12 | 事件序列分析：大家都在说的 CEP 是怎么一回事?

在前面两个课时中，我分别讲解了“时间维度聚合值”计算和“关联图谱分析”的方法。这两者分别是对流数据，在“时间”维度和“空间”维度进行的聚合分析。但其实，除了这两种聚合分析以外，针对流数据我们还会做另外一种形式的聚合分析，也就是，对流数据中数据和数据之间的关联关系进行聚合分析。这里，我将这种聚合分析，称之为“事件序列分析”。

所以，我们今天讨论的重点，就是“事件序列分析”技术。

事件序列分析（CEP）

什么是“事件序列分析”呢？在流数据中，数据不是单纯在时间上有着先来后到的关系，很多时候，数据和数据之间也有着联系。

比如，用户在手机上安装新 App 的过程，他可能是先点击了某个广告链接，然后下载并安装了App，最后成功注册了账号。从“点击”到“下载”，再到“安装”和“注册”，这就完成了一次将广告转化为用户的过程。

再比如，在网络欺诈识别场景中，如果用户在新建账号后，立马发生大量交易行为。那么这种“新建账号”到“10 分钟内 5 次交易”的行为，就是非常可疑的了。

诸如此类**从数据流表示的事件流中，检测并筛选出符合特定模式的事件序列的过程**，我们就称之为“事件序列分析”。

另外，我们也会将“事件序列分析”，称之为“复杂事件处理”，也就是我们常说的 CEP （Complex Event Processing）。为了方便起见，下面我就都统一称之为 CEP 了。

CEP 技术常用场景

CEP 是一种非常有用，并且也很有趣的技术。特别是在 Flink CEP 已经为我们做了很多优秀工作的基础上，现在 CEP 的使用场景越来越多。比如以下几种场景。

首先是银行卡异常检测。比如，如果一张银行卡在 30 分钟内，连续 3 次转账给不同银行卡，或者在 15 分钟内在 2 个不同城市取款，则意味着该银行卡行为异常，有可能被盗或被骗，需要给持卡人发送告警短信，并采取相应阻断措施。

然后是工厂环境监控。比如，某纸筒生产车间为了保证安全生产，在车间安装了温度传感器，当温度传感器上报的环境温度记录，出现 1 次高温事件时，需要发送轻微告警，而当 30 秒内连续 2 次出现高温事件时，则需要发出严重告警。

接着是推荐系统。比如，如果用户在 10 分钟之内点击了 3 次同类商品，那么他很可能对该类商品感兴趣，之后可以更加主动地给他推荐同类商品。

最后是离职员工数据泄露检测。比如，如果员工最近经常访问招聘网站，邮件的附件很大，还用 USB 拷贝数据，那么该员工准备离职的可能性就比较大，公司需要提前采取措施。

除了以上列举的几个例子外，CEP 使用的场景还有很多。 CEP 是一个我觉得非常有趣的技术，因为只要设置好了感兴趣的事件发生模式，再将这个模式安装到数据流上，之后就会从数据流中不断冒出符合我们所设置模式的事件序列。这些事件序列有着明确的业务含义，告诉我们现在系统正在发生着什么，以及我们需要做什么。

看了这么多例子，接下来我们就来看下，到底该如何进行 CEP 编程?

CEP 编程方法

说到能够提供 CEP 编程工具的产品，其实还比较多，比如 Siddhi、Drools、Pulsar、Esper、Flink CEP 等。但由于在这些产品中，只有 Flink CEP 和我们课程的主题最相关，而且它就是构建在流计算框架 Flink 之上的。所以，接下来我们以 Flink CEP 来讲解 CEP

的编程方法。

在 Flink CEP 中，我们**将事件之间各种各样的关系，抽象为“模式（Pattern）”**。在定义好“模式”后，再将这个“模式”安装到数据流上，之后当数据流过时，如果匹配到定义的“模式”，就会触发一个“复合事件”。这个“复合事件”包含了所有参与这次“模式”匹配的事件。

