DolphinDB 白皮书

复杂事件处理引擎

内容

前言

流数据是一种持续实时生成且动态变化的时间序列数据，涵盖了金融交易、物联网（IoT）传感器采集、物流运营、零售订单等各类持续生成动态数据的场景。DolphinDB 之前版本中的普通流数据处理引擎只能处理单一事件类型或者简单的关联，不能解决更加复杂的业务场景。为此，DolphinDB 在流数据框架中增加了复杂事件处理（Complex Event Processing，下文简称 CEP）引擎，用于在大规模和复杂的实时数据流中提取信息、识别模式、进行实时分析和决策。它适用于金融交易监控、供应链实时优化、智能物联网、安全监控、市场营销等领域，能够处理和解决普通流数据处理无法满足的复杂需求。

本白皮书旨在介绍 DolphinDB 的 CEP 引擎框架、功能特性及应用场景。

第 1 章. 概述

本章主要介绍复杂事件处理引擎（CEP 引擎）相关概念，包括什么是复杂事件处理和复杂事件处理引擎和普通流计算引擎的区别。除此之外，本章简单介绍了 DolphinDB CEP 引擎的框架，具体细节将在后续章节展开。

1.1 基本概念

事件

事件是描述对象变化的属性值的集合。例如，下图显示了股票报价事件。每个股票报价都有许多属性，包括当前买价、当前卖价和当前交易量。在实际生产场景中，事件一般以数据流的方式存在并源源不断的产生。

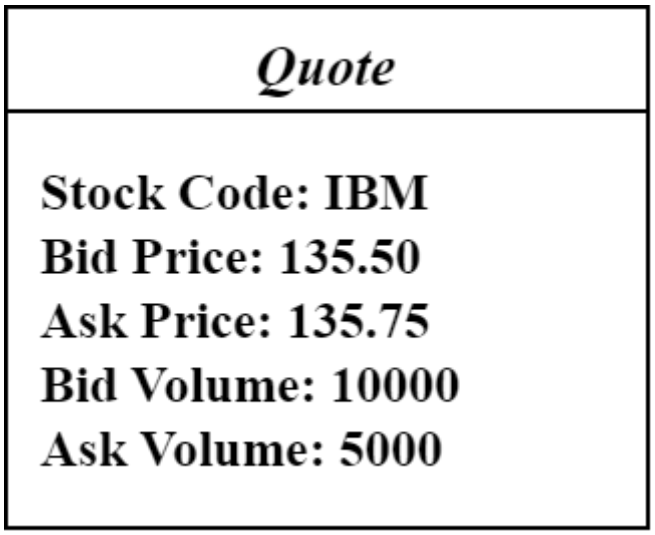

图 1-1 事件示例

复杂事件

通常，用户在事件流中除了查找并处理特定的事件外，还会对事件进行额外的限制，比如：

- 限制事件顺序：A 事件之后跟着事件 B。
- 限制事件属性值：A 的属性 A.a > 100。
- 限制时间：在5分钟内收到事件 B。

CEP

CEP 全称为 Complex Event Processing，中文意思为复杂事件处理。CEP 技术主要用于处理和分析实时事件流，以快速识别和响应关键的业务事件。它旨在从大量的实时事件流中提取符合指定规则的信息，根据事件的内容和事件发生的时间进行关联，并采取相应的行动。CEP 广泛应用于金融、物联网、电信、供应链等领域，以实现实时监控、异常检测、决策支持等功能。

下图清晰地展示了复杂事件处理特点：从大量的实时事件流中找到符合规则的特定事件，如在正方形代表的这类事件发生后再发生圆形代表的另一类事件。

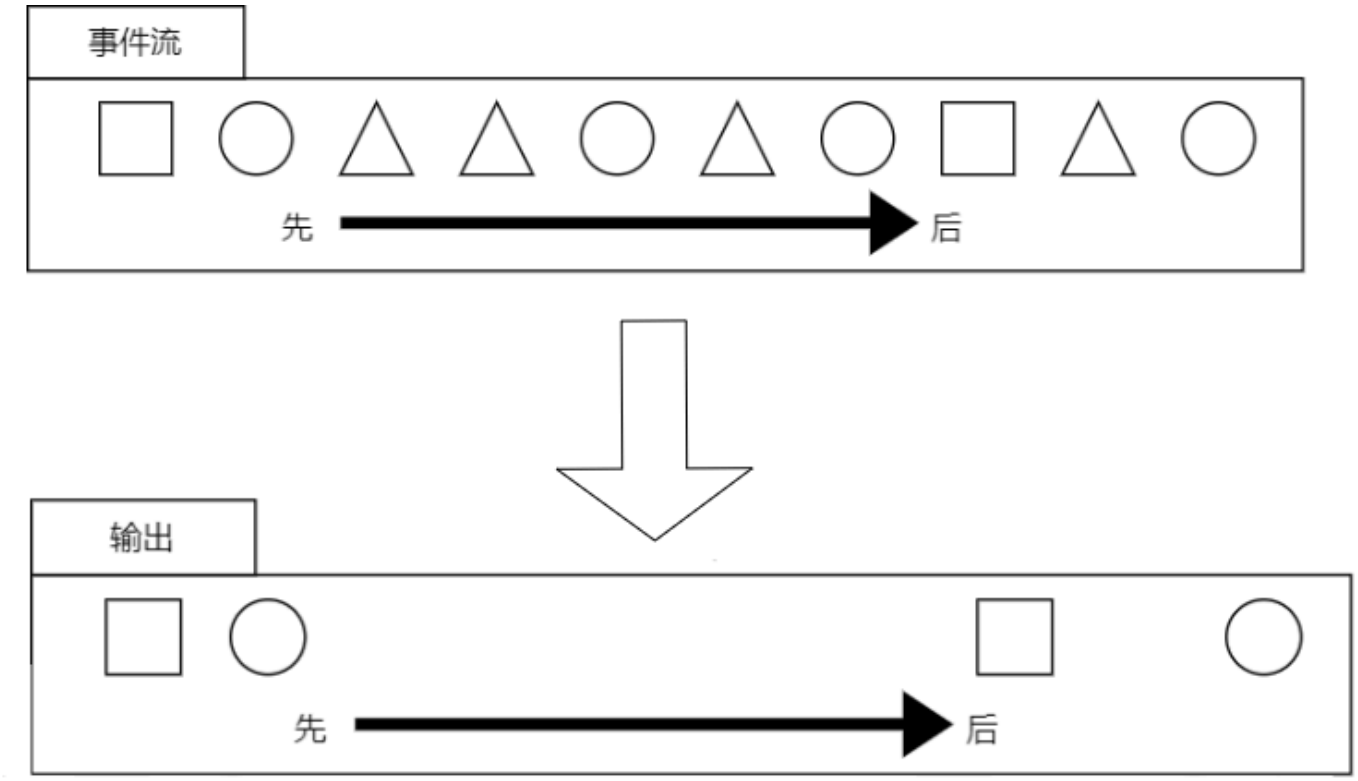

图 1-2 复杂事件处理示例

1.2 CEP 和普通流计算引擎的区别

- **数据结构**：CEP 引擎专门处理复杂事件。同一个数据源中的数据可以被定义为多种不同的事件。多个简单的事件组合成复杂事件。它具备识别事件间特定模式和关系的能力，并根据不同的事件执行相应的处理逻辑。普通流计算引擎中没有事件的概念。同一个数据源中的数据具有相同的数据结构，使用相同的处理逻辑。一个普通流计算引擎中不能处理具有不同结构的数据流，更不能识别数据流之间的关系。

- **目标**：复杂事件处理的目标是从实时数据流中提取指定的事件和模式，以便进行实时的分析、决策支持和响应。它强调在大规模、高速和复杂的数据流中发现和识别重要的事件和关联关系。普通流计算引擎的目标一般是对数据做高效的聚合、计算等，以满足具体的业务需求。

- **处理方式**：复杂事件处理采用事件驱动的方式进行处理，主要通过识别和匹配复杂事件模式、窗口操作和时序分析来提取有用的信息。它通常需要定义和管理一组模式，以筛选和处理数据流中的事件。普通流计算引擎应用同样的计算逻辑来处理所有接收到的数据。

总的来说，复杂事件流处理和普通流数据处理在数据结构、目标和处理方式上存在一些区别。复杂事件流处理专注于从实时数据流中提取复杂事件和模式，以支持实时的分析和决策。普通流计算引擎更加注重对某一数据结构的高性能处理与计算。有关 DolphinDB 流数据的相关内容请参阅《DolphinDB 流数据白皮书》。

