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

目 录

1 概述

上述思路能大体上将所有的目标对象引用链找出，但仍存在一些问题。由于运行时堆模型中记录的对象数目庞大，因此整个运行时堆模型中对象之间的引用关系是非常复杂的，不断地遍历引用了对象的新对象，可能经过若干轮次后这一流程仍没法终止，最终导致程序在时间和空间上都消耗过度的资源甚至崩溃。因此，方法因限制遍历引用关系的深度。根据开发人员的实际开发经验，构造对象的有效引用链时，一般链长不会特别长，因此方法限制遍历最深层次为 8 层，亦即引用链最长只能包含 9 个对象。这一限制能有效限制方法执行的时间与空间。

另一方面，对象之间可能存在循环引用。普通的循环引用即两个对象相互引用对方，更广义的循环引用则是若干个对象之间的引用形成了一条引用链，同时链尾元素又引用了链头部元素，形成了一个引用环。即使限定了遍历的最深层次，引用环的存在仍使得引用链寻找方法的额外开销非常大，在环中的遍历步骤都是无效的。为了消除寻找过程中的引用环，方法会实时记录当前引用链中已经使用的对象，如果遍历过程中遇到了已经使用过的对象，则跳过该对象。解决了对象引用链生成过程中遇到的问题后，方法能够较快速地正确生成所有最长长度限制以内的对象引用链。对于每一条引用链，其长度越短，其生成代码的可用性可能越高，同时生成代码的难度越小。因此方法将所有得到的引用链按长度从短到长排序，提供给开发人员。开发人员需要选定一条对象引用链，作为之后生成对象构造代码的基本材料，其选择时应尽量避免引用链中存在可能会出现构造错误或者构造后不可用的对象，如 UI 相关类型的对象。

对于开发人员选定的目标对象引用链，构造代码是从链头的类对象开始的。在网构软件行为反射技术框架中，构造出的代码最终会运行在应用类加载器环境下，如果需要使用一个类加载器不是应用类加载器的类的相关内容，则需要在构造该类时指定正确的类加载器。运行时堆模型中记录了每个类对象的类加载器，若链头的类对象的类加载器不是应用类加载器，方法需要构造出该类对象的类加载器。类加载器在堆中也是一个普通对象，其同样可以通过不断遍历引用了对象的新对象的引用链构造方法得出。构造出类加载器后，开发人员可以得到一条完整的引用链，以此为基础进行目标对象构造的代码生成。

根据开发人员之前选择的完整的对象引用链，可以生成对象构造代码。由于方法的目标是不依赖任何多余内容的前提下构造出对象，相比起直接显示地构造对象，方法选择使用 Java 的反射机制来生成代码。使用 Java 反射机制有两个好处，一是不需要依赖应用的 jar 包来识别类型和方法调用，二是在构造引用链头部的类对象时，能方便地指定类加载器。

这一节请主要描述组成原理课程（cs11007）中所介绍的单周期处理器的基本结构及工作机理。Latex 中文添加了 ctex 包，请使用 xelatex 编译该文件，即在对应文件夹命令行输出，xelatex ReportTemp_ch_xelatex.tex 可使用分列项目（itemize）的方法说明不同的属性介绍。例如：

- 处理器支持的指令包括：*sub, add, or, ori, lw, sw, lst, beq, j*。
- 所有指令在一个周期内完成在目前阶段，单周期处理器模型讲计算、控制通路分开设计，为理解处理器结构提供了良好的支持。
- 本设计包含处理器数据通路、控制逻辑以及存储结构为了说明计算机系统结构的特点，我们推荐奖处理器的数据通路、控制通路分开设计。
- 使用 **Chisel3** 描述结构 Chisel 是一种强大的结构化硬件描述语言，可以对模块、接口

以及操作等进行高效率的描述，与 Verilog 语言相比较，对电路的结构属性具有更好的封装，对系统的描述更加简化。我们选择 Chisel3 是因为在 Linux 系统中，使用 Chisel3 完全不再依赖工业 EDA 软件实现电路的逻辑仿真，更有利于学术研究和教学应用。虽然 Chisel 语法在不同版本差异不大，但是部分修饰符的描述方式仍然具有差别，详细区别请参考: https://github.com/ucb-bar/chisel3/wiki/Chisel3-vs-Chisel2.

