



(12) 发明专利申请

(10) 申请公布号 CN 119547410 A

(43) 申请公布日 2025. 02. 28

(21) 申请号 202380047625.0

(22) 申请日 2023.05.18

(30) 优先权数据

63/343,934 2022.05.19 US

63/392,039 2022.07.25 US

18/198,759 2023.05.17 US

18/198,762 2023.05.17 US

18/198,764 2023.05.17 US

18/198,765 2023.05.17 US

(85) PCT国际申请进入国家阶段日

2024.12.17

(86) PCT国际申请的申请数据

PCT/US2023/022765 2023.05.18

(87) PCT国际申请的公布数据

W02023/225219 EN 2023.11.23

(71) 申请人 甲骨文国际公司

地址 美国

(72) 发明人 T·兰格尔

(74) 专利代理机构 中国贸促会专利商标事务所
有限公司 11038

专利代理师 边海梅

(51) Int.Cl.

H04L 45/76 (2006.01)

H04L 67/02 (2006.01)

H04L 45/302 (2006.01)

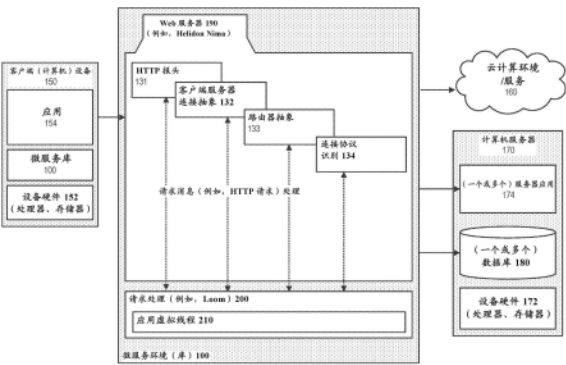
权利要求书3页 说明书72页 附图28页

(54) 发明名称

用于服务器计算环境中的报头处理的系统
和方法

(57) 摘要

根据实施例,本文描述了用于与微服务或其他计算环境一起使用的系统和方法,包括web服务器以及可用于构建云原生应用或服务的相关库和特征。根据各种实施例,系统和方法能够包括使用支持以下各项的各种组件或特征:(a) 报头处理、(b) 客户端和服务端连接抽象、(c) 路由器抽象以及 (d) 识别连接的协议。



1. 一种用于与微服务或其他计算环境一起使用的系统,包括:
包括一个或多个处理器的计算机,该计算机提供对微服务或其他计算环境的访问,以便与软件应用或服务一起使用;
其中该系统提供允许识别与请求消息相关联的网络通信协议的软件方法;以及
其中网络通信协议的识别被用于提供报头处理、客户端-服务器连接抽象、路由器抽象或者对套接字上使用多种网络通信协议的支持中的一个或多个。
2. 如权利要求1所述的系统,其中该系统经由报头枚举提供允许将消息报头视为对象的抽象,其中报头枚举可经由支持多种通信协议并允许客户端和服务端使用几种支持的协议中的任何一种协议来传达请求和响应消息的应用程序接口访问。
3. 如权利要求2所述的系统,其中当遇到具有已知报头类型的请求消息时,系统能够从枚举中获得如果可用的加索引的值,并将相关联的数据直接提供给请求所指向的进程和/或高速缓存报头值以供以后使用。
4. 如权利要求1所述的系统,其中在软件应用或服务的测试期间,客户端连接被替换为存储器内实施方式,该存储器内实施方式模拟网络通信协议的使用并向服务器连接暴露反函数,包括:
写入客户端连接的数据写入器的数据可通过服务器连接上的数据读取器获得,以及
写入服务器连接的数据写入器的数据可通过客户端连接上的数据读取器获得。
5. 如权利要求4所述的系统,其中存储器内实施方式使用存储器中的阻塞队列,该阻塞队列模拟网络通信协议的使用而无需使用实际套接字,其中测试已经完成,客户端和/或服务端代码能够对照实际套接字实现。
6. 如权利要求1所述的系统,其中该系统使得能够配置服务器,该服务器能够操作以经由多种网络或通信协议来服务于请求,包括:
提供将服务器支持的每种网络或通信协议与其自己的路由相关联的路由抽象,以及
路由抽象能够包括一个或多个特定路由以用于与特定网络或通信协议一起使用。
7. 如权利要求6所述的系统,其中对于经由特定协议接收的请求消息,该系统能够为该请求选择适当的路由并根据路由抽象对其进行处理。
8. 如权利要求1所述的系统,还包括可扩展服务提供商接口,该可扩展服务提供商接口提供允许识别与请求消息和客户端-服务器连接相关联的网络通信协议的软件方法;
其中网络通信协议的识别被用于提供客户端-服务器连接抽象、路由器抽象或者对单个套接字上使用多种网络通信协议的支持中的一个或多个。
9. 如权利要求8所述的系统,其中当从客户端接收到连接请求消息并将其指向服务器时,做出关于是否能够使用提供的网络通信协议来处置该请求的确定,其中协议识别处理返回系统不支持请求网络通信协议、或者其目前没有足够的数据来做出决定或者需要附加数据来做出这种确定的指示。
10. 如权利要求1所述的系统,其中该系统作为web服务器的一部分提供或者与web服务器一起使用,并且其中请求是HTTP请求。
11. 如权利要求1所述的系统,其中该系统包括使用虚拟线程以及可用于构建云原生服务的相关库和特征的请求处理组件。
12. 如权利要求1所述的系统,其中该系统在提供对一个或多个云、数据库或其他系统

或服务的访问的云计算环境内提供。

13. 一种用于与微服务或其他计算环境一起使用的方法,包括:

提供用于与软件应用或服务一起使用的微服务或其他计算环境;以及

提供允许识别与请求消息相关联的网络通信协议的软件方法;

其中网络通信协议的识别被用于提供报头处理、客户端-服务器连接抽象、路由器抽象或者对套接字上使用多种网络通信协议的支持中的一个或多个。

14. 如权利要求13所述的方法,还包括由系统经由报头枚举提供允许将消息报头视为对象的抽象,其中报头枚举可经由支持多种通信协议并允许客户端和服务端使用几种支持的协议中的任何一种协议来传达请求和响应消息的应用程序接口访问。

15. 如权利要求14所述的方法,其中当遇到具有已知报头类型的请求消息时,系统能够从枚举中获得如果可用的加索引的值,并将相关联的数据直接提供给请求所指向的进程和/或高速缓存报头值以供以后使用。

16. 如权利要求13所述的方法,还包括在软件应用或服务的测试期间,用存储器内实施方式替换客户端连接,该存储器内实施方式模拟网络通信协议的使用并向服务器连接暴露反函数,包括:

写入客户端连接的数据写入器的数据可通过服务器连接上的数据读取器获得,以及

写入服务器连接的数据写入器的数据可通过客户端连接上的数据读取器获得。

17. 如权利要求16所述的方法,其中存储器内实施方式使用存储器中的阻塞队列,该阻塞队列模拟网络通信协议的使用而无需使用实际套接字,其中测试已经完成,客户端和/或服务端代码能够对照实际套接字实现。

18. 如权利要求13所述的方法,由使得能够配置服务器的系统执行,该服务器能够操作以经由多种网络或通信协议来服务于请求,包括:

提供将服务器支持的每种网络或通信协议与其自己的路由相关联的路由抽象,以及

路由抽象能够包括一个或多个特定路由以用于与特定网络或通信协议一起使用。

19. 如权利要求18所述的方法,其中对于经由特定协议接收的请求消息,系统能够为该请求选择适当的路由并根据路由抽象对其进行处理。

20. 如权利要求13所述的方法,包括提供可扩展服务提供商接口,该可扩展服务提供商接口提供允许识别与请求消息和客户端-服务器连接相关联的网络通信协议的软件方法;

其中网络通信协议的识别被用于提供客户端-服务器连接抽象、路由器抽象或者对单个套接字上使用多种网络通信协议的支持中的一个或多个。

21. 如权利要求20所述的方法,由系统执行,其中当从客户端接收到连接请求消息并将其指向服务器时,做出关于是否能够使用提供的网络通信协议来处置该请求的确定,其中协议识别处理返回系统不支持请求网络通信协议、或者其目前没有足够的数据来做出决定或者需要附加数据来做出这种确定的指示。

22. 如权利要求13所述的方法,其中该方法作为web服务器的一部分提供或者与web服务器一起使用,并且其中请求是HTTP请求。

23. 如权利要求13所述的方法,其中该方法与使用虚拟线程以及可用于构建云原生服务的相关库和特征的处理组件一起操作。

24. 如权利要求13所述的方法,其中该方法在提供对一个或多个云、数据库或其他系统

或服务的访问的云计算环境内操作。

25.一种非暂态计算机可读存储介质,该非暂态计算机可读存储介质包括存储在其上的指令,指令在由一个或多个计算机读取并执行时使得所述一个或多个计算机执行一种方法,包括:

提供用于与软件应用或服务一起使用的微服务或其他计算环境;以及

提供允许识别与请求消息相关联的网络通信协议的软件方法;

其中网络通信协议的识别被用于提供报头处理、客户端-服务器连接抽象、路由器抽象或者对套接字上使用多种网络通信协议的支持中的一个或多个。

用于服务器计算环境中的报头处理的系统和方法

[0001] 版权声明

[0002] 本专利文档的公开内容的一部分包含受版权保护的素材。版权拥有者不反对任何人对专利文档或专利公开内容按照在专利商标局的专利文件或记录中出现的那样进行传真复制,但是除此之外在任何情况下都保留所有版权。

[0003] 优先权要求:

[0004] 本申请要求享有以下专利申请的优先权权益:于2022年5月19日提交的美国临时专利申请,标题为“SYSTEMS AND METHODS FOR USE WITH A MICROSERVICES OR OTHER COMPUTING ENVIRONMENT”,申请No.63/343,934;于2022年7月25日提交的美国临时专利申请,标题为“SYSTEMS AND METHODS FOR USE WITH A MICROSERVICES OR OTHER COMPUTING ENVIRONMENT”,申请No.63/392,039;于2023年5月17日提交的美国专利申请,标题为“SYSTEMS AND METHODS FOR HEADER PROCESSING IN A SERVER COMPUTING ENVIRONMENT”,申请No.18/198,759;于2023年5月17日提交的美国专利申请,标题为“SYSTEMS AND METHODS FOR CLIENT-SERVER CONNECTION ABSTRACTION IN A SERVER COMPUTING ENVIRONMENT”,申请No.18/198,762;于2023年5月17日提交的美国专利申请,标题为“SYSTEMS AND METHODS FOR ROUTER ABSTRACTION IN A SERVER COMPUTING ENVIRONMENT”,申请No.18/198,764;以及于2023年5月17日提交的美国专利申请,标题为“SYSTEMS AND METHODS FOR IDENTIFYING CONNECTION PROTOCOLS IN A SERVER COMPUTING ENVIRONMENT”,申请No.18/198,765;上述每个申请及其内容通过引用并入本文。

技术领域

[0005] 本文描述的实施例一般而言涉及云计算和其他计算环境、软件开发和微服务体系架构,并且特别针对用于支持应用开发以及可用于构建云原生应用或服务的相关库和特征的系统和方法。

背景技术

[0006] 微服务环境将软件应用呈现为松散耦合的服务的集合,这些服务可独立部署并通过网络(例如,云计算环境可能提供的云网络)彼此通信。微服务方法可以被用于例如开发要在云计算环境中作为云原生应用或服务提供的软件应用或服务。

发明内容

[0007] 根据实施例,本文描述了与微服务或其他计算环境一起使用的系统和方法,包括web服务器以及可用于构建云原生应用或服务的相关库和功能。

[0008] 根据各种实施例,系统和方法可以包括使用支持例如以下各项的各种组件或特征:(a)HTTP报头、(b)客户端和服务端连接抽象、(c)路由器抽象、以及(d)识别连接的协议。

[0009] 根据实施例,系统借助于报头枚举提供一种抽象,该抽象允许将消息报头视为对

象,可经由支持多种通信协议并允许客户端和服务端使用任何支持的协议来传达请求/响应消息的应用程序接口(API)访问。当遇到具有已知报头类型的请求消息时,系统可以从枚举中获得如果可用的索引值,并将相关联的数据(例如,作为数据字节)直接提供给请求所针对的进程和/或高速缓存报头值以供以后使用。

[0010] 根据实施例,为了支持软件应用或服务的测试,为了测试目的,客户端连接可以用存储器内实施方式来代替,该存储器内实施方式不调用例如HTTP,而是代替地向服务器连接暴露反函数;例如,写入客户端连接的数据写入器的数据字节可通过服务器连接上的数据读取器获得,反之亦然。在测试期间,存储器内实施方式使用存储器中的阻塞队列,该阻塞队列模拟HTTP协议的使用,但不使用实际套接字。一旦使用存储器内实施方式完成了测试(例如,单元测试或集成测试),就可以对照实际套接字测试客户端和/或服务器代码,以用于处理请求。

[0011] 根据实施例,该系统启用服务器的配置,该服务器可以经由多种网络或通信协议(诸如例如HTTP1、HTTP2上的gRPC和/或WebSocket)来处理请求。路由抽象允许每个协议具有其自己的路由,例如HTTP路由。子抽象可以提供更具体的路由,例如HTTP/1.1路由或HTTP/2路由。对于经由特定协议接收的请求消息,系统可以为该请求挑选适当的路由,并对其进行相应的处理。

[0012] 根据实施例,为了支持诸如客户端/服务器连接抽象、路由器抽象或在单个套接字上使用不同协议之类的功能,可扩展服务提供商接口(SPI)提供了允许对请求消息和连接进行协议识别的软件方法。当从客户端接收到指向服务器的连接请求消息时,做出关于是否可以适当处置该请求的确定。协议识别处理可以例如返回系统不支持请求协议、没有足够的数据来做出决定或需要附加的数据来做出这样的决定的指示。

附图说明

[0013] 图1图示了根据实施例的提供软件开发框架的微服务环境的各种示例。

[0014] 图2进一步图示了根据实施例的提供软件开发框架的微服务环境的各种示例。

[0015] 图3进一步图示了根据实施例的提供软件开发框架的微服务环境的各种示例。

[0016] 图4进一步图示了根据实施例的提供软件开发框架的微服务环境的各种示例。

[0017] 图5图示了根据实施例的用于开发在云环境中使用的应用的包括微服务环境的系统。

[0018] 图6进一步图示了根据实施例的使用微服务环境开发在云环境中使用的应用。

[0019] 图7图示了根据实施例的包括微服务环境的系统。

[0020] 图8图示了根据实施例的与微服务环境一起使用的web服务器。

[0021] 图9图示了根据实施例的对HTTP报头处理的支持。

[0022] 图10进一步图示了根据实施例的对HTTP报头处理的支持。

[0023] 图11进一步图示了根据实施例的对HTTP报头处理的支持。

[0024] 图12进一步图示了根据实施例的对HTTP报头处理的支持。

[0025] 图13图示了根据实施例的与微服务环境一起使用的、用于处理HTTP报头的方法。

[0026] 图14图示了根据实施例的对客户端和服务端抽象的支持。

[0027] 图15进一步图示了根据实施例的对客户端和服务端抽象的支持。

- [0028] 图16进一步图示了根据实施例的对客户端和服务端抽象的支持。
- [0029] 图17进一步图示了根据实施例的对客户端和服务端抽象的支持。
- [0030] 图18图示了根据实施例的与微服务环境一起使用的、用于提供客户端和服务端抽象的方法。
- [0031] 图19图示了根据实施例的对路由器抽象的支持。
- [0032] 图20进一步图示了根据实施例的对路由器抽象的支持。
- [0033] 图21进一步图示了根据实施例的对路由器抽象的支持。
- [0034] 图22进一步图示了根据实施例的对路由器抽象的支持。
- [0035] 图23图示了根据实施例的与微服务环境一起使用的、用于提供路由器抽象的方法。
- [0036] 图24图示了根据实施例的对识别连接的协议的支持。
- [0037] 图25进一步图示了根据实施例的对识别连接的协议的支持。
- [0038] 图26进一步图示了根据实施例的对识别连接的协议的支持。
- [0039] 图27进一步图示了根据实施例的对识别连接的协议的支持。
- [0040] 图28图示了根据实施例的与微服务环境一起使用的、用于识别连接的协议的方法。

具体实施方式

- [0041] 微服务环境将软件应用呈现为松散耦合的服务的集合,这些服务可独立部署并通过网络(例如,云计算环境可能提供的云网络)彼此通信。微服务方法可以被用于例如开发要在云计算环境中作为云原生应用或服务提供的软件应用或服务。
- [0042] 诸如Helidon之类的软件开发框架可以在微服务的开发中使用。例如,如下所述,Helidon环境提供了软件库的集合,这些软件库支持诸如配置、安全性或web服务器功能性之类的特征;并且为软件应用开发人员提供了开发云原生应用或服务的基础。
- [0043] 根据实施例,Helidon库可以与其他软件开发、部署和/或监视工具(诸如例如Docker、Kubernetes、Prometheus或Jaeger)互操作。
- [0044] 微服务环境
- [0045] 图1-图4图示了根据实施例的提供软件开发框架的微服务环境的各种示例。
- [0046] 根据实施例,图1-图4中所示的以及如本文关于各种实施例进一步描述的组件和处理可以被提供为可由计算机系统或其他类型的处理设备(例如云计算系统)执行的计算机软件或程序代码。
- [0047] 如图1中所示,根据实施例,微服务环境(例如,如由微服务库提供的)100可以提供例如一个或多个Helidon SE 110、Helidon MP 120和/或Helidon Nima 130编程模型或环境。微服务环境可以启用通过所开发的应用与云、数据库或其他系统或服务140的交互。
- [0048] 本文描述的Helidon微服务环境的各种实施例的示例是为了说明微服务计算环境或微服务环境的示例而提供的。根据各种实施例,本文描述的系统和方法可以与其他类型的微服务环境或计算环境一起使用。
- [0049] 根据各种实施例,系统和方法可以与其他类型的软件开发环境一起使用,或者与其他类型的web服务器以及可用于构建云原生服务的相关库和特征一起使用。

[0050] 如图2中所示,根据实施例,Helidon SE环境支持直接使用web服务器、安全性和配置组件的函数式编程风格;为应用开发人员提供透明度和控制力;并支持诸如反应式流以及异步和函数式编程之类的Java特征。

[0051] 例如,根据实施例,Helidon SE环境提供了一个框架,应用开发人员可以通过该框体构建轻量级反应式微服务。该环境可以包括各种库、应用程序接口(API)或其他组件,诸如例如反应式web服务器111,它提供用于创建web应用的异步和反应式API;配置API 112,它提供Java API以将配置特性以键/值形式加载和处理到配置对象中,然后应用可以使用该配置对象来检索配置数据;以及安全性组件113,它提供认证、授权和出站安全性;并且还可以包括度量114、健康检查115和跟踪116或其他组件。

[0052] 如图3中所示,根据实施例,Helidon MP环境通过使用构建在Helidon库之上的MicroProfile系列API支持声明式编程风格。MicroProfile定义(例如,如Eclipse MicroProfile项目所指定的)可以被用于支持跨多个MicroProfile运行时的应用可移植性。

[0053] 例如,根据实施例,Helidon MP环境可以包括各种库、API或其他组件,诸如例如JAX-RS122、JSON-P 126、上下文和依赖性注入(CDI) 124、度量121、健康检查123、容错125、MicroProfile(MP)配置137和JSON web令牌(JWT)认证128组件。Web服务器可以由非阻塞客户端/服务器/web框架118(诸如例如Netty)提供。

[0054] 如图4中所示,根据实施例,Helidon Nima环境提供了web服务器的基于Loom的实施方式以及可用于构建云原生服务的相关库和特征。

[0055] 例如,根据实施例,Helidon Nima环境可以包括使用支持以下各项的各种组件或特征:HTTP报头131、客户端和服务连接抽象132、路由器抽象133和连接协议识别134,应用开发人员可以通过它们构建例如阻塞(而非反应式)微服务。

[0056] 图5图示了根据实施例的用于开发在云环境中使用的应用的包括微服务环境的系统。

[0057] 如图5中所示,根据实施例,微服务环境将软件应用呈现为松散耦合的服务的集合,这些服务可独立部署并通过网络(例如通过云计算环境可能提供的云网络)彼此通信。

[0058] 例如,根据实施例,Helidon微服务环境可以支持使用远程过程调用(例如,gRPC)框架或组件,这使得(客户端和/或服务器)应用能够在微服务环境内进行通信,以构建连接的系统。

[0059] 根据实施例,远程过程调用框架启用可以被远程调用的服务和方法的定义。服务器或服务可以经由存根(stub)处置来自客户端的调用,该存根使得客户端应用能够直接调用服务器应用上的方法,就像它是存根一样。服务器/服务实现处置客户端调用的方法,包括解码传入请求、执行服务方法和编码服务响应。存根(stub)实现与服务相同的方法,将用于调用的参数包装在适当的协议缓冲区消息类型中,然后将其作为请求提供给服务器。

[0060] 根据实施例,微服务环境(例如,如由微服务库提供的)使得在具有设备硬件152(例如,152A、152B)的客户端计算机设备150(例如,150A、150B)处的客户端应用154(例如,154A、154B)能够进行访问,以与微服务通信或与在具有设备硬件172的计算机服务器设备170处的云160、服务器应用174、数据库180或其他系统或服务交互,以便访问数据、处理交易或执行与那些系统或服务相关联的其他操作。

[0061] 图6进一步图示了根据实施例的使用微服务环境开发在云环境中使用的应用。

[0062] 如图6中所示,根据实施例,微服务环境使得客户端应用能够访问以与云原生服务180通信或以其他方式与云、数据库或其他系统或服务交互,以便访问数据、处理交易或执行与那些系统或服务相关联的其他操作。

[0063] HTTP请求处理

[0064] 在云环境中,使客户端应用能够与服务器交互的web服务可以是基于HTTP的,即客户端应用和服务器可以使用HTTP通信协议作为以数据消息的形式在各个参与者之间传达请求和响应的手段。

[0065] 例如,在提供对一个或多个数据库或其他服务的访问的云服务中,客户端请求的处理可以致在导致几个初始调用,每个初始调用然后可以调用另一个服务来执行处理请求的一些附加方面。

[0066] 在一些云环境中,可以使用反应式编程技术来支持请求的并发处理。但是,由于反应式编程是非阻塞的,因此会难以对反应式应用的操作进行故障排除。例如,本来可能对评估各种线程的状况有用的线程转储(thread-dump)在反应式环境中使用时为测试或调试目的提供很少或不提供任何信息。

[0067] 根据实施例,为了解决此类考虑,系统可以包括请求处理组件,例如web服务器(WebServer)的基于Loom的实施方式,以及可用于构建云原生服务(本文中在一些实施例中称为Helidon Nima)的相关库和特征。

[0068] 根据实施例,这种请求处理组件支持使用虚拟线程的客户端-服务器通信的并发性。此类虚拟线程允许客户端HTTP应用以正常阻塞方式操作,例如请求与服务器进程的连接,而底层请求处理(例如,基于Loom的)环境处置请求处理的并发性。

[0069] 这提供了改进的代码开发处理,因为应用开发人员不必在应用本身内内置反应式编程。此外,在测试或调试期间,线程转储提供更多有用的信息以进行故障排除。因此,该方法提供了传统编程环境的特征以及反应式编程的并发性益处。

[0070] 例如,根据实施例,Helidon Nima环境允许应用开发人员设置HTTP端点,服务器可以通过该端点为web客户端提供服务。客户端应用可以在该环境内以阻塞方式运行,同时采用为每个连接提供虚拟线程的虚拟线程化,使得实现高并发性水平。

[0071] 图7图示了根据实施例的包括微服务环境的系统。

[0072] 如图7中所示,根据实施例,当与web服务器190(WebServer,例如,Helidon Nima环境)一起使用时,请求处理组件200(例如,web服务器的基于Loom的实施方式)允许使用阻塞代码,而无需使用大量平台线程,包括使用允许针对服务器的请求在应用虚拟线程210上运行的阻塞API。

[0073] 根据实施例,虚拟线程(在一些示例中称为纤程(fiber))作为使得能够执行高吞吐量并发应用的轻量级线程操作。一般而言,虚拟线程提供执行线程(例如,java.lang.Thread)的实例,该实例不绑定到特定操作系统(O/S)线程。应用代码可以在请求的持续时间内在虚拟线程中运行;虚拟线程进而在处理器上执行操作时消耗O/S线程。

[0074] 根据实施例,非阻塞API可以被同步API封装,并通过阻塞虚拟线程的进一步执行直到对应操作完成而使非阻塞API处于虚拟线程就绪状态。以这种方式,每个虚拟线程可以暂停一段时间,然后在后续时间继续。

[0075] 例如,如图7中所示,根据实施例,当在虚拟机220中执行时,可以在阻塞API 222处接收请求。多个载体线程224与底层操作系统230通信,例如经由在非阻塞API 232处注册事件和回调,或者经由O/S线程池226和阻塞API 234。

[0076] 根据实施例,这种请求处理组件允许应用开发人员在软件开发期间利用非阻塞API,而无需改变其每个请求的线程编程模型。所描述的方法还允许系统使用阻塞代码,诸如出站HTTP请求或数据库请求,而无需处置复杂的执行器服务。

[0077] 图8图示了根据实施例的与微服务环境一起使用的web服务器。

[0078] 如图8中所示,根据实施例,可以使用虚拟线程的web服务器(例如,web服务器的基于Loom的实施方式)来作为否则由反应式web服务器(例如,反应式web服务器111)和/或非阻塞客户端/服务器/web框架(例如,框架118)提供的功能性的替代方案或替换。

[0079] 如上所述,请求处理组件使用虚拟线程支持客户端-服务器通信的并发性。此类虚拟线程允许客户端HTTP应用以正常阻塞方式操作,例如,向服务器进程请求连接,而底层请求处理(例如,基于Loom的)环境处置请求处理的并发性。

[0080] 例如,根据Helidon SE环境的实施例,可以提供请求处理组件来作为通常在其中提供的反应式web服务器的替换,其中应用开发人员可以使用由web服务器提供的虚拟线程来支持阻塞的(而非反应式)应用处理,而不是使用反应式编程,同时提供类似的性能方面。

[0081] 根据各种实施例,在此可以提供的请求处理组件可以类似地被提供为web服务器,以与其他/新版本的Helidon(例如,Helidon Nima)一起使用。

[0082] 例如,根据实施例,从应用开发人员的角度来看,他们可以在其应用中包括或者阻塞代码,该阻塞代码直接使用请求处理组件(例如,基于Loom的web服务器)的虚拟线程化特征;或者包括本身是反应式但(例如,基于Loom的)web服务器可以使用web服务器提供的虚拟线程化特征对其进行处理(对应用开发人员而言是透明的)的应用代码。

