

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 9 日

摘 要

请在这里输入摘要内容.

版 权 声 明

该文件受《中华人民共和国著作权法》的保护。ERCESI 实验室保留拒绝授权违法复制该文件的权利。任何收存和保管本文件各种版本的单位和个人，未经 ERCESI 实验室（西北工业大学）同意，不得将本文档转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍著作权之问题，将可能承担法律责任。

目 录

1	技术背景	6
2	微服务动态演化一致性保障协议	7
2.1	基本原理	7
2.2	一致性保障协议	8
2.3	微服务动态演化过程	10
3	基于 Istio 的协议实现	12
3.1	Service Mesh 与 Istio	12
3.2	协议实现具体流程	14
3.2.1	插入 Envoy 容器和 TraceManager 容器	14
3.2.2	创建初始的路由规则和版本信息	14
3.2.3	新版本上线	15
3.2.4	路由规则更新	17
3.2.5	旧版本撤销请求	18
3.2.6	旧版本撤销与清理操作	20
4	人机物融合平台架构与技术集成	21
4.1	总体的设计和结构	21
4.2	人机物资源的描述与编排	21
4.2.1	人机物资源描述规范	22
4.2.2	人机物资源的运行时编排 (WHG)	22
4.3	消息驱动的人机物资源协作模型	23
4.3.1	资源状态的同步	23
4.3.2	资源代理服务 (TC)	23
4.3.3	消息队列 (TC)	24
4.4	面向人机物融合的资源服务治理框架	24
4.4.1	人机物融合场景下的服务路由 (TC)	24
4.4.2	资源服务的动态更新 (WDY)	24
5	动态更新系统实现	24
5.1	基本设计概念和结构	24
5.2	代码管理器	24
5.2.1	模块描述	24
5.2.2	功能	24
5.3	构建管理器	24

5.4 更新控制器	24
---------------------	----

1 技术背景

2 微服务动态演化一致性保障协议

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。

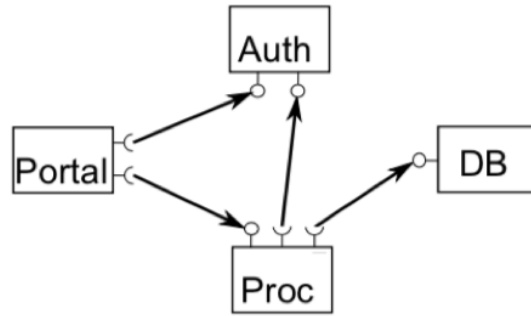


图 1: 示例系统

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。

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 2.1 (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。

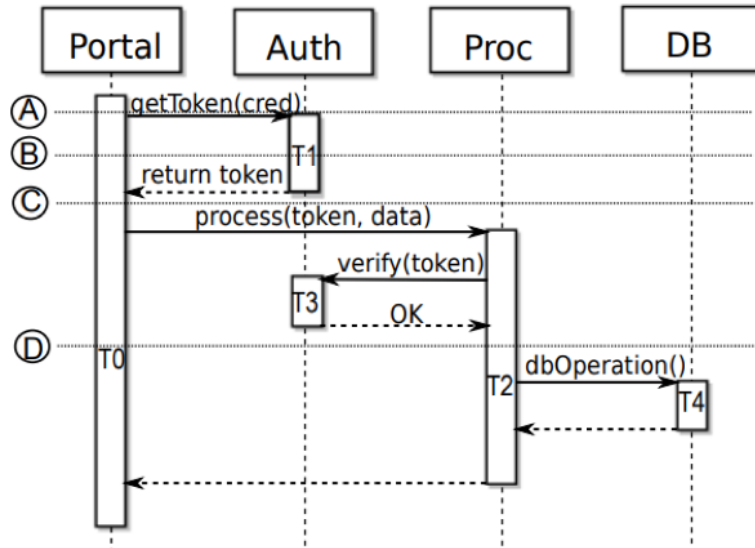


图 2: 示例系统运行时序图

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。

Definition 2.2 (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)。

Definition 2.3 (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：

Definition 2.4 (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。

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3;