2 系统设计

2.1 System Overview

请在这一节描述单周期处理器的系统设计。推荐使用图表进行说明的方法，Latex 中插入图并进行索引的方式如下：图1.

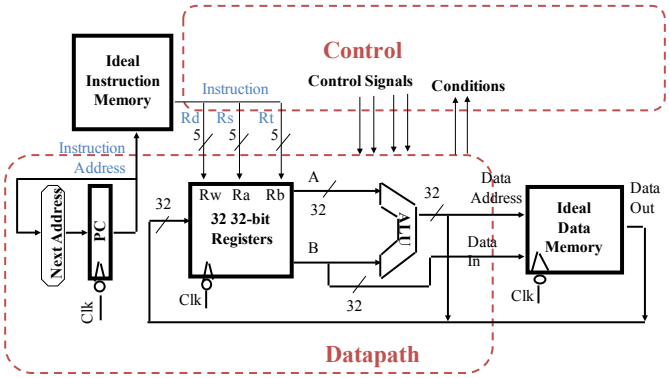

图 1: 单周期处理器结构图

2.2 接口定义

请详细描述系统顶层接口信号的名称、逻辑特征、时许特性。可以建立类似表1这样的列表进行说明。

2.3 接口的（时许）逻辑

请在这个小节描述顶层结构中接口信号使用的逻辑、时序关系等。同样请利用图、表的方法进行描述。特别是时序，请画出时许图。

3 模块详细设计

这一节，主要描述各个模块的功能、接口、逻辑控制方法（状态机控制方法）等。

表 1: 测试模式信号定义

信号名	方向	位宽	功能描述
boot	Input	1-bit	触发测试模式, 当处理器正常工作时被置为 0
test_im_wr	Input	1-bit	Instruction memory write enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if writing instructions to imem, otherwise it is set to 0.
test_im_re	Input	1-bit	Instruction memory read enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if reading instructions out, otherwise it is set to 0.
test_im_addr	Input	32-bit	Instruction memory address
test_im_in	Input	32-bit	Instruction memory data input for test mode.
test_im_out	Output	32-bit	Instruction memory data output for test mode.
test_dm_wr	Input	1-bit	Data memory write enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if writing data to dmem, otherwise it is set to 0.
test_dm_re	Input	1-bit	Data memory read enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if reading data out, otherwise it is set to 0.
test_dm_addr	Input	32-bit	Data memory address
test_dm_in	Input	32-bit	Data memory input for test mode.
test_dm_out	Output	32-bit	Data memory output for test mode.
valid	Output	1-bit	If CPU stopped or any exception happens, valid signal is set to 0.

3.1 ALU

3.1.1 功能描述

3.1.2 接口定义

3.1.3 逻辑控制

3.2 控制模块

3.2.1 功能描述

3.2.2 接口定义

3.2.3 逻辑控制

3.3 数据通路

3.3.1 功能描述

3.3.2 接口定义

3.3.3 逻辑控制

4 基于 Istio 的协议实现

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。

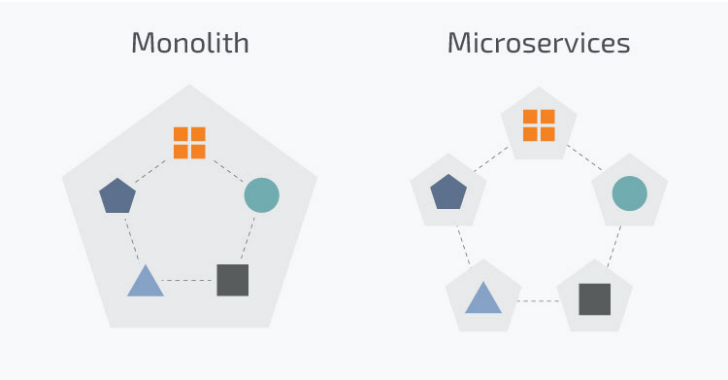

图 2: 单体应用和微服务应用

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)。

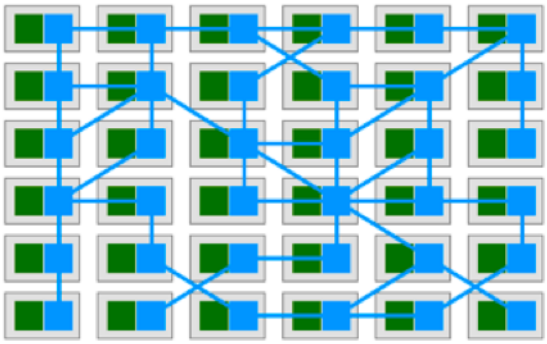

图 3: 服务网格 (Service Mesh)

