



(12) 发明专利申请

(10) 申请公布号 CN 115629829 A

(43) 申请公布日 2023. 01. 20

(21) 申请号 202211226634.X

(22) 申请日 2017.03.23

(30) 优先权数据

62/312,106 2016.03.23 US

62/312,187 2016.03.23 US

62/312,223 2016.03.23 US

62/312,255 2016.03.23 US

(62) 分案原申请数据

201780019780.6 2017.03.23

(71) 申请人 江森自控泰科知识产权控股有限公司
任公司

地址 美国威斯康星州

(72) 发明人 杰森·卢卡斯 阿比史克·夏尔马

(74) 专利代理机构 北京东方亿思知识产权代理
有限责任公司 11258

专利代理师 杨佳婧

(51) Int.Cl.

G06F 9/448 (2018.01)

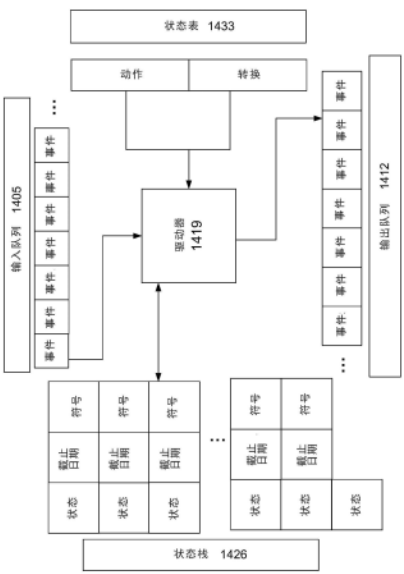
权利要求书3页 说明书40页 附图11页

(54) 发明名称

用于实时数据流编程的高效状态机

(57) 摘要

本公开涉及用于实时数据流编程的高效状态机。高效的基于状态机的模式匹配技术处理输入队列中的令牌并识别与一个或多个预定输入模式匹配的令牌序列中的模式而无需回溯。令牌可以包括数据或不包括数据,以及时间组分。令牌可以由传感器生成的数据流,其将物理属性转换为数字量。模式匹配技术在单个方向上处理输入队列,并且不检查任何先前检查的令牌。在实现中,使用状态机指定要匹配的特定模式,其中状态机在状态表中指定并使用状态栈操作。



1. 一种系统,包括:

输入队列,所述输入队列包括计算机存储器中的第一存储器位置,其中所述输入队列在所述第一存储器位置中存储要处理的令牌的先进先出序列,其中所述队列中的每个令牌包括数据或不包括任何数据,以及存储指示所述令牌进入所述输入队列的时间的时间戳,并且所述令牌由所述输入队列通过网络接收;

驱动器组件,所述驱动器组件被耦合到所述输入队列,其中所述驱动器组件以不回溯的方式处理所述输入队列中的令牌,并识别与一个或多个预定输入模式匹配的令牌序列中的模式,并且在识别到匹配的预定输入模式时,生成传出事件输出;

输出队列,所述输出队列被耦合到所述驱动器组件,所述输出队列包括所述计算机存储器中的第二存储器位置,其中所述输出队列在所述第二存储器位置中存储由所述驱动器组件生成的所述输出事件的先进先出序列;

状态表组件,所述状态表组件被耦合到所述驱动器组件,其中所述状态表组件以状态表格式存储所述预定输入模式;以及

状态栈组件,所述状态栈组件被耦合到所述驱动器组件,所述状态栈组件包括所述计算机存储器中的第三存储器位置,其中所述状态栈组件在所述第三存储器位置中存储帧的后进先出顺序存储,并且帧包括转换状态号、符号和截止日期。

2. 如权利要求1所述的系统,其中在所述输入队列处接收的令牌由硬件传感器生成,所述硬件传感器将物理属性转换为数字量。

3. 如权利要求1所述的系统,其中所述驱动器组件的操作包括:

检查所述状态栈的顶部帧上引用的状态;

检查所述输入队列中的下一输入令牌,并基于所述状态栈确定其类别;以及

针对由所述状态表指示的类别的令牌,执行由所述状态表指示的动作。

4. 如权利要求1所述的系统,其中所述输入队列基于由所述输入队列接收的时间将所述令牌存储在有序列表中,并且所述队列中的第一方向是从最早接收的令牌到最新接收的令牌的时间顺序,以及

所述驱动器组件通过在所述第一方向上检查所述输入队列来处理所述令牌,并且不检查任何先前检查的令牌。

5. 如权利要求1所述的系统,其中所述输入队列按照所述输入队列接收的时间顺序将所述令牌存储在列表中,所述队列中的第一方向是从最早接收的令牌到最新接收的令牌的时间顺序,所述队列中的第二方向是从最新接收的令牌到最早接收的令牌的时间顺序,并且所述第二方向与所述第一方向相反,

所述驱动器组件通过仅在所述第一方向或所述第二方向上而不是既在所述第一方向上也在所述第二方向上检查所述输入队列来处理所述令牌。

6. 如权利要求1所述的系统,其中所述驱动器仅读取所述输入队列的每个令牌一次,并且不将读取的令牌保留在缓冲器中以便稍后再次读取。

7. 如权利要求1所述的系统,其中要匹配的预定输入模式具有多个不定长度的令牌。

8. 如权利要求1所述的系统,其中所述状态表组件指定状态机并包括多个行,每个行提供所述状态机的转换状态,包括:

动作集合,其将输入令牌类别值或输入中断类别值映射到接收到该值的输入令牌时要

采取的动作，

转换集合，其将衍生符号类型映射到所述驱动器在合成该类型的衍生符号时应改成的状态，以及

超时，其指示允许所述状态机保持特定状态的时间间隔。

9. 如权利要求1所述的系统，其中所述转换状态号是对所述状态表中的行的引用，所述符号是从输入令牌或其他符号或组合导出的令牌或中间符号，并且所述截止日期是所述驱动器将所述帧从所述栈中弹出的未来时刻。

10. 如权利要求1所述的系统，其中所述驱动器组件和所述状态表组件实现在可编程门阵列中。

11. 如权利要求1所述的系统，其中所述驱动器组件和所述状态表组件使用计算机实现。

12. 如权利要求1所述的系统，其中所述状态表组件指定状态机，并且所述驱动器组件基于在所述输入队列中没有接收到具有流数据的令牌一段时间，而从所述状态机的第一状态改变到所述状态机的第二状态。

13. 如权利要求1所述的系统，其中所述状态表组件指定状态机，并且所述驱动器组件基于在第一时间段期间在所述输入队列中接收到具有流数据的令牌序列，而从所述状态机的第一状态改变到所述状态机的第二状态，以及

所述驱动器组件基于在第二时间段期间在所述输入队列中没有接收到具有流数据的令牌序列，而从所述状态机的第三状态改变到所述状态机的第四状态。

14. 一种方法，包括：

提供包括计算机存储器中的第一存储器位置的输入队列，其中所述输入队列在所述第一存储器位置中存储要处理的令牌的先进先出序列、以及与所述队列中的每个令牌相关联的时间戳，所述时间戳指示相关联的令牌进入所述输入队列的时间，并且所述令牌由所述输入队列通过网络接收；

提供驱动器组件，所述驱动器组件被耦合到所述输入队列，其中所述驱动器组件以不回溯的方式处理所述输入队列中的令牌，并识别与一个或多个预定输入模式匹配的令牌序列中的模式，并且在识别到匹配的预定输入模式时，生成传出事件输出；

提供输出队列，所述输出队列被耦合到所述驱动器组件，所述输出队列包括计算机存储器中的第二存储器位置，其中所述输出队列在所述第二存储器位置中存储由所述驱动器组件生成的所述输出事件的先进先出序列；

提供状态表组件，所述状态表组件被耦合到所述驱动器组件，其中所述状态表组件以状态表格式存储所述预定输入模式；

提供状态栈组件，所述状态栈组件被耦合到所述驱动器组件，所述状态栈组件包括计算机存储器中的第三存储器位置，其中所述状态栈组件在所述第三存储器位置中存储帧的后进先出顺序存储，并且帧包括翻译状态号、符号和截止日期；以及

为所述状态表组件生成状态表，包括：

识别终端符号集，其中每个符号表示输入数据的类别；

识别非终端符号集，其中每个非终端符号表示终端或非终端符号中的至少一个的模式；

识别语法规则集；

识别顶级规则；

识别属性集；

针对每个识别的终端符号和非终端符号计算第一集合；

针对每个识别的终端符号和非终端符号计算跟随集合；以及

使用相应跟随集合的每个成员从所述顶级规则的闭包生成起始状态。

15. 如权利要求14所述的方法，其中生成状态表包括：

通过递归地生成从所述起始状态起存在状态改变转换的每个状态的闭包，生成从所述起始状态可到达的所有状态；

通过组合和缩减来优化状态集合；以及

为每个状态生成动作、转换和超时值。

16. 如权利要求14所述的方法，其中所述语法规则集包括：

每个非空规则采用 $A \rightarrow B_0 \cdots B_n$ 的形式，其中A是非终端符号，并且 B_0 到 B_n 各自是终端符号或非终端符号；以及

每个空规则采用 $A \rightarrow \tau$ 的形式，其中 τ 是以某个有用的时间单位表征的有限的时间量，或者是无限的，表示无限的时间量。

17. 如权利要求14所述的方法，其中所述属性集包括：

每个属性采用形式 $\alpha: T = \{A_0 \rightarrow \lambda_0, \cdots A_n \rightarrow \lambda_n\}$ ，其中 α 是属性的名称，T是属性的类型，A是非终端符号，并且 λ 是用于计算该符号的属性值的函数。

18. 如权利要求14所述的方法，其中顶级规则具有包括 $\Omega \rightarrow B$ 的形式，其中B是终端符号或非终端符号。

19. 一种方法，包括：

从硬件传感器接收数据流，所述硬件传感器监视物理量并将监视的物理量转换成数字形式的数据流；

将所述数据流存储在输入队列中，其中所述数据流中的每个令牌与接收该令牌时的时间戳一起被存储；

通过在单个方向上通过所述输入队列读取令牌，并且不再次读取任何先前读取的令牌，识别所述数据流中与一个或多个预定模式匹配的模式；以及

在识别到所述数据流中的预定模式时，在输出队列中输出已经识别出预定模式的肯定指示。

20. 如权利要求19所述的方法，包括：

当在所述数据流中未识别出预定模式时，在所述输出队列中输出已经识别出预定模式的否定指示，其中所述预定模式被存储在状态表和状态栈中。

用于实时数据流编程的高效状态机

[0001] 分案申请说明

[0002] 本申请是申请日为2017年03月23日、申请号为201780019780.6、名称为“用于实时数据流编程的高效状态机”的发明专利申请的分案申请。

[0003] 相关申请交叉引用

[0004] 本专利申请要求2016年3月23日提交的美国专利申请62/312,106、62/312,187、62/312,223和62/312,255的权益,其与本申请中引用的所有其他参考文献一起通过引用合并于此。

背景技术

[0005] 本发明涉及计算领域,更具体地涉及用于实时数据流编程的高效状态机,其可以用于边缘计算以处理由工业机器生成的大量数据。

[0006] 传统的企业软件应用程序托管依赖于数据中心或“云”基础设施来利用规模经济和系统效率。然而,这些数据中心可能任意地远离物理操作点(例如,工厂、仓库、零售店等),企业在这些地方进行其大部分业务操作。工业物联网(IIoT)是指依赖于具有传感器的物理操作仪器的设备或用例集合,这些传感器跟踪频率非常高的事件。

[0007] 许多行业的工业机器都属于这种物联网(IoT),包括制造业、石油和天然气、采矿、运输、电力和水力、可再生能源、医疗保健、零售、智能建筑、智能城市和联网车辆。尽管云计算取得了成功,但存在许多缺点:将所有数据发送到云存储是不切实际的,因为连接可能并不总是存在,带宽不够,延迟变化太大,或者即使带宽存在但成本太高。即使连接、带宽和成本都不是问题,也没有实时决策并且预测性维护可能导致对机器的重大损害。

[0008] 因此,需要改进的计算系统、架构和技术,包括改进的边缘分析和用于实时数据流编程的更高效的状态机,以处理由工业机器生成的大量数据。

发明内容

[0009] 高效的基于状态机的模式匹配技术处理输入队列中的令牌并识别与一个或多个预定输入模式匹配的令牌序列中的模式而无需回溯。令牌可以包括数据或不包括数据和时间组分。令牌可以是由传感器生成的数据流,其将物理属性转换为数字量。模式匹配技术在单个方向上处理输入队列,并且不检查任何先前检查的令牌。在实现中,使用状态机指定要匹配的特定模式,其中状态机在状态表中指定并使用状态栈操作。

[0010] 在具体实现中,模式匹配技术用于边缘计算系统;一种在边缘实现智能的方法。功能包括:在网关设备或嵌入式系统上托管的软件层中通过传感器数据触发。软件层连接到局域网。软件层可以访问服务、应用程序和数据处理引擎的存储库。通过软件层提供的表达式将传感器数据与特定条件出现的语义描述相匹配。通过连续执行表达式自动发现模式事件。智能地组合由软件层管理的跨网络嵌入式系统和跨网关设备的服务和应用程序,以链接应用程序和分析表达式。根据资源可用性优化应用程序和分析的布局。监控软件层的运行状况。将原始传感器数据或表达式结果存储在本地时间序列数据库或云存储中。可以

对服务和组件进行容器化,以确保在任何网关环境中平稳运行。

[0011] 在物联网 (IoT) 数据源处启用边缘智能。系统为实时边缘分析和应用程序提供对于IoT设备传感器数据的丰富的访问 (流或批处理模式,或两者)。该系统包括一个高效且富有表现力的计算机语言,用于通过在低内存占用机器中运行的高性能分析引擎,执行分析功能和表达式。该系统允许将聚合数据发布到云以进一步进行机器学习。该系统包括用于开发边缘应用程序的软件开发工具包。基于云的管理控制台允许管理边缘部署、配置、应用程序和分析表达式。

[0012] 边缘基础设施和平台的具体实现是FogHorn Systems公司。(FogHorn)。FogHorn网站www.foghorn-systems.com、出版物(包括白皮书、用户指南、教程、视频和其他)以及关于FogHorn技术和产品的其他出版物通过引用并入本文。

[0013] FogHorn提供平台以为工业和商业物联网 (IoT) 数据提供边缘智能。数百亿工业和商业IoT设备产生的数据量将足以压倒整个互联网。FogHorn平台在IoT数据产生的地方来--网络边缘处理、分析和响应IoT数据。FogHorn的“智能边缘”软件平台可实现前所未有的自动化水平、运营效率、成本节约等等。

[0014] 工业物联网 (IIoT) 由互联的工业和商业设备组成,如传感器、机械和计算机。IIoT的目标是跨分布式企业实现更好的设备控制、数据管理、机器自动化和运营效率。公司可以在边缘应用雾计算,以利用实时分析和自动响应捕获绿色地带IIoT机会,同时利用云计算进行系统级管理和优化。如果添加额外的计算资源不可行,FogHorn边缘计算平台还设计为在现有可编程逻辑控制器 (PLC) (例如,棕色地带机会) 中运行。棕色地带指的是新系统的实现以解决信息技术 (IT) 问题领域,同时考虑已建立的系统。新的软件架构考虑了现有和正在运行的软件。

[0015] 边缘智能平台是一种基于雾计算概念的基于软件的解决方案,其可将数据处理和分析扩展到靠近IIoT设备所在的边缘。保持与边缘设备的紧密接近,而不是将所有数据发送到远程集中式云,最大限度地降低了延迟,从而实现最大化性能、更快的响应时间以及更高效的维护和运营策略。它还显著降低了总体带宽要求和管理广泛分布式网络的成本。

[0016] 专注于边缘IIoT操作降低了整体带宽需求,并能够立即自动响应时间敏感条件。工业界正在增加数十亿台新的IIoT设备,这些设备每天总共会产生数拍字节的数据。将所有这些数据发送到云不仅成本过高,而且还会产生更大的安全风险。边缘操作可确保更快的响应时间、降低风险并降低总体成本。

[0017] 2015年8月27日提交的美国专利申请62/210,981和2016年8月29日提交的美国专利申请15/250,720通过引用并入本文并描述了边缘计算环境和平台。2017年3月23日提交的美国专利申请15/467,313通过引用并入本文,并描述了实时数据流编程中的模式驱动反应的组合。2017年3月23日提交的美国专利申请15/467,318通过引用结合于此,并描述了用于实时数据流编程语言的工具和方法。

[0018] 在一种实现中,一种方法包括:从传感器(例如,监视物理量并将所监视的物理量转换为数字形式的数据流的硬件设备)接收数据流;将数据流存储在输入队列(例如,FIFO存储器)中,其中每个令牌包括与接收数据时的时间戳一起存储的数据流的数据(或无数据);通过在单个方向上通过输入队列读取令牌,并且不重新读取任何先前读取的令牌,识别数据流中与一个或多个预定模式匹配的模式;并且在识别数据流中的预定模式时,在输

出队列中输出已经识别出预定模式的肯定指示。

[0019] 在另一实现中,一种系统包括:输入队列,包括组织为FIFO存储器的计算机存储器中的存储器位置,其中输入队列存储要处理的令牌,其中每个令牌包括数据(或没有数据)和当令牌进入输入队列时的时间戳。一种驱动器组件,其在没有回溯的情况下处理输入队列中的令牌并识别与一个或多个预定输入模式匹配的令牌序列中的模式,并且在识别匹配的预定输入模式时,生成输出事件输出。输出队列,包括组织为FIFO存储器的计算机存储器中的存储器位置,其中输出队列存储由驱动器组件生成的输出事件序列。状态表组件,以状态表格式存储预定输入模式。状态栈组件,包括组织为LIFO存储器的计算机存储器中的第三存储器位置,其具有包括转换状态号、符号或截止时间或任何组合的栈帧。

[0020] 在另一实现中,一种方法包括:提供包括在组织为FIFO存储器的计算机存储器中的第一存储器位置的输入队列,其中输入队列存储要处理的令牌,其中每个令牌包括数据(或数据)和令牌进入输入队列时的时间戳。提供驱动器组件,其中驱动器组件在没有回溯的情况下处理输入队列中的令牌,并识别与一个或多个预定输入模式匹配的令牌序列中的模式,并且在识别匹配的预定输入模式时,生成输出事件输出。提供包括组织为FIFO存储器的计算机存储器中的第二存储器位置的输出队列,其中输出队列存储由驱动器组件生成的输出事件。提供以状态表格式存储预定输入模式的状态表组件。提供存储栈帧的状态栈组件(例如,LIFO存储器)。

[0021] 考虑到以下详细描述和附图,本发明的其他目的、特征和优点将变得显而易见,其中相同的附图标记在所有附图中表示相同的特征。

附图说明

[0022] 图1示出了客户端-服务器系统和网络的框图。

[0023] 图2示出了客户端或服务器的更详细的图。

[0024] 图3示出了计算机系统的系统框图。

[0025] 图4是边缘计算平台的框图,其位于传感器流和云之间。

[0026] 图5示出了包括边缘分析的边缘计算平台的更详细框图。

[0027] 图6显示了边缘基础设施和云基础设施之间的操作流程。

[0028] 图7示出了转换为确定性有限自动机(DFA)和状态减少机器的增强非确定性有限自动机(NFA)。

[0029] 图8示出了在接收到令牌alpha时从状态A到B的转换。

[0030] 图9示出了通过额外状态转换,状态X,从状态A到B的转换。

[0031] 图10示出了由句法分析形成的抽象语法树的示例。

[0032] 图11显示了用于变更的子图。

[0033] 图12示出了用于结合的子图。

[0034] 图13示出了具有结构的闭包。

[0035] 图14示出了使用被称为定时前向最右(1)或TFR(1)模式匹配的技术的模式匹配器的组件。

[0036] 图15A-15B示出了“宠物”示例的状态机图。

具体实施方式

[0037] 图1是结合了本发明实施例的分布式计算机网络100的简化框图。计算机网络100包括多个客户端系统113、116和119,以及经由多个通信链路128耦合到通信网络124的服务器系统122。通信网络124提供用于允许分布式网络100的各种组件彼此通信和交换信息的机制。

[0038] 通信网络124本身可以包括许多互连的计算机系统和通信链路。通信链路128可以是硬连线链路、光链路、卫星或其他无线通信链路、波传播链路或用于信息通信的任何其他机制。通信链路128可以是DSL、电缆、以太网或其他硬连线链路,无源或有源光链路,3G、3.5G、4G和其他移动性机制,卫星或其他无线通信链路,波传播链路或用于信息通信的任何其他机制。

[0039] 可以使用各种通信协议来协助图1中所示的各种系统之间的通信。这些通信协议可以包括VLAN、MPLS、TCP/IP、Tunneling、HTTP协议、无线应用协议(WAP)、供应商特定协议、定制协议等。虽然在一个实施例中,通信网络124是因特网,但在其他实施例中,通信网络124可以是任何合适的通信网络,包括局域网(LAN)、广域网(WAN)、无线网络、内联网、专用网络、公共网络、交换网络以及这些网络的组合等。

[0040] 图1中的分布式计算机网络100仅仅是说明结合本发明的实施例,并不限制权利要求中所述的本发明的范围。本领域普通技术人员将认识到其他变型、修改和替代方案。例如,一个以上的服务器系统122可以连接到通信网络124。作为另一个例子,多个客户端系统113、116和119可以经由接入提供商(未示出)或通过一些其他服务器系统耦合到通信网络124。

[0041] 客户端系统113、116和119通常从提供信息的服务器系统请求信息。出于这个原因,服务器系统通常具有比客户端系统更多的计算和存储容量。然而,取决于计算机系统是在请求还是提供信息,特定计算机系统可以充当客户端或服务器。另外,尽管已经使用客户端-服务器环境描述了本发明的各方面,但是应该显而易见的是,本发明也可以实施在独立的计算机系统中。

