# Flink-复杂事件（CEP）

## 什么是复杂事件CEP?

一个或多个由简单事件构成的事件流通过一定的规则匹配，然后输出用户想得到的数据，满足规则的复杂事件。

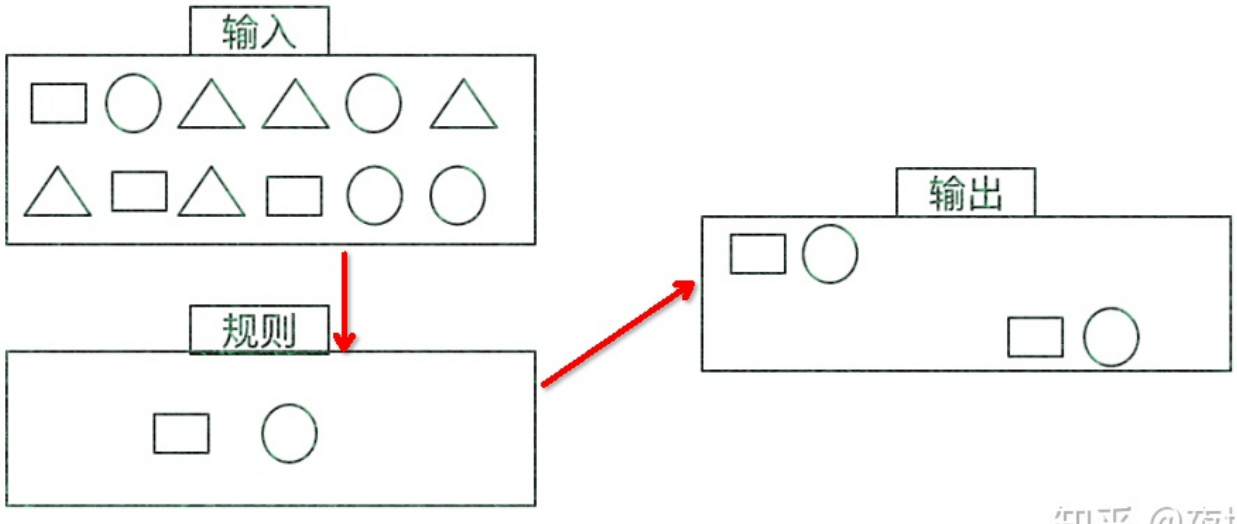
## 特征：

目标：从有序的简单事件流中发现一些高阶特征

输入：一个或多个由简单事件构成的事件流

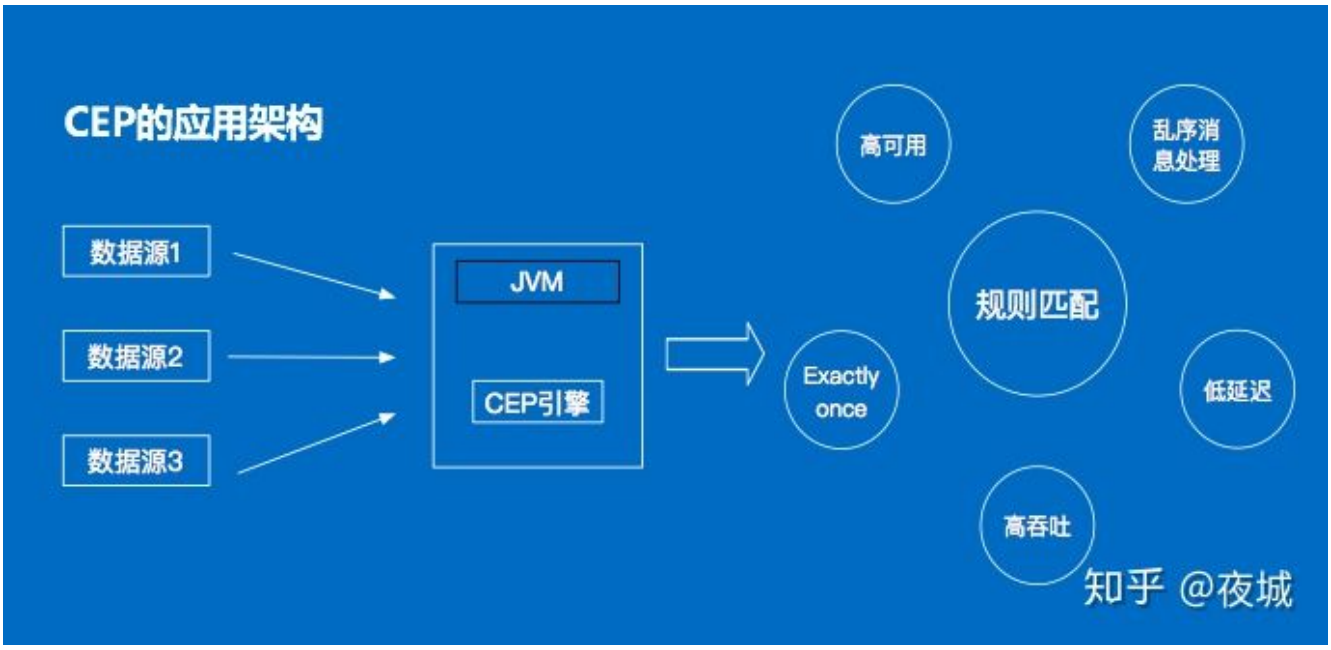
处理：识别简单事件之间的内在联系，多个符合一定规则的简单事件构成复杂事件

输出：满足规则的复杂事件





## CEP架构

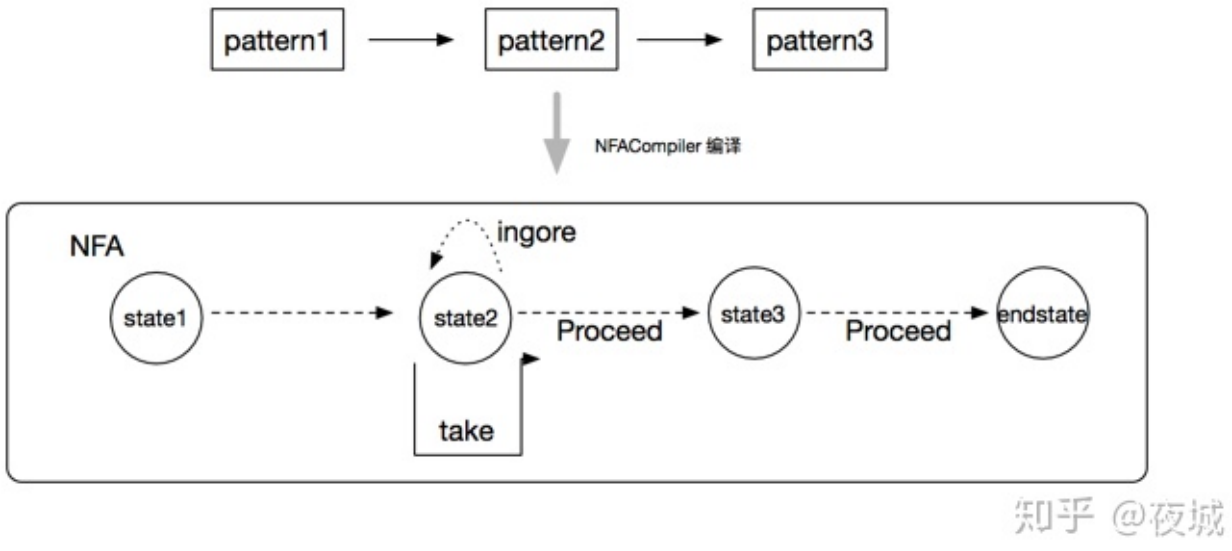



## CEP-NFA

## CEP-NFA是什么?

Flink 的每个模式包含多个状态，模式匹配的过程就是状态转换的过程，每个状态(state)可以理解成由Pattern构成，为了从当前的状态转换成下一个状态，用户可以在Pattern上指定条件，用于状态的过滤和转换。

实际上Flink CEP 首先需要用户创建定义一个个pattern，然后通过链表将由前后逻辑关系的pattern串在一起，构成模式匹配的逻辑表达。然后需要用户利用NFACompiler，将模式进行分拆，创建出NFA(非确定有限自动机)对象，NFA包含了该次模式匹配的各个状态和状态间转换的表达式。整个示意图就像如下：



