



(12)发明专利申请

(10)申请公布号 CN 106664322 A

(43)申请公布日 2017.05.10

(21)申请号 201580046306.3

T·塔纳维斯基 P·佩里奥瑞里斯

(22)申请日 2015.08.25

(74)专利代理机构 北京市金杜律师事务所

(30)优先权数据

62/044,090 2014.08.29 US

11256

14/515,382 2014.10.15 US

代理人 王茂华 辛鸣

(85)PCT国际申请进入国家阶段日

(51)Int.Cl.

H04L 29/08(2006.01)

2017.02.27

(86)PCT国际申请的申请数据

PCT/US2015/046622 2015.08.25

(87)PCT国际申请的公布数据

W02016/032986 EN 2016.03.03

(71)申请人 微软技术许可有限责任公司

地址 美国华盛顿州

(72)发明人 O·纳诺 I·J·G·d·桑托斯

权利要求书2页 说明书18页 附图5页

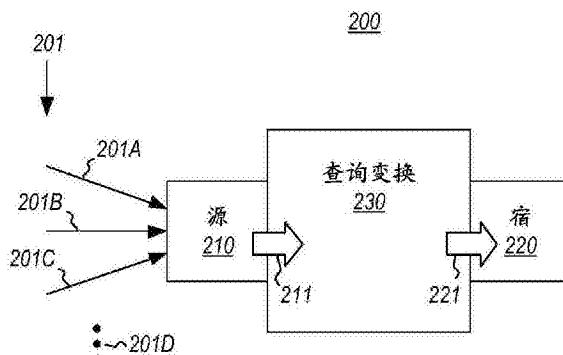
E·亚科楚林 L·诺维克

(54)发明名称

事件流变换

(57)摘要

制定对一个或多个输入事件流的变换以生成一个或多个输出事件流。因此，变换可以被认为是对一个或多个原始输入事件流的查询。事件查询包括表示在特定执行上下文中可用的输入事件流的事件流源表示。事件查询还包括变换模块，该变换模块标识要对执行上下文中的输入事件流执行的变换集。一旦查询被正确地形成，执行模块然后就可以使得对一个或多个指定的输入事件流执行变换以生成输出事件流。



1. 一种计算机程序产品,包括其上具有计算机可执行指令的一个或多个计算机可读介质,所述计算机可执行指令当被一个或多个处理器执行时,使得一个或多个所述处理器实现用于实例化和执行以下各项的系统架构:

事件流源表示,其表示在执行上下文中可用的事件流;

变换模块,其标识要对由所述变换模块表示的一个或多个输入事件流执行的一个或多个变换的变换集,所述变换模块通过参考所述事件流源表示来将所述事件流标识为事件流之一,所述变换模块通过参考所述事件流源表示来将所述事件流标识为所述输入事件流之一;以及

执行模块,其解译所述变换模块以使得在所述执行上下文中对标识出的所述输入流执行所述变换集,以生成作为结果的事件流。

2. 根据权利要求1所述的计算机程序产品,其中所述计算机可读介质的所述计算机可执行指令还包括:

创作模块,其被配置为允许作者创作所述事件流源表示和所述变换模块。

3. 根据权利要求1所述的计算机程序产品,其中所述计算机可读介质的所述计算机可执行指令还包括:

上下文标识模块,其向所述执行模块标识所述执行上下文,从而使得所述执行模块能够使得所述变换集在所述执行上下文中被执行。

4. 根据权利要求1所述的计算机程序产品,其中所述计算机可读介质的所述计算机可执行指令还包括:

发起控件,其被配置为在检测到用户输入时发起所述执行模块。

5. 一种用于通过对一个或多个输入事件流执行变换集来使得作为结果的事件流被生成的计算机实现的方法,所述计算机实现的方法通过一个或多个处理器执行用于所述计算机实现的方法的计算机可执行指令而被执行,并且所述计算机实现的方法包括:

访问表示在执行上下文中可用的特定事件流的事件流源表示,

变换模块标识要对由所述变换模块表示的一个或多个输入事件流执行的一个或多个变换的变换集,所述变换模块通过参考所述事件流源表示来将所述特定事件流标识为所述一个或多个输入事件流中的一个输入事件流;以及

执行模块解译所述变换模块以使得在所述执行上下文中对标识的所述一个或多个输入事件流执行所述变换集,以生成所述作为结果的事件流。

6. 根据权利要求5所述的计算机实现的方法,其中所述特定事件流是当代事件流,并且其中所述事件流源表示是当代事件流源表示。

7. 根据权利要求6所述的计算机实现的方法,还包括:

访问表示在执行上下文中可用的历史事件流的历史事件流源表示。

8. 根据权利要求5所述的计算机实现的方法,其中所述一个或多个变换包括被应用于所述特定事件流的过滤操作。

9. 根据权利要求8所述的计算机实现的方法,其中所述一个或多个变换包括用于生成所述作为结果的事件流的事件的事件生成逻辑,所述事件生成逻辑使用经过滤的所述特定事件流作为输入。

10. 根据权利要求5所述的计算机实现的方法,其中所述一个或多个变换包括用于生成

所述作为结果的事件流的事件的事件生成逻辑,所述事件生成逻辑使用所述特定事件流作为输入。

事件流变换

背景技术

[0001] 事件是与一个或多个时间戳相关联的一段数据。事件流是事件的流。事件源可以接收事件，按照时间戳对它们排序，并提供有序的事件流。存在用于处理事件流的各种常规机制。每个都涉及对事件流的变换的表达和执行。然而，由于并行处理和事件的异步性质，流域自然是复杂的。因此，新用户针对事件流进行查询的学习曲线通常是非常陡峭的。

[0002] 目前，在事件处理中表示数据变换有两种主要方法：域特定语言（DSL）和通用编程语言（GPPL）。DSL通常采用某种形式的类SQL语言，其具有处理时间维度的附加能力。DSL提供了以高度抽象描述查询的声明性方式。此外，DSL类似于SQL，以帮助减少学习曲线，并且使得甚至非开发人员能够编写查询。DSL的主要问题是用户定义的扩展和与应用的集成的困难，与应用程序的集成通常以GPPL被编写。直接以GPPL编写查询可以使得能够更顺利地与使用数据的应用集成，但需要作者知道GPPL。

[0003] 在这里所要求保护的主题不限于解决任何缺点或仅在诸如上述的那些之类的环境中操作的实施例。而是，提供该背景仅是为了说明其中可以实践在这里所描述的一些实施例的一个示例性技术领域。

发明内容

[0004] 在这里所描述的至少一些实施例涉及制定对一个或多个输入事件流的变换以生成一个或多个输出事件流。因此，变换可以被认为是对一个或多个原始输入事件流的查询。事件查询包括表示在特定执行上下文中可用的输入事件流的事件流源表示。事件查询还包括标识要对执行上下文中的输入事件流执行的变换集的变换模块。一旦查询被正确地形成，执行模块然后就可以使得对一个或多个指定的输入事件流执行变换以生成输出事件流。

[0005] 在一些实施例中，可用于表达输入事件流的编程抽象是相同的，而无论输入事件流如何。例如，相同的编程抽象可以用于指定当代事件流以及历史事件流。编程抽象可以允许将当代事件流与同一事件流的历史版本结合。

[0006] 提供本发明内容以便以简化形式介绍将在以下详细描述中进一步描述的一些概念。这一发明内容并不旨在标识所要求保护的主题的关键特征或必要特征，也并不旨在用于帮助确定所要求保护的主题的范围。

附图说明

[0007] 为了描述可以获得本发明的上述和其他优点和特征的方式，将通过参考在附图中被图示的其特定实施例来呈现对在上面简要描述的本发明的更具体的描述。应当理解，这些附图仅描绘了本发明的典型实施例，并且因此不应被认为是对其范围的限制，将通过使用附图、利用附加特征和细节来描述和说明本发明，在附图中：

[0008] 图1图示了示出模型的创作环境方面的事件处理编程模型；

[0009] 图2图示了示出模型的执行环境方面的事件处理编程模型；

- [0010] 图3图示了用于使得变换组件准备以用于执行环境中的操作的方法的流程图；
- [0011] 图4图示了图示变换过程的流程图。
- [0012] 图5图示了被认为是在10分钟的滚动窗口上的平均值的事件流；
- [0013] 图6图示了与两个事件流(一个当代和一个历史)的结合相关联的事件流；以及
- [0014] 图7图示了在这里所描述的原理所可以操作于的示例计算系统。

具体实施方式

[0015] 在这里描述了一种全面但易于使用的事件处理编程模型，其中可以对一个或多个事件流表达和执行查询，以由此产生一个或多个作为结果的事件流。事件是与一个或多个时间戳相关联的一段数据。在这里所描述的至少一些实施例涉及制定对一个或多个输入事件流的变换以生成一个或多个输出事件流。因此，变换可以被认为是对一个或多个原始输入事件流的查询。事件查询包括表示在特定执行上下文中可用的输入事件流的事件流源表示。事件查询还包括标识要对执行上下文中的输入事件流执行的变换集的变换模块。一旦查询被正确地形成，执行模块然后就可以使得对一个或多个指定的输入事件流执行变换以生成输出事件流。