[0042] 服务器122负责接收来自客户端系统113、116和119的信息请求,执行满足请求所需的处理,以及将对应于请求的结果转发回请求客户端系统。满足请求所需的处理可以由服务器系统122执行,或者可以替代地委托给连接到通信网络124的其他服务器。

[0043] 客户端系统113、116和119使用户能够访问和查询服务器系统122存储的信息。在一个具体的实施例中,客户端系统可以作为一个独立的应用程序运行,例如桌面应用程序或移动智能电话或平板应用程序。在另一个实施例中,在客户端系统上执行的“网络浏览器”应用程序使用户能够选择、访问、检索或查询服务器系统122存储的信息。网络浏览器的例子包括微软公司提供的Internet Explorer浏览器、Mozilla提供的火狐浏览器、谷歌提供的Chrome浏览器、Apple提供的Safari浏览器等。

[0044] 在客户端-服务器环境中,一些资源(例如,文件、音乐、视频或数据)存储在客户端,而其他资源从网络中的其他地方(例如服务器)存储或传送,并且可以通过网络(例如,互联网)存取。因此,用户的数据可以存储在网络或“云”中。例如,用户可以处理远程存储在云(例如,服务器)上的客户端设备上的文档。客户端设备上的数据可以与云同步。

[0045] 图2示出了本发明的示例性客户端或服务器系统。在实施例中,用户通过计算机工

作站系统(如图2所示)与系统交互。图2示出了包括监视器203、屏幕205、外壳207(也可以称为系统单元、机壳或外壳)、键盘或其他人体输入设备209以及鼠标或其他指示设备211的计算机系统201。鼠标211可以具有一个或多个按钮,例如鼠标按钮213。

[0046] 应当理解,本发明不将任何计算设备限于特定形状因子(例如,台式计算机形状因子),而是可以包括各种形状因子的所有类型的计算设备。用户可以与任何计算设备交互,包括智能电话、个人计算机、膝上型计算机、电子平板设备、全球定位系统(GPS)接收器、便携式媒体播放器、个人数字助理(PDA),其他网络接入设备以及能够接收或发送数据的其他处理设备。

[0047] 例如,在特定实现中,客户端设备可以是智能手机或平板设备,诸如Apple iPhone(例如,Apple iPhone 6),Apple iPad(例如,Apple iPad或Apple iPad mini),Apple iPod(例如,Apple iPod Touch),Samsung Galaxy产品(例如,Galaxy S系列产品或Galaxy Note系列产品),Google Nexus设备(例如,Google Nexus 6、Google Nexus 7或Google Nexus 9)和Microsoft设备(例如,Microsoft Surface平板电脑)。通常,智能手机包括电话部分(和相关的无线电部分)和计算机部分,它们可通过触摸屏显示器访问。

[0048] 存在用于存储电话部分(例如,联系人和电话号码)和计算机部分(例如,包括浏览器的应用程序、图片、游戏、视频和音乐)的数据的非易失性存储器。智能手机通常包括用于拍摄照片和视频的相机(例如,前置摄像头或后置摄像头或两者)。例如,智能手机或平板电脑可用于拍摄可以流式传输到一个或多个其他设备的实时视频。

[0049] 外壳207容纳熟悉的计算机组件,其中一些未示出,例如处理器、存储器、大容量存储设备217等。大容量存储设备217可以包括大规模磁盘驱动器、软盘、磁盘、光盘、磁光盘、固定盘、硬盘、CD-ROM、可记录CD、DVD、可记录DVD(例如,DVD-R、DVD+R、DVD-RW、DVD+RW、HD-DVD或蓝光光盘)、闪存和其他非易失性固态存储器(例如,USB闪存驱动器或固态驱动器(SSD))、电池备份的易失性存储器、磁带存储器,读取器和其他类似介质,以及这些介质的组合。

[0050] 可以使用、存储于、或关联于计算机可读介质来实现本发明的计算机实现的或计算机可执行的版本或计算机程序产品。计算机可读介质可以包括参与向一个或多个处理器提供指令以供执行的任何介质。这种介质可以采用许多形式,包括但不限于非易失性、易失性和传输介质。非易失性介质包括例如闪存、或光盘或磁盘。易失性介质包括静态或动态存储器,例如高速缓冲存储器或RAM。传输介质包括同轴电缆、铜线、光纤线和布置在总线中的电线。传输介质还可以采用电磁、射频、声波或光波的形式,例如在无线电波和红外数据通信期间产生的那些。

[0051] 例如,本发明的软件的二进制机器可执行版本可以存储或驻留在RAM或高速缓冲存储器中,或存储在大容量存储设备217上。本发明的软件的源代码也可以存储或驻留在大容量存储设备217(例如,硬盘、磁盘、磁带或CD-ROM)上。作为另一个例子,本发明的代码可以通过线路、无线电波或通过诸如因特网的网络进行传输。

[0052] 图3示出了用于执行本发明的软件的计算机系统201的系统框图。如图2所示,计算机系统201包括监视器203、键盘209和大容量存储设备217。计算机系统501还包括子系统,诸如中央处理器302、系统存储器304、输入/输出(I/O)控制器306、显示适配器308、串行或通用串行总线(USB)端口312、网络接口318和扬声器320。本发明还可以与具有额外或更少

子系统的计算机系统一起使用。例如,计算机系统可以包括一个以上的处理器302(即,多处理器系统),或者系统可以包括高速缓冲存储器。

[0053] 诸如322的箭头表示计算机系统201的系统总线体系结构。然而,这些箭头表示用于链接子系统的任何互连方案。例如,扬声器320可以通过端口连接到其他子系统或者具有到中央处理器302的内部直接连接。处理器可以包括多个处理器或多核处理器,其可以允许信息的并行处理。图2中所示的计算机系统201只是适用于本发明的计算机系统的示例。适用于本发明的子系统的其他配置对于本领域普通技术人员来说是显而易见的。

[0054] 计算机软件产品可以用各种合适的编程语言编写,例如C、C++、C#、Pascal、Fortran、Perl、Matlab(来自MathWorks, www.mathworks.com)、SAS、SPSS、JavaScript、AJAX、Java、Python、Erlang和Ruby on Rails。计算机软件产品可以是具有数据输入和数据显示模块的独立应用程序。或者,计算机软件产品可以是实例化为分布式对象的类。计算机软件产品也可以是组件软件,例如Java Beans(来自Oracle公司)或Enterprise Java Beans(来自Oracle公司的EJB)。

[0055] 该系统的操作系统可以是以下各项之一:Microsoft **Windows®**系统系列(例如,Windows 95、98、Me、Windows NT、Windows 2000、Windows XP、Windows XP x64 Edition、Windows Vista、Windows 7、Windows 8、Windows 10、Windows CE、Windows Mobile、Windows RT)、Symbian OS、Tizen、Linux、HP-UX、UNIX、Sun OS、Solaris、Mac OS X、Apple iOS、Android、Alpha OS、AIX、IRIX32或IRIX64。可以使用其他操作系统。Microsoft Windows是微软公司的商标。

[0056] 此外,计算机可以连接到网络并且可以使用该网络与其他计算机交互。网络可以是内联网、互联网或因特网等。网络可以是有线网络(例如,使用铜线)、电话网络、分组网络、光网络(例如,使用光纤)或无线网络,或这些的任何组合。例如,可以使用以下协议,使用无线网络在计算机和本发明的系统的组件(或步骤)之间传递数据和其他信息,诸如Wi-Fi(IEEE标准802.11、802.11a、802.11b、802.11e、802.11g、802.11i、802.11n、802.11ac和802.11ad等)、近场通信(NFC)、射频识别(RFID)、移动或蜂窝无线(例如,2G、3G、4G、3GPP LTE、WiMAX、LTE、LTE高级、Flash-OFDM、HIPERMAN、iBurst、EDGE演进、UMTS、UMTS-TDD、1xRDD和EV-DO)。例如,来自计算机的信号可以至少部分地无线传输到组件或其他计算机。

[0057] 在实施例中,利用在计算机工作站系统上执行的网络浏览器,用户通过诸如因特网之类的网络访问万维网(WWW)上的系统。网络浏览器用于以各种格式下载网页或其他内容,包括HTML、XML、文本、PDF和附录,并且可用于将信息上载到系统的其他部分。网络浏览器可以使用统一资源标识符(URL)来标识网络上的资源,并且在网络上传输文件时使用超文本传输协议(HTTP)。

[0058] 在其他实现中,用户通过本地和非本地应用程序中的任一个或两者访问系统。本地应用程序本地安装在特定计算系统上,并且特定于该计算系统的操作系统或一个或多个硬件设备或这些的组合。这些应用程序(有时也称为“app”)可以通过直接互联网升级修补机制或通过应用程序商店(例如,Apple iTunes和App商店、Google Play商店、Windows Phone商店以及Blackberry App World商店)进行更新(例如,定期)。

[0059] 该系统可以在独立于平台的、非本地应用程序中运行。例如,客户端可以使用与一个或多个服务器的网络连接通过来自一个或多个服务器的网络应用程序访问系统,并在网

络浏览器中加载该网络应用程序。例如,网络应用程序可以通过因特网由网络浏览器从应用程序服务器下载。非本地应用程序也可以从其他来源获得,例如磁盘。

[0060] 图4示出了边缘计算平台406的框图,该边缘计算平台406通常在传感器409和云412之间的边缘网关或等同物上运行。边缘计算平台使得能够实现对于管理和优化工业机器和其他工业物联网而言重要的边缘智能。边缘网关的组件包括:摄取421、富集425、复杂事件处理(CEP)引擎429、应用程序432、通过表达式语言的分析435和传输438。云可以包括边缘提供和协调443以及云和边缘分析和应用程序可移植性446。

[0061] 如上所述,边缘计算平台的具体实现来自FogHorn。FogHorn是在迅速崛起的“边缘智能”领域的领导者。通过提供更接近控制系统和物理传感器的高性能处理、分析和异构应用程序,FogHorn的突破性解决方案实现闭环设备优化的边缘智能。这为制造、石油和天然气、电力和水力、运输、采矿、可再生能源、智能城市等工业客户带来了大量数据和实时地处理。FogHorn技术受到全球领先的工业互联网创新者以及云计算、高性能边缘网关和IoT系统集成的主要参与者的欢迎。

[0062] Foghorn提供:在流和批处理模式下为边缘应用程序提供丰富的IoT设备和传感器数据访问。用于执行分析功能的高效且富有表现力的DSL。功能强大的小型化分析引擎,可在低占用空间的机器上运行。发布功能,用于将聚合数据发送到云以进行进一步的机器学习。用于开发边缘应用程序的SDK(多语言)。管理控制台,用于管理配置、应用程序和分析表达式的边缘部署。

[0063] FogHorn提供高效且高度可扩展的边缘分析平台,可实现来自工业机器的传感器数据的实时、实地流处理。FogHorn软件栈是在边缘和云上运行的服务组合。

[0064] “边缘”解决方案可以支持将传感器数据摄取到本地存储库中,并且可以选择将未处理的数据发布到云环境以进行离线分析。然而,许多工业环境和设备缺乏互联网连接,使得这些数据无法使用。但即使具有互联网连接,所产生的大量数据也可能轻易超过可用带宽,或者成本太高而无法发送到云端。此外,当数据上传到云,在数据中心处理,结果传回边缘时,采取任何行动可能为时已晚。

[0065] FogHorn解决方案通过提供高度小型化的复杂事件处理(CEP)引擎(也称为分析引擎)以及强大且富有表现力的域特定语言(DSL)以表达对大量传入传感器的数据流的规则来解决此问题。然后可以立即使用这些表达式的输出来防止昂贵的机器故障或停机,以及实时提高工业操作和过程的效率 and 安全性。

[0066] FogHorn平台包括:能够在低占用空间环境以及高吞吐量或网关环境中运行。高度可扩展且高性能的CEP引擎,可以对传入的流式传感器数据进行操作。在边缘具有丰富数据访问权限的异构应用程序开发和部署。跨云和边缘的应用程序移动性。云端和边缘之间的高级机器学习(ML)和模型转换。开箱即用,FogHorn支持主要的工业数据摄取协议(例如OPC-UA、Modbus、MQTT、DDS等)以及其他数据传输协议。此外,用户可以轻松地将自定义协议适配器插入FogHorn的数据摄取层。

[0067] FogHorn边缘服务在IIoT设备所在的网络边缘运行。边缘软件栈负责将来自传感器和工业设备的数据摄取到高速数据总线上,然后在流数据上执行用户定义的分析表达式,以获得洞察力并优化设备。这些分析表达式由FogHorn高度可扩展且占地面积小的复杂事件处理(CEP)引擎执行。

[0068] FogHorn边缘服务还包括用于基于时间的传感器数据查询的本地时间序列数据库和用于开发可以在流和批处理模式下使用数据的应用程序的多语言SDK。可选地,该数据还可以发布到客户选择的云存储目的地。

[0069] FogHorn平台还包括在云或预置环境中运行的服务,以远程配置和管理边缘。FogHorn的云服务包括用于开发和部署分析表达式,使用称为Docker (www.docker.com) 的应用程序将应用程序部署到边缘,以及用于管理服务与客户的身份访问管理和维持解决方案的集成的管理UI。该平台还能够将在云中开发的机器学习模型转换为可在边缘执行的传感器表达式。

[0070] 例如,应用程序应用实时数据监视和分析、预测性维护计划和自动流程重定向,以防止由于空穴事件对泵造成损害。另一个例子是使用FogHorn边缘智能软件的风能管理系统,以最大化发电,延长设备寿命,并应用历史分析进行准确的能源预测。

[0071] 图5示出了边缘计算平台的更详细框图。该平台具有三个逻辑层或部分,数据摄取512、数据处理515和数据发布518。数据摄取组件包括连接到生成数据的传感器或设备523的代理520。代理通过来自相应协议服务器的一个或多个协议从传感器收集或摄取数据。代理可以是针对协议(例如MQTT,OPC UA,Modbus和DDS)的客户端或代理。传感器提供或输出的数据通常是二进制数据流。可以通过推送或拉取方法将这些数据从传感器传输或传递到代理。

[0072] 推送描述了一种通信方式,其中由发送者(例如,传感器)发起针对给定交易的请求。拉取(或取得)描述了一种通信方式,其中由接收方(例如代理)发起针对信息传输的请求。另一种通信技术是轮询,其中接收器或代理周期性地查询或检查传感器是否有要发送的数据。

[0073] MQTT(先前的MQ遥测传输)是基于ISO标准的基于发布-订阅的“轻量级”消息传递协议,用于在TCP/IP协议之上。替代协议包括高级消息队列协议、IETF约束应用协议、XMPP和网络应用消息传递协议(WAMP)。

[0074] OPC统一架构(OPC UA)是OPC基金会开发的用于互操作性的工业M2M通信协议。它是开放平台通信(OPC)的继承者。

[0075] Modbus是一种串行通信协议,最初由Modicon(现为施耐德电气)于1979年出版,用于其可编程逻辑控制器(PLC)。它简单而强大,从而成为所有意图和目的的标准通信协议。它现在是连接工业电子设备的常用手段。

[0076] 数据处理515包括数据总线532,其连接到数据摄取层的代理520。数据总线是所有连接组件之间数据和控制消息的中心主干。组件订阅数据并控制流经数据总线的消息。分析引擎535是一个这样的重要组件。分析引擎基于在表达式语言538中开发的分析表达式来执行传感器数据的分析。连接到数据总线的其他组件包括配置服务541、度量服务544和边缘管理器547。数据总线还包括“解码器服务”,其通过将原始二进制数据解码为可消费数据格式(例如JSON)来丰富来自传感器的传入数据,并且还使用其他必要且有用的元数据进行修饰。此外,丰富性可以包括但不限于数据解码、元数据修饰、数据归一化等。

[0077] JSON(有时称为JavaScript Object Notation)是一种开放标准格式,它使用人类可读的文本来传输由属性-值对组成的数据对象。JSON是用于异步浏览器或服务器通信(AJAX)或两者的常见数据格式。JSON的替代是XML,它被AJAX使用。

[0078] 边缘管理器连接到云412,并且特别是连接到云管理器552。云管理器连接到用于客户身份和访问管理(IAM) 555和用户界面控制台558的代理,其也在云中。还有可以通过云访问的应用程序561。身份和访问管理是一种安全和业务规则,使正确的人能够在正确的时间以正确的原因访问正确的资源。

[0079] 在数据处理515内,软件开发工具包(SDK) 564组件还连接到数据总线,这允许创建可以部署在边缘网关上的应用程序567。软件开发工具包还连接到本地时间序列数据库以获取数据。应用程序可以集装箱化,例如使用诸如Docker之类的集装箱技术。

[0080] Docker集装箱将一个软件包装在一个完整的文件系统中,该系统包含运行所需的一切:代码、运行时间、系统工具和系统库--可以安装在服务器上的任何东西。这可确保软件始终运行一致,无论其运行环境如何。

[0081] 数据发布518包括连接到云中的存储位置573的数据发布者570。此外,软件开发工具包564的应用程序567可以访问时间序列数据库576中的数据。时间序列数据库(TSDB)是针对处理时间序列数据而优化的软件系统,由时间索引(例如,日期时间或日期时间范围)的数字阵列。时间序列数据库通常是滚动或循环缓冲区或队列,其中当新信息添加到数据库时,将删除最旧的信息。数据发布者570还连接到数据总线并订阅需要存储在本地时间序列数据库或云存储中的数据。

[0082] 图6示出了边缘602和云基础结构之间的操作流程。上面描述了一些特定的边缘基础结构。从传感器606收集数据。这些传感器可以用于工业、零售、医疗保健或医疗设备、或电力或通信应用,或这些的任何组合。

[0083] 边缘基础设施包括软件平台609,其具有数据处理612、本地时间序列数据库615、云库618、分析复杂事件处理引擎(CEP) 621、分析实时流式域特定语言(DSL) 624(例如,Foghorn的Vel语言),以及实时聚合和访问627。平台可以包括虚拟传感器630,下面将更详细地描述虚拟传感器630。虚拟传感器提供丰富的实时数据访问。

[0084] 该平台可通过一个或多个应用程序633(例如应用程序1、2和3)访问,其可使用软件开发工具包或SDK开发。应用程序可以是异构的(例如,以多种不同语言开发)并利用复杂事件处理引擎621,以及执行机器学习。可以使用应用商店637来分发应用程序,应用商店637可以由边缘平台开发者或边缘平台的客户(可以被称为伙伴)提供。通过应用商店,用户可以下载并与他人共享应用。应用程序可以执行分析和应用程序639,包括机器学习、远程监控、预测性维护或操作智能,或这些的任意组合。

[0085] 对于应用程序,边缘和云之间存在动态应用程序移动性。例如,使用FogHorn软件开发工具包开发的应用程序可以部署在边缘或云端,从而实现边缘和云之间的应用程序移动性。这些应用程序可以用作边缘的一部分或作为云的一部分。在实现中,由于应用程序是集装箱化的,因此可以实现此功能,因此它们可以独立于执行它们的平台进行操作。分析表达式也是如此。

[0086] 存在允许集成监管和管理640的数据应用程序,包括在云中或在私有数据中心644处监视或存储数据。

[0087] 物理传感器是电子传感器,其测量其环境的一些特性作为模拟或数字测量。通常使用模数转换器将模拟测量转换为数字量。传感器数据可以根据需要进行测量(轮询),也可以统一速率作为流进行测量。典型的传感器规格包括范围、精度、分辨率、漂移、稳定性和

其他属性。大多数测量系统和应用程序直接利用或传递传感器数据以进行处理、运输或存储。

[0088] 该系统具有“可编程软件定义传感器”，也称为虚拟传感器，其是使用分析表达式语言创建的基于软件的传感器。在实现中，分析表达式语言是FogHorn的分析表达式语言。这种表达式语言称为Vel。Vel语言得到有效实施，以在具有低执行延迟的限制性低占地空间环境中支持实时流分析。例如，系统的延迟可以是大约10毫秒或更短。

[0089] 在实现中，可编程软件定义传感器使用称为“传感器表达式语言”或SXL的声明性应用程序接口 (API) 来创建。SXL语言的具体实现是来自FogHorn的Vel。Vel传感器是通过该构造创建的Vel传感器，并且提供来自多个源 (包括物理传感器和Vel传感器) 生成的处理数据的导出测量。在此应用中，Vel和SXL可互换使用。

[0090] Vel传感器可以从这三种来源中的任何一种或其组合获得：

[0091] 1. 单个传感器数据。

[0092] 1.1. 源自单个物理传感器的虚拟或Vel传感器可以使用动态校准、信号处理、数学表达、数据压缩或数据分析或任何组合的来转换传入的传感器数据。

[0093] 2. 多个物理传感器数据。

[0094] 2.1. 虚拟或Vel传感器或作为来自多个异构物理传感器的转换 (使用上述方法) 导出的。

[0095] 3. 物理传感器数据和虚拟传感器数据的组合可用于Vel传感器设备的实现。

[0096] Vel传感器是特定于域的，并且在考虑特定应用的情况下创建。Vel编程接口的特定实现使应用程序能够通过转换 (例如，数学表达式) 和聚合来定义数据分析。Vel包括一组数学运算符，通常基于编程语言。Vel传感器通过执行Vel结构或程序在运行时间对数据进行操作。

[0097] Vel传感器的创建。Vel传感器被设计为软件设备，以使数据实时可用。这需要在嵌入式计算硬件上实时执行使用Vel开发的应用程序，以便以应用程序所需的速率生成Vel传感器数据。该系统包括一个高效的执行引擎来实现这一目标。

