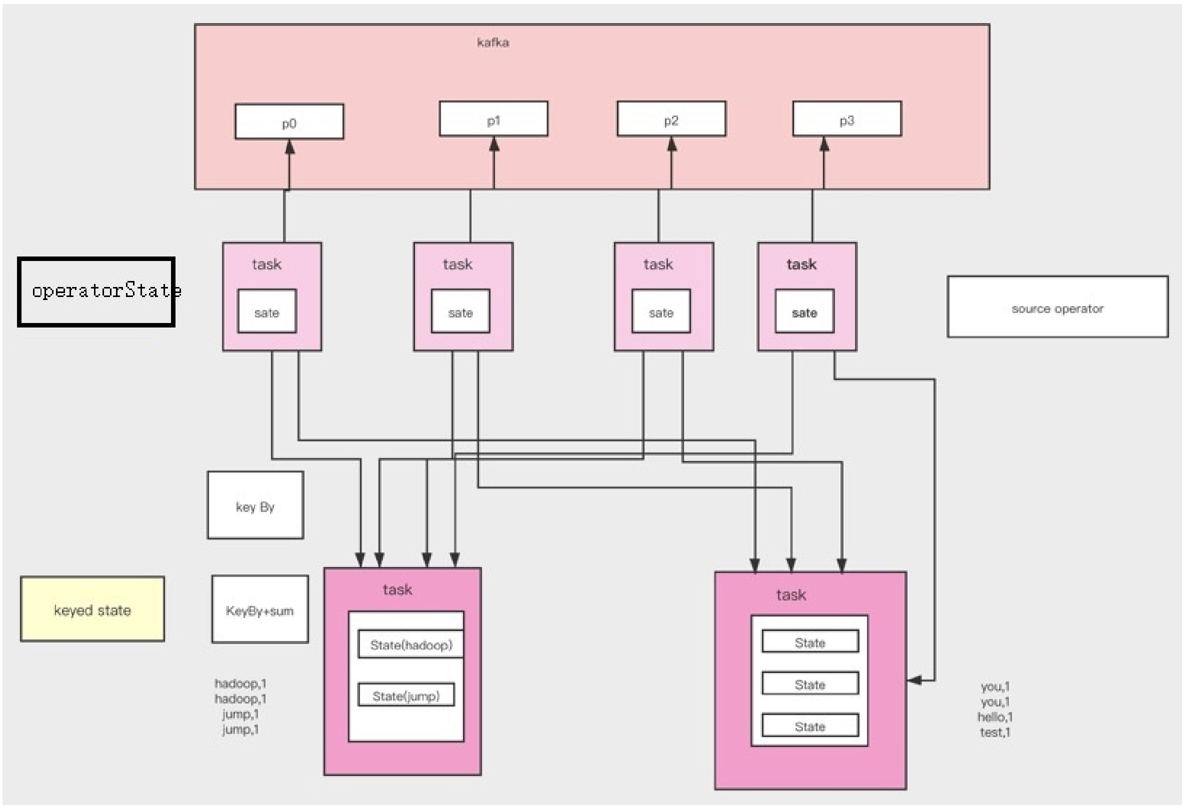


- keyed state 记录的是每个key的状态

- Keyed state托管状态有六种类型（托管）：

  1. ValueState（*）
  2. ListState（*）
  3. MapState（*）
  4. ReducingState（*）
  5. AggregatingState
  6. FoldingState

State架构图



4.3 Keyed State的案例演示

ValueState

```
/**
    需求1.
 * ValueState<T> : 这个状态为每一个 key 保存一个值
 *      value() 获取状态值
 *      update() 更新状态值
 *      clear() 清除状态
 */
public class CountWindowAverageWithValueState
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Double>> {
    // 用以保存每个 key 出现的次数，以及这个 key 对应的 value 的总值
    // managed keyed state
    //1. ValueState 保存的是对应的一个 key 的一个状态值
    private ValueState<Tuple2<Long, Long>> countAndSum;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        ValueStateDescriptor<Tuple2<Long, Long>> descriptor =
                new ValueStateDescriptor<Tuple2<Long, Long>>(
                        "average",  // 状态的名字
                        Types.TUPLE(Types.LONG, Types.LONG)); // 状态存储的数据类型
        countAndSum = getRuntimeContext().getState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Double>> out) throws Exception {
        // 拿到当前的 key 的状态值
        Tuple2<Long, Long> currentState = countAndSum.value();

        // 如果状态值还没有初始化，则初始化
        if (currentState == null) {
            currentState = Tuple2.of(0L, 0L);
        }

        // 更新状态值中的元素的个数
        currentState.f0 += 1;

        // 更新状态值中的总值
        currentState.f1 += element.f1;

        // 更新状态
        countAndSum.update(currentState);

        // 判断，如果当前的 key 出现了 3 次，则需要计算平均值，并且输出
        if (currentState.f0 >= 3) {
            double avg = (double)currentState.f1 / currentState.f0;
            // 输出 key 及其对应的平均值
            out.collect(Tuple2.of(element.f0, avg));
            //  清空状态值
            countAndSum.clear();
        }
    }
}
 (hadoop,1)  (hadoop,11)  (hadoop,20)  20 +11 + 1 / 3
/**
 * 需求：当接收到的相同 key 的元素个数等于 3 个或者超过 3 个的时候
 *  就计算这些元素的 value 的平均值。
 *  计算 keyed stream 中每 3 个元素的 value 的平均值
 */
public class TestKeyedStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L, 5L));

        // 输出:
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new CountWindowAverageWithValueState())
                .print();

        env.execute("TestStatefulApi");
    }
}

结果输出:
(1,5.0)
(2,3.6666666666666665)
```

ListState

```
/**
 *  ListState<T> : 这个状态为每一个 key 保存集合的值
 *      get() 获取状态值
 *      add() / addAll() 更新状态值, 将数据放到状态中
 *      clear() 清除状态
 */
public class CountWindowAverageWithListState
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Double>> {
    // managed keyed state
    //1. ListState 保存的是对应的一个 key 的出现的所有的元素
