20 | 场景案例：如何用 Flink 实现实时风控引擎?

在前面的课时中，我们讨论了实时流计算系统的核心概念和四种流计算框架。从今天起，我们将把其中部分知识点整合起来，以展示两个完整的实时流计算应用案例。从而帮助你更好地理解这些概念以及如何将流计算框架运用到具体业务中。

今天我们先来看第一个案例，也就是使用 Flink 实现一个包括**特征提取**和**风险评分**功能的风控引擎。

业务场景

考虑这么一种场景。有一天 Bob 窃取了 Alice 的手机银行账号和密码等信息，并准备将 Alice 手机银行上的钱，全部转移到他提前准备好的“骡子账户”（mule account，帮助转移资产的中转账户）上去。但是手机银行出于安全的原因，每次只允许最多转账 1000 元。于是 Bob 就准备每次转账 1000 元，分多次将 Alice 的钱转完。

那作为风控系统，该怎样及时检测并阻止这种异常交易呢？我们需要针对这种异常交易行为构建一个用于风险评分的规则或模型。不管是规则还是模型，它们的输入都是一些**特征**，所以我们必须先设定要提取的特征。

经过初步思考，我们想到了**如下 3 个有利于异常交易行为检测的特征**。

- 一是，过去一小时支付账户交易次数。
- 二是，过去一小时接收账户接收的总金额。
- 三是，过去一小时交易的不同接收账户数。

现在，我们已经确定了需要提取的特征，那接下来就是设定用于风险评分的规则或模型，用于判定交易行为是否异常。假设我们决定使用规则系统来判定交易是否异常，当输入的特征满足以下条件时，即判定交易是异常的。

- 一是，在过去一小时支付账户交易次数超过 5 次。
- 二是，在过去一小时接收账户接收的总金额超过 5000。
- 三是，在过去一小时交易的不同接收账户数不超过 2。

至此，风控系统要提取的特征，以及用于判定交易行为异常的规则都已经确定了。接下来，就是具体实现这个风控系统了。

实现原理

我们使用 Flink 来实现风控引擎。为了集中讨论流计算处理部分，我们这里假定从客户端上报的交易事件，已经被数据采集服务器接收后写入 Kafka 了。我们接下来只需要用 Flink 从 Kafka 读取交易事件作为输入流即可。

整体而言，我们要实现的风控引擎分成了两部分，即**特征提取部分**和**风险评分部分**。

在**特征提取部分**，我们在前面确定要提取的特征有 3 个，不过这只是为了简化讲解，在实际业务场景下可以是更多的特征，比如数十个特征。如果每计算一个特征要 50ms，那么串行执行的话，计算 3 个特征就是 150ms，计算 30 个特征就是 1500ms，这样就不太好了。因为计算时延明显增加，业务变得不那么实时，用户体验就变差了。

所以，为了减少特征计算的整体耗时，我们必须**并行计算各个特征**。

那具体怎样并行计算呢？这个时候就需要用到 Flink 中的 KeyedStream 了。Flink 中的 KeyedStream 非常贴心地提供了**将流进行逻辑分区**的功能。使用 KeyedStream，我们能够**将事件流分成多个独立的流从而实现并行计算，这正好满足了我们对特征进行并行计算的需求**。

但问题随之而来，我们应该怎样选择对流进行分区的主键呢？或许你会觉得随机分配就好了，但这是不行的！**因为特征计算的过程中，会涉及流信息状态的读写，如果特征被不受控制地随机分配到 Flink 的各个节点上去，那么就不能保证读取到与该特征相关的完整流信息状态，也就是特征计算所需使用字段的历史业务数据**。所以，必须是**按照特征使用到的字段**对流进行分区。