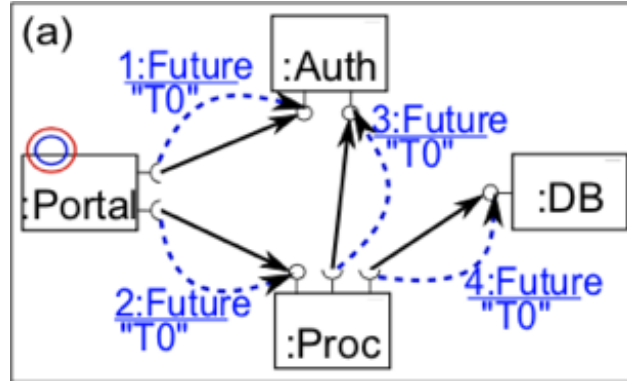


图 3: 准备阶段: process_a

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4;
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 T_0 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 T_0 的一对 future/past 边, 因此不满足更新条件。

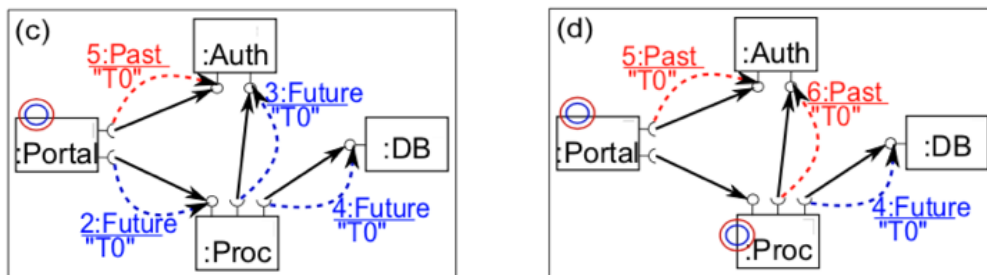


图 4: 运行阶段: process_c 和 process_d

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 $\text{root}(T)$ 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 $\text{root}(T)$ 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。

3 基于 Istio 的协议实现

3.1 Service Mesh 与 Istio

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。

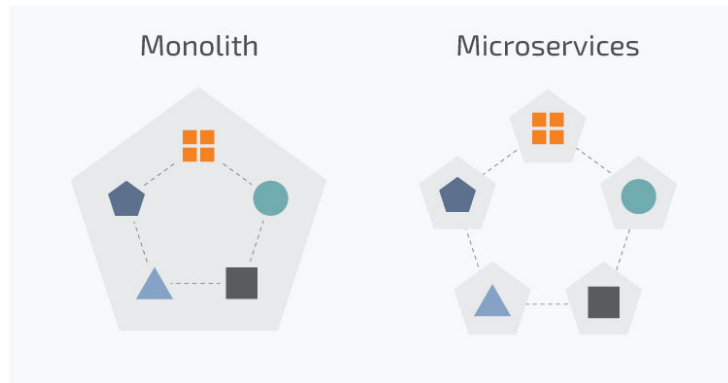


图 5: 单体应用和微服务应用

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6所示。

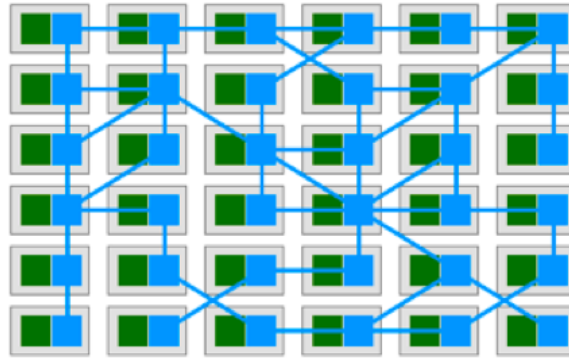


图 6: 服务网格 (Service Mesh)