[0083] Web服务器/Web客户端

[0084] 根据各种实施例,系统和方法可以包括使用支持例如以下各项的各种组件或特征:(a)HTTP报头、(b)客户端和服务器连接抽象、(c)路由器抽象、以及(d)识别连接的协议。

[0085] 根据实施例,可以将web服务器(WebServer)提供为可扩展模块,其默认实施方式支持HTTP/1.1协议。web服务器SPI提供可扩展性以支持例如:从HTTP/1协议升级,例如使用普通套接字升级到HTTP/2,或升级到WebSocket;其他协议,包括非HTTP协议,可通过基于二进制数据的发现进行扩展,诸如具有先验知识的HTTP/2;和/或对传输层安全性(TLS)请求的应用层协议协商(ALPN)支持,诸如HTTP/1.1、HTTP/2。

[0086] 根据实施例,扩展模块可以包括例如:HTTP/2协议实施方式;利用先前知识从HTTP/1升级到TCP上的HTTP/2(h2c,普通套接字);h2c(TLS)(ALPN支持);gRPC协议实施方式,作为HTTP/2的“子协议”;和/或从HTTP/1升级到WebSocket协议实施方式。

[0087] 根据实施例,web服务器支持HTTP(与版本无关)路由,支持特定于版本的路由;以及支持非HTTP协议路由的可扩展路由器API。一些扩展模块提供其自己的路由,例如:HTTP/2,它还支持默认HTTP路由(与版本无关);gRPC,包括支持所有消息类型(一元、客户端流、服务器流和双向)的特定于协议的路由;和/或WebSocket,特定于协议的路由,支持具有会话的监听器。

[0088] 根据实施例,可以提供作为可扩展模块的web客户端(WebClient),其默认实施方

式支持HTTP/1.1协议。web客户端SPI支持使用其他协议,以共享基本配置。根据实施例,可以包括以下扩展:HTTP/1.1客户端;HTTP/2客户端。

[0089] 根据实施例,WebServer和WebClient模块被设计为可测试的,例如用JUnit5测试。系统可以支持或者HTTP通信或者存储器内通信,这取决于所选择的注释,例如,@ServerTest通过套接字提供HTTP通信,以创建集成测试;和/或@RoutingTest提供存储器内通信,以创建单元测试。

[0090] 根据实施例,可以包括附加模块,例如:容错、针对“断路器”、“隔墙”、“后退”和“重试”场景的阻塞容错实施方式;可观察性端点,以获得关于应用的信息,诸如健康状况、度量、配置、应用信息等;提供可扩展性以将其他模块插入“观察”机制的手段;和/或跟踪,或开放遥测跟踪支持。

[0091] A.HTTP报头

[0092] 在经由不同通信协议支持客户端-服务器通信的云环境内,每种通信协议关于参与者之间传递的消息的格式化(包括消息报头)可以具有其自己的要求。

[0093] 例如,HTTP/1和HTTP/2协议对于经由这些协议发送请求或响应消息时使用的消息报头的格式有不同的要求。

[0094] 解析此类报头要求大量的计算开销,特别是当需要同时处理数千个请求时,如在云环境中可能遇到的。准备对此类请求的响应及其适当的报头类似地要求大量的计算开销。

[0095] 根据实施例,系统借助于报头枚举提供一种抽象,该抽象允许将消息报头视为对象,可经由支持多种通信协议并允许客户端和服务端使用任何支持的协议传达请求/响应消息的应用程序接口(API)访问。

[0096] 当遇到具有已知报头类型的请求消息时,系统可以从枚举中获得如果可用的索引值,并将相关联的数据(例如,作为数据字节)直接提供给请求所针对的进程和/或高速缓存报头值以供以后使用。

[0097] 例如,HTTP报头解析、处置和序列化是处理HTTP请求的计算更昂贵的部分之一。为了提高所有这些任务的性能,系统可以提供报头的新抽象,既用于报头的名称(接口HeaderName),也用于整个报头值(接口HeaderValue)。

[0098] 此外,随着HTTP/2的使用越来越频繁,使用在两个HTTP版本中都能很好地工作的报头是有利的。虽然Java中的实施方式可以使用String或CharSequence来识别报头,或优化报头名称的字节值,但它们不会对常用报头使用加索引的方法。

[0099] 所描述的方法避免了为每个请求查找报头映射的需要,并且提供了计算量的节省,特别是在处理大量请求时,如云环境中可能遇到的。

[0100] 图9图示了根据实施例的对HTTP报头处理的支持。

[0101] 如图9中所示,根据实施例,系统借助于报头枚举250提供了一种抽象,该抽象允许将消息报头视为对象,可经由支持多种通信协议并允许客户端和服务端使用任何支持的协议传达请求/响应消息的应用接口(API)访问。

[0102] 根据实施例,当遇到具有已知报头类型(例如,A1 252、A2 253、B1 256、B2 257)的请求消息时,系统可以从枚举中获取加索引的值(例如,A 254、B 258)(如果可用),并且将相关联的数据(例如,作为数据字节)直接提供给请求所针对的处理和/或高速缓存270报头

值以供以后使用。

[0103] 根据实施例,HTTP报头被用作对象类型。当创建已知报头时,系统可以高速缓存以下内容:值的数据字节(如果是单值);整个报头的完整HTTP/1字节(例如,“Content-Encoding:gzip”);该值是否敏感,并且不应当被日志记录,并且永远不会被加索引(HTTP/2);以及该值是否正在改变,并且不应当被高速缓存和加索引(HTTP/2)。

[0104] 根据实施例,对于从请求中解析出的报头,系统可以使用惰性评估,包括存储字节,直到服务器或其他处理实际请求报头值;因为通常存在从未被请求的报头,所以这使用较少的处理器(CPU)资源来处理每个请求。

[0105] 根据实施例,已知报头名称(即,常用的那些报头名称)在枚举(HeaderEnum)中实现。枚举保留:小写名称(String)-由HTTP/2使用,以防止频繁地将字符串小写;以及“默认大小写”名称(String)-由HTTP/1使用,以具有预期的报头名称;报头的索引,以支持请求和响应中优化的存储-以快速识别报头是否存在,并通过对数组的加索引的访问来优化对此类报头的检索。这是重大的改进,因为各种实施方式否则可能使用映射来存储报头,并且映射存储和检索操作比直接加索引的数组访问要昂贵得多。

[0106] 图10进一步图示了根据实施例的对HTTP报头处理的支持。

[0107] 如图10中所示,根据实施例,可以经由高级API访问报头对象260,该高级API可以作为名称API 262(例如,HeaderName接口)和值API 264(例如,HeaderValue接口)提供。

[0108] 根据实施例,HeaderName接口可以包括诸如以下的方法:

[0109] 方法String lowerCase()-报头的HTTP/2名称(诸如content-type)。

[0110] 方法String defaultCase()-报头的HTTP/1名称(诸如Content-Type)。

[0111] 方法int index()-用于识别“已知”报头(0+)和自定义报头(-1)。

[0112] 方法boolean isPseudoHeader()-HTTP/2伪报头,诸如HTTP/1中的“Host”是HTTP/2中的“:authority”伪报头。

[0113] 根据实施例,HeaderValue接口可以包括诸如以下的方法:

[0114] 方法HeaderName headerName()-报头的名称。

[0115] 方法String value()-报头的第一个值,可能由惰性评估支持。

[0116] 用于多值报头的其他方法。

[0117] 图11进一步图示了根据实施例的对HTTP报头处理的支持。

[0118] 如图11中所示,根据实施例,HTTP报头被用作对象类型。当创建具有已知报头282或由其报头名称284指示的请求280时,系统可以为值286高速缓存以下内容:值的数据字节(如果是单值);整个报头的完整HTTP/1字节(例如,“Content-Encoding:gzip”);该值是否敏感,并且不应当被日志记录,并且永远不会被加索引(HTTP/2);以及该值是否正在改变,并且不应当被高速缓存和加索引(HTTP/2)。

[0119] 图12进一步图示了根据实施例的对HTTP报头处理的支持。

[0120] 如图12中所示,根据实施例,对于从请求中解析出的报头,系统可以使用惰性评估并将字节存储为高速缓存的报头值288,直到服务器或其他处理实际请求该报头值。

[0121] 根据实施例,系统可以提供对各种接口、类或方法的支持,诸如例如通过下面的程序,该程序提供了仅在需要时从字节(和高速缓存的值)创建的字符串的实施方式,例如:

[0122] inner class Header

- [0123] constants on class
- [0124] methods
- [0125] create(String)
- [0126] create(String,String)
- [0127] createFromLowercase(String)
- [0128] inner class HeaderName
- [0129] inner class HeaderValue
- [0130] methods
- [0131] createCached(HeaderName,String)
- [0132] create(HeaderName,LazyString)
- [0133] create(HeaderName,String...)
- [0134] createCached(HeaderName,boolean,boolean,String)
- [0135] create(HeaderName,boolean,boolean,String...)
- [0136] inner class HeaderValues
- [0137] constants on class
- [0138] 上述示例仅用于说明目的;根据各种实施例,可以提供接口、类或方法的其他示例。
- [0139] 图13图示了根据实施例的与微服务环境一起使用的、用于处理HTTP报头的方法。
- [0140] 如图13中所示,根据实施例,在步骤290处,在包括一个或多个处理器及存储器的计算机中,提供微服务环境(例如,如由微服务库提供的),该环境使得客户端和服务端应用能够在微服务环境内通信。
- [0141] 在步骤292处,系统与HTTP报头处理组件或功能性相关联地提供存储报头值的高速缓存,以及针对多个报头类型中的每一个存储一个或多个报头类型名称和相关联的索引的报头枚举,以用于处理该报头类型的HTTP请求和响应。
- [0142] 在步骤294处,提供报头对象以及名称应用程序接口(API)和值API,用于设置或确定与HTTP请求相关联的报头名称和/或报头值。
- [0143] 在步骤296处,在接收到具有已知报头类型的请求后,系统将报头值存储到高速缓存,以便在服务器进程需要时进行后续的惰性检索和评估。
- [0144] 示例接口
- [0145] 根据实施例,下面为了说明目的而提供了用于提供HTTP报头的示例接口。根据其实施例,可以提供其他类型的接口。
- [0146] 接口HeaderName
- [0147] 根据实施例,Http.HeaderName接口可以被用于提供HTTP报头名称,并且可以包括各种方法,例如:
- [0148] 方法String lowerCase():这个报头的小写值,由HTTP/2使用,可以被用于通过HTTP/1进行的查找。这个值没有验证,因此如果包含大写字母,那么行为未定义。在实施例中,该方法返回:报头的名称,小写。
- [0149] 方法String defaultCase():如HTTP/1中使用的报头名称,或这个报头的“人类可读”(human-readable)值。在实施例中,该方法返回:报头的名称,可以使用大写和/或小写。

[0150] 方法 `int index()`: 这个报头的索引 (如果是已知的加索引的报头之一), 或 -1 (如果这是自定义报头名称)。在实施例中, 该方法返回: 这个报头的索引。

[0151] 方法 `boolean isPseudoHeader()`: HTTP/2 将伪报头定义为以: 字符开头的报头。这些被用于代替 HTTP/1 中的序言行 (用于定义路径、权限等) 以及响应中的状况行。在实施例中, 该方法返回: 这个报头是否是伪报头。

[0152] 接口 `HeaderValue`

[0153] 根据实施例, `Http.HeaderValue` 接口可以被用于提供 HTTP 报头值, 并且可以包括各种方法, 例如:

[0154] 方法 `static Http.HeaderValue createCached(Http.HeaderName name, String value)`。创建并高速缓存字节值。如果报头值存储在常量中或被重复地使用, 那么使用这个方法。可以提供的参数包括例如: `name`-报头名称; `value`-报头的值。在实施例中, 该方法返回: 新的报头。

[0155] 方法 `static Http.HeaderValue create(Http.HeaderName name, LazyString value)`。创建具有单个值的新报头。这个报头被视为不变且不敏感。可以提供的参数包括例如: `name`-报头的名称; `value`-带有值的惰性字符串。在实施例中, 该方法返回: 新的报头。

[0156] 方法 `static Http.HeaderValue create(Http.HeaderName name, String value)`。创建具有单个值的新报头。这个报头被视为不变且不敏感。可以提供的参数包括例如: `name`-报头的名称; `value`-报头的值。在实施例中, 该方法返回: 新的报头。

[0157] 方法 `static Http.HeaderValue create(Http.HeaderName name, String... values)`。创建新的报头。这个报头被视为不变且不敏感。可以提供的参数包括例如: `name`-报头的名称; `values`-报头的值。在实施例中, 该方法返回: 新的报头。

[0158] 方法 `static Http.HeaderValue create(Http.HeaderName name, List<String> values)`。创建新的报头。这个报头被视为不变且不敏感。可以提供的参数包括例如: `name`-报头的名称; `values`-报头的值。在实施例中, 该方法返回: 新的报头。

[0159] 方法 `static Http.HeaderValue createCached(Http.HeaderName name, boolean changing, boolean sensitive, String value)`。创建并高速缓存字节值。如果报头值存储在常量中或被重复地使用, 那么使用这个方法。可以提供的参数包括例如: `name`-报头名称; `changing`-值是否经常改变 (以针对 HTTP/2 禁用高速缓存); `sensitive`-值是否敏感 (以针对 HTTP/2 禁用高速缓存); `value`-报头的值。在实施例中, 该方法返回: 新的报头。

[0160] 方法 `static Http.HeaderValue create(Http.HeaderName name, boolean changing, boolean sensitive, String... values)`。创建新的报头。可以提供的参数包括例如: `name`-报头的名称; `changing`-值是否经常改变 (以针对 HTTP/2 禁用高速缓存); `sensitive`-值是否敏感 (以针对 HTTP/2 禁用高速缓存); `values`-报头的值。在实施例中, 该方法返回: 新的报头。

[0161] 方法 `String name()`。如由用户配置或在线路上接收的报头的名称。在实施例中, 该方法返回: 报头名称, 对于 HTTP/2 报头始终为小写。

[0162] 方法 `Http.HeaderName headerName()`。报头的报头名称。在实施例中, 该方法返回: 报头名称。

[0163] 方法 `String value()`。这个报头的第一个值。在实施例中, 该方法返回: 第一个值。

[0164] 方法List<String>allValues()。这个报头的所有值。在实施例中,该方法返回:所有配置的值。

[0165] 方法default List<String>allValues(boolean split)。这个报头的所有值。如果将这个报头定义为具有逗号分隔值的单个报头,那么将split设置为真。可以提供的参数包括例如:split-是否用逗号拆分单个值,如果该值已经是列表,那么不执行任何操作。在实施例中,该方法返回:值的列表。

[0166] 方法boolean sensitive()。敏感报头不应当被日志记录或加索引(HTTP/2)。在实施例中,该方法返回:这个报头是否敏感。

[0167] 方法boolean changing()。改变的报头不应当被高速缓存,并且其值不应当被加索引(HTTP/2)。在实施例中,该方法返回:这个报头的值是否经常改变。

[0168] 方法default byte[] valueBytes()。单值报头值的高速缓存的字节。在实施例中,该方法返回:值字节。

[0169] 方法default void writeHttp1Header(BufferData buffer)。将当前报头作为HTTP报头写入提供的缓冲区。可以提供的参数包括例如:buffer-要写入的缓冲区(应当不断增长)。

[0170] 报头实施方式

[0171] 根据实施例,Headers对象可以被用于存储从请求解析出的报头,或由用户配置以供后面检索的报头(或者由用户进行处理,或者由服务器通过网络写入)。以下示例说明了当获取和设置报头时要求利用直接访问的部分:

```
class Headers
```

```
// 已知报头的数组, 经由索引访问
```

```
HeaderValue[] knownHeaders = new HeaderValue[KNOWN_HEADER_SIZE]
```

[0172] **Map<HeaderName, HeaderValue> customHeaders = new Map()**

```
HeaderValue find(HeaderName name)
```

```
var index = name.index()
```

```
        return index == -1 ? customHeaders.get(name): knownHeaders[index]

    void set(HeaderValue value)

        var name = header.headerName()

[0173]    var index = name.index()

        if(index == -1){

            customHeaders.put(name, value)

            } else {

                knownHeaders[index] = value

            }
    }
```

[0174] B. 客户端和服务端连接抽象

[0175] 根据实施例,请求处理组件(诸如Loom)使用虚拟线程支持客户端-服务器通信的并发性。此类虚拟线程允许客户端HTTP应用以正常阻塞方式操作,例如请求与服务器进程的连接,而底层请求处理(例如,基于Loom的)环境处置请求处理的并发性。

[0176] 使用这种请求处理组件开发客户端应用的能力提供了更简单的代码开发处理。例如,应用开发人员不必在其应用本身内构建反应式编程。

[0177] 根据实施例,为了支持软件应用或服务的测试,为了测试目的,客户端连接可以被替换为存储器内实施方式,该存储器实施方式不调用例如HTTP,而是代替地向服务器连接暴露反函数;例如,写入客户端连接的数据写入器的数据字节可通过服务器连接上的数据读取器获得,反之亦然。

[0178] 在测试期间,存储器内实施方式使用存储器中的阻塞队列,该阻塞队列模拟HTTP协议的使用,但不使用实际套接字。一旦完成这种测试(例如,单元测试),就可以对照实际网络套接字测试客户端和/或服务器代码(例如,集成测试)。

[0179] 例如,可以期望创建可以易于测试的服务,而无需根据应用开发人员是否将执行单元测试或集成测试来重写相同的测试。通常,由于性能的优化(例如,使用事件循环机制、反应式代码等),抽象相当复杂。

[0180] 所描述的方法使得能够改进测试或调试;包括在测试或调试期间,线程转储为故障排除目的提供更多有用的信息。

[0181] 图14图示了根据实施例的对客户端和服务端抽象的支持。

[0182] 如图14中所示,根据实施例,为了支持软件应用或服务的测试,为了测试目的,客户端连接可以被替换为存储器内实施方式300,该存储器内实施方式300不调用例如HTTP,而是代替地向服务器连接暴露反函数;例如,写入客户端连接的数据写入器304的数据字节可通过服务器连接上的数据读取器306获得,反之亦然。

[0183] 图15进一步图示了根据实施例的对客户端和服务端抽象的支持。

[0184] 如图15中所示,根据实施例,在测试期间(例如,在测试针对服务器代码320的请求

310的处理时),存储器内实施方式使用存储器中的阻塞队列,其模拟HTTP协议的使用,但不使用实际套接字。

[0185] 图16进一步示出了根据实施例的对客户端和服务器的抽象的支持。

[0186] 如图16中所示,根据实施例,一旦使用存储器内实施方式完成了测试(例如,单元测试或集成测试),就可以针对实际套接字测试客户端330和/或服务器代码322,以用于处理请求312。

[0187] 图17进一步图示了根据实施例的对客户端和服务器的抽象的支持。

[0188] 如图17中所示,根据实施例,可以重复该处理,例如以更新服务器代码324、对照使用存储器中的阻塞队列的存储器内实施方式执行测试、以及对照实际套接字实现客户端332,以用于处理请求314。

[0189] 根据实施例,系统将连接抽象为两个基本操作:(1)它可以使用数据写入器(DataWriter)被写入;以及(2)它可以使用数据读取器(DataReader)被读取。

[0190] 根据实施例,该系统允许应用开发人员用存储器内实施方式替换客户端连接,该存储器内实施方式不会调用网络套接字通信,而是向服务器连接暴露反函数(写入客户端连接的数据写入器的字节可通过服务器连接上的数据读取器获得,反之亦然)。

[0191] 根据实施例,这允许应用开发人员编写单个应用代码来测试服务器代码,并且通过简单地实际或真实的HTTP客户端与存储器内HTTP客户端之间切换,应用开发人员可以在单元测试中测试路由代码的有效性,以及使用集成测试(例如,打开端口并通过HTTP协议调用它)测试整个服务器代码的有效性。

[0192] 根据实施例,使用该方法的测试的示例包括:创建实现HTTPIClient的DirectClient的实例的单元测试,该实例使用存储器内连接抽象;创建通过网络连接的HTTPIClient的实例的集成测试;或者获取HTTPIClient的实例并定义实际测试的抽象类,例如:

```
@Test
```

```
void testHealthObserver() {
```

```
    try(Http1ClientResponse response
```

[0193]

```
        = client.get("/observe/health").request()) {
```

```
        assertThat(response.status(), is(Http.Status.NO_CONTENT_204));
```

```
    }
```

```
}
```

[0194] 根据实施例,系统可以提供对各种接口、类或方法的支持,如下所示。

[0195] 根据实施例,可以提供字节的抽象(DataReader)可以作为可以拉取附加数据的数据读取器提供,例如:

```
DataReader {
```

[0196]

```
    /**
```

```
    * 来自字节的供应者的读取器。

    * @param bytesSupplier 可以拉取更多数据的供应者，

    * 而不管其源是什么（网络，或存储器）

    */
DataReader(Supplier<byte[]> bytesSupplier)

/**
    * 当前拉取的数据中可用的字节的数量。

    * @return 可用字节的数量

    */
int available()

[0197] /**
    * 拉取下一个数据。

    */
void pullData

/**
    * 跳过 n 个字节。

    * @param lenToSkip 要跳过的字节的数量（必须小于或等于当前容量）

    */
void skip(int lenToSkip)

/**
    * 确保我们有至少一个字节可用。

    */
```

```
void ensureAvailable()
```

```
/**
```

```
 * 读取 1 个字节.
```

```
 * @返回下一个字节
```

```
 */
```

```
byte read()
```

```
/**
```

```
 * 读取下一个缓冲区.
```

```
 * 将{#available()}个字节读取到缓冲区中并移动位置.
```

[0198]

```
 * @返回包装可用字节的缓冲区数据
```

```
 */
```

```
BufferData readBuffer()
```

```
/**
```

```
 * 读取定义的字节的下一个缓冲区. 如果长度不可用, 那么将拉取附加数据
```

```
 * 将移动位置.
```

```
 * @param length 要读取的数据的长度
```

```
 * @返回具有所请求的长度的缓冲区数据
```

```
 */
```

```
BufferData readBuffer(int length)
```

[0199] 根据实施例,可以消耗字节的抽象(DataWriter)可以被提供为数据写入器,其可以将数据写入底层传输(例如,套接字),例如:

```
interface DataWriter {  
    /**  
        * 写入缓冲区，可以延迟写入并且可以在不同的线程上写入。  
        * 这个方法还可以将对单个写入的多个调用组合到底层传输。  
        * @param buffers 要写入的缓冲区  
        */  
    void write(BufferData... buffers)  
}
```

[0200]

[0201] 根据实施例,可以提供允许使用自定义连接进行测试的客户端连接 (ClientConnection) 的抽象,例如:

```
interface ClientConnection {  
    /**  
        * 提供响应字节的数据读取器。  
        * @返回从这个连接读取的读取器  
        */  
    DataReader reader();  
    /**  
        * 客户端请求写入的数据写入器。  
        * @返回写入这个连接的写入器  
        */  
    DataWriter writer();  
}
```

[0202]

[0203] 根据实施例,可以提供存储器内客户端连接 (DirectClientConnection) 的实施方式,这种连接将绕过网络协议直接路由到服务器侧路由,例如:

```
class DirectClientConnection implements ClientConnection {

    Router router;

    DataReader clientReader;

    DataWriter clientWriter;

    DataReader serverReader;

    DataWriter serverWriter;

    DirectClientConnection(Router router)

        this.router = router;

    var serverToClient = new ArrayBlockingQueue<>(1024);
[0204] var clientToServer = new ArrayBlockingQueue<>(1024);

        this.clientReader = reader(serverToClient);

        this.clientWriter = writer(clientToServer);

        this.serverReader = reader(clientToServer);

        this.serverWriter = writer(serverToClient);

    }

    @Override
    public DataReader reader(){
        return clientReader;
    }

    @Override
    public DataWriter writer(){
```

```
        return clientWriter;

    }

    private DataWriter writer(ArrayBlockingQueue<byte[]> queue) {
        return new DataWriter() {

            @Override
            public void write(BufferData... buffers) {
                for(BufferData buffer : buffers) {
                    write(buffer);
                }
            }

            write(BufferData buffer) {

                byte[] bytes = new byte[buffer.available()];

                buffer.read(bytes);

[0205]         try {

                    queue.put(bytes);

                        } catch (InterruptedException e) {

                            throw new IllegalStateException("Thread interrupted", e);

                                }

                                    }

                                        };

                                            }

private DataReader reader(ArrayBlockingQueue<byte[]> queue) {
    return new DataReader() -> {

        byte[] data;

        try {

            data = queue.take();
```

```
        } catch (InterruptedException e) {  
            throw new IllegalArgumentException("Thread interrupted", e);  
        }  
        if (data.length == 0) {  
            return null;  
[0206]        }  
        return data;  
    });  
}
```

[0207] 根据实施例,可以提供用于HTTP/1.1的通过网络连接的实施方式,该实施方式使用带有网络套接字的读取器和写入器而不是存储器内数组,例如:

```
class Http1ClientConnection implements ClientConnection {  
    private DataReader reader;  
    private DataWriter writer;  
    private String host;  
[0208]    private int port;  
  
    @Override  
    public DataReader reader() {  
        return reader;  
    }  
}
```

```
        @Override
        public DataWriter writer() {
            return writer;
        }

        HttpClientConnection connect() {
            try {
                socket = new Socket();

                socket.connect(new InetSocketAddress(host, port),
                               (int)options.connectTimeout().toMillis());

            } catch(IOException e) {
[0209] throw new UncheckedIOException("Could not connect to
                " + host + ":" + port, e);
            }

            var helidoSocket = createSocket (socket) ;

            // 包装输入和输出流

            this.reader = new DataReader(helidonSocket);

            this.writer = new DataWriter() {
                @Override
                public void write(BufferData... buffers) {
                    for(BufferData buffer : buffers){
                        write(buffer);
                    }
                }
            }
        }
    }
}
```

```
        void write(BufferData buffer) {  
            helidonSocket.write(buffer);  
        }  
[0210]    };  
  
        return this;  
    }  
}
```

[0211] 根据实施例,客户端与服务器之间的通信可以针对客户端被抽象到各种组件中: ClientConnection(如上所述),以及基于请求/响应的协议的ClientRequest和ClientResponse。

[0212] 根据实施例,客户端与服务器之间的通信可以针对服务器被抽象到各种组件中: ServerConnection,以及基于请求/响应的协议的ServerRequest和ServerResponse。