我们用下面图 1 展示的 Flink 实现风控系统的原理图进行详细讲解。

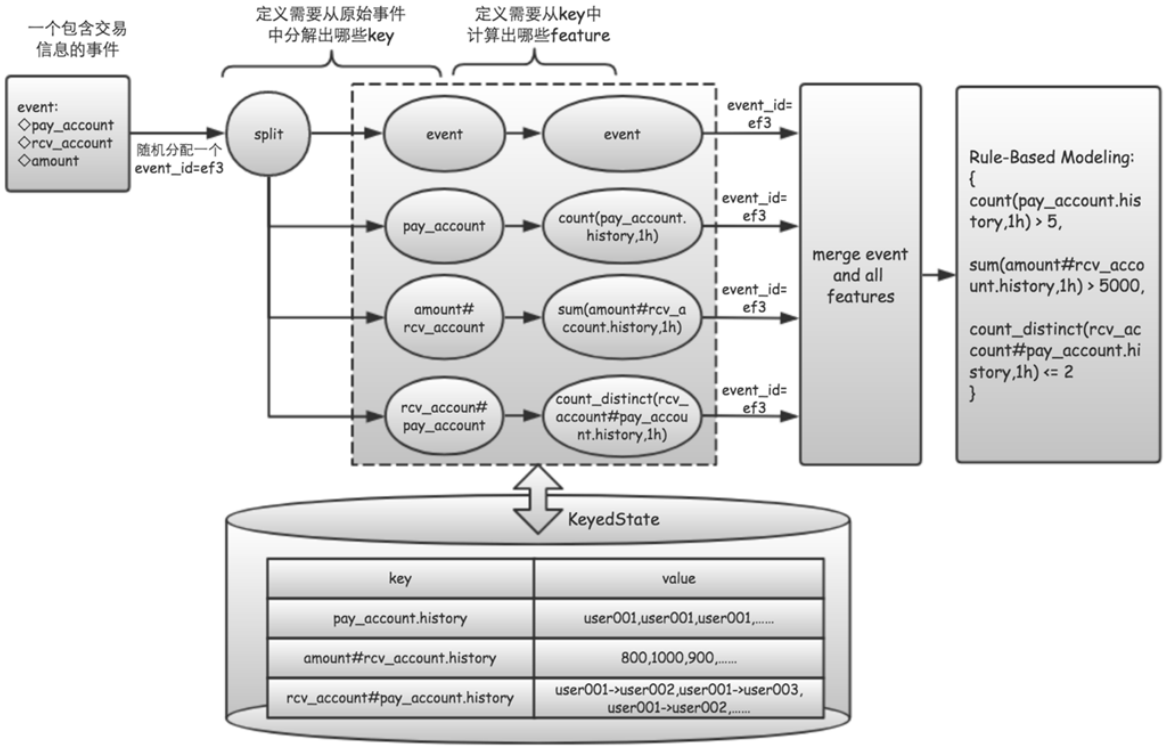

图 1 使用 Flink 实现风控系统

在上面的图 1 中，当接收到交易事件时，我们给该交易事件分配一个随机生成的事件 ID，也就是图中的 event_id。这个事件 ID 在之后会帮我们将分散的并行计算特征结果合并起来。

当分配好事件 ID 后，我们立刻通过 flatMap 操作（第 09 课时已经讲解过 flatMap），将事件分解（split）成多个“事件分身”，每个“事件分身”都包含了事件的**全部或部分字段**。并且，每个“事件分身”还需要包含**分区键（key）**信息相关字段，用于指示该“事件分身”应该划分到哪个分区流 KeyedStream 中。

比如，图 1 中我们用“count(pay_account.history,1h)”这种**自定义的 DSL**（Domain Specific Language，领域专用语言）来表示“过去一小时支付账户交易次数”这个特征，那么这个特征计算时需要用到的“事件分身”就是由 event_id、pay_account、timestamp(事件发生时间) 和**分区键名**、**分区键值**这五个字段组成的 JSON 对象，其中**分区键名**是 "pay_account.history"，**分区键值**是 “pay_account字段名#pay_account字段值.history”，代表了支付账户的历史事件。

再比如，图 1 中我们用“sum(amount#rcv_account.history,1h)”来表示“过去一小时接收账户接收的总金额”这个特征，那么这个特征计算时需要用到的“事件分身”就是由 event_id、rcv_account、amount、timestamp(事件发生时间) 和**分区键名**、**分区键值**这六个字段组成的 JSON 对象，其中**分区键名**是“amount#rcv_account.history”，**分区键值**是“amount字段名#rcv_account字段值.history”，代表了接收账户的历史交易金额。

这样，具有相同**分区键值**的“事件分身”会被划分到相同的分区流，继而路由到相同的**状态存储**节点上。在状态存储节点上，我们根据新到的“事件分身”，对 Keyed State 中保存的**历史业务数据**进行读取和更新。

比如，对于“count(pay_account.history,1h)”这个特征，我们在 Keyed State 里保存了 pay_account.history 流信息状态，也就是 pay_account 的历史交易事件数据（为了简化问题，只记录最近 100 条历史记录）。再比如，对于“sum(amount#rcv_account.history,1h)”这个特征，由于需要计算的是交易金额，所以我们在 Keyed State 里保存了 amount#rcv_account.history 流信息状态，也就是 rcv_account 的历史交易金额数据。