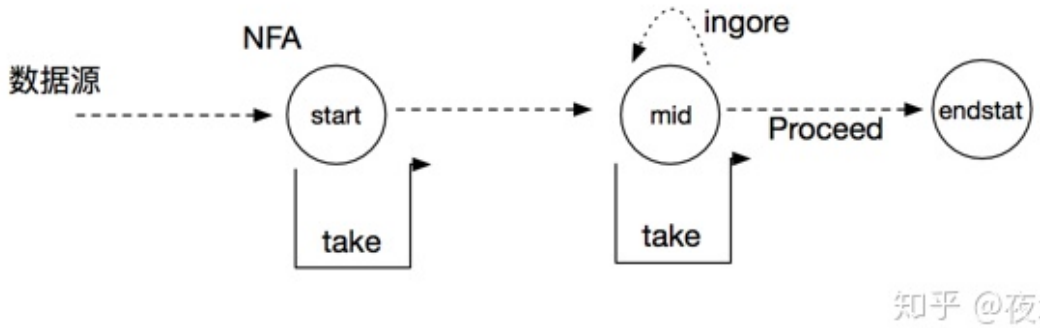
## CEP-三种状态迁移边

Take: 表示事件匹配成功，将当前状态更新到新状态，并前进到“下一个”状态；
Procceed: 当事件来到的时候，当前状态不发生变化，在状态转换图中事件直接“前进”到下一个目标状态；
IGNORE: 当事件来到的时候，如果匹配不成功，忽略当前事件，当前状态不发生任何变化。

为了更好的理解上述概念，我们利用下面代码，构建一个nfa：

```
//构建链接
patterns Pattern pattern = Pattern.begin("start").where(new SimpleCondition() {
 private static final long serialVersionUID = 5726188262756267490L;
@Override public boolean filter(Event value) throws Exception {
return value.getName().equals("c");
            }
        }).followedBy("middle").where(new SimpleCondition() {
 private static final long serialVersionUID = 5726188262756267490L;
@Override public boolean filter(Event value) throws Exception {
return value.getName().equals("a");
            }
        }).optional();
//创建nfa NFA nfa = NFACompiler.compile(pattern, Event.createTypeSerializer(), fa
```

例如构建的nfa的状态示意图如下所示：




此时如果我们加入数据源如下，理论上的输出结果应该是[event1] 和 [event1，event4]。

```
Event event0 = new Event(40, "x", 1.0);
Event event1 = new Event(40, "c", 1.0);
Event event2 = new Event(42, "b", 2.0);
Event event3 = new Event(43, "b", 2.0);
Event event4 = new Event(44, "a", 2.0);
Event event5 = new Event(45, "b", 5.0);
```

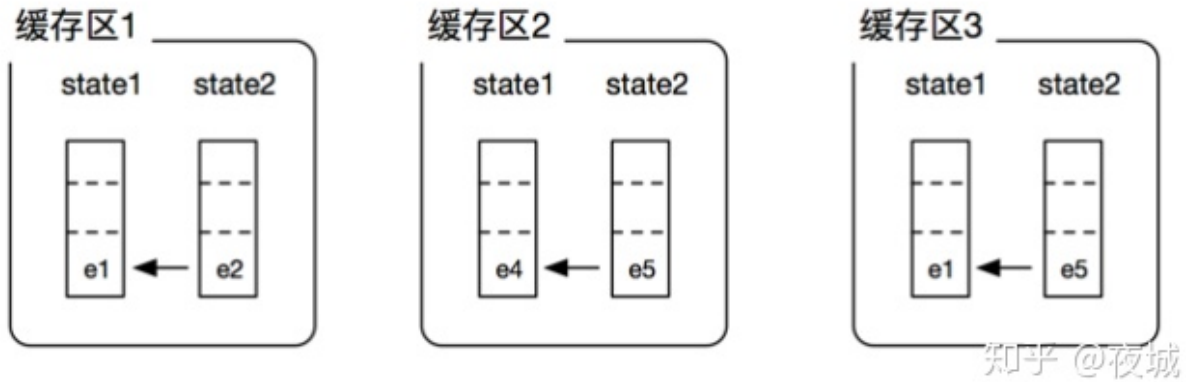
下面我们来分析下，当第一条消息event0来的时候，由于start状态只有Take状态迁移边，这时event0匹配失败，消息被丢失，start状态不发生任何变化；当第二条消息event1来的时候，匹配成功，这时用event1更新start当前状态，并且进入下一个状态，既mid状态。而这是我们发现mid状态存在Proceed状态迁移边，以为着事件来临时，可以直接进入下一个状态，这里就是endstat状态，说明匹配结束，存在第一个匹配结果[event1]；当第三条消息event2来临时，由于之前我们已经进入了mid状态，所以nfa会让我们先匹配mid的条件，匹配失败，由于mid状态存在Ingore状态迁移边，所以当前mid状态不发生变化，event2继续往回匹配start的条件，匹配失败，这时event2被丢弃；同样的event3也不会影响nfa的所有状态，被丢弃。当第五条消息event4来临时，匹配mid的条件成功，更新当前mid状态，并且进入“下一个状态”，那就是endstat状态，说明匹配结束，存在第二个匹配结果[event1, event4]。