1.3 CEP 引擎应用框架

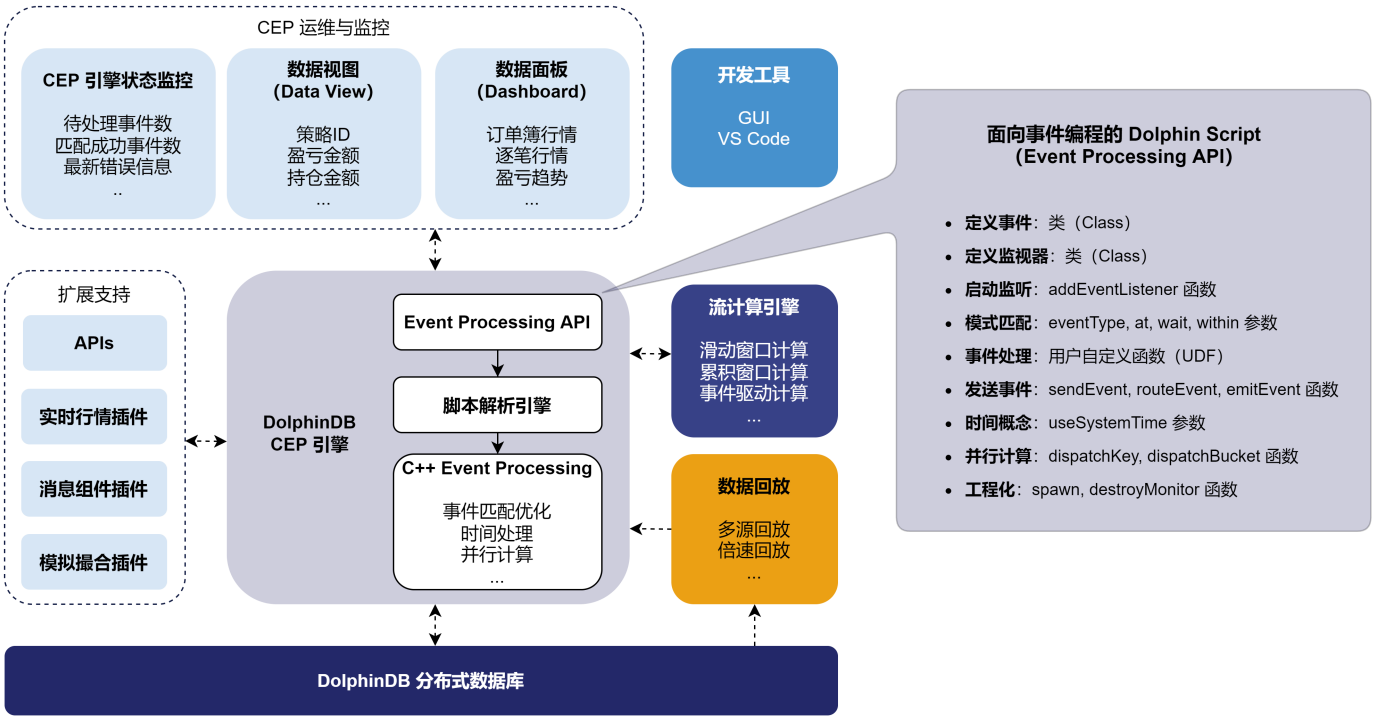

图 1-3 CEP 引擎框架

上图展示了 DolphinDB CEP 引擎应用框架，主要包含以下部分：

- DolphinDB CEP 引擎支持面向事件编程的 DolphinScript，提供了多种事件处理 API。
- DolphinDB CEP 引擎与 DolphinDB 时序数据库无缝集成，提供存储、计算一站式解决方案。
- DolphinDB CEP 引擎与流数据计算引擎结合，除了可以高效地定义和匹配复杂事件，低时延地触发相应的处理逻辑之外，还可以高效实现复杂因子与策略计算。
- DolphinDB CEP 引擎支持将 DolphinDB 数据库中的数据回放至 CEP 引擎中，实现策略回测与批流一体。
- 通过 DolphinDB API 或者插件，第三方客户端（如 Python 应用程序）或者其他第三方数据源（如 kafka）可以方便地将数据输入到 CEP 引擎或者订阅 CEP 引擎输出结果。
- 提供多种运维工具和插件，方便用户实时监控 CEP 引擎运行状态。

以下章节将着重讲解 CEP 引擎，即下图中间紫色部分。

第 2 章. 工作原理

本章主要讲解架构图中每一个组件的实现原理与功能。下图为 CEP 引擎架构图。

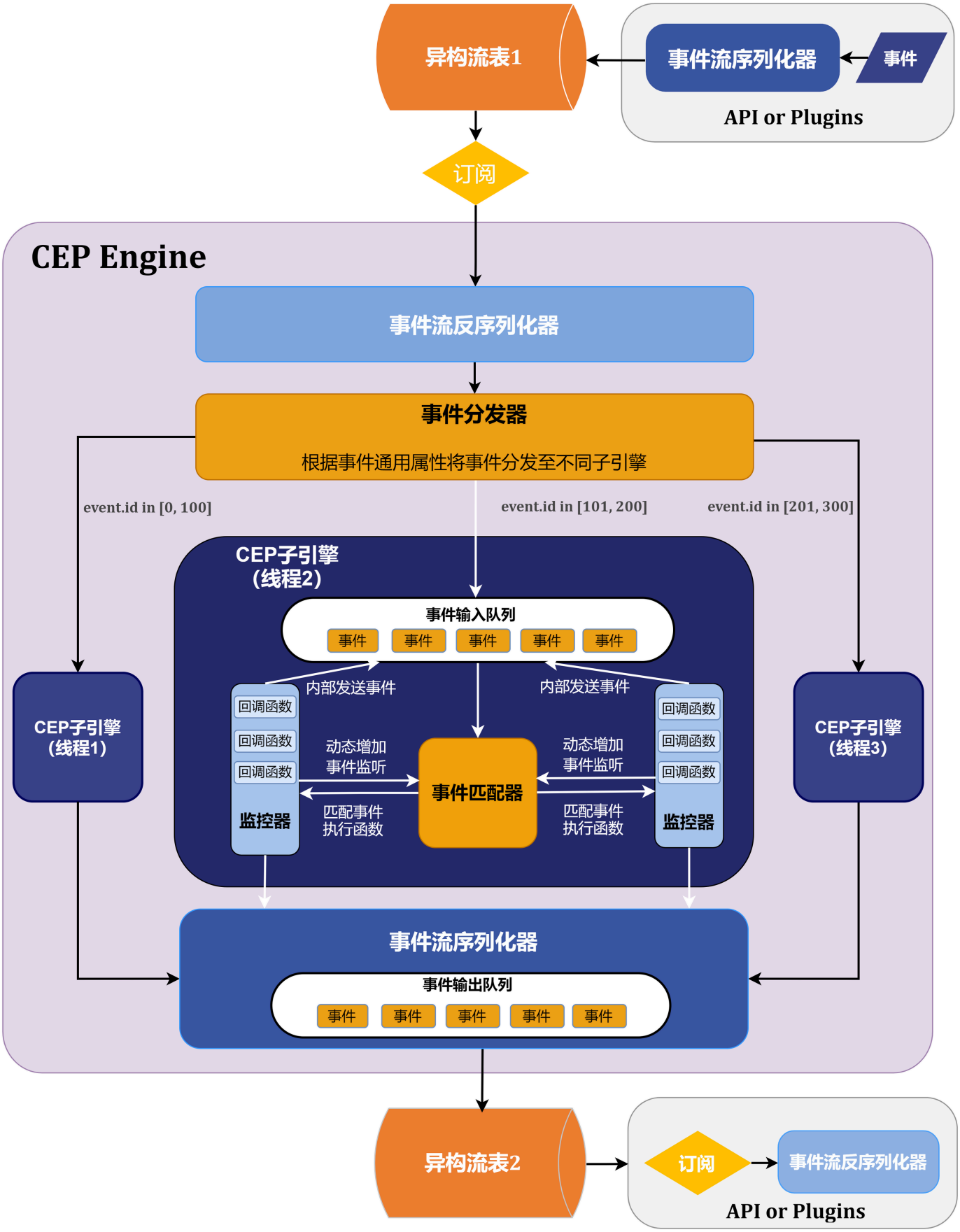
异构流表1
事件流序列化器
事件
API or Plugins
订阅
CEP Engine
事件流反序列化器
事件分发器
根据事件通用属性将事件分发至不同子引擎
event.id in [0, 100]
event.id in [101, 200]
event.id in [201, 300]
CEP子引擎
(线程1)
CEP子引擎
(线程2)
事件输入队列
事件
事件
事件
事件
事件
内部发送事件
内部发送事件
回调函数
回调函数
回调函数
监控器
动态增加
事件监听
匹配事件
执行函数
事件匹配器
动态增加
事件监听
匹配事件
执行函数
回调函数
回调函数
回调函数
监控器
CEP子引擎
(线程3)
事件流序列化器
事件输出队列
事件
事件
事件
事件
事件
异构流表2
订阅
事件流反序列化器
API or Plugins

图 2-1 CEP 引擎架构

2.1 事件流序列化器

事件流序列化器（Stream Event Serializer）负责将来自各种数据源的实时事件流序列化并写入 DolphinDB 异构流数据表，方便进行网络传输和数据持久化。DolphinDB Server 和 API 均支持了 StreamEventSerializer，其将各类事件序列化成同一形式的数据写入异构流数据表。

如下定义一个 StreamEventSerializer，将事件 `MarketData`、`Orders`、`Trades` 序列化到一个异构流表。

```
class MarketData{
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    def MarketData(m,c,p,q){
        market = m
        code = c
        price = p
        qty = q
    }
}

class Orders{
    trader :: STRING
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    def Orders(t, m,c,p,q){
        trader = t
        market = m
        code = c
        price = p
        qty = q
    }
}

class Trades{
    trader :: STRING
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    def Trades(t, m,c,p,q){
        trader = t
        market = m
        code = c
        price = p
        qty = q
```

```
    }
}

share streamTable(array(STRING, 0) as eventType, array(BLOB, 0) as blobs) as events
serializer = streamEventSerializer(name=`serOutput, eventSchema=[MarketData, Orders,
 Trades], outputTable=events)
```

2.2 事件流反序列器

CEP 引擎通过订阅异构流数据的方式实时获取最新的数据。事件流反序列化器（Stream Event Deserializer）将序列化的流事件数据转换回原始事件对象。CEP 引擎内部实现了 StreamEventDeserializer，因此无需额外定义反序列化器，引擎便会自动反序列化从订阅的异构流表中获取到的数据。

2.3 事件分发器

事件分发器（Event Dispatcher）根据事件中的特定属性将数据分发给不同的子引擎。这些子引擎之间相互独立，从而实现相同的事件处理逻辑进行分组并行处理，能够提升性能，降低延迟。分发方式包含以下几种：

- 按属性值分发：每个不同的属性值分配一个 CEP 子引擎。属性值相同的事件分发到同一子引擎。
- 按属性值哈希分发：指定子引擎数量，根据属性哈希值映射到不同的 hash bucket 中。

