科技报告

报告名称:	陈家镇国际生态社区能源管理中心建设关键技术
支持渠道:	科技部等中央单位与上海市共同推进重大任务科研专项
报告类型:	最终报告
编制单位:	上海陈家镇建设发展有限公司
编制时间:	2020 年 11 月 10 日

摘 要

请在这里输入摘要内容.

版 权 声 明

目　　录

1 微服务动态更新

1.1 技术背景

1.1.1 研究背景

随着软件系统规模的扩大，系统维护和演化也变得越来越复杂。传统的应用程序将所有功能都打包成一个独立的单元，因此也被称为单体应用。单体应用架构简单，在开发、测试和部署等方面都比较方便。但随着软件应用的发展，单体应用的弊端开始显现：不够灵活，对应用程序做任何细微的修改都需要将整个应用程序重新构建、重新部署，妨碍软件应用的持续交付；技术栈受限，开发团队的所有成员通常都必须使用相同的开发语言。因此，微服务架构应运而生。微服务架构将大型的单体应用拆分成若干个更小的服务，使得每个服务都可以独立地进行部署、升级、扩展和替换，服务间使用轻量级的通信协议进行通信 (例如，同步的 REST，异步的 AMQP 等)，降低服务间耦合度的同时满足了软件程序对于快速持续集成和持续交付的需求。

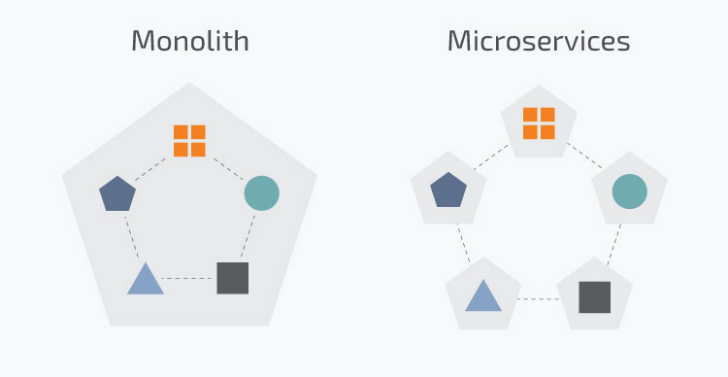

图 1: 单体应用和微服务应用

微服务可以帮助我们对应用进行有效地拆分，缩短软件开发周期，降低软件维护成本。但随着用户需求和软件运行环境的不断变化，已经部署的软件需要不断进行更新修复，以扩展整个系统的功能，提高软件的性能，或者是修复软件存在的故障缺陷。然而，随着用户越来越依赖于系统，他们越来越不能接受传统的停止，更新，重启的软件更新方式。因为直观来说，系统中断会给用户带来不便，引起管理代价的显著提高，比如一些高可靠服务的平台会因为服务中断而蒙受名誉和经济的损失，如云服务平台、银行交易系统。因此我们需要在不中断软件正常运行的前提之下，对软件系统进行更新，即动态软件更新（dynamic software update)。具体来讲，动态软件更新（也称动态适应（dynamic adaption)、运行时更新（run-time update)、动态演化（dynamic evolution)）是指软件在运行过程中，不停止已有应用的执行，而实现的运行时行为变更。

软件的动态更新并不容易实现，在安全性、及时性、低干扰性方面面临着巨大的挑战。安全性保证应用程序在更新过程中及更新过程后均能正确运行。及时性要求动态更新操作能够尽快完成，从而减少动态更新影响的持续时间。低干扰性要求更新过程尽量减少对用户影响的程度，提升用户体验。

1.1.2 相关技术

微服务架构的一大特点就是不同的服务间可以独立分布式地进行部署，常见的微服务更新部署方式有：

- 蓝绿部署（Blue/Green Deployment）

  蓝绿部署是一种基于硬件的动态更新技术，关键在于使用冗余的硬件设备，即在保持当前系统正常运行的前提下，在额外的一台机器上加载新的软件系统，测试完成之后，将流量切到新版本的系统上，这样的动态更新方式在安全性，及时性上没有太大的问题，但是在更新时，会阻断用户正在进行的业务，存在一定的干扰性，且成本较大，需要准备两套

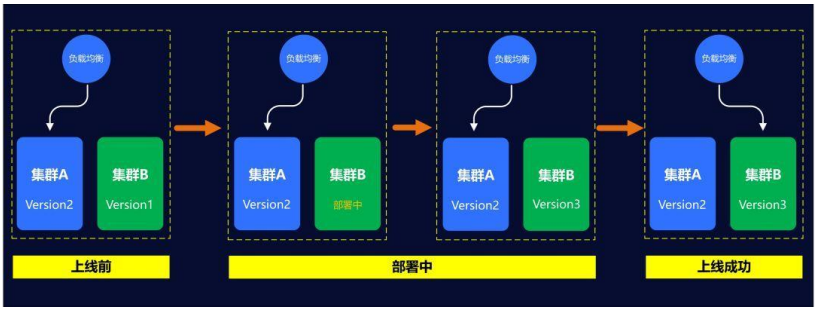

图 2: 蓝绿部署

- 灰度发布/金丝雀部署（Canary Update）

  灰度发布是一种软件发布方式，以 A/B 新旧两个版本为例：让大部分用户继续用老版本 A，同时让一小部分用户开始使用新版本 B，在部署新版本应用中，需要排掉原有的流量并进行部署。然后根据用户反馈，逐步扩大新版本应用范围，最终实现版本的更新。灰度发布是目前工业界中比较流行的动态更新方式，但是不可避免地会在更新过程中干扰用户正在进行的事务。

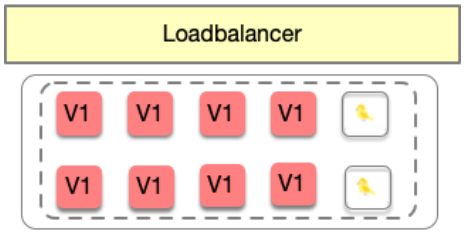

图 3: 灰度发布/金丝雀部署

- 滚动发布（Rolling Update）

  滚动发布是指取出一个或多个服务器停止服务，执行更新，并重新将其投入使用。周而复始，直到集群中所有的实例都更新成最新版本。这种动态更新方式相较于蓝绿部署，虽然节约了硬件资源，但是会遇到更多的问题，在金丝雀发布基础上的进一步优化改进，是一种自动化程度较高的发布方式。

上述的微服务更新部署方式，都保证了更新过程中服务的可用性。但由于更新的时机是任意可选的，用户的请求也是会实时产生的，这样就难免会出现用户某一次请求前一次访问

的是旧版本的微服务实例，随后服务进行了更新，本次请求再次调用到的变成了新版本的微服务实例。如新旧版本的服务存在不可兼容性，这样的行为显然将返回错误，也即破坏了动态更新对于安全性的要求。

1.2 微服务动态演化一致性保障协议

1.2.1 基本原理

本小节将阐述微服务动态演化一致性保障协议的基本原理，说明如何保证正确且高效地完成系统的动态演化。首先引入一个简单但又不失一般性的例子，在系统中存在着这样四个微服务：Portal、Proc、Auth 和 DB。用户首先会向 Portal 微服务发起请求，收到请求后的 Portal 微服务先是调用 Auth 微服务，得到当前请求的一个验证令牌。然后 Portal 使用得到的令牌接着对 Proc 微服务发起请求，收到请求的 Proc 微服务利用随请求而来的令牌向 Auth 微服务进行验证，若收到的回复是验证通过，则进一步调用 DB 微服务以读取数据库中的具体信息并返回；否则，返回请求失败消息，然后将最终结果反馈给发起请求的用户。

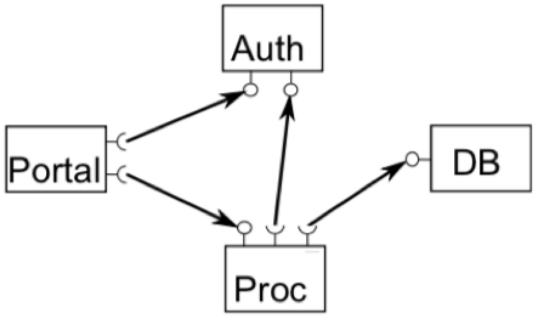

图 4: 示例系统

考虑到这样一种情况：在系统运行时，由于采用了更为安全和快速的加解密算法，由于整个系统所有的加解密过程都在 Auth 微服务中进行，因此我们可以仅对 Auth 微服务进行版本的更新及替换，同时对其它的微服务保持透明。在前述的系统运行流程中，若我们在 Portal 微服务调用完 Auth 微服务得到对应的验证令牌后且 Proc 微服务还未使用该令牌向 Auth 微服务发起验证之前的这个时间段内，Auth 微服务不服务于任何的事务，处于空闲状态，我们对 Auth 微服务进行版本的更新及替换。随着系统的继续运行，当 Proc 微服务利用使用旧版本算法所加密生成的验证令牌来向 Auth 微服务发起调用，但此时的 Auth 微服务已完成更新，新旧算法的不兼容性导致验证失败，最终返回失败信息。显然 Auth 微服务不正确的动态演化，导致了系统在无故障的情况下不能正确地处理用户请求。

动态演化要求系统能够在不停止服务的情况下进行更新，用户发出的请求在更新前后都能被正确地处理且返回结果。因此，微服务动态演化一致性保障协议的基本原理是考虑在系统运行的过程中，当一次请求需要多次涉及同一个微服务，且不同版本的微服务因兼容性等原因将返回不同的结果时，需要依据事务的运行流程来决定调用的服务版本，从而保证用户请求不受服务动态演化的影响。