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。

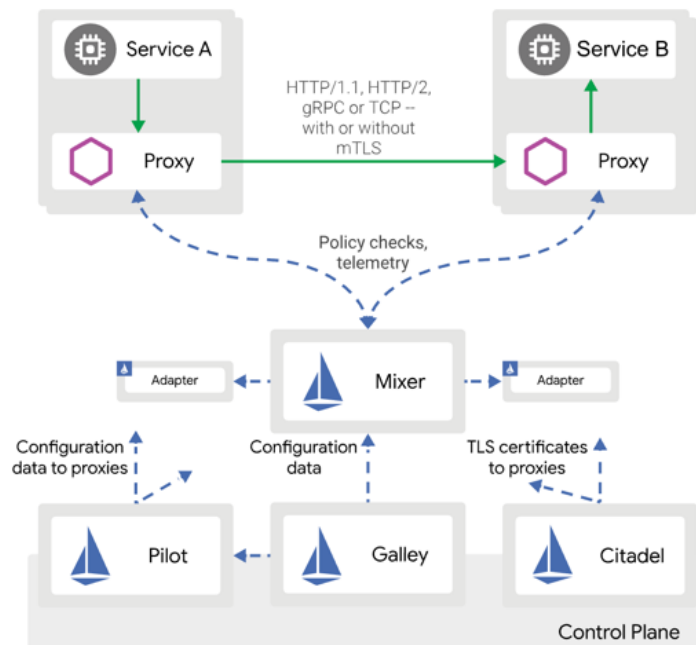


图 7: Istio 架构图 (v1.4)

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8。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 Gateway 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。

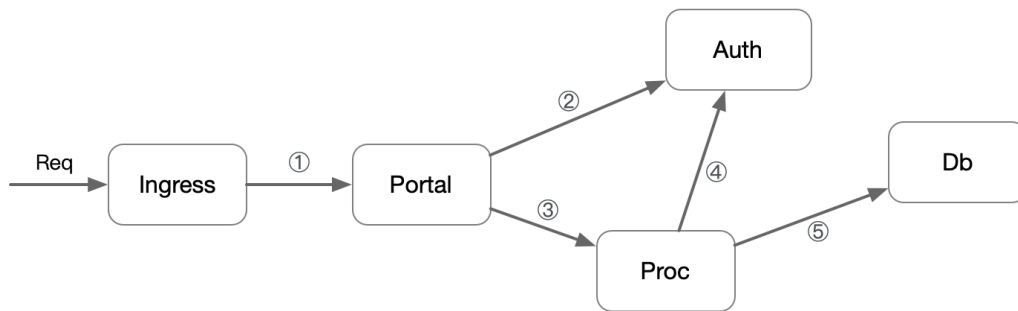


图 8: Istio 中的示例系统

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

3.2.1 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。

3.2.2 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10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；

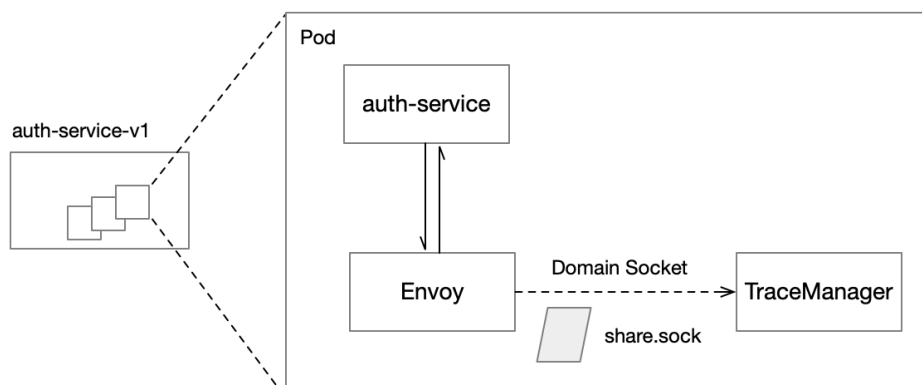


图 9: 插入容器

图11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12

```
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 10: v1 路由规则

```
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 11: 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。

3.2.3 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14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，