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。

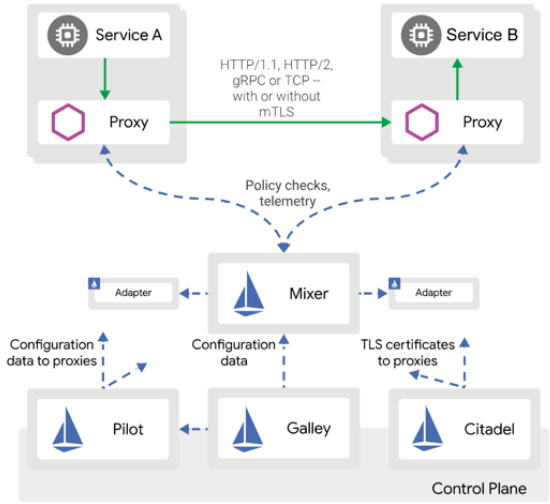

图 4: Istio 架构图（v1.4）

基于 Istio，部署前述的示例系统，具体架构变为5。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。

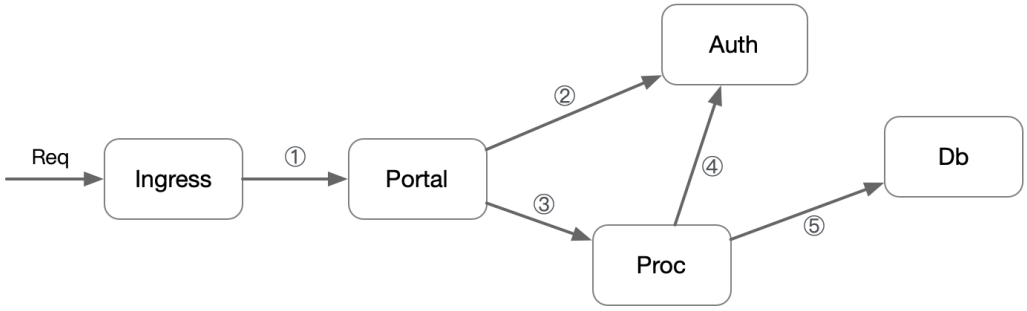

图 5: Istio 中的示例系统

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

0. 插入 Envoy 容器和 TraceManager 容器：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。

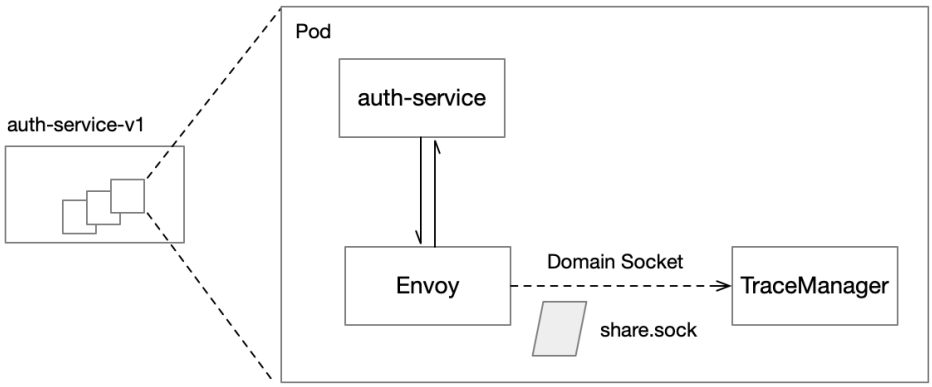

图 6: 插入容器

1. 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7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8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9