    private ListState<Tuple2<Long, Long>> elementsByKey;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        ListStateDescriptor<Tuple2<Long, Long>> descriptor =
                new ListStateDescriptor<Tuple2<Long, Long>>(
                        "average",  // 状态的名字
                        Types.TUPLE(Types.LONG, Types.LONG)); // 状态存储的数据类型
        elementsByKey = getRuntimeContext().getListState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Double>> out) throws Exception {
        // 拿到当前的 key 的状态值
        Iterable<Tuple2<Long, Long>> currentState = elementsByKey.get();
        // 如果状态值还没有初始化，则初始化
        if (currentState == null) {
            elementsByKey.addAll(Collections.emptyList());
        }

        // 更新状态
        elementsByKey.add(element);

        // 判断，如果当前的 key 出现了 3 次，则需要计算平均值，并且输出
        List<Tuple2<Long, Long>> allElements =
Lists.newArrayList(elementsByKey.get());
        if (allElements.size() >= 3) {
            long count = 0;
            long sum = 0;
            for (Tuple2<Long, Long> ele : allElements) {
                count++;
                sum += ele.f1;
            }
            double avg = (double) sum / count;
            out.collect(Tuple2.of(element.f0, avg));

            // 清除状态
            elementsByKey.clear();
        }
    }
}


/**
 * 需求：当接收到的相同 key 的元素个数等于 3 个或者超过 3 个的时候
 *  就计算这些元素的 value 的平均值。
 *  计算 keyed stream 中每 3 个元素的 value 的平均值
 */
public class TestKeyedStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L, 5L));

        // 输出：
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new CountWindowAverageWithListState())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

结果输出：

```
3> (1,5.0)
4> (2,3.6666666666666665)
```

MapState

```
/**
 *  MapState<K, V> : 这个状态为每一个 key 保存一个 Map 集合
 *      put() 将对应的 key 的键值对放到状态中
 *      values() 拿到 MapState 中所有的 value
 *      clear() 清除状态
 */
public class CountWindowAverageWithMapState
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Double>> {
    // managed keyed state
    //1. MapState : key 是一个唯一的值，value 是接收到的相同的 key 对应的 value 的值
    private MapState<String, Long> mapState;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        MapStateDescriptor<String, Long> descriptor =
                new MapStateDescriptor<String, Long>(
                        "average",  // 状态的名字
                        String.class, Long.class); // 状态存储的数据类型
        mapState = getRuntimeContext().getMapState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Double>> out) throws Exception {
        mapState.put(UUID.randomUUID().toString(), element.f1);

