Flink CEP基础学习与使用02 --主要是API 学习

好了~ 上一篇是3个案例，我们先学API 熟悉之后再搞复杂点的案例，最后实战.....

一，模式定义

1，个体模式

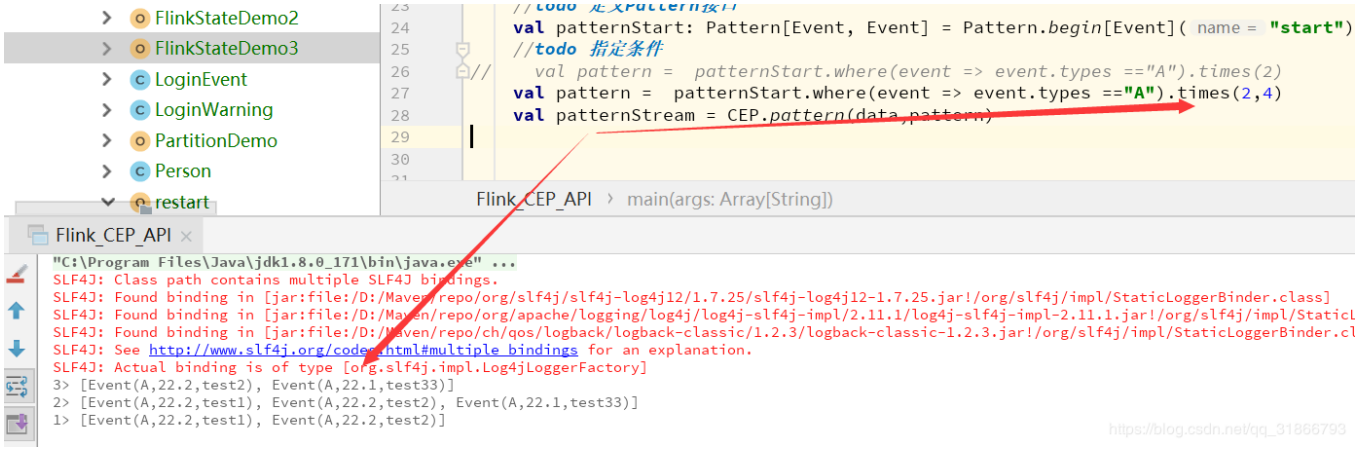
类型	API	含义
量词API	times()	模式发生次数 示例： pattern.times(2,4)，模式发生2,3,4次
	timesOrMore（） oneOrMore()	模式发生大于等于N次 示例： pattern.timesOrMore(2)，模式发生大于等于2次
	optional()	模式可以不匹配 示例： pattern.times(2).optional()，模式发生2次或者0次
	greedy()	模式发生越多越好 示例： pattern.times(2).greedy()，模式发生2次且重复次数越多越好
条件API	where()	模式的条件 示例： pattern.where(_.ruleId=43322)，模式的条件为ruleId=433322
	or()	模式的或条件 示例： pattern.where(_.ruleId=43322).or(_.ruleId=43333)，模式条件为ruleId=43322或者43333
	util()	模式发生直至X条件满足为止 示例： pattern.oneOrMore().util(condition)模式发生一次或者多次，直至condition满足为止


```
//todo 定义Pattern接口
val patternStart: Pattern[Event, Event] = Pattern.begin[Event]("start")
//todo 指定条件
patternStart.where(event => event.types =="A")
```

1）times 指定模式发生次数，所以可能有多个结果

```
 val pattern =  patternStart.where(event => event.types =="A").times(2)

 val pattern =  patternStart.where(event => event.types =="A").times(2,4)
```



2）optional 通过此关键字指定要么不触发，要么触发指定的次数

```
val pattern =  patternStart.where(event => event.types =="A").times(2).optional

val pattern =  patternStart.where(event => event.types =="A").times(2,4).optional
```

3）greedy 标记为贪婪模式，在匹配成功的前提下，尽可能多的触发。

```
//todo greedy 模式
val pattern2 =  patternStart.where(event => event.types =="A").times(2,4).greedy
val pattern3 =  patternStart.where(event => event.types =="A").times(2,4).optional.greedy
```

4）oneOrMore 可以通过oneOrMore方法指定触发一次或者多次。

```
//todo oneOrmore
val pattern4 =  patternStart.where(event => event.types =="A").oneOrMore

//todo 尽可能重复执行
val pattern5 =  patternStart.where(event => event.types =="A").oneOrMore.greedy

//todo 触发0次或者多次
val pattern6 =  patternStart.where(event => event.types =="A").oneOrMore.optional

//todo 触发0次或者多次 尽可能的重复
val pattern7 =  patternStart.where(event => event.types =="A").oneOrMore.optional.greedy
```

6） timesOrMor 指定固定触发固定次数以上，例如执行两次以上

```
//todo 触发两次以上
val pattern8 =  patternStart.where(event => event.types =="A").timesOrMore(2)
val pattern9 =  patternStart.where(event => event.types =="A").timesOrMore(2).greedy
val pattern10 =  patternStart.where(event => event.types
=="A").timesOrMore(2).optional.greedy
```