这里补充说明下我们的特征描述 DSL 是怎样定义的，下面的图 2 进行了说明。

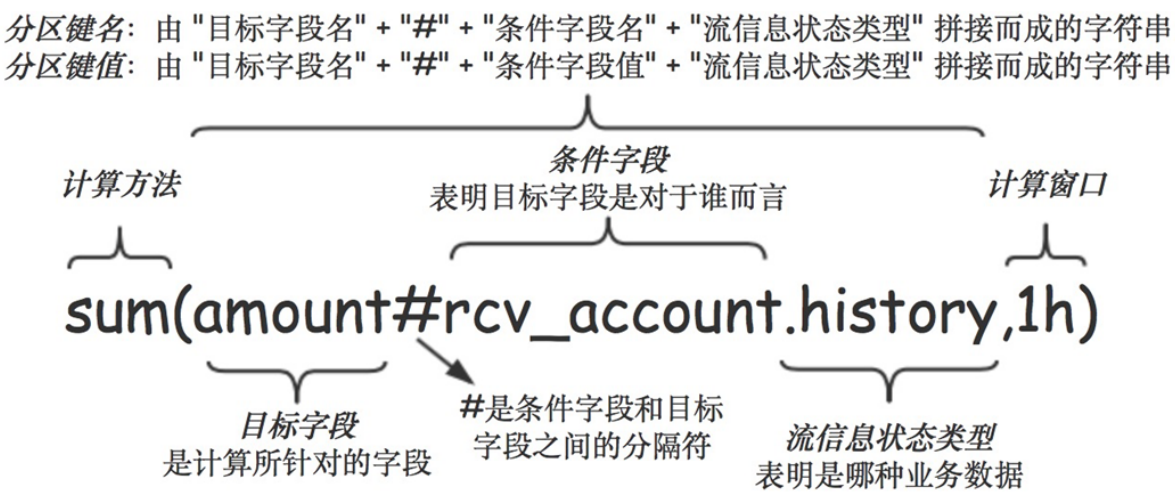

图 2 特征描述 DSL 定义

在上面的图 2 中，我们使用了“#”符号来分割“目标字段”和“条件字段”。其中，“目标字段”是计算所针对的字段，“条件字段”则表明目标字段是对于哪个字段而言。如果没有“条件字段”，也就是没有“#”符号的话，那就将“目标字段”同时也作为“条件字段”使用。

比如，在“sum(amount#rcv_account.history,1h)”中，目标字段是 amount，意味着 sum 计算是针对 amount 字段进行的，也就是说，我们是针对 amount 字段的值进行 sum 求和计算。条件字段是 rcv_account， 则意味着前面的 amount 是对于接收账户 rcv_account 而言的。而后缀“.history”则表明在 Keyed State 中保存的是**历史数据类型**的**流信息状态**，也就是接收账户 rcv_account 的交易金额 amount 的“所有历史数据”。

我们也可以自行增加其他有意义的后缀，比如用“.window”后缀表明保存的是“窗口聚合数据”等，这样就可以支持保存其他类型的**流信息状态**。最后的计算窗口“1h”，则表明了计算的时间窗口，也就是我们需要 sum 求和的是过去 1 小时的数据。

定义好各个字段后，我们再将“amount字段名#rcv_account字段值.history”作为划分 KeyedStream 的**分区键值**，这样相同接收账户 rcv_account 的交易金额 amount 数据才会路由到相同的 KeyedStream 上，进而保存到相同的 Keyed State 中。

之后，我们将从 Keyed State 中读取出的历史业务数据，附加到“事件分身”上。接下来就可以**根据这些历史业务数据计算特征**了，比如 count、sum 和 count_distinct 等。当特征计算完成后，此时**计算结果是分散在各个节点**上的，我们还需要将这些包含特征计算结果的“事件分身”合并起来，所以这一次是根据原始事件的事件 ID（也就是 event_id）对“事件分身”进行路由。由于使用的是事件 ID，所以先前被分解为多个部分的“事件分身”会被路由到相同的节点上。这样，就能**将分开的特征计算结果重新合并起来，从而得到完整的特征集合了**。

最后，将特征集合输入基于规则的**风险评分部分**，就可以判定本次转账事件是否异常了。在图 1 中，最右边的矩形就是风险评分部分。这个相对简单，只需要使用 map 操作（第 09 课时已经讲解过 map）就可以完成风险评分计算了。