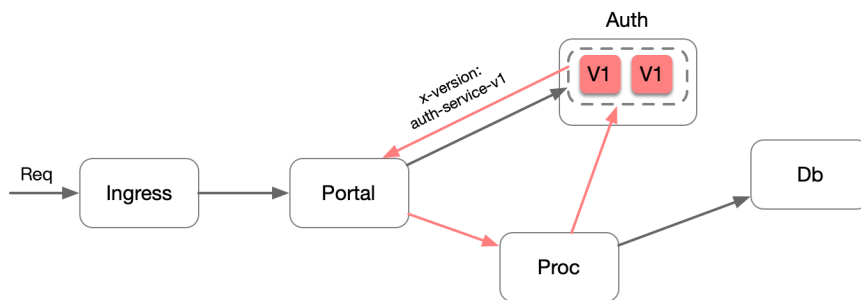


图 12: 流量全部发往 v1

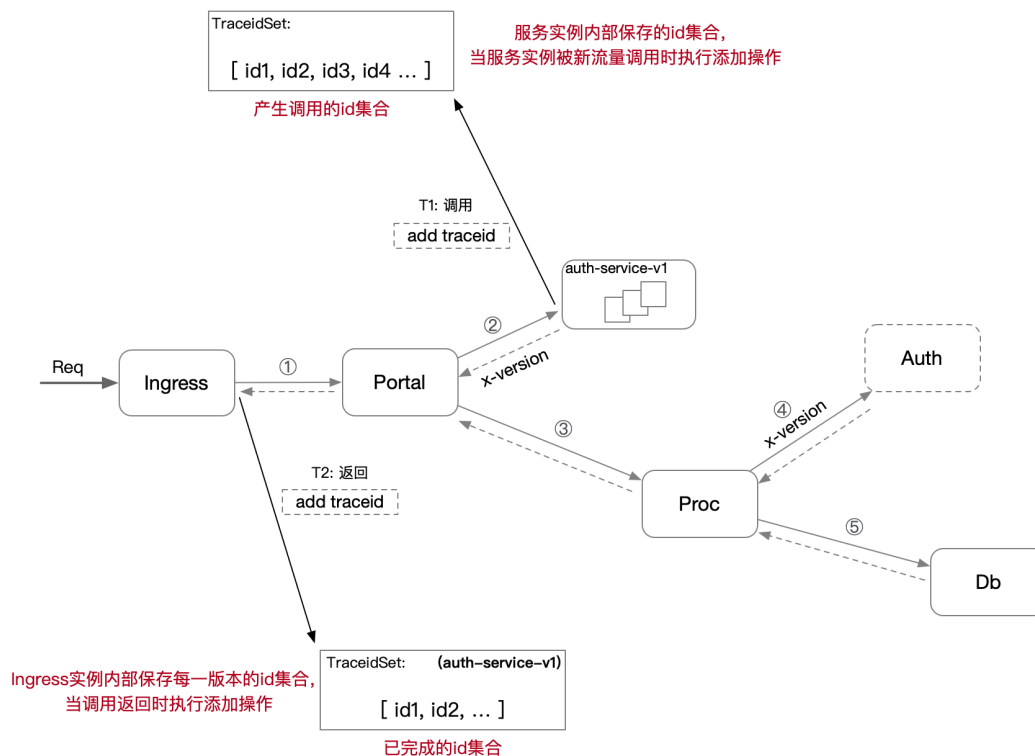


图 13: 事务记录

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15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，但还没有流量经过。对应的系统运行状态如图16所示