1.2.2 一致性保障协议

在介绍一致性保障协议之前，首先介绍协议中将要用到的几个基本概念的定义：

definition [**事务与分布式事务**] **一个事务表示在某一微服务上执行且在有限时间内结束的一系列操作，操作包括本地数据的计算和微服务间消息的传递等。在某一微服务上运行的事务可以向其它微服务发起请求，从而在其它的微服务中生成一个新的事务。前者和后**

者之间的事务关系我们称为父子事务，外界调用系统所产生的第一个事务为根事务。根事务及其随后的调用所产生的所有子事务，我们称为一个分布式事务，并使用其根事务的符号进行标识。

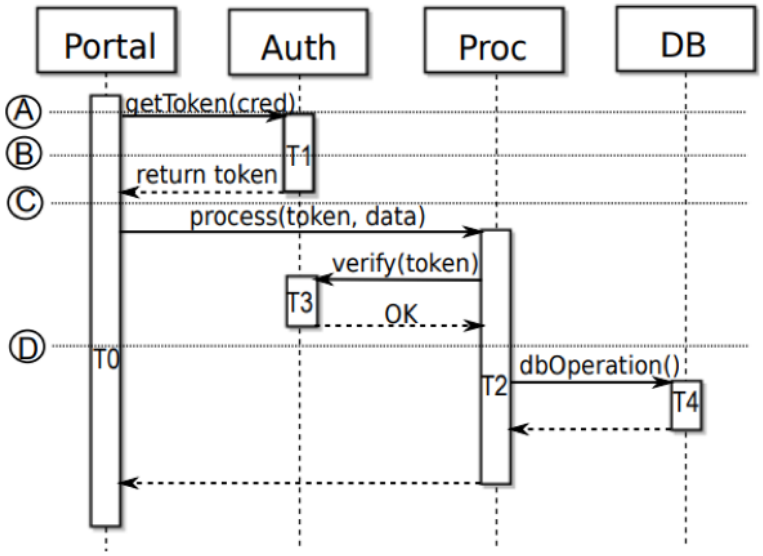

图 5: 示例系统运行时序图

上图即表示了我们前面所描述的系统中四个微服务之间的运行流程。外部请求首先在 Portal 微服务上生成 T0 根事务，然后通过微服务间的调用 T1、T2、T3、T4 事务依次生成。这样一次完整的调用过程我们便称为分布式事务，用根事务符号 T0 来进行表示，而 T1 ~ T4 事务都可称为是 T0 的子事务。

[**版本一致性**] **一个分布式事务是版本一致的，当且仅当不存在它的两个子事务，它们在系统运行时先后在同一个微服务的不同版本上运行。一次动态演化所导致的系统演化是版本一致的，当且仅当所有的分布式事务是版本一致的。**

在我们前面所描述的系统运行的例子中，考虑对 Auth 微服务进行更新，如果更新发生在根事务 T0 生成之后，但在生成 T1 之前 (即时刻 A)，那么后续在 Auth 微服务上生成的 T1、T3 子事务都使用的是 Auth 微服务的新版本，分布式事务 T0 满足版本一致性；如果更新发生在 T3 子事务完成之后 (即时刻 D)，那么在 Auth 微服务执行过的 T1、T3 子事务都使用的是 Auth 微服务的旧版本，分布式事务 T0 满足版本一致性；而在前面系统出现错误的例子中，更新发生在子事务 T1 完成之后，但 T2 子事务还未生成之前 (即时刻 C)，那么将导致 T1 子事务和 T3 子事务在 Auth 微服务的不同版本上运行，此时分布式事务 T0 便不满足版本一致性。

版本一致性虽然描述了微服务动态演化的准则，但我们并不能直接利用这个规则来对微服务动态演化的时机进行判断，因此我们需要引入动态依赖边，使得我们可以分布式地在各自的微服务上进行本地的操作，便可以判断当前微服务是否满足动态演化的条件 (即版本一致性)。

[**动态依赖边**] $C \xrightarrow[T]{future(past)} C'$ **表示在分布式事务 T 的运行过程中，有一条从微服务 C 到微服务 C' 的 future(past) 边。其中，future 边表示在 T 之后的运行过程中，微服务 C 有可能还会在微服务 C' 上生成新的子事务；past 边表示在 T 之前的运行过程中，微服务 C 已经在微服务 C' 上成功地生成过子事务并返回。**

有了动态依赖边的定义，我们使用如下的规范性准则，使得依赖边可以在每一个分布式事务的运行过程中动态且正确地进行增加和删除操作：

- 在微服务 C 上运行的每一个事务 T，开始时添加一对动态依赖边（$C \xrightarrow[root(T)]{future} C$ 和 $C \xrightarrow[root(T)]{past} C$），标识为 T 的根事务，事务结束时删除；
- 每一条动态依赖边，都需要微服务 C 和微服务 C' 之间存在真正的静态边，即存在对应调用关系；
- $C \xrightarrow[T]{future} C'$ 边必须要在 T 的第一个子事务生成之前添加，并且应等到 T 不会通过微服务 C 在微服务 C' 上生成子事务时，才能够删除；
- $C \xrightarrow[T]{past} C'$ 边在 T 的运行过程中，当微服务 C 在微服务 C' 上生成的子事务即将结束时生成，应等到 T 结束时才能够删除。

有了上述动态依赖边的支持，在系统运行的任何时刻，每个微服务都可以知晓指向自己的动态依赖边，以及负责从自己指向其它微服务动态依赖边的增删，所以我们可以将动态演化的版本一致性条件转化为本地可判断的闲置 (FREENESS) 条件：

[**闲置 (FREENESS)**] **微服务 C 针对某一个分布式事务 T 是版本一致的，当且仅当不存在由 T 标识的一对 future/past 边指向微服务 C。微服务 C 在某一时刻下的状态是版本一致的，当且仅当微服务 C 对所有当前运行的分布式事务是版本一致的。**

1.2.3 微服务动态演化过程

利用动态依赖边的定义以及增删的规范性准则，带有动态依赖边的分布式事务 T 在运行时流程具体表现为如下三个阶段：

1. 准备阶段：分布式事务 T 在第一个微服务 C 上完成初始化但还未调用任何其它微服务，即生成子事务之前，C 通知所有可能依赖的微服务创建 future 边并等待回复。被通知的微服务同理，通知其依赖的微服务执行创建操作。当收到所有 future 边创建成功的回复后，事务 T 开始运行，如图6；

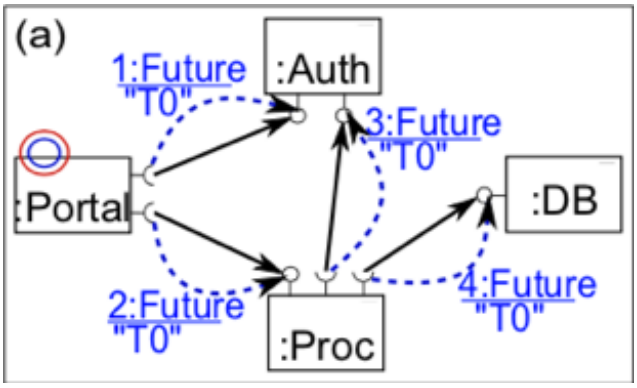

图 6: 准备阶段：process_a

2. 运行阶段：当知道微服务 C 不会再通过静态边 $C \xrightarrow{static} C'$ 在微服务 C' 上生成 T 的子事务，而且不存在指向 C 的标识为 T 的 future 边，那么 future 边 $C \xrightarrow[T]{future} C'$ 可以被删除；当微服务 C' 上标识为 T 的子事务将要结束时，通知微服务 C 创建 past 边 $C \xrightarrow[T]{past} C'$，且应保证创建成功才能结束子事务，如图7；

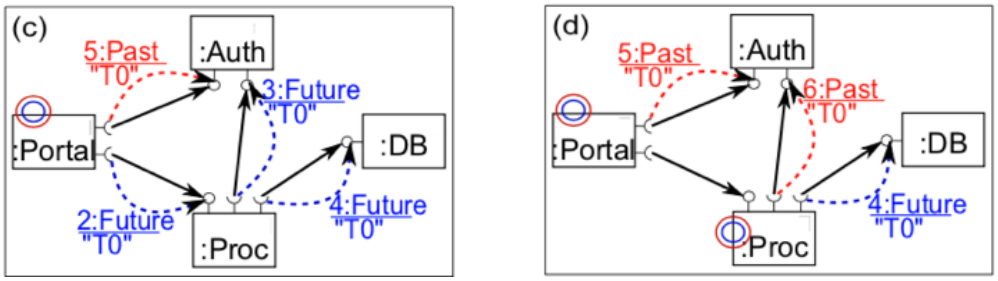

图 7: 运行阶段：procss_c 和 process_d

3. 结束阶段：和准备阶段时的创建操作类似，从第一个微服务 C 递归向所有的依赖微服务发送删除消息，删除标识为当前分布式事务 T 的所有动态依赖边。

前面的运行流程中，假设我们需要对 Auth 微服务进行一致性演化且当前只有分布式事务 T0 处于运行状态，利用闲置 (FREENESS) 的定义，在时刻 A 和时刻 D，Auth 满足更新条件；而在时刻 C，存在执行 Auth 微服务且标识为 T0 的一对 future/past 边，因此不满足更新条件。