这么讲可能有点抽象，我们以前面“推荐系统”的例子详细说明下“模式”具体是什么。在“推荐系统”的例子中，“在 10 分钟之内点击了 3 次同类商品”就是“复合事件”的“模式”。当某个用户的一连串操作，符合上面这个“在 10 分钟之内点击了 3 次同类商品”的“模式”时，就会产生一个意味着用户对此类商品感兴趣的“复合事件”。很明显，在用户的一连串操作中，任何一个单独的操作，都不足以说明用户对该类商品感兴趣，因为他有可能是手滑误点了。但是，如果是“在 10 分钟之内点击了 3 次同类商品”的话，就足以说明用户很可能对这类商品感兴趣了，于是推荐系统就可以给他继续推荐此类商品。

由此可以看出，在 Flink CEP 中，如何定义“复合事件”的“模式”是最核心的问题。为此，Flink CEP 为我们提供了丰富的有关构建“模式”的 API 。通过这些 API ，我们能够定义出描述各种各样业务逻辑的复合事件模式。

所以接下来，我们就来看看 Flink CEP 都有哪些最常用的 API 吧。

begin(#name)

首先是**begin(#name)**，它用于定义一个 CEP 模式的开始。比如，当我们准备创建一个新的模式时，就可以用 begin(#name) 来创建。示例如下：

```
Pattern<Event, ?> startPattern = Pattern.<Event>begin("start");
```

在上面的代码中，我们用 begin("start") 创建了一个名字为 start 的模式。需要注意的是，此时创建的模式只有个名字，由于还没有给它设置任何匹配条件，所以它能够匹配任意事件。之后通过 where 等方法给它设置上条件时，它就可以按照设置的条件选出特定匹配的事件了。

next(#name)

然后是 next(#name)，它用于指定接下来的事件必须匹配的模式。比如，如果我们希望在前面的 start 模式后追加一个新模式，就可以用 next(#name) 来追加。示例如下：

```
Pattern<Event, ?> nextPattern = startPattern.next("next");
```

在上面的代码中，我们用 next("next") 在 start 之后新增了一个名字为 next 的模式。这样，当使用 nextPattern 模式进行匹配时，就必须要先匹配上名为 start 的模式，然后再匹配上名为 next 的模式，这样才能完整匹配上 nextPattern 模式。

另外需要注意的是，使用 next 指定接下来事件的匹配模式时，匹配的事件必须是紧接着前面的事件，中间不能有其他事件存在。我们可以用下面的图 1 说明下。

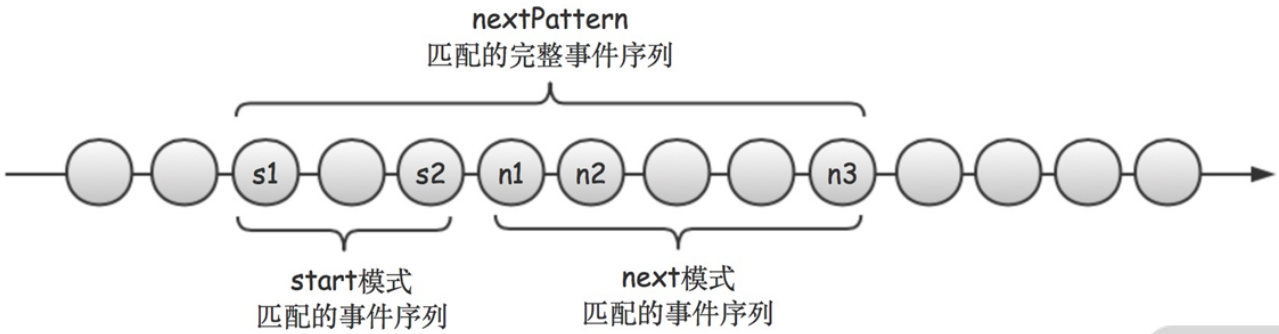


图 1 next 模式

在上面的图 1 中，名为 start 的模式匹配上了 s1 + s2 事件序列，名为 next 的模式匹配上了 n1 + n2 + n3 事件序列。可以看到，n1 + n2 + n3 事件序列和 s1 + s2 事件序列，这两者是紧接着的，它们之间没有其他事件存在。

followedBy(#name)

接着是 followedBy(#name)，它用于指定跟随其后的事件匹配模式，功能与 next 类似，但是中间可以有其他事件存在。比如当我们需要在前面的 start 模式后跟随一个新模式，就可以用 followedBy 。示例如下：

```
Pattern<Event, ?> followedByPattern = start.followedBy("followed");
```

