文档中心

项目相关文档

- Http to Dubbo默认转化规则
- triple 和 dubbo 协议的相互转换

proxy 设计

- trace in pixiu
- prometheus pull in pixiu
- Polit支持服务映射
- pilot-agen
- 最佳实践&宣传
- Cluster 健康检查
- XDS-LDS
- 开源之夏
- 代理注册
- websocket 支持
- Proxy Mesh方案建设
  - 与istio一起工作
  - 部署调研和方案思考
- 非标准 Rete
  - 基于离散线段树的BatchNumberRangeIn优化
  - Pixiu路由原理简介
  - 模型设计
  - 标准字典树做contains 批量判断
  - urlMatch 的 BatchTester 实现
- WIP：开发手册
- Dubbo-go-pixiu性能测试
- dubbo多协议转换机制
- Pixiu & gRPC：基于 Reflection 调用技术方案
- Http Connection Manager & Filter Chain
- 调研中|Pixiu: Dubbo与SpringCloud互通方案
- 流量回放
- 鉴权
- DBMesh 实现计划
- Event Mesh 实现调研 && 实现方案

Control Plane 任务项

丛国庆、华钟明、熊聘、李凌志

- [ ] 支持服务元数据 蔡俊铭、李欢欢
- [ ] 支持服务映射 梦超、徐扬清
- [ ] 支持接入 Zookeeper / Naocs 岳枫博、王亚峰
- [ ] 支持服务查询 张国强、马兴
- [ ] 支持服务测试 杨博源、陈仕博
- [ ] 与 pixiu 数据面整合，支持管控数据面 麻志辉
- [ ] 面向用户的部署方案 @bobtthp

- [ ] 支持 Standalone 工作模式
- [ ] 下发 Dubbo 已有治理规则
- [ ] 探索与 tracing / metrics 组件工作的方式
- [ ] 支持实现应用间可信

项目方向

项目登记：

- 立项
- 项目方案

🔥【项目】Dubbo Pixiu Control Panel

项目步骤：

- 立项
  - 背景：Dubbo Mesh 形态
  - 目的：Pixiu 承担 Dubbo Control Panel
- Pixiu
  - Istio + Sidecar（Envoy）
- 时间点
  - 项目上线 2023-01-01
  -
- 计划
  - 一期
  - 二期
  - 三期
- 调研
  - Istio
    - pilot
      - pilot-agent
      - pilot-discovery
  - Pixiu
    - pixiu-admin：xDS Server （现状）
    - pixiu ： gateway/sidecar（现状）
- 产品
  - 功能
    -
- 方案
  - 控制面 工程结构/仓库 形态：
    - 1、独立仓： fork
    - 2、Pixiu 集成仓：集成 Istio 代码

TODO：

- Merge Pixiu PR @岳枫博
- 调整 Pixiu 目录 @岳枫博
- 合并 Isito 核心代码 @bob

Pilot 服务注册发现

Draft|技术方案

前言

背景

Dubbo Mesh

目的

dubbo-control-plane 提供 dubbo 服务注册发现的标准化能力，dubbo （数据面）通信不再关注当前使用的是什么注册中心，数据面对接 dubbo-control-plane 以实现服务注册与发现，该能力同时支持包括 mesh 环境 和 非 mesh 环境。

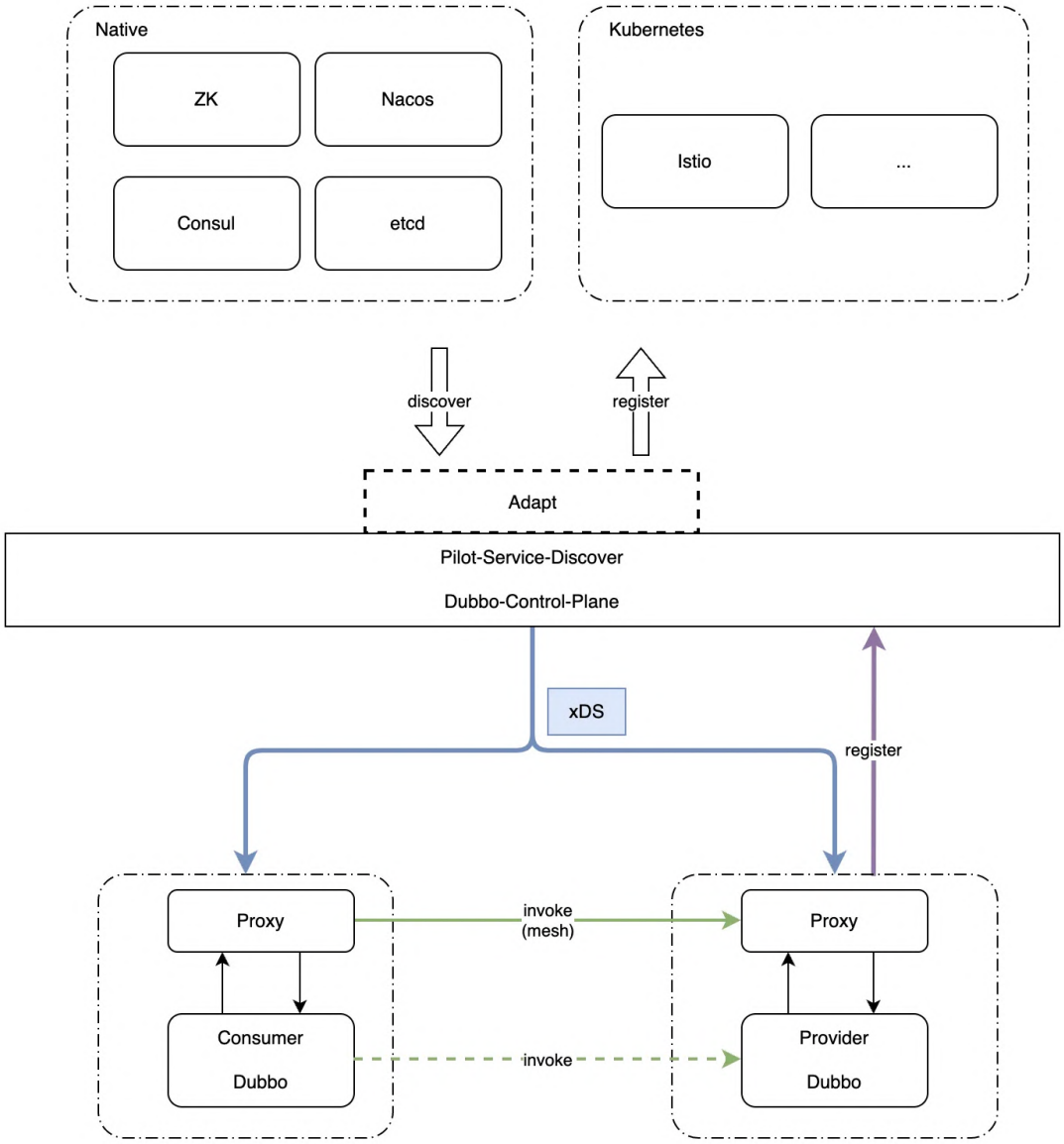
方案梳理

梳理步骤

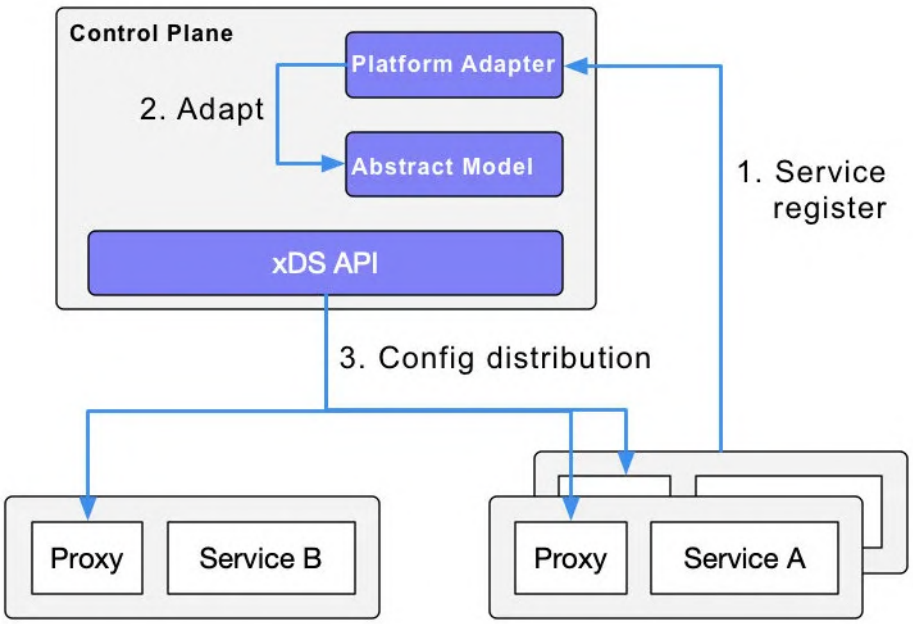
- 【TODO】调研 dubbo 、spring cloud、istio 服务注册发现相关的数据格式
- 【TODO】梳理服务信息上报和下发、监听 等流程

Pilot 服务发现

Pilot – Dubbo 服务发现架构


Native
ZK
Nacos
Consul
etcd
Kubernetes
Istio
...
discover
register
Adapt
Pilot-Service-Discover
Dubbo-Control-Plane
xDS
register
Proxy
invoke
(mesh)
Proxy
Consumer
Dubbo
invoke
Provider
Dubbo

服务发现模型



概述

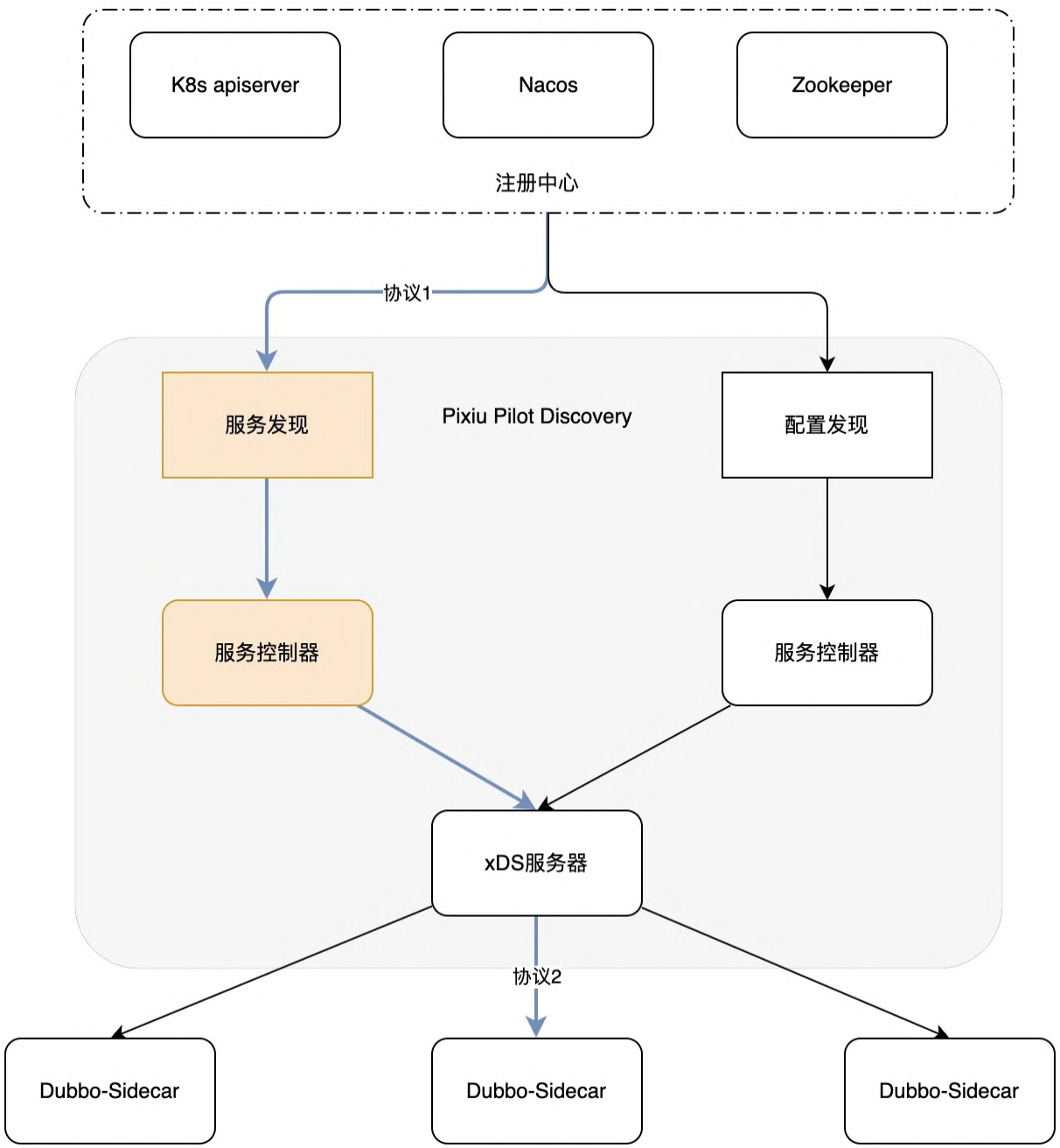
pilot-discovery 主要功能是 从注册中心（k8s、nacos、zk）获取信息并汇集，也包括从 kubernetes API server 获取流量规则，将服务信息和流量规则转化为数据面可以理解的格式，通过标准的数据面 API 下发到网格中的各个 Pixiu-Sidecar。

pilot-discovery 包括：

- 服务发现
- 配置规则发现
- xDS配置下发

本次主要聚焦于 服务发现 相关的技术方案整理设计。

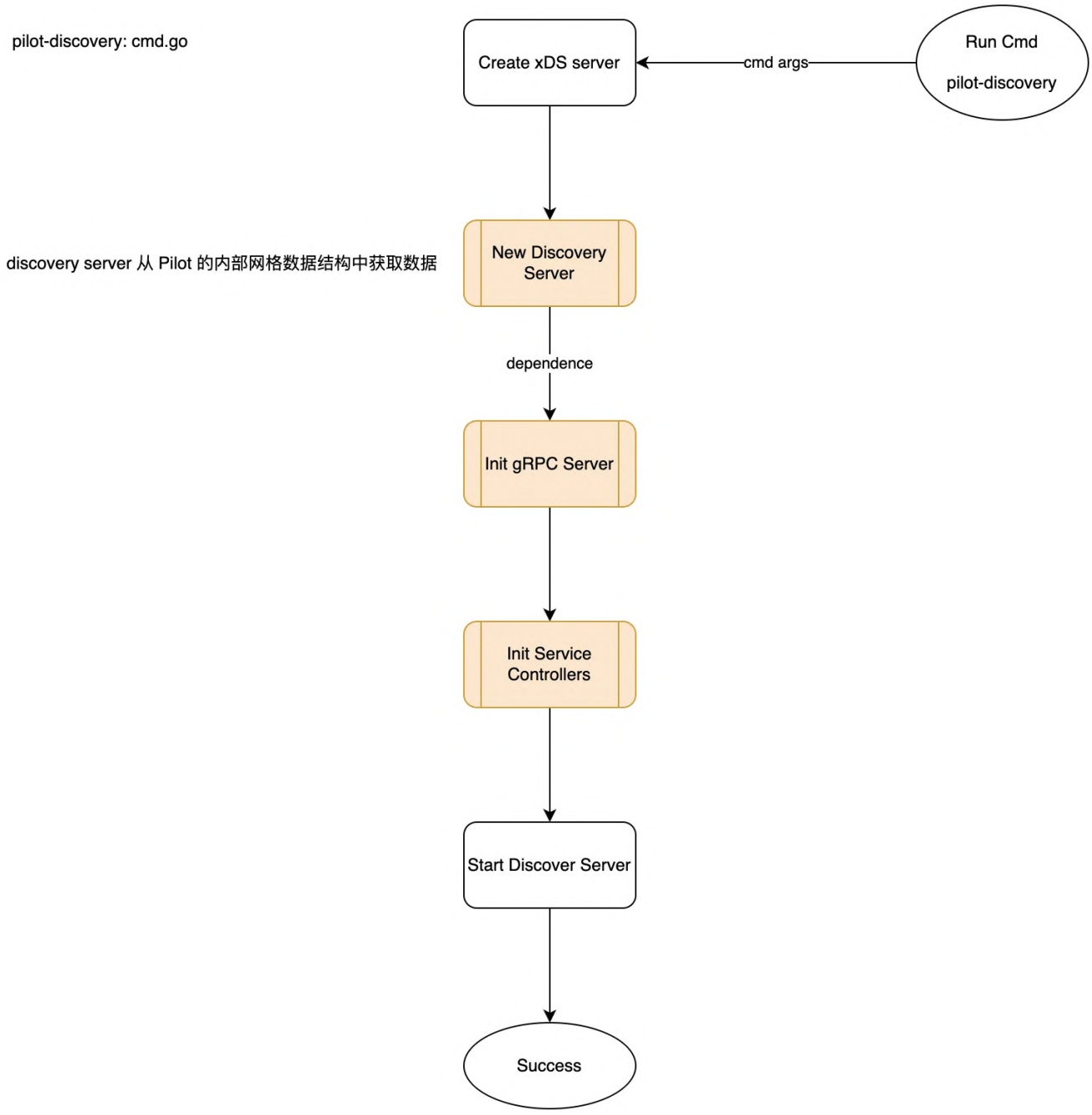
Dubbo Plane 服务发现模型：



> Istio Pilot 服务发现是通过 监听底层平台服务注册中心来缓存 Istio 的服务模型，并且 watch 服务模型的变化，再服务模型更新时触发相关事件回调处理函数。