   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

```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 7: 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 8: v1 版本规则

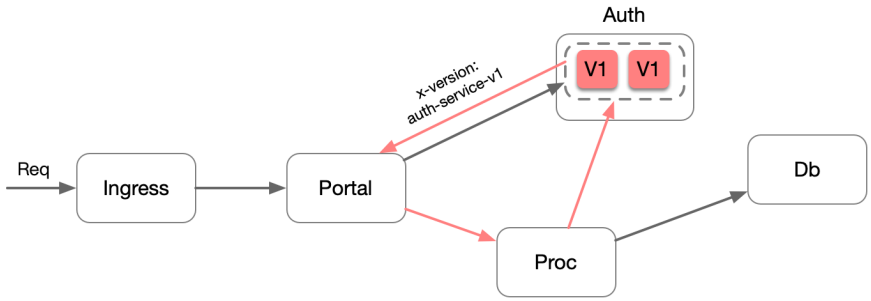

图 9: 流量全部发往 v1

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 的用户请求；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用户请求集合。

2. 新版本上线
3. 创建默认的路由规则和版本信息
4. 创建默认的路由规则和版本信息
5. 创建默认的路由规则和版本信息

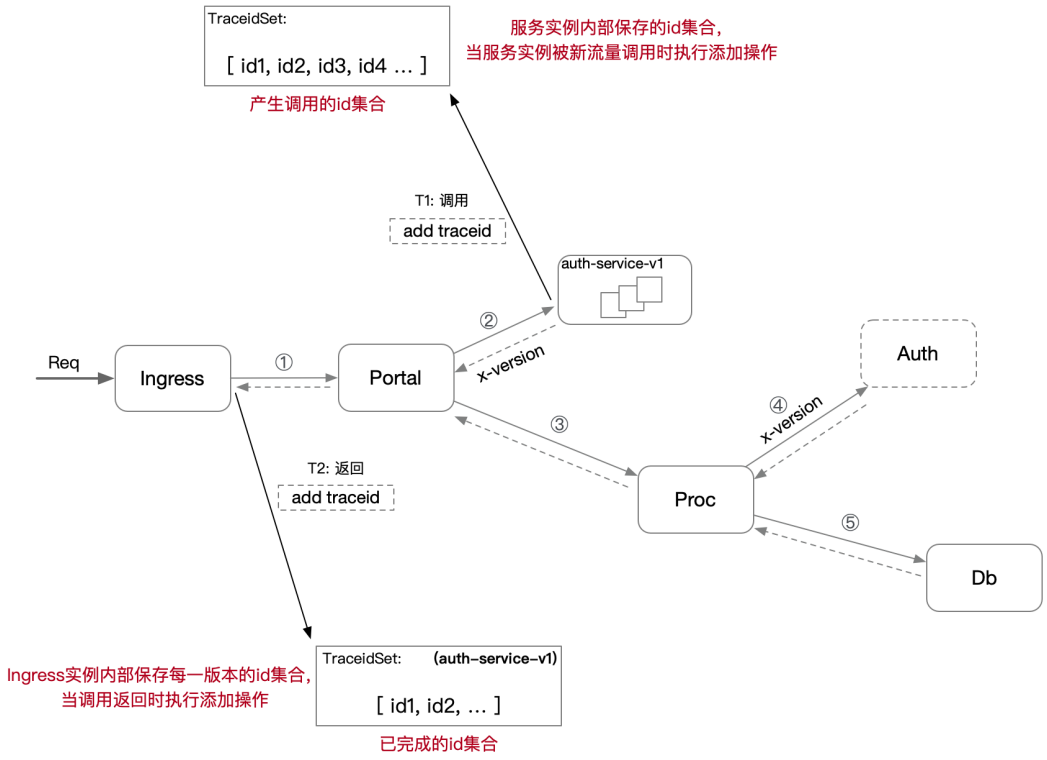
TraceidSet:
[id1, id2, id3, id4 ...]
产生调用的id集合
服务实例内部保存的id集合，
当服务实例被新流量调用时执行添加操作
T1: 调用
add traceid
auth-service-v1
Req
Ingress
①
Portal
②
x-version
③
Auth
④
x-version
Proc
⑤
Db
T2: 返回
add traceid
Ingress实例内部保存每一版本的id集合，
当调用返回时执行添加操作
TraceidSet: (auth-service-v1)
[id1, id2, ...]
已完成的id集合

图 10: 事务记录