[0016] 在一些实施例中，可用于表达输入事件流的编程抽象是相同的，而无论输入事件流如何。例如，相同的编程抽象可以用于指定当代(例如，实况)事件流以及历史事件流，甚至可能在同一查询中。考虑以自然语言被表达的以下查询：“当一天时间范围中的平均能量消耗比一年前同一天消耗的能量高50%时触发警报”。传统上，这样的情形通过用户编写复杂的定制逻辑而被解决。编程抽象可允许将当代事件流与同一事件流的历史版本结合。因此，描述了对事件流的类型不可知(agnostic)的统一事件查询模型(只要事件流在执行上下文中可用)。

[0017] 对事件流查询变换集进行编程的经验可以基本上相同，而与执行上下文无关(例如，无论执行上下文是在本地计算系统上、在远程服务器上还是在云中)。执行上下文使用上下文标识来获得标识执行上下文的必要信息，并使得查询变换集在指定的执行上下文中被运行。在执行事件查询处理之处的这种灵活性允许计算与生成一个或多个输入事件流的地方和/或消耗一个或多个输出事件流的地方更靠近地发生。因此，事件处理编程模型的实施例对执行上下文是不可知的(只要执行模块具有足够的信息来在该执行上下文中标识和部署查询)。

[0018] 在这里描述的事件处理模型的实施例具有对用户的渐进学习曲线，同时仍然是表达性的和可扩展的。由于并行处理和事件的异步性质，流域自然是复杂的。因此，新用户针对事件流进行查询的学习曲线通常是非常陡峭的。为了易于使用，事件处理编程模型限制了用户所暴露于的应用程序接口(API)中的概念/抽象的数量，并且清楚地定义了这样的抽象的语义。同时，为了实现可表达性，这些抽象是可组合的，并且允许用户从原始内置事件处理计算构建更高级的查询逻辑。所描述的事件处理编程模型定义了可以按照域特定语言(DSL)和通用编程语言(GPPL)被表示的抽象。

[0019] 事件处理编程模型被描述为实现用于事件流的时间模型。在下面描述的应用程序接口(API)中，每个事件具有表示事件何时被生成的相关联的单个时间戳。然而，在这里描述的更一般的原理还可以支持序列模型、间隔/快照模型或任何其他事件处理时间模型。在

这里所描述的原理还可以可适用于允许在事件处理时间模型之间的清晰定义的用户体验间隔的情况下与多个事件处理时间模型(诸如时间模型、间隔/快照模型等)的交互的事件处理编程模型。

[0020] 事件处理编程模型还明确地定义时间的概念并且在虚拟时间(即,应用时间)中工作。许多监视和遥测情形涉及对具有与挂钟时间(即,系统时间)不同的其自己的时间线的日志记录的分析。这是自然语言形式的这样的查询的一个示例:“当在一小时时间范围内没有响应时提醒所有情况”。这样的查询不涉及对系统时间的评估。

[0021] 事件处理编程模型通过以它们可以被运行查询的底层服务自动并行化的方式定义事件处理计算来提供可表达性。因此,事件处理编程模型可以通过提供方便的分组抽象来弹性地横向扩展计算。

[0022] 首先,将介绍时间数据处理模型的概念。考虑以下情形。假设存在从数据中心中的不同机器获取的一组性能计数器值。在这种情形下,系统将发现性能计数器值随时间的异常。性能计数器值的时间是读取该值时的时间,而不是处理对应的事件时的时间。因此,事件处理编程模型引入时间模型。

[0023] 这一时间模型基于虚拟时间,并且被建立在复杂事件检测和响应(CEDR)代数的子集上并且基于虚拟时间。时间模型中的所有运算符都指的是每个事件中的虚拟时间时间戳。在一个实施例中,为了简化用于用户的语义,时间模型中的事件是点事件,因为事件各自具有恰好一个时间戳并且携带用于一个时间点的有效载荷。点事件的示例包括仪表读取、电子邮件的到达、用户Web点击、股票报价或对日志的录入。

[0024] 这一时间模型中的所有运算符可以是确定性的,并且运算符的输出可以是可重复的。在一个示例中,查询接收被称为源的事件流作为其输入,使用该事件流输入执行变换(例如,过滤、选择等等)(即,对该输入事件流执行查询),并生成被称为宿(sink)的作为结果的数据流。源从各个位置接收各种事件,并且提供作为结果的事件流。在一些实施例中,源还可以对事件执行更多一些以产生事件流,诸如按时间上的时间对事件排序。

[0025] 图1图示了示出模型的创作环境100方面的事件处理编程模型。创作模块141允许作者创作事件流源表示110、事件流宿表示120、变换模块130和上下文标识模块150。

[0026] 在创作之后,发起命令160可以被发起,其自动地使得在上下文标识模块150中标识的执行上下文中执行变换模块130。例如,图2图示了在执行环境200中执行的可执行变换组件230。

[0027] 查询变换模块130表示要对输入事件流执行以由此生成输出(或作为结果的)事件流的特定查询。具体地,查询变换模块130可以表示要对一个或多个输入事件流执行以由此生成一个或多个输出事件流的一个或多个变换的变换集。变换操作的示例可包括对一个或多个事件流源中的每个事件流源进行过滤以使得仅允许事件中的一些事件通过。其他变换可包括使用事件流源事件而被执行以由此生成一个或多个作为结果的事件的逻辑。这样的逻辑可以直接对事件流源执行,或者仅对通过一个或多个过滤器而被过滤的事件执行。

[0028] 在执行环境中,对应的查询变换组件230接收事件流源210作为输入。事件流源210接收事件201(诸如如由椭圆201D表示的潜在的许多其它当中的事件201A、201B、201C),并提供结果作为输入事件流211。查询变换组件230处理输入事件流以由此执行在查询变换模块130中表示的查询。作为处理的结果,查询变换组件230生成被提供给事件流宿220的输出

(作为结果的)事件流221。输出事件流221表示当查询被应用于输入事件流211时与查询变换组件相关联的查询的查询结果。事件流源210和事件流宿220自身可以被认为是事件流。因此,事件流宿可以稍后用作用于笔筒的查询处理的事件流源。

[0029] 在创作时,作者使用创作模块141来制定表示在执行上下文200中可用的事件流源210的事件流源表示110。创作模块141还用于制定查询变换模块,查询变换模块标识要对由变换模块表示(或与变换模块相关联)的一个或多个输入事件流执行的一个或多个变换130的变换集。变换模块130可以引用事件流源表示110或者与事件流源表示110相关联。创作模块141还可以用于制定表示在执行上下文200中可用的事件流宿220的事件流宿表示120。变换模块130可以引用事件流宿表示120或可以与事件流宿表示120相关联。创作模块141还可以用于制定标识执行上下文的上下文标识模块150。

[0030] 执行上下文150包含用于在上下文中可用的事件流源和事件流宿的定义。通过扩展,可用操作的集合也可以从可用的源和宿定义被推断。例如,如图2中所示,执行上下文200包括事件流源110的实例210,以及事件流宿120的实例220。上下文标识模块150可以隐式地标识上下文(例如,在没有明示上下文标识的情况下作为缺省上下文),或者明确地标识上下文。例如,缺省上下文可以简单地仅是在发起查询的同一系统上执行查询变换组件130的实例。仅作为示例,执行上下文可以是本地计算系统、云计算环境或任何其他执行环境。

[0031] 创作环境100还包括执行模块161,其在检测到一个或多个条件和/或事件时解译变换模块130以促进对应的查询变换组件230在执行上下文200中被执行。例如,执行模块161可以包括被配置为在检测到用户输入时发起执行模块161的发起命令或控件160(或“管理API”)。发起命令160自动地使得事件处理编程模型在执行上下文200中被运行。例如,如果在内容标识组件150中标识的上下文指示查询将在本地被运行,则查询变换组件230可以在本地计算系统上在本地被运行。如果在上下文标识组件150中标识的执行上下文是云计算环境或某种类型的远程服务器,则查询变换模块130将被上传到该云计算环境或其他类型的远程服务器,实例化查询变换组件230并且然后在该远程环境中被运行。注意,其中运行查询变换组件130的实例230的上下文可以在上下文标识组件150内被标识,而不改变查询变换组件130自身的任何查询逻辑。