        // 判断，如果当前的 key 出现了 3 次，则需要计算平均值，并且输出
        List<Long> allElements = Lists.newArrayList(mapState.values());
        if (allElements.size() >= 3) {
            long count = 0;
            long sum = 0;
            for (Long ele : allElements) {
                count++;
                sum += ele;
            }
            double avg = (double) sum / count;
            out.collect(Tuple2.of(element.f0, avg));
            // 清除状态
            mapState.clear();
        }
    }
}


/**
 * 需求：当接收到的相同 key 的元素个数等于 3 个或者超过 3 个的时候
 *  就计算这些元素的 value 的平均值。
 *  计算 keyed stream 中每 3 个元素的 value 的平均值
 */
public class TestKeyedStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L, 5L));

        // 输出：
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new CountWindowAverageWithMapState())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

输出结果：

4> (2,3.6666666666666665)

3> (1,5.0)

ReducingState

```
/**
 *  ReducingState<T> ：这个状态为每一个 key 保存一个聚合之后的值
 *      get() 获取状态值
 *      add()  更新状态值，将数据放到状态中
 *      clear() 清除状态
 */
public class SumFunction
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Long>> {
```

```
    // managed keyed state
    // 用于保存每一个 key 对应的 value 的总值
    private ReducingState<Long> sumState;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        ReducingStateDescriptor<Long> descriptor =
                new ReducingStateDescriptor<Long>(
                        "sum",  // 状态的名字
                        new ReduceFunction<Long>() { // 聚合函数
                            @Override
                            public Long reduce(Long value1, Long value2) throws
Exception {
                                return value1 + value2;
                            }
                        }, Long.class); // 状态存储的数据类型
        sumState = getRuntimeContext().getReducingState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Long>> out) throws Exception {
        // 将数据放到状态中
        sumState.add(element.f1);

        out.collect(Tuple2.of(element.f0, sumState.get()));
    }
}


public class TestKeyedStateMain2 {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L, 5L));

        // 输出:
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new SumFunction())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

```
```

输出：

4> (2,4)

4> (2,6)

4> (2,11)

3> (1,3)

3> (1,8)

3> (1,15)

AggregatingState

```java
public class ContainsValueFunction
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, String>> {

    private AggregatingState<Long, String> totalStr;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        AggregatingStateDescriptor<Long, String, String> descriptor =
                new AggregatingStateDescriptor<Long, String, String>(
                        "totalStr",  // 状态的名字
                        new AggregateFunction<Long, String, String>() {
                            @Override
                            public String createAccumulator() {
                                return "Contains: ";
                            }

                            @Override
                            public String add(Long value, String accumulator) {
                                if ("Contains: ".equals(accumulator)) {
                                    return accumulator + value;
                                }
                                return accumulator + " and " + value;
                            }

                            @Override
                            public String getResult(String accumulator) {
                                return accumulator;
                            }

                            @Override
                            public String merge(String a, String b) {
                                return a + " and " + b;
                            }
                        }, String.class); // 状态存储的数据类型
        totalStr = getRuntimeContext().getAggregatingState(descriptor);
    }
    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, String>> out) throws Exception {
        totalStr.add(element.f1);
        out.collect(Tuple2.of(element.f0, totalStr.get()));
    }
}

public class TestKeyedStateMain2 {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L, 5L));


        dataStreamSource
                .keyBy(0)
                .flatMap(new ContainsValueFunction())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

输出：

```
4> (2,Contains： 4)
3> (1,Contains： 3)
3> (1,Contains： 3 and 5)
3> (1,Contains： 3 and 5 and 7)
4> (2,Contains： 4 and 2)
4> (2,Contains： 4 and 2 and 5)
```