[0098] Vel传感器的优点包括：

[0099] 1. 可编程性。Vel使Vel传感器可编程以合成数据，以满足数据质量、频率和信息的特定应用要求。Vel传感器可以广泛分布为空中软件升级，以插入来自物理传感器和其他 (例如，预先存在的) Vel传感器的数据。因此，应用程序开发人员可以创建有助于业务逻辑高效执行的数字基础结构，而与物理基础结构的布局无关。

[0100] 2. 可维护性或透明度。Vel传感器在应用程序和物理传感器之间创建数字抽象层，使开发人员能够与由于对物理传感器的升级和服务所导致的物理基础结构中的变化隔离开来。

[0101] 3. 效率：Vel传感器通过将来自物理传感器的原始数据转换为包含在其中的信息的精确表示，从而提高信息管理的效率。这种效率转化为有效利用IT资源，如应用程序下游的计算、网络和存储。

[0102] 4. 实时数据：Vel传感器提供从真实世界或物理传感器数据流计算的实时传感器数据。这使得数据可用于具有最小时间延迟的应用程序。

[0103] 实现方式。该系统基于Vel接口构建了Vel传感器的可扩展、实时实现。Vel包括

Java语言支持的运算符,并与物理传感器及其协议很好地集成。

[0104] 该系统带来了一种新颖的方法,用于精确地表达对要执行的物理传感器数据的操作。该声明性表达式将数字抽象的定义与物理传感器上的实现分开。

[0105] 给定一组不同类型的数据流和一组用于响应和处理那些流中的特定数据模式的函数,本发明是一种描述和转换这些函数以便当数据到达流时可以适当并且高效调用它们的技术。

[0106] 解决这类问题的需要通常出现在所有形式的数据流编程中。它适用于非常大规模的体系结构,例如企业数据中心内部和之间的数据流,以及非常小规模 of 体系结构,例如嵌入式设备中的事件流。

[0107] 本发明适用于数据流编程的所有领域;然而,它最适合于能够检测到匹配并且应用处理功能的速度具有最高重要性的情况,以及用于执行的存储和计算资源有限的情况。

[0108] 示例。从给定的整数流中,我们希望匹配一个或多个非零值,后跟一个或多个零。当匹配此模式时,我们希望计算非零值的总和,并将结果写入另一个流。

[0109] 我们可以用正则表达式表示法编写该问题的模式匹配部分,然后另外将和的计算写为算术的表达式。此时,设计用于边缘计算中的数据流应用程序的Vel编程语言允许我们以统一的表示法编写整个转换,因此:

[0110]

```
stream("output") =  
( a: {!= 0} .. {>0}, :0 .. {>0} -> sum(a) )  
from stream("input")
```

[0111] 该技术将上述函数参数化转换为状态机。然后,它将基于该状态机将匹配实现为确定性有限自动机,并将得到的匹配馈送到求和表达式。该流程在图7中示出。这是状态0 705,状态1 710,“来自列表a”块715,以及“推送和(a)”块720。

[0112] 可以通过为每个处理函数生成匹配函数来解决该问题。匹配函数接受来自流的数据窗口作为输入,并且对于匹配返回真(true),对于非匹配返回假(false)。当数据流过窗口时,必须重复应用匹配函数,直到找到匹配为止。找到匹配后,将应用处理函数。

[0113] 出现这种解决方案是因为处理函数以类似于用于数据库查询的方式进行规定。类似SQL的WHERE子句提供了一个描述匹配条件的布尔表达式,匹配函数是该表达式的直接编译。

[0114] 当新数据流入流缓冲区时,必须单独评估单独的匹配函数。为每个函数独立确定匹配。

[0115] 使用状态机来执行匹配比重复应用多个任意布尔表达式更高效。

[0116] 本发明从声明函数的参数的模式描述语言导出状态机。与传统的布尔表达式匹配函数相比,所导出的状态机更高效地检测数据流中的匹配。

[0117] 所导出的状态机还可以为数据流中检测到的匹配实现一组处理函数。可以组合多个匹配和相应的处理函数,并将其简化为高效地识别任何处理函数的匹配的单个状态机。

[0118] 所导出的状态机还可以被扩充以包括通过附加节点的自由(ϵ)转换,而不改变状态机识别的序列。

[0119] 通过这样的附加节点的转换可以触发对数据的各种动作。例如,它可以触发将确定性有限自动机(DFA)或栈机器的移位缓冲器中的数据收集到保持区域中。这些数据可能

稍后形成函数应用参数的基础。

[0120] 此应用程序使用术语DFA,但这些自动机或单元可以称为栈机器。严格地说,确定性有限自动机意味着有限的空间性能。然而,该专利中的自动机不一定是有限的,也可以是非限定但仍然简单的。因此,该专利中描述的DFA可以是非限定的。

[0121] 通过这样的附加节点的转换还可以使用在先前节点中捕获的数据作为函数应用参数来触发处理函数的调用。

[0122] 从结合了正则表达式和值表达式方面的脚本进行的转换产生了一个增强状态机或DFA,它可以高效地匹配模式和计算值。

[0123] 所得到的组合的匹配或计算算法比模式匹配和值计算的单独组织更高效。

[0124] 一种从词汇源构建DFA或状态机的方法,从非确定性有限自动机(NFA)开始,然后将其减少到最小DFA。DFA的目的是识别输入数据系列中的模式。出于本讨论的目的,我们将调用流经状态机令牌的数据和DFA识别的特定模式作为令牌的语言。

[0125] 考虑图8中NFA的部分。这部分也恰好是DFA,但这对于这个例子来说并不重要。一旦接收到令牌 α ,它就从状态A 805转换到状态B810。

[0126] 我们可以通过添加具有 ϵ 转换920的附加节点来增加该NFA,如图9所示。可以随时跟随 ϵ 边--可以说是自由,无论输入状态如何。

[0127] 一个或多个 ϵ 边的存在使状态机不确定;然而,可以通过算法去除 ϵ 边,通过这种方式将NFA减少到等效DFA,其可以通过表驱动方法有效地实现。因此,我们可以引入这些额外的 ϵ 转换,同时仍保留高效实施的策略。

[0128] 图9中的状态机在接收到令牌 α 925时将从状态A 905转换到状态X915,然后可以随意从状态X继续到状态B 910。 α 的推动仍然导致从状态A到状态B的转换,就像在图8中的简单机器中那样,并且不需要额外的输入来实现这种转换。因此可以看出,图9中的NFA翻译了与图8中相同的语言。它只是通过状态X进行额外的状态转换,以实现这种转换。

[0129] 额外的状态很有用,因为我们可以将副作用的表现与之联系起来。只要这些副作用既不改变状态机的定义也不改变流经状态机的数据,附加节点对语言识别没有影响,但副作用可以做额外的工作。

[0130] 在数据流反应实现中,额外的工作可以包括对数据的任何数量的有用动作或使用数据。在一个示例性实现中,这些工作可以包括:

[0131] 1.检查流经节点的数据并将其副本发送给外部收集器;

[0132] 2.当数据流过节点时对数据应用转换并收集转换的数据再临时缓冲区中;或者

[0133] 3.将收集的数据从临时缓冲区刷新到另外的转换中,并将结果推送到另一个DFA或栈机器。

[0134] 作为示例,让我们考虑源片段:

[0135] $(a: \{! = 0\} \dots \{> 0\}, : 0 \dots \{> 0\} \rightarrow \text{sum}(a))$

[0136] 该片段描述了由两项组成的模式:(1)第一项,称为 a ,其匹配非零值的一个或多个重现。(2)第二项,未给出名称,与零的一个或多个重复匹配。

[0137] 让我们假设我们希望将此作为反应的基础。我们将从一个被调入的源中读取值,当我们在输入中识别出片段的模式时,我们将通过评估片段的右侧并将结果推送到被调出的目的地来做出反应。

[0138] 例如,如果由值[101,202,303,0,0]组成,我们将通过将前三个值与a绑定,最后两个值与匿名第二项绑定来匹配模式。然后我们通过将求和函数应用于绑定到a的值列表[101,202,303]来评估右侧,返回606。然后我们将606推送出。

[0139] 诸如在根据本发明的该示例中的函数模式的转换可以通过计算机执行的转换程序来实现。该程序必须执行两种不同的转换形式:将面向函数的部分“sum(a)”转换为可执行计算的可执行语句块,并将面向模式的部分“a:{!=0}..{>0},:0..{>0}”转换为识别模式、捕获参数并调用函数的DFA或栈机器。我们称前者任务为函数转换,称第二任务为模式转换。

[0140] 专门编写编译器和解释器的计算机程序员很好地理解函数转换。模式转换、函数转换和模式转换的拟合以及模式识别和函数分派的后续自动化是本发明的主题。

[0141] 函数转换包括接受源文本,将文本分解为令牌,然后在语法的引导下,安排令牌以使它们形成描述源文本的语法内容的抽象语法树(AST)的叶子。然后,抽象语法树由一系列算法遍历,这些算法最终产生评估源所描述的函数所需的指令块。

[0142] 模式转换以由上述语法分析形成的抽象语法树开始。抽象语法树将包含一个或多个节点,这些节点构成模式声明的根。例如,我们上面的模式可能包含一个根节点有两个子节点,每个子节点描述模式的一项,如图10的左下部分所示。在图10中,有反应根节点1005,模式根节点1010,sum(a)节点1015,a节点1020和<no name>节点10。

[0143] 指定它作为匹配的示例和与其匹配的重现,识别模式项节点携带与正则表达式中的项相同的信息。另外,子节点序列一起按顺序指定与正则表达式项的线性连接相同的信息。正则表达式或正则表达式项的线性连接可以被翻译成NFA。我们已经发现可以在本发明中使用相同的算法,其中模式项代表正则表达式项。

[0144] 一旦基本的NFA形成,我们可以在模式项需要动作的位置注入我们的额外的、副作用诱导状态,并且在接受状态之后,调用反应的函数。

[0145] 为了继续我们的示例,项a要求我们收集与其匹配的值的列表,以便我们最终可以将它们作为参数传递给反应的函数。因此,我们将图9中描绘的转换应用于由项a产生的NFA状态,并使用新状态来进行收集匹配项的工作。然后我们再次应用转换,这次是NFA接受状态,并使用收集的值来调用反应的函数,将结果推送给反应的消费者,并清除收集缓冲区。在此增强的NFA转换为DFA并且状态减少之后,我们留下图7中描绘的机器。

[0146] 这些步骤用于将NFA转换为DFA,状态减少DFA,以及将DFA呈现为状态-动作表,因为使用状态-动作表来驱动状态机引擎的算法。

[0147] 通过本发明的技术产生的NFA可以被转化并呈现为表格。但是,所产生的表包括额外的列,该列由在通过每个状态时要执行的副作用 λ 组成。与其他技术不同,使用这种状态-动作- λ 表的自动化引擎每次进行转换时都会执行额外的 λ 。

[0148] 一种用于描述和转换用于数据流计算环境的反应函数的方法,包括:(i)识别反应函数;(ii)识别为该函数提供输入的参数模式;(iii)根据传递给函数的参数识别要评估的表达式;(iv)将参数模式转换为能够识别与模式匹配的输入序列的状态机;(v)使用附加的状态来增强状态机,这些状态用于收集和转换输入数据以准备将其用作函数的参数;以及(vi)将状态机减少到能够通过简单的软件或硬件自动化的状态-动作-效果表。

[0149] 给定一组函数和一系列值作为参数,本发明是一种将执行分派给参数匹配的函数

或确定参数不匹配任何函数的方法。这种方法的新颖之处在于,通过组合值表达式、类型表达式和正则表达式,它可以在没有歧义的情况下匹配类型系统中可表示的任何值序列。

[0150] 解决这类问题的需要出现在转换器、解释器和编译器的发展中,并且与多态调度的概念密切相关。如果考虑构成序列的任意前缀的元素构成单个对象(元组),那么调度到正确函数的任务可以被认为等同于元组类的方法的多态调度。

[0151] 本发明适用于需要这种多态分派的任何情况。这包括所有必须响应源自程序外部的数据流的事件驱动或反应程序方式。本发明在涉及多个数据流的实时处理的应用中特别有用,例如经常发生在边缘或雾计算或网络环境中。

[0152] 正则表达式通常用于检测符合特定模式的字符串。有许多密切相关的正则表达式语言,以及许多基于它们实现高效匹配引擎的工具。这些通常限于匹配字符序列。

[0153] 还有其他基于模式的符号,它们在字符串以外的域上运行。一个例子是XPATH,它描述了XML文档中的模式。这些符号通常不如正则表达式完整且功能较弱,并且针对特定域进行了定制。

[0154] 一些编程语言通过基于类型的模式匹配系统实现运行时多态调度。定义函数的多个重载,每个重载采用不同的类型和值的模式,并且通过将参数的类型和值与函数参数的模式相匹配,在运行时解析调度。Haskell就是这样一种编程语言。

[0155] 语言规范语言将无上下文语法描述为一系列生产规则。这些规则构成了语言的语法。编译器-编译器将这些规则转换为表驱动的确定性有限状态机,该机器可以识别该语言的实例。Bison是这种语言规范语言及其相关编译器-编译器的一个例子。

[0156] 语法驱动的模式匹配系统(如正则表达式)由于可以表示为确定性有限自动机(DFA)或状态机等简单机器而具有高效执行的优势,但它们缺乏完整类型系统的广泛建模功能。类型驱动的模式匹配系统(例如Haskell中使用的系统)具有更丰富的建模功能,但通常会牺牲可支持合理有效实现的功能,但仍然不如基于DFA的高速匹配系统那样高效。

[0157] 本发明涉及一种基于类型的匹配系统,该系统可以匹配其类型中可表示的所有状态,并且仍然可以高效地实现为状态机。类型和状态的通用模式被转换为表驱动状态机,其将高效地识别模式的实例。

[0158] 基于这些模式定义函数参数允许函数恰好匹配任何任意数据模式,并且在匹配中,从匹配数据元素中绑定其参数。描述函数联合的匹配模式的状态机是通过合并成员函数的状态机,然后将结果减少到最小数量的状态而形成的。过载之间的消歧或整体不匹配的检测尽可能早地在序列中发生,从而加速了函数应用的解析。匹配也可以延迟到序列中尽可能晚,产生函数的“贪婪”版本,该版本将接受尽可能多的输入。

[0159] 一种方法结合了值表达式、类型表达式和正则表达式,使得它可以在没有歧义的情况下匹配类型系统中可表示的任何值序列。此方法解析函数应用程序,并以最少的决策调度到正确的重载。该方法允许重载的函数应用程序执行与无上下文语法相同的工作,通过递归地识别语法子组件来识别特定语言并向其应用转换函数。

[0160] 该方法适用于包括多种不同类型的类型系统,例如:(1)基础单态类型集合,例如整数、实数和字符串。(2)多态类型及其构造函数集合,特别是具有某些属性的多态集合类型,我们将在下面简要讨论。(3)总和类型。(4)记录形式的产品类型。(5)模式形式的产品类型,它是元组的一般化,包括其字段的重复。(6) λ 类型,它将模式类型映射到任何类型。(7)

以及,多 λ 类型,由 λ 的列表组成。

[0161] 集合是由一个或多个元素范围组成的多态类型。集合类型在其包含的元素类型上进行参数化,使得例如整数集合是与字符串集合不同的类型。集合类型的特征还在于其内容的限制。特别地,集合类型可以被约束为有限的或无限的,或者在其左侧或右侧闭合或开放,或者这些的任何组合。

[0162] 请考虑以下整数集的示例:

[0163] 表A

[0164]	符号	长度	闭合性	含义
	[1]	1	左右闭合	由单个整数1组成的集合。
[0165]	[1, 2, 3]	3	左右闭合	由三个整数1、2和3组成的集合。
	[5000 .. 6000]	1001	左右闭合	从5000到6000的整数, 首尾包括。
	[10 ..]	无限	左闭合, 右开放	大于等于10的所有整数。
	[.. 10]	无限	左开放, 右闭合	小于等于10的所有整数。
	[> 5]	无限	左闭合, 右开放	大于5的所有整数。 与[6 ..]相同。
	[>= 5]	无限	左闭合, 右开放	大于等于5的所有整数。 与[5 ..]相同。
	[< 5]	无限	左开放, 右闭合	小于5的所有整数。 与[.. 4]相同。
	[<= 5]	无限	左开放, 右闭合	小于等于5的所有整数。 与[.. 5]相同。
	[!= 5]	无限	左右开放	除5之外的所有整数。
	[>= 1] and [<= 3]	3	左右闭合	由三个整数1、2和3组成的集合。 与[1, 2, 3]或1 .. 3]相同。
	[<= -10] or [>= 10]	无限	左右开放	绝对值大于等于10的所有整数。
	not [1 .. 3]	无限	左右开放	除1、2和3之外的所有整数。

[0166] [$>=1$]和[>0]之间没有区别,因为元素是整数类型,并且整数可以明确地枚举。如果元素是不可数类型的,例如实数或字符串,则必须明确包含或包含特定端点。例如,集合[$>=$ “cat”]由字符串“cat”和在“cat”之后按字典顺序排序的所有字符串组成。

[0167] 我们可以使用集合的实例作为类型。这种类型的实例必须是该集合的成员。例如,用作类型的集合[>0]将仅允许正整数作为值。事实上,人们可能会以这种方式思考所有类型。例如,单态整数类型可以被认为是包括所有整数的集合的集合类型。

[0168] 我们的总和类型是其他类型的简单联合。例如,整数或字符串类型是其两种组成类型的总和。任何总和类型的组成类型的任何实例也是总和类型的实例。例如,这允许我们描述类型列表(整数或字符串)(int or string),它是值列表,每个值都是整数或字符串。联合的并集变平,因此类型表达式(整数或字符串)(int or string)或(整数或实数)(int or real)等效于整数或实数或字符串。联合中类型的顺序并不重要,但为了规范性,我们在此提供所有联合类型,使得它们的组成按字母顺序排列。

[0169] 我们的记录类型使用命名字段并将每个字段与类型相关联。例如:{birthday: date;first_name:string;last_name:string}。记录类型始终具有有限数量的字段,并且

每个字段在该类型中具有唯一的名称。字段的顺序并不重要；`{x:int;y:int}`与`{y:int;x:int}`相同；但是，正如我们为联合所做的那样，我们将按字母顺序列出其组成的记录类型。

[0170] 请注意，记录的类型本身就是记录。值`{x:3;y:4}`的类型为`{x:int;y:int}`。

[0171] 我们的模式类型与元组类似，因为它被定义为类型序列；然而，虽然元组隐式假设它的每个元素只出现一次，但是模式允许它的每个元素都重复出现。重复出现作为整数集合给出。例如，模式`<a:int#[1..3];b:string#[1..3]>`匹配一到三个整数，后跟一到三个字符串。

[0172] 当用作 λ 的参数时，模式的字段会产生在 λ 评估中绑定的参数。例如，在我们匹配前一段中给出的模式之后，我们将在范围内有两个本地标识符`a`和`b`。`A`的值是一到三个整数的列表，并且`b`的值是一到三个字符串的列表。

[0173] 对于模式中的一个或多个字段没有名称也是有效的。没有名称匹配但没有值的字段作为参数绑定。例如，如果我们匹配`<a:int#[1..3];string#[1..3]>`，我们将按照之前的方式匹配--一到三个整数后跟一到三个字符串--并将整数绑定为一个名为`a`的列表，但我们不会绑定字符串。

[0174] 模式可以具有无限长度。例如，模式`<a:int#[1..]>`将匹配一个或多个整数而没有上限。这是有效的；但是，如果用于处理无限输入流，则无限模式必须与其他触发配对，例如时间间隔，它将指示何时停止收集值。

[0175] 通常，模式将消耗它匹配的数据；但是，可能只消耗该数据的一部分，或者根本不消耗。模式可能包含`at`标记，称为顶点，超出该标记，它将匹配数据并绑定参数，但不会从输入流中消耗。例如，模式`<a:int;b:int;顶点;c:int>`将匹配三个整数并绑定三个本地标识符，但只消耗输入中的两个整数。

[0176] 拥有没有字段的记录或没有字段的模式是有效的。这两种情况在意义上彼此无法区分，因为它们都表示产品类型。从词汇上讲，我们用关键字`空 (void)`指定这个概念。空是一种独特的值；它也是它自己的类型。在联合 (union) 中使用时，空会产生一个可选类型的概念，例如整数或空，这意味着一个值，如果存在则为整数，但可能根本不存在。

[0177] 就我们的目的而言，类型匹配是结构性的，而记名性的。类型没有名称，只有描述。具有相同描述的两类型是相同类型。描述是另一种类型的子集的类型是该类型的概括。例如，考虑类型`{x:int;y:int}`和`{x:int;y:int;z:int}`。具有两个字段--`x`和`y`--的类型是具有三个字段--`x`、`y`和`z`--的类型的子集，因此前者可以被认为是后者的概括。对于模式也是如此。作为另一个模式的前缀的模式也是它的概括。

[0178] 我们的 λ 类型将输入模式映射到输出类型。例如`<int#[1..3]>→int`，这是一个函数的类型，它取一到三个整数并返回整数。我们的多 λ 类型由 λ 类型列表组成。 λ 的顺序在这里很重要。当我们解析多 λ 应用程序时，我们将调度到匹配的组成 λ 的第一个。

[0179] 以这种方式定义，分派多 λ 所需的模式匹配可以简化为确定性有限自动机 (DFA)。为了演示如何简化，我们将使用状态机构造方法作为比较的基础，并在必要时对其进行扩充。描述包括首先构造一个非确定性有限自动机 (NFA)，然后将其简化为DFA；然而，实际上，这通常可以在一个步骤中完成。