[0032] 图3图示了用于使得变换组件230准备以用于执行环境200中的操作的方法300的流程图。可以响应于作者发起图1的发起命令160来执行方法300。

[0033] 事件流源表示被访问(动作301A)。例如,在图1中,事件流源表示110被访问。对仅一个事件流源表示没有限制。因此,另一事件流源呈现可被访问(动作301B)。另一事件流源表示还表示在执行上下文200中可用的特定事件流(尽管未在图2中示出)。作为一个示例,事件流源110可以是当代事件流,而附加事件流源(未示出)可以是历史事件流。变换可包括执行输入当代事件流的历史时移版本。

[0034] 变换模块也被访问(动作302)。例如,在图1中,变换模块130可被访问。事件流宿表示也与上下文标识组件(动作304)一起被访问(动作303)。例如,执行模块161可以访问变换模块130、事件流宿表示120和上下文标识组件150。

[0035] 然后,执行模块解译(动作310)变换模块,以使得在执行上下文200中对标识的一个或多个输入事件流执行变换集以生成作为结果的事件流。为此,执行模块161使用上下文

标识模块150内的上下文标识。例如,执行模块获得变换组件的实例(动作311),诸如变换组件230。执行模块还将变换组件耦合到由事件流源表示(诸如事件流源表示110)标识的任何事件流源(动作312)(诸如事件流源210)。执行模块还将变换组件耦合到由事件流源表示(诸如事件流源表示120)标识的任何事件流宿(动作313)(诸如事件流宿220)。变换组件230然后可被执行(动作314)。

[0036] 回到图1,创作模块141包括编程抽象的集合140,其可被作者用来制定事件流源表示110、事件流宿表示120、查询变换模块130和上下文标识组件150。例如,抽象140被图示为包括抽象141至144,但是省略号145表示可以存在由程序员用来创作模块事件流表示110和120以及模块130和150的任何数量的抽象。

[0037] 在这里所描述的实施例中,可能除了仅在对事件流源110本身的标识中,和/或除了仅在对事件流宿120本身的标识中,在没有对查询变换模块130的任何基本改变的情况下,可以改变事件流源110和事件流宿120的身份和类型。

[0038] 例如,在图1的示例中,抽象140之一可以用于标识事件流源(诸如事件流源110)。以下是被称为ITemporalSourceDefinition接口的抽象的示例。当然,在这里描述的特定抽象的任何名称是任意的。更重要的是抽象的功能。以下是使用ITemporalSourceDefinition接口的示例HTTP源定义:

```
ITemporalSourceDefinition<int, CpuReadings> input =  
    service.GetTemporalHttpSource<int, CpuReading>(  
        [0039]         "http://europe.azure.com/cpu-readings", // 名称  
        e => e.MachineId, // 键选择符  
        ...);
```

[0040] 时间源(诸如刚刚定义的HTTP源)产生按虚拟时间排序的事件。它们可以具有一些特定的参数,这取决于源的类型。例如,在某些类型的时间事件流源中,队列名称可能具有相关性。其他类型的时间事件流源可能具有时间戳选择或分组键选择符。

[0041] 在分布式情形中,事件流源(例如,事件流源210)可能无序地接收事件(例如,事件201)。事件流源跟踪虚拟时间。当事件被接收到时,事件流源确定与事件相关联的点时间,并相应地使应用时间前进。然而,由于事件有时无序地到达,具有当前应用时间之前的时间戳的一些事件可以到达事件流源。换句话说,在过去通过推算虚拟时间对事件加上时间戳。

[0042] 想象以下情况:存在具有时间戳T1、T2和T3的三个事件。根据应用时间推算,时间戳T1在时间戳T3之前的时间戳T2之前。如果事件采用到事件流源的不同路由,则事件可能到达,从而使得具有时间戳T1和T3的事件首先到达,并且仅在此之后,具有时间戳T2的事件到达。如果事件流源在具有时间戳T3的事件到达时将虚拟时间移动到T3—不清楚对具有时间戳T2的事件该做什么。

[0043] 存在时间事件流源的两个参数帮助用户应对事件无序地到达的问题。一个参数是无序策略,其可以是“调整”或“丢弃”。如果应用调整策略,则具有小于当前虚拟时间的时间戳的所有事件将被用当前虚拟时间重新加时间戳。在应用丢弃策略时,所有这样的事件将被丢弃。然而,事件流源可能施加其他类型的无序策略。另一无序策略的示例是“中止”策

略,其中无序到达的事件使得对作为结果的事件流的查询被中止。

[0044] 使得能够容忍无序事件的另一参数是标点生成设置。它们定义服务应如何推进输入的虚拟时间。在事件无序地到来但用户不想用无序策略对其重新加时间戳的情况下,用户可以显式地设置如何使用前进虚拟时间函数来在源上生成标点。前进虚拟时间函数将当前虚拟时间和所有被缓冲事件的列表作为输入,并返回新的虚拟时间。例如,为了在虚拟时间中容忍10秒的延迟,用户可以提供以下函数:

[0045]

```
settings.AdvanceTime =  
(currentTime, bufferedEvents) =>  
    bufferedEvents.Last().Timestamp - bufferedEvents <  
    TimeSpan.FromSeconds(10)  
    ? current  
    : bufferedEvents.Last().Timestamp)
```

[0046] 抽象140的另一示例是用于标识和潜在地定义作为结果的事件流所将去往的事件流宿221的抽象。在这里的示例中,这样的抽象被称为ITemporalSinkDefinition接口。以下是使用ITemporalSinkDefinition接口的事件流宿定义的样本:

[0047]

```
ITemporalSinkDefinition<string, int> storage =  
    service.GetTemporalAzureQueueSink<string, CpuAverage>(  
        "http://europe.azure.com/azurequeuesink",  
        "connectionString");
```

[0048] 与查询变换模块130本身相关联的查询可以使用将查询变换模块130与事件流源110相关联的抽象而被定义,从而使得当查询变换组件130的实例230被执行时,实例接收输入来自对应的事件流源210的输入事件流211。可以用于这样做的抽象140的一个示例在这里被称为“From(来自)”方法。

[0049] 与查询变换模块130相关联的查询也可以使用将查询变换组件230与事件流宿210相关联的抽象而被定义,从而使得当查询变换组件230的实例被执行时,实例将输出事件流221提供给对应的事件流宿220。可以用于这样做的抽象140的一个示例在这里被称为“To(去往)”方法。

[0050] 以下是示例端到端查询,其使用各种抽象140来定义事件源、定义事件流宿、定义查询逻辑以及将查询逻辑连接到定义的事件源和事件宿。

[0051]

```

var httpSource = service.GetTemporalHttpSource<string, CpuReading>(
    "http://europe.azure.com/cpu-readings",
    reading => reading.Region,
    AdvanceTimePolicy.Adjust);

var avgCpuStream = service.GetTemporalStream<int,
    AvgCpuByMachine>(
    "http://europe.azure.com/AvgCpu");

var q = TemporalInput.From(httpSource)
    .RegroupBy(e => e.MachineId)
    .Where(e => e.Payload.CpuUsage > 20)
    .Avg(new TumblingWindow(TimeSpan.FromMinutes(5)),
        e => e.Payload.CpuUsage,
        (key, avg) => new AvgCpuByMachine()
            { Avg = avg, MachineId = key })
    .To(avgCpuStream);

service.StartQuery("streamAggQuery", q);

```

[0052] 在这一示例中，事件流源是HTTP源，并且查询逻辑涉及计算每台机器的平均CPU使用率：

[0053] 在这里描述的事件处理编程模型允许它的用户表达对数据的低延时、增量的、缩放的和可靠的计算。现在将描述数据的处理和数据的计算。如上所述，数据被组织成事件流。每个事件流描述随时间推移而附加的数据的潜在无限集合。例如，每个事件由数据和相关联的时间戳组成。对数据的计算由查询表示，其可以被表示为图。因此，当用户定义查询时，用户使用基本构造块（例如，事件源和宿定义）来定义数据变换的图结构，并且还定义相关联的变换本身。这样的查询定义可以被声明性地执行。

[0054] 例如，图4图示了流程图，该流程图图示了变换过程的。在图4中，用户取得性能计数器源定义，应用过滤器（“where（其中）”）变换和选择变换，并将所有内容输出到队列中。当这一图被提交给适当上下文（如由上下文标识组件150所标识的）以用于执行（例如，通过发起命令160的执行）时，该图在用于计算的服务内被执行。服务解译该图以在运行时执行该图。服务从性能计数器取得运行时数据，对其进行过滤和投影并输出给队列。这一图的运行时实例化是查询。因此，事件处理编程模型由映射到运行时实体的定义实体组成。

[0055] 定义实体用于声明性地描述数据流和在它们上的计算。这些概念只定义行为—它

们不具有任何运行时成分。例如，“filter(过滤器)”的定义规定传入数据将根据某些谓词被过滤。为了比你编程语言，定义类似于类。查询的组成在客户端侧上使用事件处理编程模型发生，但是在事件处理引擎上执行查询的实际实例化和执行(其可以是任何方式和在任何设备上)。