## CEP 共享缓存SharedBuffer

在引入SharedBuffer概念之前，我们先把上图的例子改一下，将原先pattern1 和 pattern2的连接关系由next，改成followedByAny。

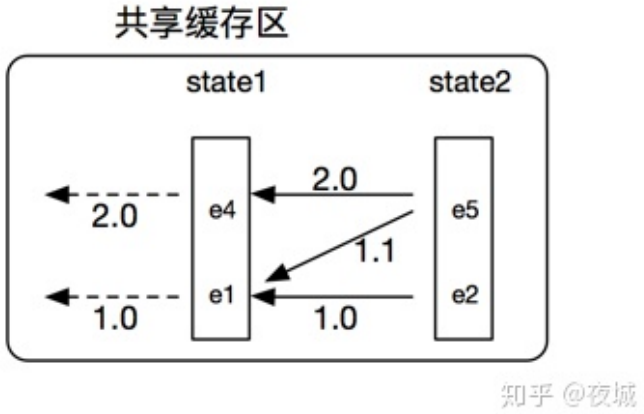
```
Pattern pattern2 = pattern1.followedByAny("pattern2").where(new SimpleCondition()
private static final long serialVersionUID = 5726188262756267490L;
@Override public boolean filter(Event value) throws Exception {
 return value.getName().equals("b");
            }
        });
```

followedByAny是非严格的匹配连接关系。表示前一个pattern匹配的事件和后面一个pattern的事件间可以间隔多个任意元素。所以上述的例子输出结果是[event1, event2]、[event4, event5]和[event1, event5]。当匹配event1成功后，由于event2还没到来，需要将event1保存到state1，这样每个状态需要缓冲堆栈来保存匹配成功的事件，我们把各个状态的对应缓冲堆栈集称之为缓冲区。由于上述例子有三种输出，理论上我们需要创建三个独立的缓冲区。




做三个独立的缓冲区实现上是没有问题，但是我们发现缓冲区3状态stat1的堆栈和缓冲区stat1的堆栈是一样的，我们完全没有必要分别占用内存。而且在实际的模式匹配场景下，每个缓冲区独立维护的堆栈中可能会有大量的数据重叠。随着流事件的不断流入，为每个匹配结果独立维护缓

存区占用内存会越来越大。所以Flink CEP 提出了共享缓存区的概念(SharedBuffer),就是用一个共享的缓存区来表示上面三个缓存区。




在共享缓冲区实现里头，Flink CEP 设计了一个带版本的共享缓冲区。它会给每一次匹配分配一个版本号并使用该版本号来标记在这次匹配中的所有指针。但这里又会面临另一个问题：无法为某次匹配预分配版本号，因为任何非确定性的状态都能派生出新的匹配。而解决这一问题的技术是采用杜威十进制分类法[^1]来编码版本号，它以(.)?(1≤j≤t)的形式动态增长，这里t关联着当前状态。直观地说，它表示这次运行从状态开始被初始化然后到达状态，并从中分割出的实例，这被称之为祖先运行。这种版本号编码技术也保证一个运行的版本号v跟它的祖先运行的版本号兼容。具体而言也就是说：（1）v包含了v’作为前缀或者（2）v与v’仅最后一个数值不同，而对于版本v而言要大于版本v’。根据对这段话的理解，上述共享区从e5往回查找数据，可以达到两条路径分别是[e4,e5]和[e1, e5]。

## CEP-SQL参数介绍

### 语法：

```
SELECT [ ALL | DISTINCT ]
{ * | projectItem [, projectItem ]* }
FROM tableExpression
[MATCH_RECOGNIZE (
[PARTITION BY {partitionItem [, partitionItem]*}]
[ORDER BY {orderItem [, orderItem]*}]
[MEASURES {measureItem AS col [, measureItem AS col]*}]
[ONE ROW PER MATCH|ALL ROWS PER MATCH|ONE ROW PER MATCH WITH TIMEOUT ROWS|ALL ROW
[AFTER MATCH SKIP]
PATTERN (patternVariable[quantifier] [ patternVariable[quantifier]]*) WITHIN inte
```

```
DEFINE {patternVariable AS patternDefinationExpression [, patternVariable AS patt
)];
```