[0213] 当实现客户端时,开发人员可以选择与新的ClientRequest一起使用的连接,从而提供或者基于网络的连接或者存储器内连接。

[0214] 当实现不基于请求/响应范式的客户端(诸如使用监听器的概念的WebSocket协议)时,客户端直接处置抽象,并在使用存储器连接时调用存储器中的服务器监听器,或通过网络调用服务器来处置服务器侧监听器。可以提供使用存储器内通信的直接WebSocket客户端,例如:

```
    DirectWsClient implements WsClient {  
        WsRouting routing; // WebSocket 路由,看路由抽象发明  
  
[0215]    // 存储器中的连接,使用所提供的客户端监听器  
  
        connect(URI uri, WsListener listener) {  
            HttpPrologue prologue = HttpPrologue.create ( "ws" , "ws" , ...)
```

```
        WsRoute route = routing.findRoute(prologue);

        // 发现要使用的 WebSocket 路由

        // 创建客户端监听器与服务器路由之间的存储器内连接

[0216]        DirectWsConnection conn = DirectWsConnection.create (prologue,
listener, route) ;

        conn.start () ; // 开始通信

    }

}
```

[0217] 根据实施例,可以提供使用网络上WebSocket客户端连接的实施方式(WsClientImpl)的方法连接,例如:

```
    DirectWsClient implements WsClient {

        connect(URI uri, WsListener listener){

            Socket socket = new Socket();

            Socket.connect(uri.getHost(), uri.getPort());

            var helidonSocket = helidonSocket(socket);

[0218]        handleUpgradeToWs(helidonSocket);

            ClientWsConnection session = new ClientWsConnection(listener,
helidonSocket);

            // 在执行器上运行客户端/服务器会话, 诸如 VirtualThread 执行器

            executor().submit(session);

        }

    }
```

[0219] 根据实施例,可以提供服务器侧等同物(ConnectionContext),其使用要由服务器连接使用的数据写入器和数据读取器,例如:

```
public interface ConnectionContext extends SocketContext {  
    /**  
        * 写入响应字节的数据写入器。  
        * @返回数据写入器  
        */  
    DataWriter dataWriter();  
  
    /**  
        * 要读取请求字节的数据读取器  
[0220]    * @返回数据读取器  
        */  
    DataReader dataReader();  
  
    /**  
        * 可以包含不同类型的路由的路由器 (HTTP, WebSocket, grpc) .  
        * @返回路由器  
        */  
    Router router();  
}
```

[0221] 上述示例仅出于说明目的而提供;根据各种实施例,可以提供接口、类或方法的其他示例。

[0222] 图18图示了根据实施例的与微服务环境一起使用的、用于提供客户端和服务端抽象的方法。

[0223] 如图18中所示,根据实施例,在步骤340处,在包括一个或多个处理器及存储器的计算机处,提供微服务环境(例如,如由微服务库提供的),该环境使得客户端和服务端应用能够在微服务环境内通信。

[0224] 在步骤342处,系统提供客户端服务器连接抽象组件或功能性,其中代替客户端连接,存储器内实施方式将反函数暴露给例如HTTP服务器连接,其中写入客户端连接的数据写入器(DataWriter)的数据字节可通过服务器连接上的数据读取器(DataReader)获得。

[0225] 在步骤344处,用户可以准备用于例如HTTP客户端的路由代码来测试服务器代码,并且在实际或真实客户端与客户端的存储器内实施方式之间切换以测试路由代码的有效性。

[0226] 在步骤346处,用户可以更新路由代码和/或服务器代码,并继续在实际或真实客户端与存储器内实施方式之间切换以继续测试。

[0227] 示例接口

[0228] 根据实施例,ClientConnection接口(上面说明了其示例)可以被用于提供允许使用自定义连接进行测试的客户端连接。

[0229] C. 路由器抽象

[0230] 在经由不同通信协议支持客户端-服务器通信的云环境中,每种通信协议对于参与者之间传递的消息的格式(包括消息报头)可以具有自己的要求。

[0231] 例如,HTTP/1和HTTP/2协议对经由这些协议发送请求或响应消息时使用的消息报头的格式有不同的要求。

[0232] 根据实施例,该系统启用服务器的配置,该服务器可以经由多种网络或通信协议(诸如例如HTTP1、HTTP2上的gRPC和/或WebSocket)来处理请求。

[0233] 路由抽象允许每种协议具有自己的路由,例如HTTP路由。子抽象可以提供更具体的路由,例如作为HTTP1路由或HTTP2路由。对于经由特定协议接收的连接/请求,系统可以为该请求挑选适当的路由,并相应地进行处理。

[0234] 例如,可以期望在服务器构建器上配置路由,该路由可以处置任何支持的协议,而无需对每种协议实施方式的依赖性。这要求对允许应用开发人员向服务器添加新路由的路由器和各种路由的抽象。例如,在Java环境中,只需在类路径上添加库即可提供这种抽象。其他实施方式可以支持跨多个协议版本的HTTP路由,但没有创建特定于版本的路由的能力。此外,一些实施方式要求根据用户所需的协议版本进行特定处置,或明确依赖这些协议。

[0235] 所描述的方法使得服务器的配置易于定义,以支持多种协议。

[0236] 图19图示了根据实施例的对路由器抽象的支持。

[0237] 如图19中所示,根据实施例,系统启用服务器的配置350,该服务器可以经由多种网络或通信协议来操作以服务于请求,每种协议与路由类型352(例如具有方法355的路由类型A 354)相关联,以与服务器代码360和构建器362一起使用。

[0238] 路由抽象允许每种协议有自己的路由,例如HTTP路由。子抽象可以提供更具体的路由,例如作为HTTP/1路由或HTTP/2路由。对于经由特定协议接收到的请求消息,系统可以为该请求挑选适当的路由,并相应地进行处理。

[0239] 图20进一步图示了根据实施例的对路由器抽象的支持。

[0240] 如图20中所示,根据实施例,构建器可以在服务器代码内为第一协议A提供协议支持364。系统类似地启用服务器的配置以支持具有方法359的路由类型B 358,以与服务器代码一起使用。

[0241] 图21进一步图示了根据实施例的对路由器抽象的支持。

[0242] 如图21中所示,根据实施例,构建器可以在服务器代码内为第二协议B提供协议支持366。

[0243] 图22进一步图示了根据实施例的对路由器抽象的支持。

[0244] 如图22中所示,根据实施例,配置的服务器可以经由多种网络或通信协议操作来服务于请求,诸如例如使用协议A 372的请求A 370;以及使用协议B 382的请求B 380。

[0245] 例如,根据实施例,这个路由抽象支持对任何HTTP版本的路由,具有创建特定于HTTP版本的路由的能力。可以添加针对其他类型协议的路由,例如从HTTP升级的路由(例如,WebSocket)、作为HTTP的子集的路由(例如,gRPC),或者甚至与HTTP无关的路由。

[0246] 根据实施例,这是通过对路由器(Router)进行抽象来实现的,该路由器可以包含多种类型的路由。当设置服务器时,应用开发人员可以向其构建器注册路由。然后,路由包含特定于协议的方法来启用该协议的配置。这些的组合将创建能够使用单个路由器配置服务于对所有支持的协议的请求的单个服务器,例如:

```
[0247] .addRouting(HttpRouting.builder())
[0248] .get("/", (req,res) ->res.send("Hello World"))
[0249] .route(Http1Route.route(GET,"/http1", (req,res)
[0250] ->res.send("Hello World 1")))
[0251] .route(Http2Route.route(GET,"/http2", (req,res)
[0252] ->res.send("Hello World 2"))))
[0253] .addRouting(GrpcRouting.builder())
[0254] .unary(Strings.getDescriptor(),
[0255] "StringService",
[0256] "Upper",
[0257] ProtocolsMain::grpcUpper))
[0258] .addRouting(WebSocketRouting.builder())
[0259] .endpoint("/websocket/echo",ProtocolsMain::wsEcho))
```

[0260] 根据实施例,所述方法的各种优点可以包括例如在同一服务器中组合协议的能力。协议是单独的模块,并且从路由API到每种协议实施方式没有依赖关系(例如,无需对服务器的修改即可添加附加的协议及其路由)。可以创建由任何HTTP版本提供服务的HTTP路由(上面所示的示例中的"/"路由),或者具有特定于HTTP版本的路由(上面所示的示例中的"/http1"和"/http2"路由)。该方法为用户提供了HTTP版本的完全抽象—代码可以完全与版本无关。该方法还提供了可扩展的机制,因此例如为了支持HTTP/3,不应当要求修改路由代码。

[0261] 根据实施例,系统可以提供对各种接口、类或方法的支持,如下所示。

[0262] 根据实施例,路由器接口(Router)可以被用于设置路由,可以包含多于一个协议路由。方法可以包括例如:routing(Class<T>,T) - 获得特定类型的路由,其中用于服务器的路由器包含各种类型的路由,因此服务器可以尽可能快地处置每种协议。例如:

```

public interface Router {
    /**
     * 获得特定类型的路由.
     * @param routingType 路由的类型
     * @param defaultValue 如果在这个路由器中未定义路由, 那么要使用的默认
[0263] 值
     * @param <T> 路由的类型
     * @返回定义的路由, 或者如果没有找到, 那么返回默认值
     */
    <T extends Routing> T routing(Class<T> routingType, T defaultValue);

    /**
     * 用于独立路由器的构建器.
     */
    interface Builder {
        /**
         * 向这个路由器添加新路由.
         * @param routing 要添加的路由,
[0264] * 诸如{@code webserver.http.HttpRouting}
         * @返回更新后的构建器
         */
        B addRouting (Routing routing) ;
    }
}

```

[0265] 根据实施例, 可以提供路由标记接口 (Routing), 例如:

```
/**  
 * Routing abstraction.  
[0266] */  
public interface Routing {  
}
```

[0267] 根据实施例,可以提供HTTP路由实施方式 (HttpRouting),其中路由能够处置任何HTTP版本,例如:

```
public final class HttpRouting implements Routing {  
[0268] private final ServiceRoute rootRoute;  
private HttpRouting(Builder builder){
```

```
        this.rootRoute = builder.rootRules.build();
    }

    /**
     * 创建{io.webserver.http.HttpRouting.Builder router builder}
     * 的新实例.
     * @返回新实例
     */
    public static Builder builder(){
        return new Builder();
    }

    /**
[0269]    * 路由请求.
     * 处置 HTTP 过滤器并找到路由.
     * @param ctx 上下文
     * @param request 路由请求
     * @param response 路由响应
     */
    public void route(ConnectionContext ctx,
        RoutingRequest request,
        RoutingResponse response){
        HttpPrologue prologue = request.prologue();

        RoutingResult result = RoutingResult.ROUTE;

        int counter = 0;
```

```
        while(result == RoutingResult.ROUTE){

            counter++;

            if(counter == 10){

                LOGGER.log (System.Logger.Level.ERROR,

                    "被重新路由多于 10 次。不会尝试进一步路由");

                throw HttpException.builder()

                    .request(HttpSimpleRequest.create(prologue,

request.headers()))

                    .type(SimpleHandler.EventType.INTERNAL_ERROR)

                    .build();

            }

            result = doRoute(ctx, request, response);

        }

        // 完成

        if(result == RoutingResult.FINISH){

            return;

        }

        throw HttpException.builder()

            .request(request)

            .response(response)

            .type(SimpleHandler.EventType.NOT_FOUND)

            .message("未找到端点")

            .build();

    }
```

[0270]

```
private RoutingResult doRoute(ConnectionContext ctx,
RoutingRequest request,
RoutingResponse response){

    HttpPrologue prologue = request.prologue();

    RouteCrawler crawler = rootRoute.crawler(ctx, request);

while(crawler.hasNext()){

response.resetRouting();

        RouteCrawler.CrawlerItem next = crawler.next();

request.path(next.path());

try {

[0271] next.handler().handle(request, response);

        if(response.shouldReroute()){
            if(response.isSent()){

return RoutingResult.FINISH;

                }

            HttpPrologue newPrologue =
response.reroutePrologue(prologue);

request.prologue(newPrologue);

response.resetRouting();

return RoutingResult.ROUTE;

        }

        if(response.isNexted()){
            if(response.isSent()){
```

```
        return RoutingResult.FINISH;
    }

    continue;
}

if(!response.hasEntity()){
    // 不接下来，不重新路由-就发送！

    response.send();
}

return RoutingResult.FINISH;
    } catch(CloseConnectionException | HttpException e){
throw e;
    } catch(Exception thrown){
[0272] if(thrown.getCause() instanceof SocketException se){
throw new UncheckedIOException(se);
    }

    if(!response.isSent()){
        boolean keepAlive = true;
        try {
            request.content().consume();

            } catch (Exception e) {

keepAlive = request.content().consumed();
        }

        // 我们必须关闭连接，因为我们不会消耗请求

        throw HttpException.builder()
```

```
OR)                                .type(SimpleHandler.EventType.INTERNAL_ERR

                                    .cause(thrown)
                                    .request(request)
                                    .response(response)
                                    .message(thrown.getMessage())
                                    .setKeepAlive(keepAlive)
                                    .build();
                                }
                            }
                        }

return RoutingResult.NONE;

[0273]     }

private enum RoutingResult {
    ROUTE,
    FINISH,
    NONE
}

/**
 * 用于{io.webserver.http.HttpRouting}的流利 API 构建器.
 */
public static class Builder implements HttpRules,
common.Builder<Builder, HttpRouting> {
    private final ServiceRules rootRules = new ServiceRules();
```

```
private Builder(){  
    }  
  
    @Override  
public HttpRouting build(){  
    return new HttpRouting(this);  
    }  
  
    @Override  
public Builder route(HttpRoute route){  
    rootRules.route(route);  
    return this;  
    }
```

[0274]

```
    @Override  
public Builder route(Http.Method method, String pathPattern,  
    Handler handler){  
    return route(HttpRoute.builder()  
        .methods(method)  
        .path(pathPattern)  
        .handler(handler));  
    }  
  
    /**  
    * 添加获取路由。  
    * @param pathPattern 路径模式  
    * @param handler 要使用的处理程序
```

```

        * @返回更新后的构建器
        */

    public Builder get(String pathPattern, Handler handler) {
        return route(HttpRoute.builder()
[0275]             .methods(Http.Method.GET)
                .path(pathPattern)
                .handler(handler));
    }
}

```

[0276] 根据实施例,HTTP服务抽象 (HttpService) 提供了形成单个单元的路由的逻辑分组,例如:

```

/**
 * 封装{HttpRouting routing}规则的集合和相关逻辑.
 * <p>
 * 实例可以使用
 * {HttpRouting.Builder#register (java.util.function.Supplier[])}
 * 方法被分配给{HttpRouting routing}
[0277] */
@FunctionalInterface
public interface HttpService {
    /**
     * 更新路由以添加这个服务的处理程序.
     * @param rules 要更新的规则
     */
    void routing(HttpRules rules);
[0278] }

```

[0279] 根据实施例,HTTP路由抽象 (HttpRoute) 提供了路径表达式、HTTP方法谓词和处置匹配请求的处理程序的组合,包括基本HTTP路由 (应当可由任何HTTP协议版本使用,例如:

```
public interface HttpRoute extends Route {  
    /**  
        * 构建新 HTTP 路由的构建器。  
        * @返回构建器  
        */  
    static HttpRouteImpl.Builder builder() {  
        return new Builder();  
    }  
  
    /**  
[0280]    * 这个路由是否接受提供的请求。  
        * @param prologue 请求的序言  
        * @返回验证的结果  
        * @see PathMatchers.MatchResult#notAccepted ()  
        */  
    PathMatchers.MatchResult accepts (HttpPrologue prologue) ;  
  
    /**  
        * 这个路由的处理程序。  
    */  
}
```

```
    * @返回处理程序
    */

    Handler handler();

    /**
     * 用于{io.webserver.http.HttpRoute}的流利 API 构建器.
     */

    class Builder implements common.Builder<Builder, HttpRoute> {
        private Predicate<Http.Method> methodPredicate =
            Http.Method.predicate();

        private PathMatcher pathMatcher = PathMatchers.any();

        private Handler handler;
[0281] private Builder(){

            }

            @Override
            public HttpRoute build(){
                Objects.requireNonNull(handler, "Handler must be provided");

                return new HttpRouteImpl(this);

            }

            /**
             * 这个路由应当处置的 HTTP 方法.
             * @param methods 要处置的方法
             * @返回更新后的构建器
            */
        }
```

```
        */

    public Builder methods(Http.Method... methods) {

        return methods(Http.Method.predicate(methods));

    }

    /**

    * 要使用的方法谓词.

    * @param methodPredicate 方法谓词

    * @返回更新后的构建器

    */

    public Builder methods(Predicate<Http.Method> methodPredicate) {

        this.methodPredicate = methodPredicate;

        return this;

    }

    /**

    * 要处置的路径模式.

    * @param pathPattern 路径模式

    * @返回更新后的构建器

    */

    public Builder path(String pathPattern) {

        return this.path(PathMatchers.create(pathPattern));

    }

    /**

    * 处置路径的路径匹配器.
```

```
        * @param pathMatch 路径匹配器

        * @返回更新后的构建器

        */

    public Builder path(PathMatcher pathMatch) {

        this.pathMatcher = pathMatch;

    return this;

    }

    /**

    * 要使用的处理程序.

    * @param handler 处理程序

    * @返回更新后的构建器

    */

    public Builder handler(Handler handler) {

        this.handler = handler;

    return this;

    }

    Predicate<Http.Method> methodPredicate() {

    return methodPredicate;

    }

    PathMatcher pathPredicate() {

    return pathMatch;

    }
```

[0283]

```
        Handler handler() {  
            return handler;  
[0284]        }  
    }  
}  
[0285] 根据实施例,系统可以支持多种实施方式,例如,特定于HTTP/1的路由实施方式  
(Http1Route):  
    /**  
        * 创建特定于 HTTP/1 的路由.  
        * @param method 接受的方法  
        * @param path 路径模式  
        * @param handler 处理程序  
        * @返回新的特定于 HTTP/1.1 的路由  
        */  
[0286] public static Http1Route route(Http.Method method, String path,  
    Handler handler) {  
    return new Http1Route(Http.Method.predicate(method),  
    PathMatchers.create(path), handler);  
    }  
  
    @Override  
    public PathMatchers.MatchResult accepts(HttpPrologue prologue) {  
        if(!prologue.protocolVersion().equals("1.1")) {  
            return PathMatchers.MatchResult.notAccepted();  
        }  
    }  
}
```

```
        }

        if(!methodPredicate.test(prologue.method())){

            return PathMatchers.MatchResult.notAccepted();

        }

        return pathMatcher.match(prologue.uriPath());

    }

    @Override
    public Handler handler(){

        return handler;

[0287]     }

    @Override
    public void beforeStart(){

        handler.beforeStart();

    }

    @Override
    public void afterStop(){

        handler.afterStop();

    }

}

[0288] 类似地,根据实施例,系统可以支持特定于HTTP/2的路由实施方式(Http2Route),
例如:
[0289] /**
```

- * 创建新的仅 HTTP/2 路由.
- * @param method 要处置的方法
- * @param path 路径模式
- * @param handler 处理程序
- * @返回新的路由
- */

```
public static Http2Route route(Http.Method method, String path,
    Handler handler) {
    return new Http2Route(Http.Method.predicate(method),
        PathMatchers.create(path), handler);
}
@Override
```

```
[0290] public PathMatchers.MatchResult accepts(HttpPrologue prologue) {
    if(!prologue.protocolVersion().equals("2.0")) {
        return PathMatchers.MatchResult.notAccepted();
    }
    if(!methodPredicate.test(prologue.method())) {
        return PathMatchers.MatchResult.notAccepted();
    }
}
```

```
return pathMatcher.match(prologue.uriPath());
}
```

```
@Override
public Handler handler() {
    return handler;
}
```

```
}
```

```
    @Override  
    public void beforeStart() {  
        handler.beforeStart();  
    }  
}
```

[0291]

```
    @Override  
    public void afterStop() {  
        handler.afterStop();  
    }  
}
```

[0292] 根据实施例,可以提供提供特定于WebSocket的路由的WebSocket路由实施方式(WebSocketRouting),例如:

```
public class WebSocketRouting implements Routing {  
    private final List<WebSocket> routes;  
    private WebSocketRouting(Builder builder) {  
        this.routes = new ArrayList<>(builder.routes);  
    }  
}
```

[0293]

```
/**  
 * 用于 WebSocket 路由的构建器。  
 * @返回新的构建器  
 */  
public static Builder builder() {
```

```
return new Builder();

    }

    WebSocket findRoute(HttpPrologue prologue){
for(WebSocket route : routes){

    PathMatchers.MatchResult accepts = route.accepts(prologue);
if(accepts.accepted()){
return route;

    }

}

return null;

}

[0294]

/**
 * 用于{websocket.webserver.WebSocketRouting}的流利 API 构建器.
 */
public static class Builder implements common.Builder<Builder,
WebSocketRouting> {

private final List<WebSocket> routes = new ArrayList<>();

private Builder(){

    }

    @Override
public WebSocketRouting build(){
return new WebSocketRouting(this);
```

```

    }

    /**
     * 添加端点.
     * @param path 端点的路径
     * @param listener 要使用的监听器
     * @返回更新后的构建器
     */
    public Builder endpoint(String path, WsListener listener){
        return route(WebSocket.create(path, listener));
    }
    /**
     * 添加端点.
     * @param path 端点的路径
     * @param listener 监听器供应者
     * @返回更新后的构建器
     */
    public Builder endpoint(String path , Supplier<WsListener>
listener){
        return route(WebSocket.create(path, listener));
    }

    private Builder route(WebSocket wsRoute){
        routes.add(wsRoute);
        return this;
    }
}
}

```

[0297] 根据实施例,上面使用WsListener(WsListener)类来配置WebSocket端点,例如:

```
public interface WsListener {  
    /**  
     * 接收文本碎片。  
     * @param session WebSocket 会话  
     * @param text 接收到的文本  
     * @param last 这是否是最后一个碎片  
     */  
    void onMessage(WsSession session, String text, boolean last);  
  
[0298]    /**  
     * 接收二进制碎片。  
     * @param session WebSocket 会话  
     * @param buffer 具有数据的缓冲区  
     * @param last 这是否是最后一个碎片  
     */  
    void onMessage(WsSession session, BufferData buffer, boolean last);  
  
    /**  
     * 接收到的 ping.
```

```
    * @param session WebSocket 会话

    * @param buffer 具有数据的缓冲区

    */

void onPing(WsSession session, BufferData buffer);

/**

    * 接收到的 pong.

    * @param session WebSocket 会话

    * @param buffer 具有数据的缓冲区

    */

void onPong(WsSession session, BufferData buffer);
```

[0299]

```
/**

    * 接收到的关闭.

    * @param session WebSocket 会话

    * @param status 关闭状况

    * @param reason 关闭的原因

    */

void onClose(WsSession session, int status, String reason);

/**

    * 错误发生.

    * @param session WebSocket 会话

    * @param t 可抛出的被捕获
```

```
        */

        void onError(WsSession session, Throwable t);

        /**
[0300]     * 会话打开.

        * @param session WebSocket 会话

        */

        void onOpen(WsSession session);
    }
[0301] 根据实施例,可以提供提供特定于gRPC的路由的gRPC路由实施方式(GrpcRouting),
例如:

    public class GrpcRouting implements Routing {

        private final ArrayList<GrpcRoute> routes;

        private GrpcRouting(Builder builder) {

            this.routes = new ArrayList<>(builder.routes);

        }

[0302]     /**

        * 新路由构建器.

        * @return 新构建器

        */

        public static Builder builder() {

            return new Builder();

        }
    }
```

```
        Grpc<?, ?> findRoute(HttpPrologue prologue){
    for(GrpcRoute route : routes){
        PathMatchers.MatchResult accepts = route.accepts(prologue);
        if(accepts.accepted()){
            return route.toGrpc(prologue);
        }
    }

    return null;
}

/**
[0303]     * 用于{GrpcRouting}的流利 API 构建器.
        */

    public static class Builder implements common.Builder<Builder,
GrpcRouting> {
        private final List<GrpcRoute> routes = new LinkedList<>();

        private Builder(){
        }

        @Override
        public GrpcRouting build(){
            return new GrpcRouting(this);
        }
    }
}
```

```
/**
 * 配置 grpc 服务.
 * @param service 要添加的服务
 * @返回更新后的构建器
 */
public Builder service(GrpcService service){
return route(GrpcServiceRoute.create(service));
}

/**
 * 一元路由.
 * @param proto proto 描述符
[0304] * @param serviceName 服务名称
 * @param methodName 方法名称
 * @param method 要处置这个路由的方法
 * @param <ReqT> 请求类型
 * @param <ResT> 响应类型
 * @返回更新后的构建器
 */
public <ReqT, ResT> Builder unary(Descriptors.FileDescriptor
proto, String serviceName, String methodName,
ServerCalls.UnaryMethod<ReqT, ResT> method){
return route(Grpc.unary(proto, serviceName, methodName,
method));
```

```
    }

    /**
     * 双向路由.
     * @param proto proto 描述符
     * @param serviceName 服务名称
     * @param methodName 方法名称
     * @param method 要处置这个路由的方法
     * @param <ReqT> 请求类型
     * @param <ResT> 响应类型
     * @返回更新后的构建器
     */
[0305] public <ReqT, ResT> Builder bidi(Descriptors.FileDescriptor
    proto, String serviceName, String methodName,
        ServerCalls.BidiStreamingMethod<ReqT, ResT> method) {
    return route(Grpc.bidi(proto, serviceName, methodName, method));
}

    /**
     * 服务器流传输路由.
     * @param proto proto 描述符
     * @param serviceName 服务名称
     * @param methodName 方法名称
     * @param method 要处置这个路由的方法
```

```

        * @param <ReqT> 请求类型

        * @param <ResT> 响应类型

        * @返回更新后的构建器

        */

    public <ReqT , ResT> Builder serverStream
    (Descriptors.FileDescriptor
    proto, String serviceName, String methodName,
    ServerCalls.ServerStreamingMethod<ReqT, ResT> method) {
    return route(Grpc.serverStream(proto, serviceName, methodName,
    method));
    }

```

[0306]

```

    /**

    * 客户端流传输路由.

    * @param proto proto 描述符

    * @param serviceName 服务名称

    * @param methodName 方法名称

    * @param method 要处置这个路由的方法

    * @param <ReqT> 请求类型

    * @param <ResT> 响应类型

    * @返回更新后的构建器

    */

    public <ReqT , ResT> Builder
    clientStream(Descriptors.FileDescriptor

```

```
proto, String serviceName, String methodName,  
  
ServerCalls.ClientStreamingMethod<ReqT, ResT> method) {  
  
return route(Grpc.clientStream(proto, serviceName, methodName,  
method));  
  
}
```

[0307]

```
private Builder route(GrpcRoute route) {  
  
routes.add(route);  
  
return this;  
  
}  
  
}  
  
}
```