[0056] 因此，运行时实体是实现定义的运行的实例。运行时实体由事件处理引擎或服务托管和执行，并且进行数据的实际处理。它们类似于编程语言中的对象。例如，在运行时期间，存在事件流源110的一个实例在操作，查询变换链130的一个实例在操作，以及事件流宿130一个实例在操作。

[0057] 这一描述现在将提供可以如何实现事件处理编程模型以使用C#编程语言进行示例查询的示例。在这一示例中，假设存在一个HTTP端点，其中来自房间的所有传感器发送它们的当前温度。在这一情形下，如果来自任何传感器的1分钟的窗口内的平均温度超过90摄氏度，则应用应产生报警。另外，警报应被写入Azure队列中。以下是查询的对应的C#示例，其描述查询逻辑、要被使用的事件流源、要被使用的事件流宿、查询将被运行于的上下文以及对应的发起命令。提供行编号和适当的空格以用于后续的参考和组织。

[0058]

```
1: var service = new ServiceContext(serviceUri);

2: var source = service.GetHttpSource<int, Temperature>(ingressUri, e
=> e.SensorId);

3: var sink = service.GetAzureQueueSink<int, Alarm>(connectionString,
4:                                         queueName);

5: var q = TemporalInput.From(source)
6:     .Average(new TumblingWindow(TimeSpan.FromMinutes(1)),
7:             reading => reading.Temperature)
8:     .Where(temperature => temperature > 90)
9:     .Select(temperature => new Alarm())
10:    .To(sink);

11: service.StartQuery("AlarmQuery", q);
```

[0059] 在这一该示例中，以五个逻辑步骤实现查询。在行1中，标识查询要被运行在其中的上下文。这是图1的上下文标识组件150的示例。在这一示例中，使用ServiceContext方法来获得上下文，该方法表示图1的抽象140的示例。

[0060] 在行2中，事件流源被定义。在这一示例中，事件流源是HTTP事件流源。然而，事件

处理引擎可能能够处理各种各样的事件流源。因此，抽象140可包括对不同的事件流源类型的内置支持。当定义源时，用户可以对事件类型（在这种情况下为温度）进行选择。可以存在特定于具体源类型的一些其他参数（在该示例中为ingressUri）。行2的调用中的最后参数是分组标准，其是图1的抽象140的另一示例。查询可以使用如由某个键标识的这样的分组标准，因此在查询中被定义的所有操作将被分别应用于群组的每个成员。在上面的示例中，查询按照SensorId被分组。这意味着average（平均）/where（其中）/select（选择）操作（行5-10）将被独立地应用于来自每个传感器的数据。行2是图1的事件流源110的示例。

[0061] 在行3-4中，事件流宿被定义。事件流宿定义输出数据（例如，输出事件流）将在哪里结束。在上面的示例中，行3-4指定将把数据写入Azure队列中的宿。行3-4是图1的事件流宿120的示例。

[0062] 在行5-10中，存在查询逻辑本身，其将数据从源变换到宿。查询包括彼此组合的不同源/宿定义。在该示例中，首先计算1分钟的窗口内的平均值。之后，温度高于90度的所有事件被滤过。对于任何这样的事件，使用“select”变换来创建报警。行5-10是图1的查询变换组件130的示例。

[0063] 在行11中，在事件处理服务（行11）内运行查询。最后操作将在事件处理服务中创建具有指定名称的查询的运行的实例。为了向服务提交查询，用户使用在行1中获取的服务上下文。因此，行11是图1的发起命令160的示例。

[0064] 即使该示例以C#被给出，但是事件处理编程模型可以是语言不可知的。例如，模型可以提供按照不同语言（包括C#、JavaScript等）的SDK。

[0065] 在先前部分中介绍的事件源定义的概念允许事件处理编程模型统一实况数据和历史数据以及源于设备的数据和源于云的数据之间的体验。从编程模型的角度来看，数据是在运行时生成（如在HttpSource的情况下）还是从表中取得（如在AzureTableSource的情况下）没有区别—它是具体源定义的细节。源于设备和源于云的定义也是如此。统一不同类型的数据源，编程模型允许用户专注于查询的业务逻辑，其即使在数据源将来会改变的情况下也保持不变。例如，在图1中，用户可以集中于创作查询变换组件130，而基本抽象140可以用于允许用户定义适当的源、宿和上下文标识。

[0066] 回到前面的情形，代替创建HTTP源（上面的示例中的行2），并且代替将所有数据推送给云并且在那里分析它（上面的示例中的行1和11），用户可以使用将在设备上被实例化的内置设备源定义（例如，下面示例的行1中的service.GetDeviceSource）。代码的示例可能如下：

[0067]

```

1: var deviceSource = service.GetDeviceSource<Temperature>(...);

2: var sink = service.GetAzureQueueSink<Alarm>(connectionString,
3:                                         queueName);

4: var q = TemporalInput.From(deviceSource)
5:     .Average(new TumblingWindow(TimeSpan.FromMinutes(1)),
6:             reading => reading.Temperature)
7:     .Where(temperature => temperature > 90)
8:     .Select(temperature => new Alarm)
9:     .To(sink);

10: service.StartQuery("AlarmQuery", q);

```

[0068] 数据变换逻辑(行-10)根本没有改变。只存在两个主要的差别。首先，行1的源定义已经稍微改变(与先前示例的行2相比)，以将源标识为对设备可用的。其次，移除了先前示例的行1，从而导致在缺省上下文(例如，在对设备可用的事件处理服务上)中执行行10的发起命令。服务本身然后可以决定如何更好地分发和执行查询。对于上面的示例，假设平均值的计算、对温度的过滤和报警的生成可以在设备上发生，同时仅在必要时将报警事件传播到云。以相同的方式，查询计算的一些部分可以被推送给表示关系数据库、映射/减少执行引擎等的源定义。

[0069] 事件处理编程模型原理和API

[0070] 这一节讨论事件处理编程模型背后的主要原理，并更详细地介绍示例应用程序接口的元素。

[0071] 对于在虚拟时间中操作的事件计算，应当存在将应用时间向前移动的方式。通常，在事件处理引擎中，虚拟时间的前进由被称为标点的特殊事件来传达。在查询处理期间，虚拟时间由标点事件驱动。它们用于通过向服务告知时间线的某些部分对于这一特定输入将不再改变来提交事件并将计算出的结果发布给查询输出：通过在时间T处使标点入列，输入承诺不产生将影响T之前的时段的任何后续事件这。这意味着，在标点在输入中已经入列之后，所有其他事件的虚拟时间应不小于入列的标点。这一规则的违反被称为“标点违规”。可以存在不同的策略来处理标点违规(即，丢弃事件、中止查询或调整事件时间)。

[0072] 在事件处理编程模型中，所有事件也可以被认为是标点，因此用户不必显式地将标点插入到数据流中。然而，在一些情况下仍然可以使用标点，诸如当用户想要冲刷有状态的运算符的当前状态时。例如，考虑图5的以下示例，其中考虑在10分钟的滚动窗口中的平均值。

[0073] 在图5中,在运算符看到具有属于时段T2的虚拟时间的下一事件之前,不允许运算符为时段T1产生任何输出。如果事件不是均匀分布的并且新事件仅在时段T2结束时到来,则这可能导致长等待时间。为了应对这种情况,用户可以发出向前移动时间的标点符号,并允许运算符关闭当前窗口(右图)。

[0074] 除了移动虚拟时间之外,还可能存在其他类型的标点,诸如“出错”和“完成”标点。如果在事件处理期间发生错误,则将“出错”标点传送给下游运算符。当预计不再有此输入的事件时,发出“完成”。

[0075] 事件处理编程模型在运算符中提供可重复性。如果对于相同的初始状态和相同的输入,运算符确定性地产生相同的输出,则运算符是可重复的。非重复性有两个主要原因:运算符的非确定性和不能提供相等的输入(如在取决于物理时间的运算符的情况下,诸如定时器)。非确定性运算符可以被进一步细分为内在非确定性的运算符(它们的状态取决于某种形式的随机函数/外部改变状态)和输入非确定性的运算符(输出取决于不同输入之间的事件的次序)的运算符。

[0076] 分组

[0077] 事件处理编程模型使用分组的概念来提供事件处理计算的弹性缩放。例如,以下查询使用分组:

```
TemporalInput
    .From(table)
    .RegroupBy(e => e.MachineId)
[0078]    .Avg(new TumblingWindow(TimeSpan.FromMinutes(5)),
        e => e.CpuUsage)
    .To(anotherTable);
```

[0079] 事件处理编程模型中的事件流可以使用用户指定的分组键被细分为不相交的组。因此,每个事件流是一组子流,每个键的唯一值一个。查询中的所有运算符被独立地应用于每个子流,并且不共享状态。在上面的查询中,用户将接收用于每个子流的独立结果——例如,每个机器每个五分钟时间窗口的一个平均CPU使用值。