在上面的代码中，我们用 followedBy("followed") 在 start 之后跟随了一个名字为 followed 的新模式。这样，当使用 nextPattern 模式进行匹配时，就必须要先匹配上名为 start 的模式，然后再匹配上名为 followed 的模式，这样才能够完整地匹配上 followedByPattern 模式。我们可以用下面的图 2 来帮助理解。

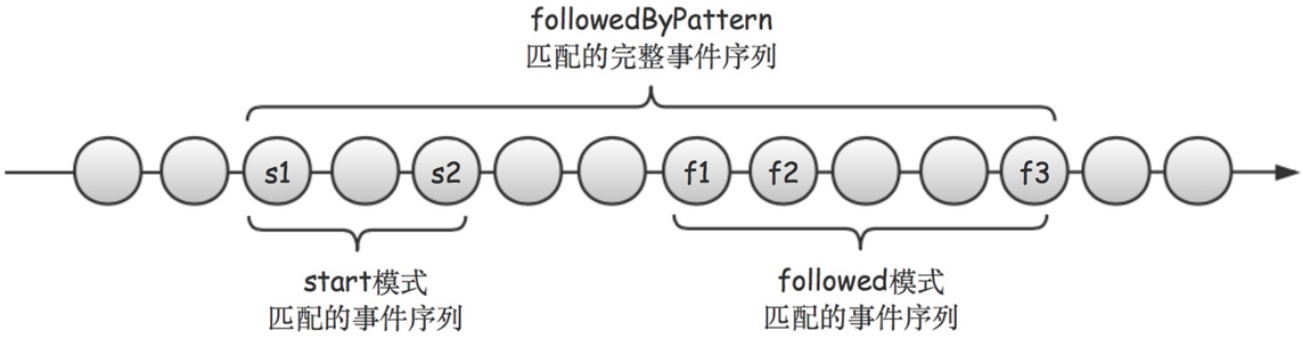


图 2 followedBy 模式

在上面的图 2 中，名为 start 的模式匹配上了 s1 + s2 事件序列，名为 followed 的模式匹配上了 f1 + f2 + f3 事件序列。可以看到，f1 + f2 + f3 事件序列和 s1 + s2 事件序列，这两者之间是可以有其他事件存在的。

notNext(#name)

再接着是 notNext(#name)。它用于指定接下来一个事件匹配的反模式。同样注意必须是紧接着前面的事件，中间不能有其他事件存在。比如，如果你不想匹配 start 模式之后还存在一个 notNext 模式的事件序列的话，可以用下面的示例代码：

```
Pattern<Event, ?> notNextPattern = start.notNext("notNext");
```

我们可以进一步用下面的图 3 辅助理解。

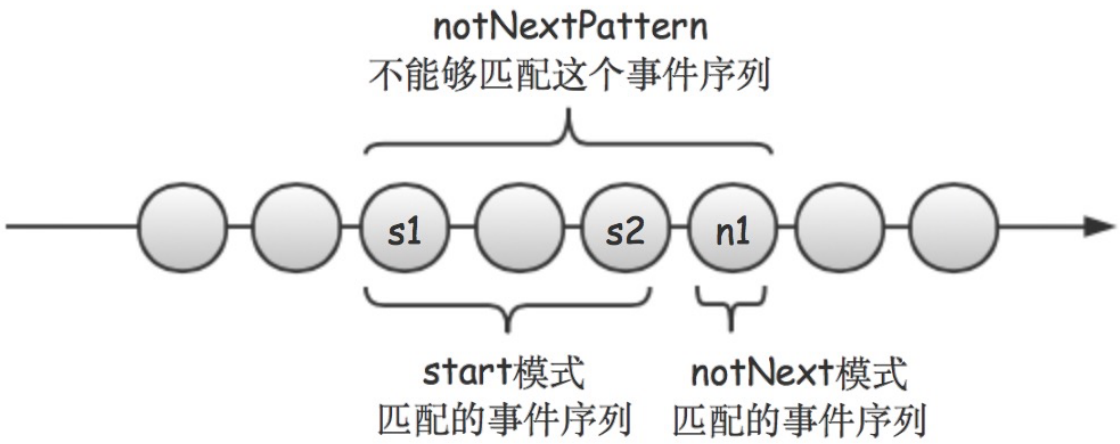


图 3 notNext 模式

在上面的图 3 中，名为 start 的模式匹配上了 s1 + s2 事件序列，名为 notNext 的模式匹配上了 n1 事件序列。但由于我们是使用的 notNext 这种反模式，所以 notNextPattern 模式就不能匹配 s1 + s2 + n1 事件序列了。但是，如果图 3 中没有 n1 事件的话，notNextPattern 模式就能够匹配 s1 + s2 事件序列。

notFollowedBy(#name)