微服务 C 只有满足了闲置 (FREENESS) 条件才能够进行动态演化，但由于系统处于持续运行且不断接收请求的状态，同一时刻存在不同的分布式事务在系统中运行，因此若仅仅通过等待然后判断的策略 (WF)，微服务有可能永远不能满足闲置 (FREENESS) 条件，从而无法及时地进行动态演化。除了 WF 策略，我们还可以使用下面的两种策略，使得微服务能够及时地完成动态演化操作：

- CV：允许不同版本的微服务实例同时存在并提供服务

  当微服务 C 需要进行动态演化时，启动新版本的实例并同时提供服务。当需要在微服务 C 上生成子事务 T 时，若已经有一条标识为 root(T) 的 past 边指向微服务 C，那么新生成的子事务 T 将在旧版本的微服务实例上运行，否则在新版本的微服务实例上运行。由于事务会在有限时间内完成，因此旧版本的微服务实例上运行的事务逐渐减少为零，最终将旧版本实例从系统中移除。

- BF：暂时阻塞新事务的生成

  当微服务 C 需要进行动态演化时，若需要在其上生成新的子事务 T，首先判断是否存在标识为 root(T) 的一条 past 边指向自己，若存在则子事务 T 可以在微服务 C 上生成并继续执行；不然，阻塞子事务 T。旧版本的微服务实例上运行的事务会在有限时间内完成，最终微服务 C 将满足闲置 (FREENESS) 条件，此时进行版本的更新，然后恢复之前阻塞的所有事务。

1.3 基于 Istio 的协议实现

1.3.1 Service Mesh 与 Istio

与单体应用相比，微服务架构将系统应用的复杂度从单体应用内部的测试、部署、维护等转变到了微服务的连接调用、管理部署和监控等方面，因此微服务架构会极大地增加运维工作量，开发人员需要投入更多的精力来保证远程调用的可靠性与数据一致性。为了简化开发，开发人员通过典型的类库和框架 (如 Netflix OSS 套件、Spring Cloud 框架)，编写较少的代码和注解就可以完成微服务间的服务发现、负载均衡、熔断、重试等功能。但此种办法缺点在于，开发人员需要掌握并熟练使用的内容较多，而且服务治理的功能不够齐全，很多功能需要自己进行拓展，编程语言也有所受限。

这样所开发出来的微服务中，关键的业务逻辑代码和其它的用于管理服务间关系的非功能性代码混杂在一起。此时，若考虑为每一个微服务实例部署一个 Sidecar，将所有前述的非功能性代码移到 Sidecar 中，由 Sidecar 来负责提供服务发现等辅助功能，而开发人员只需专注于微服务的业务逻辑，从而有效地进行了解耦。当为大型的微服务系统部署 Sidecar 时，微服务之间的服务调用关系便形成了服务网格 (Service Mesh)，如图8所示。

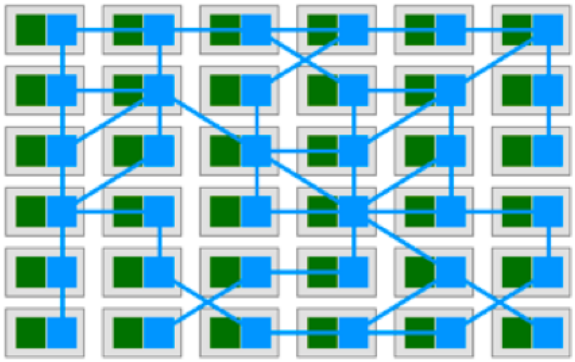

图 8: 服务网格 (Service Mesh)

当前最主流的 Service Mesh 方案，是 Google 和 IBM 两个公司联合 Lyft 合作的开源项目 Istio。在 Istio 的架构中，直接在 Lyft 开发的 Envoy 之上进行了拓展，然后将其作为 Sidecar 代理进行部署。Envoy Proxy 作为数据面，除了提供服务发现、负载均衡、限流熔断等功能，还可以协调微服务的所有出入站流量，收集相关的性能指标，与控制面进行交互。而有了数据面的支持，控制面一方面通过 Pilot 组件下发配置信息到相应的 Envoy Proxy 中，负责流量管理，另一方面通过 Mixer 组件收集遥测数据，从而实现了对整个微服务系统多方面的掌管与监控。

1.3.2 协议实现具体流程

基于 Istio，部署前述的示例系统，具体架构变为图10。用户将访问请求发往 Ingress Gateway，由 Ingress Gateway 依据相应的路由规则，再将请求转发至具体的后端服务中，然后收集运行结果并进行返回。每一次用户的请求都将产生一个分布式事务，其中可能会涉及多个服务的远程调用，但可以使用请求头中的 x-b3-traceid 字段进行唯一标识。Ingress Gaeway 一方面将对应的服务接口暴露出来供用户进行访问，另一方面为用户屏蔽了后端服务的具体实现和路由转发等服务治理功能。

以示例系统中的 auth-service 服务为例，具体的实现和更新流程为:

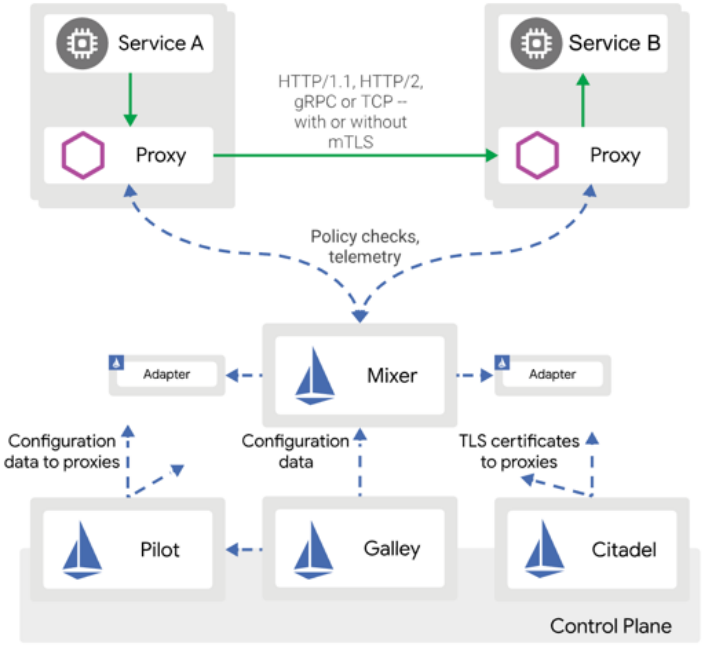

图 9: Istio 架构图（v1.4）

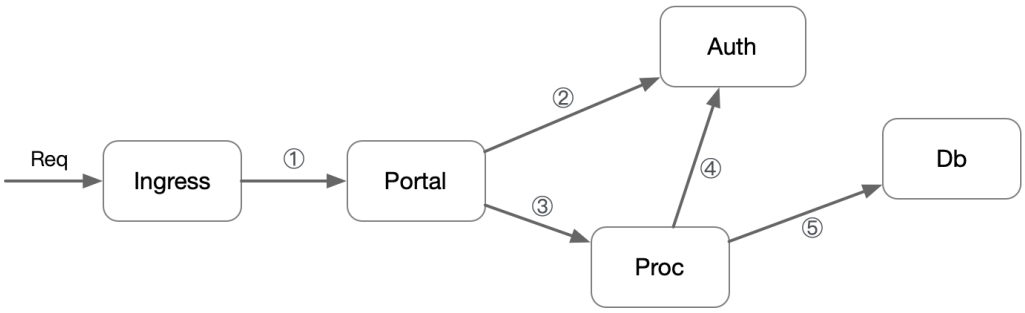

图 10: Istio 中的示例系统

插入 Envoy 容器和 TraceManager 容器 修改 Istio 框架中的自动注入功能，当每个服务实例部署时，我们将为每个服务实例对应部署 Envoy 容器和 TraceManager 容器。其中，Envoy 容器负责从 Istio 控制面中接收用户的配置信息，然后将配置信息转换成对应的规则，对所有的出入站流量进行管理；与此同时，当存在具体的入站流量访问对应的服务时，Envoy 会将此次分布式调用的唯一标识 (x-b3-traceid) 和相关信息发送至 TraceManager 中。而 TraceManager 容器则负责接收从 Envoy 容器中发送过来的信息，对其进行记录管理，为后续的更新过程提供支持。插入的 Envoy 和 TraceManager 容器分别作为客户端和服务端，通过 Unix Domain Socket 的方式进行通信，从而最大程度地减少了消息传送对 Envoy 快速处理用户请求的影响。

创建初始的路由规则和版本信息 当整个系统部署完成时，为每个服务创建初始的路由规则和版本信息。图12中定义了路由规则，要求流量调用到 auth-service 时，所有流量都发往 v1 版本，且在返回的头信息中会带上自定义的信息，后续将利用此头信息进行流量的转发；图13中定义了版本信息，说明带有 version: v1 标签的所有 auth-service 的服务实例都属于 v1 版本；此时系统的运行状态对应显示为图14

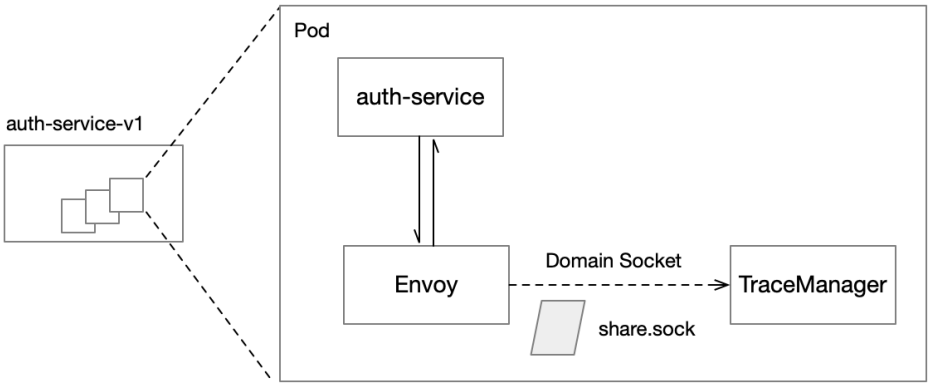

图 11: 插入容器

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: auth-service
spec:
  hosts:
  - auth-service
  http:
  - route:
    - destination:
        host: auth-service
        subset: v1
      headers:
        response:
          add:
            x-version: auth-service-v1
```

所有流量发往v1版本，
返回头中添加自定义信息

图 12: v1 路由规则

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: auth-service
spec:
  host: auth-service
  subsets:
  - name: v1
    labels:
      version: v1
```