[0180] 如前所述，该申请使用术语DFA，但是这些自动机或单元可以称为栈机器。严格地说，确定性有限自动机意味着有限的空间性能。然而，该专利中的自动机不一定是有限的，

但可以是非限定但仍然简单的。因此,该专利中描述的DFA可以是非限定的。

[0181] 首先,多 λ 的组成--各个模式--必须被认为是替换的元素。在翻译正则表达式时,语法 $a|b$ (a OR B) 是替换:匹配 a 1105或匹配 b 1110。在我们的例子中, a 和 b 每个都是 λ 模式。我们根据图11构建用于替换的子图。

[0182] 我们首先通过连接表示单个模式的字段。在翻译正则表达式时,语法 ab 1210是一个连接:匹配 a 1205,然后是 b 1215。在我们的例子中, a 和 b 每个都是模式的字段。我们根据图12构造了用于连接的子图。

[0183] [177]字段的重复因子与正则表达式中的闭包相同,通常写为 a^+ 或 a^* 或 $a\{n:m\}$ 。同样,我们可以使用类似于图13中的结构来表示这些闭包。在这种情况下,基于重复集的值,子图中的一些变化是必要的。例如,如果该集合包括零,则仅包括从节点 i 1305到节点 j 1310的前向 ϵ 1315。这些变化在很大程度上是显而易见的,并且与此处提出的相同基本思想一致。

[0184] 中间NFA完成后,我们将其减少为DFA,然后状态减少DFA,直到达到最小DFA。然后,我们将DFA表现为状态-动作表,适用于自动化状态机中常用的软件或硬件的自动化。此表的接受状态标记多 λ 的入口点,并且中间状态提供用于绑定参数的数据集合。

[0185] 当DFA如此自动化并被提供输入流时,它将匹配来自流的输入的前缀并调度到正确的重载以处理它们,从而产生计算结果。如果允许重复此过程,则结果是一系列产生结果,每个来自输入流的匹配一个产生结果。这通过由数据流中检测到的各种类型的参数的相应模式触发的多态函数提供输入数据流的高效实时处理。

[0186] 一种用于响应于包含多种函数参数的数据流来调度多态函数的执行的方法,包括值和类型标识符的混合,包括:(i) 识别要执行的多态函数,所述多态函数具有多个重载,每个重载与不同种类的参数模式相关联。(ii) 通过匹配重载的参数模式,为每个重载标识要在输入流绑定的参数值集合上评估的输出表达式。(iii) 将每个重载的参数模式转换为DFA,它将有效地识别输入流中模式的匹配。(iv) 将单个重载的DFA组合成单个DFA,用于作为整体的多态函数,得到的组合DFA能够匹配各个DFA匹配的任何模式,并选择应处理匹配输入的重载。(v) 将数据流应用于组合DFA,然后DFA根据需要从流中检查或消费数据或两者以确定匹配或不存在匹配,并且在匹配的情况下,适当地绑定输入参数值并选择要评估的适当输出表达式。(vi) 调度输出表达式的评估并返回结果。

[0187] 给定由反应函数产生的一组不同类型的数据流,本发明是一种表示那些流的技术,使得它们的输出可以有效地组成单个统一类型的流。

[0188] 解决这类问题的需要通常出现在所有形式的数据流编程中。它适用于非常大规模的体系结构,例如企业数据中心内部和之间的数据流,以及非常小规模的结构,例如嵌入式设备中的事件流。

[0189] 本发明适用于数据流编程的所有领域;然而,它最适合于能够检测到匹配并且应用处理函数的速度具有最高重要性的情况,以及用于执行的存储和计算资源有限的情况。

[0190] 示例。考虑由一组 n 个单独的输入流组成的流入, $A_i:0 < k < n$ 。每个流由 T_i 类型的元素队列组成。每个流由 $T_i \rightarrow U_i$ 类型的反应函数 f_i 消耗和变换,这样存在流出的 n 个流 B_i ,每个流由 U_i 类型的元素队列组成。我们希望使用 $\sum T_k \rightarrow \sum U_k$ 类型的合并函数 m 将所有流 B_i 合并到单个流 C 中。

[0191] 以下是使用Vel语言编写的三个流之间发生此类合并的示例：

[0192]

```
B0 = f0 from A0
B1 = f1 from A1
B2 = f2 from A2
C = B0 or B1 or B2
```

[0193] 流C将由来自B0、B1和B2的值组成，在生成它们时交织。请注意，没有必要实现B流的内容，因为它们仅用于组成C流。它们可以很容易地表示为匿名的临时子表达式：

[0194]

```
C = (f0 from A0) or (f1 from A1) or (f2 from A2)
```

[0195] 本发明描述了将每个变换函数 f_i 转换为确定性有限自动机(DFA)，并且将作为这些DFA的联合的合并函数 m 转换为单个最小的DFA。结果是将流入 A_i 合并到流出C的最高效的手段，而不必实现中间流 B_i 的内容。

[0196] 可以重复应用该技术，将后续的中间流层合并为单个反应函数。这与在声明性数据流语言中由中缀或运算符表示的合并的概念一致，如Vel中的情况。

[0197] 这个问题可以通过蛮力来解决；即，通过实现中间流然后消耗它们，即使合并功能是中间流的唯一消费者。

[0198] 通常情况下，合并功能要求其流入和流出都是相同类型的，或者在无类型系统的情况下的无差别类型。这是由于类型系统中缺少联合类型(也称为求和类型)。数据流系统中存在真正的合并要求使用联合类型。

[0199] 某些数据流系统缺少真正的合并，而是实现多输入单输出反应变换。尽管这些构造本身是有用的构造，但它们并不像真正的合并函数那样简单或通用，并且不能完全优化。

[0200] 将匹配函数表示为DFA比将其表示为布尔类型的任意表达式更高效。多个匹配函数的DFA(每个都具有其自己的驱动流入)被统一以形成单个高效DFA，其表示具有单个流出的合并功能。可以进行DFA的合并，使得结果尽可能早地匹配或尽可能晚地匹配，从而产生两种不同的、可能期望的行为。将多个反应组合成单个DFA会产生最小的机器；也就是说，使用最少数量的决策执行所有匹配的算法。对于小型平台而言，最小机器最适合实现多个反应。最小机器比匹配表达式的多个单独评估具有算法优势，因此，在其他条件相同的情况下，将更高效地执行。

[0201] 要将转换DFA集合并为单个DFA，我们必须将它们视为正则表达式中的替换。在翻译正则表达式时，语法 $a|b$ 是替换：匹配 a 或匹配 b 。在我们的例子中， a 和 b 每个都是来自转换函数的DFA。我们根据图11构建了替换的子图。

[0202] 在中间非确定性有限自动机(NFA)完成之后，我们将其减少为DFA，然后将DFA状态减少直到达到最小DFA。然后，我们将DFA表现为状态-动作表，适用于自动化状态机中常用的软件或硬件的自动化。此表的接受状态标记合并数据元素发送到输出流的点。

[0203] 当DFA如此自动化并被提供一组输入流时，它将根据与该输入相关联的原始变换函数变换每个输入，并将所有结果在单个输出上交织在一起。

[0204] 一种用于将多个独立的输入数据流合并成单个输出数据流的方法，包括：(i) 识别多个潜在的输入数据流。(ii) 识别多个变换函数，每个输入流一个，这些变换函数将对每个输入流中的数据执行，并且其结果被合并在一起。(iii) 识别合并函数，该合并函数同时从

多个流接收输入数据元素并将数据元素交织成单个输出流。(iv) 将每个变换函数转换为 DFA, 其将高效地执行变换。(v) 将变换 DFA 合并为单个组合 DFA, 其将高效地执行变换并将结果交织成单个流。(vi) 将数据流应用于组合 DFA, 然后 DFA 执行变换和合并工作。(vii) 将合并的输出调度到目的地以供使用。

[0205] 本发明是用于以 Vel 编程语言开发软件的工具和相关方法。Vel 是一种用于表达数据流程序的编程语言。正确的数据流编程带来了许多挑战。有些是所有形式的计算机编程常见的挑战, 而其他则是特定于数据流范例的。该工具解决了 Vel 编程的许多方面, 包括: (1) 检查语法和语义的正确性。(2) 检查逻辑正确性。(3) 调试协助。(4) 将源代码翻译成安全可移植的形式 (即打包代码)。(5) 将源代码或打包代码翻译成适合各种计算平台 (特别是小平台) 的本机和最佳二进制形式。(6) 描述打包的代码并确认其签名。(7) 打包代码的批处理模式解释。(8) Vel 源的交互解释。(9) 模拟运行打包代码或本机代码的数据流环境。(10) 在实时数据流环境中远程执行、监视和控制二进制代码。

[0206] 这些是任何以 Vel 语言开发软件的人都需要完成的任务。本发明在所有这些领域中提供了足够的支持, 以使精通 Vel 编程的人能够生产正确和有用的软件。

[0207] 检查句法和语义正确性是许多形式的自动软件翻译所共有的任务。用于检查逻辑正确性的工具通常不包含在翻译工具本身中。这些类型的工具通常是单独存在的, 通常对它们正在测试的代码有不完整的了解。

[0208] 虽然调试是软件开发中的常见任务, 但大多数调试工具都专注于命令式编程。功能和反应式编程的调试不太常见, 因为它提出了与命令式调试截然不同的挑战。特别是, 在这些语言中检查“飞行中”的计算可能很困难, 因为它们的值通常没有调试器 (和调试程序员) 可以窥视的地址。

[0209] 对于系统语言的编译器 (例如 C) 来说, 定位多个本机平台体系结构的能力并不少见, 但它不是在脚本级语言中常见的能力。脚本语言往往不会被编译, 或者为其主机进行部分编译或即时编译 (jitted), 但是交叉编译 (在一个体系结构上运行但在另一个体系结构上生成代码的编译器) 并不常见。专门编译脚本级语言以便在小型平台上执行是非常罕见的。

[0210] 交互式壳 (shell) 是脚本语言的共同特征。例如, Python 实现了壳。连接到数据流环境 (实际或模拟) 的壳远不常见。

[0211] 编译代码的远程执行是某些操作系统的特征, 也可以从几个第三方工具获得, 包括开源和商业。这些往往不是专门针对小型平台, 但确实存在一些针对小型平台的远程执行工具的示例。它们不是特定于数据流编程的, 并且不包含在用于开发远程执行程序的工具中。

[0212] 用于开发 Vel 代码的单一集成工具对于使用 Vel 语言的软件开发人员来说非常有用和方便。该工具主要是一个编译器, 翻译 Vel 语言, 但它还提供了与 Vel 编程相关的几组其他功能。让工具执行逻辑正确性测试以及语法和语义正确性测试有助于开发人员提高效率并提高代码的正确性。逻辑测试利用编译器对代码的洞察力, 因此诊断消息可以更完整。交互式壳使开发人员能够测试代码并立即得到响应。这对于开发和调试很有用。壳还为程序员提供了对数据流环境的可视性。

[0213] 生成适用于小型平台的独立二进制可执行代码实现了物联网用例, 其通常依赖于

在各种小型设备上执行复杂计算。提供模拟数据流环境有助于开发人员解决代码中的错误,并与逻辑正确性测试协作,证明程序包正常运行。远程执行已编译的包,特别是当远程平台很小时,允许程序员快速迭代他的程序,在单个命令中编译和测试目标硬件上的程序,即使目标平台不是他正在其上进行开发的平台。

[0214] 将语言从其词汇表示转换为中间符号表示(阶段1编译),然后将该中间表示转换为可由计算硬件执行的形式(阶段2编译)的过程。

[0215] Ve1阶段1翻译工具遵循编译器常用的一般策略,具体为:(1)分析输入字符串以将其分解为令牌序列。(2)分析令牌序列以形成语法树。(3)识别树中的符号声明。(4)识别和解析树中的符号引用。(5)早期优化,例如常见的子表达式消除和常量折叠。(6)类型检查。(7)优化和符号成熟的附加阶段。(8)符号的最终确定和中间表示的发出。

[0216] Ve1阶段-1翻译器的一个显著特征是使用确定性有限自动机或DFA来执行函数应用所需的模式匹配并触发反应。阶段1翻译工具包括:(1)语法分析器,将输入语言转换为语法树。(2)词汇绑定组件,允许翻译中的程序进行自我引用,以便分析中的语言可以通过分析器(DSL或宏分析器的方式)进行修改。(3)语义分析算法,将绑定语法树翻译成代表数据流、模式、反应、函数表达式、定时器和输入/输出参数化的符号。(4)表达式转换器,它将表达式树转换成适合于或多或少直接转换成微处理器ALU指令的栈。(5)DFA生成器,用于将反应的模式和表达式转换为潜在的非最小DFA的中间集合。和(6)DFA组合和简化算法,用于从DFA的中间集合生成统一的、最小的DFA。

[0217] 阶段1翻译工具的输出包括:(1)翻译中涉及的每个流的逻辑标识,使得每个流可以是多个流中的唯一指示对象。(2)每个流中数据流的描述,每个流向内(朝向反应;即,对外部源的订阅),向外(远离反应;即,到外部目的地的发布),向内和向外(发布/订阅对)或内部(仅用作其他反应的中间步骤,因此不作为发布或订阅表现)。(3)每个流中流动的数据类型的描述,每次用有限项描述,使得可以静态地检查被注入流中或从流中提取的数据的类型正确性。(4)描述DFA状态和转换的表的集合。(5)表达式栈的集合,描述在反应期间要执行的计算。(6)映射流输入到DFA输入的表。(7)将定时事件映射到DFA输入的表。(8)将DFA输出映射到动作对的表,每对包括对表达式栈和流输出的引用,从而指示DFA的输出将由给定表达式转换,然后推送到给定流。

[0218] Ve1解释器和数据流模拟器直接使用阶段1翻译的输出。解释器在执行代码时模拟硬件平台,数据流模拟器模拟流数据环境,为Ve1流提供输入和收集Ve1流的输出。让我们将这两个任务称为指令解释和数据流仿真。

[0219] 指令解释是专门编写编译器和解释器的计算机程序员很好理解的一类任务。该任务包括构造执行上下文,其中可以存储运行中变量的状态,然后一次一个地逐步执行程序的指令,从执行上下文访问数据并根据需要更新它。

[0220] 在Ve1的情况下,执行上下文还必须包括一组队列以在转换过程中保存数据流,并且包括表驱动的状态机引擎以执行由DFA描述的转换。队列由于Ve1源中的声明而产生,其描述了数据的流动通道。其中一些是Ve1程序的外部输入或输出,而其他是纯内部通道,描述输入和输出之间的中间状态。

[0221] 数据流仿真包括提供对诸如文件或套接字之类的数据的外部源和接收器的访问,以及在外部系统和解释下的Ve1程序之间交换数据所需的编程。这将包括注入器功能,

它从外部源读取数据并将它们推送到表示程序输入的队列,以及提取器功能,它们从表示程序输出的队列中弹出数据并将它们写入外部接收器。

[0222] 其中,根据本发明的Ve1解释与规范的不同之处在于涉及DFA的方式。状态机引擎从队列中读取数据并使用它们来推进其DFA的状态。DFA表包含副作用的列,这些副作用在DFA通过其状态时执行。这些副作用调用指令解释来执行计算,其结果被推送到其他队列并且这触发其他DFA。

[0223] 以这种方式,根据本发明的解释下的Ve1程序首先由一组快速且小的状态机表示,并且在必要时仅回退到一般指令解释。这使得程序的执行效率高于仅由指令解释处理的程序。

[0224] Ve1阶段-2翻译工具在很大程度上不是特定于Ve1语言,而是特定于执行目标的平台。阶段2翻译器的Ve1语言相关组件是:(1)阶段1产生的中间表示的初始摄取。(2)生成反应系统的阶段-2代码生成的整体组织。(3)提供运行时支持组件库,例如执行数据格式的外部编码和解码或实时时钟的内部调节的组件。

[0225] 一种用于在多源多目的地数据流环境中创建用于实时处理数据流的程序的工具,包括:(1)识别多个潜在数据流。(2)识别与流中的数据模式相对应的一组反应函数和参数。(3)识别一组处理函数和参数,用于转换与声明的模式相匹配的数据。(4)识别比较数据流模式的一组定时事件,例如收集或丢弃数据的时间间隔或收集或丢弃数据之前或之后的特定时间点。(5)创建描述已识别的流、反应、函数和定时事件的数据流程序。(6)提供程序作为两阶段翻译工具的输入,包括阶段-1翻译工具和阶段-2翻译工具,阶段-1翻译工具包含用于将Ve1程序语句翻译成相应DFA的DFA生成器,阶段-2翻译工具用于生成对应于翻译的Ve1语句以供在平台上执行的特定于平台的硬件指令。(7)接收翻译工具各阶段的输出。

[0226] 解释器组件可以使用阶段-1翻译工具的输出,包括:(1)在执行代码时模拟硬件平台的指令解释器。(2)数据流模拟器,它模拟流数据环境,为Ve1流提供输入和收集Ve1流的输出。

[0227] 阶段-1翻译工具的输出可以用作阶段-2翻译工具的输入,包括:(1)硬件指令生成器,其将来自中间表示的指令翻译成适合于目标硬件平台的执行的形式。(2)程序组织模块,其将输出的生成指向适合用作数据流环境中的反应程序的形式。(3)执行所必需的运行支持组件库。阶段-2翻译工具的输出是适用于目标硬件平台的可执行程序。

[0228] 给定一组不同类型的数据流和一组用于响应和处理那些流中的特定数据模式的函数,本发明是一种描述和转换这些函数以便当数据到达流时可以适当且高效调用它们的技术。

[0229] 解决这类问题的需要通常出现在所有形式的数据流编程中。它适用于非常大规模的体系结构,例如企业数据中心内部和之间的数据流,以及非常小规模的结构,例如嵌入式设备中的事件流。

[0230] 本发明适用于数据流编程的所有领域;然而,它最适合于能够检测到匹配并且应用处理函数的速度具有最高重要性的情况,以及用于执行的存储和计算资源有限的情况。

[0231] 示例。例如,假设从给定的整数流中我们希望匹配一个或多个非零值,跟着是一个或多个零。当匹配此模式时,我们希望计算非零值的总和,并将结果写入另一个流。

[0232] 我们可以用类似正则表达式的符号来编写这个问题的模式匹配部分,然后单独将

和的计算写成算术表达式。此时,设计用于边缘计算中的数据流应用程序的Vel编程语言允许我们以统一的表示法编写整个变换,因此:

```
[0233] 1. stream foo is int
        2. p = pn a: {!= 0}... :0.. -> sum(a)
        3. bar = p(foo)
```

[0234] 在第1行,我们声明foo是一个整数流。在第2行,我们将p定义为匹配非零值后跟零的模式,然后计算非零值的总和。在第3行,我们将p应用于foo来定义一个新的流条,应用程序的结果将被推送到该流条。

[0235] 该技术将上述模式转换为状态机。然后,它将基于该状态机将匹配实现为确定性下推自动机,并将得到的匹配馈送到求和表达式。

[0236] 上述示例中描述的那种问题可以通过生成两个函数来解决:一个用于匹配,一个用于在找到匹配之后计算结果。匹配函数接受来自流的数据窗口作为输入,并且对于匹配返回真(true),对于非匹配则返回假(false)。找到匹配项后,数据窗口将传递给结果计算函数以生成输出。

[0237] 当数据流过窗口时,必须重复应用匹配函数,直到找到匹配为止。因此,效率受限于可以执行匹配函数的速度。在特定调用中由匹配函数计算的任何值通常不会在以后的调用中重用,当找到匹配时,它们通常也不能由结果计算函数使用。缺乏对可能有用的中间计算结果的保留是显着低效的潜在来源。

[0238] 当在诸如流式传输SQL之类的语言中指定流处理时,这种双函数安排直接产生。流式SELECT语句的where子句提供描述匹配条件的布尔表达式,匹配函数是此表达式的直接编译。SELECT的product子句中指定的值元组然后提供了定义输出生成函数的基础。

[0239] 使用状态机来执行匹配比重复应用多个任意布尔表达式更高效。

[0240] 本发明从声明函数的参数的模式描述语言中导出状态机。与传统的布尔表达式匹配函数相比,所导出的状态机更高效地检测数据流中的匹配。

[0241] 所导出的状态机还可以实现一组处理函数,以基于在数据流中检测到的匹配来产生输出。可以组合多个匹配和相应的处理函数并将其简化为单个状态机,其高效地同时识别许多模式的匹配并同时产生多种输出。

[0242] 组合正则表达式和值表达式的各方面的脚本可以自动转换为确定性状态机,其可以通过保留重要中间计算的结果来高效地匹配模式和计算输出。得到的组合匹配/计算算法比模式匹配和输出生成的单独组织更高效。

[0243] 因此,总体方法是:(1)从源脚本开始,该源脚本使用正则表达式类的表示法指定匹配,并且使用函数表达式类的表示法指定基于这些匹配的输出生成。(2)将脚本的正则表达式类部分转换为一组语法规则,并将脚本的函数表达式类部分转换为这些语法规则的一组属性。(3)从语法和属性生成状态机。(4)将状态机实现为下推自动机,向其流入要匹配的输入并从中流出结果。

[0244] 一种技术称为定时前向最右(1)模式匹配或TFR(1)。以下讨论指定了TFR(1)模式匹配状态机,并详细说明了如何应用它来完成步骤4。以下部分描述了如何根据步骤3的目标将语法转换为TFR(1)状态机(即,从语法和属性生成状态机)。