[0308] 根据实施例,可以提供gRPC服务 (GrpcService),作为由单个proto文件支持的单个gRPC服务,例如:

```
public interface GrpcService {  
  
/**  
  
* 这个服务的 proto 描述符.  
  
* @返回 proto 文件描述符  
  
*/
```

[0309]

```
Descriptors.FileDescriptor proto();
```

```
/**  
  
* 服务名称, 默认为这个类的简单名称.  
  
* @返回服务名称  
  
*/
```

```
default String serviceName() {  
    return getClass().getSimpleName();  
}  
  
/**  
 * 更新路由.  
 * @param routing 路由  
 */  
void update(Routing routing);  
  
/**  
 * 特定于服务的路由 (proto 描述符由{GrpcService#proto()}提供).  
 */  
[0310] interface Routing {  
    /**  
     * 一元路由.  
     * @param methodName 方法名称  
     * @param method 要处置该路由的方法  
     * @param <ReqT> 请求类型  
     * @param <ResT> 响应类型  
     * @返回更新后的路由  
     */  
    <ReqT, ResT> Routing unary(String methodName,  
        ServerCalls.UnaryMethod<ReqT, ResT> method);  
}
```

```
/**
 * 双向路由.
 * @param methodName 方法名称
 * @param method 要处置该路由的方法
 * @param <ReqT> 请求类型
 * @param <ResT> 响应类型
 * @返回更新后的路由
 */
<ReqT, ResT> Routing bidi(String methodName,
    ServerCalls.BidiStreamingMethod<ReqT, ResT> method);

[0311] /**
 * 服务器流传输路由.
 * @param methodName 方法名称
 * @param method 要处置该路由的方法
 * @param <ReqT> 请求类型
 * @param <ResT> 响应类型
 * @返回更新后的路由
 */
<ReqT, ResT> Routing serverStream(String methodName,
    ServerCalls.ServerStreamingMethod<ReqT, ResT> method);

/**
 * 客户端流传输路由.
```

```

        * @param methodName 方法名称

        * @param method 要处置该路由的方法

        * @param <ReqT> 请求类型

        * @param <ResT> 响应类型

[0312]      * @返回更新后的路由

        */

    <ReqT, ResT> Routing clientStream(String methodName,

        ServerCalls.ClientStreamingMethod<ReqT, ResT> method);

    }

}

```

[0313] 上述示例是为了说明目的而提供的;根据各种实施例,可以提供接口、类或方法的其他示例。

[0314] 图23图示了根据实施例的与微服务环境一起使用的、用于提供路由器抽象的方法。

[0315] 如图23中所示,根据实施例,在步骤390处,在包括一个或多个处理器及存储器的计算机处,提供微服务环境(例如,如由微服务库提供的),该环境使得客户端和服务器应用能够在微服务环境内通信。

[0316] 在步骤392处,系统提供路由器抽象组件或功能性与例如HTTP请求路由一起使用,包括允许通过在类路径上添加库以向服务器添加新的路由来配置服务器代码的路由器和路由的抽象。

[0317] 在步骤394处,当设置服务器时,服务器可以向其构建器注册路由,其中路由包含特定于协议的方法以启用该协议的配置,并且其中路由器支持与其相关联的多种协议。

[0318] 在步骤396处,配置的服务器使用该服务器配置和支持的协议接收和处理来自客户端的请求。

[0319] 示例接口

[0320] 根据实施例,下面为了说明目的提供用于提供路由器抽象的示例接口。根据其他实施例,可以提供其他类型的接口。

[0321] 接口路由器

[0322] 根据实施例,路由器(Router)接口可以被用于提供各种类型的路由,以便服务器可以快速处置每种协议,并且可以包括各种方法,例如:

[0323] 方法static Router.Builder builder()。路由器的构建器。在实施例中,该方法返回:新的构建器。

[0324] 方法static Router empty()。空路由器。在实施例中,该方法返回:新的空路由

器。

[0325] 方法 `routing<T extends Routing>T routing(Class<T>routingType, T defaultValue)`。获取特定类型的路由。可以提供的类型参数包括例如：`T`-路由的类型。可以提供的参数包括例如：`routingType`-路由的类型；`defaultValue`-如果这个路由器中未定义路由，那么要使用的默认值。在实施例中，该方法返回：定义的路由或如果未找到则返回默认值。

[0326] D. 识别连接的协议

[0327] 在经由不同通信协议支持客户端-服务器通信的云环境内，每种通信协议对于参与者之间传递的消息的格式化（包括消息报头）可以具有自己的要求。

[0328] 例如，HTTP/1和HTTP/2协议对于当经由这些协议发送请求或响应消息时使用的消息报头的格式有不同的要求。

[0329] 根据实施例，为了支持诸如客户端/服务器连接抽象、路由器抽象或在单个套接字上使用不同协议之类的特征，可扩展服务提供商接口（SPI）提供了允许协议识别的软件方法用于请求消息和连接。

[0330] 当从客户端接收到指向服务器的连接请求消息时，将确定是否可以适当地处置该请求。例如，协议识别处理可以返回系统不支持请求协议、没有足够的数据来做出决定或需要附加数据来做出这种确定的指示。

[0331] 例如，在一些系统中，当服务器接收到连接请求时，可能不可能知道所请求的连接是什么类型。如本文所述，该方法允许具有服务于不同类型的连接的单个服务器套接字。通常，对于不兼容的连接类型，必须使用不同的套接字。对于HTTP/2，一些服务器只能支持从HTTP/1升级，或者将先前的知识请求映射到一些特殊的HTTP/1请求。

[0332] 所述方法使得能够用支持的任何协议调用服务器并且服务器将自动确定它是否可以接受该协议。

[0333] 图24图示了根据实施例的对识别连接的协议的支持。

[0334] 如图24中所示，根据实施例，为了支持诸如客户端/服务器连接抽象、路由器抽象或在单个套接字上使用不同协议之类的特征，可扩展服务提供商接口（SPI）402提供了允许协议识别的软件方法用于请求消息和连接。

[0335] 根据实施例，构建器可以在服务器代码内为第一协议A 412和第二协议B 414提供协议支持。

[0336] 当从客户端接收到指向服务器的连接请求消息时，做出关于是否可以适当处置该请求的确定。例如，协议识别处理可以返回系统不支持请求协议、没有足够的数据来做出决定或需附加数据来做出这种确定的指示。

[0337] 图25进一步图示了根据实施例的对识别连接的协议的支持。

[0338] 如图25中所示，根据实施例，可扩展服务提供商接口（SPI）402可以识别请求消息和连接的协议，例如具有与第一协议A相关联的第一连接提供程序404、与第二协议B相关联的第二连接提供程序406，并且在这个示例中，具有与服务器目前可能不支持的第三或其他协议N相关联的第三连接提供程序408。

[0339] 图26进一步图示了根据实施例的对识别连接的协议的支持。

[0340] 如图26中所示，根据实施例，对于使用第一协议A 372的连接的请求A 370；以及对

于使用第二协议B 382的连接的请求B 380可以由适当的服务器代码处理。

[0341] 图27进一步图示了根据实施例的对识别连接的协议的支持。

[0342] 如图27中所示,根据实施例,对于使用未知的其他协议X 412的连接的请求X 410被确定为不受服务器支持,并且SPI可以确定是否接受该连接请求并以某种方式处置它,或者不接受该连接请求。

[0343] 根据实施例,系统可以使用可扩展机制来识别传入连接的类型,以便正确处置它。SPI可以使用传入连接的初始字节来识别它是什么种类的连接(诸如HTTP/1、具有先验知识的HTTP/2)。这种机制使用第一字节量(可由每个协议处理程序配置)来尝试识别。如果识别失败,那么不接受连接。

[0344] 示例服务提供商接口

[0345] 根据实施例,具有HTTP/1.1连接提供程序(provider)的SPI实施方式使用字节直到第一个CRLF序列,尝试解析它,并且如果它形成有效的HTTP/1.1序言(例如,GET/pathHTTP/1.1),那么它作为HTTP/1.1请求被接受。

[0346] 根据实施例,具有HTTP/2.2连接提供程序的SPI实施方式使用形成由规范定义的连接前言的固定数量的字节。

[0347] 这种方法是可扩展的,使得其他连接提供程序可以插入这个机制,以提供对其他协议的支持。

[0348] 根据实施例,可以提供SPI和API作为服务器连接提供程序(ServerConnectionProvider),并且可以提供各种接口、类或方法,例如:

[0349] 方法int bytesToIdentifyConnection()-需要多少字节(可以为0以识别“未知”)。

[0350] 方法Support supports(BufferData data)-BufferData是连接的初始字节的抽象。Support定义这是否是这个提供程序的有效请求,是否需要更多数据来识别,或者这是否是这个提供程序的无效请求。

[0351] 方法ServerConnection connection(ConnectionContext)-为这个提供程序接受的连接创建服务器连接,上下文提供对读取器和写入器的访问。

[0352] 方法SupportedApplicationProtocols(),当使用TLS时与协议协商一起使用。

[0353] 根据实施例,可以提供用于服务器连接提供程序的java.util.ServiceLoader提供程序接口,其中连接提供程序有机会分析请求字节并决定这是否是它可以接受的连接,例如:

```
public interface ServerConnectionProvider {  
    /**  
     * 需要多少字节来识别这个连接。  
     * @返回所需字节的数量，如果这不是固定值，那么返回 0  
     */  
    int bytesToIdentifyConnection();  
[0354]  
  
    /**  
     * 这个提供程序是否支持当前服务器连接。  
     * 相同的缓冲区将被发送到{ServerConnection#handle()}  
     * @param data 字节 (具有至少{#bytesToIdentifyConnection()}的可用  
     字节)
```

```
* @返回支持响应
*/

Support supports(BufferData data);

/**
 * 这个提供程序支持的应用协议，例如用于 ALPN 协商。
 * @返回支持的协议的集合
 */

Set<String> supportedApplicationProtocols();

/**
 * 创建新连接。
[0355] * 所有方法将从单个虚拟线程被调用。
 * @param ctx 具有对数据写入器、数据读取器和其他有用信息的访问的上下文
 * @返回新的服务器连接
 */

ServerConnection connection(ConnectionContext ctx);

/**
 * 由这个提供程序支持。
 */

enum Support {
    /**
     * 是，这是这个提供程序可以处置的连接。
     */
}
```

```

        SUPPORTED,
        /**
         * 否, 这个连接与这个提供程序不兼容.
         */
        UNSUPPORTED,
[0356]    /**
         * 我们没有足够的字节要决定, 请用更多字节再次询问这个提供程序.
         */
        UNKNOWN
    }
}

```

[0357] 根据实施例, 可以提供用于服务器连接 (ServerConnection) 的SPI, 其可以由 ServerConnectionProvider 创建, 并且具有单个方法 handle(), 该方法预计会阻塞直到连接被关闭, 例如作为服务器连接抽象, 并且可以由任何提供程序用来处置套接字连接, 例如:

```

public interface ServerConnection {
    /**
     * 开始处置连接. 数据是通过
     * {ServerConnectionProvider#connection(webserver.ConnectionCon
[0358] text) 提供的}.
     * @抛出 InterruptedException 以中断任何等待状态并终止这个连接
     */
    void handle() throws InterruptedException;
}

```

[0359] 根据实施例, 可以提供HTTP/1服务器连接提供程序 (Http1ConnectionProvider), 作为例如用于HTTP/1.1服务器连接提供程序的 (java.util.ServiceLoader) 提供程序实施方式:

```
public class Http1ConnectionProvider implements
ServerConnectionProvider {

    private static final String PROTOCOL = " HTTP/1.1\r";

    private final int maxPrologueLength;

    private final int maxHeadersSize;

    private final boolean validateHeaders;

    private final boolean validatePath;

    private final Map<String , Http1UpgradeProvider>
upgradeProviderMap;
```

[0360]

```
    /**
     * 创建具有默认配置的新实例。
     * @已启用，仅供{java.util.ServiceLoader}使用
     */
    @Deprecated
    public Http1ConnectionProvider() {
        this(builder());
    }

    private Http1ConnectionProvider(Builder builder) {
        this.maxPrologueLength = builder.maxPrologueLength;
```

```
        this.maxHeadersSize = builder.maxHeaderSize;

        this.validateHeaders = builder.validateHeaders;

        this.validatePath = builder.validatePath;

        this.upgradeProviderMap = builder.upgradeProviders();
    }

    /**
     * 设置这个提供程序的构建器。
     *
     * @返回新构建器
     */
    public static Builder builder() {
[0361] return new Builder();
    }

    @Override
    public int bytesToIdentifyConnection () {

        // 请求必须以由\r\n终止的行开始，并且我们不知道长度

        return 0;
    }

    @Override
    public Support supports(BufferData request) {

        // 我们寻找第一个\n，如果前面是\r -> 那么尝试 if ours,

        // 否则不支持
    }
```

```
        /**
    > GET /loom/slow HTTP/1.1
    > Host: localhost:8080
    > User-Agent: curl/7.54.0
    > Accept: */*
        */

    int lf = request.indexOf(Bytes.LF_BYTE);
    if(lf == -1){
        // 如果我们已达到最大序言长度，那么我们刚好将其视为 HTTP/1.1,
        // 因此我们可以发送适当的错误。这意味着 maxPrologueLength
        // 应当总是高于任何协议要求，以识别连接（例如，这是回退协议）
        return(request.available() <= maxPrologueLength)?
[0362] Support.SUPPORTED : Support.UNSUPPORTED;
        } else {
        return request.readString(lf).endsWith(PROTOCOL)?
        Support.SUPPORTED : Support.UNSUPPORTED;
        }
    }

    @Override
    public Set<String> supportedApplicationProtocols(){
        return Set.of("http/1.1");
    }

    @Override
    public ServerConnection connection(ConnectionContext ctx){
```

```
        return new Http1Connection(ctx, maxPrologueLength,

        maxHeadersSize      ,      validateHeaders      ,      validatePath      ,

        upgradeProviderMap);

    }

    /**
     * 用于{io.webserver.http1.Http1ConnectionProvider}
     * 的流利 API 构建器.
     */
    public static class Builder implements common.Builder<Builder,
    Http1ConnectionProvider> {

[0363]     private final Helidon ServiceLoader.Builder<Http1UpgradeProvider>
        upgradeProviders =
            Helidon
            ServiceLoader.builder(ServiceLoader.load(Http1UpgradeProvider.class)
            ss)
        );

        private int maxPrologueLength = DEFAULT_MAX_PROLOGUE_LENGTH;

        private int maxHeaderSize = DEFAULT_MAX_HEADERS_SIZE;

        private boolean validateHeaders = DEFAULT_VALIDATE_HEADERS;

        private boolean validatePath = DEFAULT_VALIDATE_PATH;

        private Builder(){
            }
    }
```

```
        @Override
        public Http1ConnectionProvider build(){
            return new Http1ConnectionProvider(this);
        }

        /**
         * 接收到的 HTTP 序言 (GET /path HTTP/1.1) 的最大尺寸.
         * @param maxPrologueLength 按字节的最大尺寸
         * @返回更新后的构建器
         */
        public Builder maxPrologueLength(int maxPrologueLength){
            this.maxPrologueLength = maxPrologueLength;
            return this;
        }

        /**
         * 按字节的接收到的报头的最大尺寸.
         * @param maxHeadersSize 最大报头尺寸
         * @返回更新后的构建器
         */
        public Builder maxHeadersSize(int maxHeadersSize){
            this.maxHeaderSize = maxHeadersSize;
            return this;
        }

        /**
```

[0364]

```

    * 是否验证报头.

    * 如果设置为假, 那么接受任何值, 否则验证报头+已知报头通过格式进行验证

    * (内容长度总是被验证, 因为它是协议处理的一部分 ((如果特征使用其他报头,

    * 那么可以验证其他报头)) .

    *

    * @param validateHeaders 是否验证报头

    * @返回更新后的构建器

    */

public Builder validateHeaders(boolean validateHeaders){

    this.validateHeaders = validateHeaders;

return this;

}

[0365]

/**

    * 如果设置为假, 那么接受任何路径 (即使包含非法字符) .

    * @param validatePath 是否验证路径

    * @返回更新后的构建器

    */

public Builder validatePath (boolean validatePath) {

    this.validatePath = validatePath;

return this;

}

/**

    * 添加配置的升级提供程序. 这将代替通过服务加载器发现的实例 (如果存在)
```

```
        * @param provider 添加提供程序

        * @返回更新后的构建器

        */
    public Builder addUpgradeProvider (Http1UpgradeProvider provider) {
[0366]    upgradeProviders.addService (provider);

    return this;

    }

}

}
```

[0367] 根据实施例,可以提供HTTP/2服务器连接提供程序 (Http2ConnectionProvider), 以支持“先前知识”请求,例如:

```
/**

    * 用于 HTTP/2 服务器连接提供程序的 {java.util.ServiceLoader}

    * 提供程序实施方式

    */
public class Http2ConnectionProvider implements
ServerConnectionProvider {

[0368]    @Override

    public int bytesToIdentifyConnection() {

    return PREFACE_LENGTH;

    }

    @Override

    public Support supports (BufferData request) {
```

```
byte[] prefaceBytes = new byte[PREFACE_LENGTH];  
  
request.read(prefaceBytes, 0, PREFACE_LENGTH);
```

// 现在我们可以请协议处理程序识别这个协议

```
if (isPreface(prefaceBytes)) {  
    // 这是 HTTP/2 先验知识  
  
    return Support.SUPPORTED;  
}
```

```
return Support.UNSUPPORTED;  
}
```

[0369]

```
    @Override  
    public Set<String> supportedApplicationProtocols() {  
        return Set.of("h2");  
    }  
  
    @Override  
    public ServerConnection connection(ConnectionContext ctx) {  
        Http2Connection result = new Http2Connection(ctx);  
  
        result.expectPreface();  
  
        return result;  
    }  
}
```

[0370] 上述示例是为说明目的提供的;根据各种实施例,可以提供接口、类或方法的其他示例。

[0371] 图28图示了根据实施例的与微服务环境一起使用的、用于识别连接的协议的方

法。

[0372] 如图28中所示,根据实施例,在步骤420处,在包括一个或多个处理器及存储器的计算机处,提供微服务环境(例如,如由微服务库提供的),该环境使得客户端和服务端应用能够在微服务环境内通信。

[0373] 在步骤422处,系统提供连接协议识别组件或功能性与例如HTTP请求路由一起使用,包括可扩展以支持服务器代码支持的多种协议的连接提供程序服务提供商接口(SPI)。

[0374] 在步骤424处,连接提供程序SPI提供多个连接提供程序类型,每个类型与一种支持的协议相关联。

[0375] 在步骤426处,在接收到经由传入连接指向服务器的请求后,系统可以使用传入请求的初始字节来识别连接类型(例如,HTTP/1,HTTP/2),并处理连接请求,否则拒绝不支持的连接请求。

[0376] 示例接口

[0377] 根据实施例,下面为了说明目的提供用于识别连接的协议的示例接口。根据其他实施例,可以提供其他类型的接口。

[0378] 接口ServerConnectionProvider

[0379] 根据实施例,ServerConnectionProvider接口可以被用于为服务器连接提供程序提供接口,其中连接提供程序有机会分析请求字节并决定这是否是它可以接受的连接,并且可以包括各种方法,例如:

[0380] 方法int bytesToIdentifyConnection()。需要多少字节来识别这个连接。在实施例中,该方法返回:所需的字节数,如果这不是固定值,那么返回0。

[0381] 方法

[0382] ServerConnectionProvider.Supports supports(common.buffer.BufferData data)。这个提供程序是否支持当前服务器连接。相同的缓冲区将被发送到ServerConnection.handle()。可以提供的参数包括例如:数据-字节(可用字节至少为bytesToIdentifyConnection())。在实施例中,该方法返回:支持响应。

[0383] 方法Set<String>supportedApplicationProtocols()。这个提供程序支持的应用协议,例如用于ALPN协商。在实施例中,该方法返回:支持的协议的集合。

[0384] 方法ServerConnection connection(ConnectionContext ctx)。创建新连接。所有方法都将从单个虚拟线程调用。可以提供的参数包括例如:ctx-可访问数据写入器、数据读取器和其他有用信息的连接上下文。在实施例中,该方法返回:新的服务器连接。

[0385] 根据各种实施例,本公开的方面在以下编号条款中阐述:

[0386] 1.一种用于与微服务或其他计算环境一起使用的系统,包括:包括一个或多个处理器的计算机,该计算机提供对微服务或其他计算环境的访问,以便与软件应用或服务一起使用;并且其中该系统经由报头枚举提供允许将消息报头视为对象的抽象,其中报头枚举可经由支持多种通信协议并允许客户端和服务端使用几种支持的协议中的任何一种协议来传达请求和响应消息的应用程序接口访问。

[0387] 2.如条款1所述的系统,其中当遇到具有已知报头类型的请求消息时,系统可以从枚举中获得如果可用的加索引的值,并将相关联的数据直接提供给请求所指向的进程和/或高速缓存报头值以供以后使用。

[0388] 3.一种用于与微服务或其他计算环境一起使用的方法,包括:提供用于与软件应用或服务一起使用的微服务或其他计算环境;以及借助于报头枚举提供抽象,该抽象允许将消息报头视为对象,可经由支持多种通信协议并允许客户端和服务端使用几种支持的协议中的任何一种协议来传达请求/响应消息的应用程序接口访问。

[0389] 4.如条款3所述的方法,其中当遇到具有已知报头类型的请求消息时,系统可以从枚举中获得如果可用的加索引的值,并将相关联的数据直接提供给请求所指向的进程和/或高速缓存报头值以供以后使用。

[0390] 5.一种用于与微服务或其他计算环境一起使用的系统,包括:包括一个或多个处理器的计算机,该计算机提供对微服务或其他计算环境的访问,以便与软件应用或服务一起使用;并且其中在软件应用或服务的测试期间,客户端连接被替换为存储器内实施方式,该存储器内实施方式模拟网络通信协议的使用并向服务器连接暴露反函数,包括:写入客户端连接的数据写入器的数据可通过服务器连接上的数据读取器获得,并且写入服务器连接的数据写入器的数据可通过客户端连接上的数据读取器获得。

[0391] 6.如条款5所述的系统,其中存储器内实施方式使用存储器中的阻塞队列,该阻塞队列模拟网络通信协议的使用而无需使用实际套接字,其中测试已经完成,客户端和/或服务端代码可以对照实际套接字实现。

[0392] 7.一种用于与微服务或其他计算环境一起使用的方法,包括:提供用于与软件应用或服务一起使用的微服务或其他计算环境;以及在软件应用或服务的测试期间,用模拟网络通信协议的使用并向服务器连接暴露反函数的存储器内实施方式替换客户端连接,包括:写入客户端连接的数据写入器的数据可通过服务器连接上的数据读取器获得,并且写入服务器连接的数据写入器的数据可通过客户端连接上的数据读取器获得。

[0393] 8.如条款7所述的方法,其中存储器内实施方式使用存储器中的阻塞队列,该阻塞队列模拟网络通信协议的使用而无需使用实际套接字,其中测试已经完成,客户端和/或服务端代码可以对照实际套接字实现。

[0394] 9.一种用于与微服务或其他计算环境一起使用的系统,包括:包括一个或多个处理器的计算机,该计算机提供对微服务或其他计算环境的访问,以便与软件应用或服务一起使用;并且其中该系统使得能够配置服务器,该服务器可以操作以经由多种网络或通信协议来服务于请求,包括:提供将服务器支持的每种网络或通信协议与其自己的路由相关联的路由抽象,并且路由抽象可以包括一个或多个特定路由以用于与特定网络或通信协议一起使用。

[0395] 10.如条款9所述的系统,其中对于经由特定协议接收的请求消息,该系统可以为该请求选择适当的路由并根据路由抽象对其进行处理。

[0396] 11.一种用于与微服务或其他计算环境一起使用的方法,包括:提供用于与软件应用或服务一起使用的微服务或其他计算环境;并且其中系统使得能够配置服务器,该服务器可以操作以经由多种网络或通信协议来服务于请求,包括:提供将服务器支持的每种网络或通信协议与其自己的路由相关联的路由抽象,并且路由抽象可以包括一个或多个特定路由以用于与特定网络或通信协议一起使用。

[0397] 12.如条款11所述的方法,其中对于经由特定协议接收的请求消息,系统可以为该请求选择适当的路由并根据路由抽象对其进行处理。

[0398] 13.一种用于与微服务或其他计算环境一起使用的系统,包括:包括一个或多个处理器的计算机,该计算机提供对微服务或其他计算环境的访问,以便与软件应用或服务一起使用;以及可扩展服务提供商接口,该可扩展服务提供商接口提供允许识别与请求消息和客户端-服务器连接相关联的网络通信协议的软件方法;其中网络通信协议的识别被用于提供客户端-服务器连接抽象、路由器抽象或者对单个套接字上使用多种网络通信协议的支持中的一个或多个。

[0399] 14.如条款13所述的系统,其中当从客户端接收到连接请求消息并将其指向服务器时,做出关于是否可以使用提供的网络通信协议来处置该请求的确定,其中协议识别处理返回系统不支持请求网络通信协议、或者其目前没有足够的数据来做出决定或者需要附加数据来做出这种确定的指示。

[0400] 15.一种用于与微服务或其他计算环境一起使用的方法,包括:提供用于与软件应用或服务一起使用的微服务或其他计算环境;以及提供可扩展服务提供商接口,该可扩展服务提供商接口提供允许识别与请求消息和客户端-服务器连接相关联的网络通信协议的软件方法;其中网络通信协议的识别被用于提供客户端-服务器连接抽象、路由器抽象或者对单个套接字上使用多种网络通信协议的支持中的一个或多个。

[0401] 16.如条款15所述的方法,其中当从客户端接收到连接请求消息并将其指向服务器时,做出关于是否可以使用提供的网络通信协议来处置该请求的确定,其中协议识别处理返回系统不支持请求网络通信协议、或者其目前没有足够的数据来做出决定或者需要附加数据来做出这种确定的指示。

[0402] 17.如前述编号条款中的任一项所述的系统或方法,其中系统或方法作为web服务器的一部分提供或者与web服务器一起使用,并且其中请求是HTTP请求。

[0403] 18.如前述编号条款中的任一项所述的系统或方法,其中系统或方法包括使用虚拟线程以及可用于构建云原生服务的相关库和特征的请求处理组件。

[0404] 19.如前述编号条款中的任一项所述的系统或方法,其中系统或方法在提供对一个或多个云、数据库或其他系统或服务的访问的云计算环境内提供。

[0405] 根据各种实施例,可以使用一个或多个计算机、计算设备、机器或微处理器(包括一个或多个处理器、存储器和/或根据本文的教导编程的计算机可读存储介质)实现本文的教导。如对软件领域的技术人员将显而易见的,熟练的程序员可以基于本公开的教导容易地准备适当的软件编码。