然后是 notFollowedBy(#name) 。它用于指定跟随其后的事件匹配的反模式，与 notNext 类似，但中间可以有其他事件存在。比如，如果你不想匹配 start 模式之后还存在一个 notNext 模式的事件序列的话，可以用下面的代码：

```
Pattern<Event, ?> notFollowedByPattern = start.notFollowedBy("notFollowedBy");
```

同样，我们进一步用下面的图 4 帮助理解。

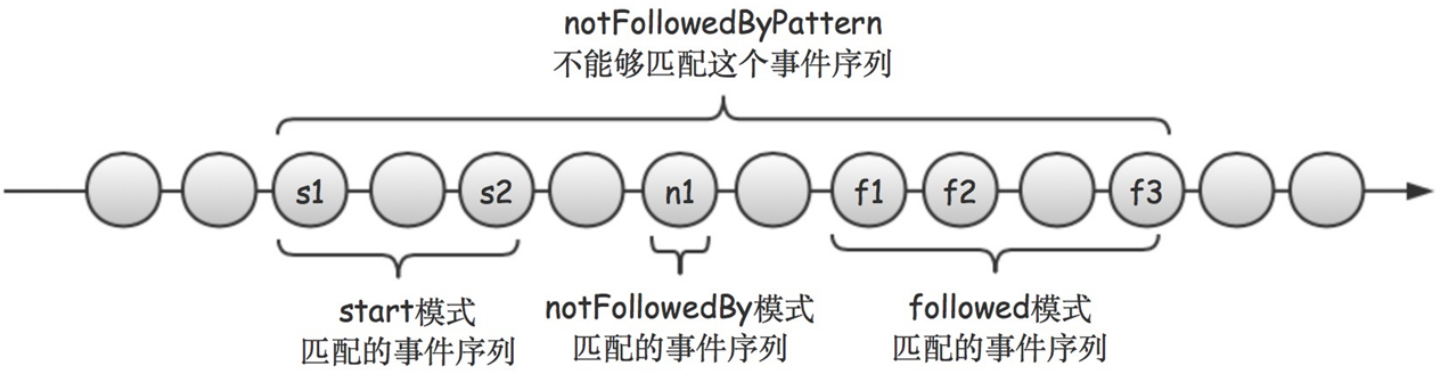


图 4 notFollowedBy 模式

在上面的图 4 中，名为 start 的模式匹配上了 s1 + s2 事件序列，名为 notFollowedBy 的模式匹配上了 n1 事件序列，名为 followedBy 的模式匹配上了 f1 + f2 + f3 事件序列。但由于我们使用的是 notFollowedBy 这种反模式，所以 notFollowedByPattern 模式就不能够匹配 s1 + s2 + n1 + f1 + f2 + f3 事件序列了。但是，如果图 4 中没有 n1 事件的话，notFollowedByPattern 模式就能够匹配 s1 + s2 + f1 + f2 + f3 事件序列。

within(#time)

再然后是 within(#time)。它用于指定模式匹配的事件必须是在特定的时间内完成，并且过期不候。比如，如果你想指定必须是在 10 秒钟之内，完成一个模式的匹配，那么可以用下面的代码：

```
pattern.within(Time.seconds(10));
```

within 是一个非常重要的方法，我们经常需要用它来指定模式超时的时间。毕竟，在 Flink CEP 内部保存事件序列的完成状态，是需要占用存储空间的。如果有太多匹配了一半模式，而另外一半模式迟迟不能匹配上，甚至永远匹配不上的话，那这些占用的存储空间永远得不到释放，最终就有可能将存储空间占满了。

where(condition)