定义v1版本对应的标签

图 13: v1 版本规则

在系统的运行过程中，当存在入站流量调用到 auth-service 时，Envoy 容器执行拦截操作：一方面将流量转发往负责的服务实例，另一方面将此次调用所对应的 traceid 信息从头信息中提取出来并发送往 TraceManager 容器，TraceManager 容器则将对应的 traceid 存到对应的 TraceidSet 中，该集合代表了当前所有调用过 auth-service 的分布式事务 id；同时，当本次用户请求处理完成并返回 Ingress Gateway 时，Ingress Gateway 实例内部会执行同样的添加操作，这里的 TraceidSet 则代表了当前已完成并返回的分布式事务 id 集合。

新版本上线 当我们需要对 auth-service 进行更新时，首先需要将新版本的服务进行上线部署。与此同时，相关的路由规则和版本信息修改为：图16中共包含三个路由规则，依次进行匹配，第一条规则表示若请求流量中的头信息包含 v1 版本信息，则将该流量转发到 v1 版本中；第二条规则表示若请求流量中的头信息包含 v2 版本信息，则将该流量转发到 v2 版本中；若第一第二条规则均不满足，则应用第三条规则，将流量转发到 v1 版本中，同时添加对应的自定义头信息。图17则添加了新版本服务对应的标签定义。

此时，若某个分布式事务还未调用过 auth-service，则请求的头信息中不包含相关的版本信息，依据第三条规则将流量转发往 v1 版本；若此分布式事务已调用过 auth-service 服务，则由于第三条规则的存在，其请求的头信息中必然包含了 v1 的版本信息，依据第一条规则同样将流量转发往 v1 版本。因此，虽然新版本的 auth-service 已完成上线部署，但还

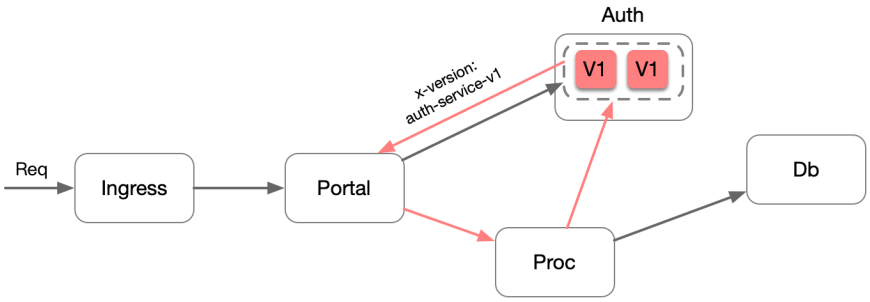

图 14: 流量全部发往 v1

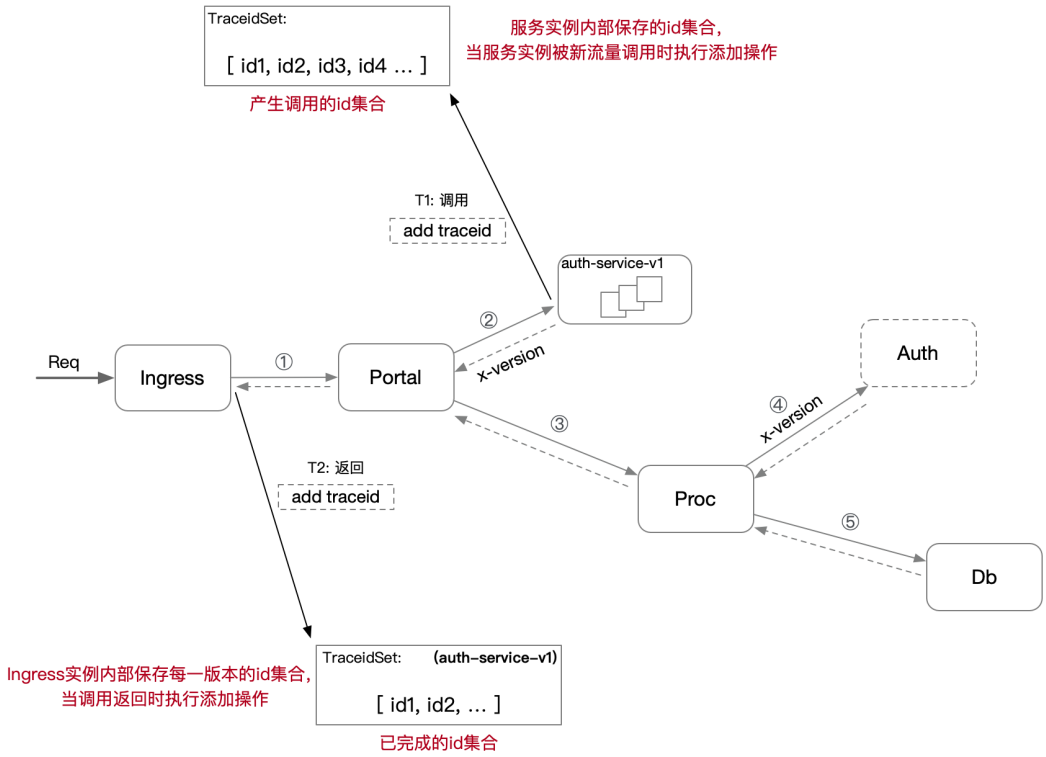

图 15: 事务记录

没有流量经过。对应的系统运行状态如图18所示

路由规则更新 待前一步骤的配置完成，我们接着对路由规则进行配置修改：如图19所示，前两条规则并未发生修改，依然是优先根据头信息中的相关内容来进行流量的转发。关键在于将第三条规则的默认流量转发从 v1 改为 v2。

进行第三步的路由规则更新，将默认的流量转发目标从 v1 改为 v2，不影响任何分布式事务的版本一致性，无论该分布式事务处于系统运行中的哪个阶段。

对于任何一个分布式事务 T，若 T 在运行的过程中还未调用过 auth-service，则请求的头信息中不包含相关的版本信息，依据规则 T 将使用 v2 版本，不违背版本一致性；若 T 在过往子事务的运行过程中调用过 auth-service，则请求的头信息中必然包含相关的版本信息，此时路由规则将进行相关的头信息版本匹配，并转发往同一版本的 auth-service，保证了版本一致性；若 T 在将来的运行过程中不会再调用 auth-service，同样不违背版本一致性。因此证明了进行第三步1.3.2的路由规则更新，任何分布式事务 T 都满足版本一致性的要求。

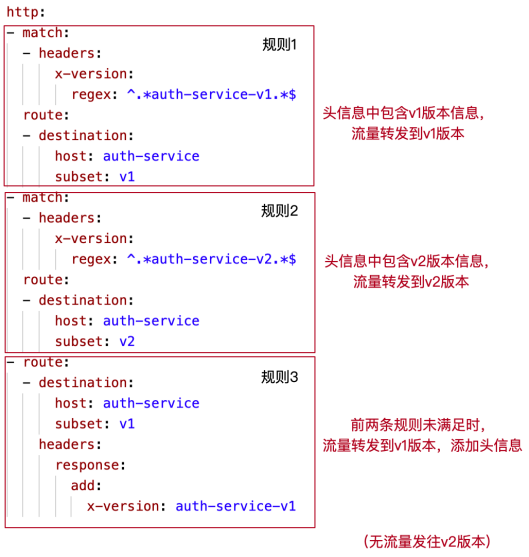

图 16: 默认发往 v1 路由规则

图 17: v1、v2 版本规则

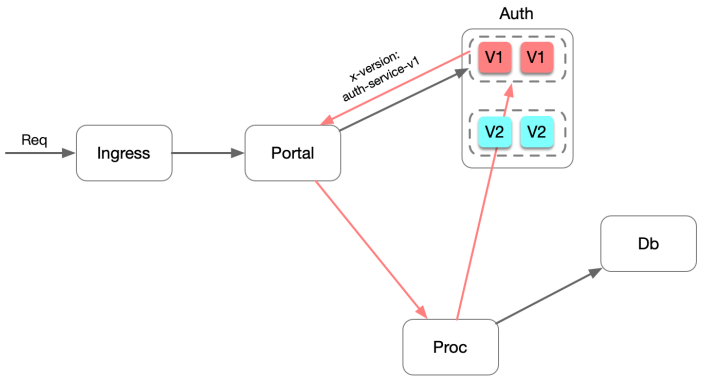

图 18: 流量默认发往 v1

更新完路由规则对应的系统运行状态如图20所示，新产生的请求将发往新版本 v2，运行中的请求则依据头信息内容进行匹配。

旧版本撤销请求 完成前述步骤后，我们便完成了新旧版本流量的正确导向，新产生的用户请求也可以及时地使用新版本的服务。进一步我们希望能够及时地对旧版本的服务实例进行撤销，避免其长时间地占用系统的资源。我们需要进行判断：何时旧版本的服务实例上的事务全部结束且不会再为其它事务提供服务？这时，便需要使用 TraceManager 容器中所保存的 TraceidSet。具体实现包括以下步骤：