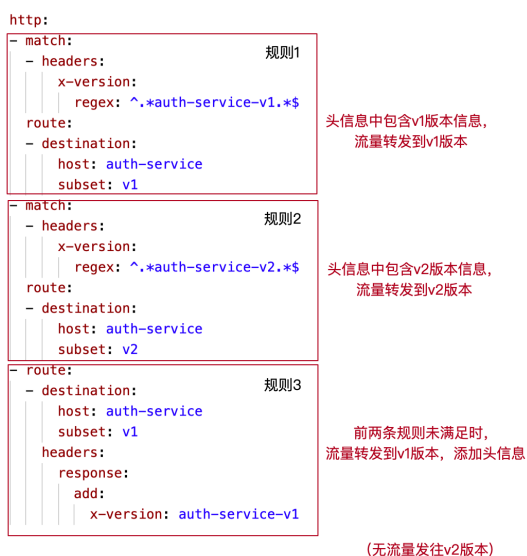


图 14: 默认发往 v1 路由规则



图 15: v1、v2 版本规则

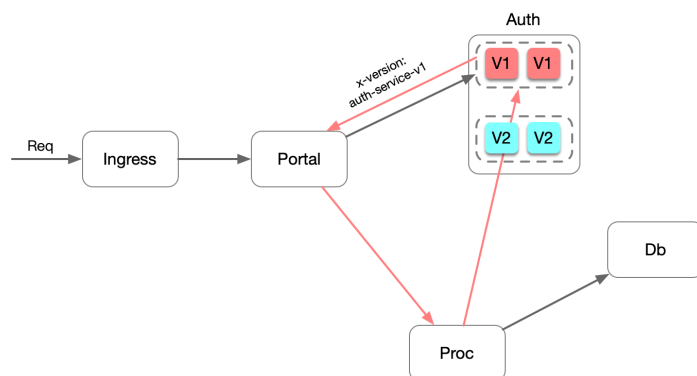


图 16: 流量默认发往 v1

3.2.4 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17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。

Lemma 1. 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。

Proof. 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3.2.4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。 □