[0080] 在这一示例中,分组的概念由在这里被称为RegroupBy运算符的抽象140之一的示例公开,RegroupBy运算符采用键选择符作为参数。如果未指定组标准,则可将数据流视为包含所有事件的单个逻辑组。

[0081] 为了方便起见,一些事件流源还可以将组函数作为参数。以下是这样的事件流源定义的示例:

[0082] var table=service.GetAzureTableSource<int,CpuReading>(...,e=>e.MachineId);

[0083] 为了对所有事件进行操作,用户可以利用返回常数的键选择符对数据流重新分组。

[0084] 示例API

[0085] 在这一部分中,详细描述了主API构造元素(其是图1的抽象140的每个示例)的示

例。也就是说，这只是示例。

[0086] 定义实体概要

[0087] 源：定义用于产生事件流的机制或计算。例如，`HTTPSourceDefinition` 定义了一种用来根据被发送给特定 HTTP URL 的所有请求来创建事件流的机制。

[0088] 宿：定义用于消费事件流的机制或计算。例如，`AzureBLOBSinkDefinition` 定义了一种将事件流存储到 Azure BLOB 中的机制。

[0089] `QueryDefinition`：定义从源获取所有事件并将它们放在宿中的计算。例如，从 `HTTPSource(ingressUri)` 取得所有事件，并将它们放在 `AzureBLOBSink(blobUri, storageAccountConnectionString)` 中。

[0090] 运行时实体概要

[0091] 尽管这高度依赖于消耗查询的底层事件处理服务：

[0092] 查询：根据 `QueryDefinition` 被创建，它表示运行时计算，该运行时计算从源（从 `SourceDefinition` 实例化）取得所有事件，并将它们放入宿（从 `SinkDefinition` 实例化）。

[0093] 流：作为服务中的设施，它是从由事件处理服务托管和控制的流实例化的事件流。该服务向流提供了用来消耗或产生事件的源和宿。

[0094] 事件和流

[0095] 事件是记录：键值/关联集合。值可以是任何原始类型 (.net/JSON)、数组或关联集合（映射/字典）。

[0096] 流提供对有序和时间事件的确定性：给定相同的有序输入和相同的查询，其将总是生成相同的有序输出。

[0097] 事件流的特征在于其时间和有序性质。用于流的源实现 `ISource`。它可以用用于分组的选择符表达式被配置。如果缺少这些元素中的一个，则底层源将为该特定元素使用其缺省行为。下面是示例 `ISource` 定义。

[0098] //取得HTTP源定义作为事件流

```
1: var input = svc.GetHttpSource<int, CpuReading>(
    "http://europe.azure.com/cpu-readings",
    e => e.MachineId,
    e => e.TimeStamp,
    TimeSpan.FromSeconds(10));
```

[0100] 用于事件流的宿（实现 `ISink`）保留事件流的时间和次序属性。

[0101] //得到BLOB宿定义作为事件流

```
1: var storage = svc.GetAzureBlobSink<string, CpuAverage>(
    "https://myaccount.blob.core.windows.net/mycontainer/myblob",
    "DefaultEndpointsProtocol=https;AccountName=...;AccountKey=...");
```

[0105] 事件流可以被细分为组。组由分组键指定。实际上，在这种情况下，事件流是一组子流，键的每个唯一值一个。计算被独立地应用于每个子流。重新分组构造可以用于改变分组。为了跨组在所有事件上操作，用户可以将所有组重新分组为一个组。

[0106] 服务上下文

[0107] 查询被部署在服务内部。该服务能够处理查询定义和对诸如启动/停止查询之类的查询执行管理操作。用于初始化服务上下文的简单姿势可以如下：

[0108] //从事件流得到源和宿

[0109] 1:ServiceContext svc=new ServiceContext(serviceUri);

[0110] From(来自)/To(去往)

[0111] 从取得ISource作为参数并返回定义可用于事件流的运算符集合的IGroupedStream接口的TemporalInput.From(源)姿势开始定义查询。IGroupedStream表示分割的事件流。当利用ISink作为参数在IGroupedStream上调用To姿势时，返回查询定义。

[0112] SteamR运算符

[0113] Filter(where)

[0114] 基于应用于每个单独事件的用户定义的谓词表达式，确定应该保留哪些事件或从分组的流移除哪些事件。

[0115] TemporalInput.From(httpSource)

[0116] .Where(e=>e.Payload.CpuUsage>50)

[0117] .To(blobSink);

[0118] Projection(选择)

[0119] 将用户表达式应用于每个单独的事件并且产生可能是不同类型的新事件。以下编码示例示出了如何使用过滤和投影(projection)：

[0120]

```
TemporalInput.From(httpSource)
```

```
.Select(e => new CpuReading() { CpuUsage = e.Payload.CpuUsage /  
10 })
```

```
.To(blobSink);
```

Multicast/union (多播/联合)

[0121] 多播运算符创建每个分组流的多个副本(组成员资格/键不被多播修改)。联合运算符将多个分组流组合成单个分组流(组成员资格/键不被修改，具有相同键的事件将以同一组告终)。排序被联合保留。

[0122] Aggregate(聚合)

[0123] 聚合运算符将聚合函数应用到分组流的子集中。可以通过时间属性(例如，窗口持续时间)、每个组中的单独事件的数量(计数窗口)或定义子集的开始和结束的谓词(基于范围或条件的窗口)来定义这一窗口。聚合函数：平均值、最小值、最大值、计数、总和和StdDev。

[0124] 聚合可被设置如下：

[0125]

```
TemporalInput.From(source).Avg(new
    TumblingWindow(TimeSpan.FromSeconds(5)),
        e => e.Payload.CpuUsage,
        (avg) => new CpuReading() {CpuUsage = avg})
    .To(cpu5sSink);
```

[0126] 还支持多聚合计算。API允许将多个聚合函数应用于有效载荷。以下示例图示了将3个聚合函数应用于事件有效载荷。

[0127]

```
TemporalInput.From(source).
    .MultiAggregate(new TumblingWindow(TimeSpan.FromSeconds(30)),
        w => w.Average(e => e.Payload.CpuUsage), // 第一聚合
        w => w.Min(e => e.Payload.CpuUsage), // 第二聚合
        w => w.Max(e => e.Payload.CpuUsage), // 第三聚合
        (key, avg, min, max) => new CpuAggregates() // 投影
    {
        AvgCpu = avg,
        MinCpu = min,
        MaxCpu = max,
        Region = key
    })
    .To(blobRegionEndpoint);
```

[0128]

[0129] TopK (topK)

[0130] TopK是聚合函数,其对形成分组流的子集的事件进行排序,并且产生用于分组流内的每个组的前K个值作为输出。这一子集(或窗口)可以由时间属性(诸如窗口持续时间)、每个组中的单独事件的数量(计数窗口)或定义子集的开始和结束的谓词(范围或条件)来定义。它以与在以上代码片段中应用函数的相同方式被应用。

[0131] Window definition(窗口定义) (*window)

[0132] 窗口基于计数(计数窗口)、基于条件或基于时间属性(诸如窗口持续时间)来缓冲事件子集。然后,可以对这一子集应用任意集合的运算符。滚动窗口运算符的代码样本如下:

[0133]

```
TemporalInput.From(source)
    .MultiAggregate(new TumblingWindow (TimeSpan.FromSeconds
(5)),
    win => win.Average(e => e.Payload.CpuUsage),
    (avg) => new CpuReading ()
    {
        CpuUsage = avg
    }
)
.To(cpu5sSink);
```

[0134] 事件处理编程模型定义了都从窗口运算符继承的4个特定窗口；那些是CountWindow、ConditionalWindow、TumblingWindow和HoppingWindow。

[0135] ReGroup by/Merge (regroupBy/合并)

[0136] Regroupby运算符通过将新的键控函数应用于每个单独的事件来重新定义事件组成员资格。作为输出，regroupby运算符基于新键来产生新的一组的分组流。合并运算符仅仅是基于regroupby的宏，其中键是常量值（结果是使所有事件具有相同的组成员资格，即具有仅一个组的分组流的集合）。

```
TemporalInput.From(cpu5sSrc)
    .RegroupBy(e => e.ClusterID)
    .Avg(new TumblingWindow(TimeSpan.FromSeconds(5)),
[0137]     e => e.Payload.CpuUsage,
        (avg) => new CpuReading (){CpuUsage = avg})
    .To(cpu5sSink);
```

[0138] Join (结合)

[0139] 在其最简单的形式中，其在计数窗口上使两个分组流相关。

[0140] Temporal join (时间结合)

[0141] 基于用户定义的谓词表达式和被应用于一对事件的用户定义的投影表达式，在给定时间窗口（持续时间）内使两个分组流相关。

[0142] Time Travel Join (时间旅行结合)

[0143] 类似于时间结合，因为它使两个分组流相关，具有回顾分组流集合之一的历史的附加能力（也被称为时间旅行）。以下是具有历史和实况流的查询的示例。

IQueryDefinition<int, CpuAlarm> q3Def =

[0144] SteamR.From(cpu5sSrc)

.TimeTravelJoin(SteamR.From(cpu5sSrc), // 右流

[0145]

TimeSpan.FromHours(1), // 应用于右流的有效性

TimeSpan.FromDays(-365), // 应用于右流的时间旅行

(live, hist) => live.Payload.MachineId == hist.Payload.