对于没有该属性2.4 CEP 子引擎的事件，将分发到所有子引擎。

2.4 CEP 子引擎

下面介绍 CEP 引擎中有关事件匹配的组件，分别是事件输入队列、事件匹配器、事件监听器和监视器。其中，CEP 每个子引擎都单独维护上述组件。

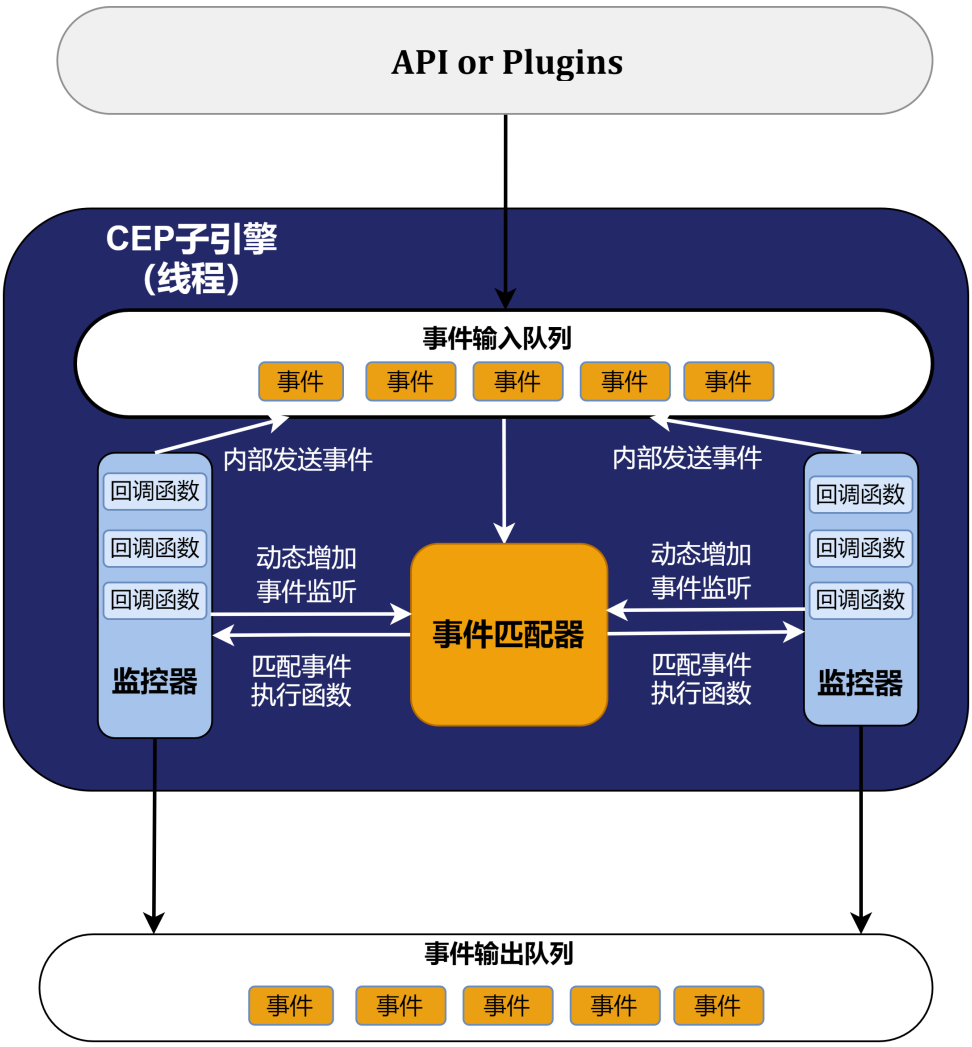

图 2-2 CEP 子引擎组件

2.4.1 事件输入队列

所有外部输入的事件进入 CEP SubEngine 之后将暂时放入事件输入队列（Event Input Queue）。事件输入队列为一个双端队列，外部输入事件只能从队尾入队，但是监视器内部可以将事件添加至队尾或者是队首。每个 SubEngine 维护一个输入队列。事件匹配器从队首按顺序取出事件，识别出满足预定义模式的事件，并执行相应的回调函数。

2.4.2 监视器

监视器（Monitor）被定义为一个类（class）。用户需要在 Monitor 中定义事件对应的处理逻辑。每次 SubEngine 启动之后，会实例化每个 Monitor。一个 Monitor 主要包含以下内容：

- 一个或多个事件监听（Event Listener)。 监听器是用于指定需要匹配的事件。它从进入 CEP 引擎的事件流中检测并筛选匹配的事件。
- 一个或多个回调函数。 回调函数是 CEP 引擎检测到匹配事件之后执行的操作。

一个简单的 Monitor 定义如下：

- 首先定义 StockTick 事件类型，这是监视器感兴趣的事件类型。
- Monitor 中定义一个 newTick 成员属性，该监视器中的所有操作都可以访问该变量。 newTick 可以保存 StockTick 事件。
- 监听所有的 StockTick 事件。
- 当找到 StockTick 事件时调用 processTick(stockTickEvent)。 processTick 将收到的所有 StockTick 事件的名称和价格写入 log 中。

```
class StockTick {
    name :: STRING
    price :: FLOAT
    def StockTick(n, p){
        name = n
        price = p
    }
}
class SimpleShareSearch {
 newTick :: StockTick //保存最新的 StockTick 事件
 //构造函数
 def SimpleShareSearch(){
  newTick = StockTick("init", 0.0)
 }
 def processTick(stockTickEvent)
 // 创建 CEP 子引擎之后，系统会自动构造 SimpleShareSearch 类对象为 Monitor 实例并调用
 onload 函数
 def onload() {
  //监听StockTick事件
  addEventListener(handler=processTick, eventType="StockTick", times="all")
 }
 // 收到StockTick事件之后执行回调函数，此处为在日志中记录StockTick相关信息。
 def processTick(stockTickEvent) {
  newTick = stockTickEvent
  str = "StockTick event received" +
   " name = " + newTick.name +
   " Price = " + newTick.price.string()
  writeLog(str)
 }
}
```

- 从已有的监视器中生成新的监视器

我们通常需要单个监视器来同时监听多个相同类型的事件。 例如，希望一台监视器监听并统计每个股票的价格变动。我们可以通过多次拷贝该监视器实例来生成多个相同类型的监视器，每个新生成的子监视器监听一只股票。

```
class NewStock {
    code :: STRING
```

```
    price :: DOUBLE
}
class SimpleShareSearch {
    numberTicks :: INT
    price :: DOUBLE
    stockName :: STRING

    def SimpleShareSearch(){
        stockName = ""
  numberTicks = 0
  price = 0.0
 }

    def matchTicks(newStock)
    def spawnTicks(newStock){
        numberTicks = numberTicks+1
        spawnMonitor(matchTicks, newStock)
    }
    // 监听所有 NewStock 事件。当发现 NewStock 事件时,
    // 调用从当前监视器深拷贝构造一个监视器实例，并执行 matchTicks() 方法。
    // 这个操作深拷贝了当前监视器的状态。
    def onload() {
        addEventListener(handler=spawnTicks, eventType="NewStock", times="all")
    }
    def processTick(ticks)
    def matchTicks(newStock) {
        stockName=newStock.code
        price = newStock.price
        addEventListener(handler=processTick, eventType="StockTick",
 condition=<StockTick.code==stockName>,times="all")
    }

   def processTick(ticks) {
        price = newStock.price
  str = "StockTick event received" +
   " name = " + ticks.name +
   " Price = " + ticks.price.string()
  writeLog(str)
   }
}
```

如果当前 CEP 引擎中收到三条代码（code）分别为 "sz001.hz", "sz002.hz", "sz003.hz" 的 NewStock 事件，则当前引擎中会有四个监听器，分别为：

- 在引擎启动时实例化的监听器，其监听 NewStock 事件，属性 numberTicks 为3。
- spawn 拷贝构造的监听器，其监听 StockTick.code = sz001.hz" 的 StockTick 事件，属性 numberTicks 为1。

- spawn 拷贝构造的监听器，其监听 StockTick.code = sz002.hz" 的 StockTick 事件，属性 numberTicks 为2。
- spawn 拷贝构造的监听器，其监听 StockTick.code = sz003.hz" 的 StockTick 事件，属性 numberTicks 为3。

2.4.3 事件监听器

事件监听器（Event Listener）监听相关事件流。事件匹配器依次分析每个事件，直到找到与其事件表达式匹配的事件序列时，由事件侦听器触发并执行相关回调函数。

目前 Event Listener 支持以下几种类型。

事件匹配：监听单一事件或者所有事件，并且限定事件条件

监听价格大于10.0 的股票。下例中 eventType 为事件，condition 为事件匹配条件，handler 为监听到符合条件的事件之后的回调函数。

```
addEventListener(handler=action, eventType=`Stock, condition=<Stock.price > 10.0>)
```

监听所有的股票

```
addEventListener(handler=action, eventType=`Stock)
```

监听任意事件

```
addEventListener(handler=action, eventType="any")
```

按时间触发

在固定时间触发，比如：在每天的8:30触发。下例中 at 指定触发时间频率，它是一个长度为6的元组，每个元素分别代表（秒，分钟, 小时，星期，日期, 月份）。

```
addEventListener(handler=action, at=(0,30,8,,,))
```

等待固定时间之后触发，比如：

- 从监听器被添加开始，每隔60秒触发一次。

```
addEventListener(handler=action, wait=60s)
```

- 从监听器被添加开始，等待60秒触发一次

```
addEventListener(handler=action, wait=60s,times=1)
```

同时限定时间和事件

在限定时间内匹配事件，比如在60秒内匹配到价格大于10.0的 Stock 事件，则执行回调函数。

```
addEventListener(handler=action, eventType=`Stock, condition=<Stock.price >
 10.0>, within = 60s,times=1)
```

在限定时间内未匹配事件，如在60秒内没有匹配到价格大于10.0的 Stock 事件，则执行回调函数。

```
addEventListener(handler=action, eventType=`Stock, condition=<Stock.price >
 10.0>, exceedTime= 60s,times=1)
```

时间相关的监听器又被称为计时器（timer）。

2.4.4 事件匹配器

事件匹配器（Event Matcher）对输入的事件流进行实时分析，以识别出满足预定义模式的事件。它可以处理高吞吐量的事件流，并快速地进行匹配操作。事件匹配器包含专为高性能、多维事件过滤而设计的数据结构和算法。

匹配器工作流程如下：

解析事件监听器，拆分事件匹配条件。一个事件监听器定义如下：

```
addEventListener(handler=action, eventType=`Stock, condition=<Stock.price > 10.0 and
 Stock.code == `IBM and Stock.volume > 10000>)
```

匹配器将监听条件拆分为多个子查询：<Stock.price > 10.0>, <Stock.code == `IBM>, <Stock.volume > 10000>。

事件匹配器采用树形结构搜索监听器。每种事件维护一个树，每个子节点为一个子查询，每个叶子节点为事件监听器。从根节点到叶子节点的路径组成了一个事件监听器完整的匹配条件。当有事件流过事件匹配器时，依次计算节点的子查询结果，如果结果为 true，则继续判断子节点的子查询结果。比如，当前有两个事件监听器，匹配条件分别是 <Stock.price >10.0 and Stock.volume > 10000> 和<Stock.price > 10.0 and Stock.volume <= 10000>。 此时 Stock 事件的匹配树结构如下：

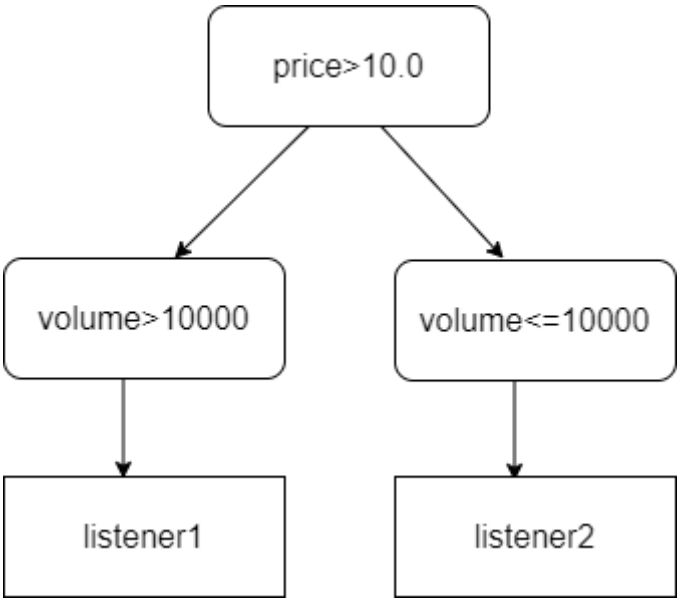

图 2-3 事件匹配树实例1

当某个 Stock 事件到达匹配器时，系统会先找到对应的 Stock 事件的匹配树。其次计算 price > 10.0 的结果，如果结果为 false，则匹配结束；如果结果为 true，则继续判断 volume > 10000 和 volumn <= 10000，直到匹配到叶子节点，则执行监听器绑定的回调函数。