[0406] 在一些实施例中,本文的教导可以包括计算机程序产品,该计算机程序产品是其上/其中存储有指令的(一个或多个)非暂态计算机可读存储介质,该指令可以用于对计算机进行编程以执行本教导的任何处理。这种存储介质的示例可以包括但不限于硬盘驱动器、硬盘、硬驱动器、固定盘、ROM、RAM、EPROM、EEPROM、DRAM、VRAM、闪存设备或其他类型的存储介质或适用于指令和/或数据的非临时存储的设备。

[0407] 前述描述是为了说明和描述的目的而提供的。其并非旨在详尽无遗或将保护范围限制为所公开的精确形式。另外,对于本领域技术人员来说,修改和变化将是显而易见的。

[0408] 例如,虽然本文描述的系统和方法的各种实施例说明了在Helidon微服务环境中的使用,但各种实施例可以与其他类型的微服务环境或其他计算环境一起使用。

[0409] 选择并描述实施例是为了最好地解释本文的教导的原理及其实际应用,从而使本

领域的其他技术人员能够理解各种实施例以及适合预期特定用途的各种修改。范围旨在由以下权利要求及其等同形式限定。

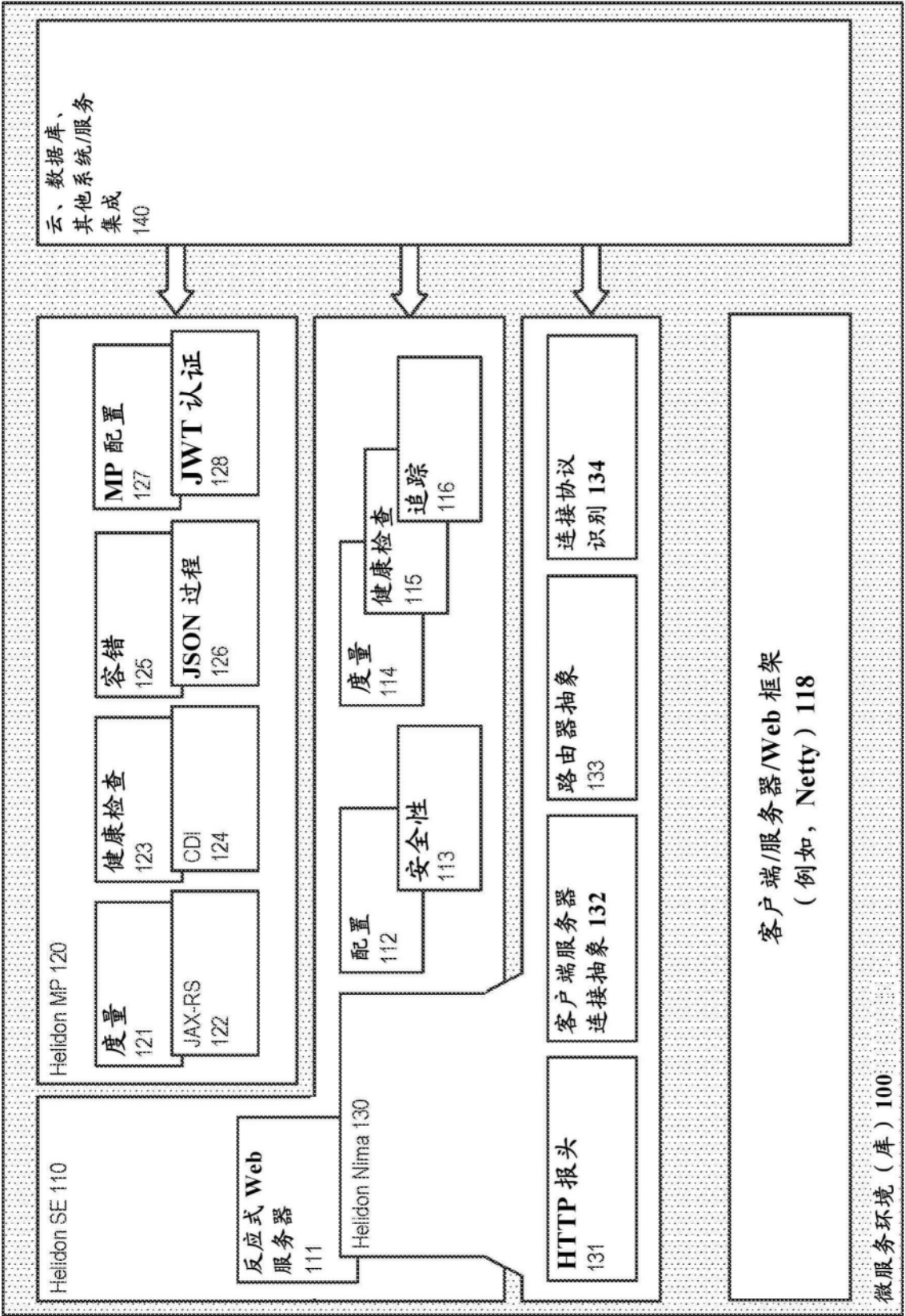


图1

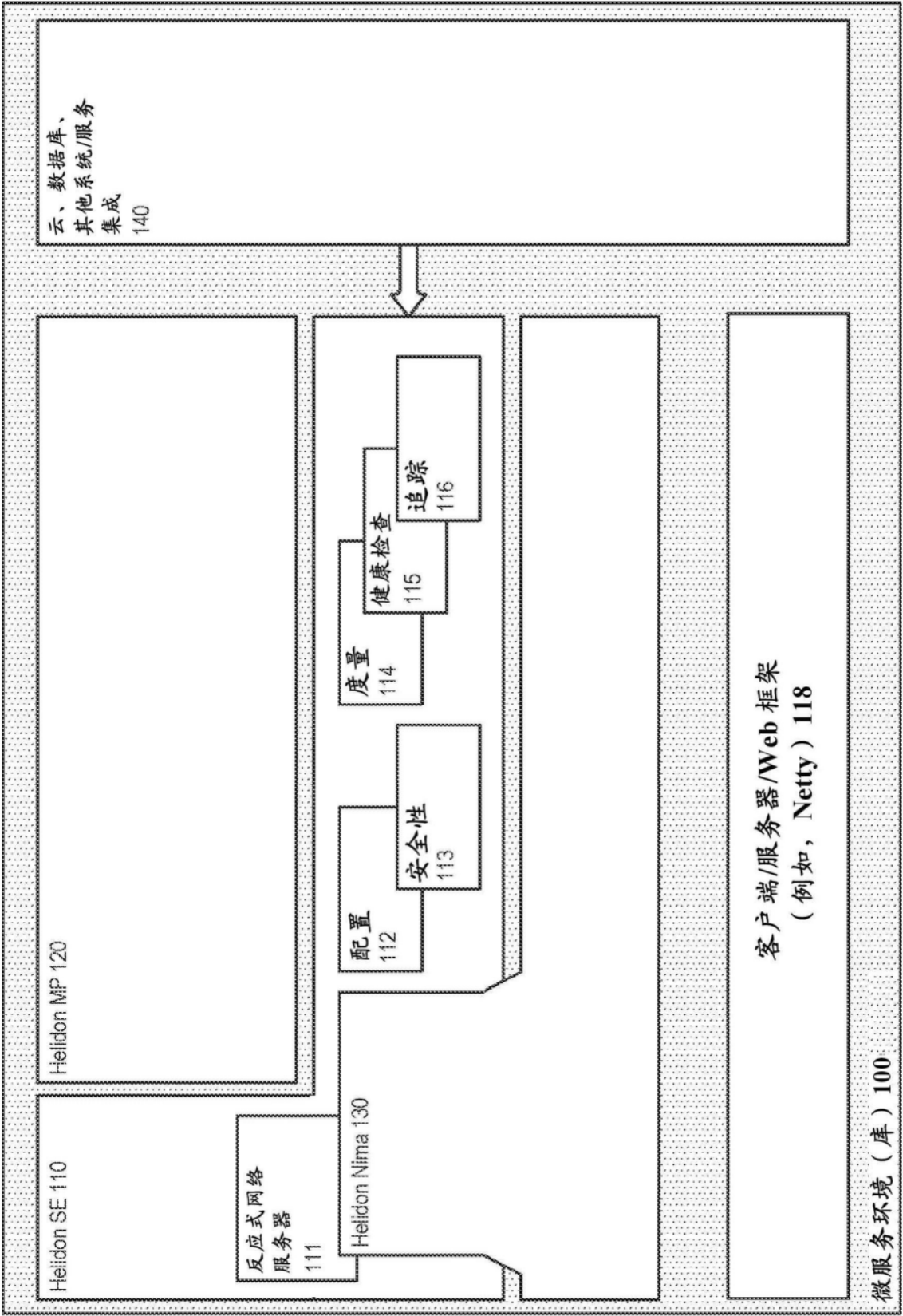


图2

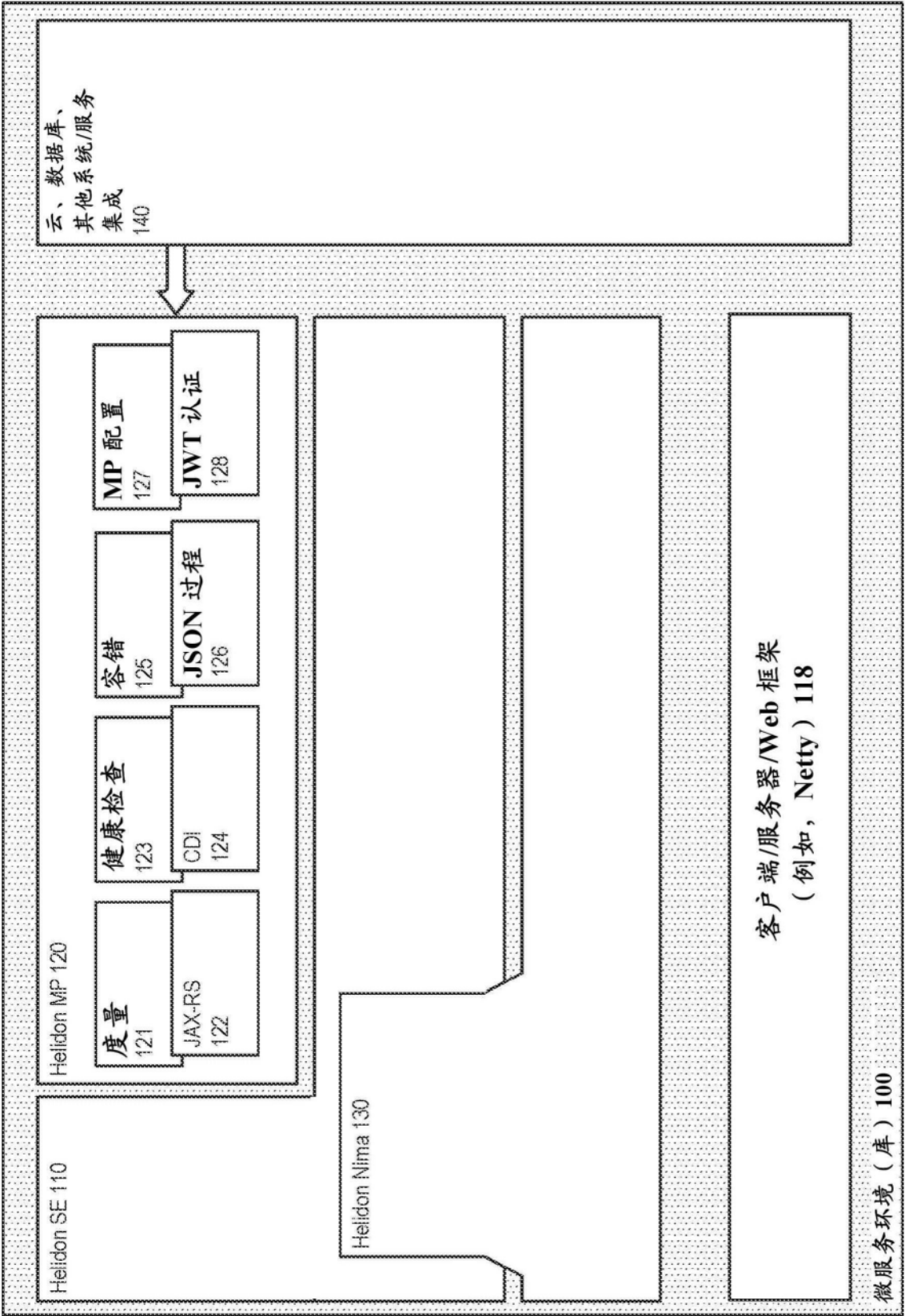


图3

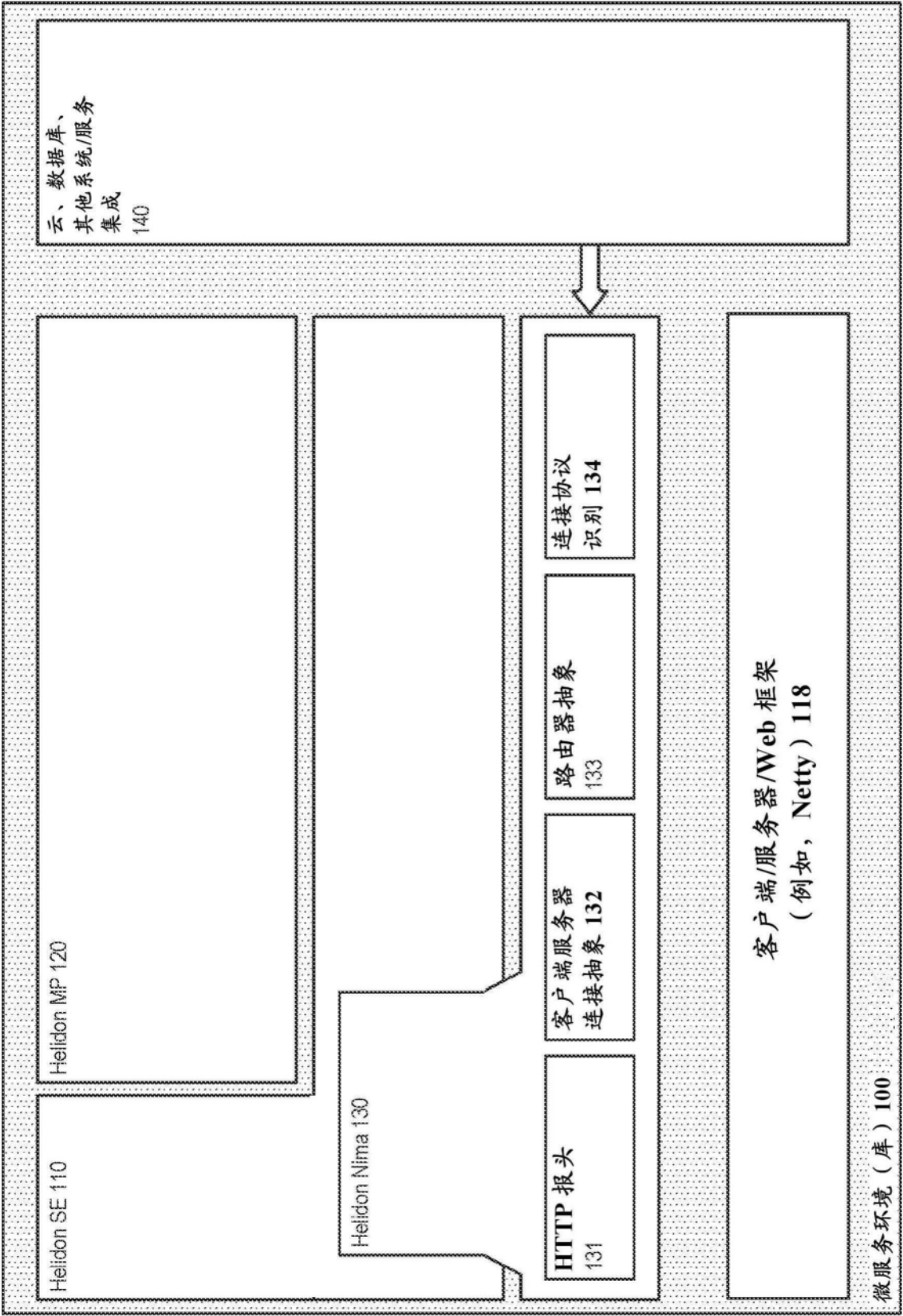


图4

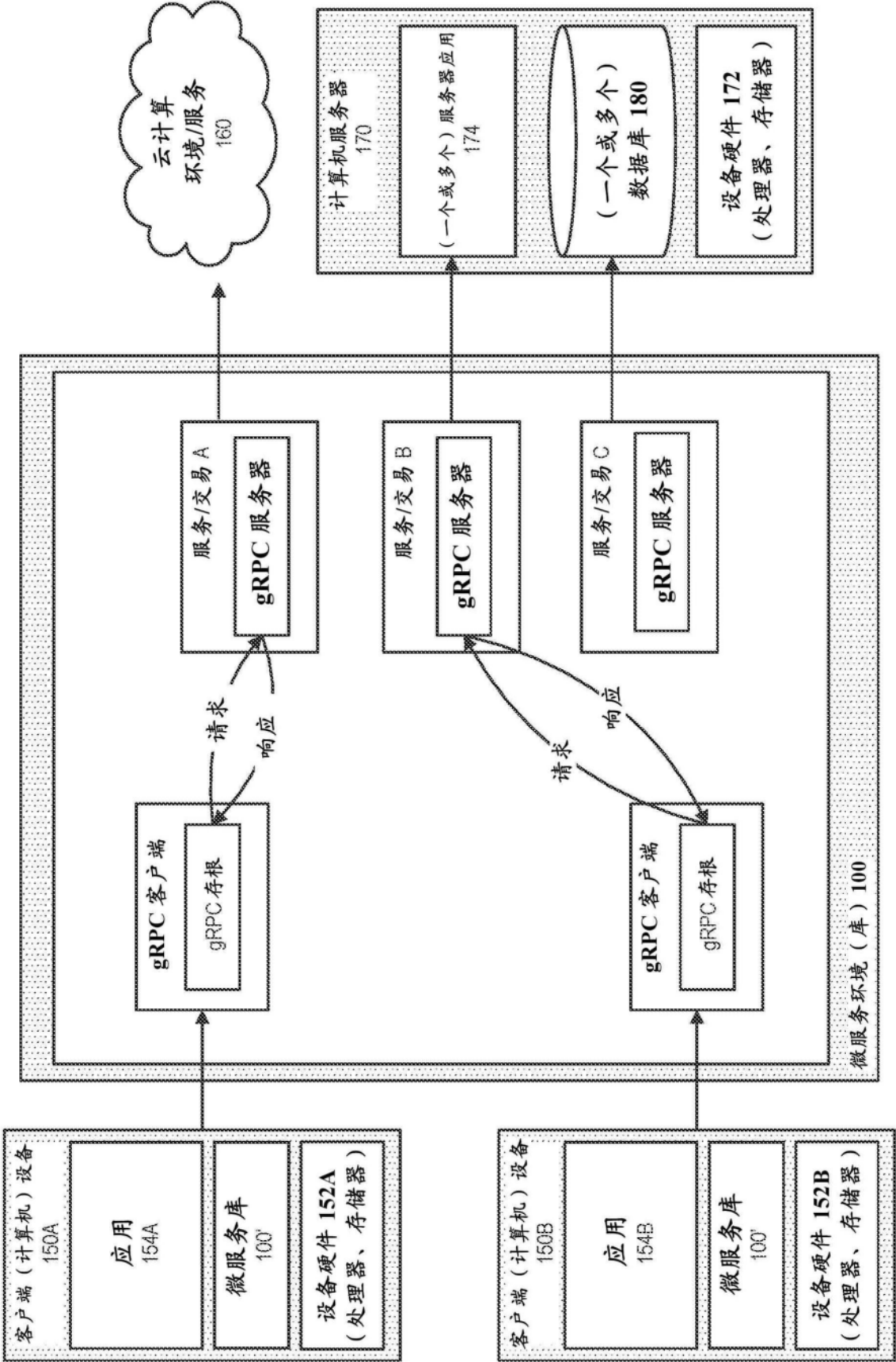


图5

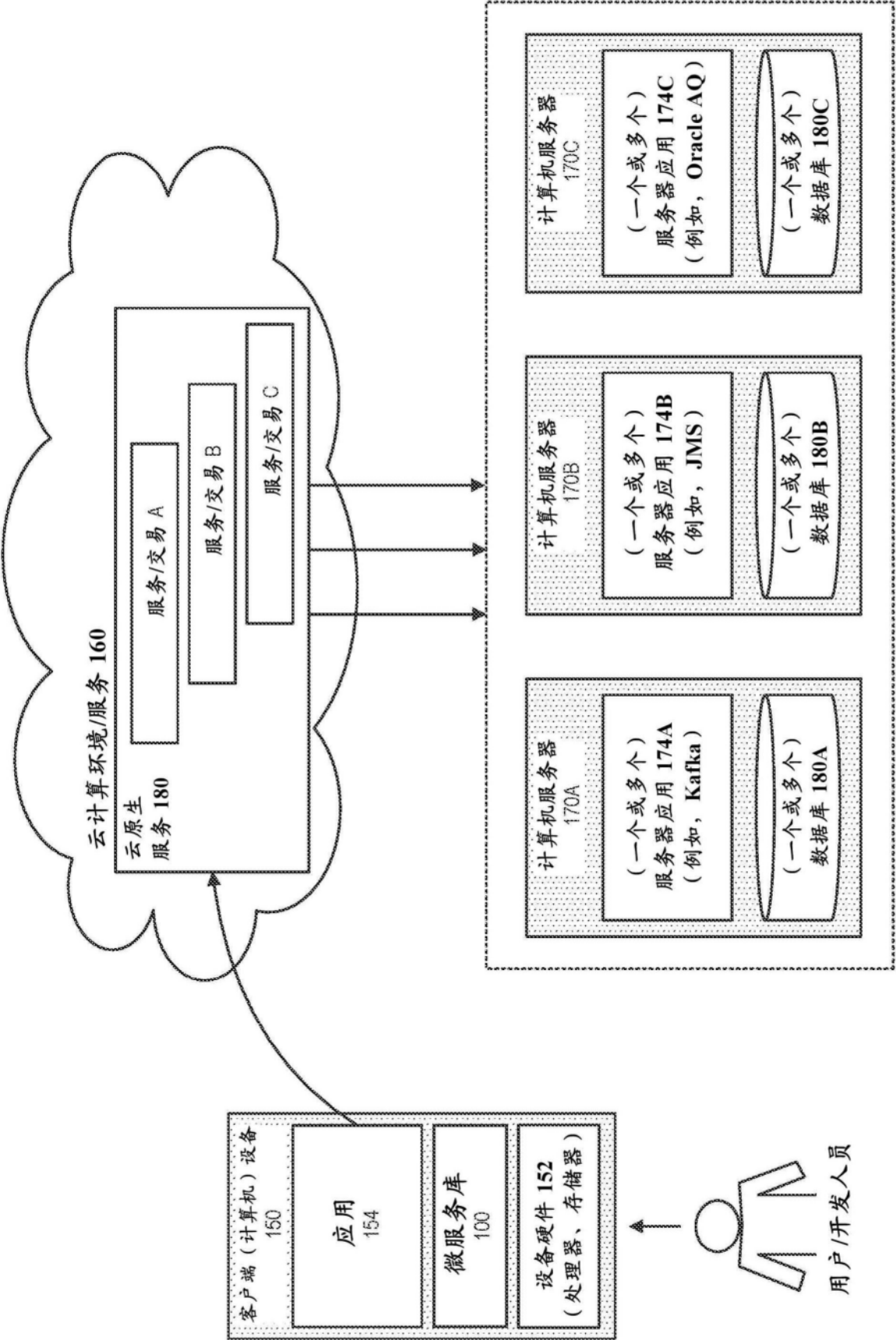


图6

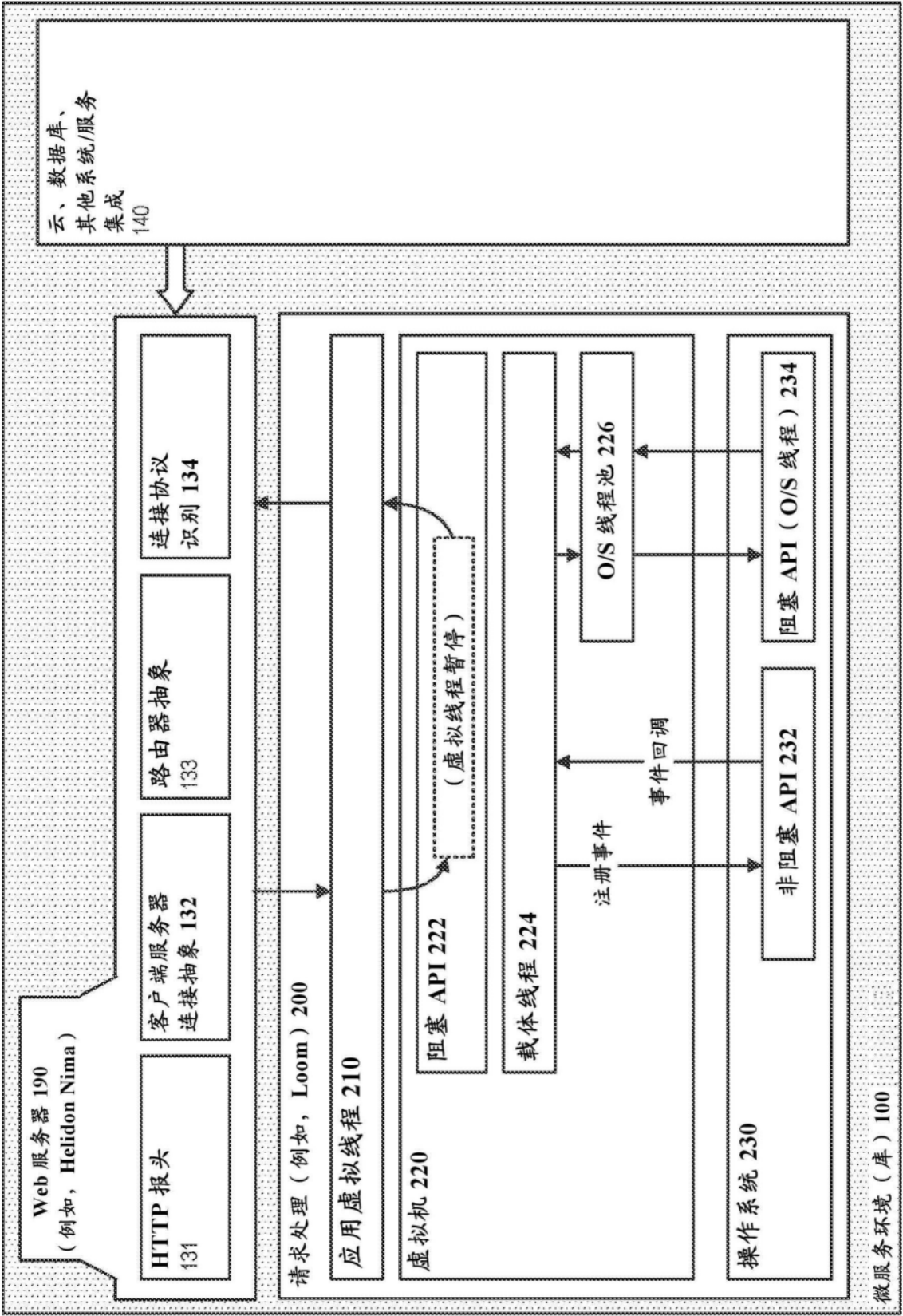


图7

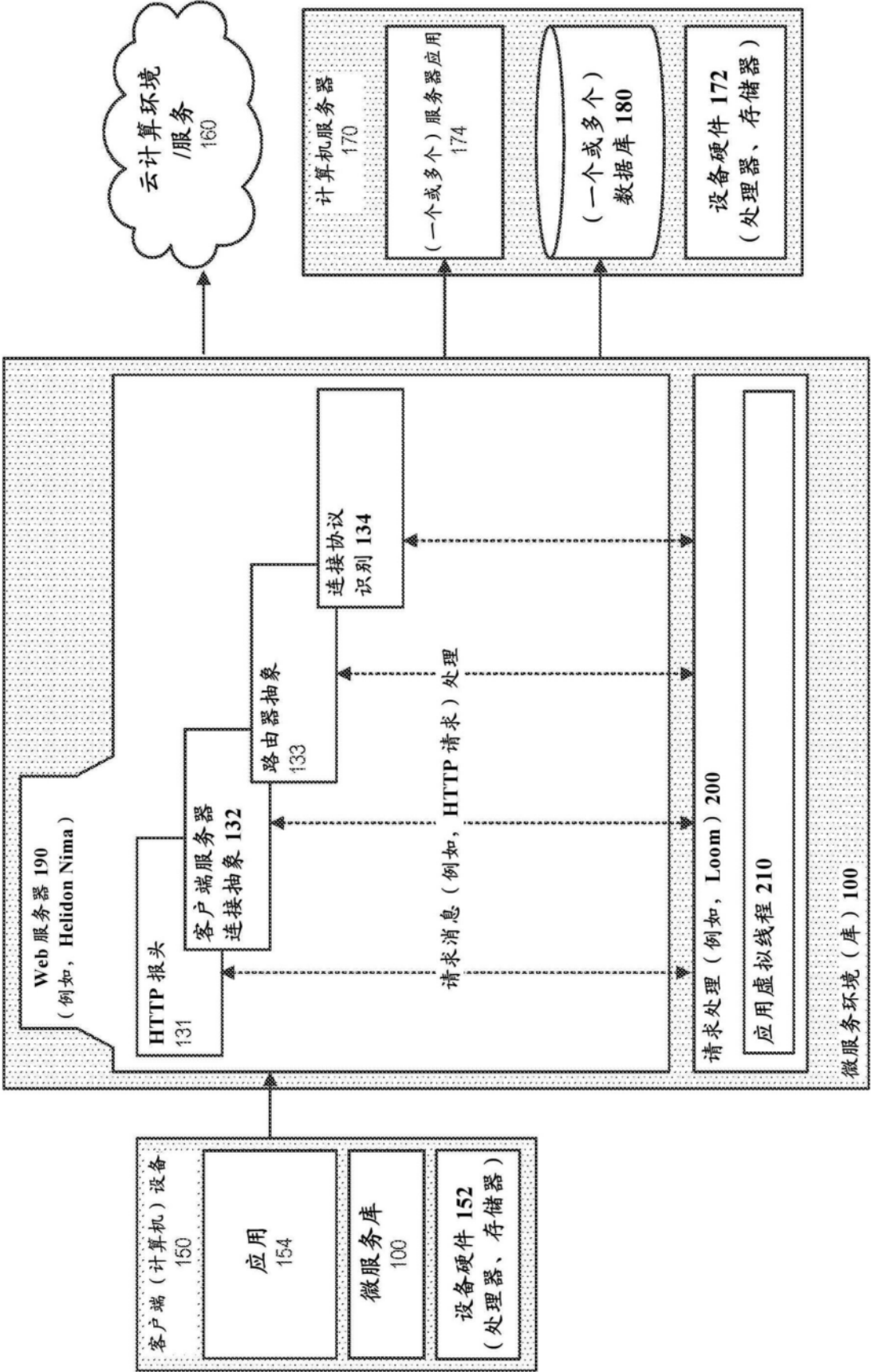


图8

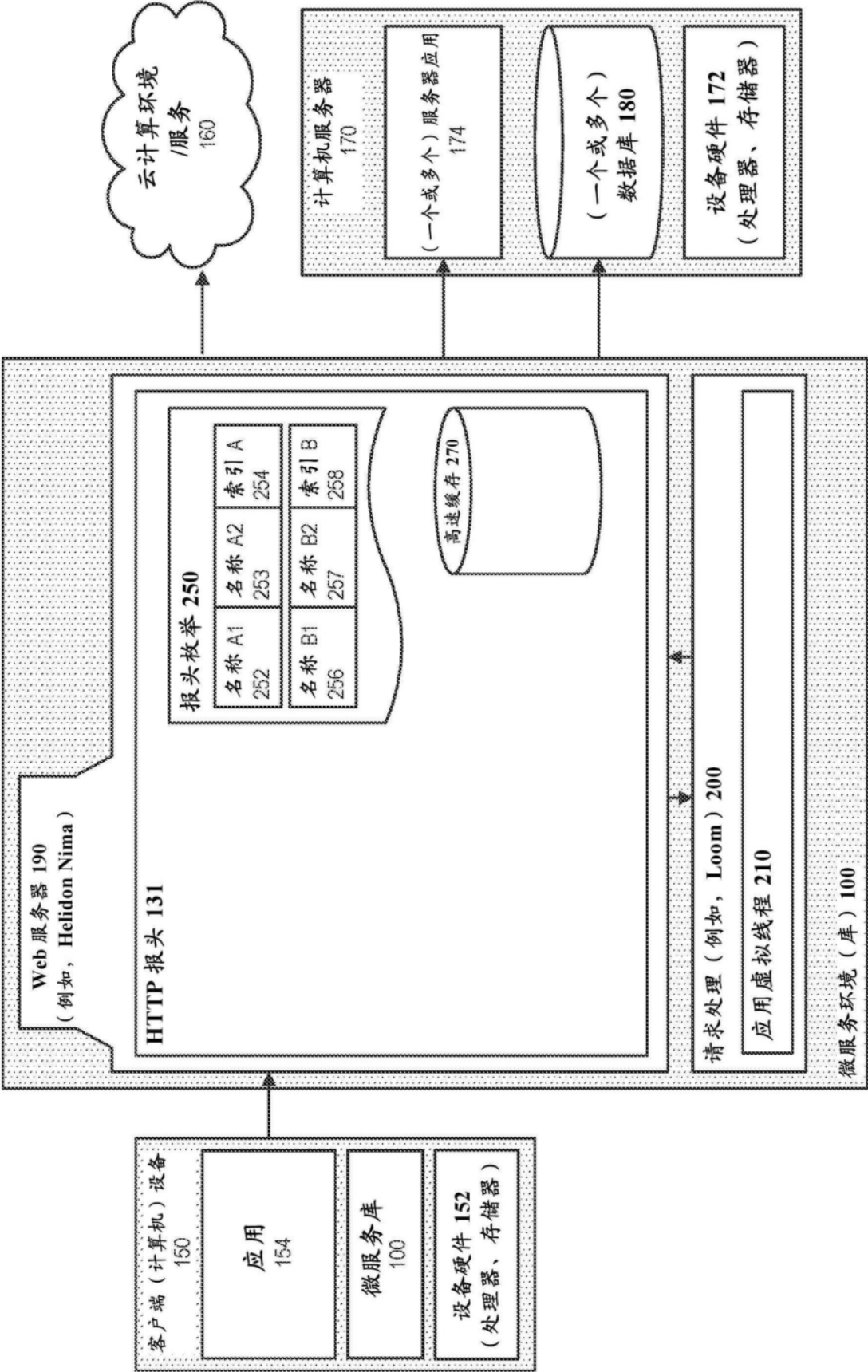


图9