[0245] 从源脚本转换为一组语法规则和属性的特定方法必然依赖于源脚本语言本身,因

此超出了本文档的范围。然而,语法规则和属性的使用是计算机语言实现中的常用技术,因此可以合理地假设步骤2不会给任何实现者带来太大的障碍。

[0246] TFR (1) 模式匹配

[0247] 在本节中,我们提出了TFR (1) 模式匹配,这种技术可用于基于一大类无上下文语法而有效地识别流输入的模式。缩写代表“定时前向最右 (1) (Timed Forward Right-most (1))”,指的是技术的时间感知性质,事实上它以前向顺序 (相对于时间) 扫描输入,并产生最右边的句法推导,而一次只查看一个输入令牌。

[0248] 概述。TFR (1) 模式匹配器 (下文中简称为“匹配器”) 的目的是通过产生输出流来对输入流作出反应。提前描述了预期输入的模式,并且预先构建了匹配该模式的匹配器。在匹配输入模式之后,匹配器基于匹配的输入产生一个或多个输出。

[0249] TFR (1) 匹配器具有若干显着特征: (1) 匹配器可以从一组语法类规则以算法方式构造,该规则描述要匹配的模式和要应用的转换。(2) 匹配器可以在匹配过程中加入超时概念,使得匹配由于时间的推移而进行。(3) 匹配器可以从任何明确的无上下文语法以算法方式构造。(4) 匹配器紧凑且高效,适用于存储有限或低端处理器或两者兼有的计算机上的实现。它需要最少的步骤来匹配输入。(5) 匹配器在没有回溯的情况下运行,在输入中严格向前移动,使其适合于处理流动的流。(6) 匹配器能尽快检测到不匹配 (错误) 输入,并且可以快进通过它们,与输入流重新同步,并恢复匹配。(7) 匹配输入序列后,匹配器可以使用匹配数据作为任意函数的参数,其结果作为输出产生。

[0250] 具有时间和无尽的语法。可以在语法上分析流中流动的数据。该方法类似于解析计算机源代码时使用的语法分析。事实上,TFR (1) 类似于规范LR (1) 解析算法,并且对规范LR (1) 以及通常的句法分析的熟悉将对任何对TFR (1) 感兴趣的读者有所帮助。

[0251] 流以两种方式与文件不同: (1) 流中的每个数据元素与其进入流的特定时间相关联 (隐式或显式),而文件中的数据元素没有与之相关联的具体时间。(2) 流不需要有明确的结束但可以无限期地继续流动,而文件具有固定的大小,因此它的结束总是可以预期的。

[0252] 与大多数其他解析算法一样,LR (1) 意在对文件进行操作。它不考虑时间,并且取决于文件结束标记以确保正确操作。但是,TFR (1) 旨在与流输入一起使用。它知道时间,并且不需要文件结束标记。

[0253] 输入

[0254] 输入队列是要一次一个地处理的离散、传入事件的先进先出非终止序列。这可能是网络套接字、串行端口、内存中结构,或无限队列或流的概念的任何其他材料或抽象实现。匹配器的目的是识别此流中的模式。

[0255] 每个都带有一个戳,其标记令牌进入队列的时间,以及可能的其他一些信息。有三种事件:令牌 (token)、流逝 (lapse) 和中断 (break)。

[0256] 令牌是携带要分析的数据的事件。它是迄今为止最常见的一种事件,因此术语“事件”和“令牌”有时可以互换使用。令牌携带的数据元素可以是任何强数据类型的实例,并且特定输入队列中的所有令牌携带相同类型的数据。令牌通常携带单个字节或字符 (当匹配器解码有线格式或识别单词时) 或多值数据记录 (当匹配器正在分析来自传感器的多维数据时)。

[0257] 在某些情况下,令牌类型的可能状态的数量非常大。8位字节不够,它只有256种可

能的状态,而是需要更大的类型,例如32位整数或由许多字段组成的记录。在这些情况下,通常需要提供令牌分类器函数。给定令牌,分类器函数返回令牌所属的类。例如,模式可能涉及两类32位整数:零和非零。然后,驱动器在决定要执行的操作时使用令牌的类来代替令牌本身。

[0258] 流逝是除了其戳之外不携带任何数据的事件。从本质上讲,它是一个“非事件”,只表示时间的流逝。在输入队列中包含这种事件是有用的,因为它区分了活动接收事件、即使没有要报告的数据的输入队列和什么也没接收到的输入队列。

[0259] 中断是指示输入队列中的瞬时中断的事件。匹配器通过结束正在进行的任何部分匹配并立即报告其输出来响应中断。中断不是输入结束标记;中断之后可以(并且通常会)有其他事件。它只是一个可以随意注入输入流以加速匹配器动作的事件。

[0260] 规则。我们将语法写成一系列规则。在每个规则中,缩减出现在左侧,右侧出现一系列符号。每个符号都是令牌或缩减。例如:

[0261]	$(r1) E \rightarrow E '+' E$ $(r2) E \rightarrow \text{num}$
--------	--

[0262] 这两个规则描述了一个由令牌num和“+”组成的语言和缩减E。通过r1,每次输入中出现num,它可以被认为是E。通过r2,每次我们看到输入中的E+E序列,我们可以把这三个符号共同组成新的E。因此序列num、num+num和num+num+num都是该语言的有效句子。

[0263] 规则也可以是空的,这意味着它的右侧没有符号。这对于将特定语法片段视为可选的情况非常有用。例如:

[0264]	$(r3) L \rightarrow L 'x'$ $(r4) L \rightarrow \lambda$
--------	---

[0265] 我们在右侧写下小写λ来表示空。在这种情况下,缩减L由本身后跟“x”构成(通过r3),或者什么都没有(通过r4)。因此,序列x、x x和x x x都是该语言的有效句子。空序列也是该语言的有效句子,因为L会匹配它。指定超时的空规则也是可能的。例如:

[0266]	$(r5) C \rightarrow C 'x'$ $(r6) C \rightarrow \tau_5$
--------	--

[0267] 这种语言的C缩减与上述语言的L缩减非常相似;然而,L将无限期地收集x,而C将仅在有限的时间内收集x。这里给出的时间量为5个单位。该单位可以是任何方便的时间量度。在实践中,可能以毫秒为单位给出超时。要了解这里的差异,考虑带有时间戳的输入序列:

[0268]	1000	'x'
	1001	'x'
	1002	'x'
	1003–1009	no input
	1010	'y'

[0269] 这里我们有三个x快速连续到达,然后是一个间歇,然后是“y”。使用r3和r4匹配此输入,匹配器将在时间1010报告L,因为“y”的出现表示x字符串的结束。然而,使用r5和r6,

匹配器将在时间1005报告C,因为仅时间的流逝就足以调用缩减。

[0270] 直观地说, λ 规则(没有超时的空规则)将等待直到它能够看到下一个令牌再决定做什么,这意味着它将无限期地等待。 τ 规则(具有超时的空规则)也将等待下一个令牌,但如果该令牌不能足够快地到来,则该规则将决定在其不存在时执行动作。

[0271] 属性。属性是在一组语法规则上定义的函数。由于匹配器应用规则来执行缩减,因此它还应用该规则的属性函数。结果是一个值,可用于计算其他属性或成为匹配器的输出,或两者。此过程称为属性合成。让我们再看一下上面的一个示例语言:

[0272]
$$\begin{aligned} (r1) \ E &\rightarrow a:E \text{ '+' } b:E \\ (r2) \ E &\rightarrow a:\text{num} \end{aligned}$$

[0273] 此处识别的语言与前一个示例相同;但是,我们在符号中添加了明确的名称。我们可以在属性的定义中使用这些名称:

[0274]
$$\begin{aligned} \text{int val } \{ \\ \quad r1: a, \\ \quad r2: \text{val}(a) + \text{val}(b) \\ \} \end{aligned}$$

[0275] 这里我们定义val,一个产生int类型值的属性。当应用r1时,val的值被定义为简单的数字a,它是从输入流中获取的文字上的数字。当应用r2时,递归定义val的值,评估a和b的val并对结果求和。符号a是b是E的实例。

[0276] 直观地,属性合成开始于可以从单个令牌(或甚至仅仅是令牌本身)导出的值。当匹配器应用规则并产生缩减时,合成继续进行更大范围的操作。高级别缩减的属性直接或间接地取决于它跨越的任何或所有令牌。

[0277] 增强语法和重启。增强语法是恰好包含一个 Ω 规则的语法。 Ω 规则类似于普通语法规则,除了它右边只有一个符号,左边有特殊的 Ω 符号。例如:

[0278] 这里我们通过添加r0, Ω 规则,来扩充前面示例中的语法。

[0279]
$$\begin{aligned} (r0) \ \Omega &\rightarrow E \\ (r1) \ E &\rightarrow E \text{ '+' } E \\ (r2) \ E &\rightarrow \text{num} \end{aligned}$$

[0280] 语法的 Ω 规则告诉匹配器从哪里开始和结束。首次启动时,匹配器会尝试匹配 Ω 规则右侧的符号。在这种情况下,r0表示匹配器最初会尝试匹配缩减E。

[0281] 在匹配 Ω 规则的右边的符号后(或从未能匹配它恢复之后),匹配器将自动重新启动,并尝试再次匹配符号。因此,匹配器无限期地匹配 Ω 规则。

[0282] 输入队列中的中断事件使匹配器接受可能目前待决的任何 λ 或 τ 规则并且立即进行到缩减到 Ω 规则。然后匹配器重新开始接受中断后的事件。

[0283] Ω 规则必须出现在一个属性的定义中。此属性用于计算匹配器的输出,如下一节中所述。

[0284] 输出。输出队列是离散的传出事件的先进先出非终止序列。输出队列在概念上类似于输入队列,但是从匹配器流出而不是流入。像输入队列那样,输出队列可以是一个网络套接字、串行端口、内存内结构,或无限队列或流的概念的任何其它材料或抽象实现。匹配

器的目的是将事件发送到此队列。匹配器从不尝试从输出队列中读取。

[0285] 输出队列中的每个事件都包含戳,用于标记匹配器生成事件的时刻。每个事件还携带一个数据元素,该数据元素可以是任何强数据类型的实例,并且特定输出队列中的所有事件都携带相同类型的数据。这可以是与输入队列中的事件相同的类型,也可以是不同的类型。

[0286] 当匹配器通过 Ω 规则缩减时,匹配器评估 Ω 规则的 (唯一) 属性并将结果作为事件推送到输出队列。如此生成的输出事件将具有等于最近输入事件的戳。

[0287] 体系架构。图14显示了TFR (1) 模式匹配器的组件。每个TFR (1) 匹配器包括输入队列1405、输出队列1412、驱动程序1419、状态栈1426和状态表1433。

[0288] 匹配器在输入队列的头部维持读取位置。它一次一个地从队列的前面弹出事件。匹配器从不在输入队列中向前看,也不会尝试将其读取位置倒回到队列中的较早点,也不会尝试将事件推送到输入队列。这种一次一个、仅向前、只读的输入处理是使TFR (1) 非常适合高效处理实时流的部分原因。

[0289] 所有匹配器的驱动程序都是相同的。它是由单独表格中定义的状态驱动的下推自动机。下面给出了驱动程序的算法。驱动程序的每个实现都特定于其主机硬件、操作系统平台和其环境的其他细节;但是,驱动程序并不特定于要匹配的模式。驱动程序总是一样的;它是特定于要匹配的模式的状态表。

[0290] 驱动程序使用状态栈来跟踪状态表定义的状态的进度。栈中的第一个条目总是包含对表中第一个状态的引用。栈中的每个后续条目包括对表中某个状态的引用、状态必须被解析的截止时间、以及从输入队列中获取的令牌或从语法规则的左侧获取的缩减。

[0291] 状态栈的缩减与可以根据它们的规则来计算的属性相关联。当准备应用新的缩减时,栈顶部的状态提供用于应用与新缩减相关联的属性定义函数的调用帧。这样,状态栈也是函数调用栈。这也意味着中间计算的结果会自动保留供以后使用,但当它们不再具有潜在兴趣时也会被丢弃(当栈被弹出时)。

[0292] 状态表是从一组语法规则和属性以算法方式生成的,该语法规则和属性描述了要匹配的模式和要产生的输出。下面描述生成表的方法。一旦生成,状态表不会更改。驱动程序由表引导,但不会更改表。实际上,驱动程序的多个实例可以共享相同的状态表。如果希望在多个输入流中匹配相同的模式并产生多个输出流,则这将是有益的配置。

[0293] 状态表由有限数量的行组成,每个行定义解析状态。解析状态是一种抽象概念,它定义了语法驱动的模式匹配的决策过程中的不同点。每个解析状态由三部分组成:动作、转换和超时值。

[0294] 解析状态的动作将每个令牌值映射到驱动程序要采取的动作。驱动程序在其主循环中使用表的这一部分。表的这一部分由语法的令牌或令牌分组的类来组成。中断事件也是表格这一部分的组成。

[0295] 解析状态的转换将每种符号映射到解析状态。执行缩减操作时,驱动程序使用表的一部分。表格的这一部分是通过语法的缩减来组成的。

[0296] 解析状态的超时表示允许驱动程序保持在该状态的最大时间量。这可以 (并且通常是) 明确无限,这意味着驱动程序可以在该状态中花费任何时间量。

[0297] 驱动程序算法。驱动程序以包含单个条目的栈开始。该条目指起始状态 (通常表示

为 s_0),其截止时间等于开始时间加上 s_0 的超时。第一个栈条目没有关联的符号,但后面的条目将带有令牌或缩减。

[0298] 然后驱动程序进入其主循环,其无限迭代。在每次迭代开始时,驱动程序查看栈顶部的条目和输入队列前端的事件。

[0299] 驱动程序将当前事件的戳与当前栈条目的截止时间进行比较。如果戳等于或大于截止日期,则状态已过期且驱动程序必须立即尝试离开该状态。在过期条件下,驱动程序会将当前输入视为中断,而不管它实际是什么类型的事件。

[0300] 如果当前事件是流逝(并且当前状态尚未到期),则驱动程序只是丢弃它(见下文)并返回到主循环的顶部。流逝除了允许状态到期之外没有任何其他目的。

[0301] 如果当前事件是令牌或中断(或者如果当前状态已经到期并且我们将当前事件视为中断),则驱动程序查找在当前状态的动作部分要采取的动作。该动作将使输入队列前进到下一个令牌,推送或弹出栈,推送到输出队列或这些的任意组合。执行动作后,驱动程序返回主循环的顶部。

[0302] 驱动程序可以执行五种动作:移位、丢弃、缩减、重启或恐慌。在每次迭代中,驱动程序将执行其中一个。

[0303] 移位动作包括对状态的引用。该动作使驱动程序从输入队列中弹出当前事件,将队列推进到下一个事件。该事件必须是令牌;驱动程序不会移位流逝或中断。然后,驱动程序将新的条目推送到栈,该条目包括由该动作指示的状态、新的截止日期和弹出的令牌。新的截止日期等于以下较小者:当前截止日期,或事件戳加上状态的超时时间。

[0304] 丢弃动作使得驱动程序从输入队列弹出当前事件,使队列前进到下一个事件。驱动程序保持相同的状态。这就是流逝和真正的中断的情况。

[0305] 缩减动作包括对语法规则的引用。缩减动作分三个阶段进行:合成、从栈弹出、和推送到栈。

[0306] 合成。驱动程序评估正在应用的规则的属性。

[0307] 从栈弹出。驱动程序从栈中弹出一个或多个条目。弹出的条目数由所应用规则的右侧长度给出。将弹出之后的栈顶部的状态称为 P ,并将应用规则左侧的符号称为 L 。

[0308] 推送到栈。驱动程序在状态 P 的转换中查找符号 L ,给出新的状态 N 。然后,驱动程序将新的条目推送到栈,该条目包括状态 N 、新的截止日期和符号 L 。

[0309] 重启动作分三个阶段进行:合成、重置栈和丢弃中断。

[0310] 合成。驱动程序评估 Ω 规则的属性并将结果推送到输出队列。

[0311] 重置栈。驱动程序清除栈中的所有条目,然后推送新的起始条目。新条目包括对 s_0 的引用和等于输入事件戳加上 s_0 的超时的截止时间。该条目不包含符号。

[0312] 丢弃中断。如果当前输入是真正的中断(并且由于当前状态已经过期而不被视为一个中断),则驱动程序现在将其丢弃。

[0313] 恐慌动作导致驱动程序丢弃输入,直到它可以重新同步,然后重新启动栈。

[0314] 解析

[0315] 直观地说,匹配器将模式匹配的任务分解为一系列子任务,每个子任务与一条模式匹配。这些任务中的每一个被次第分解,并且这个过程会重复,直到要匹配的部分只是输入令牌本身,这可以简单地匹配。

[0316] 将模式分解成可匹配的片段的任务由匹配器生成器完成,并且结果在状态表的状态中被编码。驱动程序遵循这些状态,从而执行匹配子任务。

[0317] 当驱动程序采取移位动作时,这是因为输入令牌是驱动程序当前匹配子任务的一部分。驱动程序正在扩展当前匹配以包括令牌。

[0318] 当驱动程序采取缩减动作时,这是因为输入标记不是驱动程序当前匹配子任务的一部分。驱动程序正在标记当前匹配的结束并继续到下一匹配,其中输入标记将是下一匹配的第一部分。

[0319] 当驱动程序采取恐慌动作时,这是因为输入令牌不是任何匹配子任务的一部分,并且直到该点的整个匹配状态是错误的。驱动程序丢弃其累积状态以及违规令牌并重新开始。

[0320] 驱动程序的行为完全是确定性的,因此可以在图表中轻松建模。本文档中有几个这样的图表,它们采用一致的表示法。

[0321] 圈。表中的状态。其也显示了状态的超时,其可以是无限的。

[0322] 实线箭头。转位动作。边上的标签表示匹配的令牌。箭头指向下一个状态。

[0323] 向后箭头。缩减动作,弹出阶段。边上的标签表示匹配的令牌。箭头指向正在应用的语法规则。

[0324] 菱形。语法规则。括号中的文本表示从栈弹出的条目数和要匹配的符号。

[0325] 虚线箭头。缩减动作,推送阶段。边上的标签表示匹配的符号。

[0326] 可以使用该图来执行驱动程序的算法。

[0327] 1. 使用单独的纸张跟踪栈。首先将状态s0与第一截止日期一起推送到栈。

[0328] 2. 查看栈顶部的状态和当前输入令牌。沿着从状态引出并用当前令牌标记的边。

[0329] 3. 如果边是指向另一个状态的实线箭头,则将其指向的状态与新截止日期和当前令牌一起推入栈,将输入前进到下一令牌,然后返回步骤2。

[0330] 4. 如果边是连接规则的向后箭头,则查看它连接的规则。该规则将列出一些要弹出的条目和缩减。从栈中弹出指定数量的条目,然后查看新的顶部状态。找到从该状态引出并用缩减标记的虚线边。将虚线边指向的状态以及新的截止期限和缩减推入栈。如果缩减不是 Ω ,那么回到步骤2;否则,返回步骤1。

[0331] 示例。在本节中,我们提供了几个动作中的TFR (1) 的示例。在可行的情况下,我们将其与基于流的模式匹配的其他常见形式(其与流式SQL类似,基于表达式评估)的行为进行比较。

[0332] 示例1:宠物

[0333] 在该示例中,我们考虑查找宠物列表的匹配器。每个清单可以包含任意数量的宠物(或没有宠物),每只宠物是猫(cat)、狗(dog)或金丝雀(canary)。以50个时间单位的间隔从输入流收集列表。

[0334] 表B显示了语法。 Ω 规则(r0)匹配宠物列表。每个列表包括宠物列表,其后是宠物(r5)或无(r4)。这种用于收集列表的语法结构是解析器的一个共同特征,它寻求输入的最右推导。规则r1、r2和r3定义了宠物的含义。

[0335] 表C示出了从该语法导出的状态。诸如“s4”之类的动作意味着“移位并转到状态4”,而诸如“r2”之类的动作意味着“按规则2缩减并跟随转换。”弯曲箭头()表示重新启

动。空白动作意味着恐慌。

[0336] 图15A-15B示出了与表C相同的信息,但是以更直观理解的方式。在这个图中,人们可以很容易地看到,例如,“狗”和其他两种宠物之间的区别在匹配的早期发生,而“猫”和“金丝雀”之间的区别需要另外两个步骤。

[0337] 表B:“宠物”的语法

[0338]	规则	语法
	r0	$\Omega \rightarrow \text{pets}$
	r1	$\text{pet} \rightarrow \langle\langle c \rangle\rangle \langle\langle a \rangle\rangle \langle\langle n \rangle\rangle \langle\langle a \rangle\rangle \langle\langle r \rangle\rangle \langle\langle y \rangle\rangle$
	r2	$\text{pet} \rightarrow \langle\langle c \rangle\rangle \langle\langle a \rangle\rangle \langle\langle t \rangle\rangle$
	r3	$\text{pet} \rightarrow \langle\langle d \rangle\rangle \langle\langle o \rangle\rangle \langle\langle g \rangle\rangle$
	r4	$\text{pets} \rightarrow \tau 50$
	r5	$\text{pets} \rightarrow \text{pets pet}$

[0339] 表C:“宠物”的解析表

[0340]

状态	动作										转换		超时
	a	c	d	g	n	o	r	t	y	—	pet	pets	
s0		r4	r4							r4		s1	$\tau 50$
s1		s3	s4							$\emptyset$	s2		$\tau \infty$
s2		r5	r5							r5			$\tau \infty$

[0341]

s3	s5											∞
s4					s6							∞
s5				s7			s8					∞
s6			s9									∞
s7	s10											∞
s8		r2	r2						r2			∞
s9		r3	r3						r3			∞
s10						s11						∞
s11							s12					∞
s12		r1	r1						r1			∞