对于以下情况，匹配器进行了进一步的优化。

- 匹配条件是事件属性的某个值，比如<Stock.code== “sz001.hz” >:

分别监听事件中同一个属性的不同值，匹配器采用哈希方式查找对应的监听器，从而将复杂度从 O(n) 降到 O(1)。这种查询的节点结构如下：

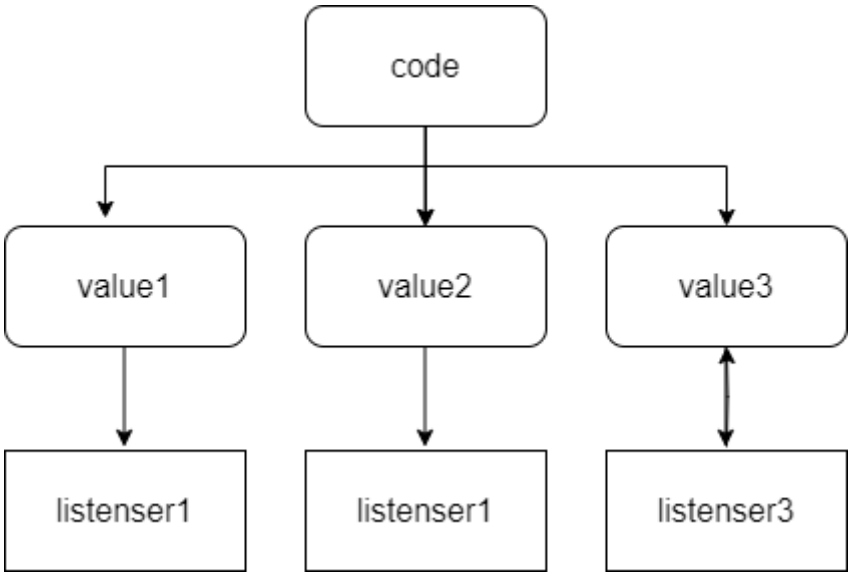

图 2-4 事件匹配树实例2

2.5 事件在 CEP 引擎中的处理顺序

在 CEP 引擎中，发送事件的方式不同，处理顺序也存在差别。本节将介绍 CEP 引擎对不同发送事件方式的处理顺序。

2.5.1 从外部写入的事件

所有从外部输入 CEP 引擎的事件，经反序列化之后进行 CEP SubEngine 的事件输入队列。比如通过 API、插件或 appendEvent 接口注入 CEP 引擎的事件，它们将被放入事件输入队列。

```
class MarketData{
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    def MarketData(m,c,p,q){
        market = m
        code = c
        price = p
        qty = q
    }
}
engine = createCEPEngine(name='cep1', monitors=<MainMonitor()>, dummyTable=dummy,
 eventSchema=[MarketData])
marketData = MarketData('sz', 's000', val[0], 100)
appendEvent(engine, [marketData])
```

2.5.2 routeEvent

routeEvent 将事件插入到当前 SubEngine 事件输入队列的队首。在事件匹配器处理完当前事件之后，会优先处理此事件。

如下为一个 Monitor 中的方法，CEP 执行此方法将事件 event1 和 event2 插入到事件输入队列的队首，其中 event2 位于 event1 前面。事件匹配器处理完当前事件之后，会优先处理 event2；在处理完 event2 之后，如果没有新的事件被插入到输入队列的队首，则处理 event1。

```
def simulate(name) {
  routeEvent(StockTick(name, 50.0)) //route event1
  routeEvent(StockTick(name, 30.0)) //route event2
}
```

2.5.3 sendEvent

sendEvent 将事件插入到当前 SubEngine 事件输入队列的队尾。CEP 引擎将按事件入队的顺序依次处理它们。

```
def simulate(name) {
  sendEvent(StockTick(name, 50.0))
}
```

2.5.4 emitEvent

emitEvent 将事件写入事件输出队列。事件流序列化器根据事件入队的顺序依次序列化事件，之后被写入异构流表。API（客户端）或者插件可以订阅此异构流表。

以下示例演示如何将最新的因子计算结果发送到事件流序列化器。

```
class UpdateFactor{
    sym :: STRING
    factor :: DOUBLE
    def UpdateFactor(code, val){
        sym = code
        factor = val
    }
}
class MarketData{
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    def MarketData(m,c,p,q){
        market = m
        code = c
        price = p
        qty = q
    }
}
class MainMonitor{
    maxPrice :: DOUBLE
    def MainMonitor(){ maxPrice = 0.0}
    def updateMarketData(event)
    def onload(){ addEventListener(updateMarketData,'MarketData',,'all') }
    def updateMarketData(event){
        if(event.price > maxPrice ){
            maxPrice = event.price
            emitEvent(UpdateFactor("maxPrice", maxPrice))
        }
    }
}
share streamTable(array(STRING, 0) as eventType, array(BLOB, 0) as blobs) as
 simulateResult
serializer = streamEventSerializer(name=`simulate, eventSchema=[UpdateFactor],
 outputTable=simulateResult)
dummy = table(array(STRING, 0) as eventType, array(BLOB, 0) as blobs)
engine = createCEPEngine(name='cep1', monitors=<MainMonitor()>, dummyTable=dummy,
 eventSchema=[MarketData], outputTale=serializer)
```

客户端订阅 simulateResult 流表，即可实时获取最新的最高价。

2.6 流计算引擎在 CEP 中的使用

依托 DolphinDB 的流数据框架，CEP 引擎能够轻松集成并管理各类流计算引擎。这极大地提升了从事件流中筛选数据进行复杂计算的便捷性和效率，无论是窗口聚合计算还是序列计算，都能得到高效处理。

以下示例在 CEP 引擎中收集股票事件并通过时序引擎计算行情 K 线。MainMonitor 属性的 streamMinuteBar_1min 是一个内存表，时序引擎的计算结果将保存至 streamMinuteBar_1min。

```
class MarketData{
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    eventTime :: TIMESTAMP
    def MarketData(m,c,p,q){
        market = m
        code = c
        price = p
        qty = q
        eventTime = now()
  }
}

class MainMonitor{
    streamMinuteBar_1min :: ANY //行情 K 线计算结果
    tsAggrOHLC :: ANY //时间序列聚合引擎
    def MainMonitor(){
        colNames =
 ["time","symbol","open","max","min","close","volume","amount","ret","vwap"]
        colTypes = [TIMESTAMP, STRING, DOUBLE, DOUBLE, DOUBLE, DOUBLE, INT, DOUBLE,
 DOUBLE, DOUBLE]
        streamMinuteBar_1min = table(10000:0,colNames, colTypes)
    }

    def updateMarketData(event)
    // 监听行情数据并创建时间序列聚合引擎，计算一分钟行情 K 线。
    def onload(){
        addEventListener(updateMarketData,'MarketData',,'all')
        colNames=["symbol","time","price","type","volume"]
        colTypes=[STRING, TIMESTAMP, DOUBLE, STRING, INT]
        dummy = table(10000:0,colNames,colTypes)
        colNames =
 ["time","symbol","open","max","min","close","volume","amount","ret","vwap"]
        colTypes = [TIMESTAMP, STRING, DOUBLE, DOUBLE, DOUBLE, DOUBLE, INT, DOUBLE,
 DOUBLE, DOUBLE]
        output = table(10000:0,colNames, colTypes)
        tsAggrOHLC = createTimeSeriesEngine(name="tsAggrOHLC", windowSize=60000,
 step=60000, metrics=<[first(price) as open ,max(price) as max,min(price)
```

```
 as min ,last(price) as close ,sum(volume) as volume ,wsum(volume, price)
 as amount ,(last(price)-first(price)/first(price)) as ret, (wsum(volume,
 price)/sum(volume)) as vwap]>, dummyTable=dummy, outputTable=streamMinuteBar_1min,
 timeColumn='time', useSystemTime=false, keyColumn="symbol", fill=`none)
    }

    def updateMarketData(event){
        tsAggrOHLC.append!(table(event.code as symbol, event.eventTime as time,
 event.price as price, event.market as type, event.qty as volume))
    }
}
dummy = table(array(STRING, 0) as eventType, array(BLOB, 0) as blobs)
engine = createCEPEngine(name='cep1', monitors=<MainMonitor()>, dummyTable=dummy,
 eventSchema=[MarketData])
```

第 3 章. 运维与可视化

本章将介绍如何运维 CEP 引擎以及如何可视化监控引擎中的数据动态变化。

3.1 状态监控

在 DolphinDB 中创建 CEP 引擎之后，所有的计算和处理都会在后台进行。用户可以执行 getCEPEngineStat 函数查看 CEP 引擎的运行状态。该函数返回一个字典，包含以下内容：

```
engineStat->
  name->
  user->
  status->
  lastErrorMessage->
  lastErrorTimestamp->
  useSystemTime->
  numOfSubEngine->
  queueDepth->
  eventsReceived->
  eventsEmitted->
  eventsOnOutputQueue->
eventSchema->
eventType eventField fieldType fieldTypeId fieldFormId
--------- ---------- --------- ----------- -----------
subEngineStat->
subEngineName eventsOnInputQueue listeners timers eventsRouted eventsSent
 eventsReceived eventsConsumed lastEventTime lastErrorMessage lastErrorTimestamp
------------- ----------------- --------- ------ ------------ ----------
 -------------- -------------- ------------- ---------------- ------------------
dataViewEngines->
name user status lastErrorMessage lastErrorTimestamp keyColumns outputTableName
 useSystemTime throttle numItems memoryUsed
---- ---- ------ ---------------- ------------------ ---------- --------------
 ------------- -------- -------- ----------
streamEngineStat->
```

为方便用户进行可视化监控，DolphinDB 的 Web 管理页面集成了 CEP 监控模块。用户通过功能面板里的CEP 流计算引擎访问该模块，从而查看并管理 CEP 引擎信息、监控其运行状态。