至此，我们就整理清楚风控引擎的整体实现思路了。接下来，我们根据这个思路来实现具体的风控引擎。

具体实现

我们按照图 1 所示的原理，来实现基于 Flink 的风控引擎。

整体流程实现

下面的代码就是 Flink 风控引擎的整体流程。注意由于代码有点长，所以我对代码做了详细注释，你在阅读代码时可以重点关注下用"//"注释的部分（完整代码参考这里）。

```
public class FlinkRiskEngine {
    // 定义风控模型需要使用的特征，这里是用一种自定义 DSL 来定义特征的，
    // DSL 的定义规则描述见"实现原理"部分的讲解。
    private static final List<String[]> features = Arrays.asList(
        parseDSL("count(pay_account.history,1h)"),
        parseDSL("sum(amount#rcv_account.history,1h)"),
        parseDSL("count_distinct(rcv_account#pay_account.history,1h)")
    );
    // 将特征 DSL 分割成字符串数组。分割所得结果中：
    // 第 0 号元素是计算方法，比如 count、sum、count_distinct
    // 第 1 号元素是特征相关字段，同时也是分区键名，比如 amount#rcv_account.history。
    // 第 2 号元素是计算窗口，比如 1h
    private static String[] parseDSL(String dsl) {
        return Arrays.stream(dsl.split("[(,)]")).map(String::trim)
                .collect(Collectors.toList()).toArray(new String[0]);
    }
    // 获取所有特征使用到的分区键名，也就是 parseDSL 解析结果的第 1 号元素
    private static final Set<String> keys = features.stream().map(x -> x[1]).collect(Collectors.t
    // Flink 风控引擎的完整流程
    public static void main(String[] args) throws Exception {
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        env.enableCheckpointing(5000);
        // 创建从 Kafka 读取消息的流
        FlinkKafkaConsumer010<String> myConsumer = createKafkaConsumer();
        DataStream<String> stream = env.addSource(myConsumer);
        DataStream counts = stream
                // 从 Kafka 中读取的消息是 JSON 字符串，将其解码为 JSON 对象
                .map(new MapFunction<String, JSONObject>() {
                    @Override
                    public JSONObject map(String s) throws Exception {
                        if (StringUtils.isEmpty(s)) {
                            return new JSONObject();
                        }
                        // JSON 字符串解析为 JSON 对象
                        return JSONObject.parseObject(s);
                    }
                })
                // 将事件根据"特征计算所需要的字段"分解为多个"事件分身"，每个"事件分身"包含一个分区键，用于指示
                // 这里实现了 Map/Reduce 计算模式中的 Map 部分
                .flatMap(new EventSplitFunction())
                // 根据"分区键值"划分为 KeyedStream
                .keyBy(new KeySelector<JSONObject, String>() {
                    @Override
                    public String getKey(JSONObject value) throws Exception {
                        // 获取"分区键值"
                        return value.getString("KEY_VALUE");
                    }
                })
                // 将历史业务数据(也就是"流信息状态")添加到事件上
                .map(new KeyEnrichFunction())
                // 根据历史业务数据计算出特征，将特征计算结果添加到事件上
```

```
                .map(new FeatureEnrichFunction())
                // 根据事件ID将原本属于同一个事件的不同特征部分路由到相同的分区流 KeyedStream 中
                .keyBy(new KeySelector<JSONObject, String>() {
                    @Override
                    public String getKey(JSONObject value) throws Exception {
                        // 这次是用事件 ID 作为分区键值
                        return value.getString("EVENT_ID");
                    }
                })
                // 将原本属于同一个事件的特征合并起来，形成包含原始内容和特征计算结果的事件。这里实现了 Map/Red
                .flatMap(new FeatureReduceFunction())
                // 已经有了所有特征的计算结果，于是就可以根据基于规则的模型进行判定了
                .map(new RuleBasedModeling());
        counts.print().setParallelism(1);
        env.execute("FlinkRiskEngine");
    }
```

在上面的代码中，我们先从 Kafka 读取出代表交易事件的 JSON 字符串，这个步骤是通过 createKafkaConsumer 方法完成。然后，我们将输入的字符串，解析为 JSON 对象。这一步是通过 map 函数和 JSONObject.parseObject 函数完成。

再然后，由于需要并行计算所有特征，所以我们将事件根据计算特征所需要的字段，分解为多个“事件分身”。这一步是通过 flatMap 函数和 EventSplitFunction 类完成。注意，这里就是我们在 第 09 课时 讲到过的，用 flatMap 函数实现流式处理 Map/Reduce 计算模式中的 Map 部分。

接着，我们让具有相同“分区键值”的“事件分身”，划分到相同的分区流 KeyedStream 里，进而路由到相同的状态存储节点上，存储到相同的 Keyed State 里。这一步是通过 keyBy 函数和 KeySelector 类完成的。

再接着，我们在状态节点上，完成对 Keyed State 里**流信息状态**的读取和更新，并将读取出的流信息状态，也就是**历史业务数据**附加到“事件分身”上。这一步是通过 map 函数和 KeyEnrichFunction 类实现的。

之后，我们基于**历史业务数据**计算出特征，并将计算结果添加到“事件分身”上。这一步是通过 map 函数和 FeatureEnrichFunction 类完成的。

再之后，由于前面计算出特征的“事件分身”还散布在不同节点上，所以需要根据事件 ID，也就 event_id 字段，将原本属于同一事件的“事件分身”重新路由到相同的 KeyedStream 里。这一步是通过 keyBy 函数和 KeySelector 类完成的。

接着，由于属于同一事件的“事件分身”只是路由到了相同的 KeyedStream 里，要想得到完整的特征集合，还需要将这些“事件分身”合并起来。这一步是通过 flatMap 函数和 FeatureReduceFunction 类完成的。可以看到，这里就是在 第 09 课时 讲到过的，用 flatMap 函数实现流式处理 Map/Reduce 计算模式中的 Reduce 部分。