MachineId && live.Payload.CpuUsage > hist.Payload.CpuUsage, //

结合谓词

(live, hist) => new CpuAlarm()

{

LiveCpuUsage = live.Payload.CpuUsage, HistCpuUsage =

hist.Payload.CpuUsage, MachineId =

live.Payload.MachineId, TimeStamp =

live.Payload.TimeStamp

}) // 结合投影

.To(blobAlarmEndpoint);

[0146] 时间旅行结合是值得更多细节的特殊运算符。为了使能对实时和历史数据的高级分析,事件处理编程模型可以引入具有时间旅行的时间结合。上面的样本表示一个典型情形。

[0147] 这一运算符允许对历史数据回到过去(根据rightTimeTravelShift参数重放事件并修改其时间戳),并且然后通过CEDR时间结合将其与实况流相关。rightDuration参数指定如何改变历史事件的持续时间以能够将它们与来自实况流的点事件结合。时间旅行结合的一个非常有趣的是它由实况流的时间线驱动。

[0148] 在最简单的实现方式中,将从时间的开始读取完整的历史流,并将其与实况流结合。然而,在许多情况下,这是不期望的,因为历史流可以具有大量不必要的数据,并且在实况流和历史流将在时间上对齐之前可能花费相当长的时间。例如,如果历史流包含来自最近10年的数据,则在到达可以实际结合的部分之前将需要预处理9年的数据。

[0149] 图6图示了与两个事件流(一个当代和一个历史)的结合相关联的事件流。代替从开始重放所有事件,向历史流的源传达我们感兴趣的仅有数据回到一年前将是更加高效的。我们感兴趣的时间位置不能在查询的编译阶段确定。仅当结合接收到来自实况流的第一个事件时才发现开始时间位置。

[0150] 映射

[0151] 到目前为止我们一直讨论的用户表面层负责定义简单的标准数据变换(诸如“哪里”,“选择”等),从而向用户隐藏复杂性。此层由客户端侧SDK或DSL表示。它包含特定编程

语言的服务实体的客户端侧代理，并提供用于组合的手段。在该层上，用户能够获取内置的定义并将它们组合到查询中。

[0152] 该层的另一职责是将依赖于语言的SDK实体转换为它们的内部表示——其是语言无关的并且可以被实体组成层理解。该过程被称为规范化。在此过程期间是以某一规范形式映射查询，其稍后可以在边缘云中运行的引擎中被映射。虽然可能存在不同类型的真实事件处理引擎；对于用户都是一样的。

[0153] 本文档描述了事件编程模型编程模型的概念。事件处理编程模型可以统一流处理的不同维度，允许离线和在线分析的方便集成，以及以可扩展和直观的方式在云中和在设备上收集数据。

[0154] 由于各种定义的创作可使用计算系统而被执行，并且对应的查询使用计算系统而被运行，现在将描述示例计算系统。

[0155] 计算系统现在越来越多地采取各种各样的形式。计算系统可以例如是手持式设备、电器、膝上型计算机、台式计算机、大型机、分布式计算系统，或甚至通常尚未被认为是计算系统的设备。在本说明书和权利要求书中，术语“计算系统”被宽泛地定义为包括如下的任何设备或系统（或其组合），它们包括至少一个物理和有形的处理器以及物理和有形的存储器，该物理和有形的存储器能够在其上具有可由处理器执行的可执行指令。存储器可以采取任何形式，并且可以取决于计算系统的性质和形式。计算系统可以被分布在网络环境上并且可以包括多个组成计算系统。

[0156] 如图7中所示，在其最基本的配置中，计算系统700通常包括至少一个硬件处理单元702和存储器704。存储器704可以是物理系统存储器，其可以是易失性、非易失性或这两者的某种组合。术语“存储器”在这里也可以用于指非易失性大容量存储器，诸如物理存储介质。如果计算系统是分布式的，则处理、存储器和/或存储能力也可以是分布式的。如在这里所使用的，术语“可执行模块”或“可执行组件”可以指可以在计算系统上被执行的软件对象、路由或方法。在这里所描述的不同组件、模块、引擎和服务可以被实现为在计算系统上执行的对象或进程（例如，作为单独的线程）。

[0157] 在下面的描述中，参考由一个或多个计算系统执行的动作来描述实施例。如果这样的动作以软件实现，则执行动作的相关联的计算系统的一个或多个处理器响应于已经执行了计算机可执行指令来指引计算系统的操作。例如，这样的计算机可执行指令可以被体现在形成计算机程序产品的一个或多个计算机可读介质上。这样的操作的示例涉及对数据的操纵。计算机可执行指令（和被操纵的数据）可以被存储在计算系统700的存储器704中。计算系统700还可以包含通信信道708，其允许计算系统700通过例如网络710与其他消息处理器通信。计算系统700还包括显示器，其可以用于向用户显示视觉表示。

[0158] 在这里所描述的实施例可包括或利用包括计算机硬件的专用或通用计算机，计算机硬件诸如例如是一个或多个处理器和系统存储器，如在下面更详细讨论的。在这里所描述的实施例还包括用于携带或存储计算机可执行指令和/或数据结构的物理和其他计算机可读介质。这样的计算机可读介质可以是可以由通用或专用计算机系统访问的任何可用介质。存储计算机可执行指令的计算机可读介质是物理存储介质。携带计算机可执行指令的计算机可读介质是传输介质。因此，作为示例而非限制，本发明的实施例可以包括至少两种截然不同种类的计算机可读介质：计算机存储介质和传输介质。

[0159] 计算机存储介质包括RAM、ROM、EEPROM、CD-ROM或其它光盘存储装置、磁盘存储装置或其它磁存储设备,或任何其他物理和有形的存储介质,其可用于存储计算机可执行指令或数据结构的形式的所需的程序代码装置,并且可以由通用或专用计算机访问。

[0160] “网络”被定义为使得能够在计算机系统和/或模块和/或其他电子设备之间传送电子数据的一个或多个数据链路。当通过网络或另一通信连接(硬连线、无线或硬连线或无线的组合)向计算机传送或提供信息时,计算机适当地将该连接视为传输介质。传输介质可以包括网络和/或数据链路,其可以用于携带计算机可执行指令或数据结构的形式的期望的程序代码装置,并且可以由通用或专用计算机访问。上述的组合也应被包括在计算机可读介质的范围内。

[0161] 另外,在到达各种计算机系统组件之后,计算机可执行指令或数据结构的形式的程序代码装置可以被从传输介质自动地传送给计算机存储介质(反之亦然)。例如,通过网络或数据链路接收到的计算机可执行指令或数据结构可以被缓存在网络接口模块(例如,“NIC”)内的RAM中,并且然后最终被传送给计算机系统RAM和/或计算机系统处的较不易失性的计算机存储介质。因此,应当理解,计算机存储介质可以被包括在也(或甚至主要地)利用传输介质的计算机系统组件中。

[0162] 计算机可执行指令包括例如当在处理器处被执行时使得通用计算机、专用计算机或专用处理设备执行某一功能或一组功能的指令和数据。计算机可执行指令可以例如是在由处理器直接执行之前经历一些转换(诸如编译)的二进制文件或甚至指令,诸如中间格式指令(诸如汇编语言),或甚至源代码。尽管已经用对结构特征和/或方法动作专用的语言描述了主题,但是将会理解,所附权利要求中定义的主题内容不一定限于所描述的特征或上面描述的动作。而是,所描述的特征和动作作为实现权利要求的示例形式而被公开。

[0163] 本领域技术人员将认识到,本发明可以在具有许多类型的计算机系统配置(包括个人计算机、台式计算机、膝上型计算机、消息处理器、手持设备、多处理器系统、基于微处理器的或可编程的消费者电子产品、网络PC、小型计算机、大型计算机、移动电话、PDA、寻呼机、路由器、交换机等)的网络计算环境中被实践。本发明还可以在分布式系统环境中被实践,在分布式系统环境中通过网络(要么通过硬连线的数据链路、无线数据链路,要么通过硬连线的和无线数据链路的组合)被链接的本地和远程的计算机系统都执行任务。在分布式系统环境中,程序模块可以位于本地和远程存储器存储设备二者中。