图 3-1 Web 管理界面示例

3.2 实时数据视图

DataViewEngine 为实时数据视图引擎，用于实时监控和展示事件流、事件关系以及分析结果。它允许对指定数据呈现只读视图，以便外部客户端（比如 Web 集群管理器）使用，从而帮助用户及时了解数据的动态变化。

下面以监控委托信息的 DataViewEngine 为例，说明如何创建 DataViewEngine 和如何更新数据。

```
// 定义 Orders 事件属性
class Orders{
    market :: STRING
    code :: STRING
    price :: DOUBLE
    qty :: INT
    def Orders(m,c,p,q){
        market = m
        code = c
        price = p
        qty = q
    }
}
// 定义监视器
class MainMonitor{
    def MainMonitor(){
    }
    // 删除引擎时自动调用，删除共享流表
    def onunload(){ undef('orderDV', SHARED) }
 def checkOrders(newOrder)
    // 创建 DataViewEngine，指定主键为 code 列，统计每个股票的最新的委托信息
    def onload(){
        addEventListener(handler=checkOrders,eventType='Orders',times='all')
        orderDV = streamTable(array(STRING, 0) as market, array(STRING, 0) as code,
 array(DOUBLE, 0) as price, array(INT, 0) as qty, array(TIMESTAMP, 0) as updateTime)
        share(orderDV,'orderDV')
```

```
        createDataViewEngine(name='orderDV', outputTable=objByName('orderDV'),
 keyColumns=`code, timeColumn=`updateTime)
    }
    // 更新每个股票的最新的委托信息
    def checkOrders(newOrder){
  getDataViewEngine(,'orderDV').append!(table(newOrder.market as market,
 newOrder.code as code, newOrder.price as price, newOrder.qty as qty))
    }
}
// 创建 CEP 引擎
dummy = table(array(STRING, 0) as eventType, array(BLOB, 0) as blobs)
engine = createCEPEngine(name='cep1', monitors=<MainMonitor()>, dummyTable=dummy,
 eventSchema=[Orders])
```

Web 端提供了实时监控界面，点击 CEP 流计算引擎下模块下的数据视图，就可以查看当前 CEP 引擎所有的数据视图。下拉红色框可以选择数据视图，黄色框中为当前的主键值。点击主键值后，右边蓝色框中对应的属性值将会实时滚动更新。

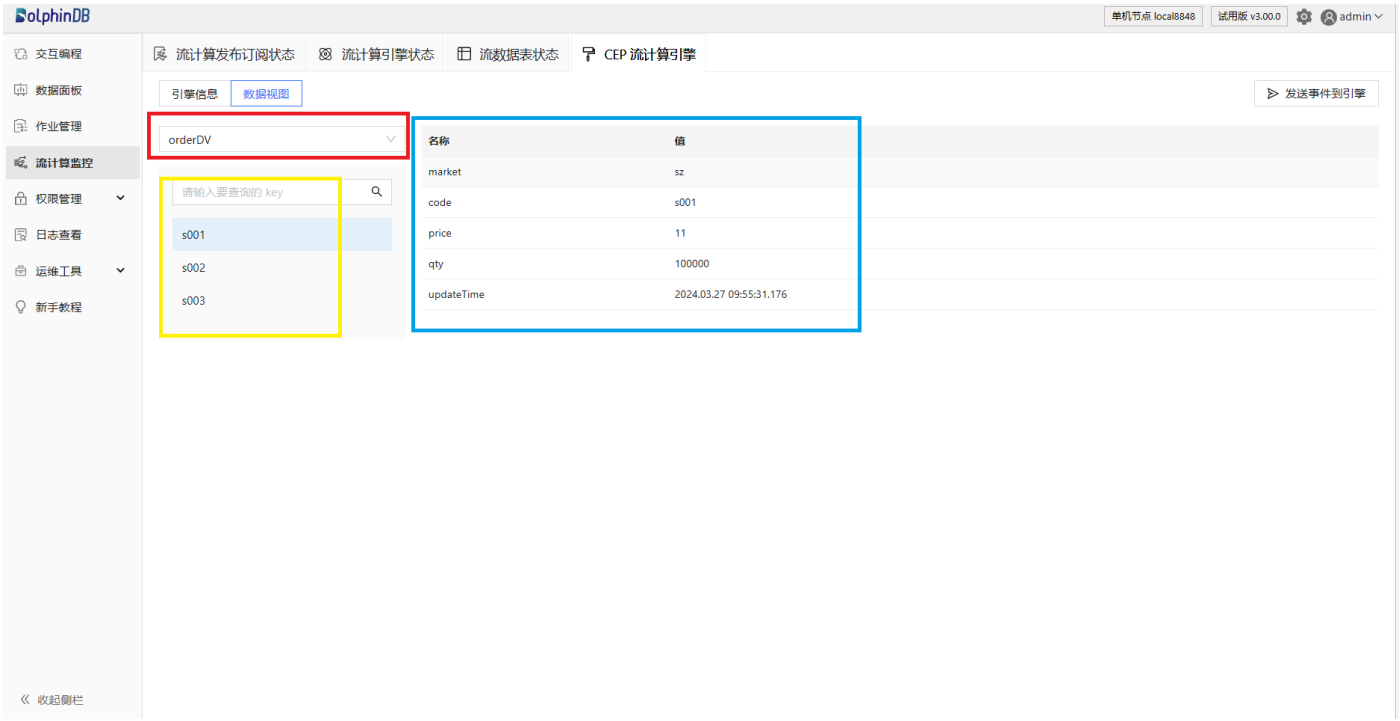

图 3-2 Web 数据视图界面示例

3.2.1 实现原理

DataViewEngine 维护一个以 keyedColumns 为主键的键值内存表。向 DataViewEngine 中添加或者更新记录时，系统会自动检查新记录的主键值。如果新记录的主键值与已有记录的主键值重复时，则更新表中对应的记录；否则，添加新记录到内存表中。新的记录和更新的记录都会被推送至向下游流表。Web 端订阅 DataViewEngine 的输出流表，则可以实时获取到每个键值最新的记录。

第 4 章. 特性与优势

本章主要介绍 CEP 引擎的特性与优势，并对事件匹配的性能进行了测试与分析。

4.1 特性与优势

- **高吞吐、低时延性**：针对不同的事件匹配规则， CEP 引擎事件匹配器进行了多种优化。同时 CEP 引擎支持多线程并行处理，使得 CEP 引擎能够实时处理大量事件流，并快速响应关键的业务事件。
- **灵活性与高扩展性**：CEP 引擎支持多种事件匹配规则以及用户自定义回调函数，以适应不断变化的业务需求。
- **强大的计算引擎**：CEP 引擎支持使用 DolphinDB 内置流计算引擎，方便对实时数据进行多种复杂计算，比如聚合计算、状态计算等，提升复杂策略的计算性能。
- **流批一体**：支持将历史数据加速回放到 CEP 引擎中，从而实现策略回测与实时场景流批一体，降低用户开发成本。
- **可视化和监控**：DolphinDB 为 CEP 引擎提供了可视化监控工具，可以实时监控和分析事件处理过程。还提供了数据面板，支持实时生成各种图表，如趋势图、订单图等。
- **多种 API 写入**：DolphinDB 支持多种 API，方便用户从客户端与 CEP 引擎交互，包括将事件写入 CEP 引擎，订阅、解析、处理 CEP 引擎输出的事件等。

4.2 性能分析

在实时计算中，端到端的响应延迟是衡量计算性能最重要的指标。对于 CEP 引擎而言，其性能的关键在于如何在接收到海量事件后，迅速而准确地从众多监听器中筛选出符合条件的监听器，并高效执行其回调函数。

本章重点对 CEP 引擎的性能进行分析。测试的事件监听数量为40万，其中监听的事件是10种。为了方便测试，这些事件具有相同的属性。

4.2.1 测试脚本

- 准备数据：MockData 模块模拟生成100支股票的 K 线数据，并随机将数据转成 OHLCBar 事件。

```
class OHLCBar{
 securityid :: STRING
 tradetime :: TIMESTAMP
 open :: DOUBLE
 close :: DOUBLE
 high :: DOUBLE
 low :: DOUBLE
 vol :: INT
 val :: DOUBLE
 vwap :: DOUBLE
```

```
 def OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_, vwap_)
 {
  securityid = securityid_
  tradetime = tradetime_
  open = open_
  close = close_
  high = high_
  low = low_
  vol = vol_
  val = val_
  vwap = vwap_
 }
}
class OHLCBar1 : OHLCBar{
 def OHLCBar1(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar2 : OHLCBar{
 def OHLCBar2(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar3 : OHLCBar{
 def OHLCBar3(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar4 : OHLCBar{
 def OHLCBar4(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar5 : OHLCBar{
 def OHLCBar5(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
```

```
class OHLCBar6 : OHLCBar{
 def OHLCBar6(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar7 : OHLCBar{
 def OHLCBar7(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar8 : OHLCBar{
 def OHLCBar8(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar9 : OHLCBar{
 def OHLCBar9(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
class OHLCBar10 : OHLCBar{
 def OHLCBar10(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_) {
        OHLCBar(securityid_, tradetime_, open_, close_, high_, low_, vol_, val_,
 vwap_)
 }
}
use DolphinDBModules::simulateData
startDate = 2020.01.01
endDate = 2020.01.05
securityNumber = 1000
t = genStockMinuteKLine(startDate, endDate, securityNumber)
priceCondition = sort(rand(100, 20))
qtyCondiyion =  int(rand(1.0, 20) * 100000)
securityidList = exec distinct securityid from t
eventSchema= [OHLCBar1 ,OHLCBar2,
 OHLCBar3,OHLCBar4,OHLCBar5,OHLCBar6,OHLCBar7,OHLCBar8,OHLCBar9,OHLCBar10]
datas = []
for(i in 0..(size(t)-1)) {
```

```
 d = rand(eventSchema,1)[0](t[i].securityid, t[i].tradetime, t[i].open, t[i].close,
 t[i].high, t[i].low, t[i].vol, t[i].val, t[i].vwap)
 datas.append!(d)
}
datas.shuffle!()
```