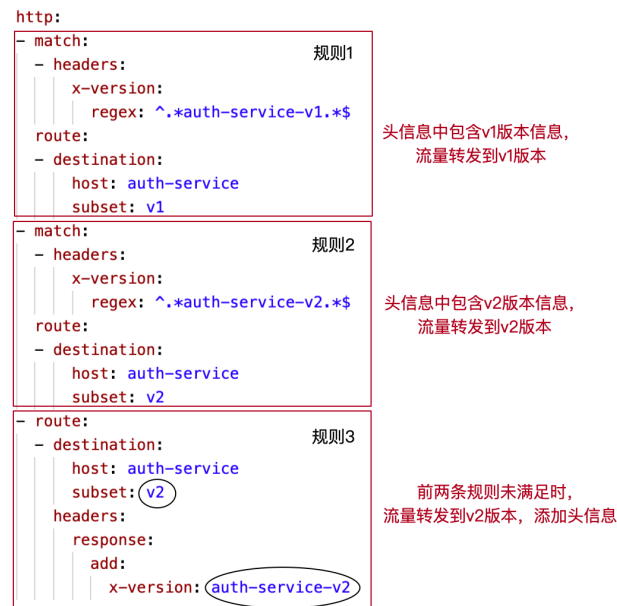


图 17: 默认发往 v2 路由规则

更新完路由规则对应的系统运行状态如图18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。

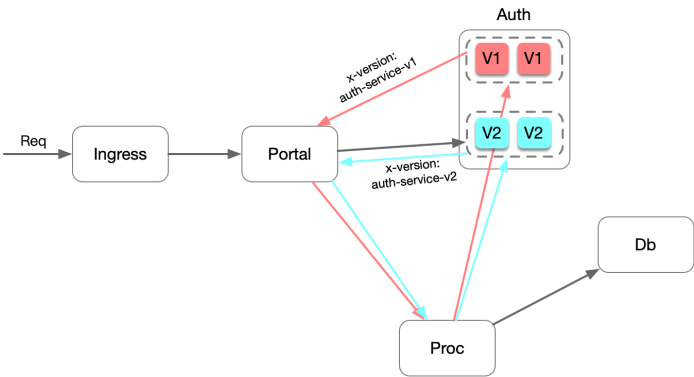


图 18: 新流量发往 v2

3.2.5 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19所示；

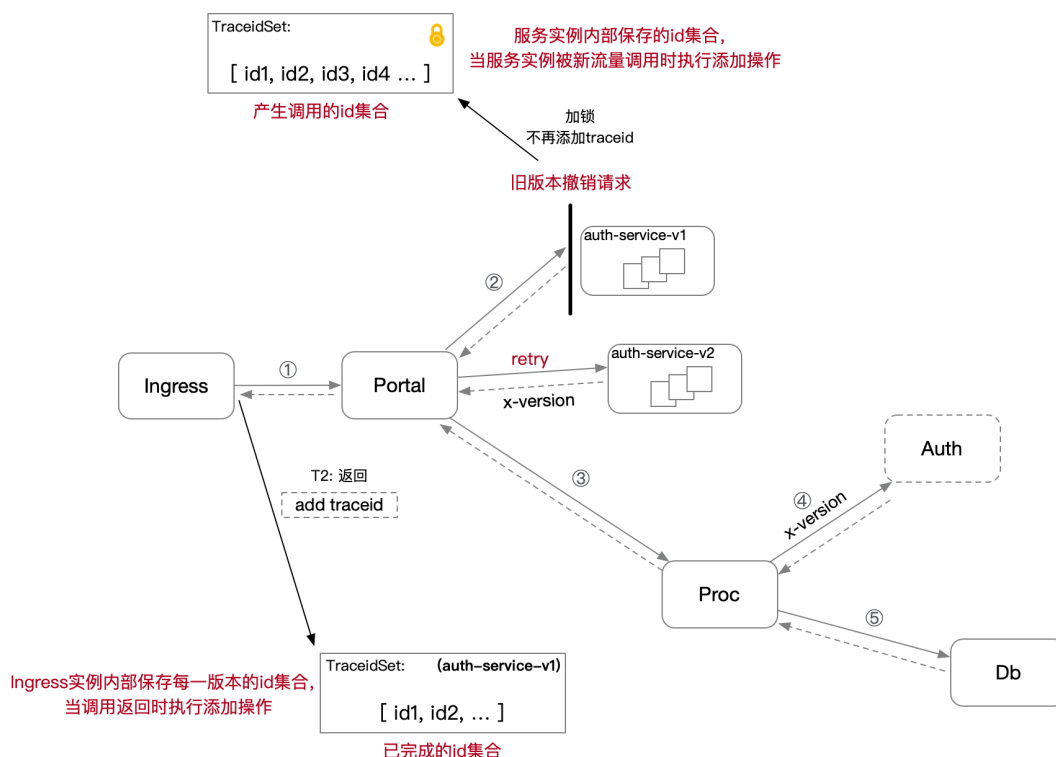


图 19: 阻塞访问旧版本

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 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20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21所示；

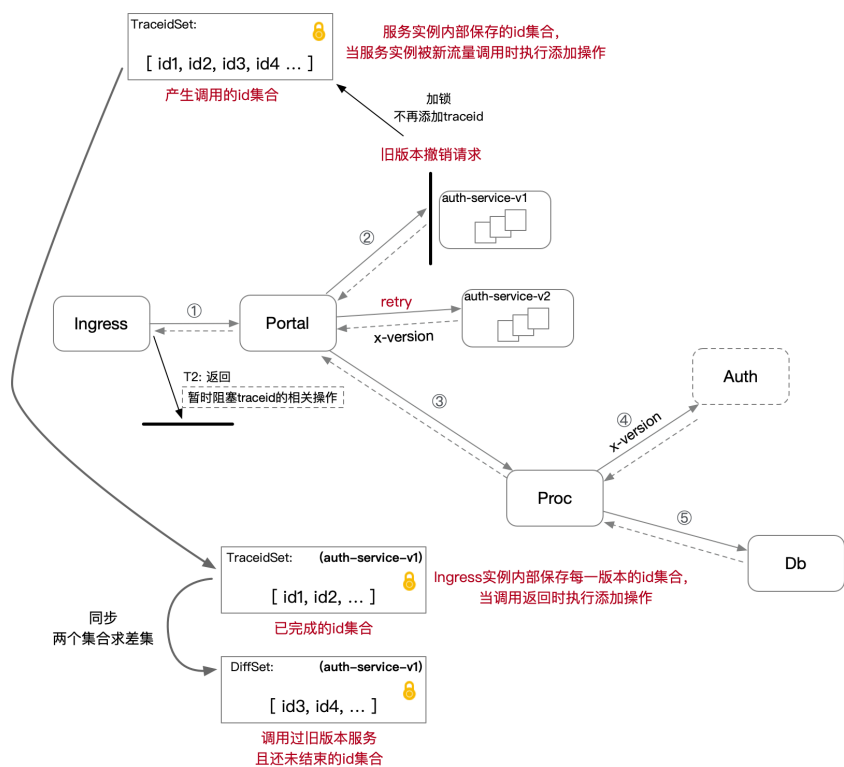


图 20: 同步 Traceid 集合

3.2.6 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 13.2.2 中的初始状态，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22、23、24所示