1. 阻塞访问旧版本：当收到旧版本的撤销请求时，阻塞所有调用旧版本服务实例的新请求，即头信息中不存在对应版本信息的流量，同时直接返回 503(Service Not Available) 错误，在调用方侧执行重试机制，此时调用方将应用较新的路由规则，流量被转发到新版本的服务中。从而保证了不会再有新请求调用到旧版本服务实例，并且对 TraceManager 容器中保存的 TraceidSet 执行加锁，不再向其中添加 traceid，此时 TraceidSet 保存了所有调用过旧版本服务实例的分布式事务 id。流程如图21所示；

2. 同步 Traceid 集合：将所有旧版本实例中所保存的 TraceidSet 发送至 Ingress Gateway。Ingress Gateway 此时暂时阻塞添加 Traceid，将收到的 TraceidSet 和自己内部保存的

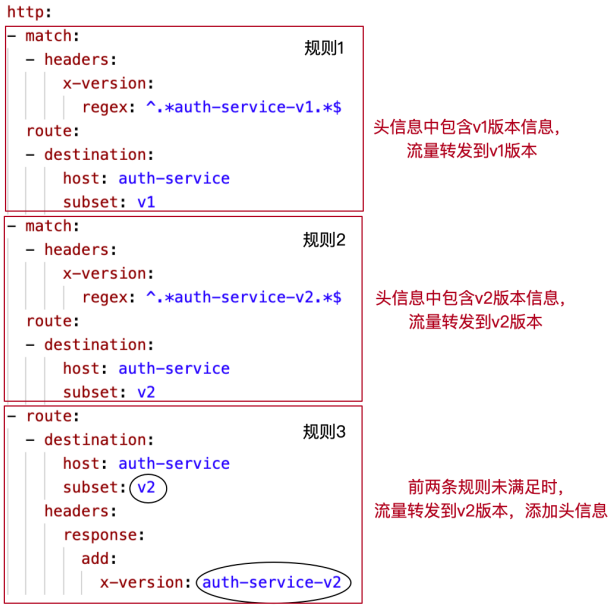

图 19: 默认发往 v2 路由规则

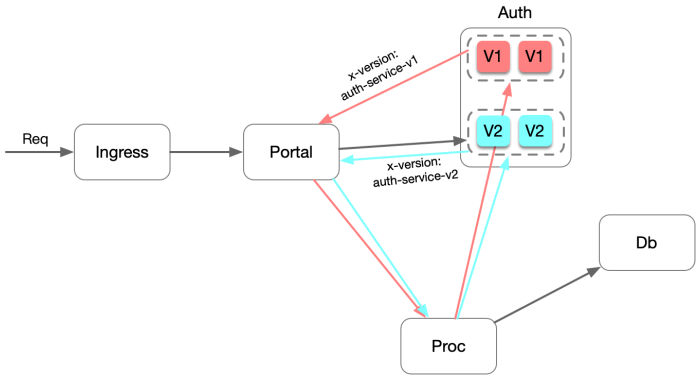

图 20: 新流量发往 v2

TraceidSet 进行同步，求得差集 DiffSet。此差集便表示了所有调用过旧版本服务，但还未将运行结果返回给 Ingress Gateway 的分布式事务 id 集合。示例如图22所示；

3. 利用差集完成旧版本的撤销：在得到前一步骤的差集结果 DiffSet 后，不再阻塞 Traceid 的相关操作，对于收到的每一个 Traceid，将其从 DiffSet 中删除，表示该分布式事务运行完成并返回结果。当差集 DiffSet 最终删除为空集时，我们便可以做出推断：所有已调用过旧版本服务的分布式事务均已返回结果，旧版本服务实例可被撤销。具体如图23所示；

旧版本撤销与清理操作 完成前述步骤后，我们就可以安全地撤销掉旧版本的所有服务实例，同时对相关的配置进行清理，还原成类似步骤 11.3.2中的初始状态，只是相关的版本从 v1 变更为 v2，为下一次更新操作奠定基础。具体的路由规则、版本规则和系统运行状态分布如图24、25、26所示

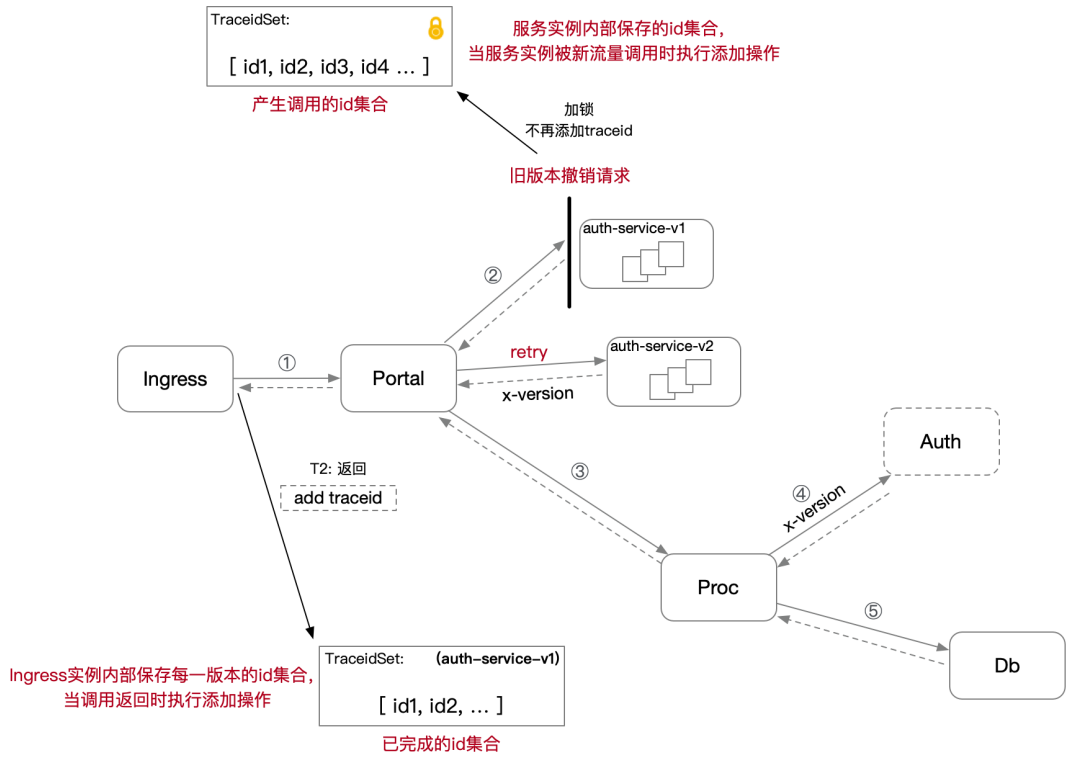

图 21: 阻塞访问旧版本

2 人机物融合平台架构与技术集成

互联网的快速发展催生新技术与新业务的出现。软件从单纯的信息处理环节逐步成为了应用价值观的主要载体，并形成将人、计算机以及物理世界相连接的人机物融合软件新形态。

而当前主流软件技术对人机物开放环境下资源的有效协同的支持存在较明显的不足。人机物融合平台旨在结合软件代理、软件协同等方面的技术创新，兼容地扩展已被广泛实践的主流技术，构建开放协同场景下针对性与实用性较好的技术框架，以灵活地协同管理各种自治的人机物资源，并支撑人机物融合应用系统的运行。

2.1 总体的设计和结构

相较于传统的计算资源管理平台，人机物融合平台除了需要能接入分布于云、网、端的各类人机物资源，还需要制定一套满足人机物资源开放性的资源协作规范，在统一协作规范的基础下，组合实现人机物融合应用的完整逻辑。

融合平台的运行环境基于当前最主流的容器化编排系统 **Kubernetes**，在其上制定了一套人机物资源的描述规范，设计了一种消息驱动的人机物资源协作模型，构建了一套面向人机物融合的资源服务治理框架。

在融合平台中，人机物资源借助于微服务的形式，表示其在现实世界和信息世界中的状态及其具有的能力，称之为资源代理服务。资源代理服务间借助于专用的消息中间件进行通信，以消息驱动的方式进行松耦合的协作。为支持人机物融合应用的复杂使用场景，消息中间件在进行服务治理时，除了需要基于传统的服务质量等指标以外，还需要基于应用描述，综合考虑物理时空与社会关系等属性，生成自适应的消息路由规则。