- 定义 CEP 引擎

```
class Monitor{
    securityidList :: STRING VECTOR
    priceList :: INT VECTOR
    qtyList ::LONG VECTOR
    classNameLits :: STRING VECTOR
    time :: TIMESTAMP
    def Monitor(securityid, price, qty){
        securityidList = securityid
        priceList=price
        qtyList=qty
        classNameLits = ["OHLCBar1" ,"OHLCBar2",
 "OHLCBar3","OHLCBar4","OHLCBar5","OHLCBar6","OHLCBar7","OHLCBar8","OHLCBar9","OHLCB
ar10"]
    }
    def callback(event){
        time=now()
    }
    def onload(){
        for(i in 1..400000){
            eventType = rand(classNameLits, 1)[0]
            securityid  = rand(securityidList, 1)[0]
            price = rand(priceList, 1)[0]
            qty = rand(qtyList,1)[0]
            condition = parseExpr(eventType+".securityid==`"+securityid+" and "+
 eventType+".close > "+string(price)+" and "+eventType+".vol < "+string(qty))
            addEventListener(handler=callback,eventType=eventType,
 condition=condition)
        }
    }
}

dummy = table(array(STRING, 0) as eventType, array(BLOB, 0) as blobs)
engineCep = createCEPEngine(name='cep1', monitors=<Monitor(securityidList,
 priceCondition, qtyCondiyion)>, dummyTable=dummy, eventSchema=eventSchema,
 eventQueueDepth=1000000)
```

- 向 CEP 引擎中注入数据。重复注入10次数据，并计算时间的平均值。

```
timeT = 0
consumedT = 0
```

```
for (i in 1..10) {
    prevConsumed = (getCEPEngineStat(`cep1)[`subEngineStat].eventsConsumed)
    testData =  rand(datas, 1024)
    startTime = now()
    appendEvent(engineCep, testData)
    sleep(5000)
    timeCost = getCEPEngineMonitor(`cep1,`cep1,`Monitor)[0].time - startTime
    Consumed = (getCEPEngineStat(`cep1)[`subEngineStat].eventsConsumed)-prevConsumed
    timeT += timeCost
    consumedT += Consumed
}
print(timeT/10)
print(consumedT/10)
```

4.2.2 测试环境

主机：OptiPlex 5000 CPU：12th Gen Intel(R) Core(TM) i5-12500 6 CPU核 12 超线程

内存：32G

OS：Ubuntu Linux release 20.04

4.2.3 测试结果

时间：11毫秒

事件数：1024

总共匹配的监听器数量：11,501

平均每个事件匹配时间：11微秒

第 5 章. 应用场景

CEP 引擎可以应用于金融、物联网、电信、供应链等领域，以实现实时监控、异常检测、决策支持等功能。本章将从金融高频策略场景和物联网异常检测两个场景介绍 CEP 引擎的应用。

5.1 金融场景

在金融场景中，通过 CEP 引擎用户可以实现交易策略（如算法策略、组合交易、套利交易等）、交易风控（如风险控制、实时熔断）、交易监控（如可视化监控）等业务。以下为一个应用 CEP 引擎实现金融高频策略场景。

5.1.1 场景需求

以下为基于股票逐笔成交数据的事件驱动策略。策略根据每支股票的最新的成交价涨幅和累计成交量判断是否执行下单操作：

- 根据每一笔成交数据触发计算两个实时因子：最新成交价相对于15秒内最低成交价涨幅（R）、过去1分钟累计成交量（V）；
- 在策略启动时设定每支股票的两个因子阈值（R0、V0），每当实时因子值更新后判断是否 R>R0 且 V>V0， 若是则触发下单；
- 下单后1分钟内仍未成交则触发对应的撤单。

5.1.2 实现脚本

- 定义事件

```
//  定义股票逐笔成交事件
class StockTick {
 securityid :: STRING
 time :: TIMESTAMP
 price ::  DOUBLE
 volume :: INT
 def StockTick(securityid_, time_, price_, volume_) {
  securityid = securityid_
  time = time_
  price = price_
  volume = volume_
 }
}
// 定义成交回报事件
class ExecutionReport {
 orderid :: STRING
 securityid :: STRING
 price :: DOUBLE
```

```
 volume :: INT
 def ExecutionReport(orderid_, securityid_, price_, volume_) {
  orderid = orderid_
  securityid = securityid_
  price = price_
  volume = volume_
 }
}
// 定义下单事件
class NewOrder {
 orderid :: STRING
 securityid :: STRING
 price :: DOUBLE
 volume :: INT
 side :: CHAR
 type :: INT
 def NewOrder(orderid_, securityid_, price_, volume_, side_, type_) {
  orderid = orderid_
  securityid = securityid_
  price = price_
  volume = volume_
  side = side_
  type = type_
 }
}
// 定义撤单事件
class CancelOrder {
 orderid :: STRING
 def CancelOrder(orderid_) {
  orderid = orderid_
 }
}
```

- 定义 Monitor 监视器

StrategyMonitor 类中实现了策略的具体逻辑：

- StrategyMonitor 作为构造函数，表示在实例化时需要传入策略编号和策略参数。策略参数为策略操作的一组股票编号以及每支股票对应的涨幅阈值和成交量阈值。
- 通过 `createDataViewEngine` 函数创建一个数据视图，用于保存内存中的变量并且供外部访问。记录的内容包括策略对应的股票编号、因子值、下单数量、超时订单数量等。虽然数据视图并非策略逻辑实现的必需元素，但它有助于观察策略运行过程中值的变化。
- 通过 `createReactiveStateEngine` 函数创建响应式状态引擎，用于逐条响应窗口因子计算，对每支股票都计算最新成交价相对于15秒内最低成交价涨幅、过去1分钟累计成交量。

- Monitor 创建时立刻监听逐笔成交事件，其仅匹配策略标的对应的事件，计算因子并更新数据视图中的因子值；根据实时因子值和策略配置来判断是否下单，若策略判断应该下单则向外部发送下单事件，更新数据视图中的下单数量和金额。
- 每一次发送下单事件后，启动成交回报计时器，仅匹配该下单事件的订单编号对应的成交回报。若 1 分钟内未收到对应的成交回报，则触发撤单，即向外部发送撤单事件，更新数据视图中的下撤单数量。
- 每一次发送下单事件后，启动监听成交回报事件，仅匹配该下单事件的订单编号对应的成交回报。收到成交回报后更新数据视图中的累计成交金额。