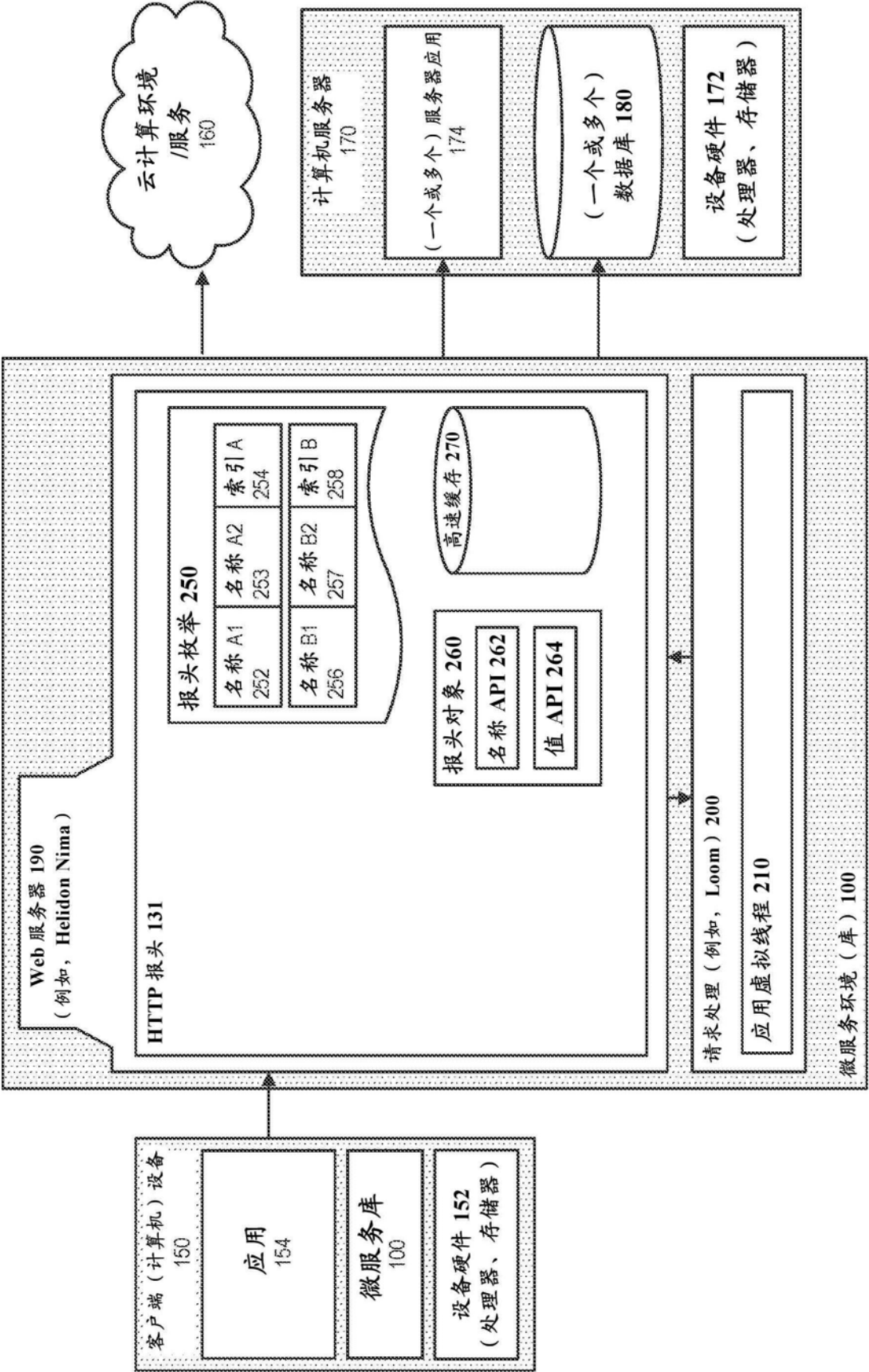


图10

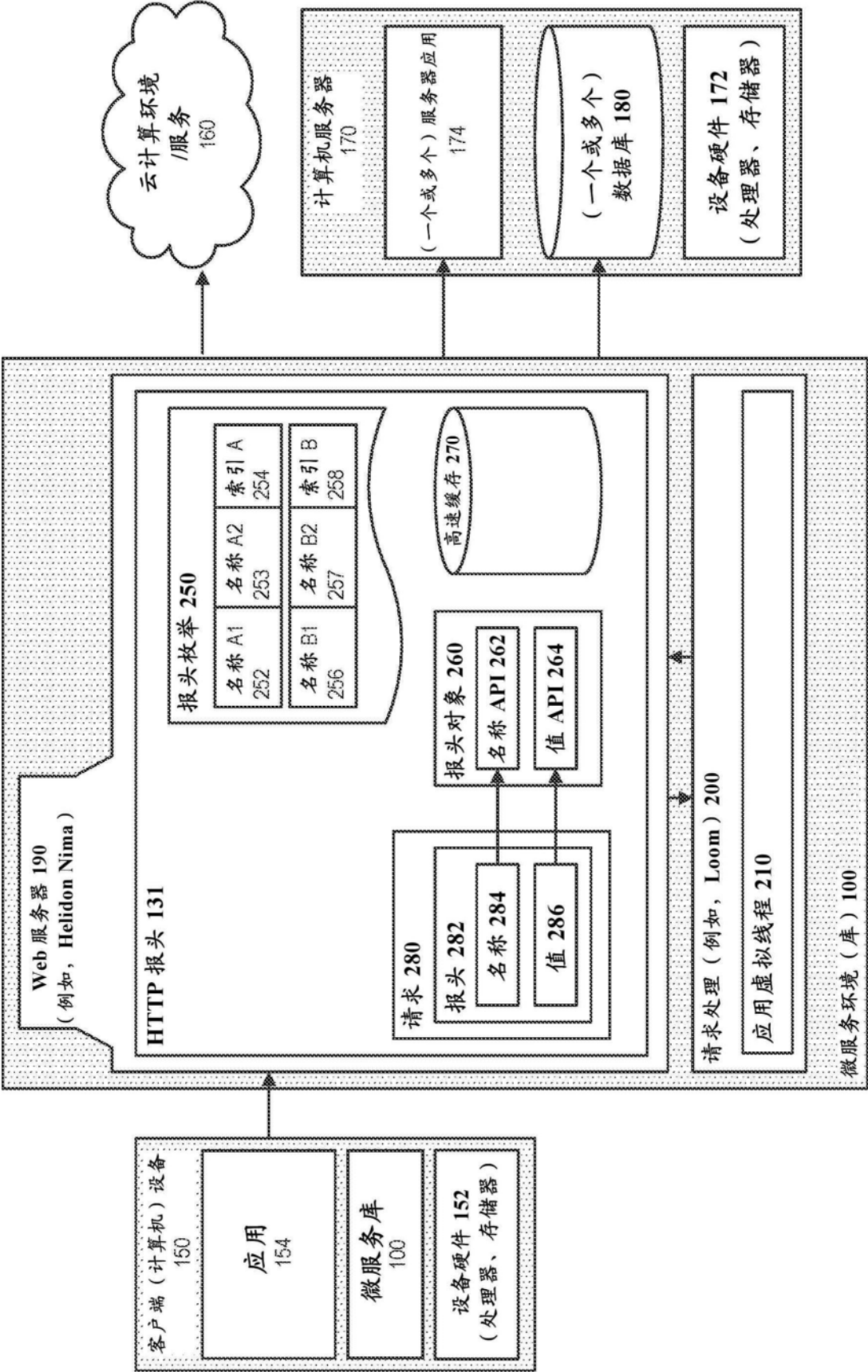


图11

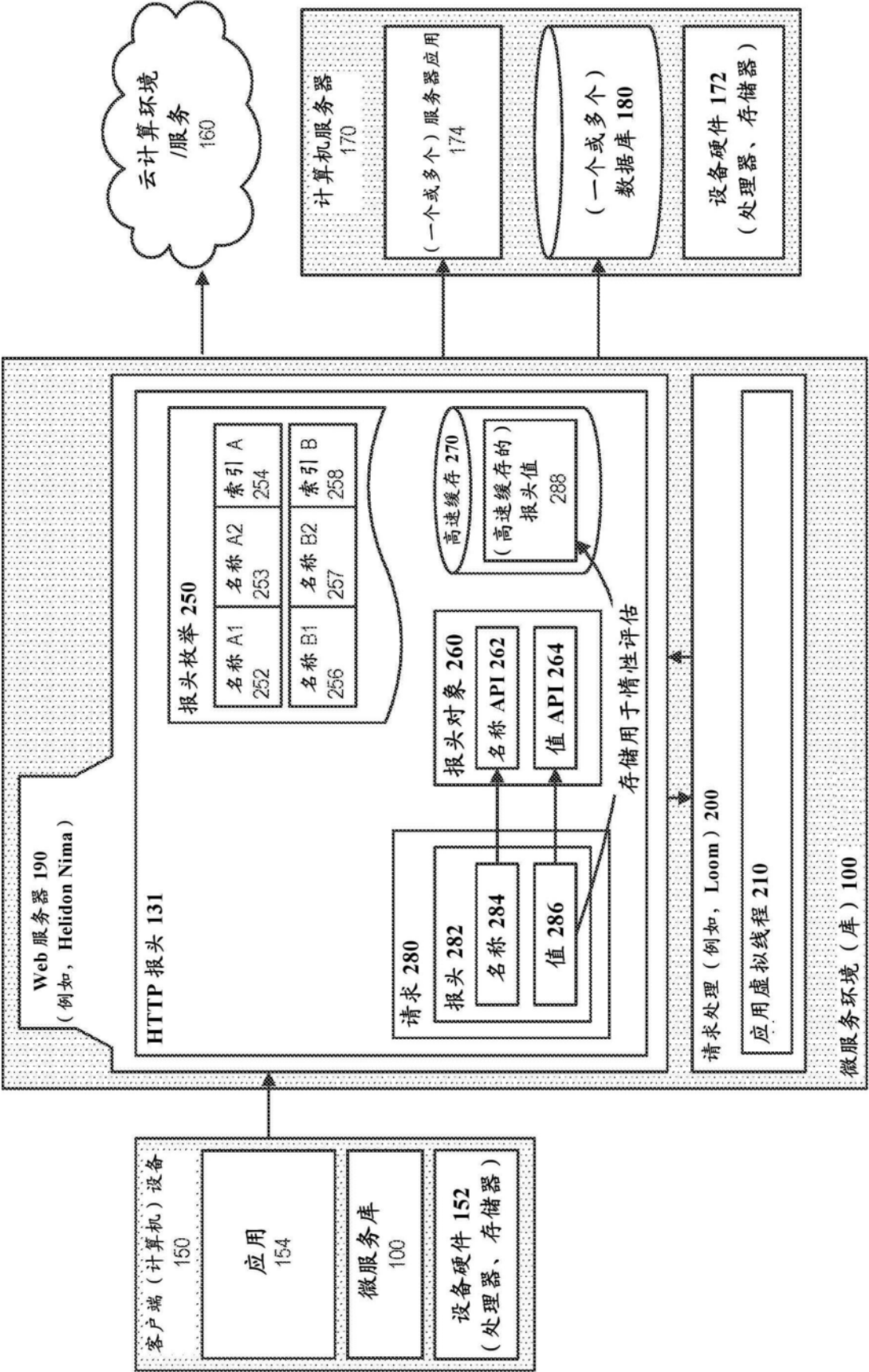


图12

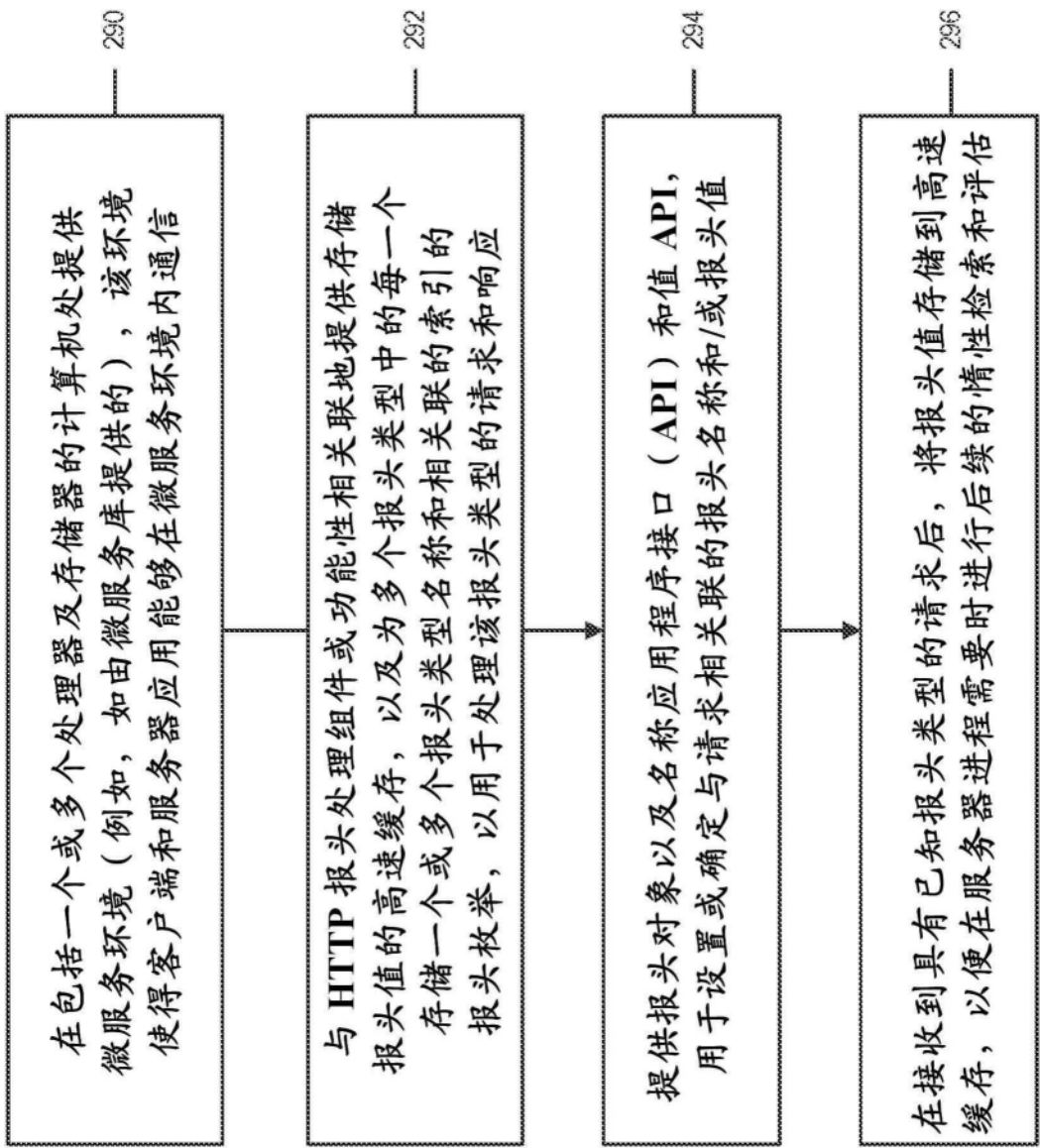


图13

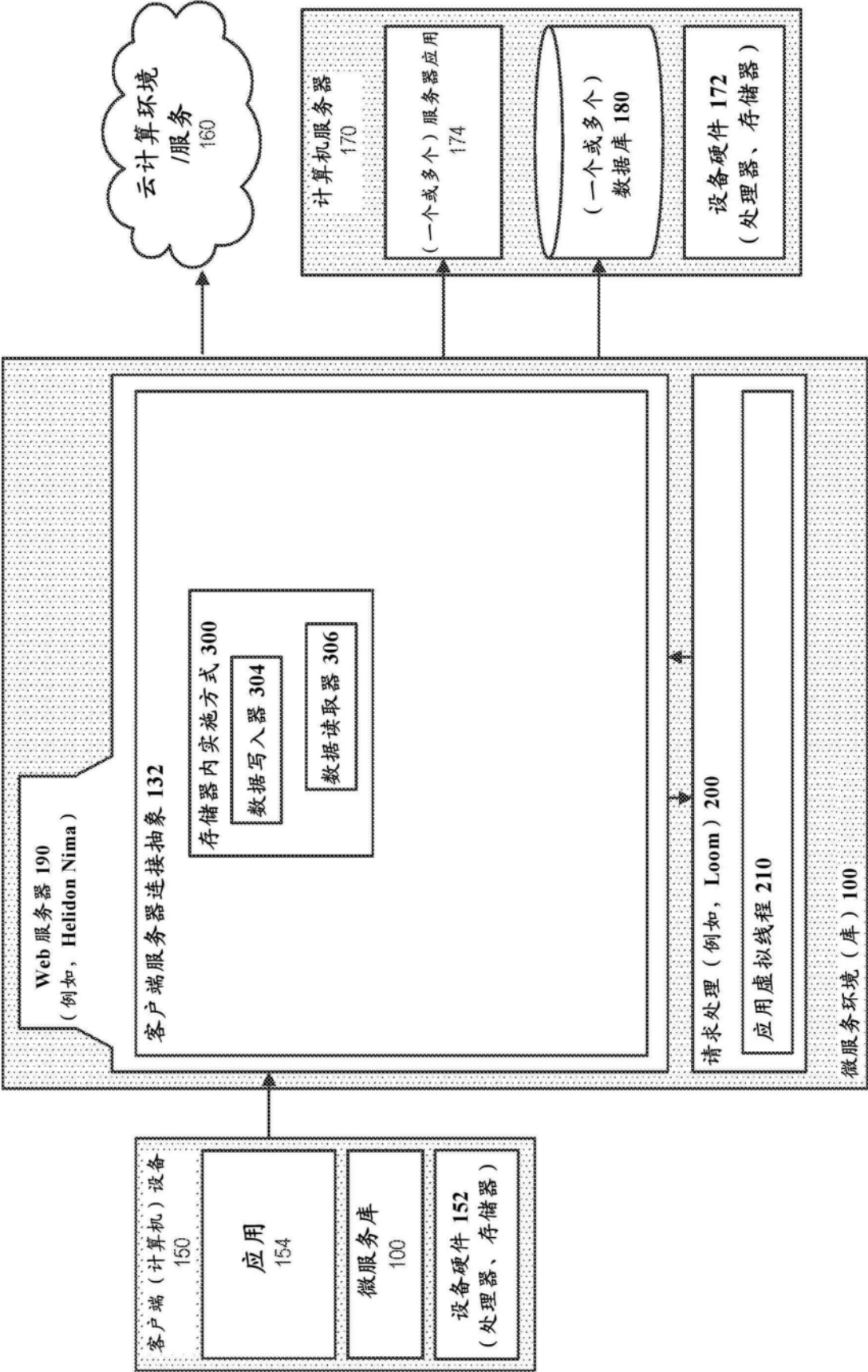


图14

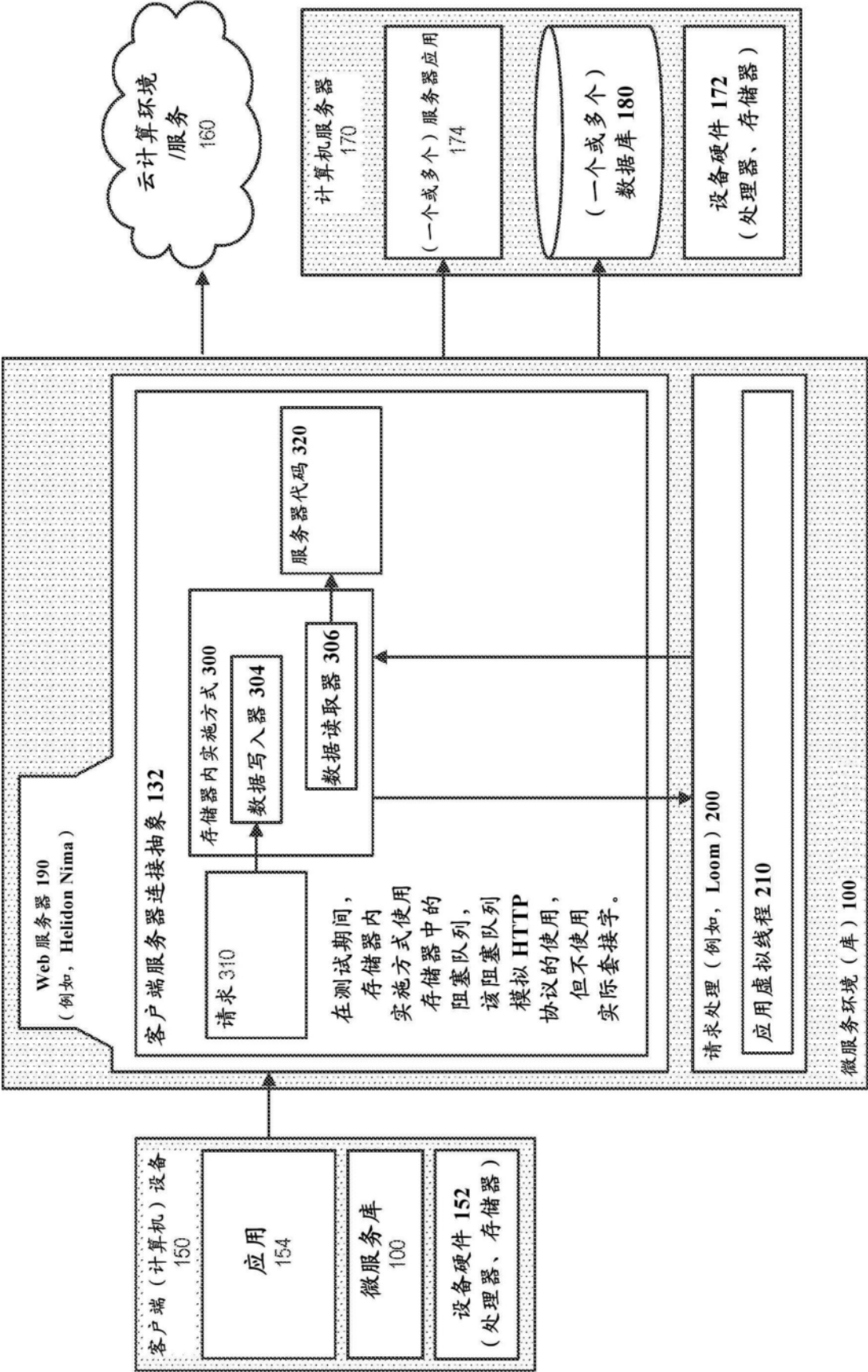


图15

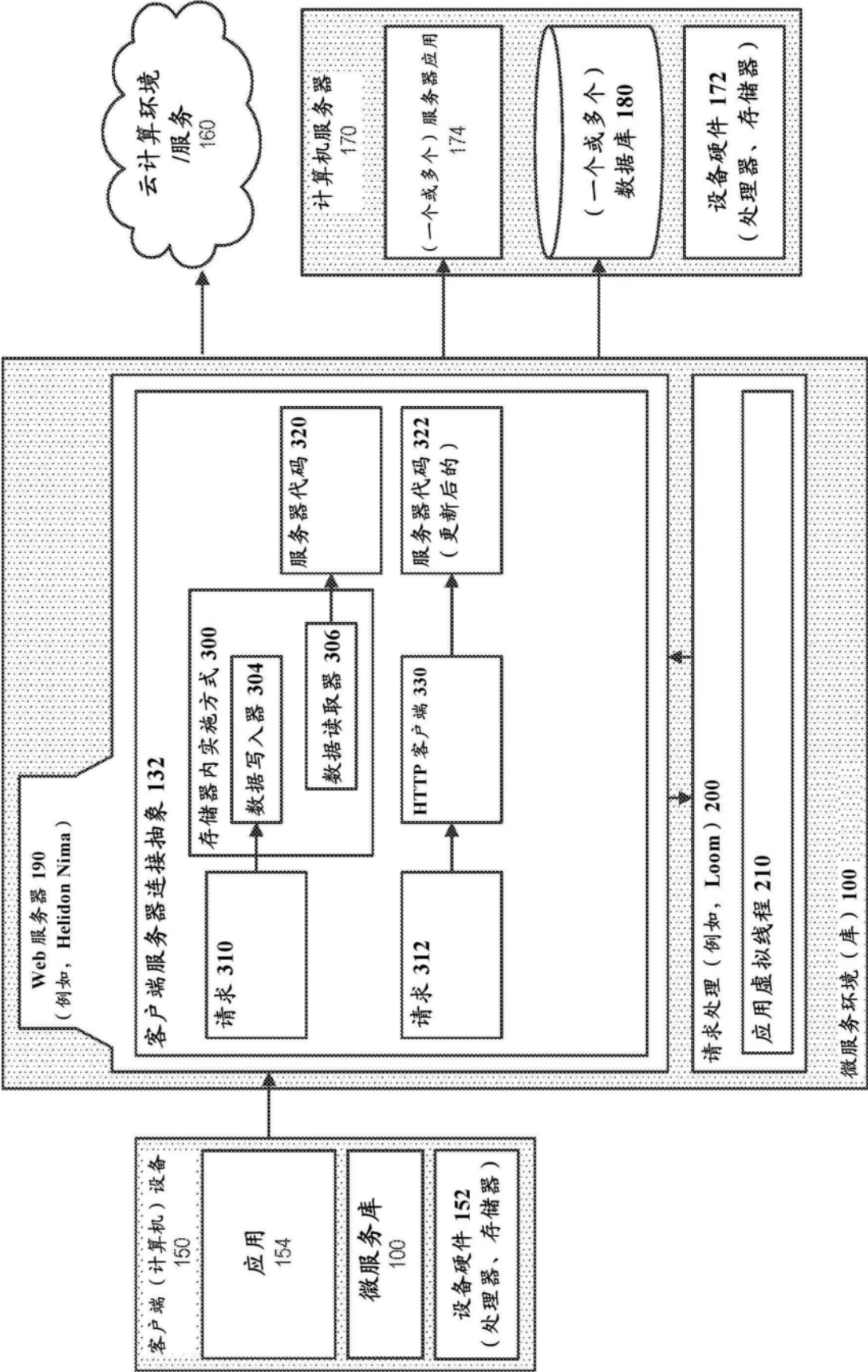


图16

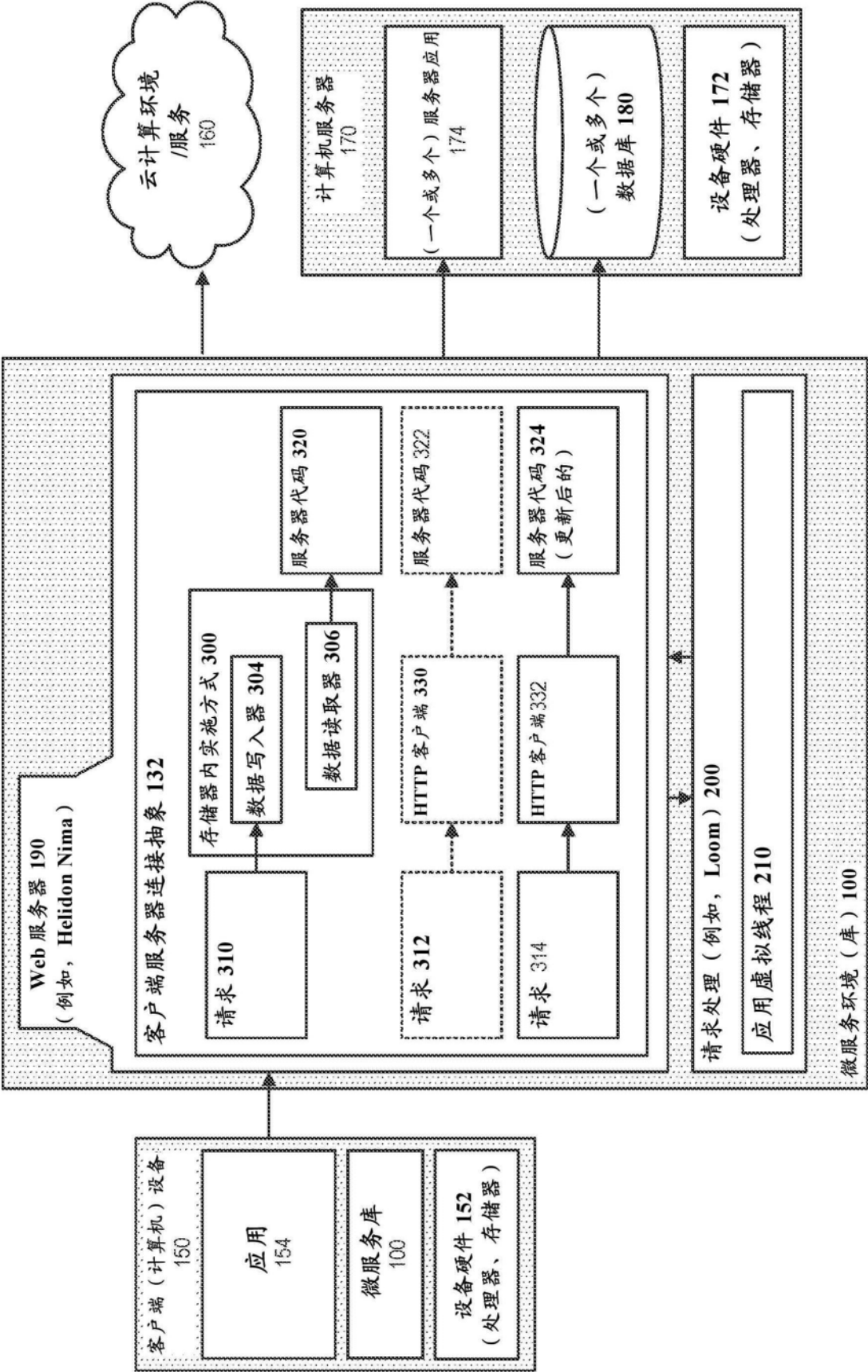


图17

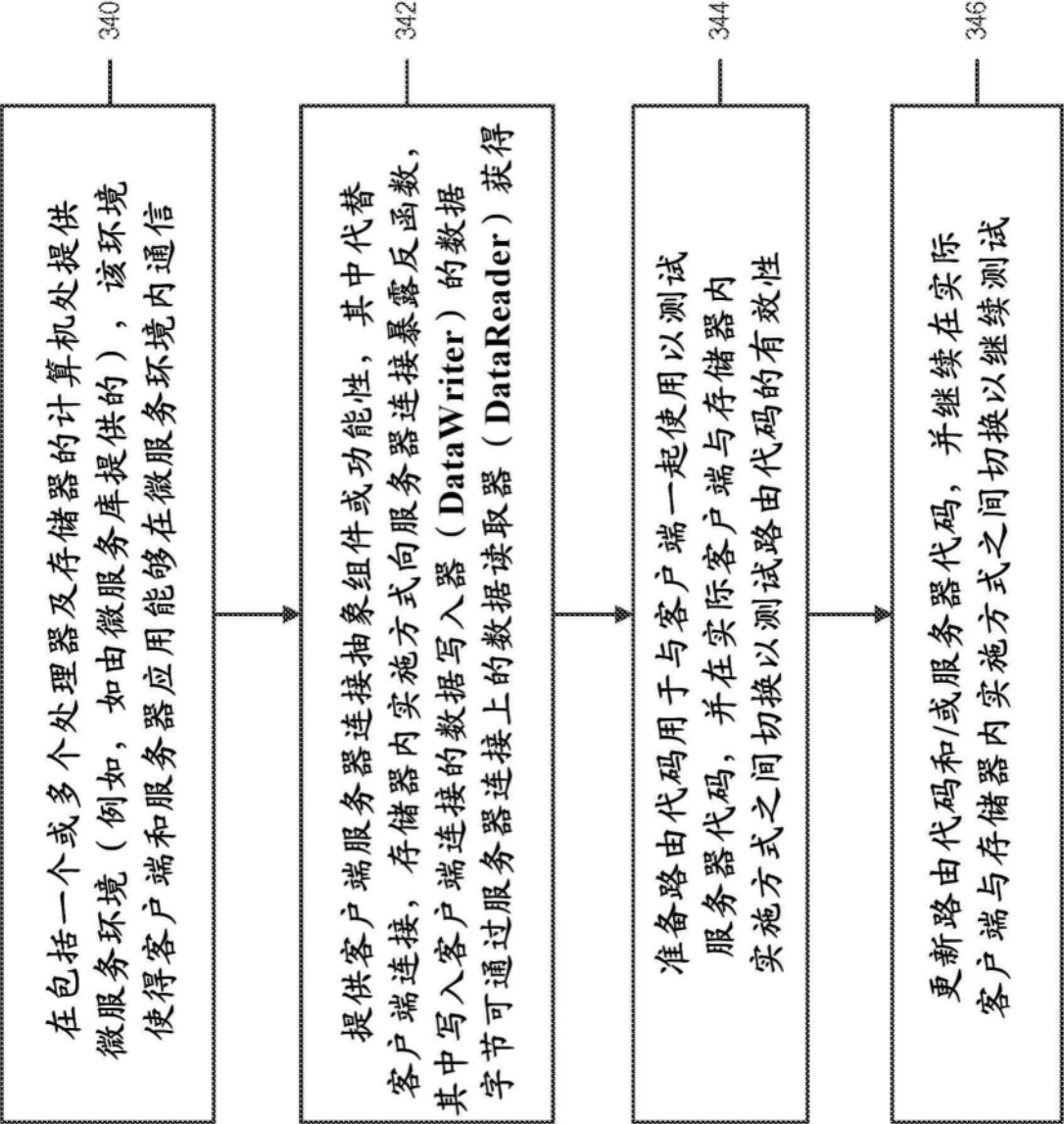


图18

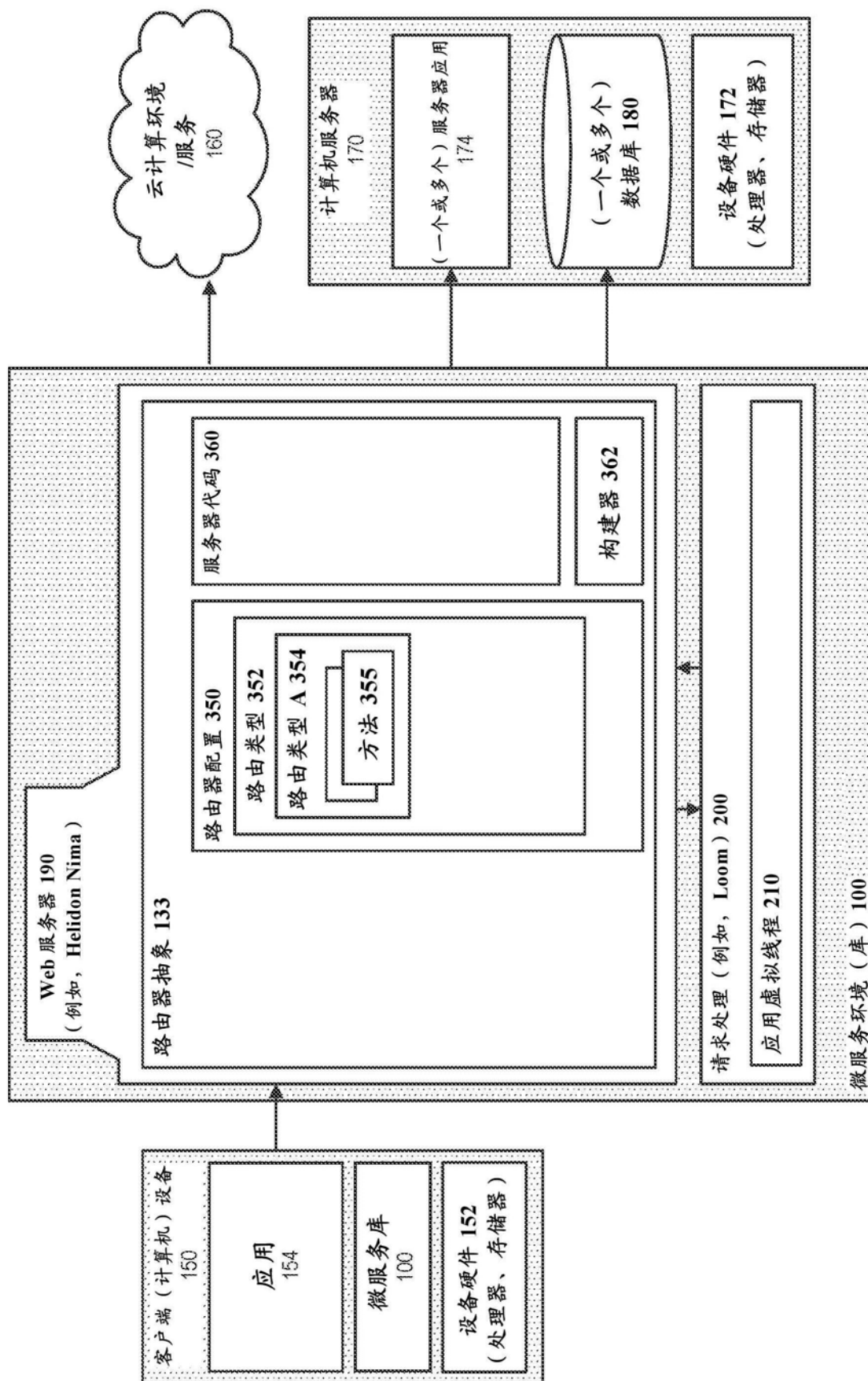