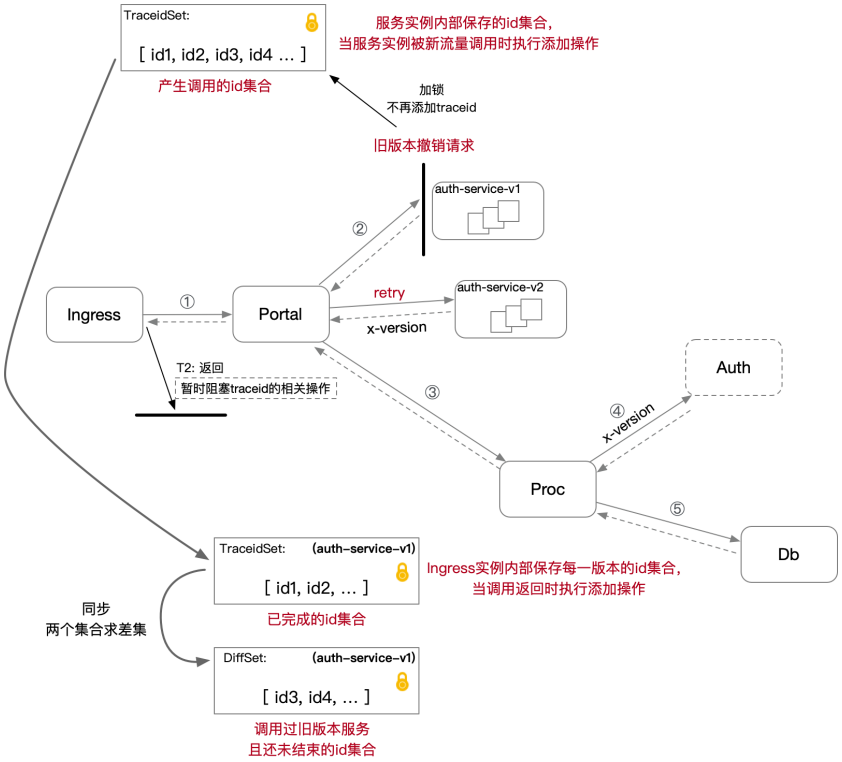

图 22: 同步 Traceid 集合

从传统计算资源管理平台的角度来看，融合平台按照用户提供的人机物资源描述，维护资源代理服务和消息中间件等组件的运行状态；从人机物资源管理平台的角度来看，融合平台向下接入现实世界的各类人机物资源，向上层的资源代理服务与消息中间件，提供相对统一的资源访问接口。

2.2 人机物资源的描述与编排

2.2.1 人机物资源描述规范

2.2.2 人机物资源的运行时编排

2.3 消息驱动的人机物资源协作模型

2.3.1 资源接入层

人机物融合场景下，为适配各类资源（特别是智能设备）连接方式的多样性，提高资源代理服务的可移植性以及通用性，平台提供了统一的资源接入层，资源代理服务通过资源接入层，以相对统一的资源访问接口与人机物资源进行交互。

资源接入层包含访问控制、状态缓存、访问请求映射等功能。资源接入层提供访问控制的功能，以保护敏感资源的访问；资源代理服务可向资源接入层描述自己所关心的资源状态，资源接入层会主动检查资源的状态是否发生变化（取决于资源实际接入平台的访问模式，也支持异步地订阅状态变化通知），如发生变化则通知资源代理服务更新数字对象状态。

对于资源接入层的访问请求映射功能。以智能设备为例，可借助于 **Home Assistant**、**OpenHab** 这类 **IoT** 管理平台进行接入，以提供资源的状态查询与操作接口，融合平台只需将统一的资源访问请求映射为 **IoT** 管理平台对应的请求即可。

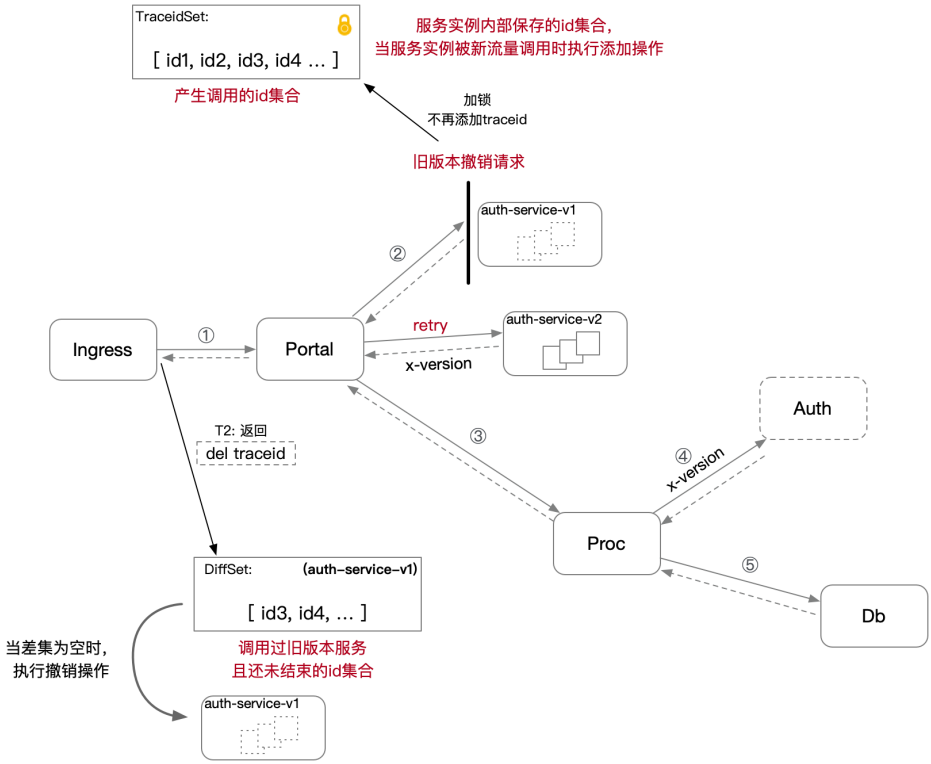

图 23: 利用差集完成旧版本的撤销

除此之外，对于典型的边缘计算等三层网络访问受限的场景，资源代理服务无法直接访问到资源暴露出的操作接口。此时，资源接入层在资源代理服务所处的容器，与智能设备实际接入进程所处容器之间，建立点对点的隧道连接，以突破扁平容器网络模型的限制。

2.3.2 资源代理服务

资源代理服务除了需要提供其状态的描述信息之外，还需提供对应的操作接口，实际上以 RESTful API 的形式描述。目前我们采用传统的 MVC 架构，在 Spring Boot 框架上构建我们的资源代理服务。

资源代理主要分为两类: 智能设备和人。具体实现方面，资源代理服务在我们的平台上是一个一个的 Spring Boot 微服务，下面分别描述智能设备和人的资源代理服务的实现方式。

- **智能设备**

  智能设备的资源代理相对固定，因为智能设备的功能是相对固定的，所以当资源代理部署完成之后，不需要对它的属性和功能接口进行大的修改。以空气净化器为例，在实现资源代理服务时，一方面开发人员对空气净化器进行建模，将空气净化器的属性信息和时空属性加入元模型，并按照资源接入层的规范提供相应的修改查询接口，另一方面开发人员需要封装资源接入层提供的资源互操作接口，生成对应的 RESTful API，使得用户可以通资源代理服务的接口来操控空气净化器。

```
@PostMapping("/turnon")
@Accessible("true")
public String turnOnAirPurifier(){
    RestTemplate restTemplate = new RestTemplate();
    String url = "http://xxx/api/services/fan/turn_on";
```

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: auth-service
spec:
  hosts:
  - auth-service
  http:
- route:
  - destination:
      host: auth-service
      subset: v2
    headers:
      response:
        add:
          x-version: auth-service-v2
```

所有流量发往v2版本，
返回头中添加自定义信息

图 24: v2 路由规则

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: auth-service
spec:
  host: auth-service
  subsets:
- name: v2
  labels:
    version: v2
```

定义v2版本对应的标签

图 25: v2 版本规则

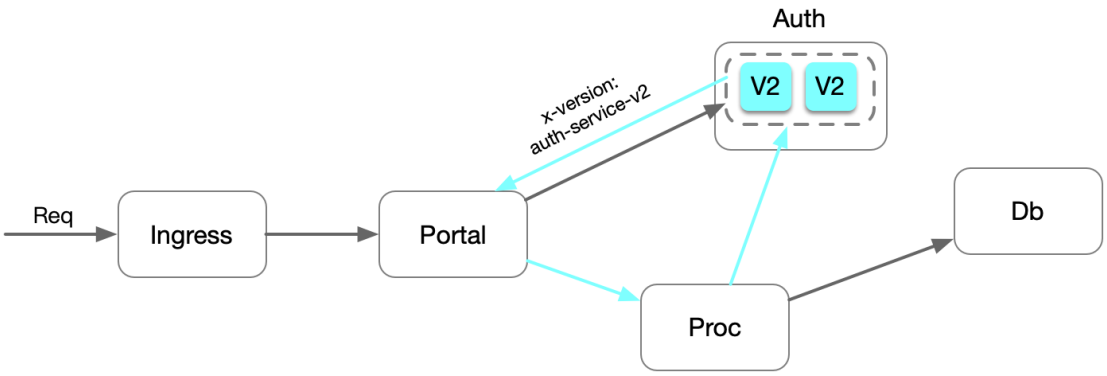

图 26: 流量全部发往 v2

```
        HttpEntity<String> request = new HttpEntity<>(
            new JSONObject().put("entity_id", "fan.xiaomi").toString(),
            new HttpHeaders().setContentType(MediaType.APPLICATION_JSON)
        );
        return restTemplate.postForObject(url, request, String.class);
    }
```

- **人**

  和智能设备相比，人的资源代理服务实现方面有一些区别，人的资源代理服务描述了在某个具体场景下该人所具有的能力，所以随着人的活动，他的资源代理服务会不断更新。举例来说，当人进入一个只有空气净化器的房间时，现实世界中的人就增加了一个使用空气净化器的能力，在这个场景下，人的资源代理服务就会增加操控空气净化器的接口。当这个人从房间离开，进入一个有咖啡机的房间，那么现实世界中的人会失去使用空气净化器的能力，增加使用咖啡机的能力，相应地，人的资源代理服务也会相应更新，删除操控空气净化器的接口，增加操控咖啡机的接口。