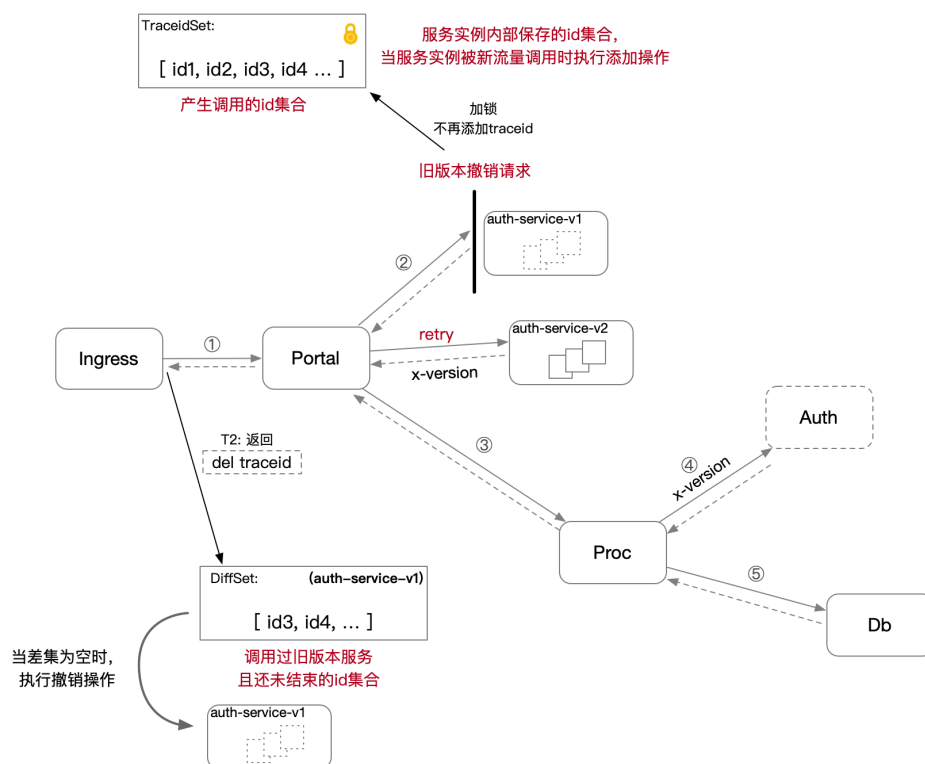


图 21: 利用差集完成旧版本的撤销

4 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。人机物融合平台旨在人机物三元融合开放多变的环境下，灵活地协同管理各种自治的资源，支撑人机物融合应用系统的运行。

4.1 总体的设计和结构

人机物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。

4.2 人机物资源的描述与编排

实际的物理资源都要抽象成一种数据结构存储在系统中，系统同时要更新和维护它们。本软件平台通过 Kubernetes CRD 在系统中表示抽象资源，存储在 etcd 中，Kubernetes Controller 负责维护它们，同步状态。

```

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 22: v2 路由规则