Pixiu Pilot Start Process

Pilot Start

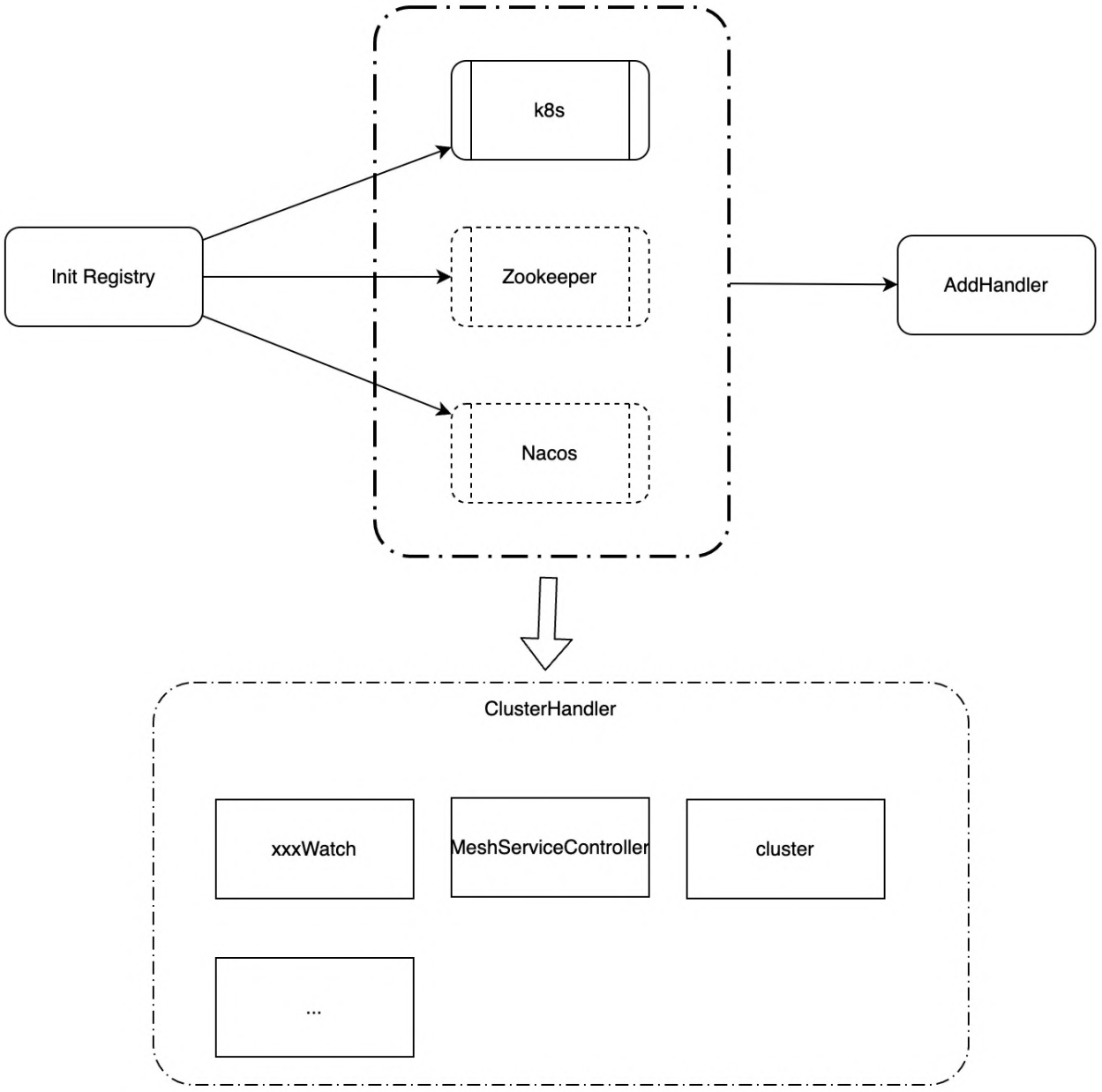


ServiceControllers

服务发现的主要逻辑在Pilot中由ServiceController（服务控制器）实现，**通过监听底层平台的服务注册中心来缓存Istio服务模型，并监视服务模型的变化，在服务模型更新时触发相关事件回调处理函数的执行**。

初始化注册中心

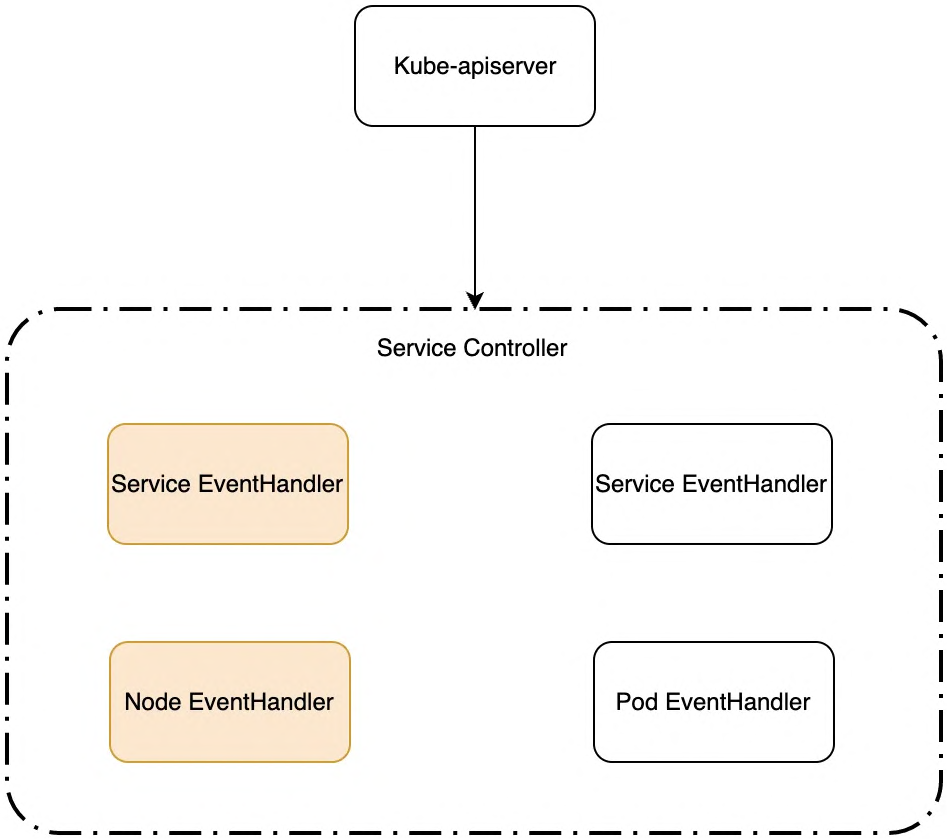
```
// Mock is a service registry that contains 2 hard-coded test services
                        Mock ID = "Mock"
    // Kubernetes is a service registry backed by k8s API server
                    Kubernetes ID = "Kubernetes"
// External is a service registry for externally provided ServiceEntries
                    External ID = "External"
```



四种监听器，监听 k8s 资源更新

- Service事件处理器会将根据事件的类型更新缓存，然后调用serviceHandlers的事件处理器进行回调。serviceHandlers事件处理器是在初始化DiscoveryService的时候设置的。

- Endpoint处理器会在调用newEndpointsController创建endpointsController的时候进行注册



Service of k8s

```
type Service struct {
Attributes              ServiceAttributes
Ports                   PortList          json:"ports,omitempty"
ServiceAccounts         []string          json:"serviceAccounts,omitempty"
CreationTime            time.Time         json:"creationTime,omitempty"
Hostname                host.Name         json:"hostname"
ClusterVIPs             AddressMap        json:"clusterVIPs,omitempty"
DefaultAddress          string            json:"defaultAddress,omitempty"
AutoAllocatedIPv4Address string          json:"autoAllocatedIPv4Address,omitempty"
AutoAllocatedIPv6Address string          json:"autoAllocatedIPv6Address,omitempty"
Resolution              Resolution
MeshExternal            bool
```

ResourceVersion string
}

Endpoint of k8s

- Endpoints
  - TypeMeta 对象类型
  - ObjectMeta 元信息
    https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#metadata
  - Subsets 端点集合

```
//ObjectMeta

Subsets: [
  {
       Addresses: [{"ip": "10.10.1.1"}, {"ip": "10.10.2.2"}],
       Ports: [{"name": "a", "port": 8675}, {"name": "b", "port": 309}]
     },
     {
      Addresses: [{"ip": "10.10.3.3"}],
      Ports: [{"name": "a", "port": 93}, {"name": "b", "port": 76}]
    },
  ]
```

IstioEndpoint

```
type IstioEndpoint struct {
    Labels                labels.Instance
    Address               string
    ServicePortName       string
    EnvoyEndpoint         *endpointv3.LbEndpoint
    ServiceAccount        string
    Network               network.ID
    Locality              Locality
    EndpointPort          uint32
    LbWeight              uint32
    TLSMode               string
    Namespace             string
    WorkloadName          string
    HostName              string
    SubDomain             string
    TunnelAbility         networking.TunnelAbility
    DiscoverabilityPolicy EndpointDiscoverabilityPolicy `json:"-"`
    HealthStatus          HealthStatus
}
```

Mesh Configuration Protocol (MCP)

https://github.com/istio/api/tree/master/mcp

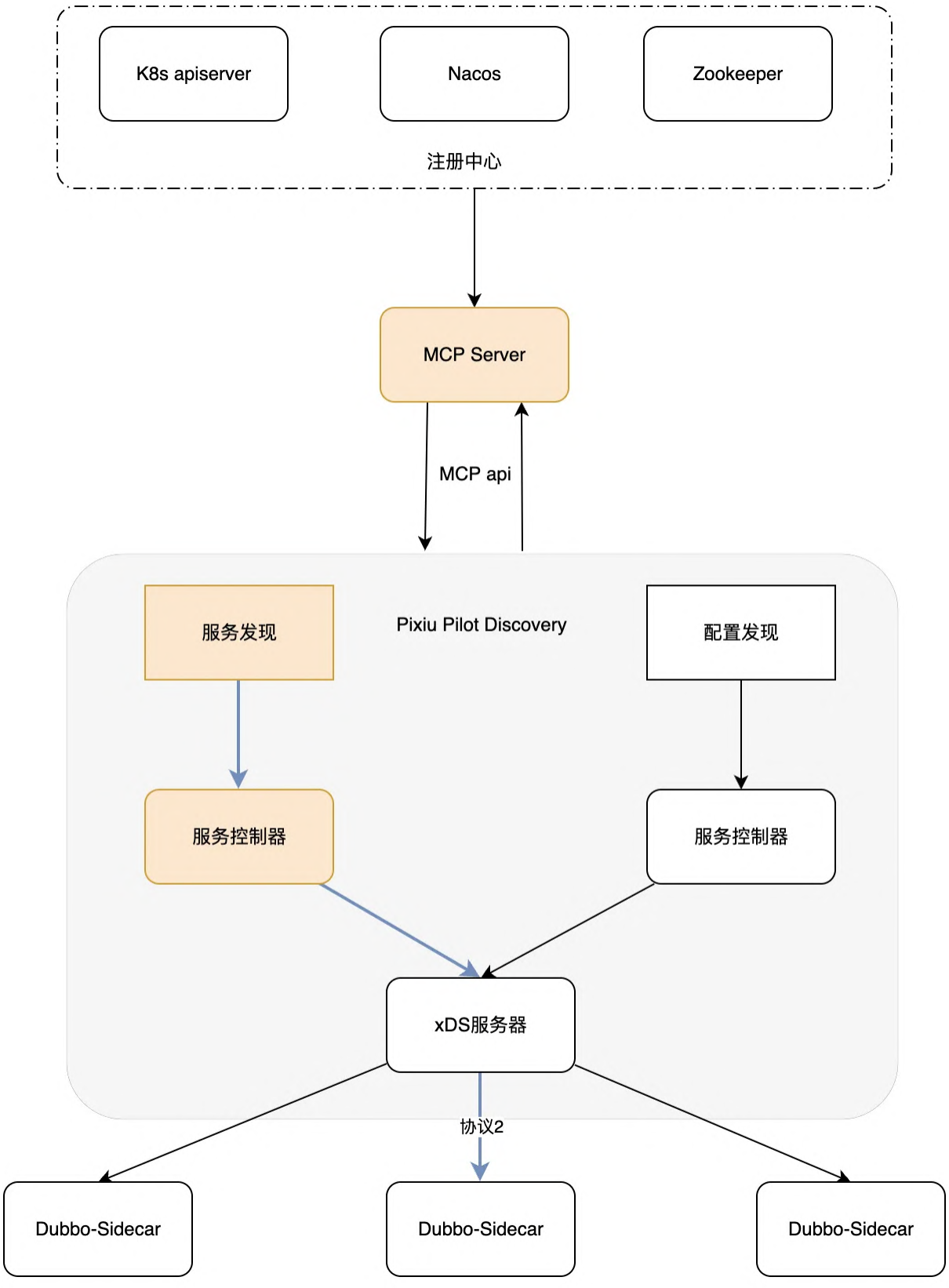
MCP是基于订阅的配置分发API。配置消费者（即sink）请求更新来自配置生产者（即source）的资源集合。添加，更新或删除资源时，source会将资源更新推送到sink。如果sink接受，则回复ACK，如果被拒绝则是NACK，例如因为资源无效。一旦对先前的更新进行了ACK/NACK，则source可以推送其他更新。该source一次只能运行一次未完成的更新（每个集合）。

MCP是一对双向流gRPC API服务（ResourceSource和ResourceSink）。

- 当source是服务器而sink是客户端时，将使用 `ResourceSource` 服务。默认情况下，Galley实现

`ResourceSource` 服务，并且Pilot连接作为客户端。

- 当source是客户端，而sink是服务器时，将使用 `ResourceSink` 服务。可以将Galley配置为可选地 `dial-out` 到远程配置sink，例如 Pilot位于另一个集群中，在该集群中，它不能作为客户端启动与Galley的连接。在这种情况下，Pilot将实现 `ResourceSink` 服务，而Galley将作为客户端进行连接。


K8s apiserver
Nacos
Zookeeper
注册中心
MCP Server
MCP api
服务发现
Pixiu Pilot Discovery
配置发现
服务控制器
服务控制器
xDS服务器
协议2
Dubbo-Sidecar
Dubbo-Sidecar
Dubbo-Sidecar

Resource:

```go
type Resource struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    // Common metadata describing the resource.
    Metadata *Metadata `protobuf:"bytes,1,opt,name=metadata,proto3" json:"metadata,omitempty"`
    // The primary payload for the resource.
    Body *any1.Any `protobuf:"bytes,2,opt,name=body,proto3" json:"body,omitempty"`
}
```

Dubbo Nacos

基于Dubbo2.7, dubbo 协议， SpringCloudAlibabaDubbo 的数据格式

Nacos 服务注册数据

```go
{
    "dubbo.metadata-service.urls": "[ \"dubbo://192.168.0.109:20880/com.alibaba.cloud.dubbo.service.DubboMetadataService?anyhost=true&application=demo&bind.ip=192.168.0.109&bind.port=20880&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&group=demo&interface=com.alibaba.cloud.dubbo.service.DubboMetadataService&methods=getAllServiceKeys,getServiceRestMetadata,getExportedURLs,getAllExportedURLs&pid=34041&qos.enable=false&release=2.7.8&revision=2.2.2.RELEASE&side=provider&timestamp=1663504121517&version=1.0.0\" ]",
    "preserved.register.source": "SPRING_CLOUD",
    "dubbo.protocols.dubbo.port": "20880"
}
```

dubbo.metadata-service.urls

```
[
    "dubbo://192.168.0.109:20880/com.alibaba.cloud.dubbo.service.DubboMetadataService?anyhost=true&application=demo&bind.ip=192.168.0.109&bind.port=20880&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&group=demo&interface=com.alibaba.cloud.dubbo.service.DubboMetadataService&methods=getAllServiceKeys,getServiceRestMetadata,getExportedURLs,getAllExportedURLs&pid=34041&qos.enable=false&release=2.7.8&revision=2.2.2.RELEASE&side=provider×tamp=1663504121517&version=1.0.0"
]
```

dubbo://192.168.0.109:20880/com.alibaba.cloud.dubbo.service.DubboMetadataService?anyhost=true&application=demo&bind.ip=192.168.0.109&bind.port=20880&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&group=demo&interface=com.alibaba.cloud.dubbo.service.DubboMetadataService&methods=getAllServiceKeys,getServiceRestMetadata,getExportedURLs,getAllExportedURLs&pid=34041&qos.enable=false&release=2.7.8&revision=2.2.2.RELEASE&side=provider×tamp=1663504121517&version=1.0.0

➜ ~ curl –X GET '127.0.0.1:8848/nacos/v1/ns/instance/list?serviceName=demo'

```
{
    "name":"DEFAULT_GROUP@@demo",
    "groupName":"DEFAULT_GROUP",
    "clusters":"",
    "cacheMillis":10000,
    "hosts":[
        {
            "instanceId":"192.168.0.109#8080#DEFAULT#DEFAULT_GROUP@@demo",
            "ip":"192.168.0.109",
            "port":8080,
            "weight":1,
            "healthy":true,
            "enabled":true,
            "ephemeral":true,
            "clusterName":"DEFAULT",
            "serviceName":"DEFAULT_GROUP@@demo",
            "metadata":{
                "dubbo.metadata-service.urls":"[ \"dubbo://192.168.0.109:20880/com.alibaba.cloud.dubbo.service.DubboMetadataService?anyhost=true&application=demo&bind.ip=192.168.0.109&bind.port=20880&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&group=demo&interface=com.alibaba.cloud.dubbo.service.DubboMetadataService&methods=getAllServiceKeys,getServiceRestMetadata,getExportedURLs,getAllExportedURLs&pid=34041&qos.enable=false&release=2.7.8&revision=2.2.2.RELEASE&side=provider&timestamp=1663504121517&version=1.0.0\" ]",
                "preserved.register.source":"SPRING_CLOUD",
                "dubbo.protocols.dubbo.port":"20880"
            },
            "instanceHeartBeatInterval":5000,
            "instanceHeartBeatTimeOut":15000,
            "ipDeleteTimeout":30000
        }
    ],
    "lastRefTime":1663506386347,
    "checksum":"",
    "allIPs":false,
    "reachProtectionThreshold":false,
    "valid":true
}
```


基于Dubbo3.0.7, dubbo 协议， spring-boot 的数据格式

Nacos 服务注册数据

dubbo-springboot-demo-provider	{ "dubbo.metadata-service.url-params": "{\"connections\":\"1\",\"version\":\"1.0.0\",\"dubbo\":\"2.0.2\",\"release\":\"3.0.7\",\"side\":\"provider\",\"port\":\"20881\",\"protocol\":\"dubbo\"}", "dubbo.endpoints": "[{\"port\":20881,\"protocol\":\"dubbo\"}]", "dubbo.metadata.revision": "0b317e73ddce16868430c8389f2d694b", "dubbo.metadata.storage-type": "local" }

providers:org.apache.dubbo.springboot.demo.DemoService::	{ "side": "provider", "release": "3.0.7", "methods": "sayHello,sayHelloAsync", "deprecated": "false", "dubbo": "2.0.2", "pid": "40190", "interface": "org.apache.dubbo.springboot.demo.DemoService", "service-name-mapping": "true", "generic": "false", "path": "org.apache.dubbo.springboot.demo.DemoService", "protocol": "dubbo", "application": "dubbo-springboot-demo-provider", "background": "false", "dynamic": "true", "category": "providers", "anyhost": "true", "timestamp": "1663505357098" }
providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider	{ "side": "provider", "release": "3.0.7", "methods": "getMetadataURL,isMetadataService,getExpor

tedURLs,serviceName,getSubscribedURLs,version,getExportedServiceURLs,getMetadataInfo,toSortedStrings,getMetadataInfos,getServiceDefinition",

"deprecated": "false",

"dubbo": "2.0.2",

"pid": "40190",

"interface": "org.apache.dubbo.metadata.MetadataService",

"service-name-mapping": "true",

"version": "1.0.0",

"generic": "false",

"revision": "3.0.7",

"path": "org.apache.dubbo.metadata.MetadataService",

"protocol": "dubbo",

"delay": "0",

"application": "dubbo-springboot-demo-provider",

"background": "false",

"dynamic": "true",

"executes": "100",

"category": "providers",

"connections": "1",

"group": "dubbo-springboot-demo-provider",

"anyhost": "true",

"timestamp": "1663505358165"

	}

服务列表：

{"count":4,"doms":["providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider","dubbo-springboot-demo-provider"]}

➜ ~ curl -X GET '127.0.0.1:8848/nacos/v1/ns/service?serviceName=dubbo-springboot-demo-provider'

{"namespaceId":"public","groupName":"DEFAULT_GROUP","name":"dubbo-springboot-demo-provider","protectThreshold":0.0,"metadata":{},"selector":{"type":"none","contextType":"NONE"},"clusters":[{"name":"DEFAULT","healthChecker":{"type":"TCP"},"metadata":{}}]}%

➜ ~ curl -X GET '127.0.0.1:8848/nacos/v1/ns/service?serviceName=providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider'

{"namespaceId":"public","groupName":"DEFAULT_GROUP","name":"providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider","protectThreshold":0.0,"metadata":{},"selector":{"type":"none","contextType":"NONE"},"clusters":[{"name":"DEFAULT","healthChecker":{"type":"TCP"},"metadata":{}}]}