[0342] 图15A-15B示出了“宠物”的状态机图。该图示出了规则 and 状态。在图15A中,存在状态s0 1510、s1 1515、s2 1520、s3 1525和s4 1530。存在规则r0 1533、r0 1535和r4 1536。箭头1540和1542连接到图15B中的对应箭头。在图15B中,存在状态s2 1550、s3 1555、s7 1560、s8 1565、s8 1568、s10 1570、s11 1575和s12 1580。存在规则r2 1583、r2 1585和r31588。

[0343] 表D示出了匹配输入“catcanarydog”的驱动器的轨迹,其中每个事件在时间上被一个单位分开。然后输入以直到时间单元50的流逝结束。

[0344] 有几点需要注意:(1) 驱动程序缓冲输入令牌(在栈上)的长度仅足以识别它们匹配的模式的部分。一旦匹配,输入就会缩减。(2) 每个输入仅由驱动程序检查一次。(3) 驱动程序对输入的缺失及其存在作出反应。最后的缩减只是在流逝时触发。(3) 驱动程序最终在启动状态结束,准备继续匹配。

[0345] 表D: “宠物”的样本跟踪

[0346]

步骤	栈	输入	动作
----	---	----	----

[0347]

1	[s0-1050]	1000:«c» 1001:«a» 1002:«t» 1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	由 r4 缩减, 去到 s1
2	[s0-1050] [s1-1050 pets]	1000:«c» 1001:«a» 1002:«t» 1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s3
3	[s0-1050] [s1-1050 pets] [s3-1050 «c»]	1001:«a» 1002:«t» 1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s5
4	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»]	1002:«t» 1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s8
5	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»] [s8-1050 «t»]	1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	由 r2 缩减, 去到 s2
6	[s0-1050] [s1-1050 pets] [s2-1050 pet]	1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	由 r5 缩减, 去到 s1
7	[s0-1050] [s1-1050 pets]	1003:«c» 1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s3
8	[s0-1050] [s1-1050 pets] [s3-1050 «c»]	1004:«a» 1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s5

[0348]

9	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»]	1005:«n» 1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s7
10	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»] [s7-1050 «n»]	1006:«a» 1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s10
11	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»] [s7-1050 «n»] [s10-1050 «a»]	1007:«r» 1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s11
12	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»] [s7-1050 «n»] [s10-1050 «a»] [s11-1050 «r»]	1008:«y» 1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s12
13	[s0-1050] [s1-1050 pets] [s3-1050 «c»] [s5-1050 «a»] [s7-1050 «n»] [s10-1050 «a»] [s11-1050 «r»] [s12-1050 «y»]	1009:«d» 1010:«o» 1011:«g» 1050:τ	由 r1 缩减, 去到 s2
14	[s0-1050] [s1-1050 pets] [s2-1050 pet]	1009:«d» 1010:«o» 1011:«g» 1050:τ	由 r5 缩减, 去到 s1
15	[s0-1050] [s1-1050 pets]	1009:«d» 1010:«o» 1011:«g» 1050:τ	移位, 去到 s4
16	[s0-1050] [s1-1050 pets] [s4-1050 «d»]	1010:«o» 1011:«g» 1050:τ	移位, 去到 s6
17	[s0-1050] [s1-1050 pets] [s4-1050 «d»] [s6-1050 «o»]	1011:«g» 1050:τ	移位, 去到 s9
18	[s0-1050] [s1-1050 pets] [s4-1050 «d»] [s6-1050 «o»] [s9-1050 «g»]	1050:τ	由 r3 缩减, 去到 s2
19	[s0-1050] [s1-1050 pets] [s2-1050 pet]	1050:τ	由 r5 缩减, 去到 s1
20	[s0-1050] [s1-1050 pets]	1050:τ	报告并重启
21	[s0-1100]	1050:τ	丢弃

[0349] 流处理器可以使用表达式评估来查找匹配。例如,在流式SQL的SELECT语句中,可以找到where子句。此子句指定要评估的表达式,当结果为真(true)时,选择匹配。例如,如果当前输入由标识符字表示,则查找宠物的where子句可能是word=“cat”或word=“dog”或word=“canary”。

[0350] 必须缓冲单词的内容直到选择达到匹配。每次新输入到达时,必须用单个字符扩展单词,然后必须重新评估整个表达式。

[0351] 从先前的迭代中学到的东西都没有进入下一迭代。例如,如果单词当前包含“do”,则表达式仍然需要将其与“cat”和“canary”进行比较,这不可能匹配。在做出决定之前,可能多次检查单词的每个字符。

[0352] 这远远低于TFR (1) 的效率,TFR (1) 从不检查输入多于一次。每个解析状态都隐含地携带有关其前面的状态的信息。例如,狗与非狗的决定是在s1做出的。像s3和s4这样的后续状态已经知道“狗”是否仍然是可能的匹配。

[0353] 构建TFR (1) 状态表。本节描述用于根据一组语法规则构造状态表的算法。总共有三个步骤:

[0354] 1. 计算所有符号的第一组和后续组。

[0355] 2. 从起始项生成起始状态,然后生成从起始状态直接或间接可到达的所有状态。

[0356] 3. 为每个生成的状态生成表条目。

[0357] 然而,在我们解释这些步骤之前,我们首先需要定义一些涉及的概念。

[0358] 概念。第一和后续。第一组符号是可以开始从该符号导出的事件序列的一组令牌。如果符号是具有空规则的缩减,则符号的第一组也包含特殊标记 ϵ 。

[0359] 为了计算First (X) :

[0360] 1. 如果X是令牌,那么First (X) 只是{X}。

[0361] 2. 如果X是具有空规则的缩减,则将 ϵ 添加到First (X) 。

[0362] 3. 对于具有X作为其左侧的每个非空规则,将每个右侧的第一个符号的第一集合添加到First (X) 。如果这个右侧符号本身是具有空规则的缩减,那么将右侧的下一个符号的第一集合添加到First (X) 。以这种方式继续,直到遇到没有空规则的右侧符号,或者直到没有更多的右侧符号。

[0363] 还为由两个或更多个符号组成的序列 α 定义第一集合。为了计算First (α) :

[0364] 1. 从 α 中第一个符号的第一集合开始。如果不包含 ϵ ,则将此集合返回作为First (α) 。

[0365] 2. 如果它确实包含 ϵ ,则从集合中移除 ϵ 并添加 α 中下一个符号的第一集合。以这种方式继续,直到遇到具有第一集合的符号不包含 ϵ 或直达到 α 的结尾。

[0366] 符号的接下来的集合是可以在可识别的序列中跟随符号的事件集。为了计算Follow (X) :

[0367] 1. 如果X是 Ω ,则其Follow (X) 包含中断事件以及First (Ω) 中的所有事件。

[0368] 2. 检查X出现在右侧的每个规则。设A是该规则的左侧,并且 β 是在该规则中出现在X右侧的符号序列。如果 β 是非空的,那么Follow (X) 包括First (β) ;否则,Follow (X) 包括First (A) 。

[0369] 项目和状态。语法项目表示匹配期间达到的特定点。项目包括对语法规则之一的引用、规则中的点位置和前瞻事件。例如, $[\text{pet} \rightarrow \langle\langle c \rangle\rangle \cdot \langle\langle a \rangle\rangle \langle\langle t \rangle\rangle, \langle\langle c \rangle\rangle]$ 是宠物语法的一个项目。

[0370] 项目中的点位置表示下一次匹配将发生的确切位置。点可以位于规则右侧的任何符号之前,也可以位于最后一个符号之后。在后一种情况下,该项目被认为是完成的。如果

项目未完成,则点后面的符号称为项目的角落。完成项目没有角落符号。点之前的符号序列称为 α 序列,而角落之后的符号序列称为 β 序列。 α 和 β 序列中的任何一个或两个都可以是空的。

[0371] 引用空规则的项目仅具有单个可能的点位置。这样的项目被认为是完成的。它没有角落符号,其 α 和 β 序列为空。

[0372] 项目的前瞻事件是令牌或中断。在完成的项目中,前瞻事件有助于解析器生成器决定是应该按原样缩减项目还是移位另一事件。

[0373] 语法的起始项目是可以开始匹配的项目之一。每个起始项指的是 Ω 规则,并且其点在最左侧。要生成语法的起始项目集合,计算 $\text{Follow}(\Omega)$ 。对于 $\text{Follow}(\Omega)$ 中的每个符号 F ,添加具有前瞻 F 的起始项目。

[0374] 状态是一组项目。例如, $\{[\text{pets} \rightarrow \text{pets pet} \cdot, \langle \langle c \rangle \rangle], [\text{pets} \rightarrow \text{pets pet} \cdot, \langle \langle d \rangle \rangle], [\text{pets} \rightarrow \text{pets pet} \cdot, -]\}$ 是宠物语法中的状态 s_2 。

[0375] 每个项目都有超时。如果项目已完成,则其超时时间与其规则的超时时间相同。非空规则或 λ 规则有无限超时。如果项目未完成,那么它的超时是无限的。

[0376] 每个状态也有超时,其等于其项目中的最小超时。

[0377] 闭包(Closure)。状态是项目集合,但并非每个项目集合都是状态。要成为状态,该集合必须是完整的闭包。闭包函数通过添加完成它所需的项目,从初始项目集生成完成状态。

[0378] 如果 J 是项目集合,那么计算 $\text{Closure}(J)$ 如下:

[0379] 1. 从项目列表(todo)和项目集合(done)开始,要做的项目列表最初包含 J 中的所有项目,已完成项目集合最初为空。

[0380] 2. 如果todo为空,则返回done作为语法状态。

[0381] 3. 从todo中弹出一个项目。如果项目已经在done中或项目已完成(即,如果项目的点在最右侧),则丢弃项目并返回步骤2。

[0382] 4. 设 B 为项目的角落符号,设 β 为项目的 β 序列,设 X 为项目的前瞻,设 next 为 $\text{First}(\beta X)$ 事件集。

[0383] 5. 将中断事件添加到 next 。

[0384] 6. 对于 B 显示为左侧的每个规则,以及对于 next 中的每个事件 Y ,将项目 $[\text{rule}, 0, Y]$ 添加到todo。

[0385] 7. 返回步骤2。

[0386] Goto。状态包含对于某些序列 γ 可行的项目集,并且每个序列具有以这种方式表示它的状态。因此,与 γ 相同但是一个更长的事件的序列 γX 也必须由状态表示。如果 γ 的状态是 J ,那么 γX 的状态被称为通过 X 的 J 的goto。

[0387] 为了计算 $\text{Goto}(J, X)$:

[0388] 1. 从项目列表todo和项目集合 next 开始,项目列表todo最初包含 J 中的所有项目,项目集合 next 最初为空。

[0389] 2. 如果todo为空,则返回 $\text{Closure}(\text{next})$ 作为goto状态。

[0390] 3. 从todo中弹出一个项目。如果项目已完成或项目的角落符号不是 X ,则丢弃项目并返回步骤2。

[0391] 4. 设规则为项目的规则, 点为项目的点位置, Y 为项目的前瞻。将项目 [rule, dot+1, Y] 添加到 next。

[0392] 5. 返回步骤 2。

[0393] 构造算法。我们现在可以解释本章开头提到的构造算法的三个步骤。第一步, 计算第一集合和后续集合, 只需迭代所有符号并应用上一节中的定义。我们现在解释第二步和第三步。

[0394] 状态生成。要生成语法的所有状态:

[0395] 1. 从状态列表 todo 和状态集合 done 开始, 状态列表 todo 最初包含起始项目的闭包, 状态集合 done 最初为空。

[0396] 2. 如果 todo 为空, 则返回 done 作为语法状态集。

[0397] 3. 从 todo 中弹出一个状态 J。如果 J 已经在 done 中, 请将其丢弃并返回步骤 2。

[0398] 4. 对于语法 (包括 Ω 和中断) 中的每个符号 X, 设 K 为 Goto (J, X)。如果 K 为非空, 则将 K 添加到 todo。

[0399] 5. 返回步骤 2。

[0400] 表 E 示出了将该算法应用于表 B 中的语法的结果。

[0401] 表 E: “宠物” 的状态项目

[0402]

状态	项目
s0	$[\Omega \rightarrow \cdot \text{pets}, \langle c \rangle]$ $[\Omega \rightarrow \cdot \text{pets}, \langle d \rangle]$ $[\Omega \rightarrow \cdot \text{pets}, \text{—}]$ $[\text{pets} \rightarrow \tau 50, \langle c \rangle]$ $[\text{pets} \rightarrow \tau 50, \langle d \rangle]$

[0403]

	[pets $\rightarrow$ $\tau 50$, —] [pets $\rightarrow$ · pets pet, «c»] [pets $\rightarrow$ · pets pet, «d»] [pets $\rightarrow$ · pets pet, —]
s1	[$\Omega \rightarrow$ pets ·, «c»] [$\Omega \rightarrow$ pets ·, «d»] [$\Omega \rightarrow$ pets ·, —] [pet $\rightarrow$ · «c» «a» «n» «a» «r» «y», «c»] [pet $\rightarrow$ · «c» «a» «n» «a» «r» «y», «d»] [pet $\rightarrow$ · «c» «a» «n» «a» «r» «y», —] [pet $\rightarrow$ · «c» «a» «t», «c»] [pet $\rightarrow$ · «c» «a» «t», «d»] [pet $\rightarrow$ · «c» «a» «t», —] [pet $\rightarrow$ · «d» «o» «g», «c»] [pet $\rightarrow$ · «d» «o» «g», «d»] [pet $\rightarrow$ · «d» «o» «g», —] [pets $\rightarrow$ pets · pet, «c»] [pets $\rightarrow$ pets · pet, «d»] [pets $\rightarrow$ pets · pet, —]
s2	[pets $\rightarrow$ pets pet ·, «c»] [pets $\rightarrow$ pets pet ·, «d»] [pets $\rightarrow$ pets pet ·, —]
s3	[pet $\rightarrow$ «c» · «a» «n» «a» «r» «y», «c»] [pet $\rightarrow$ «c» · «a» «n» «a» «r» «y», «d»] [pet $\rightarrow$ «c» · «a» «n» «a» «r» «y», —] [pet $\rightarrow$ «c» · «a» «t», «c»] [pet $\rightarrow$ «c» · «a» «t», «d»] [pet $\rightarrow$ «c» · «a» «t», —]
s4	[pet $\rightarrow$ «d» · «o» «g», «c»] [pet $\rightarrow$ «d» · «o» «g», «d»]

[0404]

	[pet → «d» · «o» «g», —]
s5	[pet → «c» «a» · «n» «a» «r» «y», «c»] [pet → «c» «a» · «n» «a» «r» «y», «d»] [pet → «c» «a» · «n» «a» «r» «y», —] [pet → «c» «a» · «t», «c»] [pet → «c» «a» · «t», «d»] [pet → «c» «a» · «t», —]
s6	[pet → «d» «o» · «g», «c»] [pet → «d» «o» · «g», «d»] [pet → «d» «o» · «g», —]
s7	[pet → «c» «a» «n» · «a» «r» «y», «c»] [pet → «c» «a» «n» · «a» «r» «y», «d»] [pet → «c» «a» «n» · «a» «r» «y», —]
s8	[pet → «c» «a» «t» ·, «c»] [pet → «c» «a» «t» ·, «d»] [pet → «c» «a» «t» ·, —]
s9	[pet → «d» «o» «g» ·, «c»] [pet → «d» «o» «g» ·, «d»] [pet → «d» «o» «g» ·, —]
s10	[pet → «c» «a» «n» «a» · «r» «y», «c»] [pet → «c» «a» «n» «a» · «r» «y», «d»] [pet → «c» «a» «n» «a» · «r» «y», —]
s11	[pet → «c» «a» «n» «a» «r» · «y», «c»] [pet → «c» «a» «n» «a» «r» · «y», «d»] [pet → «c» «a» «n» «a» «r» · «y», —]
s12	[pet → «c» «a» «n» «a» «r» «y» ·, «c»] [pet → «c» «a» «n» «a» «r» «y» ·, «d»] [pet → «c» «a» «n» «a» «r» «y» ·, —]

[0405] 表生成。生成状态后,每个状态都将转换为解析器表中的行。每一行由两部分组成:动作和转换。

[0406] 为了计算状态J的动作:

[0407] 对于J中的每个项目,设规则为项目的规则,设B为项目的角落符号(如果有的话),并设X为项目的前瞻:

[0408] a.如果项目完成且其左侧为 Ω ,则重启action(J,X)。

[0409] b.如果项目完成且其左侧不是 Ω ,则action(J,X)是reduce(rule)。

[0410] c.如果项目未完成且B是令牌且Goto(J,B)不为空,则action(J,X)为shift(Goto(J,B))。

[0411] d.在所有其他情况下,action(J,X)是恐慌。

[0412] 为了计算状态J的转换:

[0413] 对于语法中每个缩减A,设K为Goto(J,A):

[0414] a.如果K是非空的,则transition(J,A)是K。

[0415] b.如果K为空,则transition(J,A)未定义。

[0416] 将这些算法应用于表E产生表B。

[0417] 其他优化。如前所述,用于构造TFR(1)解析器表的该算法类似于用于构造规范LR(1)解析器表的算法的扩展。应用于LR(1)解析器的各种概括、优化和状态减少策略也可以在必要的变更后应用于TFR(1)解析器表。这些策略包括但不限于LR(1)至LALR(1)的优化和LR(1)至GLR(1)的概括。

[0418] 一种用于描述和转换用于数据流计算环境的反应函数的方法,包括:(1)识别反应函数。(2)识别为函数提供输入的参数模式。(3)根据传递给函数的参数识别要评估的表达式。(4)将参数模式转换为能够识别与模式匹配的输入序列的状态机。(5)通过调用将输入数据转换为输出数据的函数来扩充状态机。(6)将状态机实现为能够通过简单的软件或硬件自动化的确定性下推自动机。

[0419] 在一种实现中,一种方法包括:从监视物理量的硬件传感器接收数据流,并将所监视的物理量转换为数字形式的数据流;将数据流存储在输入队列中,其中数据流中的每个令牌与接收该令牌时的时间戳一起存储;通过在单个方向上通过输入队列读取令牌,并且不重复读取任何先前读取的令牌,识别数据流中与一个或多个预定模式匹配的模式;并且在识别数据流中的预定模式时,在输出队列中输出已经识别出预定模式的肯定指示。

[0420] 该方法可以进一步包括,在数据流中未识别出预定模式时,在输出队列中输出已经识别出预定模式的否定指示,其中预定模式存储在状态表和状态栈中。

[0421] 在另一实现中,一种系统包括:输入队列,包括计算机存储器中的第一存储器位置,其中输入队列在第一存储器位置中存储要处理的令牌的先进先出序列和与队列中的每个令牌相关联的、指示关联令牌进入输入队列的时间的时间戳,并且令牌由输入队列通过网络接收。驱动程序组件,连接到输入队列,其中驱动程序组件在没有回溯的情况下处理输入队列中的令牌,并识别与一个或多个预定输入模式匹配的令牌序列中的模式,并且在识别匹配的预定输入模式时,生成传出事件输出;输出队列,连接到驱动器组件,包括计算机存储器中的第二存储器位置,其中输出队列在第二存储器位置中存储由驱动器组件生成的输出事件的先进先出序列。状态表组件,连接到驱动器组件,其中状态表组件以状态表格式存储预定输入模式。状态栈组件,连接到驱动器组件,包括计算机存储器中的第三存储器位置,其中状态栈组件在第三存储器位置中存储帧的后进先出顺序存储。帧包括翻译状态号、

符号和截止日期。

[0422] 在输入队列处接收的令牌可以由将物理属性转换为数字量的硬件传感器生成。驱动程序组件的操作可以包括：检查在状态栈的顶部帧上引用的状态，检查输入队列中的下一输入令牌并基于状态栈确定其类别，以及针对由状态表指示的类别的令牌执行由状态表指示的动作。

[0423] 输入队列基于输入队列接收的时间将令牌存储在有序列表中。队列中的第一方向是从最早接收的令牌到最新接收的令牌的时间顺序。驱动程序组件通过在第一方向上检查输入队列来处理令牌，并且不检查任何先前检查的一个或多个令牌。

[0424] 输入队列按照输入队列接收的时间顺序将令牌存储在列表中。队列中的第一方向是从最早接收的令牌到最新接收的令牌的时间顺序。队列中的第二方向是从最新接收的令牌到最早接收的令牌的时间顺序。第二方向与第一方向相反。驱动程序组件通过仅在第一方向或第二方向上检查输入队列而不是第一和第二方向二者来处理令牌。

[0425] 驱动程序仅读取输入队列的每个令牌一次，并且不在缓冲区中保留读取令牌以便稍后再次读取。要匹配的预定输入模式可以具有多个不定长度的令牌。状态表组件指定状态机并包括多个行，每行提供状态机的转换状态。

[0426] 每行包括：动作集合，其将输入令牌类别值或输入中断类别值映射到在接收到该值的输入令牌时要采取的动作；转换集合，它将衍生符号类型映射到驱动程序在合成该类型的衍生符号时应更改的状态；和超时，表示允许状态机保持特定状态的时间间隔。

[0427] 转换状态号是对状态表中的行的引用。符号是从输入令牌或其他符号或某种组合导出的令牌或中间符号。截止日期是驱动程序将帧从栈中弹出的未来时刻。

[0428] 在一种实现中，驱动器组件和状态表组件体现在可编程门阵列中。在另一实现中，使用计算机来体现驱动程序组件和状态表组件。状态表组件指定状态机，并且驱动器组件基于在输入队列中没有接收到具有流数据的令牌的一段时间而从状态机的第一状态改变到状态机的第二状态。

[0429] 状态表组件指定状态机。基于在第一时间段期间在输入队列中接收到具有流数据的令牌序列，驱动器组件从状态机的第一状态变为状态机的第二状态。驱动程序组件基于在第二时间段期间在输入队列中没有接收到具有流数据的令牌序列，从状态机的第三状态变为状态机的第四状态。