最后，当我们得到包含完整特征集合的事件后，就可以根据基于规则的模型，对这次交易事件进行风险判定了。这一步是通过 map 函数和 RuleBasedModeling 类实现的。

至此，我们就实现了一个完整的风控引擎执行流程。这个风控引擎的代码是严格按照前面所描述的实现原理所开发的，在代码的注释部分也对各个步骤做了详细说明。

不过，上面的代码描述的是一个整体流程，其中的几个关键类实现还需要进一步详细说明。

关键类实现

首先是 EventSplitFunction 类。下面是它的具体实现代码。

```
public static class EventSplitFunction implements FlatMapFunction<JSONObject, JSONObject> {
    // keys 就是所有特征 DSL 经过解析后，得到的特征相关字段，也就是"分区键名"，
    // 比如 pay_account.history 和 amount#rcv_account.history
    private static final Set<String> keys = FlinkRiskEngine.keys;
    @Override
    public void flatMap(JSONObject value, Collector<JSONObject> out) throws Exception {
        // 分解之前生成一个事件ID，用于标记分解后的事件原本属于同一次事件
        String eventId = UUID.randomUUID().toString();
        long timestamp = value.getLongValue("timestamp");
        // 原始事件本身不需要计算特征，特故而殊处理，
        // 将其分区键名设置为"event"，分区键值设置为 eventId
        JSONObject event = new JSONObject();
        event.put("KEY_NAME", "event");
        event.put("KEY_VALUE", eventId);
        event.put("EVENT_ID", eventId);
        // 将原始事件的所有字段添加到代表原始事件的分身上来
        event.putAll(value);
        // 将代表原始事件的"事件分身"输出到"事件分身"流中
        out.collect(event);
        // 下面是根据特征计算时所需的字段，同时也是"分区键名"，
        // 将事件划分为多个"事件分身"
        keys.forEach(key -> {
            JSONObject json = new JSONObject();
            // 事件发生时间戳
            json.put("timestamp", timestamp);
            // 分区键名，可以解析出特征所需字段
            json.put("KEY_NAME", key);
            // 分区键值，按照实现原理部分的图 2 所示原理生成
            json.put("KEY_VALUE", genKeyValue(value, key));
            json.put("EVENT_ID", eventId);
            // 将分区键名中包含的字段，也就是特征计算所需字段，添加到"事件分身"中
            genKeyFields(key).forEach(f -> json.put(f, value.get(f)));
            // 将"事件分身"输出到"事件分身"流中
            out.collect(json);
        });
    }
    // 根据"分区键名"，也就是"特征计算所需字段"，计算出"分区键值"。
    // "分区键值"的计算方法，参见实现原理部分的图 2 所示原理。
    private String genKeyValue(JSONObject event, String key) {
        // 只支持".history"这种类型
        // 你也可以根据实际需要在此增加新的"流信息状态类型"
        if (!key.endsWith(".history")) {
            throw new UnsupportedOperationException("unsupported key type");
        }

        // 从"分区键名"中解析出"目标字段"和"条件字段"
        // splits 的第 0 号元素是"目标字段"，第 1 号元素是"条件字段"
        String[] splits = key.replace(".history", "").split("#");
        String keyValue;
        if (splits.length == 1) {
            // splits 第 0 号元素是"目标字段"
            // 如果没有"条件字段"，就将"目标字段"当作"条件字段"
```

```
            String target = splits[0];
            // "分区键值"的生成方式是：
            // "目标字段名" + "#" + "条件字段值" + "流信息状态类
            keyValue = String.format("%s#%s.history", target, String.valueOf(event.get(target)));
        } else if (splits.length == 2) {
            // splits 第 0 号元素是"目标字段"
            String target = splits[0];
            // splits 第 1 号元素是"条件字段"
            String on = splits[1];
            // "分区键值"的生成方式是：
            // "目标字段名" + "#" + "条件字段值" + "流信息状态类型"
            keyValue = String.format("%s#%s.history", target, String.valueOf(event.get(on)));
        } else {
            throw new UnsupportedOperationException("unsupported key type");
        }
        // 返回"分区键值"
        return keyValue;
    }
    // 根据"分区键名"解析出"特征计算所需字段"。
    // 比如，分区键名"pay_account.history"表示特征计算需要使用 pay_account 字段。
    // 再比如，分区键名"amount#rcv_account.history"表示特征计算需要使用 amount 和 rcv_account 字段。
    private Set<String> genKeyFields(String key) {
        if (!key.endsWith(".history")) {
            throw new UnsupportedOperationException("unsupported key type");
        }
        // 解析"分区键名"，得到特征计算所需字段。
        // 后续会将这些字段以及它们的值添加到"事件分身"中。
        String[] splits = key.replace(".history", "").split("#");
        return new HashSet<>(Arrays.asList(splits));
    }
}
```