➜ ~ curl -X GET '127.0.0.1:8848/nacos/v1/ns/instance/list?serviceName=dubbo-springboot-demo-provider'

```
Go
{
    "name":"DEFAULT_GROUP@@dubbo-springboot-demo-provider",
    "groupName":"DEFAULT_GROUP",
    "clusters":"",
    "cacheMillis":10000,
    "hosts":[
        {
            "ip":"192.168.0.109",
            "port":20881,
            "weight":1,
            "healthy":true,
            "enabled":true,
            "ephemeral":true,
            "clusterName":"DEFAULT",
            "serviceName":"DEFAULT_GROUP@@dubbo-springboot-demo-provider",
            "metadata":{
                "dubbo.metadata-service.url-params":"{\"connections\":\"1
\",\"version\":\"1.0.0\",\"dubbo\":\"2.0.2\",\"release\":\"3.0.7\",\"side
\":\"provider\",\"port\":\"20881\",\"protocol\":\"dubbo\"}",
                "dubbo.endpoints":"[{\"port\":20881,\"protocol\":\"dubbo
\"}]",
                "dubbo.metadata.revision":"0b317e73ddce16868430c8389f2d694
b",
                "dubbo.metadata.storage-type":"local"
            },
            "instanceHeartBeatInterval":5000,
            "instanceHeartBeatTimeOut":15000,
            "ipDeleteTimeout":30000
        }
    ],
    "lastRefTime":1663506262118,
    "checksum":"",
    "allIPs":false,
    "reachProtectionThreshold":false,
    "valid":true
}
```

➜ ~ curl -X GET '127.0.0.1:8848/nacos/v1/ns/instance/list?serviceName=providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider'

```
{
    "name":"DEFAULT_GROUP@@providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider",
    "groupName":"DEFAULT_GROUP",
    "clusters":"",
    "cacheMillis":10000,
    "hosts":[
        {
            "ip":"192.168.0.109",
            "port":20881,
            "weight":1,
            "healthy":true,
            "enabled":true,
            "ephemeral":true,
            "clusterName":"DEFAULT",
            "serviceName":"DEFAULT_GROUP@@providers:org.apache.dubbo.metadata.MetadataService:1.0.0:dubbo-springboot-demo-provider",
            "metadata":{
                "side":"provider",
                "release":"3.0.7",
                "methods":"getMetadataURL,isMetadataService,getExportedURLs,serviceName,getSubscribedURLs,version,getExportedServiceURLs,getMetadataInfo,toSortedStrings,getMetadataInfos,getServiceDefinition",
                "deprecated":"false",
                "dubbo":"2.0.2",
                "pid":"40190",
                "interface":"org.apache.dubbo.metadata.MetadataService",
                "service-name-mapping":"true",
                "version":"1.0.0",
                "generic":"false",
                "revision":"3.0.7",
                "path":"org.apache.dubbo.metadata.MetadataService",
                "protocol":"dubbo",
                "delay":"0",
                "application":"dubbo-springboot-demo-provider",
                "background":"false",
                "dynamic":"true",
                "executes":"100",
                "category":"providers",
                "connections":"1",
                "group":"dubbo-springboot-demo-provider",
                "anyhost":"true",
                "timestamp":"1663505358165"
            },
            "instanceHeartBeatInterval":5000,
```

```
            "instanceHeartBeatTimeOut":15000,
            "ipDeleteTimeout":30000
        }
    ],
    "lastRefTime":1663506292066,
    "checksum":"",
    "allIPs":false,
    "reachProtectionThreshold":false,
    "valid":true
}
```

SpringCloud – Eureka

```
"instanceInfo": {
      "instanceId": "172.18.153.43:spring-cloud-producer:9000",
      "app": "SPRING-CLOUD-PRODUCER",
      "appGroupName": null,
      "ipAddr": "172.18.153.43",
      "sid": "na",
      "homePageUrl": "http://172.18.153.43:9000/",
      "statusPageUrl": "http://172.18.153.43:9000/info",
      "healthCheckUrl": "http://172.18.153.43:9000/health",
      "secureHealthCheckUrl": null,
      "vipAddress": "spring-cloud-producer",
      "secureVipAddress": "spring-cloud-producer",
      "countryId": 1,
      "dataCenterInfo": {
        "@class": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo"
,
        "name": "MyOwn"
      },
      "hostName": "172.18.153.43",
      "status": "UP",
      "leaseInfo": {
        "renewalIntervalInSecs": 30,
        "durationInSecs": 90,
        "registrationTimestamp": 1625476088459,
        "lastRenewalTimestamp": 1625477588519,
        "evictionTimestamp": 0,
        "serviceUpTimestamp": 1625476087664
      },
      "isCoordinatingDiscoveryServer": false,
      "metadata": {

      },
      "lastUpdatedTimestamp": 1625476088459,
      "lastDirtyTimestamp": 1625476087596,
      "actionType": "ADDED",
      "asgName": null,
      "overriddenStatus": "UNKNOWN"
    },
```

调研梳理

02 Draft | Pixiu And Dubbo3

前言

思考几个问题：

- 为什么 dubbo3 需要架构升级?
- 为什么已经有 Istio 等成熟的Mesh 方案，Dubbo 还需要自己做一个控制面?
- pixiu (pixiu & admin) 作为控制面后，当前已有的 gateway / sidecar 的能力如何规划?
- pixiu 到底需要解决什么痛点或者说，pixiu 的价值?

背景

Dubbo 在当下开发过程，部署形态，运行态，运维态 都存在着一些挑战（不做过多赘述），这类挑战在云原生场景下更为突出，所以 dubbo3 的架构升级，解耦了 控制面 与 数据面，而 Pixiu 会承担 Dubbo 控制面的职责更好的与 Dubbo 一起拥抱云原生。

目的

Pixiu 团队通过本次同步讨论会议，明确好 pixiu 今后的发展方向：作为 Dubbo 的 Control Pannl 去支持 Dubbo 生态。

概述

Dubbo3 核心功能与规划

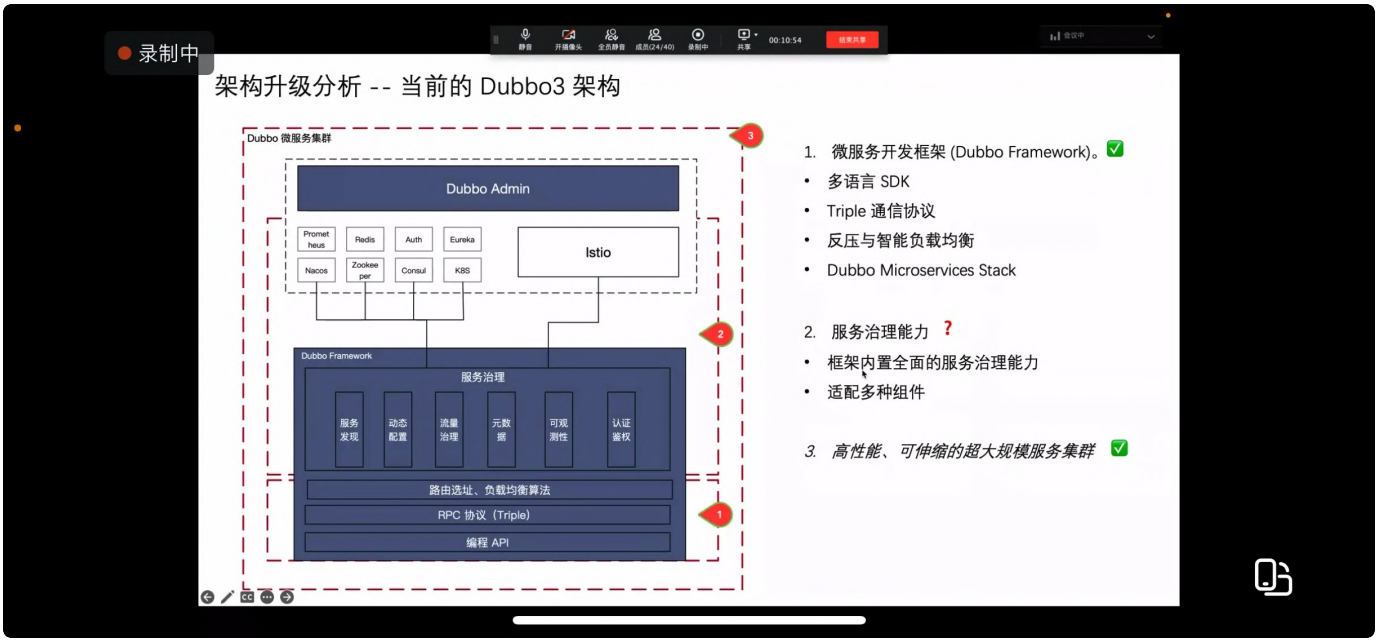
Dubbo 面临的挑战

四个视角看待：

- 开发态：多语言支持，编程模型单一，服务契约管理，生态组件支持
- 部署态：依赖冲突，启动速度慢，声明周期管理欠缺，无法适配原生基础设施与架构

- 运行态：异构体系无法互通（k8s/SC/gRPC），流量治理能力弱，吞吐量、稳定性、性能、资源利用率低，网关穿透性差
- 运维态：集群管理不透明，治理能力不友好、使用成本高，可观测性差（集群、拓扑、单机）

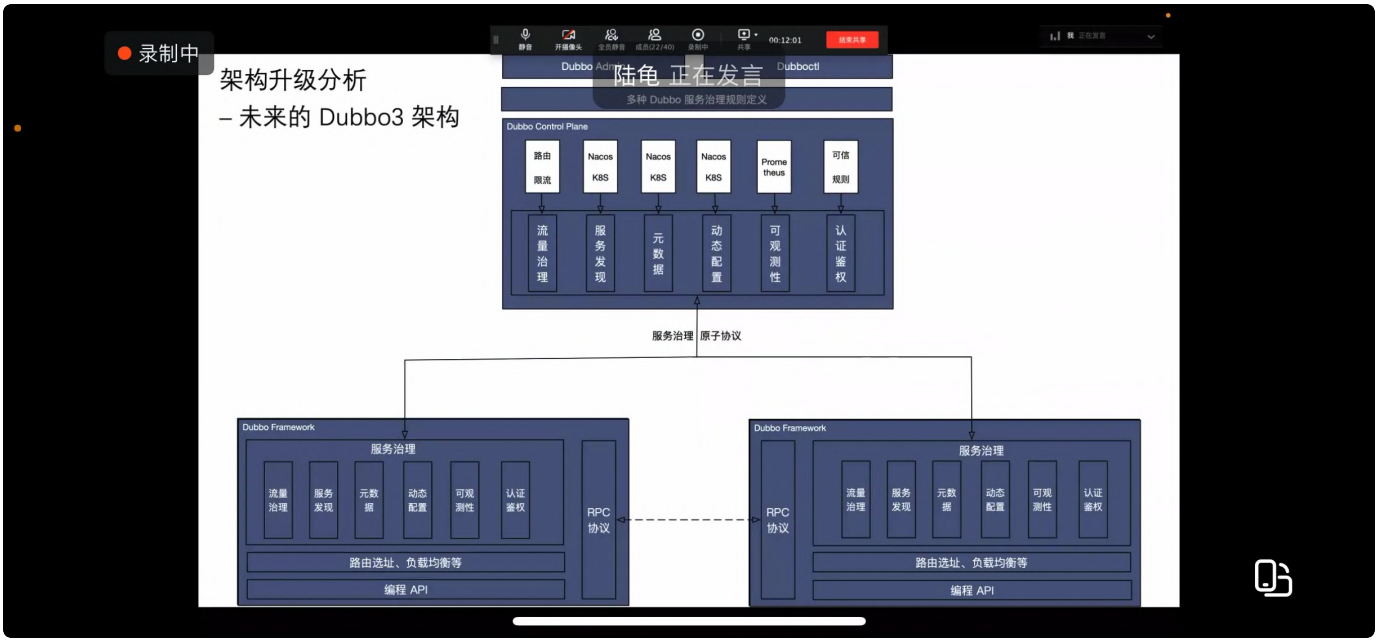
Dubbo3 当前架构



当前服务治理架构面临的问题：

- 与各微服务组件直接适配、耦合度高
- 微服务组件间治理能力没有交互
- 多语言实现复杂度高
- 增加或升级规则成本高
- 现有规则亟待升级
- 与云原生基础设施脱节严重
- Mesh体系与现有体系割裂
- 框架臃肿、复杂度极高

Dubbo3 未来架构



高内聚、低耦合！

Dubbo3

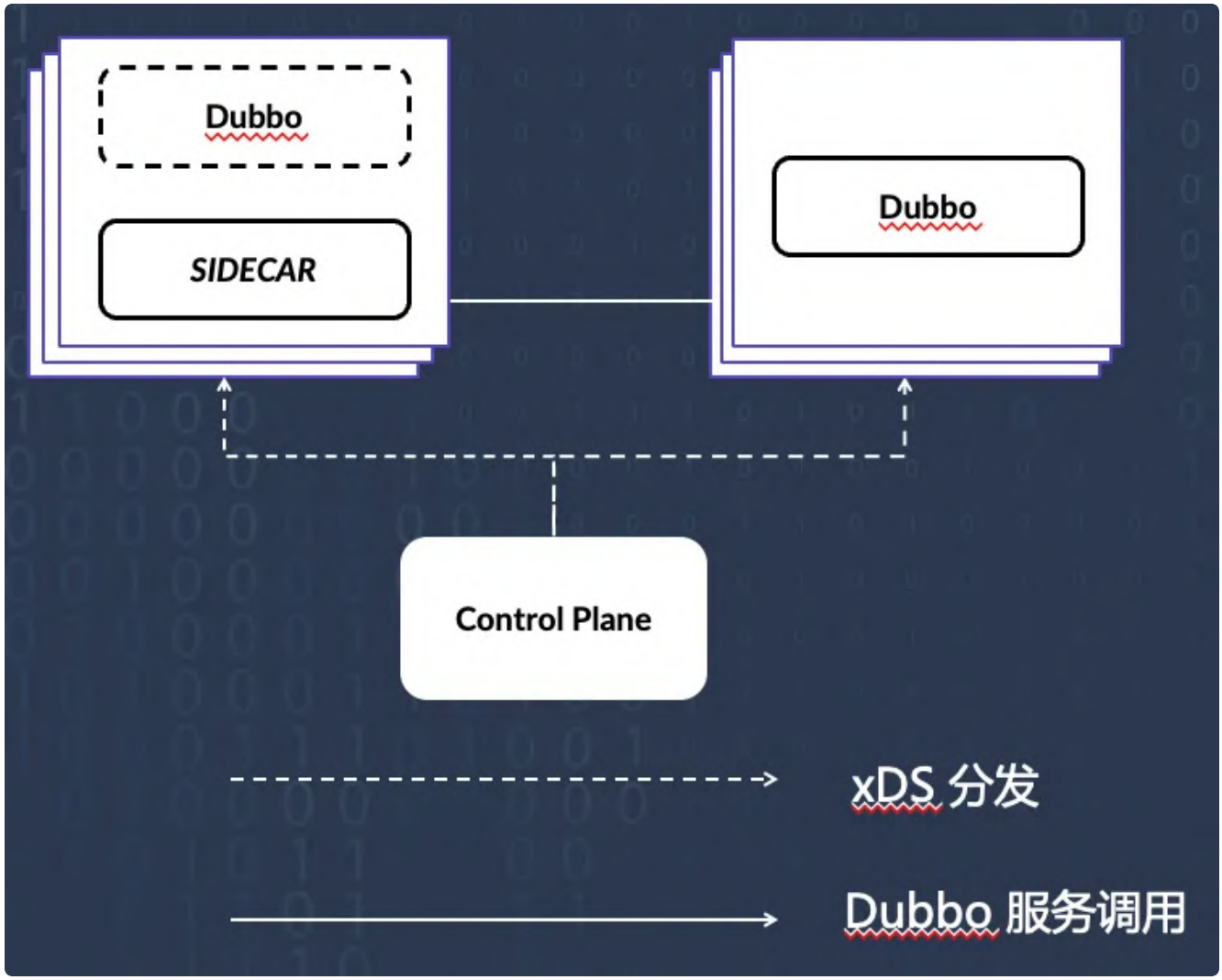
- Dubbo Framework
- Dubbo Control Plane

Dubbo Control Plane　承载服务治理能力的核心，具有抽象、统一的模型，融合了之前分散的服务治理能力，更容易扩展。

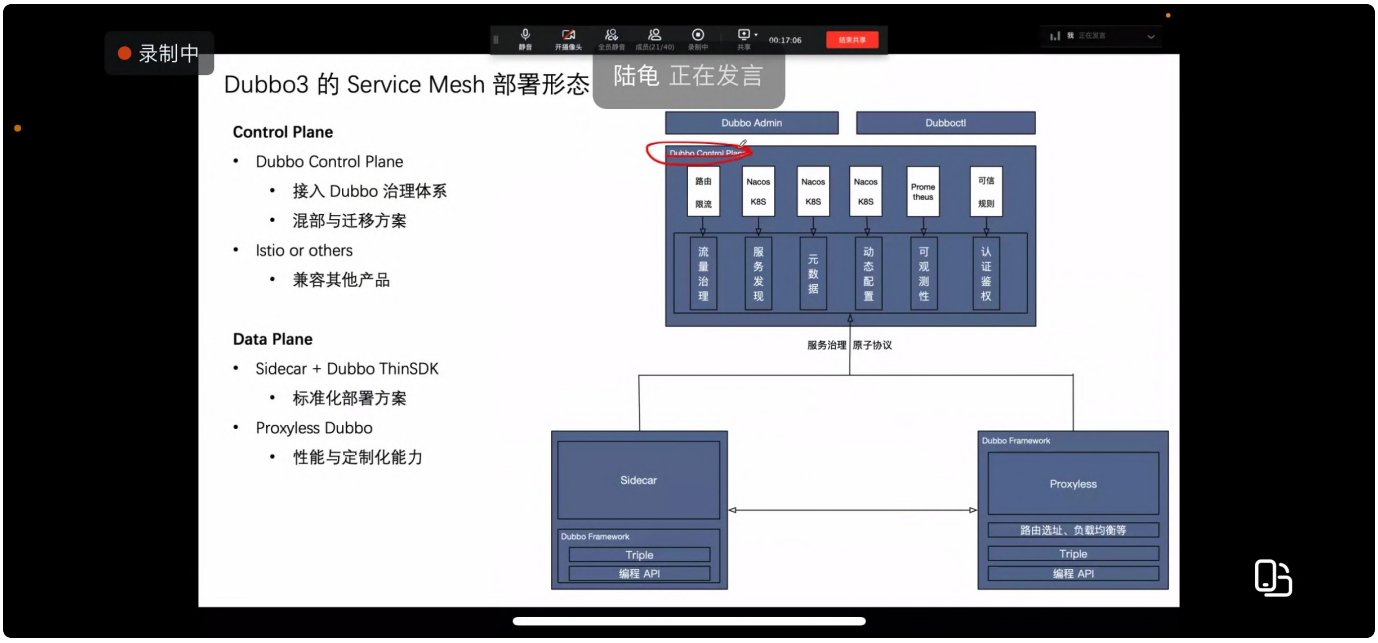
以 Control Plane 为核心分别向用户态（Dubbo Admin） 和运行态（Dubbo Framework）拓展，使得两端更纯碎、功能更强大、实现也更简单。

