Flink CEP基础学习与使用01

一，学习CEP的目的，说白了是因为业务需要，需要更深入的理解，并使用到更复杂的场景，先说一下 CEP是什么：

复杂事件处理（CEP）是一种基于流处理的技术，将系统数据看做不同类型的事件，通过分析事件之间的关系，建立不同的事件关系序列库，并利用 过滤，关联，聚合等技术，最终由简单事件产生高级事件，并通过模式规则的方式对重要信息进行跟踪和分析，从数据中发掘有价值的信息。

目前主要用于网络欺诈，故障检测，风险规避，智能营销等领域。

1，环境准备，导入依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep-scala_2.11</artifactId>
    <version>${flink.version}</version>
</dependency>
```

2，基本概念~耐着性子哟

1）事件定义

简单事件：简单事件存在于现实场景，主要处理单一事件，比如对订单统计，超过一定数量就报告。

复杂事件：复杂事件处理的不仅是单一的事件，也处理由多个事件组成的复合事件。

2）事件关系

时序关系：动作事件与动作事件之间，动作事件和状态变化事件之间，都存在时间顺序，事件与事件的时序关系决定了大部分的时序规则，例如A事件状态持续为1的同时B事件状态变为0。

聚合关系：动作事件与动作事件之间，状态变化事件和状态变化事件之间都存在聚合关系，个体聚合为整体。例如A事件状态为1的次数为10 触发警报。

层次关系：动作事件与动作事件之间，状态变化事件和状态变化事件之间都存在层次关系，父子关系，从父类到子类是具体化的，从子类到父类是泛化的。

依赖关系：A事件触发前提是B事件触发。

因果关系：A事件改变触发导致了B事件触发 。

3）事件处理

相应的规则执行相应的处理策略，这些策略包括了推断，查因，决策，预测等方面的应用。

事件推断：从一部分状态属性值推断出另一部分的状态属性值，比如已经 1+x=2 x=1。

事件查因：当出现结果状态，并且知道初始状态，可以查明是哪个动作导致的。

事件决策：知道结果状态，并且知道初始状态，可以知道要执行什么动作。

事件预测：知道初始状态，可以知道执行动作，可以知道结果状态。

3，Pattren API

FlinkCEP提供了Pattren API 用于对输入流数据的复杂事件规则定义，并从事件流中抽取事件结果。案例如下，抽取温度大于35度的信号事件结果：

```
import org.apache.flink.cep.scala.CEP
import org.apache.flink.cep.scala.pattern.Pattern
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala._

object FlinkCEP_demp {
  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val data: DataStream[Event] = env.fromElements(
      Event("A",22.2,"test1"),
      Event("B",22.2, "test2"),
      Event("C",11.1, "test3"),
      Event("D",33.3, "2"),
      Event("D",50.2, "1"),
      Event("D",35.9,"1")
    )

    //定义Pattern接口
    val pattern= Pattern.begin[Event]("start") //指定名称
        .where(_.types=="D")
        .next("middle") //接下来的名称
        .subtype(classOf[Event]) //可以将Event事件转换TempEvent事件
        .where(_.temp>=35.0)
        .followedBy("end") //结束
    //将创建好的Pattern 应用在流上面
    val patternStream = CEP.pattern(data,pattern)

    //获取触发事件
    val reslut =  patternStream.select(_.get("end"))//这里完整是要实现 PatternSelectFunction函数
    reslut.print()
//    data.print()
    env.execute()
  }

```

```
  case class  Event(types:String,temp:Double,name:String)

}
```

上面是我看书上的案例写的，在patternStream.select触发事件那个地方不是很懂~所以呢，先贴上来，等后面深入了解了再把代码改吧改吧。

感觉Flink CEP 很复杂啊~我日，看来得深入的搞搞了，还是用的太简单了，如果多个流，多个事件结合起来，会更复杂~难怪叫复杂事件处理~~~~6666666666666666，先多写几个例子练练手，从简入难~~

1）最简单的（scala版本）：

统一连续两次登陆失败的：