在上面的代码中，我们在 flatMap 函数里将代表原始事件的 value 对象，按照特征计算时所需要使用的字段进行分解，从而原本的一个事件被分解为多个“事件分身”。其中，除了一个“事件分身”代表原始事件本身以外，其他的“事件分身”则各自包含了原始事件的部分字段，之后在特征计算时需要使用到这些字段。

可以看到，这里我们用 flatMap 函数将一条数据转化成了多条数据，这正是我们在 第 09 课时 讲到过的，用 flatMap 函数实现流式处理 Map/Reduce 计算模式中的 Map 部分。

然后是 KeyEnrichFunction 类。下面是它的具体实现代码。

```
public static class KeyEnrichFunction extends RichMapFunction<JSONObject, JSONObject> {
        // ValueState 属于 Keyed State 的一种，
        // 我们用这种 ValueState 保存流信息状态，
        // 在本课时的例子中，流信息状态就是历史业务数据。
        private ValueState<Serializable> keyState;
        @Override
        public void open(Configuration config) {
            // 创建 ValueState
            keyState = getRuntimeContext().getState(new ValueStateDescriptor<>("saved keyState",
        }
        // 从 ValueState 中读取出流信息状态，也就是历史业务数据，
        // 并转化为指定类型后再返回。
        private <T> T getState(Class<T> tClass) throws IOException {
            return tClass.cast(keyState.value());
        }

        // 将流信息状态，也就是历史业务数据，保存到 ValueState 中
        private void setState(Serializable v) throws IOException {
            keyState.update(v);
        }
        @Override
        public JSONObject map(JSONObject event) throws Exception {
            // 从"事件分身"中，取出"分区键名"
            String keyName = event.getString("KEY_NAME");
            // 如果"分区键名"是event，表明这是代表原始事件的"事件分身"，
            // 所以不做处理，直接返回。
            if (keyName.equals("event")) {
                return event;
            }
            // 如果"分区键名"的后缀是".history"，
            // 表明存储的是"历史业务数据"这种类型的流信息状态。
            // 其他类型的流信息状态暂不支持，你可以自行补充，
            // 比如用".window"表明保存的是"窗口聚合数据"类型的流信息状态。
            if (keyName.endsWith(".history")) {
                // ".history"类型的流信息状态是"历史业务数据"，
                // 所以从 ValueState 中读取出来的是history
                JSONArray history = getState(JSONArray.class);
                // 如果 history 为 null，
                // 说明这是该 KeyedStream 分区流上到达的第一个事件，
                // 所以需要创建一个空的JSON数组，
                // 后续就是用这个 JSON 数组来存储"历史业务数据"，
                // 并且保存到 ValueState 里的也就是这个 JSON 数组。
                if (history == null) {
                    history = new JSONArray();
                }
                // 将最新到达的事件，添加到历史业务数据数组中
                history.add(event);
                // 我们在每个 KeyedStream 的 ValueState 中，
                // 只保存最近 100 个历史记录。
                // 这个是人为定义的，你需要根据自己的业务设置合适的值。
                if (history.size() > 100) {
                    history.remove(0);
```

```
                }
                // 将"历史业务数据"数组保存到ValueState
                setState(history);

                JSONObject newEvent = new JSONObject();
                // event 代表着"事件分身"
                newEvent.putAll(event);
                // 将读取出的"历史业务数据"添加"事件分身"上
                newEvent.put("HISTORY", history);
                return newEvent;
            } else {
                throw new UnsupportedOperationException("unsupported key type");
            }
        }
    }
```

在上面的代码中，使用 ValueState 这一 Keyed State 实现了**流信息状态**的**分布式存储**。我们将每个 KeyedStream 上最近 100 个历史记录保存在了 ValueState 中，并将历史记录附加到“事件分身”上，之后的特征计算就是基于这些历史记录。