Dubbo Mesh

总体架构



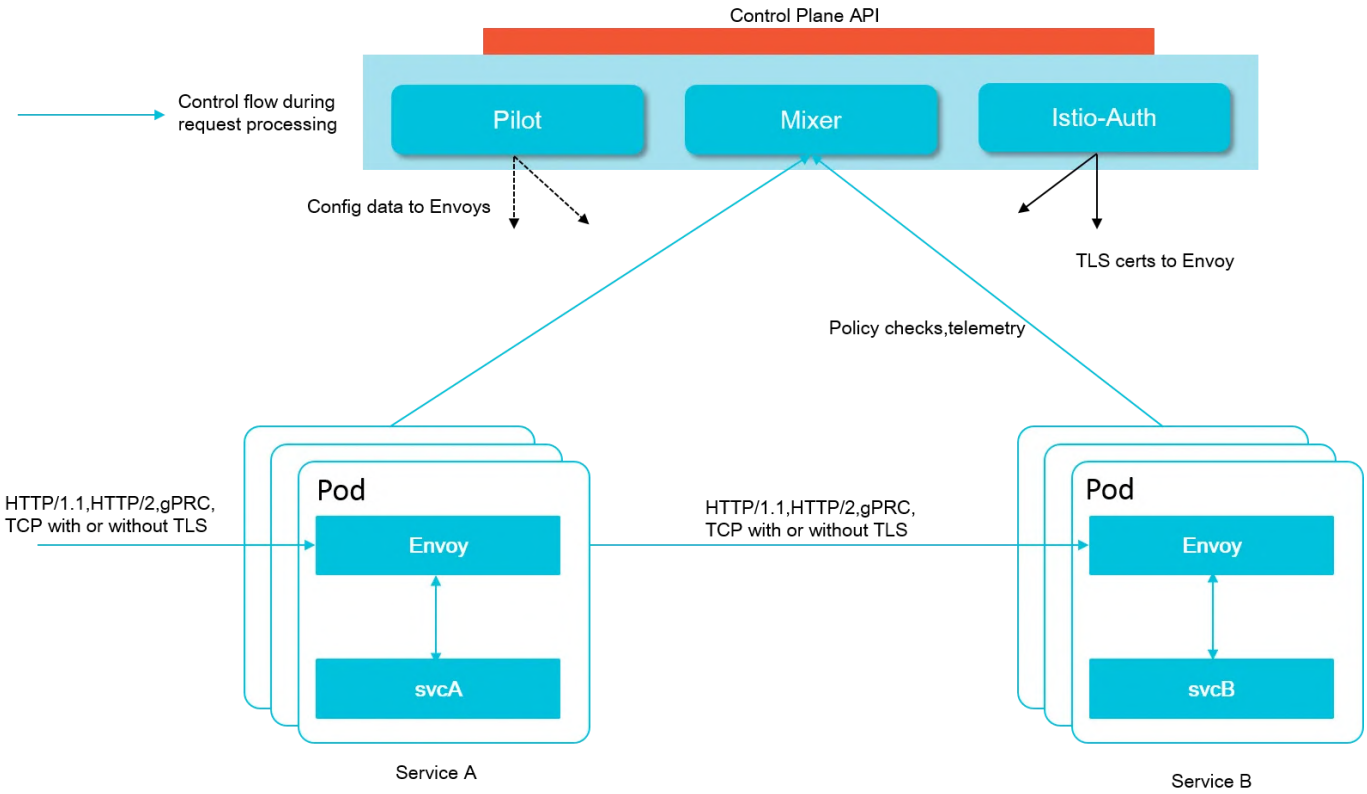
部署形态



Pixiu 在 Dubbo 生态

Pixiu Mesh

- Pixiu Console （xDS Server、）
- Pixiu Sidecar



Pixiu 与 Istio

TODO

参考文档

- 概念与架构
- Dubbo Mesh
- dubbo-awesome/D3.2-proxyless-mesh.md at master · apache/dubbo-awesome
- 无标题
- Dubbo Ecosystem – 从微服务框架到微服务生态-阿里云开发者社区
- 阿里云帮助中心-kubernetes上实现 istio 分布式追踪
- 服务网格 ASM 产品简介 | 阿里云文档中心
- The Istio service mesh
- gRPC Proxyless Service Mesh
- Cloud Native Computing Foundation

Istio 代码迁移

概述

背景

Dubbo-go-pixiu 当前主要覆盖 gateway 与 sidecar 的能力，今后会作为 Dubbo Control Plane 的主要形态去建设，目前开发人员都是同一波人，从代码仓库角度看，希望 Dubbo-go-pixiu 仓库能够作为一个主仓库，构建 Control Plane 项目能力，同时兼容 Gateway 与 Sidecar 工程能力。

总结：Dubbo-go-pixiu 代码仓库包含完整的 Mesh 形态下，Control Plane 和 Sidecar 的项目代码。

Dubbo-go-pixiu 仓库说明

当前 Pixiu 目录结构:

Markdown

```
.
├── cache
├── cmd
├── configs
├── docs
├── igt
├── log
├── operator
├── pilot
├── pkg
└── security
```

方案: 在根目录添加对应 Istios 的代码模块。如 operator、pilot、security... 等等

调研

Istio 的工程目录

```
.
├── bin
├── cni
├── common
├── docker
├── istioctl
├── licenses
├── logo
├── manifests
├── operator
├── pilot
├── pkg
├── prow
├── release
├── releasenotes
├── samples
├── security
├── tests
└── tools
```

Package/Directory/File		Introduction
bin		存放初始化依赖、编译、插件证书检查、代码生成的脚本
cni	独立进程	Istio 容器网络接口（CNI）来安装、配置和使用 Istio 网格。 https://github.com/istio/istio/tree/master/cni https://istio.io/latest/zh/docs/setup/additional-setup/cni/ https://www.servicemesher.com/blog/istio-cni-note/
common		配置文件和一些脚本

docker		
istioctl	独立进程	Istio 的命令行工具
licenses		
logo		
manifests		Kubernetes 的包管理器
operator	独立进程	Operator 控制器
pilot	独立进程	“领航员”，策略配置组件，pliot对Envoy的生命周期进行管理，同时提供了智能路由（如A/B测试、金丝雀部署）、流量管理（超时、重试、熔断）功能。
pkg		
prow		
release		
releasenotes		
samples		
security	独立进程	Istio 安全功能，强大的策略，透明的 TLS 加密，认证，授权和审计（AAA）工具来保护你的服务和数据。
tests		
tools		

Package/Directory/File	Introduction
addons	一些插件，比如展示metrics的grafana和绘制服务调用图的servicegraph
bin	存放初始化依赖、编译、插件证书检查、代码生成的脚本
cni	
common	

docker	
broker	Istio对Open Service Broker的一种实现，该API使得外部服务能自动访问Istio服务。broker目前还处于研发阶段。
galley	提供了Istio的配置管理功能，目前还处于研发阶段。
install	生成各环境（ansible、consul、ereka、kubernetes等）安装istio时需要yaml配置清单。
istioctl	istio终端控制工具（类似kubectl之于kubernetes），用户通过istioctl来修改istio运行时配置，执行服务治理策略。
mixer	“混音器”，参与到tracffic处理流程。通过对envoy上报的attributes进行处理，结合内部的adapters实现日志记录、监控指标采集展示、配额管理、ACL检查等功能。
pilot	“领航员”，pliot对Envoy的生命周期进行管理，同时提供了智能路由（如A/B测试、金丝雀部署）、流量管理（超时、重试、熔断）功能。
pkg	顶级公共包，包含istio版本处理、tracing、日志记录、缓存管理等。
release	包含Istio在各平台上进行编译的脚本。
samples	Istio提供的微服务样例，比如bookinfo。
security	Istio用户身份验证、服务间认证。
tests	测试用例、脚本等。
vendor	dep生成的第三方依赖。
Gopkg.*	dep需要version constraint和version lock文件。
Makefile	Istio Makefile，编译docker镜像时会引用tools/istio-docker.mk这个Makefile。

其他

参考：

- https://github.com/istio/istio
- Istio源码解析系列part1—Istio源码架构介绍及开发环境搭建 https://www.servicemesher.com/blog/istio-deepin-part1-framework-and-environment/
- Istio 安装 https://learnku.com/articles/70067
- 官方文档-Istio 安全介绍 https://istio.io/latest/zh/docs/concepts/security/

【项目】Pixiu 可观测性

【项目】服务注册发现

【项目】配置中心

01 Pixiu配置中心支持Nacos

技术方案（一期）

背景

当前 pixiu 的配置，启动配置项（conf.yaml），日志配置（log.yml）均是从本地文件加载，没有接入动态配置中心的能力，而日常生产环境中，配置通过配置中心集中下发管理是更为灵活和安全的方式之一。

动态配置下发的场景还有很多扩展，接入的组件也因公司的选型有所不同，当前我们主要考虑接入 nacos 配置中心。

注：在设计上可以考虑，如何在二次开发场景下更为“标准、安全、简单”的接入其他配置中心。

目的

一期目的：

- Pixiu 启动时从配置中心Nacos 拉取配置 conf.yaml 数据。

二期目的：

- 动态托管/下发用户自定义配置
- 动态下发Pixiu配置

功能

梳理步骤

- [ ] Pixiu 启动流程，加载 conf.yaml 过程梳理
- [ ] Nacos Client 动态配置接口梳理 nacos-sdk-go readme
- [ ]

技术方案

架构图

@岳枫博

流程图

启动流程图

pixiu.go.main-->cobra.command.go.execute.(c.Run(c, argWoFlags))-->gateway.go.(Run: func(cmd *cobra.Command, args []string)).(bootstrap, meta, err := initApiConfig())-->pixiu.go.initApiConfig()-->

config_load.go.Load-->configLoadFunc LoadFunc = LoadYAMLConfig-->LoadYAMLConfig()

gateway.go.server.Start(bootstrap)-->pixiu_start.go.Start()-->server = NewServer()-->server.initialize(bs)-->server.Start()-->conf := config.GetBootstrap(),

http.ListenAndServe(addr.Address+":"+strconv.Itoa(addr.Port), nil)-->server.startWG.Wait()

核心技术设计

Pixiu-Nacos 配置中心下发接口设计

国强

Nacos 动态配置下发示例 nacos-sdk-go/main.go at master · nacos-group/nacos-sdk-go

```go
//get config
content, err := client.GetConfig(vo.ConfigParam{
        DataId: "test-data",
        Group:  "test-group",
    })
```

Pixiu 接入配置中心标准化 Listener接口设计

（二期）

Pixiu Nacos Config Client 抽象适配

（二期）

项目过程

开发分支： feature/nacos_config （https://github.com/PhilYue/dubbo-go-pixiu/tree/feature/nacos_config）

01 2022半年技术债计划

@岳枫博@菜夹膜

代码重构，健壮性。

梳理步骤：

- 根据 pixiu 的架构、分层，基于核心功能 + 代码核心能力（register，remote client，），分离功能模块
- 分析梳理评估
- 拆分子任务，指定负责人设计技术方案
- 方案评估
- 进入开发
- 测试上线

相关 Issue or PR：

- develop分支上，代码重复（完全一模一样代码） https://github.com/apache/dubbo-go-pixiu/issues/445
-

Dubbo/Triple代理优化

背景

现有dgp.filter.http.dubboproxy通过泛化调用代理dubbo，注册发现/路由/负载均衡均内置在dubbo-go的泛化调用中，无法利用到Pixiu现有的Router和Cluster的能力。

另一方面，triple代理目前不支持应用级服务发现，而且仍需要adapters+api_config来路由

目标

1. 废弃adapters+api_config结合将信息写到Router的方式
2. dgp.filter.http.directdubboproxy直接代理能力扩展，配合Router和Cluster完成dubbo代理
3. triple代理配合Router和Cluster完成dubbo代理，同时支持接口级与服务级代理
4. dubbo/triple协议代理/group/version可指定也可按约定根据元数据自动选择
5. dubbo/triple代理客户端连接池管理
6. dubbo/triple统一调用方式
7. 支持nacos/zk
8. 找不到路由时的友好提示
9. 增加丰富的samples演示以上功能

How to

代理所需的信息即一个${appName}/${className}/${method} + group + version 即三元组应该被路由到哪个Cluster，最后从Cluster中选择一个Ip后，借助dgp.filter.http.directdubboproxy即可完成一次代理。

这部分信息可以从注册中心拿到，其中${appName}/${className}/${method} + group + version作为路由，即Router(三元组)->Cluster(EndPoints)。

以Nacos为例

配置文件：registry/nacos/go-server/conf/dubbogo.yml（https://github.com/apache/dubbo-go-samples）

元数据

接口级

服务列表

服务名	分组名称
providers:org.apache.dubbo.UserProvider.Test2:myInterfaceVersion:myInterfaceGroup	myGroup

服务详情

IP	端口	健康状态	元数据
192.168.128.1	20000	true	methods=GetUser interface=org.apache.dubbo.UserProvider.Test2 path=/org.apache.dubbo.UserProvider.Test2 protocol=dubbo group=myInterfaceGroup version=myInterfaceVersion application=myApp
192.168.128.1	20001	true	methods=GetUser interface=org.apache.dubbo.UserProvider.Test2 path=/org.apache.dubbo.UserProvider.Test2 protocol=dubbo group=myInterfaceGroup version=myInterfaceVersion application=myApp

应用级

服务名	分组名称

myApp	myGroup

应用详情

IP	端口	元数据
192.168.128.1	20001	...
192.168.128.1	20000	...

服务映射

Data Id	Group
org.apache.dubbo.UserProvider.Test2	mapping

服务映射详情（应用名）

myApp,myApp2,myApp3

实现

对于接口级，考虑将${appName}/${className}/${method} + group + version拼接起来做为一个Router项，并将EndPoints生成一个Cluster

对于应用级，则更为简单，以${appName}做为Router项，并将EndPoints生成一个Cluster即可。

接口级注册则可利用协议元数据中`protocol=dubbo/triple`自动选择dubbo/triple client调用。

调研 | 负载均衡算法

> 调研了 nginx、haproxy、envoy、spring cloud 四款产品中的负载均衡算法。

加权法

- [x] 轮询法 nginx spring cloud haproxy envoy

将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

- [ ] 加权轮询法 nginx envoy

不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

- [ ] 最小连接数法 nginx spring cloud haproxy envoy

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前。积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负责合理地分流到每一台服务器。

- [ ] 加权最少连接数法 envoy

根据当前连接数和权重比例选择最佳的后端服务器进行请求处理。

- [ ] 响应时间加权法 nginx spring cloud

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

随机法

- [x] 随机法 spring cloud envoy

通过系统的随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。

哈希法

- [ ] 源地址哈希法/一致性哈希法 nginx haproxy

源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

- [ ] uri 哈希法 haproxy

根据请求 URI 的 hash 值，将同一个 URI 的请求分配给同一个后端服务器。

- [ ] url 哈希法 nginx haproxy

按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，后端服务器为缓存时比较有效。

- [ ] 请求参数哈希法 spring cloud

根据请求参数的 hash 值，选择一个服务实例进行请求处理。同一个参数的请求总是分配给同一个实例。

- [ ] MAGLEV 算法 envoy

由 Google 发明，目的是为了解决现有负载均衡算法的扩展性和性能问题。

MAGLEV 算法的核心思想是将请求映射到一组虚拟桶（Virtual Buckets），每个桶又映射到一个实际的后端服务器。每个虚拟桶使用哈希函数计算出一个哈希值，该哈希值可以与后端服务器列表中的一个唯一 ID 相对应，然后将请求路由到该服务器上。

与传统的哈希算法不同，MAGLEV 算法使用了多个哈希函数，以增加桶分布的均匀性和冲突的概率。在路由请求之前，MAGLEV 算法会将请求按照所有哈希函数的结果排序，并选择一个第一个未被占用的桶，然后将该请求路由到该桶对应的服务器上。

- [x] RING HASH 算法 envoy

RING HASH 算法常被用于解决分布式系统中的负载均衡问题，其主要思想是将请求哈希到一个环状结构中，然后在该环上选择一个离请求哈希值最近的节点作为路由目标。

RING HASH 算法的核心思想是将节点的 IP 地址或主机名通过哈希函数映射到一个环上，然后将请求哈希到环上的一个位置，最后选择离请求哈希值最近的节点进行路由。具体实现过程如下：

1. 将所有节点的 IP 地址或主机名通过哈希函数映射到一个范围在 [0, 2^32−1] 的整数值上，称为哈希值
2. 将所有节点的哈希值按照顺时针排序，并构建成一个环状结构
3. 将请求的哈希值映射到环上的一个位置，然后按照顺时针方向选择离该位置最近的节点进行路由

Pilot-Metadata

服务发现元数据

CRD 定义

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: metadata.networking.dubbo.io
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: metadata.networking.dubbo.io
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                applicationName:
                  type: string
                revision:
                  type: string
                metadataInfo:
                  type: string

  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: metadata
    # singular name to be used as an alias on the CLI and for display
    singular: metadata
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: Metadata
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - md
```

Proto

```
Plain Text
```

```
syntax = "proto3";

package pixiu.pkg.metadata;

option go_package = "pixiu/pkg/metadata/protos"

service MetadataService {
    rpc Publish(MetadataPublishRequest) returns (MetadataPublishResponse);
    rpc Get(GetMetadataRequest) returns (GetMetadataResponse);
}

message PublishMetadataRequest {
    string application_name = 1;
    string revision = 2;
    string metadata_info = 3;
}

message PublishMetadataResponse {
}

message GetMetadataRequest {
    string application_name = 1;
    string revision = 1;
}

message GetMetadataResponse {
    string metadata_info = 1;
}

```

流程设计

1. 服务启动时上报元数据到控制面，通过控制面注册到 k8s CRD 中
2. 监听服务下线，对于 application 维度下的所有服务均下线时，移除对应的元数据

- ~~在服务发现中增加监听器监听对应的上下线事件~~
- 为元数据设置一个时间戳，控制面以一天的频率去监听这部分数据的更新状态，客户端以更高的频率上报数据，若在一天之内的数据没有更新则视作服务下线，将对应的元数据删除。

存疑

1. 由于控制面不关心 revision 参数的生成，需要考虑出现重复的情况
   - ~~在数据面更换其他的哈希算法~~
   - 直接更新对应 revision 的数据
2. ~~服务下线时的监听~~

服务定义

CRD

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: service-definition.networking.dubbo.io
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: service-definition.networking.dubbo.io
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                identifier:
                  type: string
                serviceDefinition:
                  type: string

  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: serviceDefinitions
    # singular name to be used as an alias on the CLI and for display
    singular: serviceDefinition
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: ServiceDefinition
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - sd
```

Proto

```yaml
syntax = "proto3";

package pixiu.pkg.metadata;

option go_package = "pixiu/pkg/metadata/protos"

service ServiceDefinitionService {
    rpc Publish(ServiceDefinitionPublishRequest) returns (ServiceDefinitionPublishResponse);
    rpc Get(GetServiceDefinitionRequest) returns (GetServiceDefinitionResponse);
}

message PublishServiceDefinitionRequest {
    string identitfier = 1;
    string serivce_definition = 2;
}

message PublishServiceDefinitionResponse {
}

message GetServiceDefinitionRequest {
   string identitfier = 1;
}

message GetServiceDefinitionResponse {
    string serivce_definition = 1;
}

```

注

1. proto 中的错误信息处理均通过 grpc error handling 进行处理
2. 元数据中心以及服务定义中的数据保存时只保存原始字符串?