图19

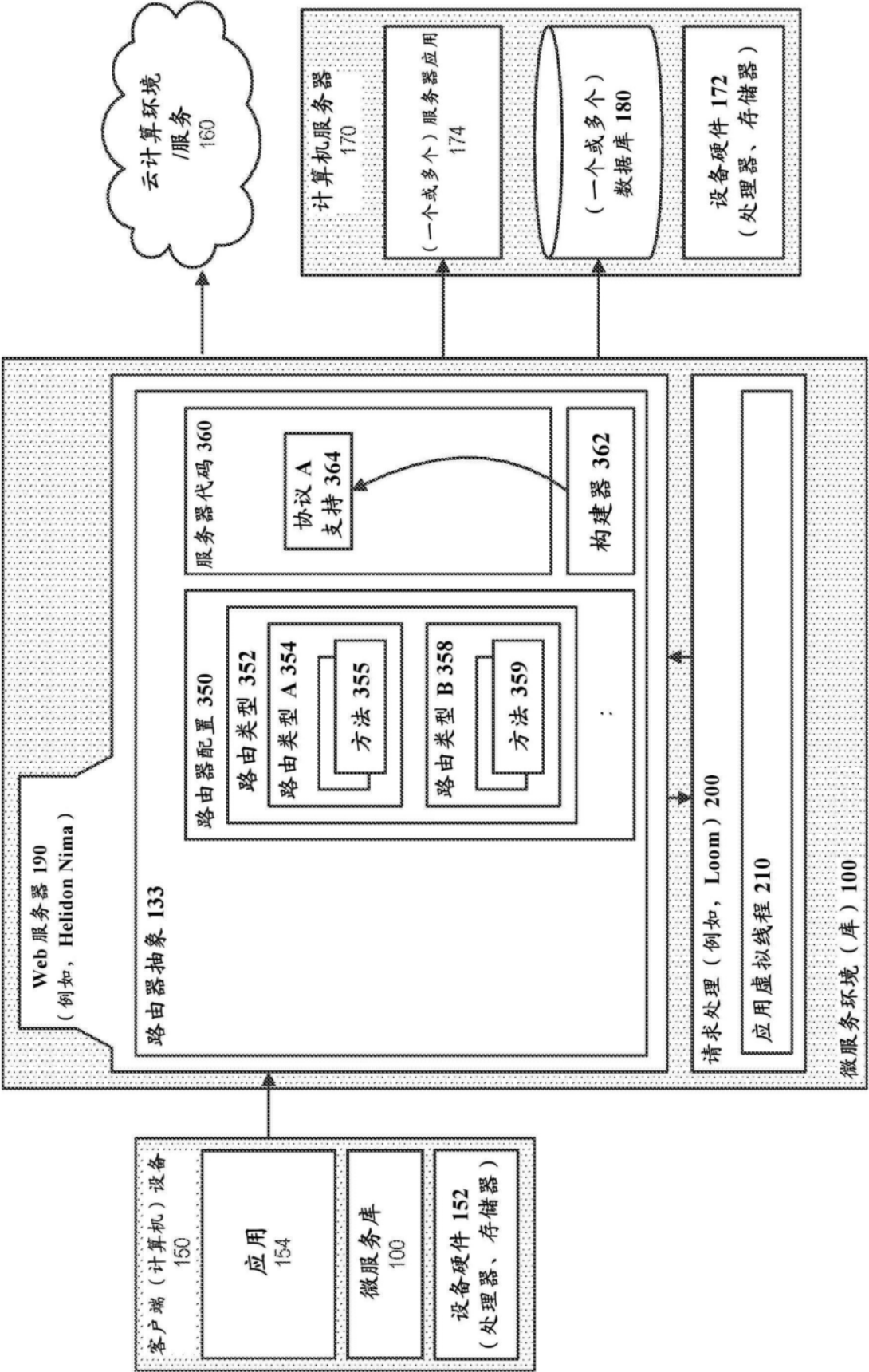


图20

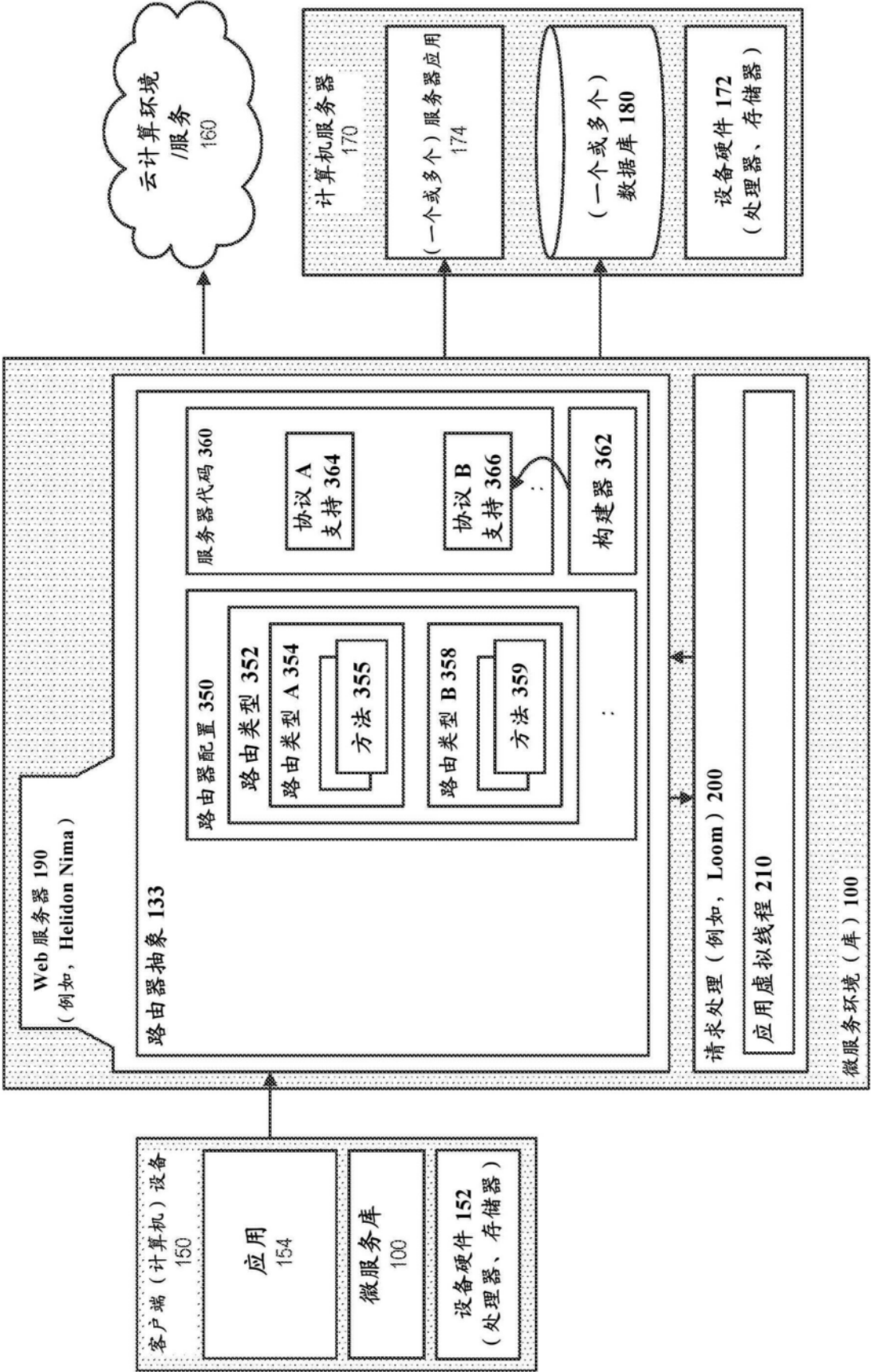


图21

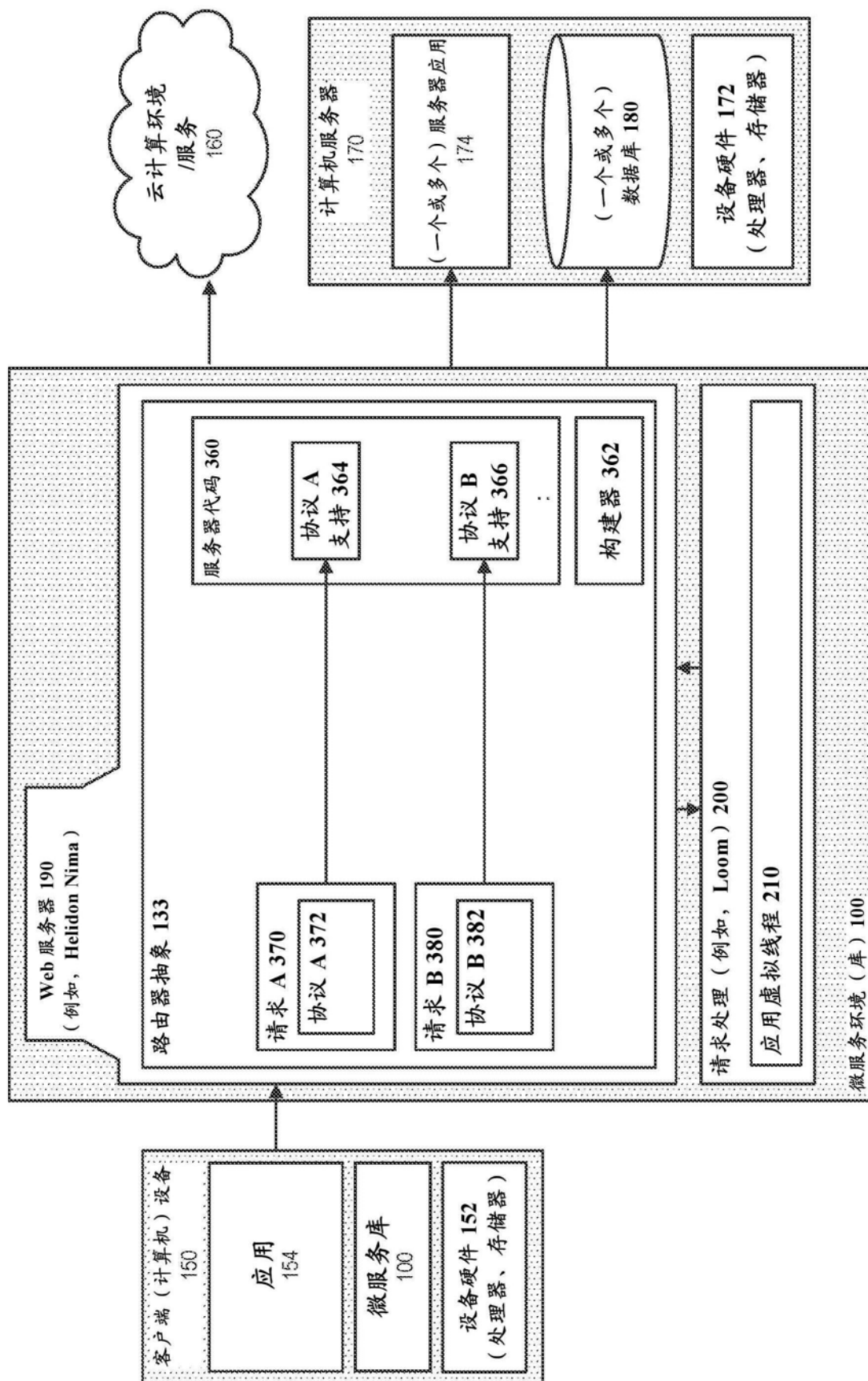


图22

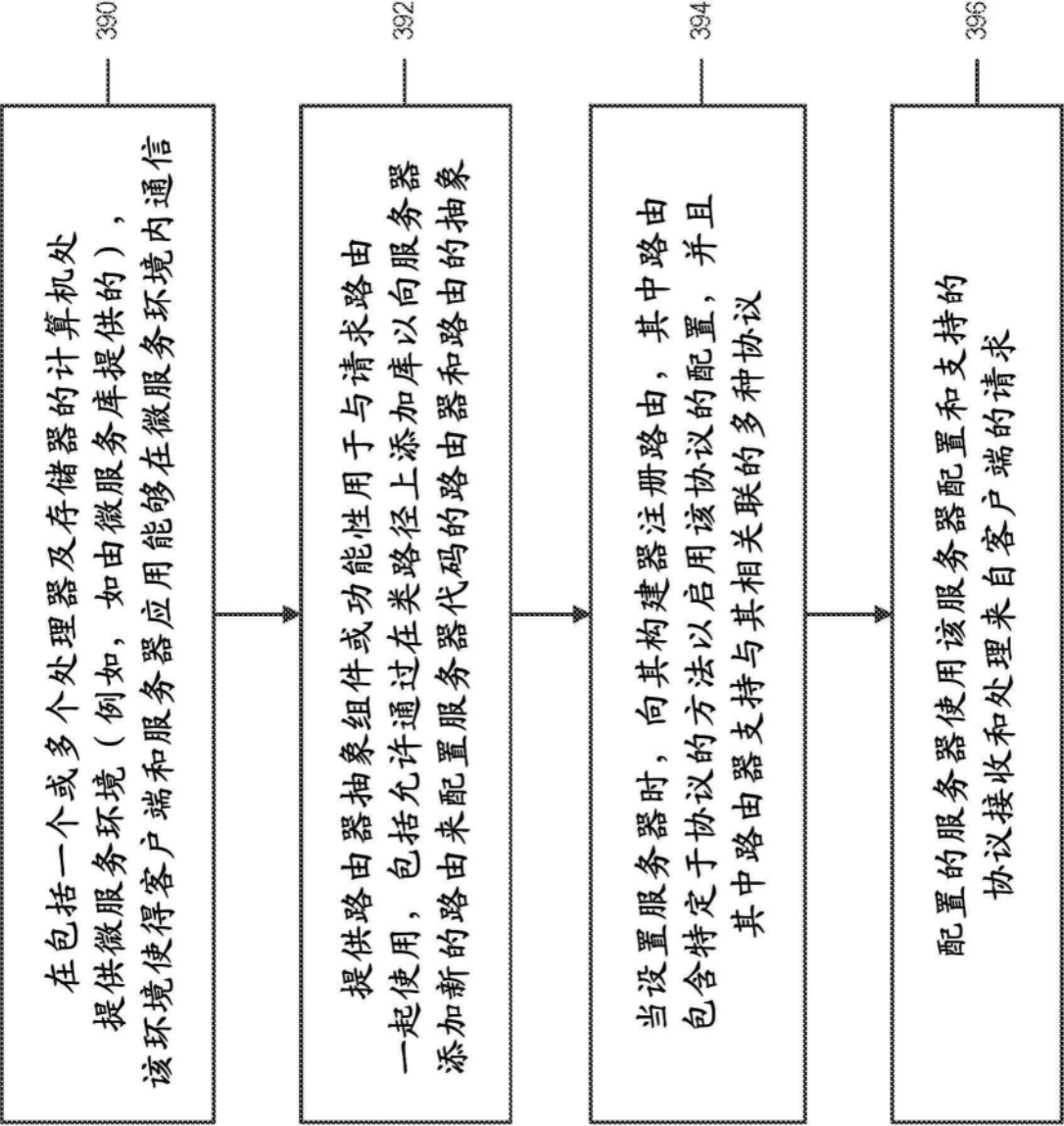


图23

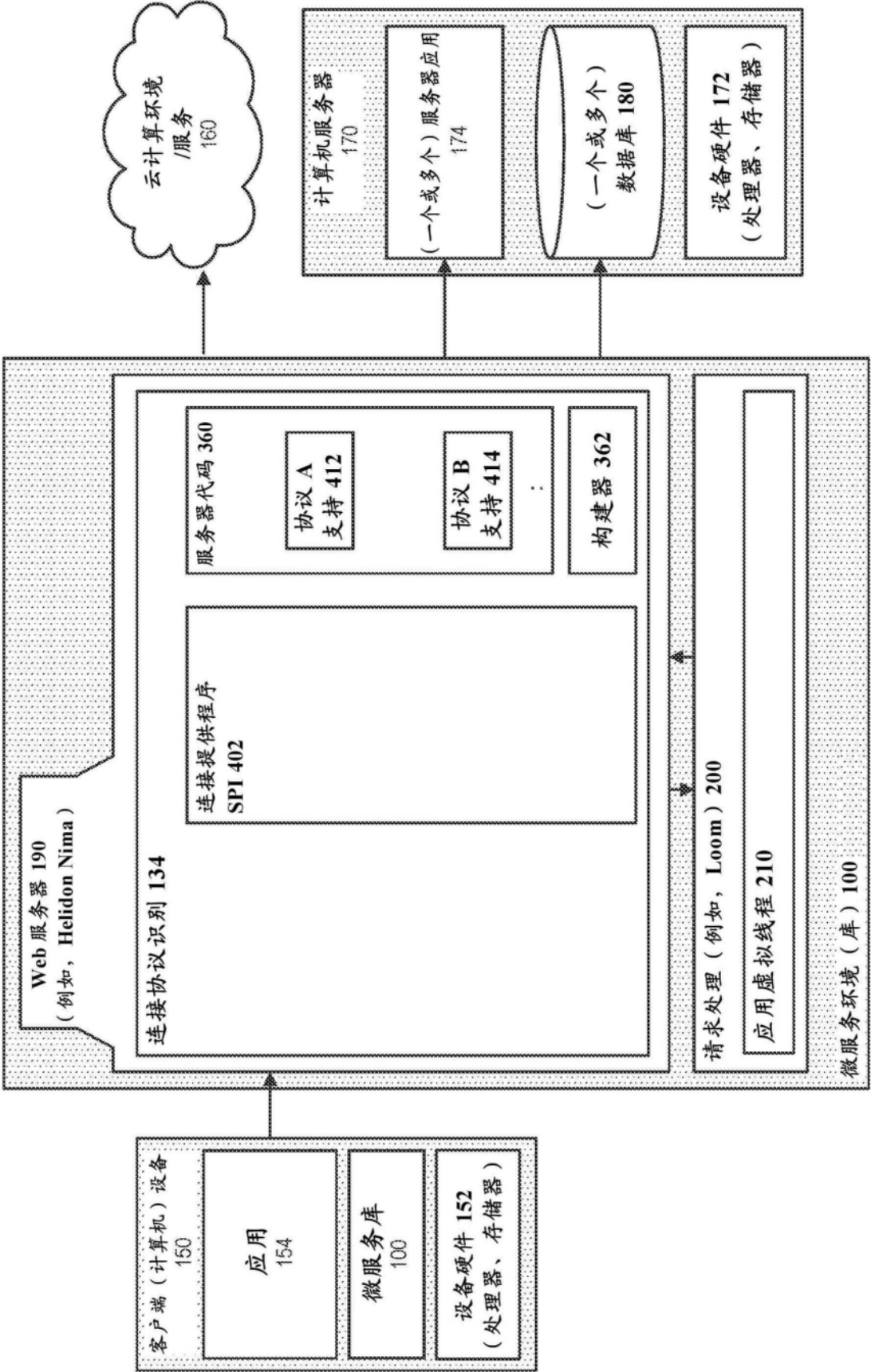


图24

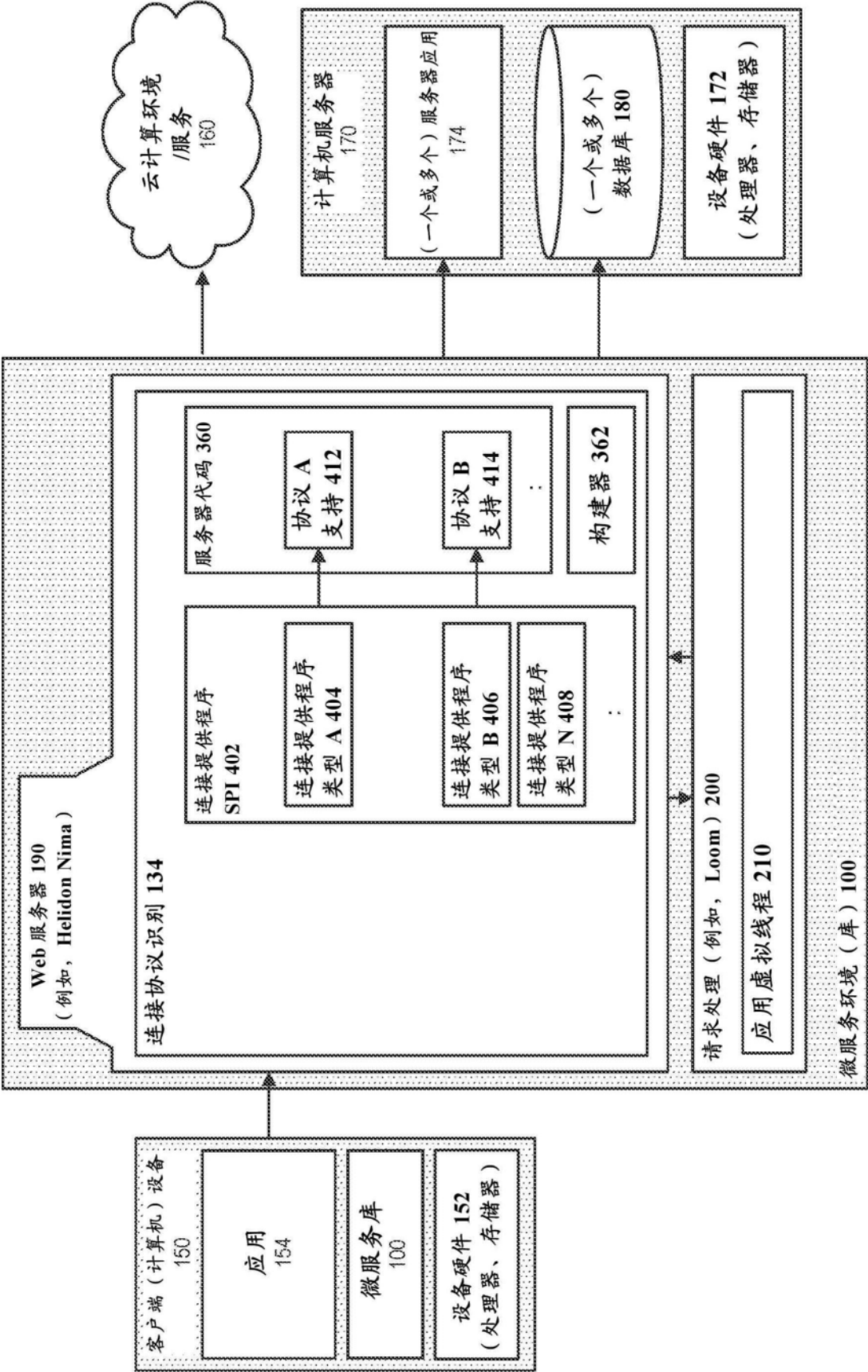


图25

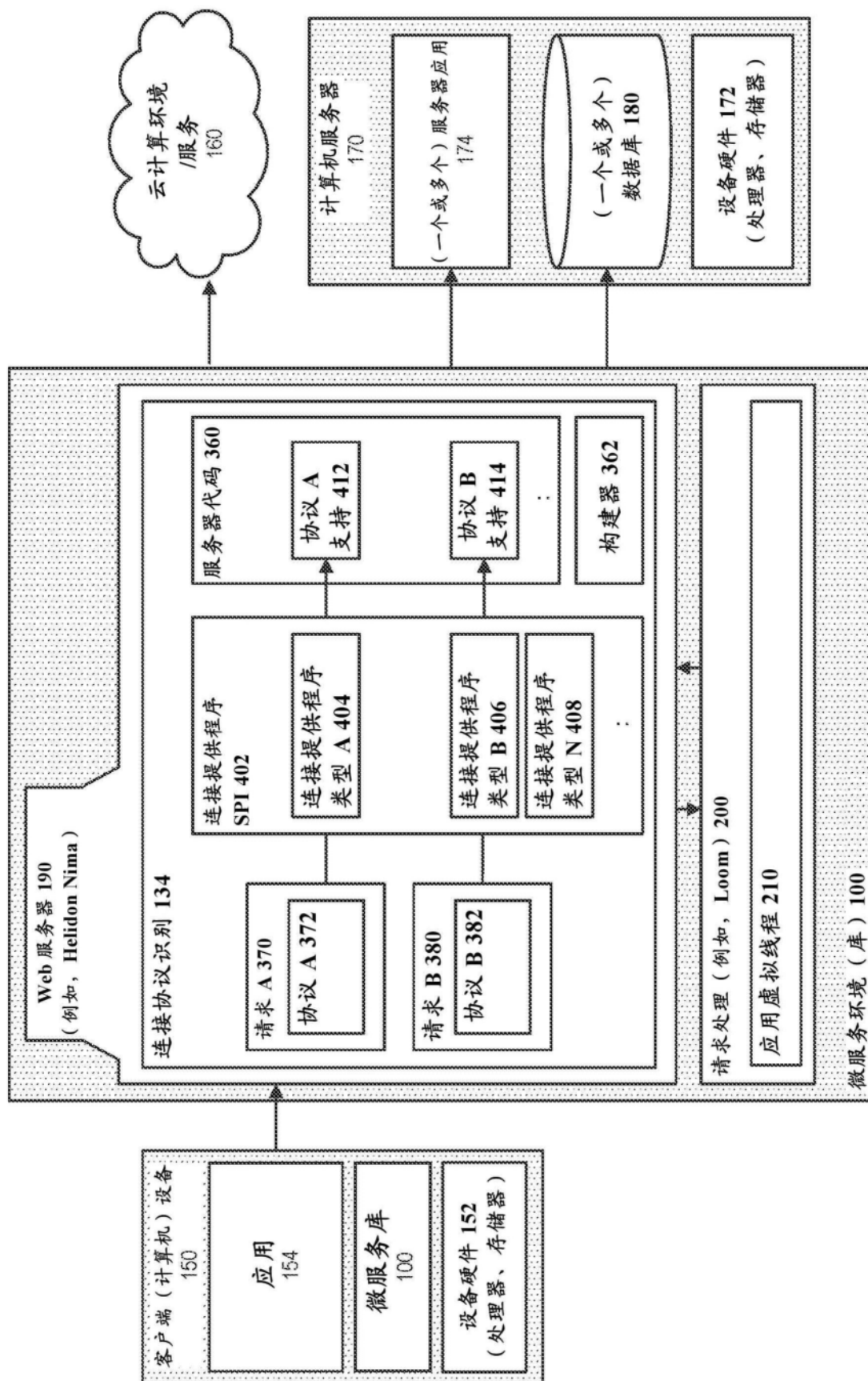


图26

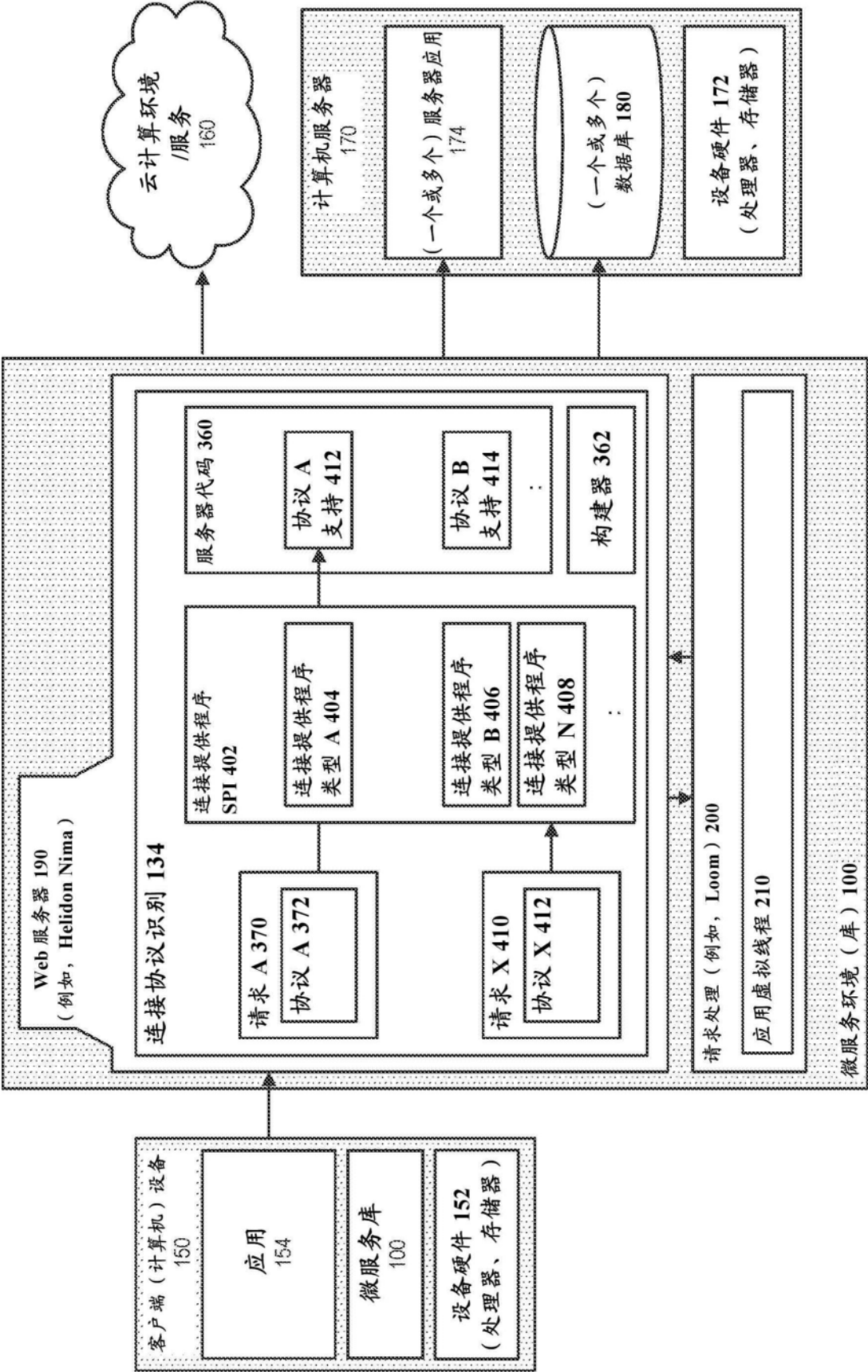


图27

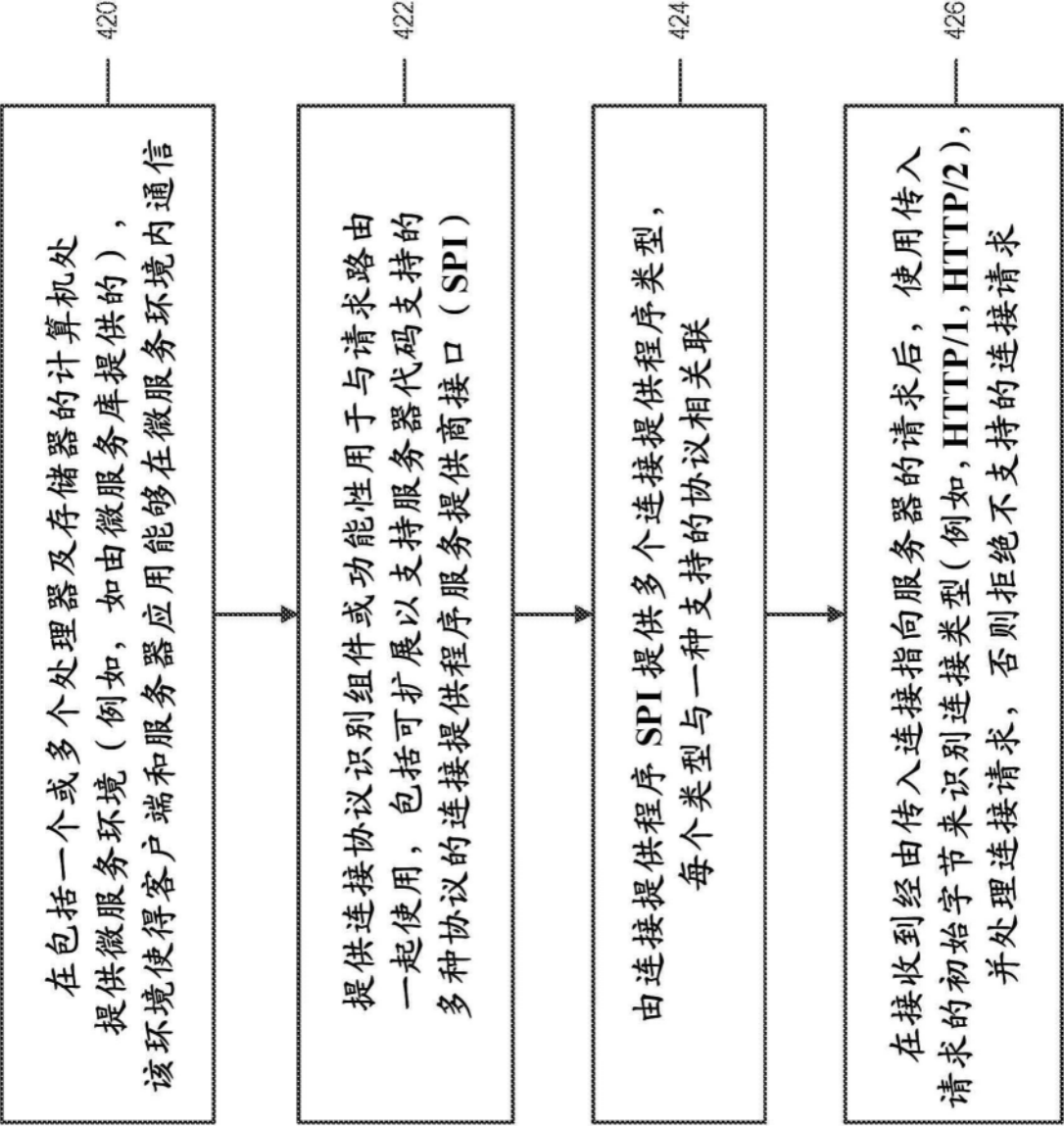


图28