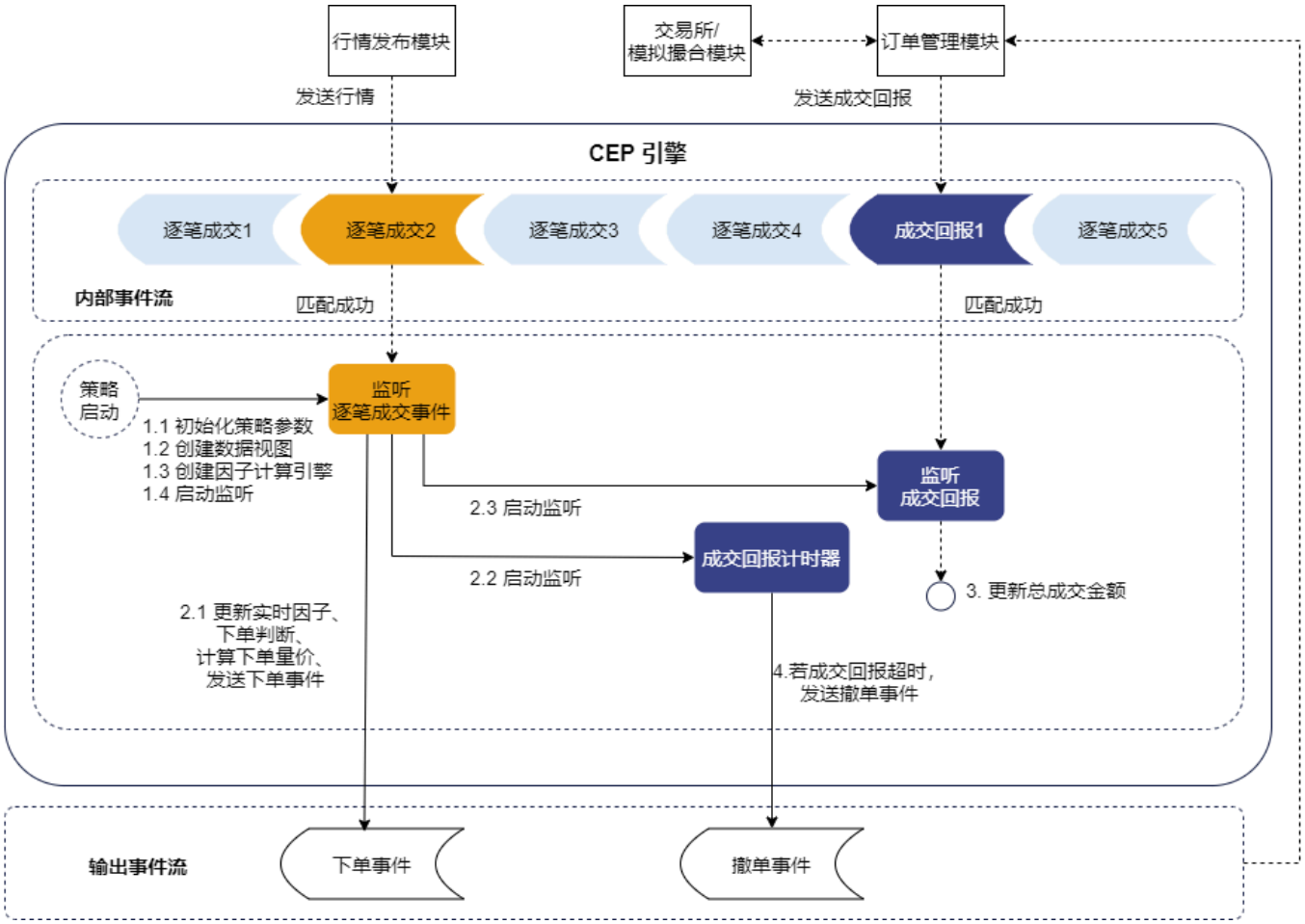

图 5-1 金融场景示意图

```
class StrategyMonitor {
 strategyid :: INT // 策略编号
 strategyParams :: ANY // 策略参数：策略标的、标的参数配置
 dataview :: ANY // Data View 监控

 def StrategyMonitor(strategyid_, strategyParams_) {
  strategyid = strategyid_
  strategyParams = strategyParams_
 }

 def execReportExceedTimeHandler(orderid, exceedTimeSecurityid){
  emitEvent(CancelOrder(orderid)) // 发送撤单事件到外部
  timeoutOrderNum = (exec timeoutOrderNum from self.dataview where
 securityid=exceedTimeSecurityid)[0] + 1
```

```
 updateDataViewItems(engine=self.dataview, keys=exceedTimeSecurityid,
valueNames=`timeoutOrderNum, newValues=timeoutOrderNum) // 更新data view
}

def execReportHandler(execReportEvent) {
 executionAmount = (exec executionAmount from self.dataview where
securityid=execReportEvent.securityid)[0] +
execReportEvent.price*execReportEvent.volume
 executionOrderNum = (exec executionOrderNum from self.dataview where
securityid=execReportEvent.securityid)[0] + 1
 updateDataViewItems(engine=self.dataview, keys=execReportEvent.securityid,
valueNames=["executionAmount","executionOrderNum"],
newValues=(executionAmount,executionOrderNum)) // 更新data view
}

def handleFactorCalOutput(factorResult){
 factorSecurityid = factorResult.securityid[0]
 ROC = factorResult.ROC[0]
 volume = factorResult.volume[0]
 lastPrice = factorResult.lastPrice[0]
 updateDataViewItems(engine=self.dataview, keys=factorSecurityid,
valueNames=["ROC","volume"], newValues=(ROC,volume))
 if (ROC>strategyParams[factorSecurityid][`ROCThreshold] &&
volume>strategyParams[factorSecurityid][`volumeThreshold]) {
  orderid = self.strategyid+"_"+factorSecurityid+"_"+long(now())
  newOrder = NewOrder(orderid , factorSecurityid, lastPrice*0.98, 10, 'B', 0) //
构造下单事件，按低于最新成交价的2%买入一手
  emitEvent(newOrder) // 发送下单事件到外部
  newOrderNum = (exec newOrderNum from self.dataview where
securityid=factorSecurityid)[0] + 1
  newOrderAmount = (exec newOrderAmount from self.dataview where
securityid=factorSecurityid)[0] + lastPrice*0.98*10
  updateDataViewItems(engine=self.dataview, keys=factorSecurityid, valueNames=
["newOrderNum", "newOrderAmount"], newValues=(newOrderNum, newOrderAmount)) //
更新data view
  addEventListener(handler=self.execReportExceedTimeHandler{orderid,
factorSecurityid}, eventType="ExecutionReport",
condition=<ExecutionReport.orderid=orderid>, times=1, exceedTime=60s, times=1) //
启动成交回报计时器
  addEventListener(handler=execReportHandler, eventType="ExecutionReport",
condition=<ExecutionReport.orderid=orderid>, times="all") // 启动成交回报监听
 }
}

def tickHandler(tickEvent){
 factorCalEngine = getStreamEngine(`factorCal)
 insert into factorCalEngine values([tickEvent.securityid, tickEvent.time,
tickEvent.price, tickEvent.volume])
```

```
 }

 def initDataView(){
  // 创建 data view，监控策略执行状态
  share(streamTable(1:0,
 `securityid`strategyid`ROCThreshold`volumeThreshold`ROC`volume`newOrderNum`newOrder
Amount`executionOrderNum`executionAmount`timeoutOrderNum`updateTime,
 `STRING`INT`INT`INT`INT`INT`DOUBLE`INT`DOUBLE`INT`TIMESTAMP), "strategyDV")
  dataview = createDataViewEngine(name="Strategy_"+strategyid,
 outputTable=objByName(`strategyDV), keyColumns=`securityId, timeColumn=`updateTime)
  num = strategyParams.size()
  securityids = strategyParams.keys()
  ROCThresholds = each(find{,"ROCThreshold"}, strategyParams.values())
  volumeThresholds = each(find{,"volumeThreshold"}, strategyParams.values())
  dataview.tableInsert(table(securityids, take(self.strategyid, num) as strategyid,
 ROCThresholds, volumeThresholds, take(int(NULL), num) as ROC, take(int(NULL),
 num) as volume, take(0, num) as newOrderNum, take(0, num) as newOrderAmount,
 take(0, num) as executionOrderNum, take(0, num) as executionAmount, take(0, num) as
 timeoutOrderNum))
 }

 def createFactorCalEngine(){
  dummyTable = table(1:0, `securityid`time`price`volume,
 `STRING`TIMESTAMP`DOUBLE`INT)
  metrics = [<(price\tmmin(time, price, 15s)-1)*100>, <tmsum(time, volume, 60s)>,
 <price> ] // 最新成交价相对于15秒内最低成交价涨幅 ,1分钟累计成交量，最新成交价
  factorResult = table(1:0, `securityid`ROC`volume`lastPrice,
 `STRING`INT`LONG`DOUBLE)
  createReactiveStateEngine(name="factorCal", metrics=metrics ,
 dummyTable=dummyTable, outputTable=factorResult, keyColumn=`securityid,
 outputHandler=handleFactorCalOutput, msgAsTable=true)
 }

 def onload() {
  // 初始化 data view
  initDataView()
  // 创建因子计算引擎
  createFactorCalEngine()
  // 启动监听逐笔成交事件
  securityids = strategyParams.keys()
  addEventListener(handler=tickHandler, eventType="StockTick",
 condition=<StockTick.securityid in securityids>, times="all")
 }
}
```

- 创建策略实例

策略编号为 1，将监控 300001、300002、3000003 三支股票。策略发送给外部的下单和撤单事件存储在异构流数据表 output 中。

```
dummy = table(array(STRING, 0) as eventType, array(BLOB, 0) as blobs)
share(streamTable(array(STRING, 0) as eventType, array(BLOB, 0) as blobs,
 array(STRING, 0) as orderid), "output")
outputSerializer = streamEventSerializer(name=`serOutput,
 eventSchema=[NewOrder,CancelOrder], outputTable=objByName("output"),
 commonField="orderid")
strategyid = 1
strategyParams = dict(`300001`300002`300003, [dict(`ROCThreshold`volumeThreshold,
 [1,1000]), dict(`ROCThreshold`volumeThreshold, [1,2000]),
 dict(`ROCThreshold`volumeThreshold, [2, 5000])])
engine = createCEPEngine(name='strategyDemo', monitors=<StrategyMonitor(strategyid,
 strategyParams)>, dummyTable=dummy, eventSchema=[StockTick,ExecutionReport],
 outputTable=outputSerializer)
```

5.1.3 模拟数据写入

以下脚本首先模拟120条逐笔成交事件，并写入 CEP 引擎。逐笔成交事件包括5支不同的股票。之后模拟多条成交回报事件写入 CEP 引擎。

```
ids = `300001`300002`300003`600100`600800
for (i in 1..120) {
 sleep(500)
 tick = StockTick(rand(ids, 1)[0], now()+1000*i, 10.0+rand(1.0,1)[0],
 100*rand(1..10, 1)[0])
 getStreamEngine(`strategyDemo).appendEvent(tick)
}
sleep(1000*20)
print("begin to append ExecutionReport")
for (orderid in (exec orderid from output where eventType="NewOrder")){
 sleep(250)
 if(not orderid in (exec orderid from output where eventType="CancelOrder")) {
  execRep = ExecutionReport(orderid, split(orderid,"_")[1], 10, 100)
  getStreamEngine(`strategyDemo).appendEvent(execRep)
 }
}
```

以上模拟脚本大约需要运行2分钟，在运行过程中可以通过 Web 页面观察策略的执行状态。Web 显示的是当前最新的策略状态，并且会自动刷新，历史状态可以通过 dataview 引擎的输出表查看。

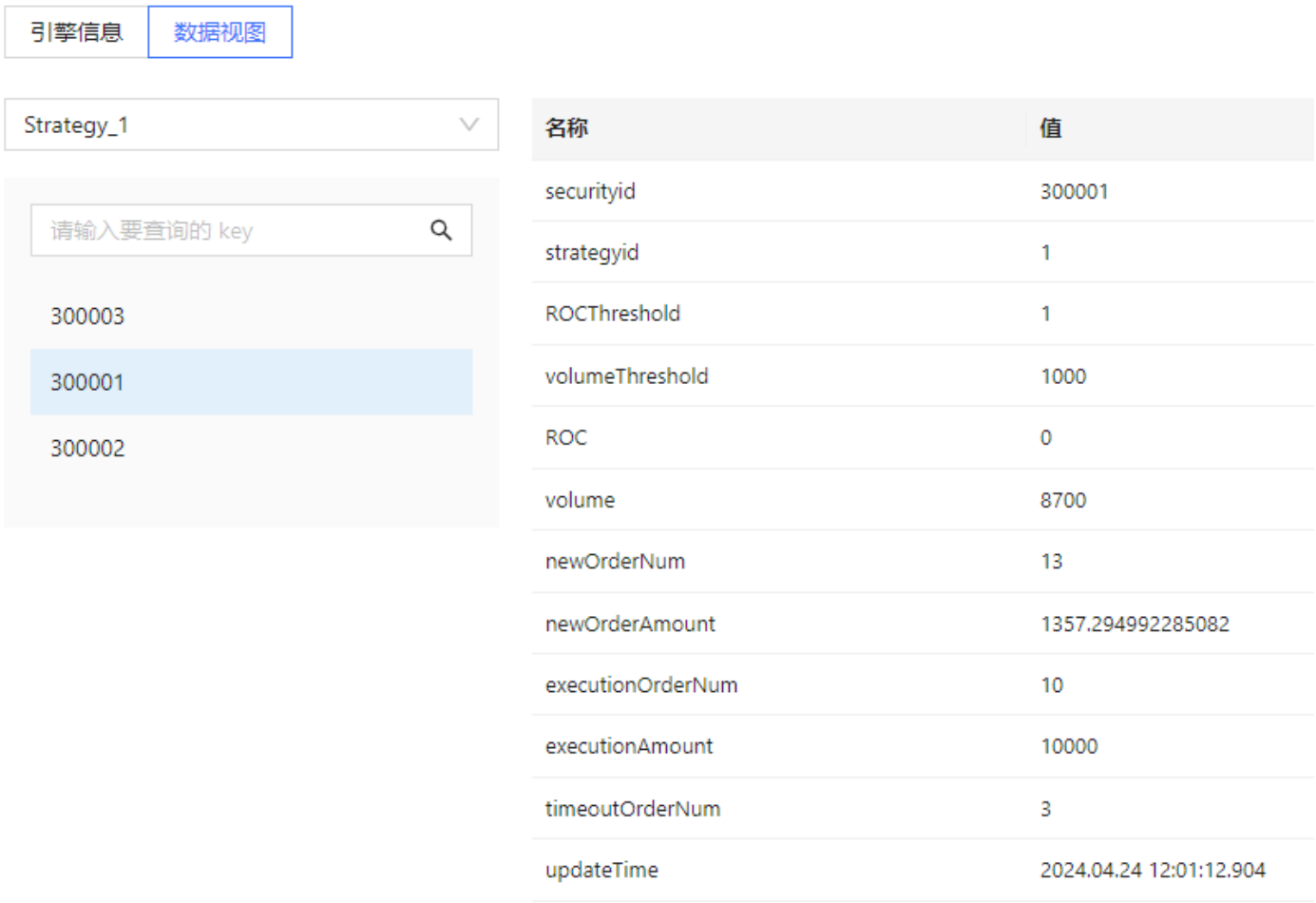

图 5-2 数据视图监控界面

查询 dataview 引擎的输出表。

```
select * from strategyDV
```

得到结果：

securityid	strategyid	ROCThreshold	volumeThreshold	ROC	volume	newOrderNum	newOrderAmount	executionOrderNum	executionAmount	timeoutOrderNum	updateTime
300001	1	1	1,000	0	8,700	13	1,357.294992285082	0	0	1	2024.04.24 12:01:03.100
300001	1	1	1,000	0	8,700	13	1,357.294992285082	0	0	2	2024.04.24 12:01:04.100
300001	1	1	1,000	0	8,700	13	1,357.294992285082	0	0	3	2024.04.24 12:01:08.104
300001	1	1	1,000	0	8,700	13	1,357.294992285082	1	1,000	3	2024.04.24 12:01:10.148
300002	1	1	2,000	0	3,300	2	206.11314598280006	1	1,000	0	2024.04.24 12:01:10.399
300002	1	1	2,000	0	3,300	2	206.11314598280006	2	2,000	0	2024.04.24 12:01:10.649
300001	1	1	1,000	0	8,700	13	1,357.294992285082	2	2,000	3	2024.04.24 12:01:10.899