参考连接

1. K8s 的核心是 API 而非容器：从理论到 CRD 实践（2022） http://arthurchiao.art/blog/k8s-is-about-apis-zh/#33-api-%E6%98%AF-sql

Pixiu Control Plane服务测试功能基本方案

1.采用grpccurl的请求控制面

JSON格式(参数待定)

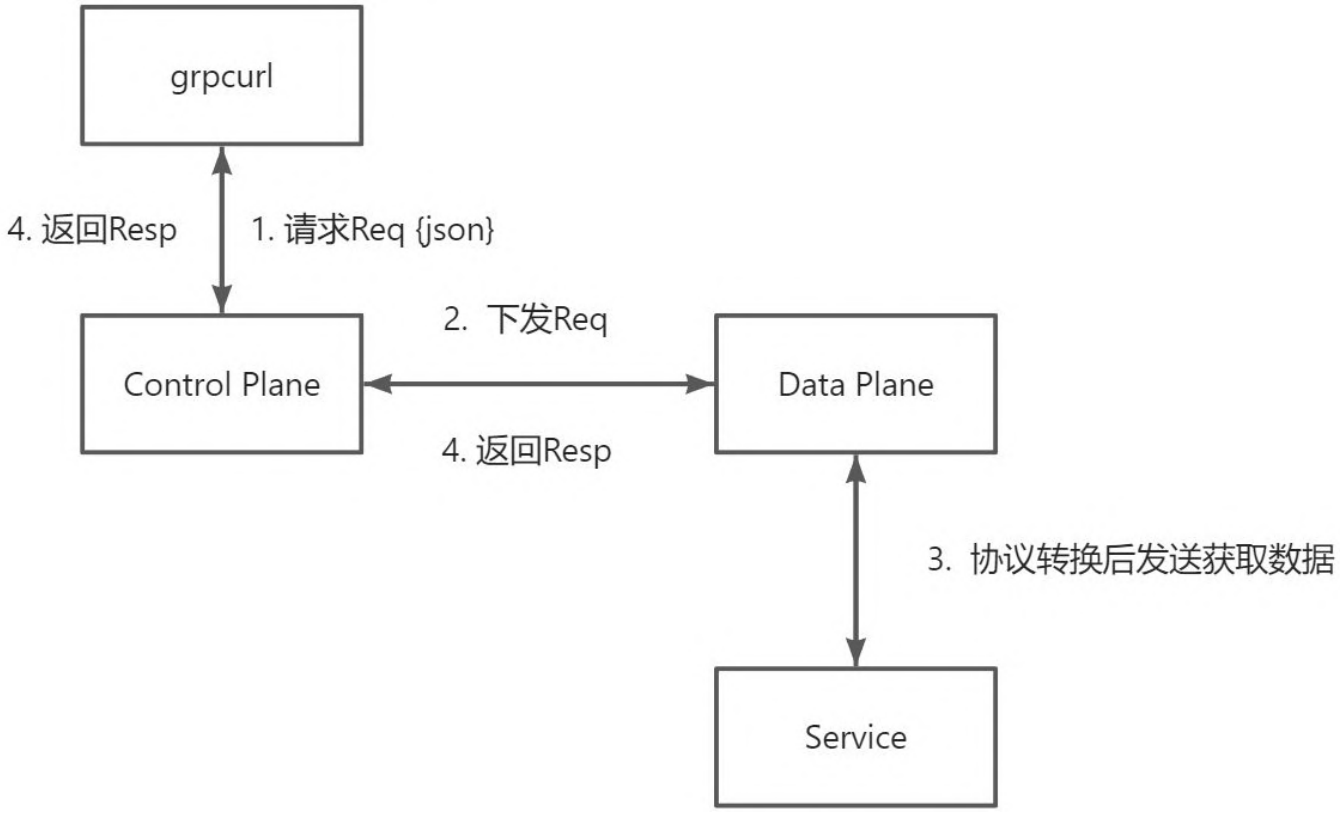
2.控制面向数据面下发请求

3.进行协议转换

4.数据面返回相关数据给控制面(数据格式待定)

5.控制面返回数据给gtpccurl

注：参考功能类似 Dubbo Admin 的服务测试。



Polit-服务映射

引语

服务映射的逻辑是提供一个 com.example.DemoInterface => demo-application 的查找逻辑，此信息由Client（provider）主动上报，由consumer按interface的颗粒度订阅，用以根据interface查找provider的endpoints。

实现

对于基本的服务网格体系，首选的也是目前可以优先实现的选项是存储至k8s的注册中心。对此唯一的方式是创建一条该类型的CRD（k8s不允许执行操作其集群的etcd），然后自定义controller，通过informer来监听资源变动，大部分代码都可以用工具生成。具体地，分为两步。

1.上报

Xds本身获取的信息大都是声明式的，所以并无客户端主动上报的途径。所以在xds的grpc server上注册一个新的service name mapping服务，用来给客户端主动上报。

存在的问题：

1.鉴权

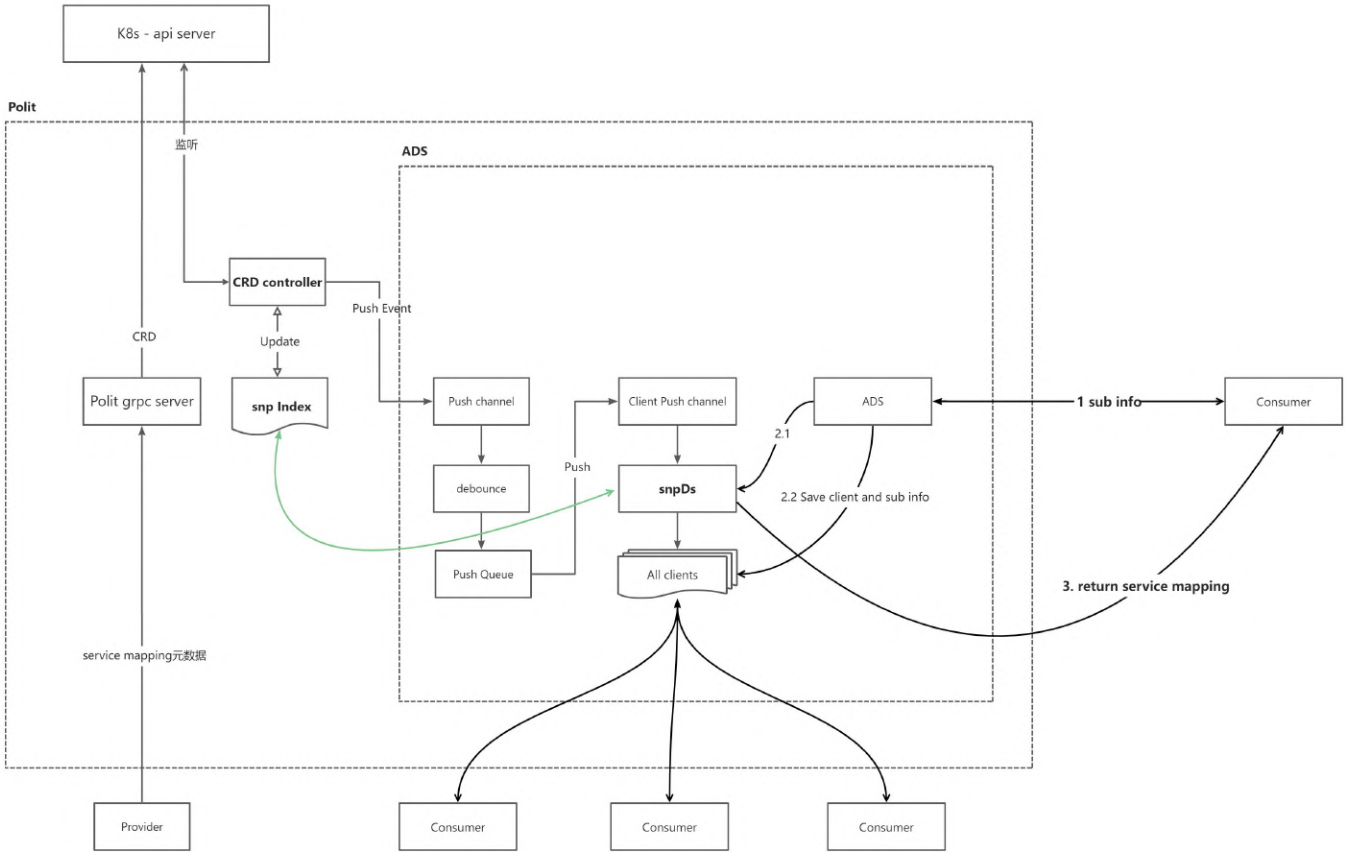
2.app name是什么，此处可能应该是一个完全限定名，例如 demoapp-default.pod.cluster.local

2.存储

yaml YAML

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: servicenamemappings.networking.dubbo.io
spec:
  group: networking.dubbo.io
  scope: Namespaced
  names:
    kind: ServiceNameMapping
    plural: servicenamemappings
    singular: servicenamemapping
    shortNames:
      - snp
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              description: >-
                it's the spec of Service Name Mapping, which is used to map the interface name
                to the application name in application discovery.
              type: object
              properties:
                interfaceName:
                  type: string
                applicationName:
                  type: array
                  items:
                    type: string
            status:
              type: object
      # subresources for the custom resource
      subresources:
        # enables the status subresource
        status: { }
```

3.监听与下发


K8s - api server
Polit
监听
ADS
CRD controller
Push Event
CRD
Update
Polit grpc server
snp Index
Push channel
Client Push channel
ADS
1 sub info
Consumer
2.1
debounce
Push
snpDs
2.2 Save client and sub info
Push Queue
All clients
3. return service mapping
service mapping元数据
Provider
Consumer
Consumer
Consumer

Dubbo-go-Pixiu适配Nacos配置中心设计文档

文档说明

> 该文档为Dubbo-go-pixiu适配Nacos的设计文档

-

适配参考

> 之前使用nacos作为服务注册发现中心的nacos配置很简单，
>
> "nacos":
>
> protocol: nacos
>
> address: "127.0.0.1:8848"
>
> timeout: "5s"

方式：

- Load函数时从本地文件路径读取配置的
- 从远端nacos读取yaml配置生成bootstrap结构体就行了
- 远端nacos的host和password等信息可以作为cmd的参数传入

参考配置：

https://github.com/apache/dubbo-go-pixiu/blob/develop/pkg/config/config_load.go

.

读取配置文件

> 为达成读取配置文件的目标，可以使用Nacos-client-go SDK

举例：

读取Nacos配置文件 Go

```go
package main

import (
    "fmt"
    "github.com/nacos-group/nacos-sdk-go/clients"
    "github.com/nacos-group/nacos-sdk-go/common/constant"
    "github.com/nacos-group/nacos-sdk-go/vo"
    "time"
)

func main() {
    // 至少一个ServerConfig
    serverConfigs := []constant.ServerConfig{
        {
            IpAddr:      "192.168.72.146",
            Port:        8848,
        },
    }

    // 创建clientConfig
    clientConfig := constant.ClientConfig{
        NamespaceId:         "1cf91be1-d0e3-4494-aef7-b3cb8177e04e", // 如果需要支持多namespace，我们可以场景多个client,它们有不同的NamespaceId。当namespace是public时，此处填空字符串。
        TimeoutMs:           5000,
        NotLoadCacheAtStart: true,
        LogDir:              "tmp/nacos/log",
        CacheDir:            "tmp/nacos/cache",
        RotateTime:          "1h",
        MaxAge:              3,
        LogLevel:            "debug",
    }
    // 创建动态配置客户端的另一种方式（推荐）
    configClient, err := clients.NewConfigClient(
        vo.NacosClientParam{
            ClientConfig:  &clientConfig,
            ServerConfigs: serverConfigs,
        },
    )
    if err != nil {
        panic(err)
    }
    //获取配置信息
    //content, err := configClient.GetConfig(vo.ConfigParam{
    //  DataId: "user-web.yaml",
```

```
    //  Group:  "dev"})
    //if err != nil {
    //  fmt.Println("GetConfig err: ",err)
    //}

    //监听配置
    err = configClient.ListenConfig(vo.ConfigParam{
        DataId: "user-web.yaml",
        Group:  "dev",
        OnChange: func(namespace, group, dataId, data string) {
            fmt.Println("group:" + group + ", dataId:" + dataId + ", data:" + data)
        },
    })
    if err!=nil{
        return
    }
    time.Sleep(time.Second*1000)
}
```

如上述代码的的方式去创建

总结

> 可以将两者结合起来

泛化调用直连和删除 api_config

- api_config 删除，使用 route 和 cluster 代替
- 泛化调用从服务发现模式修改成直连模式

Java

```
- name: dgp.filter.http.apiconfig
config:
  dynamic: true
  dynamic_adapter: test
- name: dgp.filter.http.dubboproxy
config:
  dubboProxyConfig:
    registries:
      "zookeeper":
        protocol: "zookeeper"
        timeout: "3s"
        address: "127.0.0.1:2181"
        username: ""
        password: ""
```

将 api_config 删除，使用 route 和 cluster代替

目前 http to dubbo 协议转换时，需要使用 api_config 配置从 http 请求中获取诸如 application，interface，method 等元信息，从而为 dubbo 泛化调用做准备。

其使用必须使用本地的 api_config.yaml 文件进行配置，并且需要为每个path 进行相应的配置。如下所示，具体可见 samples/dubbogo/simple/body,query,proxy。

```yaml
  - path: '/api/v1/test-dubbo/user'
    type: restful
    description: user
    methods:
      - httpVerb: POST
        enable: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
        integrationRequest:
          requestType: dubbo
          mappingParams:
            - name: requestBody._all
              mapTo: 0
              mapType: "object"
          applicationName: "UserProvider"
          interface: "com.dubbogo.pixiu.UserService"
          method: "CreateUser"
          group: "test"
          version: 1.0.0
          clusterName: "test_dubbo"
```

缺点：

- 泛化调用服务发现模式还会进行一次服务发现，增加复杂度并且影响性能；
- xDS 协议等需要进行统一，api_config 无法兼容。
- 负载均衡机制不统一
- 现在已经有 http to dubbo 默认转换规约。详见 https://dubbogoproxy.yuque.com/docs/share/0c2f8c7a-42a2-4a83-9a0a-8a4f302622a9?# 《Dubbo to http 默认转换规则》

所以希望将 api_config 删除掉或进行简化。使用 route 确定请求会转发给哪个上游 cluster 的 endpoint，然后 dubboproxy 使用默认转换规约从 http 请求中获取到泛化调用所需的元信息，再加上 endpoint 的 host 信息，进行泛化请求的直连调用。

涉及改动：

- https://github.com/apache/dubbo-go-pixiu/tree/develop/pkg/filter/http/apiconfig
- https://github.com/apache/dubbo-go-pixiu/tree/develop/pkg/adapter

泛化直连调用参考：

https://github.com/apache/dubbo-go-pixiu/blob/develop/pkg/filter/network/dubboproxy/filter/proxy/proxyfilter.go

HTTP-to-gRPC 支持 Reflection

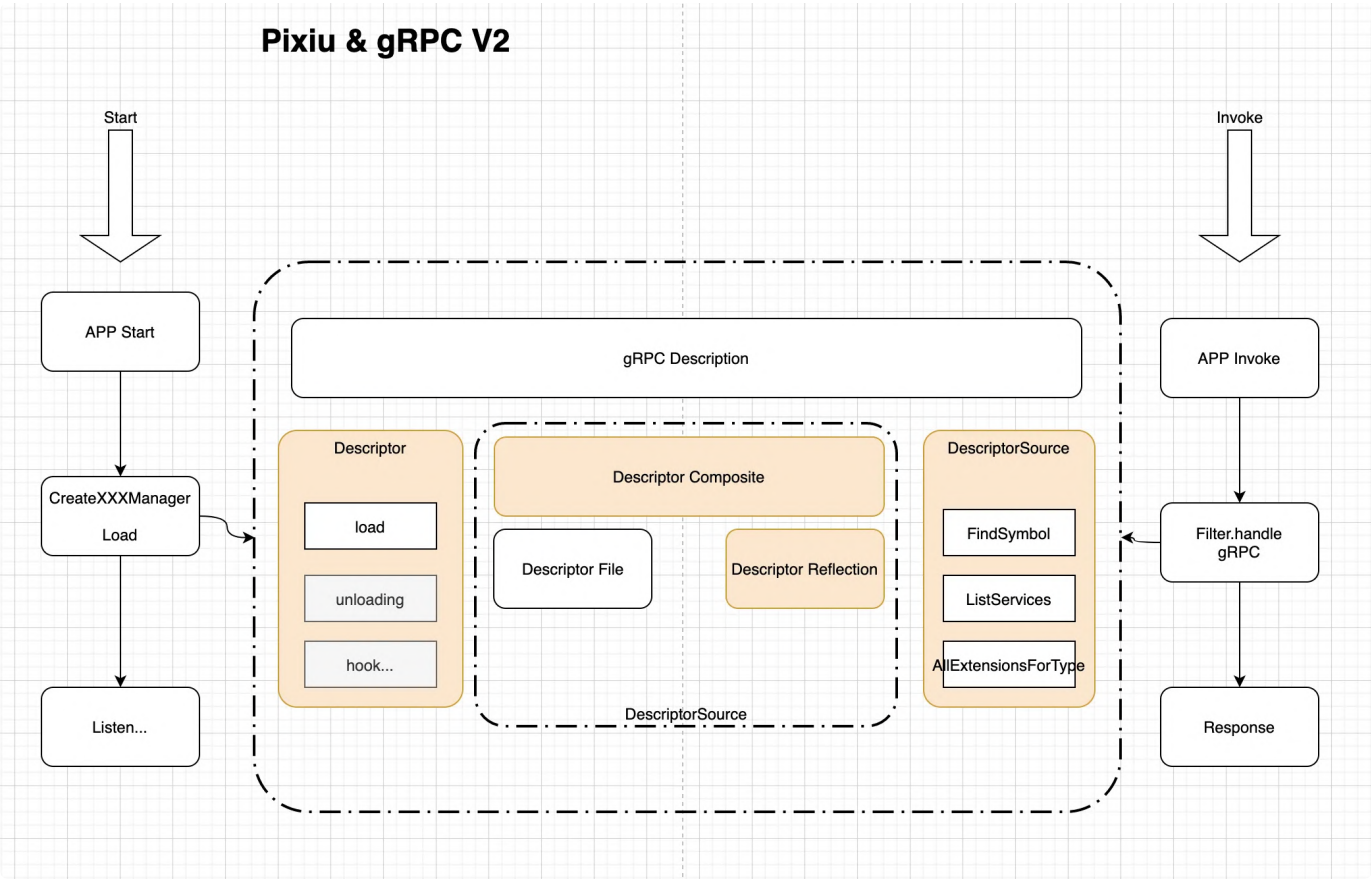
概述

该版本主要是在 http-to-gRPC 的场景，支持 gRPC **Reflection，pixiu 不用加载 proto 文件，也可以正常代理 http to gRPC 的请求，前提是，gRPC 服务必须开启** Reflection，即**在当前 gRPC 服务上注册服务器反射服务。**

注册服务器反射服务通过代码：`reflection.Register(gs)`，启动代码完整示例如下：

```go
func main() {
    l, err := net.Listen("tcp", ":50001") //nolint:gosec
    if err != nil {
        panic(err)
    }

    s := &server{users: make(map[int32]*proto.User)}
    initUsers(s)

    gs := grpc.NewServer()

    proto.RegisterUserProviderServer(gs, s)