参数说明:

`PARTITION BY` 指定分区的列，可选项
`ORDER BY` 可以指定多列，但是必须以event time列或者process time列作为排序的首列，其定义方式参考文档，可选项
`MEASURES` 定义如何根据匹配成功的输入事件构造输出事件
`ONE ROW PER MATCH` 对于每一次成功的匹配，只会产生一个输出事件;
`ONE ROW PER MATCH WITH TIMEOUT ROWS` 除了匹配成功的时候产生输出外，超时的时候也会产生输出，超时时间由 `PATTERN` 语句中的 `WITHIN` 语句定义;
`ALL ROW PER MATCH` 对于每一次成功的匹配，对应于每一个输入事件，都会产生一个输出事件;
`ALL ROW PER MATCH WITH TIMEOUT ROWS` 除了匹配成功的时候产生输出外，超时的时候也会产生输出，超时时间由 `PATTERN` 语句中的 `WITHIN` 语句定义；可选项，缺省为 `ONE ROW PER MATCH`
`AFTER MATCH SKIP TO NEXT ROW` 匹配成功之后，从匹配成功的事件序列中的第一个事件的下一个事件开始进行下一次匹配
`AFTER MATCH SKIP PAST LAST ROW` 匹配成功之后，从匹配成功的事件序列中的最后一个事件的下一个事件开始进行下一次匹配
`AFTER MATCH SKIP TO FIRST patternItem` 匹配成功之后，从匹配成功的事件序列中第一个对应于patternItem的事件开始下一次匹配
`AFTER MATCH SKIP TO LAST patternItem` 匹配成功之后，从匹配成功的事件序列中最后一个对应于patternItem的事件开始下一次匹配
`PATTERN` 定义待识别的事件序列需要满足的规则，需要定义在()中，由一系列自定义的patternVariable构成：

patternVariable之间若以空格间隔，表示符合这两种patternVariable的事件紧挨着，中间不存在其他事件；

patternVariable之间若以->间隔，表示符合这两种patternVariable的事件之间可以存在其它事件。

quantifier用于指定符合patternVariable定义的事件的出现次数：

*： 0次或多次

+： 1次或多次

?： 0次或1次

{n}： n次

{n,}: 大于等于n次

{n, m}： 大于等于n次，小于等于m次

{,m}： 小于等于m次

默认为贪婪匹配，比如对于pattern: A -> B+，

输入: a b1, b2, b3,

输出为：a b1, a b1 b2, a b1 b2 b3。

可以在quantifier符号后面加？来表示非贪婪匹配:

*?
+?
{n}?
{n,}?
{n, m}?
{,m}?

此时对于上面例子中的pattern及输入，产生的输出为：a b1, a b2, a b1 b2, a b3, a b2 b3, a b1 b2 b3。

WITHIN 定义符合规则的事件序列的最大时间跨度
静态窗口
格式：INTERVAL ‘string’ timeUnit [ TO timeUnit ]
示例：INTERVAL ‘10’ SECOND, INTERVAL ‘45’ DAY, INTERVAL ‘10:20’ MINUTE TO SECOND, INTERVAL ‘10:20.10’ MINUTE TO SECOND, INTERVAL ‘10:20’ HOUR TO MINUTE, INTERVAL ‘1-5’ YEAR TO MONTH
动态窗口
格式： INTERVAL intervalExpression
示例： INTERVAL A.windowTime + 10，其中A为pattern定义中第一个patternVariable
在intervalExpression的定义中，可以使用pattern定义中出现过的patternVariable，当前只能使用第一个patternVariable，intervalExpression中可以使用UDF，intervalExpression的结果必须为long，单位为millisecond, 表示窗口的大小。