[0430] 在另一实现中，一种方法包括：提供包括计算机存储器中的第一存储器位置的输入队列，其中输入队列在第一存储器位置中存储要处理的令牌的先进先出序列和与队列中的每个令牌相关联的时间戳，其指示关联令牌进入输入队列的时间，并且令牌由输入队列通过网络接收。提供驱动程序组件，连接到输入队列，其中驱动程序组件处理输入队列中的令牌而不回溯，并识别与一个或多个预定输入模式匹配的令牌序列中的模式，并且在识别匹配的预定输入模式时，生成传出事件输出。提供输出队列，连接到驱动器组件，包括计算机存储器中的第二存储器位置，其中输出队列在第二存储器位置中存储由驱动器组件生成的输出事件的先进先出序列。提供状态表组件，连接到驱动器组件，其中状态表组件以状态表格式存储预定输入模式。提供状态栈组件，连接到驱动器组件，包括计算机存储器中的第三存储器位置，其中状态栈组件在第三存储器位置中存储帧的后进先出顺序存储，并且帧包括翻译状态号、符号和截止日期。

[0431] 生成状态表组件的状态表包括:识别一组终端符号,其中每个符号表示输入数据的类别;识别一组非终端符号,其中每个非终端符号表示终端或非终端符号中的至少一个的模式;识别一组语法规则;确定顶级规则;识别一组属性;为每个识别的终端和非终端符号计算第一集合;为每个识别的终端和非终端符号计算跟随集合;结合相应跟随集合的每个成员,从顶级规则的闭包产生起始状态。

[0432] 生成状态表可以包括:通过递归地生成从开始状态起存在状态改变转换的每个状态的闭包,生成从开始状态可到达的所有状态;通过组合和缩减来优化状态集;并为每个状态生成动作、转换和超时值。

[0433] 该组语法规则可以包括:每个非空规则采用 $A \rightarrow B_0 \cdots B_n$ 的形式,其中A是非终端符号,B₀到B_n分别是终端或非终端符号;每个空规则采用 $A \rightarrow \tau$ 形式,其中 τ 是有限的时间量,在某个有用的时间单位中表征,或者是无限的,表示无限的时间量。

[0434] 该组属性可以包括:每个属性采用形式 $\alpha: T = \{A_0 \rightarrow \lambda_0, \dots, A_n \rightarrow \lambda_n\}$,其中 α 是属性的名称,T是属性的类型,A是非终端符号, λ 是用于计算该符号的属性值的函数。顶级规则可以具有 $\Omega \rightarrow B$ 形式,其中B是终端或非终端符号。

[0435] 已经出于说明和描述的目的呈现了本发明的描述。其并非旨在穷举或将本发明限制于所描述的精确形式,并且鉴于上述教导,许多修改和变化是可能的。选择和描述实施例是为了最好地解释本发明的原理及其实际应用。该描述将使得本领域的其他技术人员能够在各种实施例中以及适合于特定用途的各种修改中最好地利用和实践本发明。本发明的范围由以下权利要求限定。