    // registers the server reflection service on the given gRPC server.
    reflection.Register(gs)

    logger.Info("grpc test server is now running...")
    err = gs.Serve(l)
    if err != nil {
        panic(err)
    }
}
```

基于 pixiu 支持 gRPC 的迭代升级方案



功能

- http to gRPC 场景，不配置 proto 文件，基于 Reflection Server 可以完整正常代理请求
- http to gRPC 场景，配置 proto 文件，未开启 Reflection Server 也可以完整正常代理请求
- http to gRPC 场景，配置 proto 文件，并且服务端也开启了 Reflection Server ，pixiu 可以完整正常代理请求

Pixiu to SpringCloud with zookeeper

项目 README.md

操作手册

启动 Zookeeper 作为 SpringCloud 的注册中心

```Shell
services:
  zookeeper:
    image: zookeeper
    ports:
      - "2181:2181"
```

启动 SpringCloud Server： `pixiu-springcloud-server`，项目路径： `samples/springcloud/zookeeper`

```Shell
mvn spring-boot:run
```

检查一下 SpringCloud 服务是否启动：

```Shell
curl http://localhost:9127/hi
```

响应结果，表示 SpringCloud Server 正常启动：

```
Hello Pixiu World! from org.springframework.cloud.zookeeper.serviceregistry.ServiceInstanceRegistration@63a45760
```

Start Pixiu

develop-0.5.0 分支 配置文件： conf.yaml

```
gateway start -c samples/springcloud/pixiu/conf.yaml
```

通过 Pixiu 代理请求 SpringCloud 服务

```
curl http://127.0.0.1:8888/pixiu-springcloud-server/hi
```

Dubbo to http 默认转换规则

我们以 Http to dubbo 默认转换规则为例，将 dubbo 请求转换为 http 请求。

接收 dubbo 请求的实现：

为了能接收 dubbo 协议请求，pixiu 引入了 tcp_listener,它使用getty，接收tcp报文，然后交给 dgp.filter.network.dubboconnectionmanager 来解序列化为 rpc_invocation 实例，也就是 dubbogo 中代表依次 rpc 请求的实例。然后交给 dgp.filter.dubbo.proxy 使用 URI 和 Invoker 来发送该 invocation 给 upstream 的 triple server，然后将 rpcresult 返回。

具体代码可见：

- pkg/listener/tcp
- pkg/filter/network/dubboproxy/manager.go 的 OnEncode，OnDecode，OnData 三个函数

具体转换实现代码见：pkg/filter/network/dubboproxy/filter/http/httpfilter.go
案例：samples/dubbohttpproxy，可以启动 pixiu 和 对应的 http server，然后执行 dubbo2http_test.go

具体转换规则：

```
Go
POST ${host}:${port}/#{interfaceKey}/#{method}

  "x-dubbo-http1.1-dubbo-version": 1.0.0
"x-dubbo-service-protocol": #{protocol}
"x-dubbo-service-version": #{version}
"x-dubbo-service-group": #{group}

#{body}
```

其中 ${host} 和 ${port} 是根据 route 指向 cluster 后，选取的 endpoint 的信息，#{xx}则是从 dubbo 请求中获取的信息。

其中 body 应该就是泛化调用的参数，泛化调用和 http server 接口处理的对应代码如下所示。

```go
tripleRefConf := newDubboRefConf("com.dubbogo.pixiu.TripleUserService", dubbo.DUBBO)
resp, err := tripleRefConf.GetRPCService().(*generic.GenericService).Invoke(
    context.TODO(),
    "GetUserById",
    []string{"java.lang.String"},
    []hessian.Object{"0001"},
)
```

```go
func main() {
    http.HandleFunc("/com.dubbogo.pixiu.TripleUserService/GetUserById", user)
    log.Println("Starting sample server ...")
    log.Fatal(http.ListenAndServe("127.0.0.1:20001", nil))
}

func user(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case constant.Post:
        byts, err := ioutil.ReadAll(r.Body)
        if err != nil {
            w.Write([]byte(err.Error()))
        }
        var name string
        var user User
        err = json.Unmarshal(byts, &name) // 解析出来为 "0001"
        if err != nil {
            w.Write([]byte(err.Error()))
        }
        .......
    }
}
```

gRPC 请求代理

pixiu 作为网关，代理外侧客户端发送的 gRPC 请求，按照其请求的 path 根据route 确定转发的 upstream 服务，构造对应的 http2 请求进行转发。目前仅支持 unary call。

涉及的组件有 Http2Listener 用来接收 http2 请求，GrpcConnectionManager 用来处理 http2 报文，生成新的 http2 请求，转发请求，将返回的response的body，header和tailer写回。

具体案例：samples/grpc

YAML

```yaml
static_resources:
  listeners:
    - name: "net/http"
      protocol_type: "HTTP2"  # 需要使用 HTTP2 的 Listener
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8881
      filter_chains:
          filters:
            - name: dgp.filter.grpcconnectionmanager  # 需要使用 GrpcConnectionManager
              config:
                route_config:
                  routes:
                    - match:
                        prefix: "/provider.UserProvider/"
                        # 表示 provider.UserProvider相关path的grpc请求都转发到 test-grpc cluster 上
                      route:
                        cluster: "test-grpc"
                        cluster_not_found_response_code: 505
```

测试用例：

序号	测试描述	结果	备注
1	启动samples的server，和pixiu，启动test，查看单测是否正式通过	**PIXIU返回结果** 2022-02-25T02:47:23.427+0800 [35mDEBUG [0m grpc/manager.go:69 [dubbo-go-pixiu] client choose endpoint from cluster :test-grpc **单侧的控制台** === RUN TestGet --- PASS: TestGet (0.11s) PASS	
2	将 server 关闭，启动test，查看pixiu 是否能否处理 upstream down场景，应该返回 connect refused	**PIXIU返回结果** 2022-02-25T02:43:30.066+0800 [34mINFO [0m grpc/manager.go:90 GrpcConnectionManager forward request error dial tcp 127.0.0.1:50001: connectex: No connection could be made because the target machine actively refused it.	
3	启动server和pixiu，client端发送不在pixiu配置的可识别的 grpc请求，查看pixiu 的处理，应该返回404		

4	启动server和pixiu，client发送请求较大的request，查看是否正常，可以参考grpc benchmark 的案例 funcDoUnaryCall(tc testgrpc.BenchmarkServiceClient, reqSize, respSizeint) error { ，将 request的参数之一设置为可控大小		

Http to Dubbo默认转化规则

http to dubbo 协议是 pixiu 作为重要的功能之一，之前需要从 api_config.yaml 文件或者从注册中心中读取元数据生成对应的 api_config 实例。
对于大量的dubbo api，就需要大量的 api_config 配置，相对较为麻烦。
所以，考虑从 http 请求的 path 或者 header 中读取到对应的元信息，进行 dubbo 泛化调用。

具体的转换规则详见
https://www.yuque.com/docs/share/4f82b71e-fa2b-4e65-8007-ba3afe2a1740?#
目前 protocol 只能支持 dubbo

技术实现：
在 dgp.filter.http.dubboproxy 中添加一个bool 开关，开启后，会从 http 请求获取原数据

YAML

```
func (f *Filter) Decode(c *contexthttp.HttpContext) filter.FilterStatus {

  if f.conf.Dpc.AutoResolve {
    if err := f.resolve(c); err != nil {
      c.SendLocalReply(http.StatusInternalServerError, []byte(fmt.Sprintf("auto resolve err: %s", err)))
      return filter.Stop
    }
  }

  api := c.GetAPI()
  .....
  }
```

案例：
samples/dubbogo/simple/resolve

测试用例

序号	测试用例	结果	备注

	启动对应的samples中的server 和 pixiu，单测能够正常通过		
	启动对应的samples中的server 和 pixiu，将单测中的http请求中的version和group进行修改，查看是否生效(对应server配置的group和version也要修改)		
	启动对应的samples中的server 和 pixiu，将单测中的http 请求的 interface 和 method 进行修改，pixiu 应该能返回404		

triple 和 dubbo 协议的相互转换

添加了 triple 和 dubbo 协议的相互转换，为了做 dubbo2 集群和 dubbo3 集群的中转，方便推进整体 dubbo 集群分批升级。

为了能接收 dubbo 协议请求，pixiu 引入了 tcp_listener,它使用getty，接收tcp报文，然后交给 dgp.filter.network.dubboconnectionmanager 来解序列化为 rpc_invocation 实例，也就是 dubbogo 中代表依次 rpc 请求的实例。然后交给 dgp.filter.dubbo.proxy 使用 URl 和 Invoker 来发送该 invocation 给 upstream 的 triple server，然后将 rpcresult 返回。

为了接收 triple 协议请求，pixiu 引入了 triple_listener，它使用 triple 的 server proxy 功能，可以接受任意的triple请求，然后同样交给 dgp.filter.dubbo.proxy 处理。同样的也可以交给 dgp.filter.dubbo.http来将其转换为 http 请求。

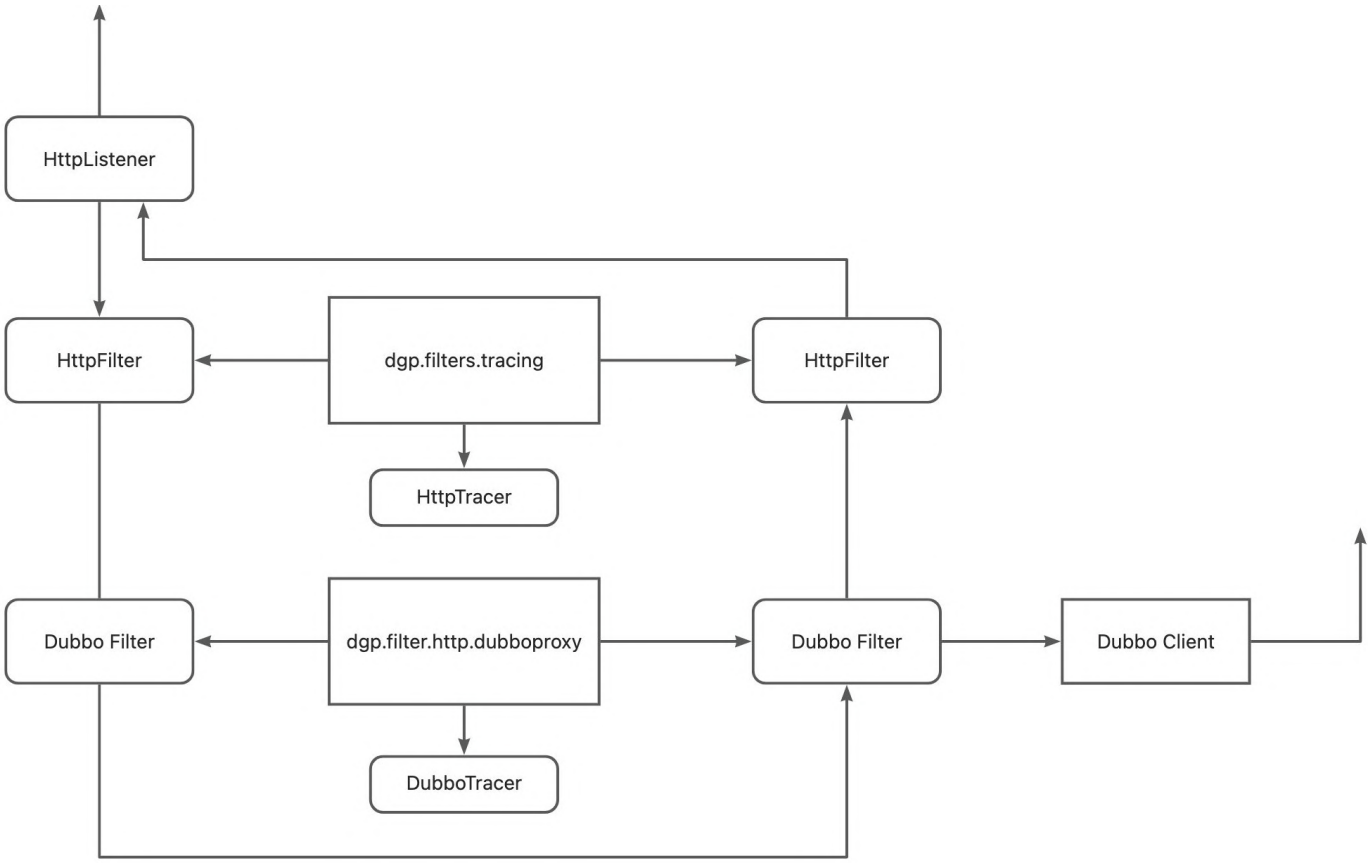
案例：具体看
案例：samples/dubbohttpproxy
案例：samples/dubbotripleproxy https://github.com/apache/dubbo-go-pixiu/tree/develop/samples/dubbotripleproxy

```yaml
  listeners:
    - name: "dubbo-listener"
      protocol_type: "TCP"  # tcp_listner
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8888
      filter_chains:
          filters:
            - name: dgp.filter.network.dubboconnectionmanager
              config:
                route_config:
                  routes:
                    - match:
                        prefix: "/com.dubbogo.pixiu.TripleUserService"
                        methods:
                          - "*"
                      route:
                        cluster: "triple-server"
                        cluster_not_found_response_code: 505
                dubbo_filters:
                  - name: dgp.filter.dubbo.proxy # dubbo.proxy 使用 tri 来转换为 triple协议请求
                    config:
                      protocol: tri
    - name: "triple-listener"
      protocol_type: "TRIPLE" # triple_listener
      address:
        socket_address:
          address: "0.0.0.0"
          port: 9999
      filter_chains:
        filters:
          - name: dgp.filter.network.dubboconnectionmanager
            config:
              route_config:
                routes:
                  - match:
                      prefix: "com.dubbogo.pixiu.DubboUserService"
                      methods:
                        - "*"
                    route:
                      cluster: "dubbo-server"
                      cluster_not_found_response_code: 505
              dubbo_filters:
```

```
                - name: dgp.filter.dubbo.proxy
                  config:
                    protocol: dubbo # ubbo.proxy 将其转换为 dubbo 请求
  clusters:
    - name: "triple-server" # triple-server 的集群
      lb_policy: "lb"
      endpoints:
        - id: 1
          socket_address:
            address: 127.0.0.1
            port: 20001
    - name: "dubbo-server"  # dubbo-server 的集群
      lb_policy: "lb"
      endpoints:
        - id: 1
          socket_address:
            address: 127.0.0.1
            port: 20000
```

序号	测试用例	结果	备注
1	dubbotripleproxy 案例运行起来，两个单测能正常通过		
2	只将 pixiu启动，运行两个单测，pixiu 应该返回正常的错误信息		

trace in pixiu



pixiu中tracing实现了http流量的追踪，通过设置dgp.filters.tracing，可以追踪http流量。设置如下：

```yaml
static_resources:
  listeners:
    - name: "net/http"
      protocol_type: "HTTP"
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8881
      filter_chains:
        filters:
          - name: dgp.filter.httpconnectionmanager
            config:
              route_config:
                routes:
                  - match:
                      prefix: "/api/v1"
              http_filters:
                - name: dgp.filters.tracing
                  config:
                - name: dgp.filter.http.apiconfig
                  config:
                    path: $PROJECT_DIR/pixiu/api_config.yaml
                - name: dgp.filter.http.dubboproxy
                  config:
                    dubboProxyConfig:
                      registries:
                        "zookeeper":
                          protocol: "zookeeper"
                          timeout: "3s"
                          address: "127.0.0.1:2181"
                          username: ""
                          password: ""
                      timeout_config:
                        connect_timeout: 5s
                        request_timeout: 5s
              server_name: "test_http_dubbo"
              generate_request_id: false
tracing:
  name: "jaeger"
  servicename: "dubbo-go-pixiu"
  sampler:
    type: "always"
  config:
    url: "http://127.0.0.1:14268/api/traces"
```

然而，要实现整体流程的追踪和traceID的传递，需要实现dubbo流量的追踪，应该子啊dubboproxy中增加trace组件。

key of jeager when extract

from traceparent -> 00-8189d1cb128fb8fa475ad8c8966b0bfe-29334edeaa281415-01

to

JaegerDebugHeader = {string} "jaeger-debug-id"

JaegerBaggageHeader = {string} "jaeger-baggage"

TraceContextHeaderName = {string} "uber-trace-id"

TraceBaggageHeaderPrefix = {string} "uberctx-"

prometheus pull in pixiu

背景

pixiu 目前具备采集相关指标，发送给 pushgateway，从而接入 prometheus 的能力。用户可以根据自身业务需要，部署 prometheus，设置其每次从 pushgateway 拉取数据。这虽然已经完成了对基础指标的实时上报，但根据官方推荐和目前业界广泛使用的情况来看，prometheus pull 方式的实现仍是必要的。

流程

定义端口：在 pixiu 的配置文件中配置一个端口，以便 prometheus server 通过该端口访问 pixiu 的数据

定义指标：采取现在定义的请求总数、请求延时、请求体大小和返回体大小四个指标

指标上报：保持大部分现有逻辑，将收集到的数据不发送给 pushgateway，直接暴露出来

方式选择：可在配置文件配置使用 pull 还是 push 模式

方案

方案一

在程序加载的同时初始化所有的监控指标，开启一个协程定期采集和更新指标数据，同时新开一个协程每隔固定周期（30秒或1分钟）检查指标的状态，如果在5分钟内指标没有更新就自动清理掉。

优点

1. 避免并发采样带来的资源消耗和抢占
2. 易于控制采样频率
3. 可自动清理掉失效的采集对象
4. 避免了是过期数据，一定程度上保证了即时性

缺点

并不是即时采样，比如在采样协程夯住后存在一定的延时

方案二

每次接收到 pull 请求时，都初始化新的监控指标，开启一个采样协程，返回请求时结束掉协程。在请求期间，协程会去采集所有的指标数据上报。

优点

数据即时性和一致性

缺点

暂无，可能增加对资源的占用

Polit支持服务映射

Provider

对于Provider，启动时上报自身应用名与要注册的服务列表，尽量一次性以列表的形式上报，但是做不到也没关系，Polit会针对性的merge一部分请求。

对于dubbo-go，直接replace istio.io/api为https://github.com/dubbo-go-pixiu/operator-api/即可调用该grpc service，对于dubbo-java，可能需要将该proto文件下载并生成stub再调用。

API: https://github.com/dubbo-go-pixiu/operator-api/blob/release-1.14/dubbo/v1alpha1/snp.proto

proto Protobuf

```proto
// Provides an service for reporting the mapping relationship between interface => cluster
// the cluster name will be versioned FQDN. such as "demo.default.svc.cluster.local"
service ServiceNameMappingService{
  rpc registerServiceAppMapping(ServiceMappingRequest) returns (ServiceMappingResponse);
}

// When dubbo provider start up, it reports its applicationName and its interfaceName,
// and Dubbo consumer will get the service name mapping info by xDS.
message ServiceMappingRequest{
  // This is namespace of proxyless dubbo server
  string namespace = 1;
  string applicationName = 2;
  repeated string interfaceNames = 3;
}
```

Consumer

对于Consumer，与其他Xds协议使用的方式完全一致，订阅方式如下所示，其中typeUrl为固定值，resourceNames则为要订阅的interface，默认订阅同namespace接口，若需跨namespace订阅，形如‘a.b.c.HelloInterface|dubbo-demo’。