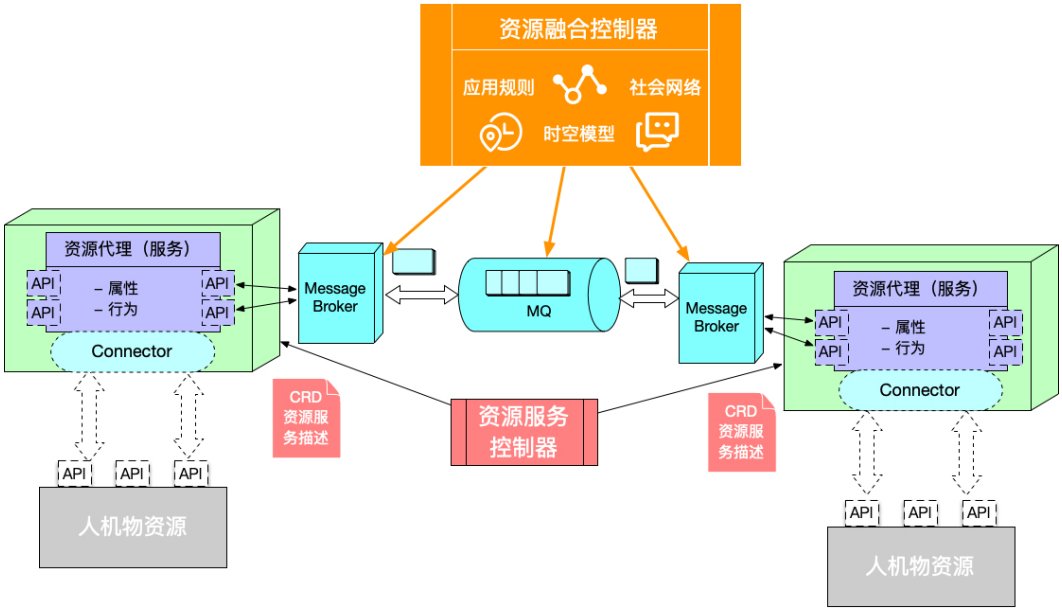

图 27: 资源平台总体架构

2.3.3 消息中间件

通过分析“人机物融合的云计算平台”的实际应用场景，首先我们需要一个松耦合的，消息驱动的服务治理平台来对资源代理服务进行治理，这样才能更好地描述和匹配现实世界中实际的业务场景需要。同时我们考虑将时间空间等现实世界中影响场景实际走向的因素作为通信协议的一部分，通过制定消息路由策略来反应现实世界中的时空属性对服务治理的影响，所以综合以上两点，我们基于 Netty 实现了一个简单的消息中间件来支撑平台的消息通信模块。

- **基于 Java 语言编写，网络通讯依赖 Netty**
  平台中的消息中间件是基于 Netty 实现的，Netty 是一个异步的，事件驱动的，可以用来快速开发可维护的高性能的服务器和客户端的网络应用框架，它极大地简化了网络编程，允许用户从 TCP/UDP 层自定义通信协议。目前比较流行的消息队列，例如 RabbitMQ，ActiveMQ 等都利用 Netty 来支撑项目中的消息通信模块。

- **通过定义 topic 来区分不同类型的资源代理服务**
  当服务绑定到 messageBroker 上时，需要声明自己的 topic，例如不同品牌的咖啡机的 topic 都是 coffeeMachine，这样可以对同种类型的资源聚合，对不同类型的资源进行区分。

- **采用 Kryo 进行消息的网络序列化传输**

- **支持基于时空属性的消息路由策略**
  基于 IndoorGML 建模规范对静态空间进行建模，维护空间中不同资源实体之间的距离信息，根据距离信息构建路由表，定期将最新的路由表发送给消息中间件。

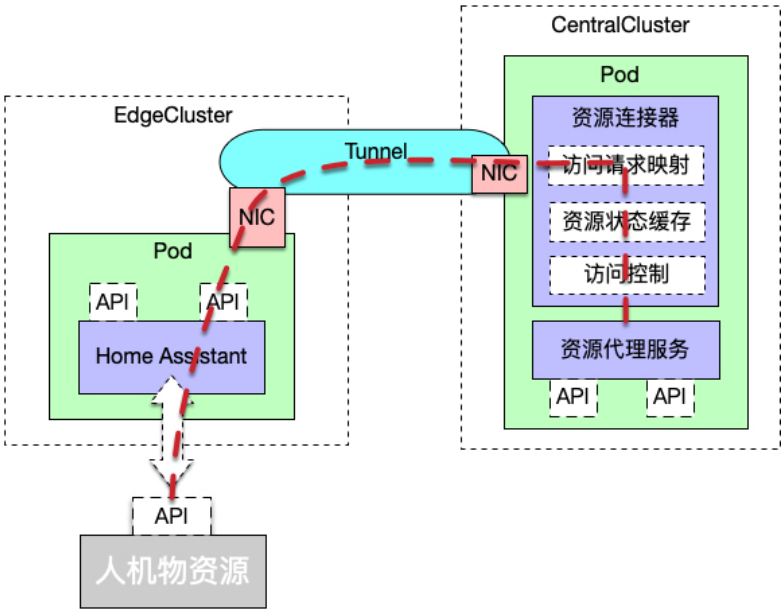

图 28: 资源接入层

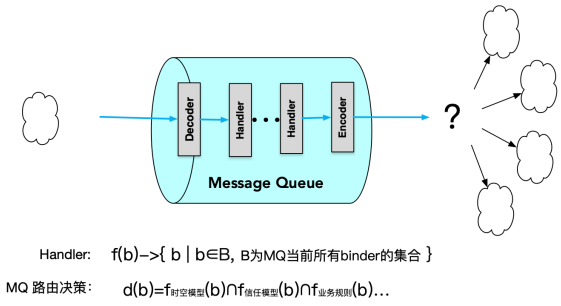

图 29: mq 架构图

消息模型 平台中的消息模型由请求消息和接收消息两种类型，分别由图 2 和图 3 表示。

消息序列化 我们基于 kryo 来实现消息的序列化，主要利用 KryoCodecUtil 和 KryoSerialize 来完成消息的序列化和反序列化工作。

```
public class KryoSerialize {
    private KryoPool pool;
    private Closer closer = Closer.create();
    public KryoSerialize(final KryoPool pool) {
        this.pool = pool;
    }

    public void serialize(OutputStream output, Object object) throws IOException {
        try {
            Kryo kryo = pool.borrow();
            Output out = new Output(output);
            closer.register(out);
            closer.register(output);
            kryo.writeClassAndObject(out, object);
            pool.release(kryo);
```

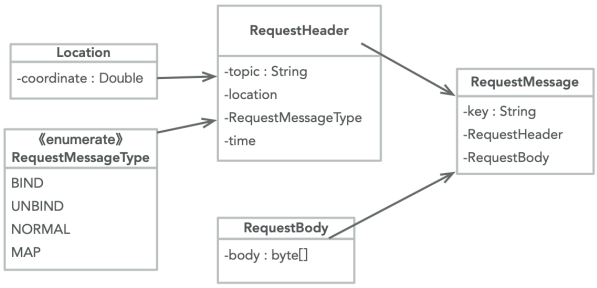

图 30: 请求消息模型

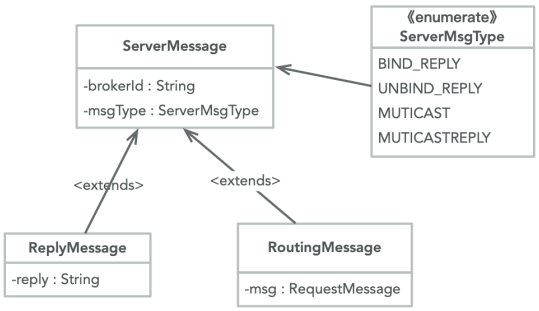

图 31: 接收消息模型

```
        } finally {
            closer.close();
        }
    }
    public Object deserialize(InputStream input) throws IOException {
        try {
            Kryo kryo = pool.borrow();
            Input in = new Input(input);
            closer.register(in);
            closer.register(input);
            Object result = kryo.readClassAndObject(in);
            pool.release(kryo);
            return result;
        } finally {
            closer.close();
        }
    }
}
```

将 kryo 作为成员变量加入 Netty 的 ByteToMessageDecoder 和 MessageToByteEncoder 中，利用上面我们实现的 kryo 序列化和反序列化方法实现 Netty 中消息的编码和解码。

```
public class RequestDecoder extends ByteToMessageDecoder {
    final public static int MESSAGE_LENGTH = MessageCodecUtil.MESSAGE_LENGTH;

    private MessageCodecUtil util; //kryo
    public RequestDecoder(MessageCodecUtil messageCodecUtil) {
        this.util = messageCodecUtil;
```

```
    }
    @Override
    protected void decode(ChannelHandlerContext ctx,
                          ByteBuf in, List<Object> out) throws Exception {
        ......
        Object obj = util.decode(messageBody);
        ......
     }
}

 public class ResponseDataEncoder extends MessageToByteEncoder<Object> {
    private MessageCodecUtil util;
    ResponseDataEncoder(MessageCodecUtil messageCodecUtil) {
        this.util = messageCodecUtil;
    }
    @Override
    protected void encode(ChannelHandlerContext ctx, Object msg
                          , ByteBuf out) throws Exception {
        util.encode(out, msg);
    }
}
```

MessageBroker 实现

- **客户端管理**
  我们定义一个 ClientInfo 对象来描述注册在消息中间件上的客户端信息。

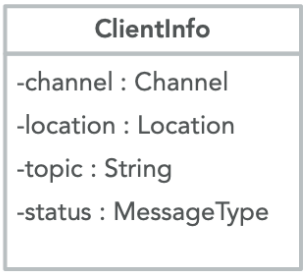

图 32: ClientInfo 类图

  channel 表示客户端通信的通道信息，location 记录该 client 对应的资源代理服务的位置信息，topic 代表该 client 对应的资源代理服务的类型，status 表示该 client 目前的连接状态。