再然后是 FeatureEnrichFunction 类。 下面是它的具体实现代码。

```
public static class FeatureEnrichFunction extends RichMapFunction<JSONObject, JSONObject> {
        // 风控模型需要使用的特征，参见 FlinkRiskEngine 类里的注释
        private static final List<String[]> features = FlinkRiskEngine.features;
        @Override
        public JSONObject map(JSONObject value) throws Exception {
            // 从"事件分身"中，取出"分区键名"
            String keyName = value.getString("KEY_NAME");
            // 如果"分区键名"是event，表明这是代表原始事件的"事件分身"，
            // 所以不做处理，
            if (keyName.equals("event")) {
                return value;
            }
            // 这里通过遍历的方式，找到 keyName 所能计算的所有特征，
            // 然后进行特征计算。这种方式讲解起来方便点，但是执行效率更低。
            // 设计更精良并且执行更高效的方式是直接通过映射表 map 的 get 方法，
            // 获取每个 keyName 能够计算的所有特征。你可以自行改造，并不难。
            for (String[] feature : features) {
                // 第 1 号元素是特征相关字段，同时也是分区键名，
                // 比如 amount#rcv_account.history。
                String key = feature[1];
                if (!StringUtils.equals(key, keyName)) {
                    continue;
                }

                // 第 0 号元素是计算方法，比如 count、sum、count_distinct
                String function = feature[0];
                // 第 2 号元素是计算窗口，比如 1h
                long window = parseTimestamp(feature[2]);
                // 取出在 KeyEnrichFunction 中附加的"历史业务数据"
                JSONArray history = value.getJSONArray("HISTORY");
                // target 是"目标字段"，也就是计算针对的字段
                String target = key.replace(".history", "").split("#")[0];
                Object featureResult;
                // 下面是根据特征计算方法选择具体实现函数，
                // 同样这里为了方便讲解，直接使用的if-else写法，
                // 更好的方法是通过映射表 map 来选择实现函数。
                if ("sum".equalsIgnoreCase(function)) {
                    // 如果计算方法是sum，就调用 sum 的实现函数 doSum，
                    // 第一个参数 history 是"历史业务数据"
                    // 第二个参数 target 是"目标字段"，也就计算针对的字段
                    // 第三个参数 window 是"计算窗口"，比如 1h
                    featureResult = doSum(history, target, window);
                } else if ("count".equalsIgnoreCase(function)) {
                    // 如果计算方法是count，就调用 count 的实现函数 doCount
                    featureResult = doCount(history, target, window);
                } else if ("count_distinct".equalsIgnoreCase(function)) {
                    // 如果计算方法是 count_distinct，
                    // 就调用 count_distinct 的实现函数 doCountDistinct
                    featureResult = doCountDistinct(history, target, window);
                } else {
                    throw new UnsupportedOperationException(String.format("unsupported function[%
                }
```

```
                value.putIfAbsent("features", new JSONObject());
                String featureName = String.format("%s(%s,%s)", feature[0], feature[1], feature[2
                // 将特征计算结果 featureResult 添加到"事件分身"的 features 字段。
                value.getJSONObject("features").put(featureName, featureResult);
            }
            return value;
        }
    }
```

在上面的代码中，在获得每个特征计算所需字段的流信息状态（在本例中也就是"历史业务数据"）后，我们根据最初特征 DSL 的定义，选择对应的计算方法和窗口，然后完成特征的计算。比如，对于“sum(amount#rcv_account.history,1h)”，计算方法 sum 对应的实现是 doSum 函数，history 参数就是接收账户 rcv_account 交易金额 amount 的历史交易数据，而 1h 就是计算窗口。

接下来是 FeatureReduceFunction 类。下面是它的具体实现代码。

```
public static class FeatureReduceFunction extends RichFlatMapFunction<JSONObject, JSONObject> {
        // ValueState 属于 Keyed State 的一种，
        // 在这里我们使用 ValueState 来记录"事件分身"的合并结果。
        // 由于所有事件分身通过事件ID event_id 路由到同一 KeyedStream 后，
        // 是依次被顺序处理，所有"事件分身"是依次逐一被合并起来的。
        // 所以，我们需要用到 ValueState 保存合并的当前状态，
        // 这就是 merged 的含义。
        private ValueState<JSONObject> merged;
        // 风控模型需要使用的特征，参见 FlinkRiskEngine 类里的注释
        private static final List<String[]> features = FlinkRiskEngine.features;
        @Override
        public void open(Configuration config) {
            // 创建 ValueState
            merged = getRuntimeContext().getState(new ValueStateDescriptor<>("saved reduceJson", 
        }
        @Override
        public void flatMap(JSONObject value, Collector<JSONObject> out) throws Exception {
            // 从 ValueState 中读取出当前"事件分身"合并的结果
            JSONObject mergedValue = merged.value();
            // 如果 mergedValue 为 null，
            // 说明这是到达的第一个"事件分身"，
            // 所以创建一个空 JSON 对象，用于之后合并所有"事件分身"。
            if (mergedValue == null) {
                mergedValue = new JSONObject();
            }

            // 从"事件分身"中，取出"分区键名"
            String keyName = value.getString("KEY_NAME");
            if (keyName.equals("event")) {
                // 将代表原始事件的"事件分身"添加到当前合并结果中
                mergedValue.put("event", value);
            } else {
                // 将包含了各个特征计算结果的"事件分身"添加到当前合并结果中
                mergedValue.putIfAbsent("features", new JSONObject());
                if (value.containsKey("features")) {
mergedValue.getJSONObject("features").putAll(value.getJSONObject("features"));
                }
            }
            if (mergedValue.containsKey("event") && mergedValue.containsKey("features")
                    && mergedValue.getJSONObject("features").size() == features.size()) {
                // 如果属于同一个事件的所有特征结果都已经收集齐全了，
                // 就可以将该合并结果输出了
                out.collect(mergedValue);
                // 由于特征已经合并完成了，状态就没有用了，
                // 将状态清掉，以免资源泄漏。
                merged.clear();
            } else {
                // 如果属于同一事件ID的各个"事件分身"尚未收集齐全，
                // 就更新下合并状态，不做任何输出
                merged.update(mergedValue);
            }
```

```
        }
    }
```