4.5 案例演示：

需求：将两个流中，订单号一样的数据合并在一起输出

我是在一家电商公司，所以大家会发现后面我举的很多例子，都跟电商有关系。

不同业务线，打印出来的日志可能不一样，然后我们有时候就是需要把不同业务线的数据拼接起来。

类似于一个实时的ETL的效果。

orderinfo1数据

```
数据就是在kafka里面，其中的一个topic里面的数据就是这个样子的
订单号，购买的商品，商品的价格
123,拖把,30.0
234,牙膏,20.0
345,被子,114.4
333,杯子,112.2
444,Mac电脑,30000.0
```

orderinfo2数据

```
数据还是在卡夫卡里面，另外的一个topic
订单号，下单时间，下单的地点
123,2019-11-11 10:11:12,江苏
234,2019-11-11 11:11:13,云南
345,2019-11-11 12:11:14,安徽
333,2019-11-11 13:11:15,北京
444,2019-11-11 14:11:16,深圳
```

15> (OrderInfo1{orderId=123, productName='拖把', price=30.0},OrderInfo2{orderId=123, orderDate='2019-11-11 10:11:12', address='江苏'})

16> (OrderInfo1{orderId=234, productName='牙膏', price=20.0},OrderInfo2{orderId=234, orderDate='2019-11-11 11:11:13', address='云南'})
2> (OrderInfo1{orderId=345, productName='被子', price=114.4},OrderInfo2{orderId=345, orderDate='2019-11-11 12:11:14', address='安徽'})
11> (OrderInfo1{orderId=333, productName='杯子', price=112.2},OrderInfo2{orderId=333, orderDate='2019-11-11 13:11:15', address='北京'})
14> (OrderInfo1{orderId=444, productName='Mac电脑', price=30000.0},OrderInfo2{orderId=444, orderDate='2019-11-11 14:11:16', address='深圳'})