2，定义模式条件：

FlinkCEP在通过 条件API (where or until)方法的时候实现的函数类型有三种：

条件API	where()	模式的条件 示例： pattern.where(_.ruleId=43322)，模式的条件为ruleId=433322
	or()	模式的或条件 示例： pattern.where(_.ruleId=43322).or(_.ruleId=43333)，模式条件为ruleId=43322或者43333
	util()	模式发生直至X条件满足为止 示例： pattern.oneOrMore().util(condition)模式发生一次或者多次，直至condition满足为止

1）Iterative Conditions （迭代条件）： 能够对前面模式所有接受的数据进行处理，根据接收的事件集合统计出计算指标，并作为本次模式匹配中的条件输出参数--------使用场景，统计数量大于小于或者平均值大小，下面的例子是两种写法，看个人接受哪种写法：

```
val patternStart: Pattern[Event, Event] = Pattern.begin[Event]("start")
patternStart.where(
  (value,ctx) =>{
   val sum =  ctx.getEventsForPattern("start").map(_.temp).sum
    value.name.equals("D") && sum >1
}
)

//todo 完整的函数写法
patternStart.where(new IterativeCondition[Event] {
  override def filter(t: Event, ctx: IterativeCondition.Context[Event]): Boolean = {
    true
  }
})

2) Simple Conditions ：其主要是根据事件中的字段信息进行判断，决定是否接受该条件 -----使用场景，更具字段判断过滤~案例写法：
```

```
patternStart.where(new IterativeCondition[Event] {
  override def filter(t: Event, context: IterativeCondition.Context[Event]): Boolean = {
    true
  }
})

patternStart.where(event => event.name.equals("A"))
patternStart.where(new SimpleCondition[Event] {
  override def filter(value: Event): Boolean = {
    true
  }
})
```

3）组合条件，就是将简单的条件组合：

```
val patternStart: Pattern[Event, Event] = Pattern.begin[Event]("start")
patternStart.where(
  (value,ctx) =>{
   val sum =  ctx.getEventsForPattern("start").map(_.temp).sum
    value.name.equals("D") && sum >1
```

```
}
).or(enent =>{
  enent.name.equals("A")
  true
})
```

4) 终止条件

```
patternStart.where(
  (value,ctx) =>{
   val sum =  ctx.getEventsForPattern("start").map(_.temp).sum
    value.name.equals("D") && sum >1
}
).until(event =>{
  event.name.endsWith("A")
})
```

3，联合模式

API	含义
next()	严格的满足条件 示例： 模式为begin("first").where(_.name='a').next("second").where(.name='b') 当且仅当数据为a,b时，模式才会被命中。如果数据为a,c,b，由于a的后面跟了c，所以a会被直接丢弃，模式不会命中。
followedBy()	松散的满足条件 示例： 模式为begin("first").where(_.name='a').followedBy("second").where(.name='b') 当且仅当数据为a,b或者为a,c,b，，模式均被命中，中间的c会被忽略掉。
followedByAny （）	非确定的松散满足条件 模式为begin("first").where(_.name='a').followedByAny("second").where(.name='b') 当且仅当数据为a,c,b,b时，对于followedBy模式而言命中的为{a,b}，对于followedByAny而言会有两次命中{a,b},{a,b}
within （）	模式命中的时间间隔限制
notNext() notFollowedBy()	后面的模式不命中（严格/非严格）

1）严格邻近---就是必须严格满足 next

```
pattern10.next("middle").where(_.name.contains("a"))
```

2）宽松邻近 ---可以理解为 or的逻辑关系 followedBy

```
pattern10.followedBy("middle").where(_.name.contains("a"))
```

3）非确定宽松邻近followedByAny---可以理解为在followedBy的基础上忽略已经匹配的条件：

```
pattern10.followedByAny("middle").where(_.name.contains("a"))
```

4）剩余的还有 notNext,NotfollewBy,注意!!!! Not类型不能跟optional关键字同时使用

```
pattern10.notNext("middle").where(_.name.contains("a"))
pattern10.notFollowedBy("middle").where(_.name.contains("a"))
```

4，模式组

多个模式组合起来：

```
val pattern8 =  patternStart.where(event => event.types
=="A").timesOrMore(2).next("next").where(_.name.equals("B")).timesOrMore(2)

```

5，AfterMatchSkipStrategy 忽略策略

在给定的pattern中，当同一事件符合多种模式条件组合之后，需要指定AfterMatchSkipStrategy处理已经匹配的事件，主要有四种事件处理策略，分别为 NO_SKIP , SKIP_PAST_LAST_EVENT , SKIP_TO_FIRST , SKIP_TO_LAST。