```
import org.apache.flink.cep.scala.pattern.Pattern
import org.apache.flink.cep.scala.{CEP, PatternStream}
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.time.Time

import scala.collection.Map

//todo 用户连续2次登陆失败的案例
object FlinkCEP_UserLoginFail {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val loginEventStream = env.fromCollection(List(
      new LoginEvent("1", "192.168.0.1", "fail"),
      new LoginEvent("1", "192.168.0.2", "fail"),
      new LoginEvent("1", "192.168.0.3", "fail"),
      new LoginEvent("2", "192.168.0.4", "fail"),
      new LoginEvent("2", "192.168.10,10", "success")
    ))

    val p = Pattern.begin[LoginEvent]("begin1")
      .where(_.getUserId.equals("1"))
      .next("second1")
      .where(_.getType.equals("fail"))
      .where(_.getUserId.equals("2"))
      .within(Time.seconds(1))
    //todo 创建流
    val stream: PatternStream[LoginEvent] = CEP.pattern(loginEventStream, p)

    val rs = stream.select((pattern: Map[String, Iterable[LoginEvent]]) => {
      val first = pattern.getOrElse("begin1", null).iterator.next()
      val second = pattern.getOrElse("second1", null).iterator.next()

      new LoginWarning(second.getUserId, second.getIp, second.getType)
    })

    rs.print()

    //todo 第一步 创建一个pattern
    val pattern: Pattern[LoginEvent, LoginEvent] = Pattern.begin[LoginEvent]("begin")
```

```
      .where(_.getType.equals("fail"))
      .next("next") //接下来操作，这里类似一个提示，切断
      .where(_.getType.equals("fail"))

      .within(Time.seconds(1)) //设置窗口期

    //todo 创建流
    val patternStream: PatternStream[LoginEvent] = CEP.pattern(loginEventStream, pattern)

    //todo 获取匹配输出
    val resultDstream = patternStream.select((pattern: Map[String, Iterable[LoginEvent]]) =>
{ //这里一定要导入 Map 要不然报错
      val first = pattern.getOrElse("begin", null).iterator.next()
      val second = pattern.getOrElse("next", null).iterator.next()

      new LoginWarning(second.getUserId, second.getIp, second.getType)

    })

//    resultDstream.print()
    env.execute()
  }
}

运行一下结果是什么呢~只会有一条数据满足。
```

思考：如果是按不同的用户分组呢？？？id分组~ 后续 实现一下。

2）第二个案例，数据是json数据，不一定要是实体类~

```
import java.util

import com.alibaba.fastjson.JSONObject
import org.apache.flink.cep.PatternSelectFunction
import org.apache.flink.cep.pattern.conditions.SimpleCondition
import org.apache.flink.cep.scala.CEP
import org.apache.flink.cep.scala.pattern.Pattern
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.time.Time

object FlinkStateDemo2 {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    val json1 = new JSONObject()
    json1.put("id", 1);
    json1.put("user", "aaa");
    json1.put("event_type", 1);
    json1.put("event_time", 1000);

    val json2 = new JSONObject()
    json2.put("id", 2);
    json2.put("user", "bbb");
```

```
    json2.put("event_type", 2);
    json2.put("event_time", 2000);

    val json3 = new JSONObject()
    json3.put("id", 3);
    json3.put("user", "ccc");
    json3.put("event_type", 3);
    json3.put("event_time", 3000);

    val json4 = new JSONObject()
    json4.put("id", 4);
    json4.put("user", "ddd");
    json4.put("event_type", 4);
    json4.put("event_time", 4000);

    val json5 = new JSONObject()
    json5.put("id", 5);
    json5.put("user", "ddd");
    json5.put("event_type", 5);
    json5.put("event_time", 5000);

    val dataStream: DataStream[JSONObject] = env.fromElements(json1, json2, json3, json4, json5)
    val keyByStream: KeyedStream[JSONObject, String] = dataStream.keyBy(_.getString("user"))