EMIT TIMEOUT 定义多个超时窗口，所有超时窗口不能超过WITHIN语句中定义的窗口，可选项

固定窗口：
格式：EMIT TIMEOUT (INTERVAL ‘string’ timeUnit [ TO timeUnit ], INTERVAL ‘string’ timeUnit [ TO timeUnit ], ...)
示例：EMIT TIMEOUT (INTERVAL ‘2’ HOUR, INTERVAL ‘6’ HOUR)
周期性窗口：
格式：EMIT TIMEOUT EVERY INTERVAL ‘string’ timeUnit [ TO timeUnit ]
示例：EMIT TIMEOUT EVERY INTERVAL ‘2’ HOUR
EMIT TIMEOUT语句必须与WITHIN语句及ONE ROW PER MATCH WITH TIMEOUT ROWS或者ALL ROW PER MATCH WITH TIMEOUT ROWS同时使用
EMIT TIMEOUT语句主要用在有多个超时指标需要计算的场景，如物流配送中可能需要同时统计订单超时2小时未配送、超时4小时未配送等

DEFINE 定义在PATTERN中出现的patternVariable的具体含义，若某个patternVariable在DEFINE中没有定义，则认为对于每一个事件，该patternVariable都成立

在MEASURES和DEFINE语句中，可以使用如下几个函数：

函数函数意义Row Pattern Column References形式为: patternVariable.col，表示访问patternVariable所对应的事件的指定的列PREV只能用在DEFINE语句中，一般与 `Row Pattern Column References` 合用，用于访问指定的pattern所对应的事件之前偏移指定的offset所对应的事件的指定的列；示例：对于DOWN AS DOWN.price < PREV(DOWN.price)，PREV(A.price)表示当前事件的前一个事件的price列的值；注意DOWN.price等价于PREV(DOWN.price, 0)，PREV(DOWN.price)等价于PREV(DOWN.price, 1)FIRST、LAST一般与 `Row Pattern Column References` 合用，用于访问指定的pattern所对应的事件序列中的指定偏移位置的事件； 示例：FIRST(A.price, 3)表示pattern A所对应的事件序列中的第3个事件LAST(A.price, 3)表示pattern A所对应的事件序列中的倒数第3个事件RUNNING、FINAL用于修饰聚合函数、FIRST或LAST函数，RUNNING表示从第一个匹配成功的事件到当前事件作为输入时，其所修饰的函数的结果，FINAL表示所有匹配成功的事件序列作为输入时，其所修饰的函数的结果。 DEFINE语句中，只能用RUNNING，且默认就是RUNNING。 在ONE ROW PER MATCH的情况下，MEASURES语句中的函数默认修饰为FINAL，且由于对于每一个匹配序列，只会有一个输出，所以此时FINAL与RUNNING的意义相同。 在ALL ROWS PER MATCH的情况下，MEASURES语句中的函数默认修饰为RUNNING。CLASSIFIER当前只能用于MEASURES中， 在ONE ROW PER MATCH的情况下，表示匹配序列中的最后一个事件所对应的patternVariable； 在ALL ROWS PER MATCH的情况下，因为对于匹配序列中的每一行，都会产生一个输出，所以CLASSIFIER表示当前输出所对应的输入的patternVariable

输出列：

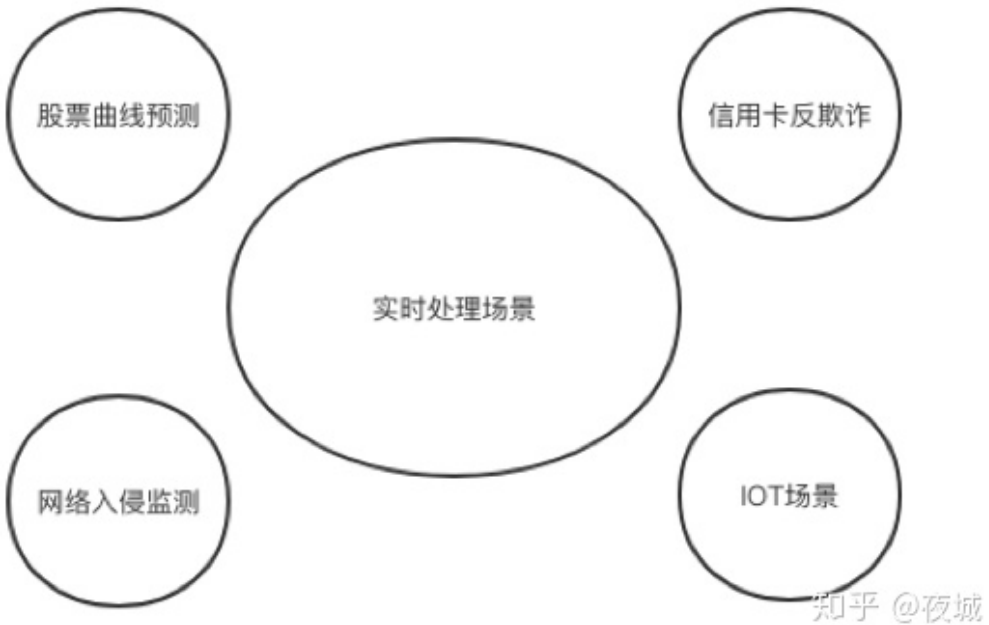
输出列ONE ROW PER MATCH包括 partition by中指定的列及measures中定义的列。 对于partition by中已经指定的列，在measures中没有必要再重复定义了ALL ROWS PER MATCH包括事件所有的列以及measures中新定义的列ONE ROW PER MATCH WITH TIMEOUT ROWS除了匹