```
typeUrl = "dubbo.networking.v1alpha1.v1.servicenamemapping"
resourceNames = ['a.b.c.HelloInterface','a.b.c.HelloInterface|default',
'a.b.c.HelloInterface|dubbo-demo']
```

关于响应：

订阅时会异步地返回要订阅的服务的列表，如果有更新，会通过ads推送下来。

对于dubbo-go，直接将其转换为https://github.com/dubbo-go-pixiu/operator-api/blob/release-1.14/dubbo/v1alpha1/snp.pb.go中的ServiceMappingXdsResponse即可。

对于dubbo-java，使用API: https://github.com/dubbo-go-pixiu/operator-api/blob/release-1.14/dubbo/v1alpha1/snp.proto生成的对应ServiceMappingXdsResponse即可。

Polit服务注册压测

对于Polit，直接压QPS没什么意义，那就变成单纯的压测api-server，所以压测一下两个场景。

配置

一个单主节点，8c16G

场景1 单应用上线实例，并发100

一个新应用上线，应用有100个interface，验证上线实例的qps。

结论：足够满足需求，单应用上线600个实例无压力

请求数60000，并发数100

```
kyll/hey #gosetup
/private/var/folders/15/nzhp26690mj4bbvt0dm4jxcm0000gn/T/GoLand/___1go_build_github_com_rakyll_hey -n 60000 -c 100 http://127.0.0.1:8080
2022/12/18 22:48:48 start proxy server at 8080

Summary:
  Total:        91.0567 secs
  Slowest:      0.5886 secs
  Fastest:      0.0161 secs
  Average:      0.1516 secs
  Requests/sec: 658.9303

  Total data:   119996 bytes
  Size/request: 2 bytes

Response time histogram:
  0.016 [1]     |
  0.073 [617]   |■
  0.131 [21557] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.188 [29051] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.245 [6123]  |■■■■■■■■
  0.302 [1081]  |■
  0.360 [656]   |■
  0.417 [458]   |■
  0.474 [214]   |
  0.531 [175]   |
```

请求数60000，并发数500

```
kyll/hey #gosetup
/private/var/folders/15/nzhp26690mj4bbvt0dm4jxcm0000gn/T/GoLand/___1go_build_github_com_rakyll_hey -n 60000 -c 500 http://127.0.0.1:8080
2022/12/18 22:52:11 start proxy server at 8080

Summary:
  Total:        96.2595 secs
  Slowest:      1.7440 secs
  Fastest:      0.0169 secs
  Average:      0.8013 secs
  Requests/sec: 623.3154

  Total data:   119080 bytes
  Size/request: 2 bytes

Response time histogram:
  0.017 [1]     |
  0.190 [180]   |
  0.362 [359]   |
  0.535 [2327]  |■■■
  0.708 [6483]  |■■■■■■■
  0.880 [37043] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  1.053 [10528] |■■■■■■■■■■■
  1.226 [1975]  |■■
  1.399 [509]   |■
  1.571 [95]    |
```

场景二 多应用上线，并发100

100个新应用同时上线，每个应用100个实例

请求数10000，并发100

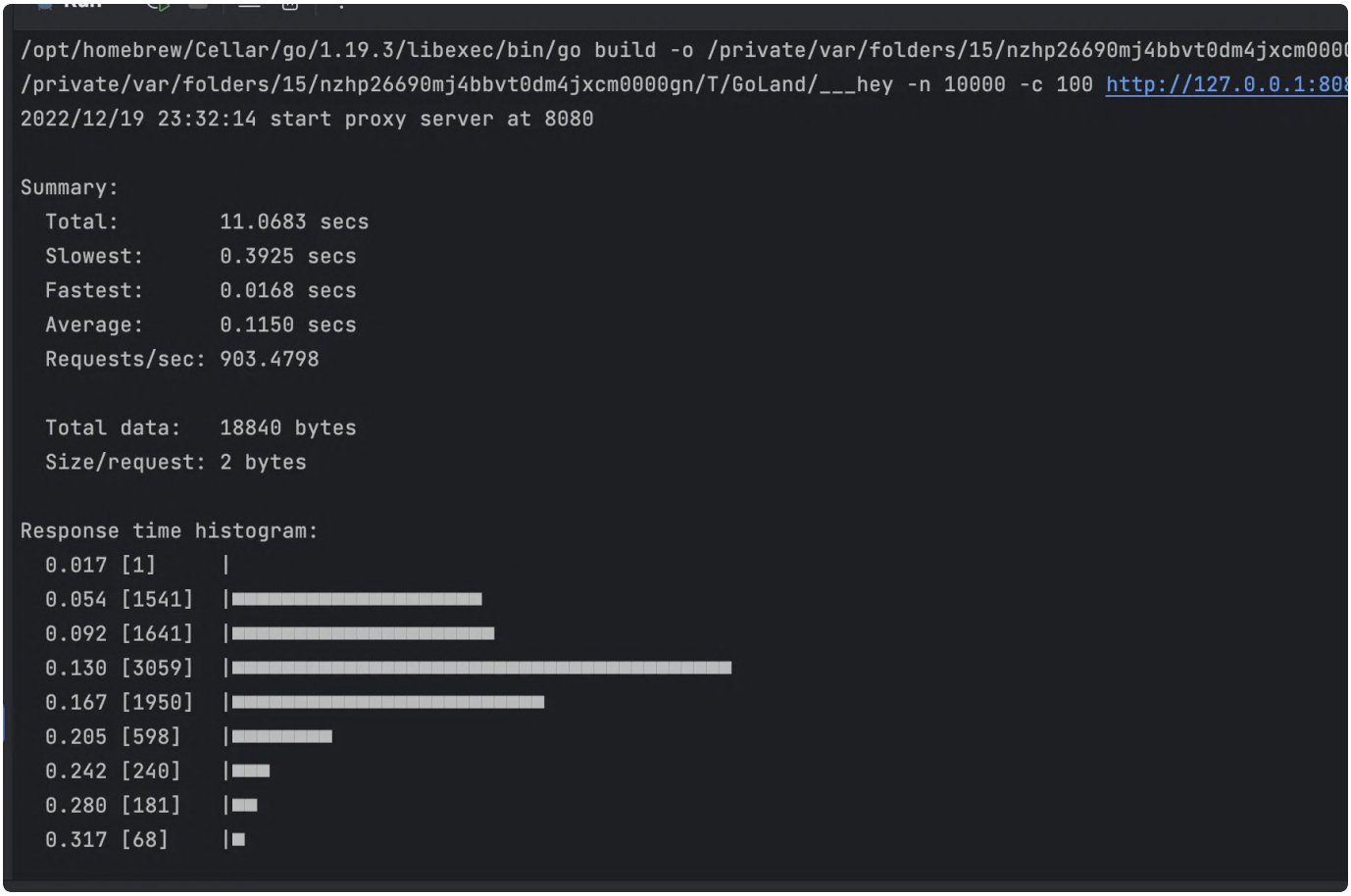
结论：请求QPS达到900，但由于是异步分批提交，10000条服务映射提交用时4分01秒。

优化：针对此种大批量上线情况，可以调整环境变量PILOT_SNP_DEBOUNCE_MAX为10s，副作用是，大批量上线时最晚会在10s后才能订阅到对应的服务映射。

"PILOT_SNP_DEBOUNCE_MAX"，此参数代表当持续收到注册请求时，合并请求的最大时长，默认为1s

"PILOT_SNP_DEBOUNCE_AFTER"，此参数代表多久没收到新的注册请求时立即提交，默认100ms

两者结合，即注册请求最短在100ms后提交，最长在1s后开始提交（异步做实际提交，并非提交完成），



最佳实践&宣传

一、最佳实践案例

对外暴露的 http 接口 http to dubbo

- http to dubbo 协议 https://github.com/apache/dubbo-go-pixiu/tree/develop/samples/dubbogo/simple/registry
- nacos注册中心 https://github.com/apache/dubbo-go-pixiu/tree/develop/samples/dubbogo/simple/registry 将 zk 修改成 nacos
- ratelimiter https://github.com/apache/dubbo-go-pixiu/tree/develop/samples/plugins/ratelimit
- 断路器 详见 https://github.com/apache/dubbo-go-pixiu/tree/develop/pkg/filter/sentinel
- tracer + jaeger https://github.com/apache/dubbo-go-pixiu/tree/develop/samples/dubbogo/simple/jaeger
- jwt 认证 https://github.com/apache/dubbo-go-pixiu/tree/develop/samples/dubbogo/simple/jwt

Cluster 健康检查

功能目的

目前系统中的定义:

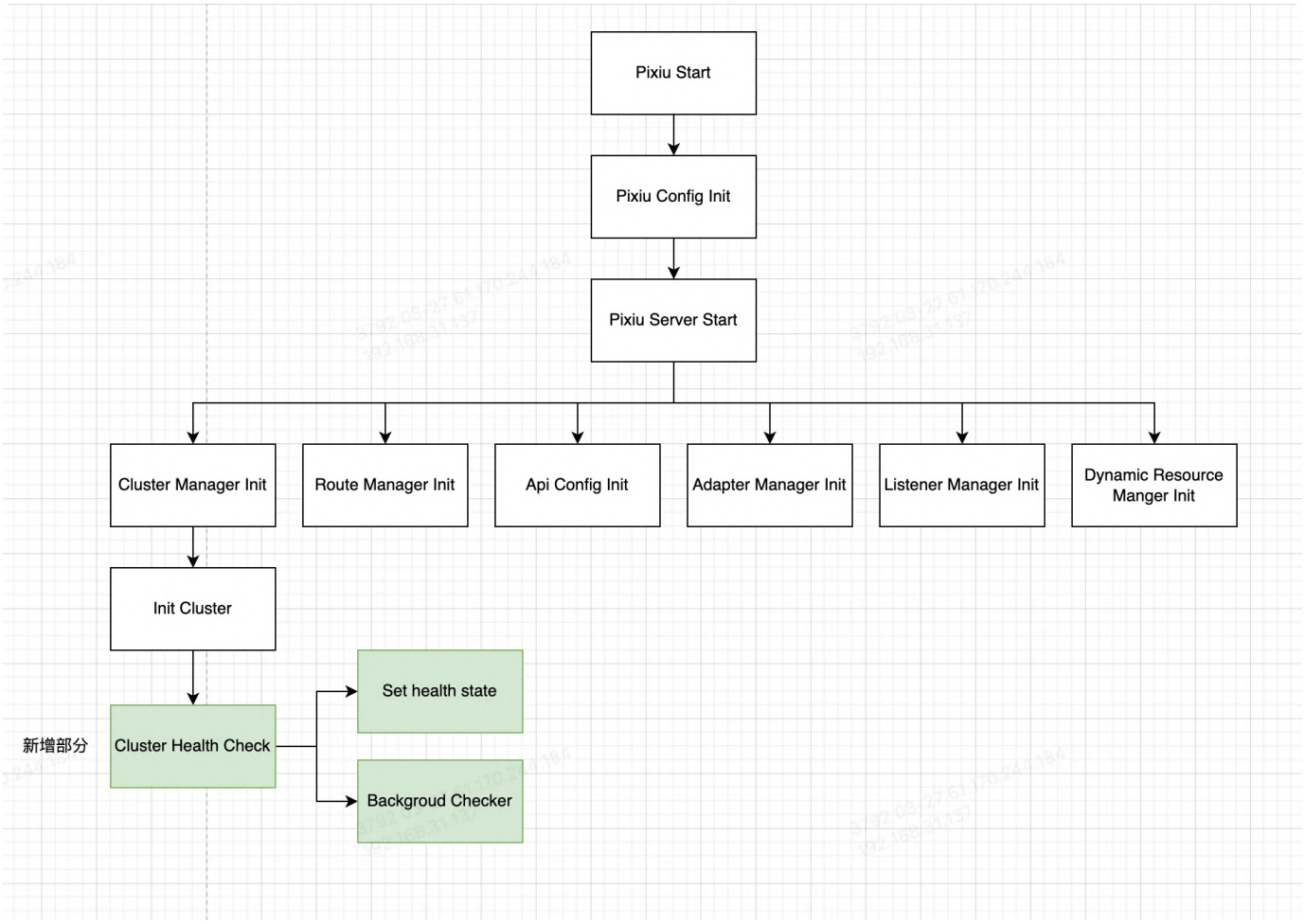
```go
type Cluster struct {
    Name                 string           `yaml:"name" json:"name"` // Name the cluster unique name
    TypeStr              string           `yaml:"type" json:"type"` // Type the cluster discovery type string value
    Type                 DiscoveryType    `yaml:"-" json:"-"`       // Type the cluster discovery type
    EdsClusterConfig     EdsClusterConfig `yaml:"eds_cluster_config" json:"eds_cluster_config" mapstructure:"eds_cluster_config"`
    LbStr                LbPolicyType     `yaml:"lb_policy" json:"lb_policy"` // Lb the cluster select node used loadBalance policy
    HealthChecks         []HealthCheck    `yaml:"health_checks" json:"health_checks"`
    Endpoints            []*Endpoint      `yaml:"endpoints" json:"endpoints"`
    PrePickEndpointIndex int
}
```

在我们 Pixiu 当前的 `statics_resource` 配置中, 其实是包含了对静态的 `cluster` 集群的健康检查配置. 但是未开启相关实现。

本期功能希望针对 `HTTP` 形式的集群节点增加健康检查功能. 对齐配置约定.

功能实现方案

目前系统已有的请求流程



Cluster Manager 改动

Add Cluster

新增一个新的 `cluster` 同时检测, 如果符合 `static_resouces` 的条件(静态配置启动cluster)，新开协程针对这个 `cluster` 内部的节点进行健康检查.

BackgroudChecker

新增方案, 进行异步的健康检查。 将健康检查失败的节点进行标记。

Pick Endpoint

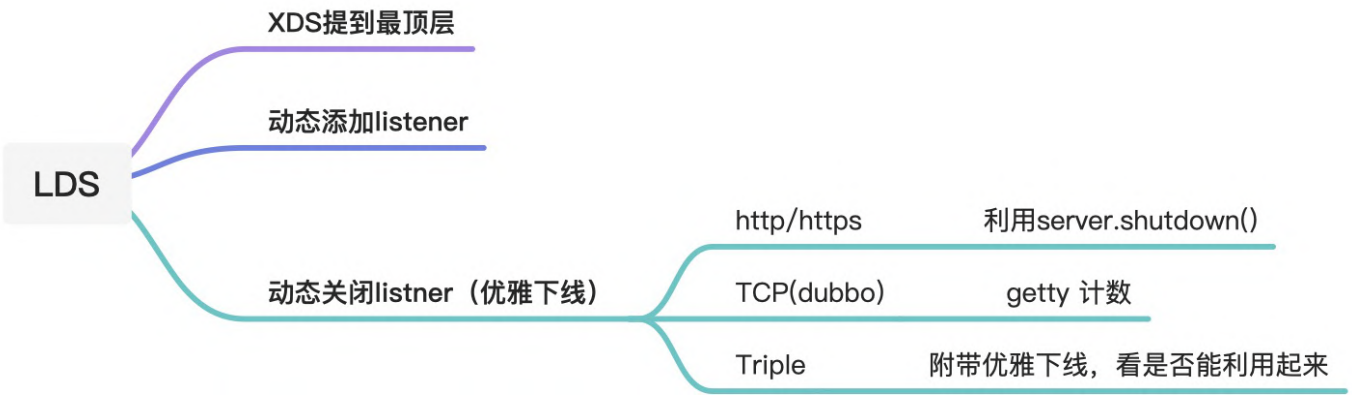
获取 `cluster` 内的节点, 检测是否属于健康状态.

Cluster 变动

将健康检查的能力实现在 Cluster 内部. 针对不同的健康检查类型维护不同的 Checker. 方便后续扩展其他类型的检查.

XDS-LDS

xds与pixiu的生命周期



Ready 检查点：

- 成功连接控制面
- Listener启动完成

结论：

分为两部分，第一步初始化所有的listener，第二部监听并动态刷新

代理注册

背景

pixiu目前希望能提供一套接入dubbo集群的多语言解决方案，简化整个接入的流程。例如让python，nodejs等各种语言的server 都可以依赖pixiu接入到dubbo/dubbogo 集群中，和集群中的dubbo server进行无感的相互调用。

方案

使用pixiu代理注册可以分为代理单个和多个服务(实例)

1. 代理多个服务的优点和缺点：
    a. 代理多个服务 简化部署以及运维成本， 一个pi xiu代理多个服务实例，pixiu的部署运维成本相对较小。
    b. 存在中心化问题 需要考虑单点可用性问题。代理的服务实例越多，pi xiu的压力会越大，除了

2. 负责流量转发还需要维护多个实例的心跳。
    a. 代理单个服务，一个pi xiu 对应单个服务实例，利用sidecar模式
    b. 部署和运维成本增加

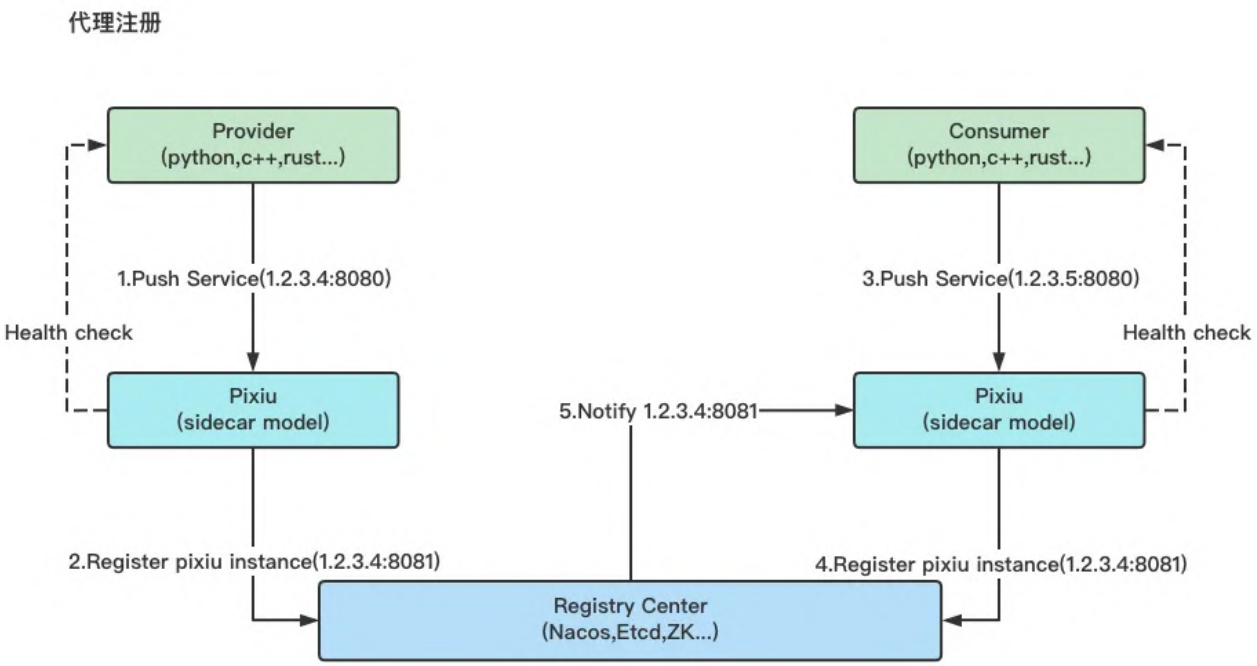
相比较之下感觉利用pixiu sidecar模式进行代理注册更适合，因为pixiu 目前代码中已有提供sidecar模式，只不过还没有具体的实现(主要功能都已经存在，可以复用gateway模式的代码)。调研mosn的流量劫持方案也是在sidecar模式基础上工作。

代理注册流程

1. 首先服务提供者(python/c++/rust...)推送代理注册请求(provider的ip和port 1.2.3.4:8080)到pixiu(sidecar模式)
2. pixiu将自身的ip和端口(1.2.3.4:8081)注册到注册中心(nacos,etcd,zk等)
3. 服务消费者(python/c++/rust...)推送代理注册请求(Consumer的ip和port1.2.3.5:8080)到

pixiu(sidecar模式)

4. pixiu将自身的ip和端口(1.2.3.5:8081)注册到注册中心(nacos,etcd,zk等)
5. pixiu目前已有的订阅功能会将注册中心的服务（provider的pixiu注册信息）通知到consumer pixiu,consumer pixiu 会在路由树中注册provider 的路由
6. 在pixiu代理注册成功后会主动对Provider/Consumer进行健康检查



代理注册设计

注册实例Struct