接着是 where(condition)。它用于指定当前模式的条件。也就是说，如果你要想匹配该模式，就必须满足 condition 指定的条件。比如下面的示例代码：

```
pattern.where(new SimpleCondition<JSONObject>() {
    @Override
    public boolean filter(JSONObject value) throws Exception {
        return value.getBoolean("门当户对");
    }
});
```

在上面的代码中，我用 where 方法指定了一个匹配条件，也就是"门当户对"字段必须为 true 。这样，当事件中存在"门当户对"字段，并且它的值为 true 时，就能够匹配上 pattern 模式了。

times()

接下来是三个同类型的方法，也就是 oneOrMore()、timesOrMore(#times) 和 times(#ofTimes)。其中：

- oneOrMore 用于指定的条件必须至少匹配 1 次。
- timesOrMore 用于指定的条件必须至少匹配 #times 次。
- times 则用于指定的条件必须精确匹配 #times 次。

我们以 times 详细讲解下。比如，如果有个推荐系统的推荐模式是“在 10 分钟之内点击了 3 次同类商品”，那么可以用下面的代码来实现：

```
clickPattern.times(3).within(Time.seconds(600));
```

在上面的代码中，clickPattern 是匹配点击事件的模式，我们用 times(3) 指定了需要匹配 3 次，也就是要点击 3 次，并用 Time.seconds(600) 指定是在 10 分钟内完成 3 次点击才行。

until(condition)

接下来是 until(condition)。它用于指定一个循环模式的结束条件，并且只能用于 oneOrMore 和 timesOrMore 这两个循环模式之后。比如，如果你想爱一个人到海枯石烂、天荒地老，就可以用下面的代码。

```
lovePattern.oneOrMore().until(new SimpleCondition<JSONObject>() {
    @Override
    public boolean filter(JSONObject value) throws Exception {
        return value.getBoolean("海枯石烂") && value.getBoolean("天荒地老");
    }
})
```

在上面的代码中，lovePattern 是你表达对心上人爱意的事件模式，然后用 oneOrMore 不止一次地对他或她表达爱意。最后在 until 的条件中，只有当事件中的"海枯石烂"和"天荒地老"这两个字段都为 true 时，你长期表达爱意的模式才结束。

需要说明下的是，早期版本的 Flink CEP 中， until 只能用在 oneOrMore 上，不过最新版中until 也能够用在 timesOrMore 上了。但官方文档 对于这点好像并没有更新。

subtype(subClass)

最后是 subtype(subClass)。它用于指定当前模式匹配的事件类型，只有属于 subClass 类或者它的子类的事件才能够匹配当前模式。比如，你如果只想匹配“苹果”，而不想匹配其他类型的水果，就可以用下面的代码。

```
pattern.subtype(Apple.class);
```

subtype 也是我们经常用到的方法，毕竟在 Java 这种面向对象的语言中，类和继承是永恒的话题。特别是当流数据中有多种不同类型的事件时，可以用 subtype 将它们轻易地区分开。比如，如果流里还有“橘子”“香蕉”等，那上面匹配“苹果”的代码，就会将“橘子”“香蕉”忽略掉。

最后需要说明的是，这里只是介绍了部分常用的 API。实际上，Flink CEP 还提供了很多用于实现其他匹配模式的 API ，建议你去 Flink 官方文档 了解下其他的 API 。

接下来，我们就通过一个具体的实例，来看下 CEP 技术究竟怎样使用吧！

实用 Flink CEP 实现仓库环境温度监控

下面我们以仓库环境温度监控的例子，来演示 Flink CEP 在实际场景中的运用。

假设现在我们收到公司老板的需求，需要监控仓库的环境温度，来及时发现和避免火灾。我们使用的温度传感器每秒钟上报一次事件到基于 Flink 的实时流计算系统。

于是，我们设定告警规则如下，当 15 秒内两次监控温度超过阈值时发出预警，当 30 秒内产生两次预警事件，且第二次预警温度比第一次预警温度高时，就发出严重告警。

所以，我们先定义“15 秒内两次监控温度超过阈值”的模式。具体如下：

```
DataStream<JSONObject> temperatureStream = env
        .addSource(new PeriodicSourceFunction())
        .assignTimestampsAndWatermarks(new EventTimestampPeriodicWatermarks())
        .setParallelism(1);
Pattern<JSONObject, JSONObject> alarmPattern = Pattern.<JSONObject>begin("alarm")
        .where(new SimpleCondition<JSONObject>() {
            @Override
            public boolean filter(JSONObject value) throws Exception {
                return value.getDouble("temperature") > 100.0d;
            }
        })
        .times(2)
        .within(Time.seconds(15));
```