```

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 23: v2 版本规则

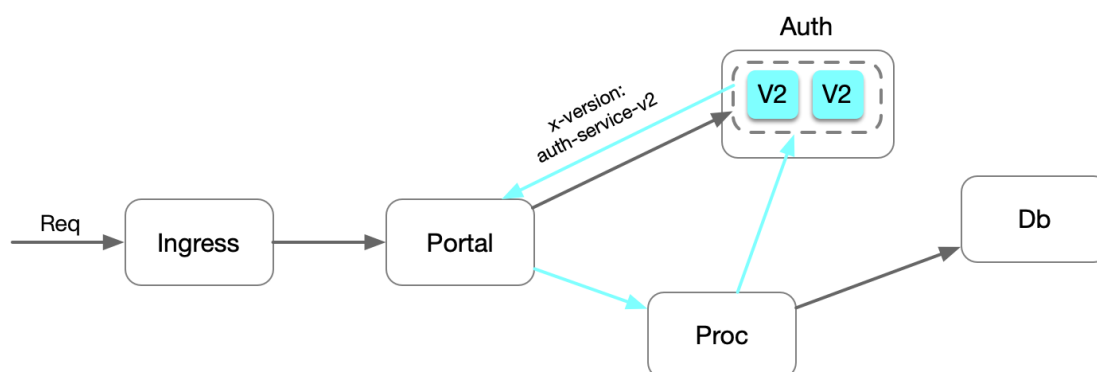


图 24: 流量全部发往 v2

4.2.1 人机物资源描述规范

CRD 是 CustomResourceDefinition 的简写，资源是 kubernetes 中的抽象概念，CRD 是自定义的资源。

所有物理资源都要在系统中注册，且在系统中表达为 Resource CRD，需要使用一些参数描述资源。这些参数告诉了 Kubernetes 平台如何去访问资源并维护资源的状态。首先，资源需要一个唯一的标识。其次，因为需要使用探针去采集资源的各项信息，需要包含探针程序的镜像地址和启动命令以及启动参数。

4.2.2 人机物资源的运行时编排 (WHG)

Resource CRD 由 Resource Controller 管理，控制器监控资源的状态，当监测资源的增删改事件时就会生成一个包含相关资源名称的 request 塞入工作队列，Reconcile 方法每隔很短的一段时间被调用一次，从工作队列中取出一个 request 作为参数进行处理。

Reconcile 首先根据 request 中包裹的资源名称和命名空间找到 Resource 的实例，如果是删除事件什么都不需要做，增改事件则会根据 ProbeEnabled 字段判断是否需要新建、更新、删除探针 Deployment 资源。

4.3 消息驱动的人机物资源协作模型

4.3.1 资源状态的同步

4.3.2 资源代理服务 (TC)

资源代理描述 资源代理在我们的平台中是以微服务的形式描述现实世界中资源的一些基本信息和它具有的能力，资源代理主要分为两类：智能设备和人。资源代理的基本信息除了包括 id, 品牌和它的一些基本属性之外，还包括它的时空属性信息；资源代理的能力实际上表示为 Restful API 定义的功能接口。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。

资源代理实现 具体实现方面，资源代理在我们的平台上是一个一个的 Spring Boot 微服务，资源代理的实现需要探针的配合，下面分别描述智能设备和人的资源代理的实现方式。

- 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。以空气净化器为例来介绍资源代理的实现。首先我们需要将空气净化器连接到 Home Assistant 这样的智能设备管理平台上，这样开发人员就可以从 Home Assistant 上获得空气净化器的属性信息和操控接口，相应地，在空气净化器的资源代理中，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，并提供相应的修改查询接口给探针调用，另一方面开发人员需要封装 Home Assistant 提供的操纵接口，生成对应的 Restful API，使得用户可以通过 Restful API，借助 Home Assistant 来远程操控空气净化器。

```
%    @PostMapping("/turnon")
    @Accessible("true")
    public String turnOnAirPurifier(){
        RestTemplate restTemplate = new RestTemplate();
        String url =
            "http://192.168.31.143:8123/api/services/fan/turn_on";
        HttpHeaders headers = new HttpHeaders();
        headers.add("Authorization", "xxxxx");
        headers.setContentType(MediaType.APPLICATION_JSON);
        JSONObject jsonObject = new JSONObject();
```

```
        jsonObject.put("entity_id","fan.xiaomi_miio_device");
        HttpEntity<String> request = new
            HttpEntity<>(jsonObject.toString(),headers);
        return
            restTemplate.postForObject(url,request,String.class);
    }
```

- 人

人的资源代理和智能设备的资源代理在实现方面有一些区别，人的资源代理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 would 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。

4.3.3 消息队列 (TC)

4.4 面向人机物融合的资源服务治理框架

4.4.1 人机物融合场景下的服务路由 (TC)

4.4.2 资源服务的动态更新 (WDY)

5 动态更新系统实现

5.1 基本设计概念和结构

5.2 代码管理器

5.2.1 模块描述

5.2.2 功能

5.3 构建管理器

5.4 更新控制器