图 5-3 数据视图输出结果

向外部发送的事件写入创建 CEP 引擎时指定的序列化器中并最终输出到异构流表 output。可以通过 API 订阅该表以对接真实的交易柜台或者模拟撮合模块：

eventType	blobs	orderid
NewOrder	1_300001_1713960038133□300001□= 7□K□$@ □□□B□□□□	1_300001_1713960038133
NewOrder	1_300001_1713960041636□300001□□□□□`$@ □□□B□□□□	1_300001_1713960041636
NewOrder	1_300001_1713960042638□300001□□□:□v□$@ □□□B□□□□	1_300001_1713960042638
NewOrder	1_300001_1713960043638□300001ק□I□□K%@ □□□B□□□□	1_300001_1713960043638
NewOrder	1_300001_1713960044140□300001□□Q□□jd%@ □□□B□□□□	1_300001_1713960044140
CancelOrder	1_300001_1713960003098□	1_300001_1713960003098
CancelOrder	1_300001_1713960004099□	1_300001_1713960004099
CancelOrder	1_300001_1713960008102□	1_300001_1713960008102

图 5-4 最终输出结果

5.2 物联网场景

在物联网场景下，用户可以通过 DolphinDB 对物联网设备产生的各种传感器数据（如温度、湿度、压力、电压等）进行实时监控，并对这些数据进行实时的异常检测、警告通知、故障诊断和预测维护，从而提高物联网系统的安全性和稳定性。以下为 CEP 引擎应用于工厂异常检测场景的例子。

5.2.1 场景需求

现有一个物联网监控系统，采集器每秒采集一次数据。不同的传感器采集不同数据。分别有温度采集器、湿度采集器和电压采集器。

希望实现以下异常检测需求：

1. 每3分钟内，若同一温度传感器出现2次40摄氏度以上并且3次30摄氏度以上，则系统输出警告信息。
2. 若温度传感器出现1次40度以上且接下来5秒内出现一次湿度大于60，系统输出警告信息。
3. 每次电压低于200V，系统输出警告信息。
4. 若传感器网络断开，5分钟内无数据，系统输出警告信息。

5.2.2 实现脚本

```
//定义传感器事件
class SensorData{
    deviceID :: INT
    deviceType :: STRING
    detectionValue :: DOUBLE
    def SensorData(id, type, value){
      deviceID = id
      deviceType = type
      detectionValue = value
    }
```

```
}
class NewSensor{
    deviceID :: INT
    deviceType :: STRING
    def NewSensor(id, type){
      deviceID = id
      deviceType = type
    }
}
// 创建共享表，记录异常信息。
share(streamTable(array(INT, 0) as deviceID, array(INT,0) as anomalyType,
 array(STRING, 0) as anomalyString, array(TIMESTAMP, 0) as updateTime),'SensorDV')
// 定义Monitor
class monitor{
    dataCount :: INT      //记录每五分钟的数据量。
    deviceID ::INT        //为每个设备绑定一个 monitor，记录设备 ID。
    dv :: ANY             // DataViewEngine。
    over40count :: INT    //记录三分钟内超过40度的次数。
    over30count :: INT    //记录三分钟内超过30度的次数。
    def monitor(){
        dataCount = 0
        deviceID = NULL
        over40count = 0
        over30count = 0
    }
    // 方法声明
    def initSensor(id)
    def checkHumidity(temperature)
    def outputAnomaly(type, str, sensor)
    def checkConnection()
    def addDataCount(sensor)
    def addTemperature(sensor)
    def checkTemperature()

    def spawnDevice(sensor){
        spawnMonitor(initSensor, sensor.deviceID)
    }
    // 加载函数，创建 DataViewEngine 监控异常。
    // 为每个新的设备 spwan 一个 monitor，用于监听第一种和第四种异常。
    // 对第二种异常增加监听，先监听温度大于40度的情况。
    // 对第三种异常增加监听，回调函数为写入 DataViewEngine。
    def onload() {
        addEventListener(handler=spawnDevice, eventType="NewSensor",times="all")
        dv = createDataViewEngine(name='SensorDV',
 outputTable=objByName('SensorDV'), keyColumns=`deviceID, timeColumn=`updateTime)
        addEventListener(handler=checkHumidity,
 eventType="SensorData",condition=<SensorData.deviceType =='temperature'and
 SensorData.detectionValue > 40>, times="all")
```

```
        addEventListener(handler=self.outputAnomaly{3,"Abnormal voltage"},
eventType="SensorData",condition=<SensorData.deviceType =='voltage'and
SensorData.detectionValue < 200>, times="all")
    }
    // 将异常信息写入数据视图。
    def outputAnomaly(type, str, sensor){
        dv.append!(table(sensor.deviceID as deviceID, type as anomalyType, str as
anomalyString))
    }
    // 匹配到温度大于40的事件之后，动态增加监听，监听5秒内湿度大于60的事件。
    def checkHumidity(temperature){
        addEventListener(handler=self.outputAnomaly{2,"Abnormal humidity"},
eventType="SensorData",condition=<SensorData.deviceType =='humidity' and
SensorData.detectionValue > 60>, times=1, within=5s)
    }
    // 将当前设备异常信息写入数据视图。
    def outputAnomalyWithID(type, str){
        dv.append!(table(deviceID as deviceID, type as anomalyType, str as
anomalyString))
    }
    // spawn之后的初始化函数，每个设备对第一种和第四种异常增加监听，在设定时间间隔内查看。
    def initSensor(id){
        deviceID = id
        addEventListener(handler=addDataCount,
eventType="SensorData",condition=<SensorData.deviceID == id>, times="all")
        addEventListener(handler=addTemperature,
eventType="SensorData",condition=<SensorData.deviceType =='temperature' and
SensorData.deviceID == id and SensorData.detectionValue > 30>, times="all")
        addEventListener(handler=checkTemperature, times="all", wait=3m)
        addEventListener(handler=checkConnection, times="all", wait=5m)
    }
    // 检查连接情况，如果5分钟内没有数据，则输出第四种异常。
    def checkConnection(){
        if(dataCount==0){
            outputAnomalyWithID(4,"Abnormal connection")
        }
        dataCount = 0
    }
    // 统计当前设备每5分钟的数据量
    def addDataCount(sensor){
        dataCount = dataCount + 1
    }
    // 统计当前温度设备每3分钟异常温度的次数，如果当前设备不是温度传感器，此方法不会被调用。
    def addTemperature(sensor){
        over30count = over30count + 1
        if(sensor.detectionValue > 40){
            over40count = over40count + 1
        }
```

```
    }
    // 检查异常温度情况，
    // 每3分钟内，若同一温度传感器出现2次40摄氏度以上并且3次30摄氏度以上，输出异常信息。
    def checkTemperature(){
        if(over40count >= 2 and over30count >= 3){
            outputAnomalyWithID(1,"Abnormal temperature")
        }
        over30count = 0
        over40count = 0
    }
}

dummy = table(array(STRING, 0) as eventType, array(BLOB, 0) as blobs)
engine = createCEPEngine(name='sensorAnomalyDetection', monitors=<monitor()>,
 dummyTable=dummy, eventSchema=[SensorData, NewSensor])
```

5.2.3 模拟数据写入

通过以下脚本模拟写入9 个设备，三种传感器的监控数据。

```
index = 1
for(i in 0..2){
  for(type in `temperature`humidity`voltage){
    data = NewSensor(i, type)
    appendEvent(engine, data)
    index = index +1
  }
}
sleep(1000)
value=[31, 60, 199, 35, 50, 201, 41, 50, 210]
index = 1
for(i in 0..2){
  for(type in `temperature`humidity`voltage){
    data = SensorData(index, type, value[index-1])
    appendEvent(engine, data)
    index = index +1
  }
}
sleep(1000)
value=[25, 70, 200, 35, 50, 201, 41, 50, 210]
index = 1
for(i in 0..2){
  for(type in `temperature`humidity`voltage){
    data = SensorData(index, type, value[index-1])
    appendEvent(engine, data)
    index = index +1
  }
```

```
}
sleep(1000)
value=[25, 35, 200, 35, 50, 201, 42, 50, 210]
index = 1
for(i in 0..2){
  for(type in `temperature`humidity`voltage){
    data = SensorData(index, type, value[index-1])
    appendEvent(engine, data)
    index = index +1
  }
}
sleep(1000)
```

模拟数据写入3分钟后，查询数据视图。

```
getDataViewEngine(`sensorAnomalyDetection, `SensorDV)
```

得到结果：

	deviceID	anomalyType	anomalyString	updateTime
0	3	3	Abnormal voltage	2024.03.19 18:44:25.336
1	2	2	Abnormal humidity	2024.03.19 18:44:26.336
2	7	1	Abnormal temperature	2024.03.19 18:47:24.337

图 5-5 物联网场景数据视图结果

也可以从 Web 监控界面查看结果：

图 5-6 物联网场景 Web 数据视图

第 6 章. 总结与未来规划

DolphinDB 作为集成了分布式计算、实时流计算及分布式存储于一体的高性能时序数据库，为不同领域的流数据处理提供了完备的解决方案。CEP 引擎进一步完善了 DolphinDB 流数据处理功能，并拓宽了其应用场景。本白皮书从复杂事件处理的基本概念出发，介绍了 DolphinDB CEP 引擎的功能与架构，并通过具体的示例展示了 DolphinDB CEP 引擎在不同场景下的应用。

在未来，DolphinDB 将持续优化 CEP 引擎的功能以及性能，不断拓展应用领域：

- 支持用户定义更多的事件匹配规则。
- 推出更多的业务插件，方便用户解决特定业务场景的问题。
- 支持在 CEP 引擎中使用 JIT 计算。