在上面的代码中，我们用 begin 定义一个模式 alarm，再用 where 指定了我们关注的是温度高于 100 摄氏度的事件。然后用 times 配合 within，指定高温事件在 15 秒内发生两次才发出预警。

然后，我们将预警模式安装到温度事件流上。具体如下：

```
DataStream<JSONObject> alarmStream = CEP.pattern(temperatureStream, alarmPattern)
        .select(new PatternSelectFunction<JSONObject, JSONObject>() {
            @Override
            public JSONObject select(Map<String, List<JSONObject>> pattern) throws Exception {
                return pattern.get("alarm").stream()
                        .max(Comparator.comparingDouble(o -> o.getLongValue("temperature")))
                        .orElseThrow(() -> new IllegalStateException("should contains 2 events, b
            }
        }).setParallelism(1);
```

在上面的代码中，我们将预警模式 alarmPattern 安装到温度事件流 temperatureStream 上。当温度事件流上有匹配到预警模式的事件时，就会发出一个预警事件，这是用 select 函数完成的。在 select 函数中，指定了发出的预警事件，是两个高温事件中，温度更高的那个事件。

接下来，还需要定义严重告警模式。具体如下：

```
Pattern<JSONObject, JSONObject> criticalPattern = Pattern.<JSONObject>begin("critical")
        .times(2)
        .within(Time.seconds(30));
```

与预警模式的定义类似，在上面的代码中，我们定义了严重告警模式，即“在 30 秒内发生两次”。

最后，我们再将告警模式安装在告警事件流上。具体如下：

```
DataStream<JSONObject> criticalStream = CEP.pattern(alarmStream, criticalPattern)
        .flatSelect(new PatternFlatSelectFunction<JSONObject, JSONObject>() {
            @Override
            public void flatSelect(Map<String, List<JSONObject>> pattern, Collector<JSONObject> o
                List<JSONObject> critical = pattern.get("critical");
                JSONObject first = critical.get(0);
                JSONObject second = critical.get(1);
                if (first.getLongValue("temperature") <
                        second.getLongValue("temperature")) {
                    JSONObject jsonObject = new JSONObject();
                    jsonObject.putAll(second);
                    out.collect(jsonObject);
                }
            }
        }).setParallelism(1);
```

在上面的代码中，这次我们的告警模式不再是安装在温度事件流，而是安装在预警事件流上。当预警事件流中，有事件匹配上告警模式，也就是在 30 秒内发生两次预警，并且第二次预警温度比第一次预警温度高时，就触发告警。从而提醒仓管人员，仓库温度过高，可能是要发生火灾了，需要立即采取防火措施！