配成功的时候产生输出外，超时的时候也会产生输出，超时时间由PATTERN语句中的WITHIN语句定义ALL ROWS PER MATCH WITH TIMEOUT ROWS除了匹配成功的时候产生输出外，超时的时候也会产生输出，超时时间由PATTERN语句中的WITHIN语句定义

备注： 1.定义pattern的时候，最好也定义WITHIN，否则可能会造成state越来越大 2.order by中定义的首列必须为event time列或者process time列，定义参考文档

## CEP应用场景

CEP的应用场景有很多股票曲线预测、网络入侵、物流订单追踪、电商订单、IOT场景等。




## CEP应用案例

案例描述：

当相同的card_id在十分钟内，从两个不同的location发生刷卡现象，就会触发报警机制，以便于监测信用卡盗刷等现象。

```
CREATE TABLE datahub_stream (
    `timestamp`               TIMESTAMP,
    card_id                   VARCHAR,
    location                  VARCHAR,
    `action`                  VARCHAR,
    WATERMARK wf FOR `timestamp` AS withOffset(`timestamp`, 1000)
) WITH (
    type = 'datahub'
    ...
);
CREATE TABLE rds_out (
    start_timestamp               TIMESTAMP,
    end_timestamp                 TIMESTAMP,
    card_id                       VARCHAR,
    event                         VARCHAR
) WITH (
    type= 'rds'
    ...
);

--案例描述
-- 当相同的card_id在十分钟内，从两个不同的location发生刷卡现象，就会触发报警机制，以便于监测信
-- 定义计算逻辑
insert into rds_out
select
`start_timestamp`,
`end_timestamp`,
card_id, `event`
from datahub_stream
MATCH_RECOGNIZE (
    PARTITION BY card_id   -- 按card_id分区，将相同卡号的数据分到同一个计算节点上
    ORDER BY `timestamp`   -- 在窗口内，对事件时间进行排序
    MEASURES               --定义如何根据匹配成功的输入事件构造输出事件
        e2.`action` as `event`,
        e1.`timestamp` as `start_timestamp`,   --第一次的事件事件为start_timestamp
        LAST(e2.`timestamp`) as `end_timestamp`--最新的事件事件为end_timestamp
    ONE ROW PER MATCH           --匹配成功输出一条
    AFTER MATCH SKIP TO NEXT ROW--匹配跳转到下一行后
    PATTERN (e1 e2+) WITHIN INTERVAL '10' MINUTE  -- 定义两个事件，e1/e2
    DEFINE                     --定义在PATTERN中出现的patternVariable的具体含义
        e1 as e1.action = 'Tom',    --事件一的action标记为Tom
        e2 as e2.action = 'Tom' and e2.location <> e1.location --事件二的action标记
);
```

## 测试数据

timestamp(TIMESTAMP)card_id(VARCHAR)location(VARCHAR)action(VARCHAR)

2018-04-13 12:00:00，1，WW，Tom

2018-04-13 12:10:00，1，WW2，Tom

2018-04-13 12:10:00，1，WW2Tom

2018-04-13 12:20:00，1，WW，Tom

## 测试结果

start_timestamp(TIMESTAMP)end_timestamp(TIMESTAMP)card_id(VARCHAR)event(VARCHAR)

2018-04-13 20:00:00.0，2018-04-13 20:05:00.01，Tom

2018-04-13 20:05:00.0，2018-04-13 20:10:00.01，Tom

发布于 2018-08-31