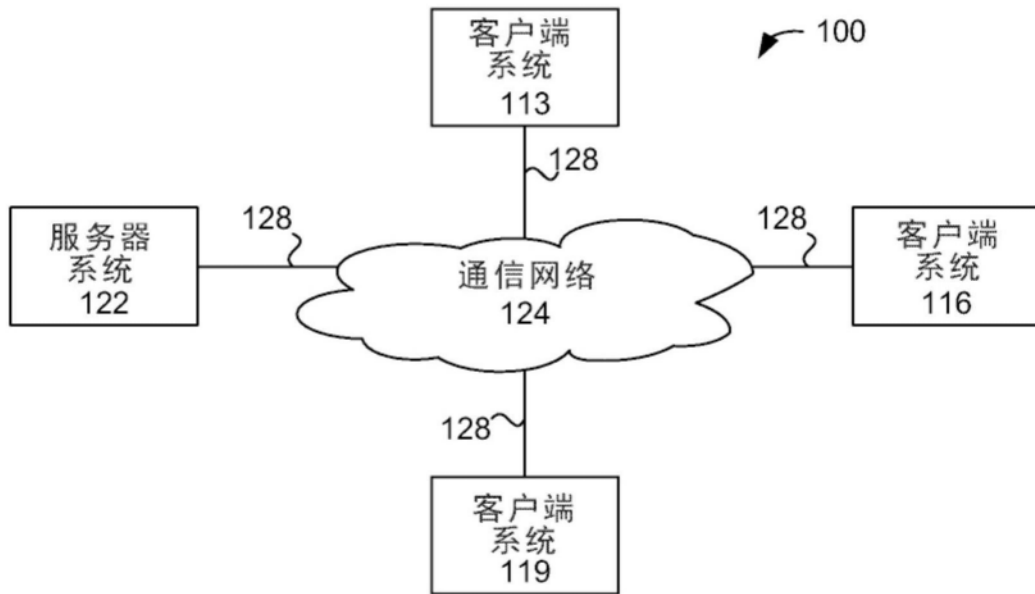


图1

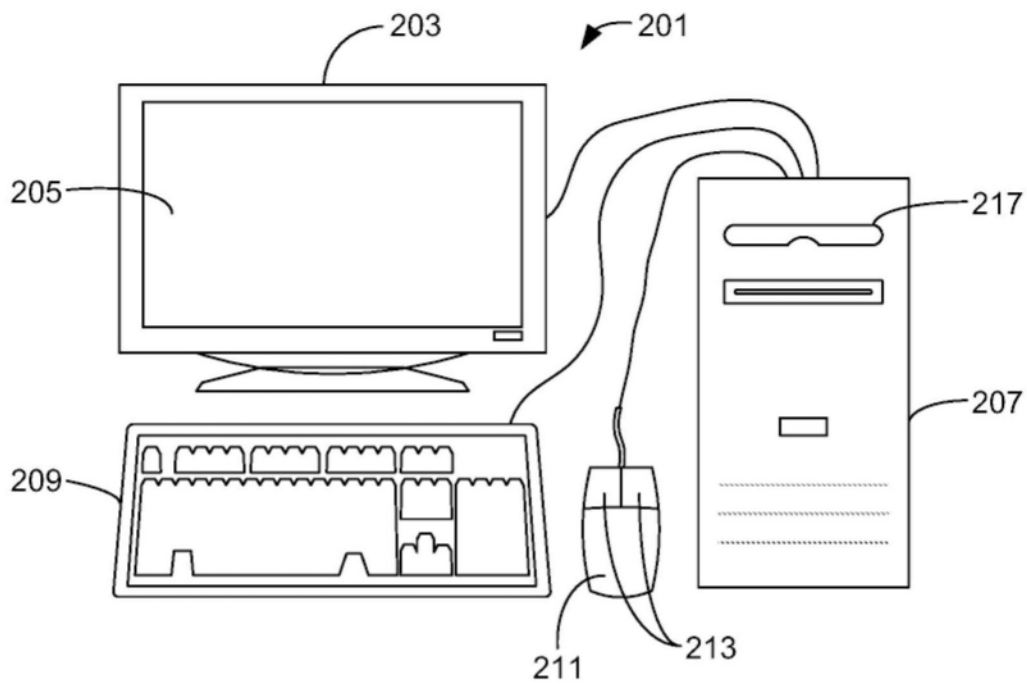


图2

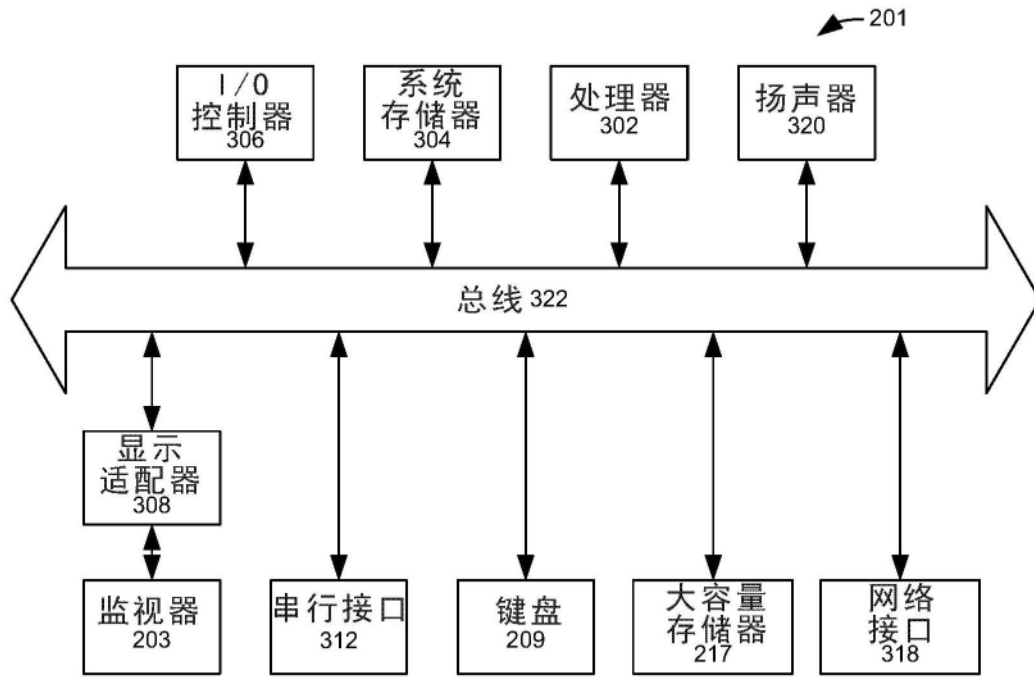


图3

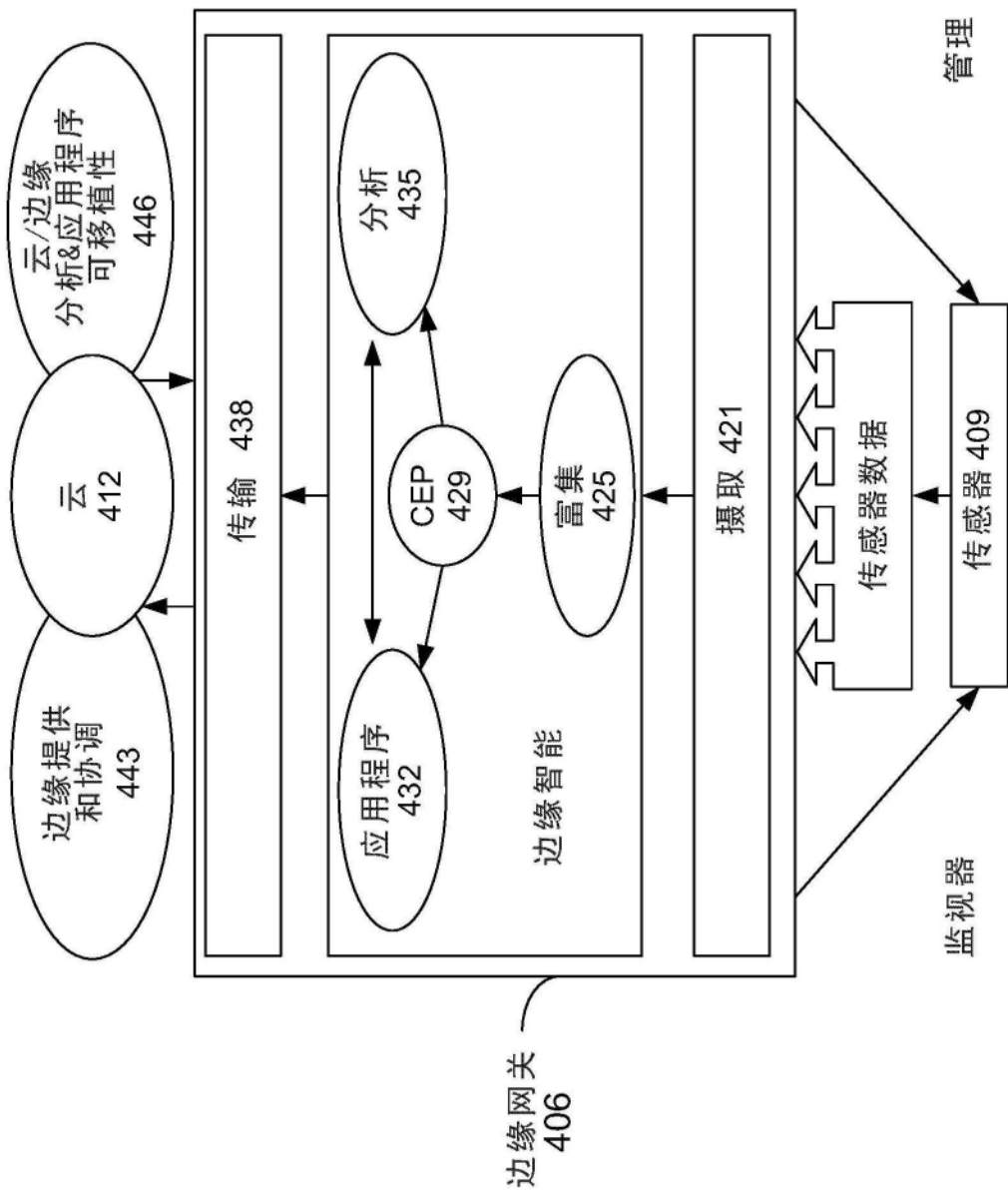


图4

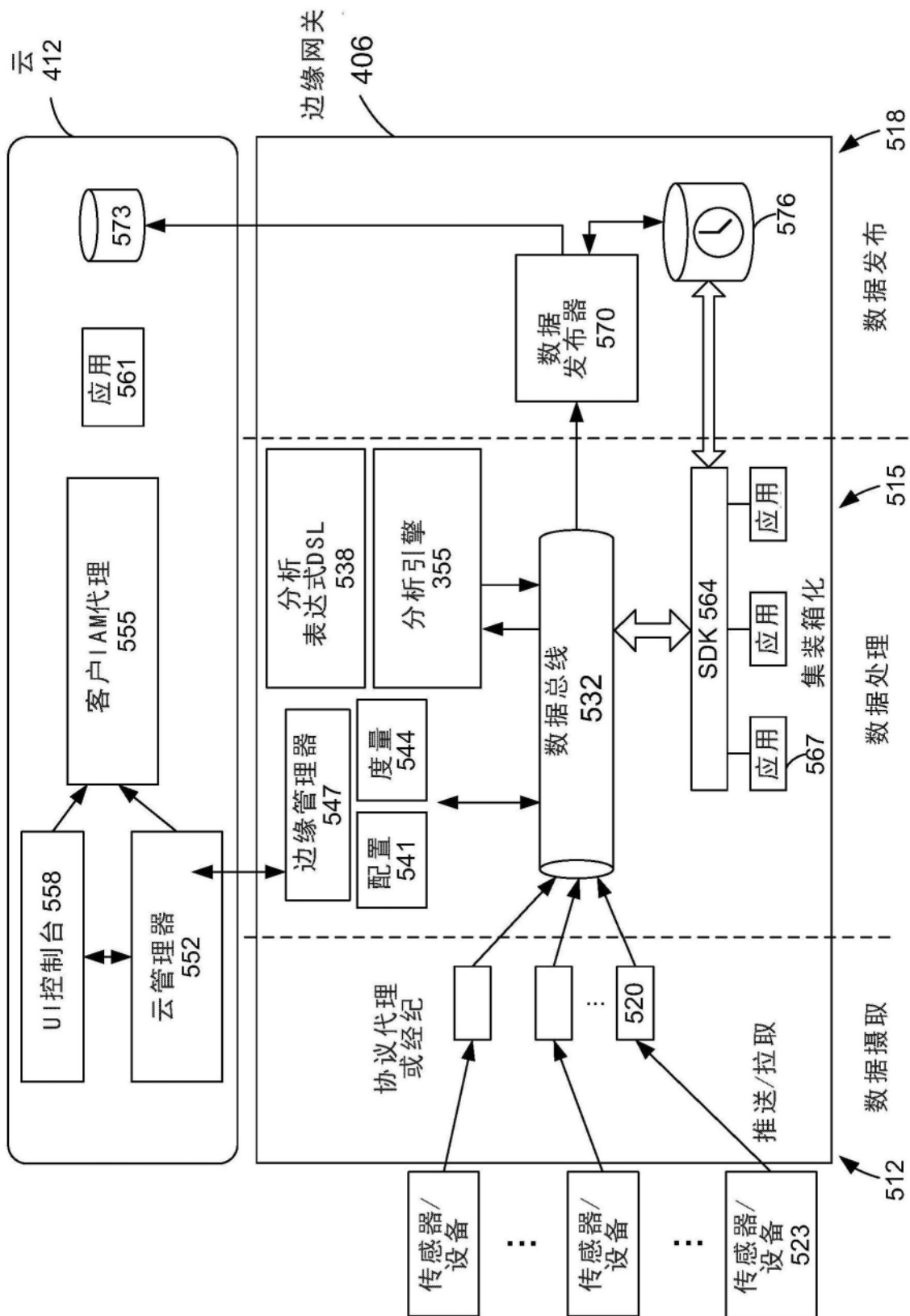


图5

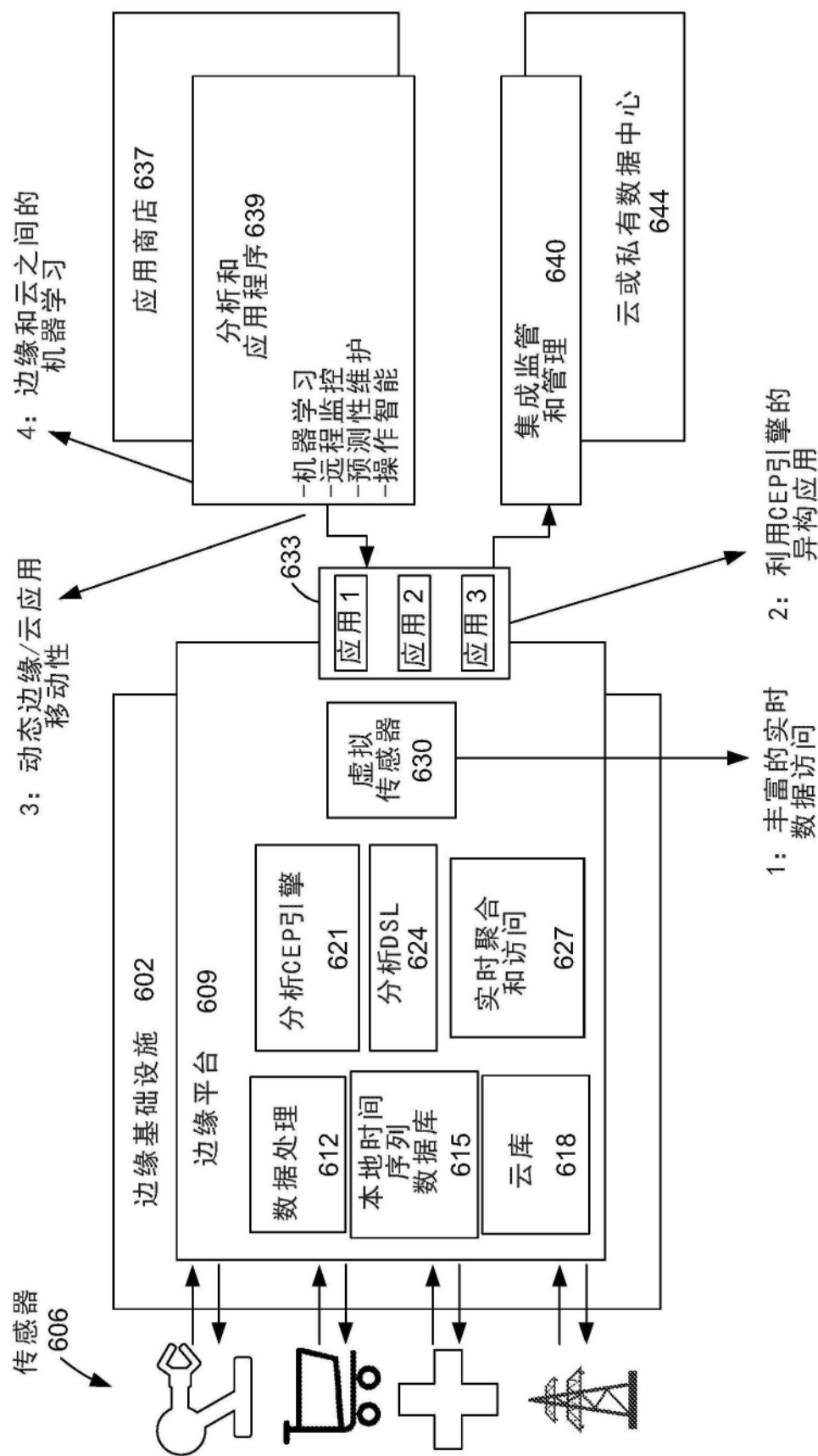


图6

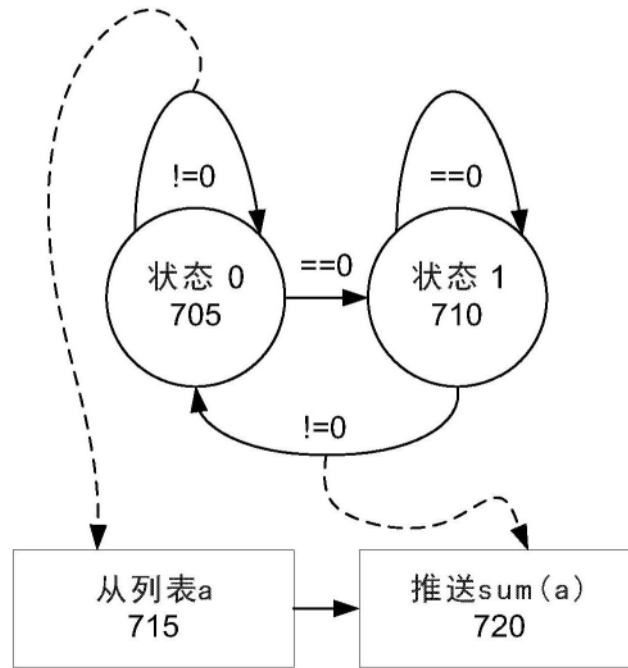


图7

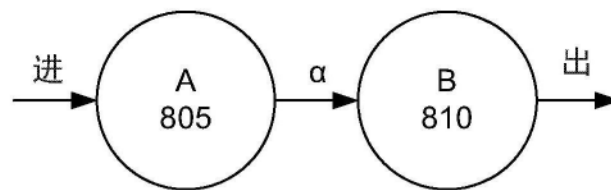


图8

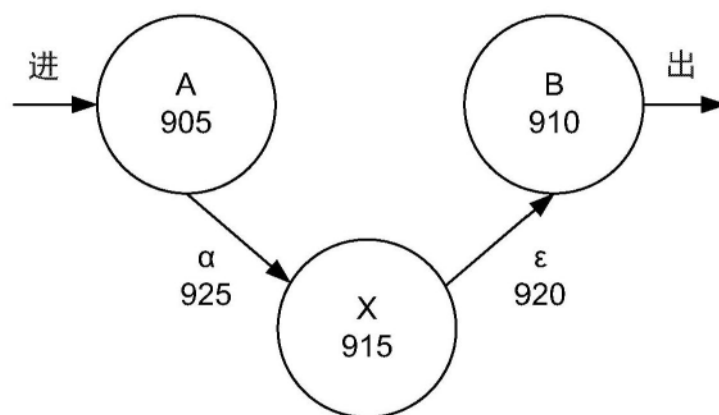


图9

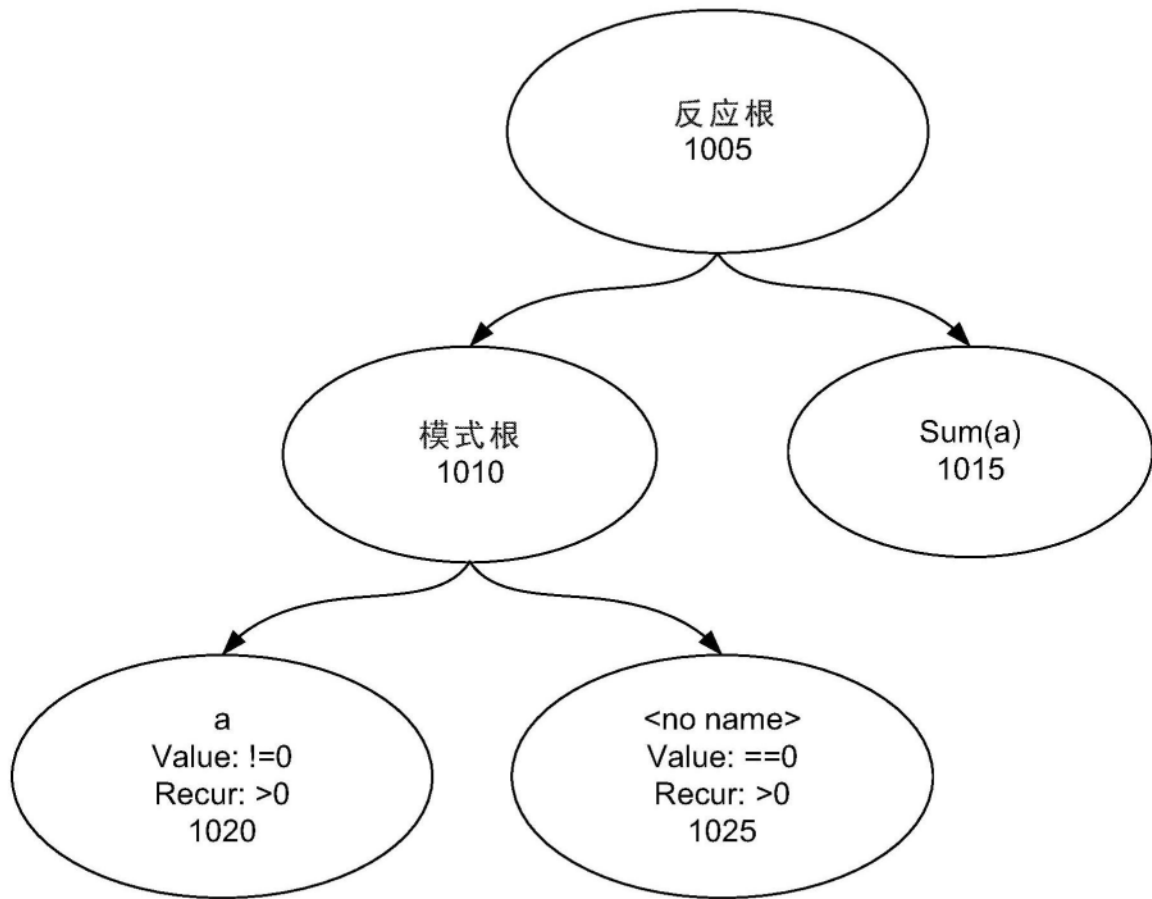


图10

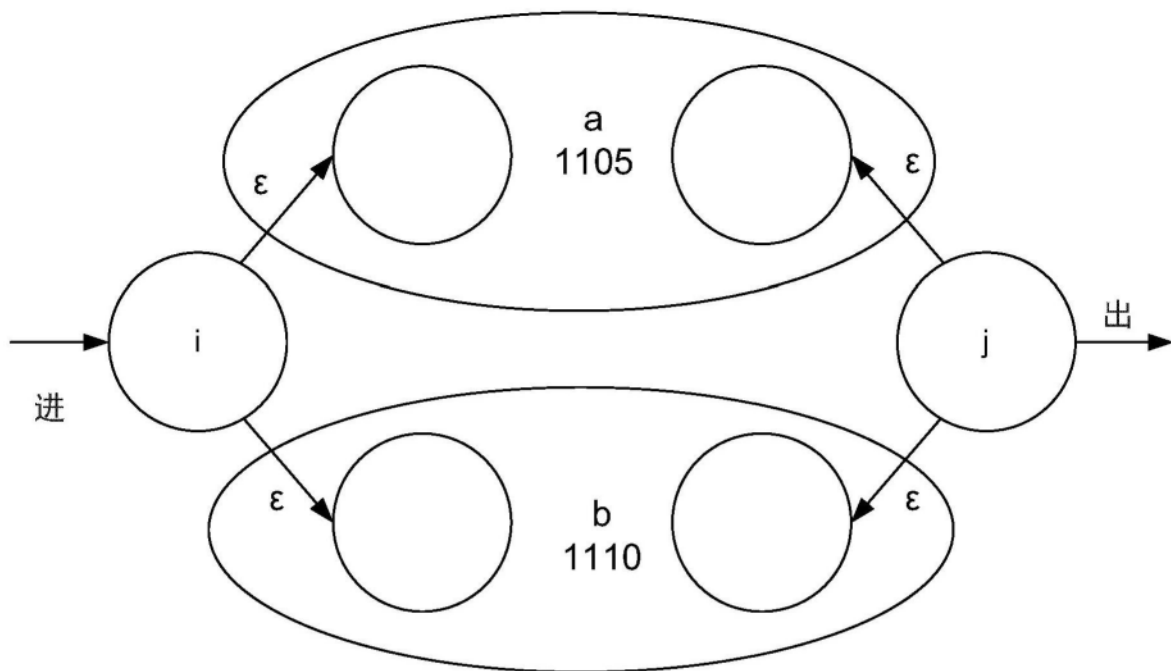


图11

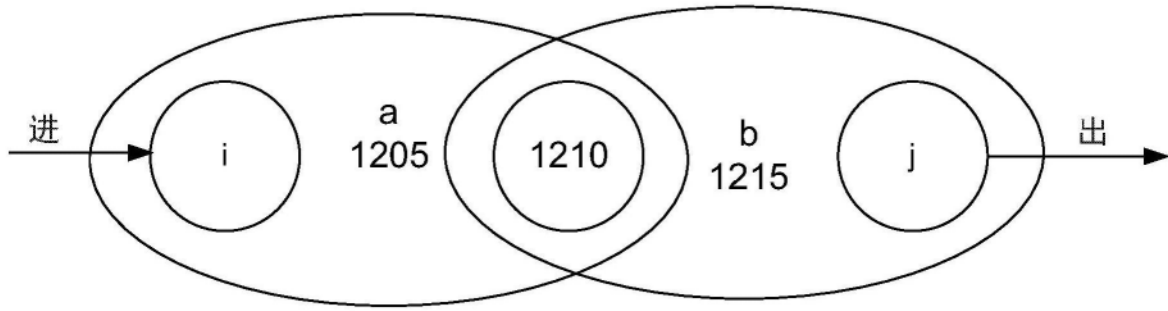


图12

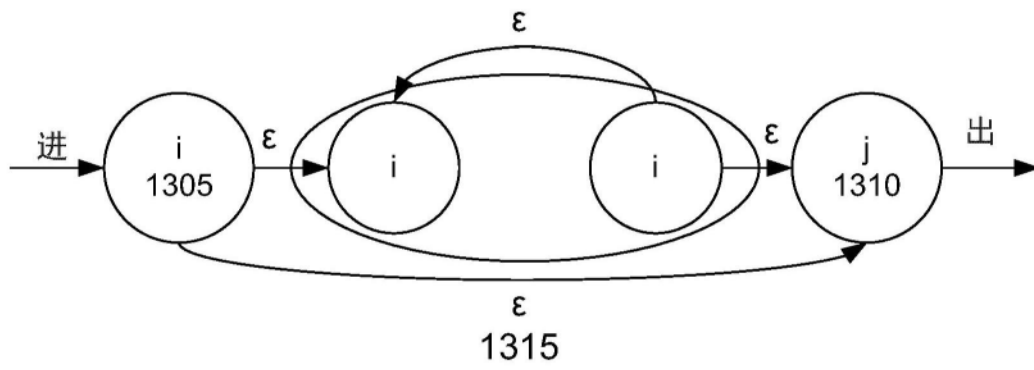


图13

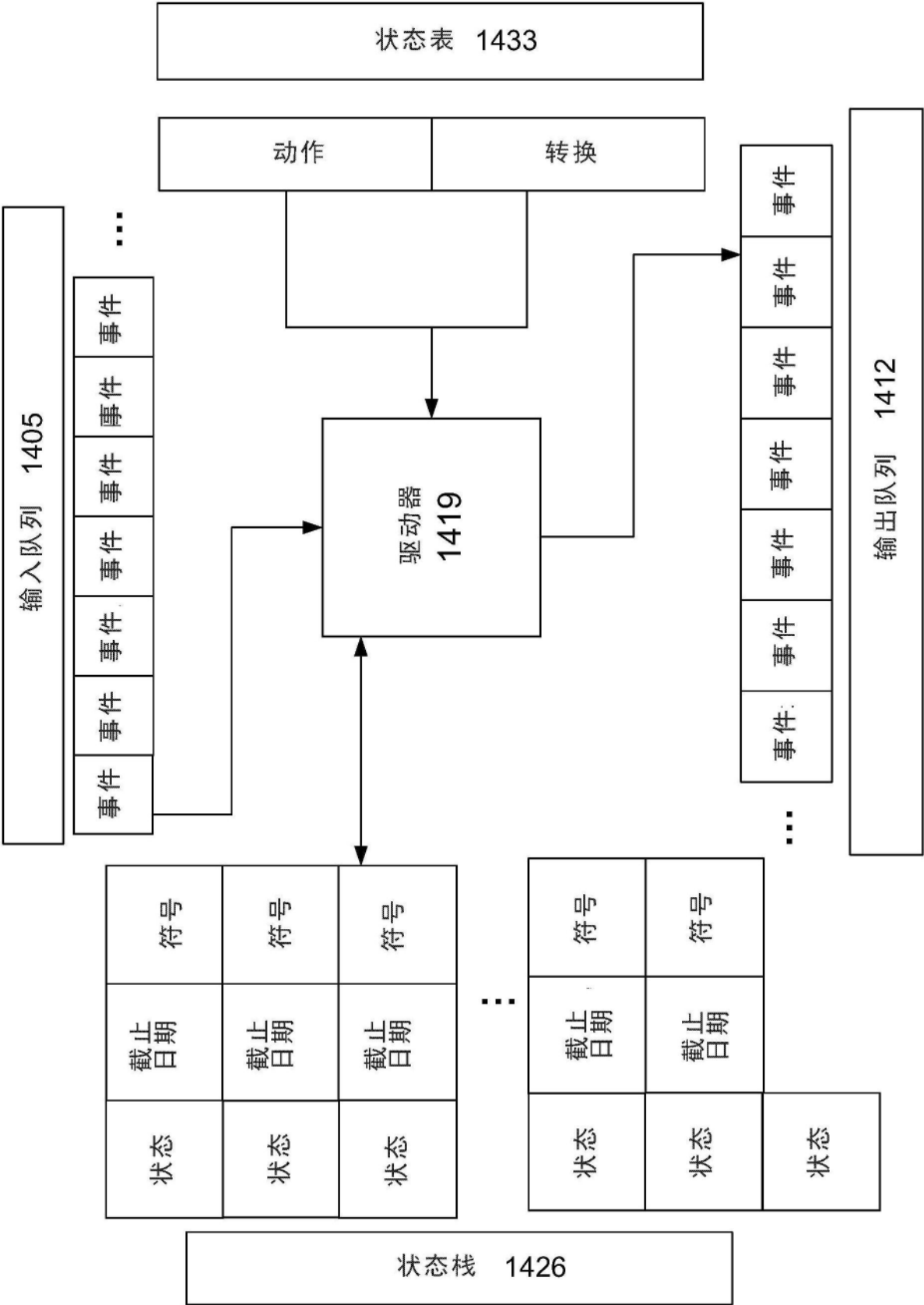


图14

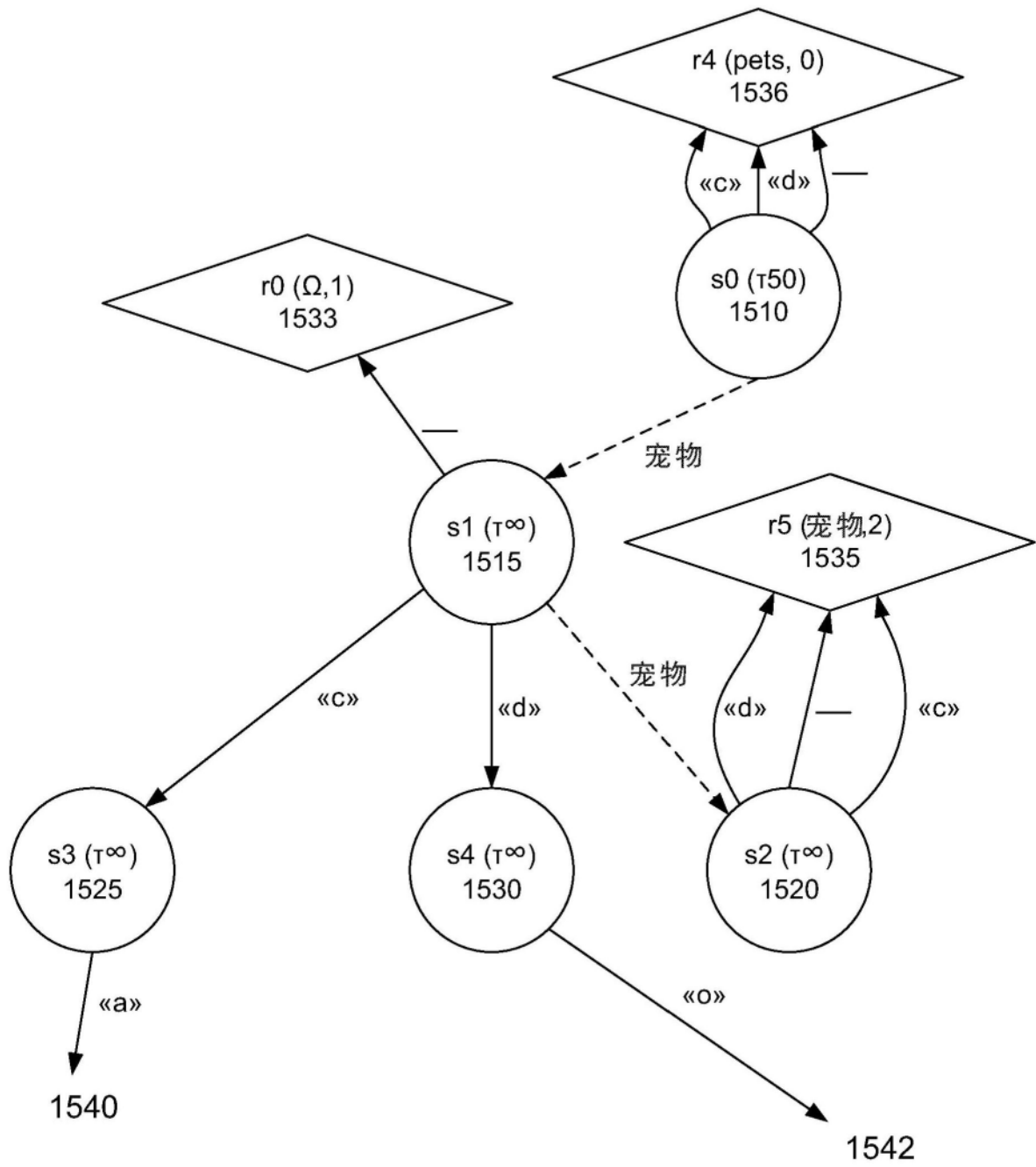


图15A

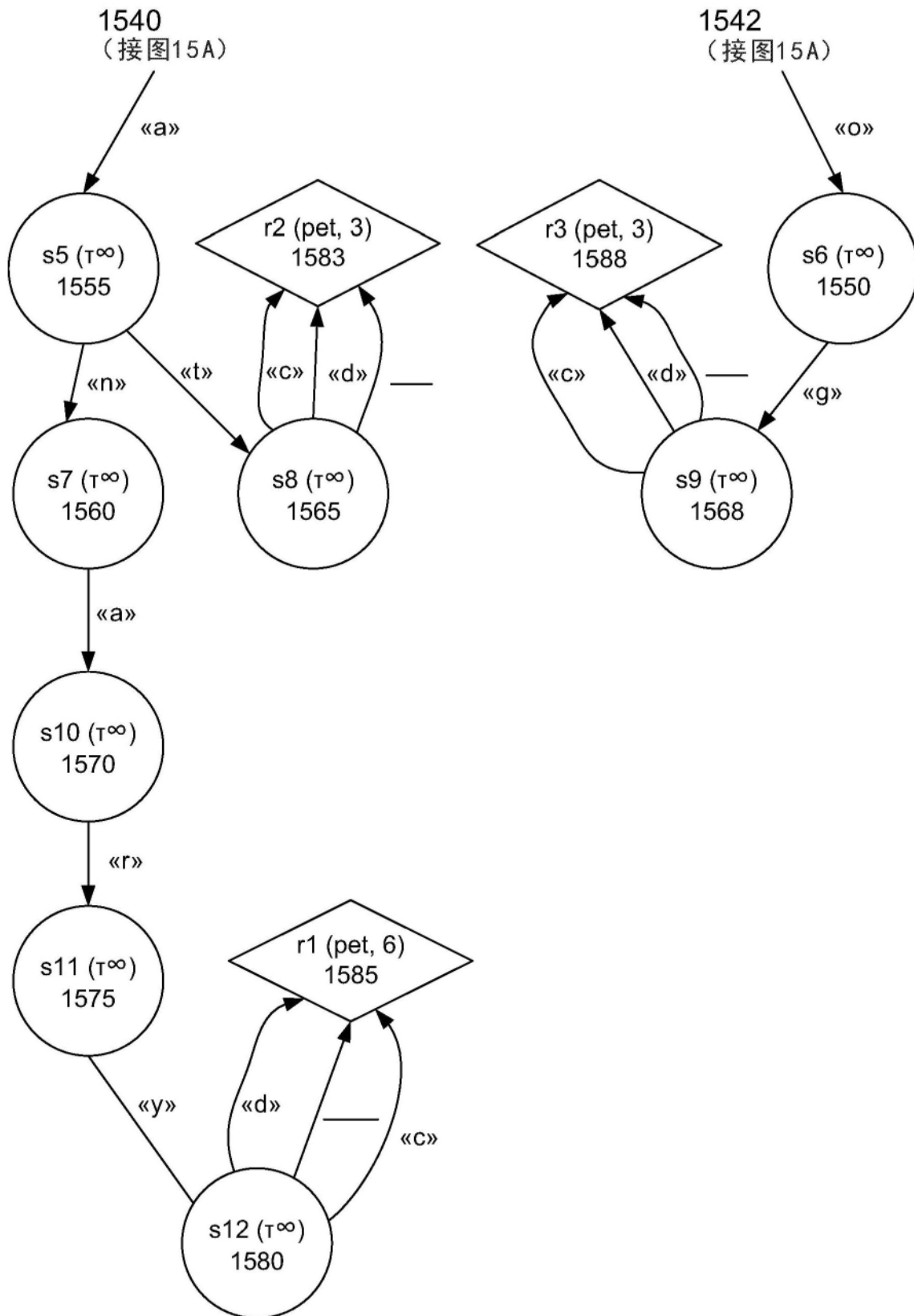


图15B