[0164] 在不脱离本发明的精神或本质特性的情况下,本发明可以按照其他具体形式被体现。所描述的实施例在所有方面都将被认为仅是说明性的而不是限制性的。因此,本发明的范围由所附权利要求书而不是由前面的描述来指示。在权利要求的等同物的含义和范围内的所有变化将被包括在其范围内。

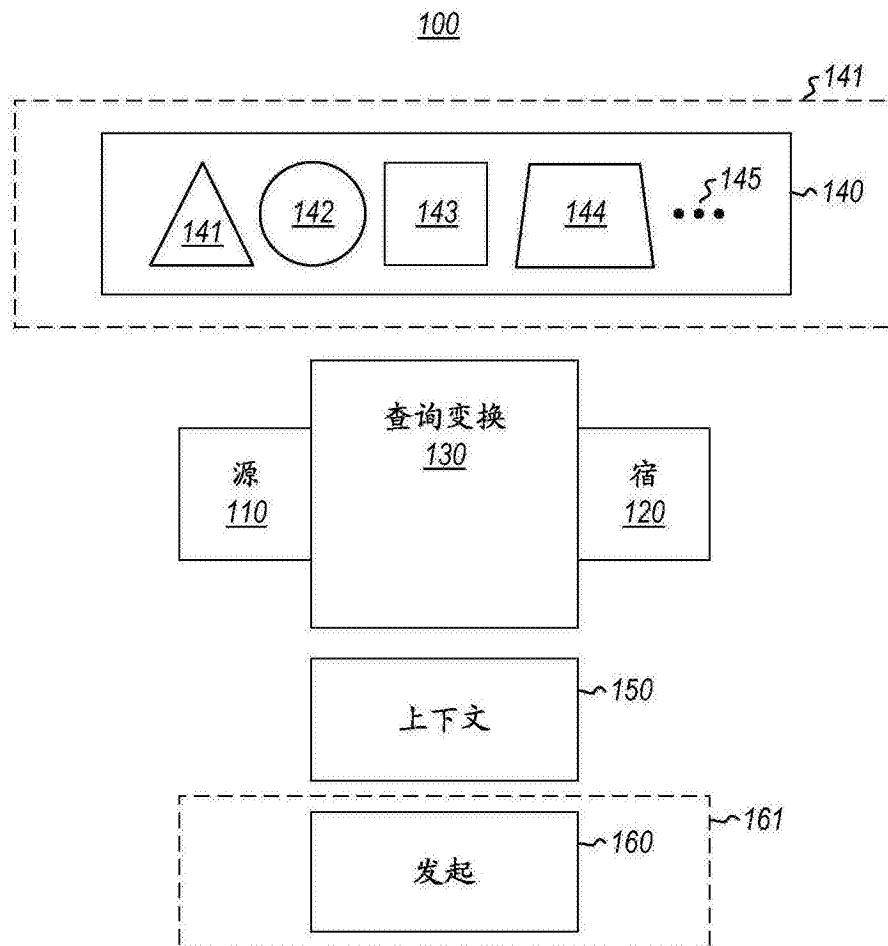


图1

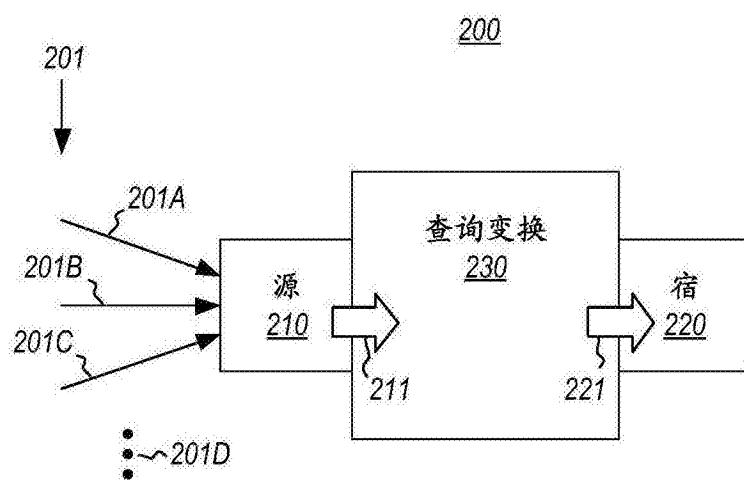


图2

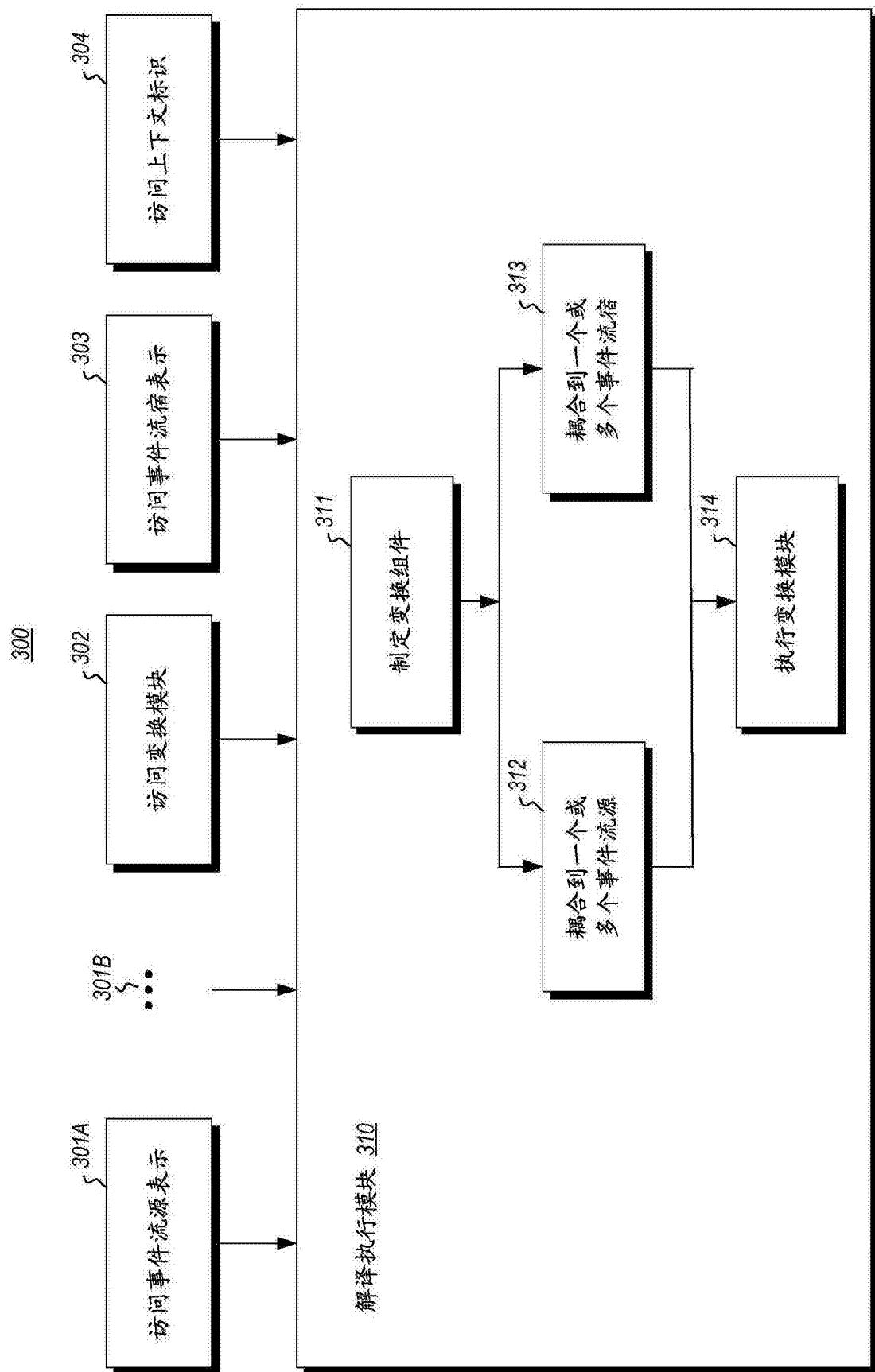
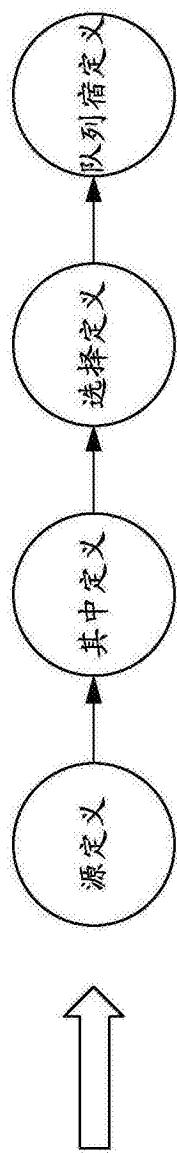