    //todo 创建 pattern
    val pattern = Pattern.begin("start")
      //todo 添加条件限制
      .where(new SimpleCondition[JSONObject] {
      override def filter(value: JSONObject): Boolean = {
        value.getString("user").equals("ddd")
      }
    })
      //todo  模式发生大于等于N次, greedy 代表越多越好
      .timesOrMore(2).greedy
      .within(Time.seconds(1));
    val finalStream = CEP.pattern(keyByStream, pattern)
    //todo 匹配数据,实现函数          //函数<IN, OUT>
    val rs = finalStream.select(new PatternSelectFunction[JSONObject, util.List[JSONObject]]{
      override def select(map: util.Map[String, util.List[JSONObject]]): util.List[JSONObject] = {
        val rs: util.List[JSONObject] = map.get("start")
        rs
      }
    })

    rs.print()
    env.execute()
  }


}
```

结果返回几条数据~~2条。。。

总结：要主要的是每个函数的写法，需要返回什么，scala就这点不好~遇到个坑 在实现 PatternSelectFunction函数的时候返回List （scala） 的，结果转半天结果实现不了，索性直接使用了java util 。 这个案例是将每个环节的函数写出来实现了，方面记忆跟理解。。。。。。

3）再来一个类似的案例

```
import java.{lang, util}

import com.alibaba.fastjson.JSONObject
import org.apache.flink.cep.PatternSelectFunction
import org.apache.flink.cep.pattern.conditions.{IterativeCondition, SimpleCondition}
import org.apache.flink.cep.scala.CEP
import org.apache.flink.cep.scala.pattern.Pattern
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.time.Time

object FlinkStateDemo3 {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    val json1 = new JSONObject()
    json1.put("id", 1);
    json1.put("user", "aaa");

    json1.put("event_type", 1);
    json1.put("event_time", 1000);

    val json2 = new JSONObject()
    json2.put("id", 2);
    json2.put("user", "bbb");
    json2.put("event_type", 2);
    json2.put("event_time", 2000);

    val json3 = new JSONObject()
    json3.put("id", 3);
    json3.put("user", "ccc");
    json3.put("event_type", 3);
    json3.put("event_time", 3000);

    val json4 = new JSONObject()
    json4.put("id", 4);
    json4.put("user", "ddd");
    json4.put("event_type", 4);
    json4.put("event_time", 4000);

    val json5 = new JSONObject()
    json5.put("id", 5);
    json5.put("user", "ddd");
    json5.put("event_type", 4);
    json5.put("event_time", 5000);

    val json7 = new JSONObject()
    json7.put("id", 7);
    json7.put("user", "ddd");
```

```
    json7.put("event_type", 1);
    json7.put("event_time", 7000);


    val json6 = new JSONObject()
    json6.put("id", 6);
    json6.put("user", "ddd");
    json6.put("event_type", 4);
    json6.put("event_time", 6000);

    val dataStream: DataStream[JSONObject] = env.fromElements(json1, json2, json3, json4, json5,json6,json7)
    val keyByStream: KeyedStream[JSONObject, String] = dataStream.keyBy(_.getString("user"))

    //todo 创建 pattern
    val pattern = Pattern.begin("start")
      //todo 添加条件限制
      .where(new SimpleCondition[JSONObject] {
      override def filter(value: JSONObject): Boolean = {
        value.getString("user").equals("ddd")
      }
    })
      //todo   使用松散模式
      .followedBy("followed")
      //todo  这里使用了IterativeCondition函数,可以拿到第一个模式的数据
      .where(new IterativeCondition[JSONObject] {
      override def filter(log: JSONObject, context: IterativeCondition.Context[JSONObject]): Boolean = {
        if (log.getString("event_type") != 4) {
          false
        }
        val startJson: lang.Iterable[JSONObject] = context.getEventsForPattern("start")
        println("打印看看: " + startJson)
        true
      }
    })
      .within(Time.seconds(1));
    val finalStream = CEP.pattern(keyByStream, pattern)
    //todo 匹配数据,实现函数          //函数<IN, OUT>
    val rs = finalStream.select(new PatternSelectFunction[JSONObject, util.List[JSONObject]]{
      override def select(map: util.Map[String, util.List[JSONObject]]): util.List[JSONObject] = {
        val rs: util.List[JSONObject] = map.get("followed")
        rs
      }
    })

    rs.print()
    env.execute()
  }
```

总结：结果是条数据~ 这里主要是想使用一下 IterativeCondition函数 使用 followedBy模式 --- 它与next 模式的区别就是 next是严格的。followedBy不是严格的。例子 next: A-B 数据 触发 A-C-B 不会触发 。 followedBy： A-B 触发 , A-C-B触发。

下一节接着讲API，经过3个案例之后再学API 感觉会更明白清晰一点。。。棒棒哒，每天有进步~~