忽略策略	含义
NO_SKIP	不忽略 在模式为:begin("start").where(_.name='a').oneOrMore().followedBy("second").where(_.name='b') 对于数据：a,a,a,a,b 模式匹配到的是:{a,b},{a,a,b},{a,a,a,b},{a,a,a,a,b}
SKIP_PAST_LAST_EVENT	在模式匹配完成之后，忽略掉之前的部分匹配结果 在模式为:begin("start").where(_.name='a').oneOrMore().followedBy("second").where(_.name='b') 对于数据：a,a,a,a,b 模式匹配到的是:{a,a,a,a,b}
SKIP_TO_FIRST	在模式匹配完成之后，忽略掉第一个之前的部分匹配结果
SKIP_TO_LAST	在模式匹配完成之后，忽略掉最后一个之前的部分匹配结果 在模式为:begin("start").where(_.name='a').oneOrMore().followedBy("second").where(_.name='b') 对于数据：a,a,a,a,b 模式匹配到的是:{a,b},{a,a,b},{a,a,a,a,b}

使用：

```
val skip1 = AfterMatchSkipStrategy.noSkip()
val skip2 = AfterMatchSkipStrategy.skipPastLastEvent()
val skip3 = AfterMatchSkipStrategy.skipToFirst("start") // start 对应哪个pattern
val skip4 = AfterMatchSkipStrategy.skipToLast("start") // start 对应哪个pattern
Pattern.begin[Event]("start",skip1)
Pattern.begin[Event]("start",skip2)
Pattern.begin[Event]("start",skip3)
Pattern.begin[Event]("start",skip4)
```

6，事件结果获取

1，通过Select Function抽取正常事件， 每次调用之后仅输出一条结果

```
patternStream.select((pattern2 : Map[String, Iterable[Event]])=> {
```

```
  val start = pattern2.get("start").get.iterator.next()
  val middle = pattern2.get("middle").get.iterator.next()
  "xx"
})
```

2，Flat Select Funcitoon 抽取正常事件,跟select 类似 不过是数据多条数据

```
patternStream.flatSelect((pattern3 : Map[String, Iterable[Event]],ctx:Collector[String])=> {
  val start = pattern3.get("start").get.iterator.next()
  val middle = pattern3.get("middle").get.iterator.next()
  for( i<- 0 to start.temp.toInt){
    ctx.collect("xx")
  }
})
```

3）通过 Select Function抽取超时事件

注意两个点， 需要创建OutputTag 来标记超时事件， 然后在select方法里面使用OutputTag，就可以将超时事件抽取出来。

```
//todo 创建OutputTag 并命名为 "time-output"
val timeOutTag = OutputTag[String]("time-output")
val rs1 = patternStream.select(timeOutTag) {
  (pattern1: Map[String, Iterable[Event]], timestap: Long) => "timeOut" //todo 超时事件获取
} {
  pattern2: Map[String, Iterable[Event]] => "normal" //todo 返回正常事件

}
//todo 调用方法，并将超时事件数据
val timeOutRs = rs1.getSideOutput(timeOutTag)
```

4）通过 Flat Select Function抽取超时事件

```
val flattimeOutTag = OutputTag[String]("flattimeOutTag")
val rs2 = patternStream.flatSelect(flattimeOutTag) {
  (pattern1: Map[String, Iterable[Event]], timestap: Long,ctx:Collector[String]) =>
    ctx.collect("xxx") //输出 超时的
} {
  (pattern2 : Map[String, Iterable[Event]],ctx2:Collector[String]) =>
    ctx2.collect("xxx") //输出正常的
}
//todo 调用方法，并将超时事件数据
val timeOutRs2 = rs1.getSideOutput(flattimeOutTag)
```

最后来一个应用案例：

```
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

//https://yq.aliyun.com/articles/259094
```

```scala
object FlinkCEP_demp2 {

  case class MonitorEvent(id: String, std: Int, name: String)

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

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream: DataStream[MonitorEvent] = env.fromElements(
      MonitorEvent("A", 1, "test1"),
      MonitorEvent("B", 2, "test2"),
      MonitorEvent("C", 3, "test3"),
      MonitorEvent("D", 4, "2"),
      MonitorEvent("D", 5, "1"),
      MonitorEvent("D", 6, "1")
    )

    // 根据ID分区
    val keybyStream = dataStream.keyBy(event => event.id)

    //创建pattern
    val pattern2 = Pattern.begin[MonitorEvent]("start")
      .next("middle").where((event, ctx) => event.name == "1")
      .followedBy("end").where((event, ctx) => event.std >= 1)
      .within(Time.seconds(1))

    //创建流
    val stream: PatternStream[MonitorEvent] = CEP.pattern(keybyStream, pattern2)

    //调用输出
    val rs: DataStream[MonitorEvent] = stream.select(event => selectFn(event))


    env.execute()
  }

  def selectFn(pattern2: Map[String, Iterable[MonitorEvent]]): MonitorEvent = {
    val startEvent: MonitorEvent = pattern2.get("start").iterator.next().toList(0)  // 这个地方又学了一招~
    startEvent
  }


}
```