图3



`Input.From(performanceCounters)`
`.Where(e => e.CpuUsage > 85)`
`.Select(e => new Alarm)`
`.To(queue);`

图4

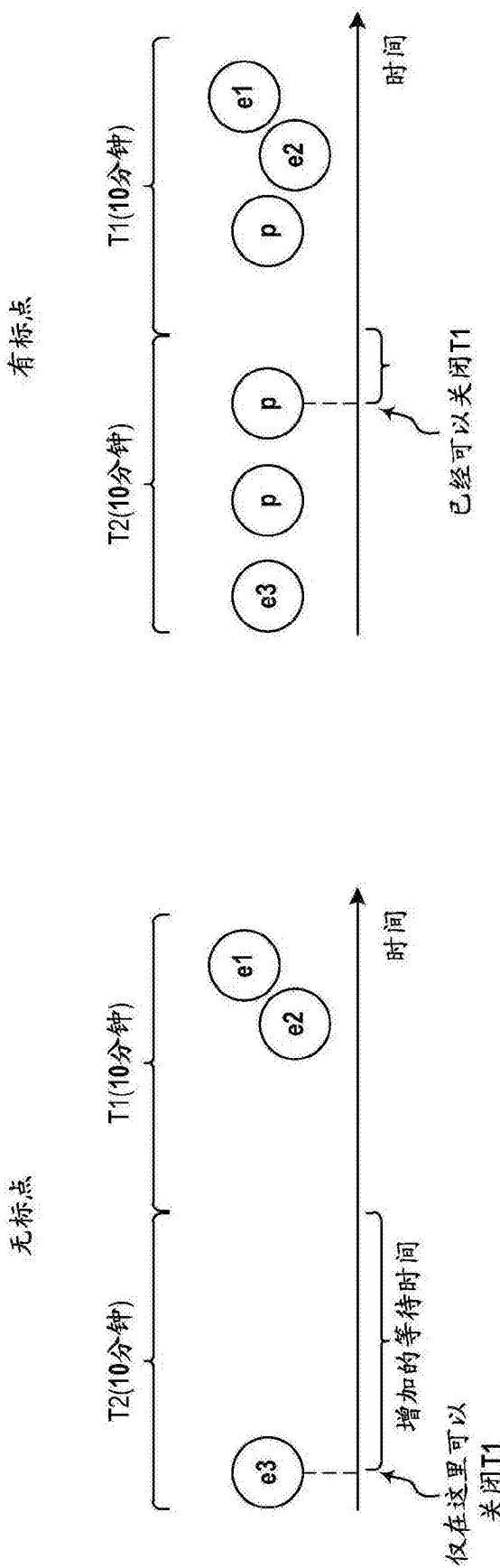


图5

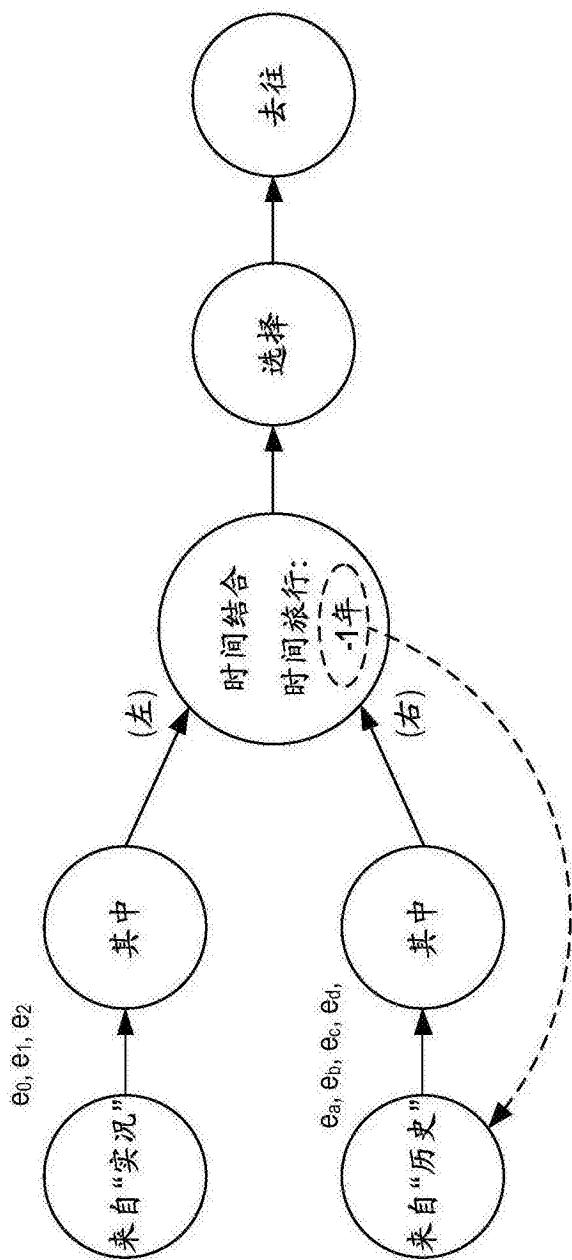


图6

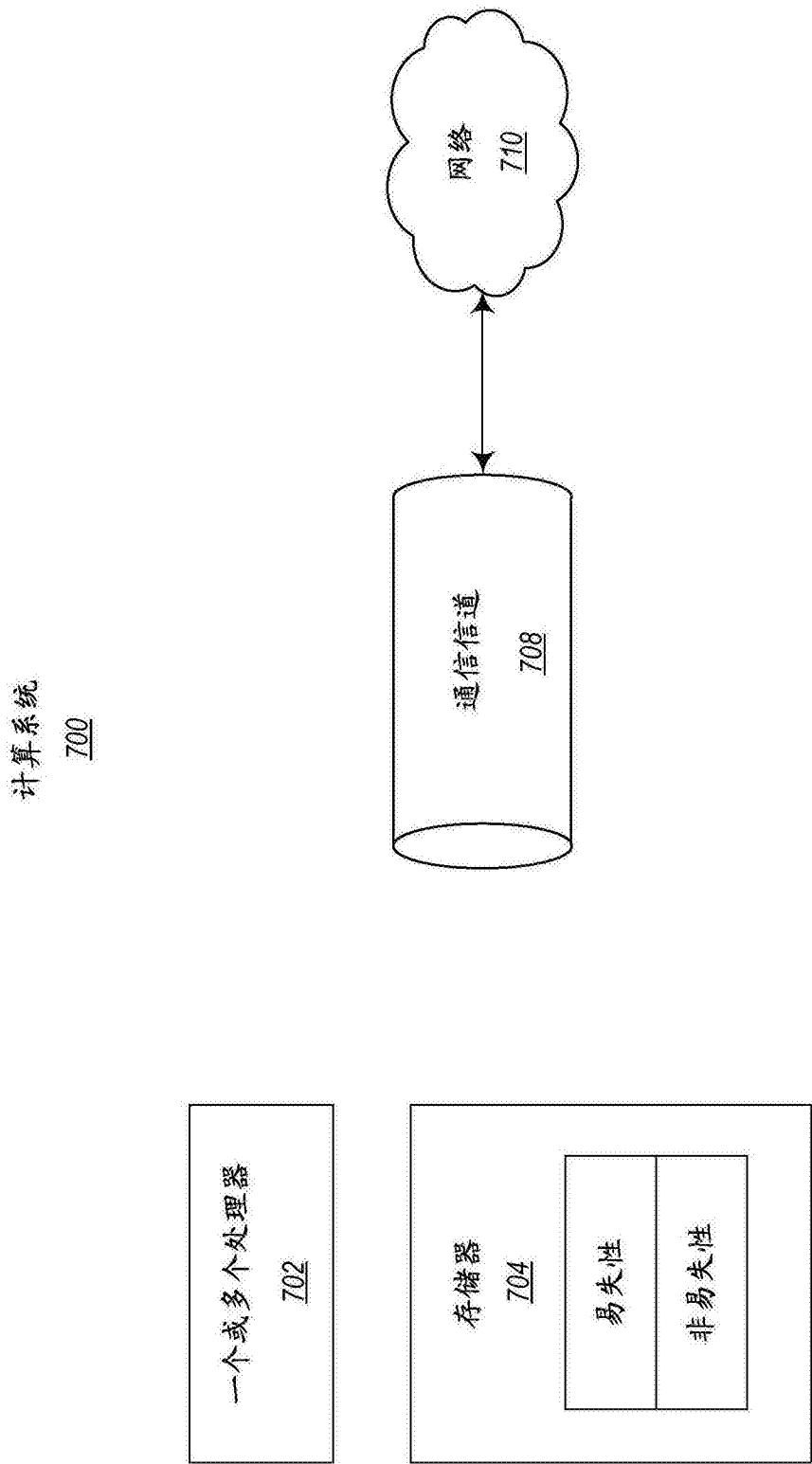


图7