代码实现：

```java
public class Constants {
    public static final String
ORDER_INFO1_PATH="D:\\kkb\\flinklesson\\src\\main\\input\\OrderInfo1.txt";
    public static final String
ORDER_INFO2_PATH="D:\\kkb\\flinklesson\\src\\main\\input\\OrderInfo2.txt";
}
```

```java
public class OrderInfo1 {
    //订单ID
```

```java
     private Long orderId;
     //商品名称
     private String productName;
     //价格
     private Double price;

    public OrderInfo1(){

    }

    public OrderInfo1(Long orderId,String productName,Double price){
        this.orderId=orderId;
        this.productName=productName;
        this.price=price;
    }

     @Override
     public String toString() {
         return "OrderInfo1{" +
                 "orderId=" + orderId +
                 ", productName='" + productName + '\'' +
                 ", price=" + price +
                 '}';
     }

     public Long getOrderId() {
         return orderId;
     }

     public void setOrderId(Long orderId) {
         this.orderId = orderId;
     }

     public String getProductName() {
         return productName;
     }

     public void setProductName(String productName) {
         this.productName = productName;
     }

     public Double getPrice() {
         return price;
     }

     public void setPrice(Double price) {
         this.price = price;
     }

     public static OrderInfo1 string2OrderInfo1(String line){
         OrderInfo1 orderInfo1 = new OrderInfo1();
         if(line != null && line.length() > 0){
            String[] fields = line.split(",");
             orderInfo1.setOrderId(Long.parseLong(fields[0]));
             orderInfo1.setProductName(fields[1]);
             orderInfo1.setPrice(Double.parseDouble(fields[2]));
        }
        return orderInfo1;
    }
}
```

```
public class OrderInfo2 {
    //订单ID
    private Long orderId;
    //下单时间
    private String orderDate;
    //下单地址
    private String address;

    public OrderInfo2(){

    }
    public OrderInfo2(Long orderId,String orderDate,String address){
        this.orderId = orderId;
        this.orderDate = orderDate;
        this.address = address;
    }

    @Override
    public String toString() {
        return "OrderInfo2{" +
                "orderId=" + orderId +
                ", orderDate='" + orderDate + '\'' +
                ", address='" + address + '\'' +
                '}';
    }

    public Long getOrderId() {
        return orderId;
    }

    public void setOrderId(Long orderId) {
        this.orderId = orderId;
    }

    public String getOrderDate() {
        return orderDate;
    }

    public void setOrderDate(String orderDate) {
        this.orderDate = orderDate;
    }
```

```java
    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }


    public static OrderInfo2 string2OrderInfo2(String line){
        OrderInfo2 orderInfo2 = new OrderInfo2();
        if(line != null && line.length() > 0){
            String[] fields = line.split(",");
            orderInfo2.setOrderId(Long.parseLong(fields[0]));
            orderInfo2.setOrderDate(fields[1]);
            orderInfo2.setAddress(fields[2]);
        }

        return orderInfo2;
    }
}
```

```java
/**
 * 自定义source
 */
public class FileSource implements SourceFunction<String> {
    //文件路径
    public String filePath;
    public FileSource(String filePath){
        this.filePath = filePath;
    }

    private InputStream inputStream;
    private BufferedReader reader;

    private Random random = new Random();

    @Override
    public void run(SourceContext<String> ctx) throws Exception {

           reader = new BufferedReader(new InputStreamReader(new
FileInputStream(filePath)));
           String line = null;
           while ((line = reader.readLine()) != null) {
               // 模拟发送数据
               TimeUnit.MILLISECONDS.sleep(random.nextInt(500));
               // 发送数据
```

```
                ctx.collect(line);
            }
        if(reader != null){
            reader.close();
        }
        if(inputStream != null){
            inputStream.close();
        }

    }

    @Override
    public void cancel()  {
      try{
          if(reader != null){
              reader.close();
          }
          if(inputStream != null){
              inputStream.close();
          }
      }catch (Exception e){

      }
    }
}
```

```
public class OrderStream {
    public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> info1 = env.addSource(new
FileSource(Constants.ORDER_INFO1_PATH));
        DataStreamSource<String> info2 = env.addSource(new
FileSource(Constants.ORDER_INFO2_PATH));

        KeyedStream<OrderInfo1, Long> orderInfo1Stream = info1.map(line ->
string2OrderInfo1(line))
                .keyBy(orderInfo1 -> orderInfo1.getOrderId());

        KeyedStream<OrderInfo2, Long> orderInfo2Stream = info2.map(line ->
string2OrderInfo2(line))
                .keyBy(orderInfo2 -> orderInfo2.getOrderId());

        orderInfo1Stream.connect(orderInfo2Stream)
                .flatMap(new EnrichmentFunction())
                .print();
        env.execute("OrderStream");

    }

    /**
     *   IN1, 第一个流的输入的数据类型
         IN2, 第二个流的输入的数据类型
         OUT，输出的数据类型
     */
    public static class EnrichmentFunction extends

 RichCoFlatMapFunction<OrderInfo1,OrderInfo2,Tuple2<OrderInfo1,OrderInfo2>>{
        //定义第一个流 key对应的state
        private ValueState<OrderInfo1> orderInfo1State;
        //定义第二个流 key对应的state
        private ValueState<OrderInfo2> orderInfo2State;

        @Override
        public void open(Configuration parameters) {
            orderInfo1State = getRuntimeContext()
                    .getState(new ValueStateDescriptor<OrderInfo1>("info1",
OrderInfo1.class));
            orderInfo2State = getRuntimeContext()
                    .getState(new ValueStateDescriptor<OrderInfo2>
("info2",OrderInfo2.class));

        }

        @Override
        public void flatMap1(OrderInfo1 orderInfo1, Collector<Tuple2<OrderInfo1,
OrderInfo2>> out) throws Exception {
            OrderInfo2 value2 = orderInfo2State.value();
            if(value2 != null){
                orderInfo2State.clear();
                out.collect(Tuple2.of(orderInfo1,value2));
            }else{
                orderInfo1State.update(orderInfo1);
            }

        }

        @Override
        public void flatMap2(OrderInfo2 orderInfo2, Collector<Tuple2<OrderInfo1,
OrderInfo2>> out)throws Exception {
            OrderInfo1 value1 = orderInfo1State.value();
            if(value1 != null){
                orderInfo1State.clear();
                out.collect(Tuple2.of(value1,orderInfo2));
            }else{
                orderInfo2State.update(orderInfo2);
            }
            }
        }
}
```