在上面的代码中，我们将分布在各个计算节点上的特征计算结果，按照事件 ID 路由到相同的 KeyedStream 上，然后用 ValueState 保存合并结果。当最终所有特征计算结果都收集齐全后，就可以将这个最终合并结果输出了。

同时为了防止资源泄漏，还清除掉不再需要的 ValueState 状态。你可以特别注意下，这里我们就是用 flatMap 函数配合 KeyedStream 和 ValueState，实现了流计算 Map/Reduce 模式中的 Reduce 部分。它与前面 EventSplitFunction 类的功能是相对应的，在 EventSplitFunction 类中，我们用 flatMap 函数配合 KeyedStream，实现了流计算 Map/Reduce 模式中的 Map 部分。

最后是 RuleBasedModeling 类。下面是它的具体实现代码。

```
public static class RuleBasedModeling implements MapFunction<JSONObject, JSONObject> {
        @Override
        public JSONObject map(JSONObject value) throws Exception {
            // 按照前面我们设定的交易异常规则，判定交易是否异常
            boolean isAnomaly = (
value.getJSONObject("features").getDouble("count(pay_account.history,1h)") > 5 && value.getJSONOb
                // 将风险判定结果添加到事件上
            value.put("isAnomaly", isAnomaly);
            return value;
        }
    }
```

在上面的代码中，我们通过基于规则的模型，来判定本次交易事件是否异常。我们将判定的结果附加在了事件的 isAnomaly 字段。这个附加了判定结果的事件，就是风控引擎的最终风险评分输出。
至此，一个基于 Flink 的风控引擎就实现了。本小节的完整代码可以 参见这里的代码 。

可以看到，我们在整个实现过程中，为了实现特征计算的并行化，充分运用了 flatMap、KeyedStream、Keyed State 的作用，实现了流计算的 Map/Reduce 或者说 Fork/Join 计算模式。

另外，我们在计算过程中，为了记录历史业务信息，还使用了 Keyed State 来保存流信息状态，这正是 Keyed State 最大的价值所在！如果没有 Flink 提供的 Keyed State，可能我们又会采用类似于 Redis 这样的外部数据库了，这会使系统复杂很多，并且一定程度会降低处理的性能以及系统的水平扩展能力。

小结

今天，我们使用 Flink 实现了一个包含特征提取和风险评分功能的风控引擎。

在实现过程中，我们充分运用了 Flink 对流的逻辑划分功能以及状态管理功能，它们分别对应着 KeyedStream 和 Keyed State。其中，KeyedStream 实现了计算能力的水平扩展，而 Keyed State 实现了状态存储的水平扩展。

如果从资源的角度看，KeyedStream 的扩展相当于是对 CPU 的水平扩展，而 Keyed State 的扩展则相当于是对“内存”的扩展。是的，你没有看错，我的意思就是“内存”，而不是“磁盘”。对于这点，你可以先仔细体会下。我在课程后续的彩蛋 1 中，对于这点还会专门详细讲解。

另外，用 flatMap、KeyedStream、Keyed State 来实现流计算的 Map/Reduce 或者说 Fork/Join 计算模式，以及用 Keyed State 来保存各种流信息状态，这两点是我希望你能够重点且熟练掌握的内容，所以你一定要好好理解下。

最后，我们今天的例子在描述特征时运用了 DSL，也就是领域定义语言。你在工作中用到过 DSL 吗？使用 DSL 有什么好处呢？可以将你的想法或问题写在留言区。

下面是本课时的知识脑图，以帮助你理解。


用 Flink 实现实时风控引擎
业务场景
为保证用户体验，需在亚秒级别时间返回，必须采用并行计算的方式来缩短处理时延
使用规则系统来判定交易是否异常
实现原理
使用 KeyedStream 实现特征并行计算，根据业务流程选择合适的“分区键”
使用 Keyed State 实现状态存储水平扩展，用 Keyed State 存储各种“流信息状态”，比如历史业务数据
充分结合 flatMap、KeyedStream、Keyed State 三者，实现流计算的 Map/Reduce 或者说 Fork/Join 计算模式
使用 DSL 来提升业务灵活扩展的能力，采用 DSL 后定义特征和规则更加轻松和灵活