这样，一个关于仓库环境温度监控的 CEP 应用就实现了。看，这是不是一个非常有用的功能！

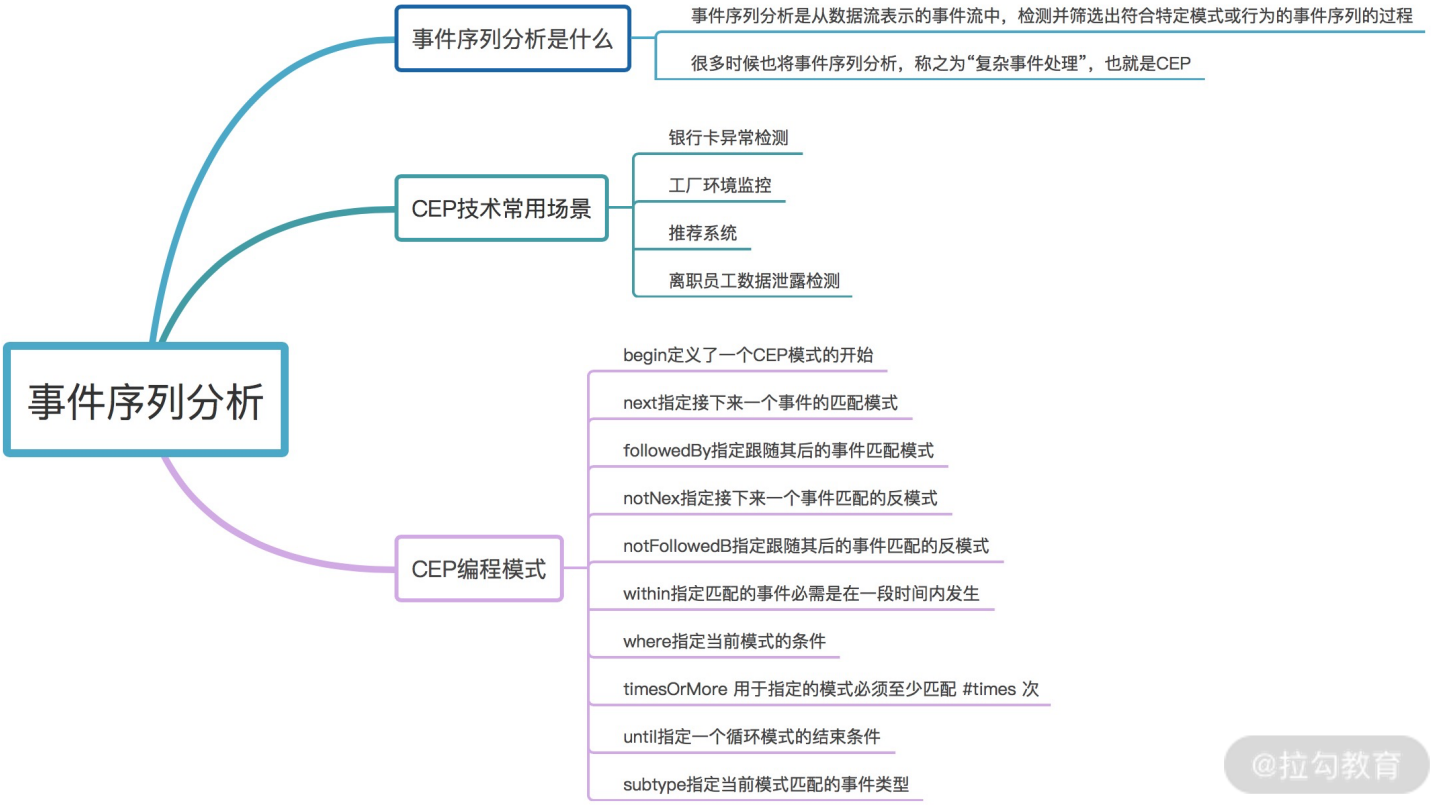
小结

总的来说，CEP 技术是一种帮助我们从业务数据流中，通过分析事件之间的关联关系，挖掘出有价值的行为模式或商业模式的过程。它的使用场景非常丰富，并且也是一种非常有趣的技术。

目前 Flink CEP 对匹配模式的支持是非常丰富的，并且它天然支持分布式流数据处理。所以，如果你有业务需要使用到 CEP 技术，或者对这方面的内容很感兴趣的话，可以考虑直接从 Flink CEP 入手。

那么在你的工作或生活中，有哪些场景是适合使用 CEP 技术呢？如果是使用 Flink CEP 来实现的话，你会怎么做呢？可以将你的问题和想法写在留言区。

下面是本课时的知识脑图，以便于你理解。


事件序列分析
事件序列分析是什么
事件序列分析是从数据流表示的事件流中，检测并筛选出符合特定模式或行为的事件序列的过程
很多时候也将事件序列分析，称之为“复杂事件处理”，也就是CEP
CEP技术常用场景
银行卡异常检测
工厂环境监控
推荐系统
离职员工数据泄露检测
CEP编程模式
begin定义了一个CEP模式的开始
next指定接下来一个事件的匹配模式
followedBy指定跟随其后的事件匹配模式
notNex指定接下来一个事件匹配的反模式
notFollowedB指定跟随其后的事件匹配的反模式
within指定匹配的事件必需是在一段时间内发生
where指定当前模式的条件
timesOrMore 用于指定的模式必须至少匹配 #times 次
until指定一个循环模式的结束条件
subtype指定当前模式匹配的事件类型
@拉勾教育