  我们用 ConcurrentHashMap 在内存中存放所有的 ClientInfo。当有 client 发送 bind 请求时，messagebroker 会先遍历 ConcurrentHashMap，然后将新的 client 加入 map 中；相应地，当有 client 发送 unbind 请求时，messagebroker 会先遍历 ConcurrentHashMap，然后将对应的 client 状态改为 DEAD；当客户端发送常规消息时，会查看 location 信息，保持 location 最新。

- **消息路由**
  当服务绑定到 messageBroker 上时，需要声明自己的 topic，例如不同品牌的咖啡机的

topic 都是 coffeeMachine，这样可以对同种类型的资源聚合，对不同类型的资源进行区分。

Producer/Consumer 实现 消息队列中的 Producer 发送消息，Consumer 接收消息并对消息处理。其实在大部分情况下，Producer 也需要接收消息，Consumer 也需要发送消息，所以我们这里对他们一并看待，统称为 Client，我们的 Client 也是基于 Netty 实现的，区别是在启动时，我们只给 Client 的 workGroup 分配了一个线程，所以可以理解为 Client 只能通过唯一的 channel 发送消息。我们给 Client 封装了 ClientMessageSender 来发送消息，ClientMessageHandler 来接收消息。

- **绑定/解绑**

```
public synchronized void bind(String key,RequestHeader requestHeader
                    ,RequestBody requestBody){
        ……
        channelFuture.channel().writeAndFlush(bindMessage);
        notify();
}

public synchronized void unbind(String key,RequestHeader requestHeader
                      ,RequestBody requestBody){
        ……
        channelFuture.channel().writeAndFlush(unbindMessage);
        try {
             channelFuture.channel().closeFuture().sync();
        } catch (InterruptedException e) {
             e.printStackTrace();
        }
}
```

- **发送业务消息**

```
public void sendMessage(RequestHeader requestHeader,RequestBody requestBody){
    try {
       wait();
    } catch (InterruptedException e) {
       e.printStackTrace();
    }
    ……
    channelFuture.channel().writeAndFlush(requestMessage);
}
```

- **接收消息**

```
/**
 * 继承ServerMessageHandler类，用来接收mq发送来的消息
 * 其中有一个channel变量，调用channel.writeAndFlush()方法可以向mq发送答复消息
 * responseMessage()对应于sendMessage()方法
 * ackMessage()对应于bind()和unbind()方法
 */
public class ResponseMessageReceiver extends ServerMessageHandler {
 @Override
```

```
    public void responseMessage(RequestMessage requestMessage) throws
        InterruptedException {
      ... ...
    }

    @Override
    public void ackMessage(String msg) {
      ... ...
    }
}
```

2.4 面向人机物融合的资源服务治理框架

2.4.1 资源服务发现

资源服务是只有资源属性和资源能力的微服务，对外提供 Restful API，我们的平台会为每一个资源服务生成一个消息适配器来支持资源服务发现，消息适配器的作用是自动地为资源服务添加基于消息中间件的消息通信模块，使得资源代理中的 Restful API 中的业务规则能在平台中以消息的形式在资源服务之间传递。总结来说，在保留资源代理本身的业务逻辑的同时，支撑平台的资源服务发现。

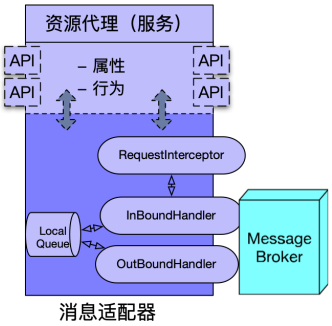

图 33: 消息适配器架构图

MSG-API Mapping Table

MSG	API
turn on	api/turnon
turn off	api/turnoff
{speed : xxx}	api/speed
{humidty: xxx}	api/humidty
.....	.....

图 34: “消息-接口”映射表

- **整体架构**

  消息适配器主要由 InBoundHandler，OutBoundHandler，LocalQueue 三部分组成，

InBoundHandler 会将从消息中间件中接收到的业务消息存入 LocalQueue 中，OutBoundHandler 会监听 LocalQueue，如果 LocalQueue 非空，就会根据用户事先定义的”消息-接口映射表”，调用对应的 Restful API 或消息进行处理，并将处理完成的 Ack 发送给消息中间件。对于人的资源服务则需要单独考虑，人的资源代理往往作为整个消息流程的发起者，除了 InBoundHandler，OutBoundHandler，LocalQueue 三部分之外，人的资源代理服务的消息适配器需要为人动态生成 Restful API 来发起整个消息流程。换言之，人的资源代理的 Restful API 的作用是发起某个消息流程。

- **InBoundHandler 实现**
  InBoundHandler 的实现比较简单，直接继承 mq client 的 ServerMessageHandler 类，重写 ServerMessageHandler 的 responseMessage 方法，该方法会接收到 mq 发送过来的消息，将接收到的消息存入 Local Queue 中，并回复 mq 消息已经收到。

```
@Override
 public void responseMessage(RequestMessage requestMessage) throws
      InterruptedException {
     try {
         String s = requestMessage.getRequestBody().getBody();
         messageQueue.add(s);
         this.notify();
         ......
         this.channel.writeAndFlush(reply);
         ...
     } catch (UnsupportedEncodingException e) {
         e.printStackTrace();
     }
 }
```

- **OutBoundHandler 实现**
  OutBoundHandler 用于发送消息，OutBoundHandler 的成员变量中需要一个资源代理的 RestController 实例，此外还需要消息和接口之间的映射表 (图 8)。这样当 Local Queue 非空时，每次获取一个请求信息，可以找到对应的消息内容，将消息通过发送给 Consumer，当获取到的是消息时，通过找到对应的请求信息，调用 RestController 实例中的对应的接口执行相应的业务代码。

- **人的资源代理 Restful API 实现**
  对于现实世界中人可以使用的智能设备，在平台中，这些资源的资源代理会向消息中间件发送心跳消息来表示它们的占用情况，当人的资源代理启动之后，当接收到这些资源代理发送的广播消息，资源代理适配器会为人生成对应的 Restful API 来表示人目前具有的能力，这些接口中的代码逻辑比较简单，主要是生成调用不同资源代理的消息，并将消息加入 LocalQueue 中，这样当这些 Restful API 被请求时，OutBoundHandler 就会发出 LocalQueue 中的消息，整个消息流程启动。

2.4.2 资源服务路由

资源服务的路由需要考虑多种因素，除了服务治理中经常用到的流量，服务负载，响应时间等因素之外，我们还需要综合考虑时空模型，用户社会网络等现实世界中的因素。举例来说，当人在一栋楼中想要打印一份文件时，如果他想要最快得到打印的文件，那么消息路由中距离因素和服务占用情况将会占据消息路由的主要权重，而如果他更看重价格而不是

时间，那么价格因素将会占据消息路由的最大权重。所以平台中的资源服务路由将会综合现实世界和信息世界中的各种可能影响路由决策的因素。以”距离最近”策略为例，首先开发

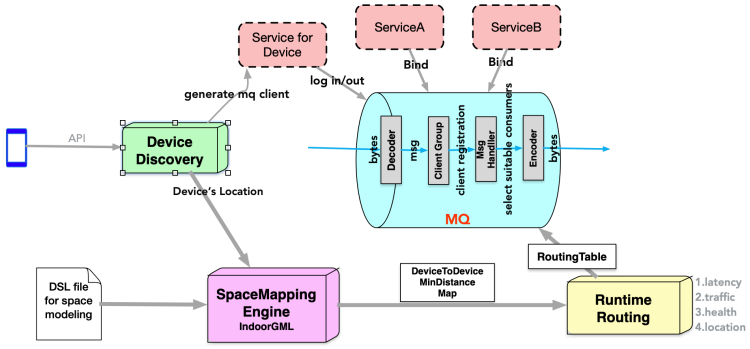

图 35: 消息路由架构图

人员需要向平台中的空间映射引擎传入一个配置文件来对整个场景进行空间建模，这样空间映射引擎就可以得到整个空间中每个子空间之间的距离信息，当资源服务注册到平台中时，资源服务的位置信息也会被注入到空间映射引擎中去，这样平台就可以得到平台中资源服务之间的距离信息，将距离信息存入一个哈希表中发送给运行路由模块，运行时路由模块中根据开发人员传入的策略信息，会生成不同策略下的路由表，如“就近优先”表，“成本最低”表等，运行时路由模块会定时向消息中间件推送这些路由表，消息中间件中的路由表会定期更新。//todo

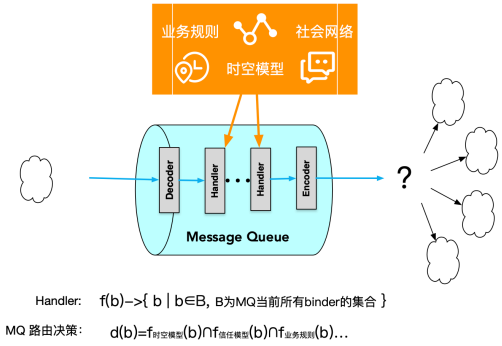

图 36: 消息路由

2.4.3 资源服务的动态更新

参考文献

[1] P. Erdős, *A selection of problems and results in combinatorics*, Recent trends in combinatorics (Matrahaza, 1995), Cambridge Univ. Press, Cambridge, 2001, pp. 1–6.