```go
type ServiceInstance struct {

    ServiceName     string
    Host            string
    Port            int
    Healthy         bool
    Metadata        map[string]string
    GroupName       string
    Namespace       string

}
```

1. 在sidecar 启动时开启http server/grpc server 提供代理注册服务，等待被代理方发起请求。在代理注册成功后，由pixiu发起心跳检测，心跳检测失败时，取消代理注册
2. 提供获取代理注册配置接口，主要由http,grpc等不同的方式去实现，通过接收到的请求转换为ServiceInstance
    a. 代理注册的信息来源可以是http/grpc/直接配置等

```Go
proxyRegisterConfig{

    GetProxyServiceInstance()instance ServiceInstance
}
```

3. 提供代理注册接口,nacos,etcd,zk等各自按需实现

```Go
ProxyRegister{

    Register(instance ServiceInstance)

    UnRegister(instance ServiceInstance)
}
```

实现思路

1. pixiu启动时并且是sidecar模式，则开启http 或者 grpc server接收对客户端的请求，待接收到请求后进行解析，获取到请求参数后将ServiceInstance 注册到注册中心。
2. 将代理的ip和端口存储在内存中。
3. 如果是服务级代理注册 相当于需要对到达pixiu sidecar的所有请求转发到代理实例上。将根路径/注册到pixiu router上即可。
4. 接口级别注册可能一个pi xiu 会代理同一个实例的多个接口进行注册，要求ip和端口必须一致，将接口注册到pixiu router上

websocket 支持

1. Pixiu websocket网关的基本能力：
    a. 连接功能
        i. 与终端建立连接，提供验证功能；验证扩展接口；
        ii. 维持与终端的长链接。
        iii. 实现基本通讯信令，特别是ping/pong，在web端没有提供ping 的 api。（https://developer.mozilla.org/en-US/docs/Web/API/WebSocket ；和WebSocket API Specification）;一般由服务端发送ping;
        iv. 提供编程接口：： onConnection; onClose; closeConnection；auth;
        v.

    b. session功能：
        i. 维护session与连接的映射；
        ii. 提供消息路由能力；
        iii. 具备session的编程接口onSessionCreate；onSessionClose； closeSession;
    c. 消息转发功能：
        i. 提供可以定制消息格式模版。
        ii. 接受消息，转发到应用（dubbo 服务）
        iii. 提供dubbo服务，对session发消息；
        iv. 消息暂存。对于消息没有对应连接；进行暂存；
    d. 统计功能：
        i. 对connection; session; message统计metrics接口。

Proxy Mesh方案建设

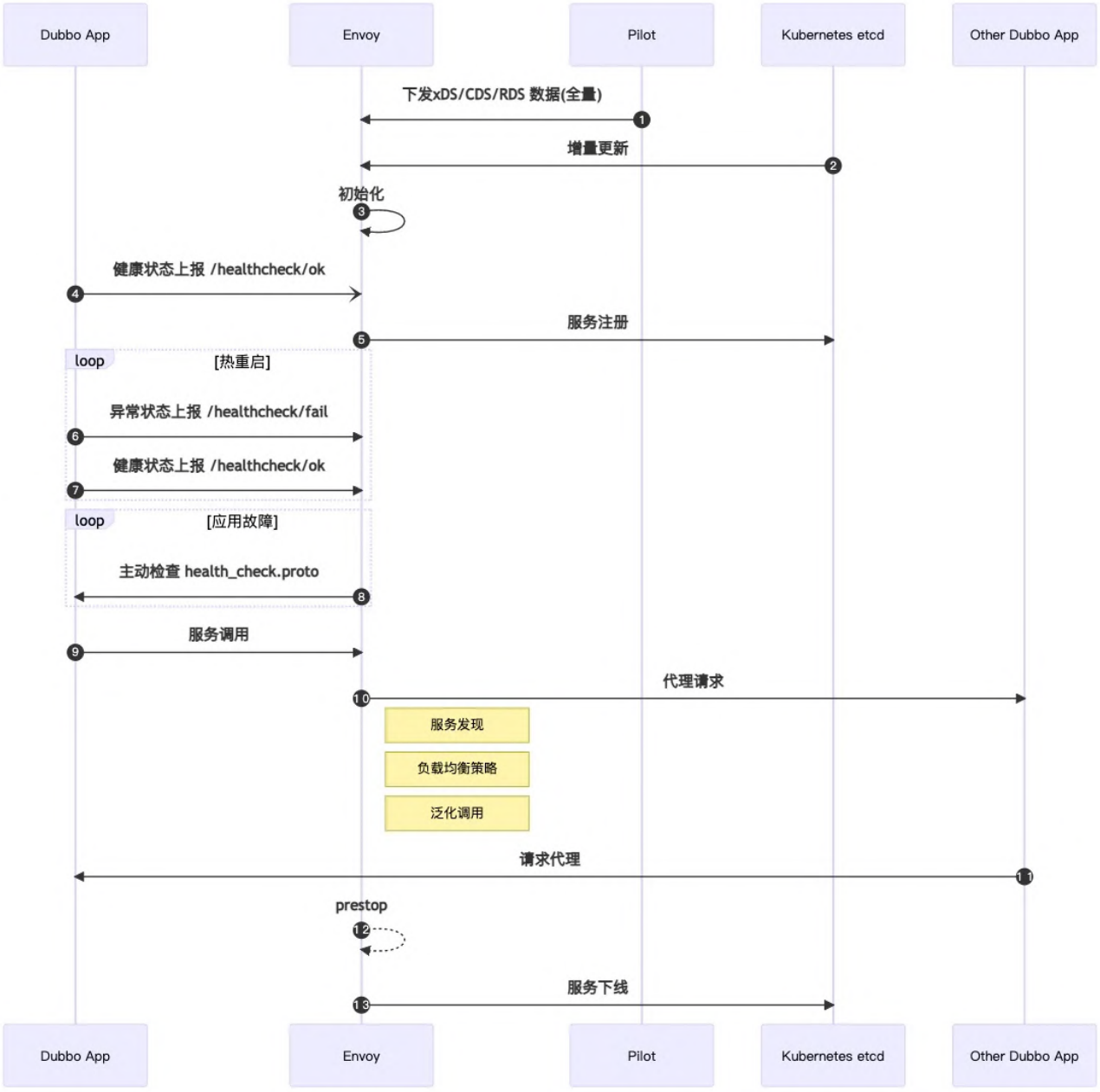
Dubbo 方案

主要诉求:

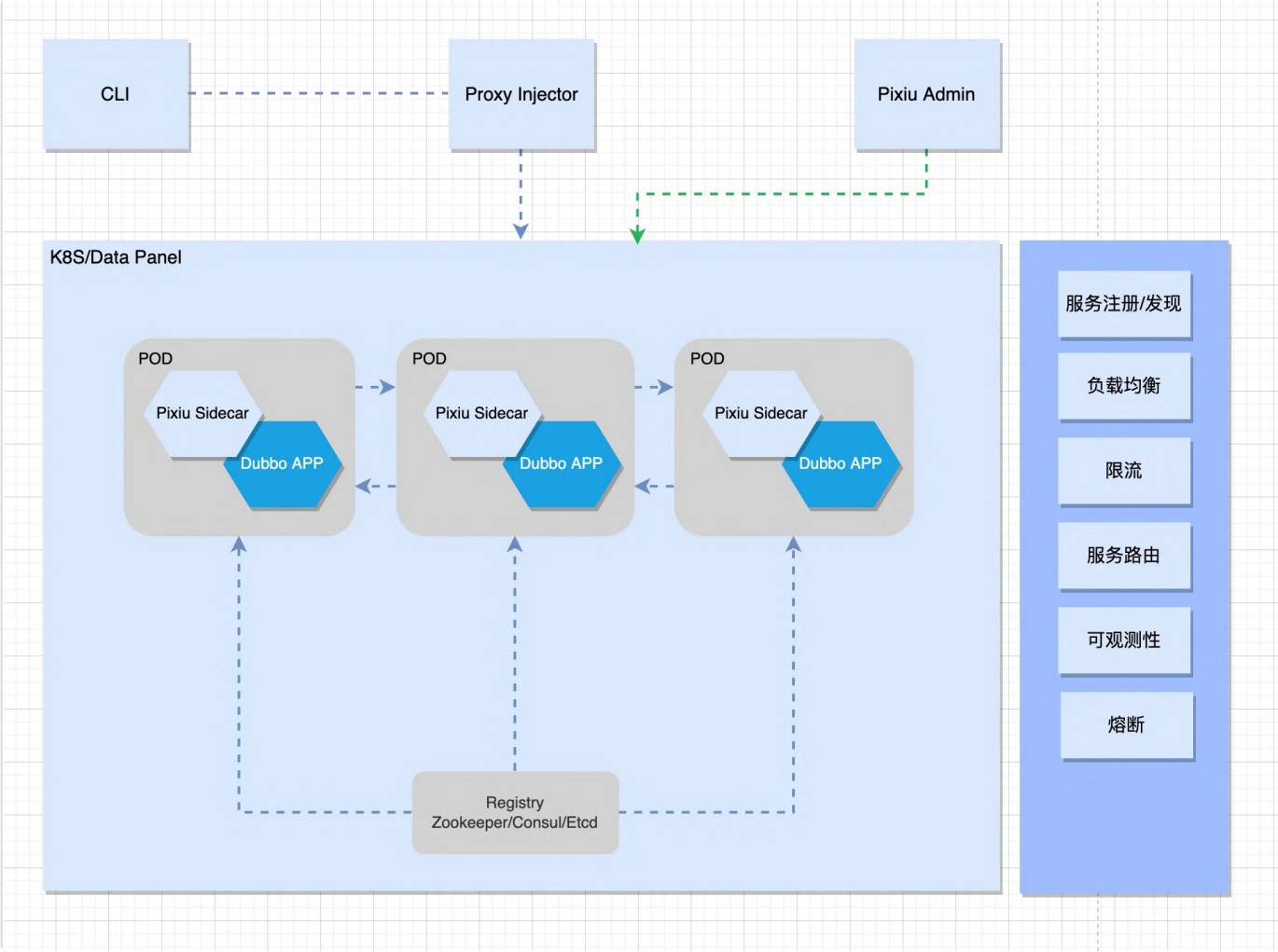
1. 应用启动生命周期对齐
2. 健康检查与服务状态控制
3. 服务代注册
4. 服务发现
5. 服务调用
6. Ingress
7. Engress

Ref: Dubbo Proxy mesh方案

Pixiu侧关注主要流程

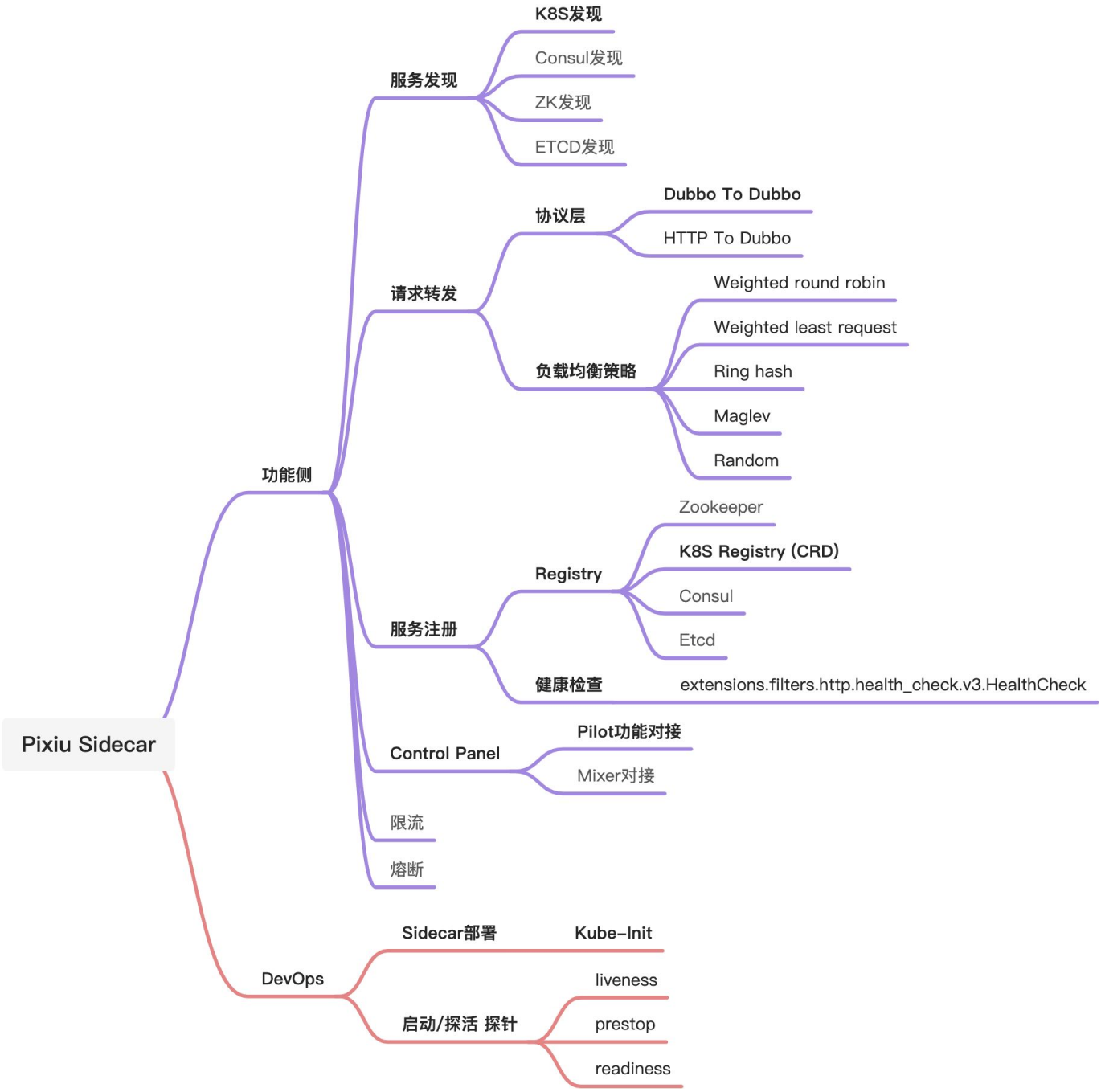

Dubbo App
Envoy
Pilot
Kubernetes etcd
Other Dubbo App
下发xDS/CDS/RDS 数据(全量)
1
增量更新
2
初始化
3
健康状态上报 /healthcheck/ok
4
服务注册
5
loop
[热重启]
异常状态上报 /healthcheck/fail
6
健康状态上报 /healthcheck/ok
7
loop
[应用故障]
主动检查 health_check.proto
8
服务调用
9
代理请求
10
服务发现
负载均衡策略
泛化调用
请求代理
11
prestop
12
服务下线
13
Dubbo App
Envoy
Pilot
Kubernetes etcd
Other Dubbo App

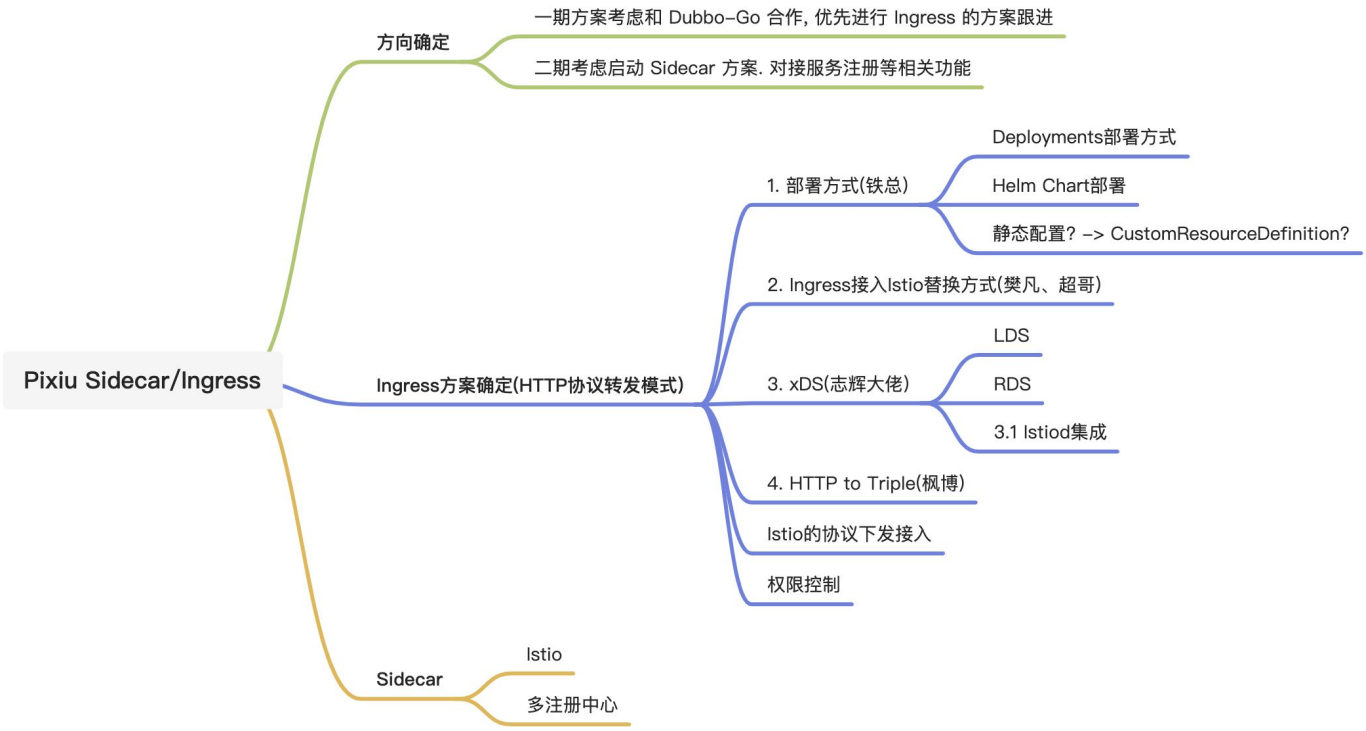
方案结构初览



功能点沟通

> 和 Dubbo 这边同学沟通下来, 一期我们主要关心以下功能和 Envoy 对齐满足方案上线需求.
>
> 一期方案采用半 Proxyless 方案, 客户端直连代理模式。


Pixiu Sidecar
功能侧
服务发现
K8S发现
Consul发现
ZK发现
ETCD发现
请求转发
协议层
Dubbo To Dubbo
HTTP To Dubbo
负载均衡策略
Weighted round robin
Weighted least request
Ring hash
Maglev
Random
服务注册
Registry
Zookeeper
K8S Registry (CRD)
Consul
Etcd
健康检查
extensions.filters.http.health_check.v3.HealthCheck
Control Panel
Pilot功能对接
Mixer对接
限流
熔断
DevOps
Sidecar部署
Kube-Init
启动/探活 探针
liveness
prestop
readiness



一期主要启动功能

启动流程

设计方案:

服务发现适配

设计方案:

请求转发

设计方案:

服务代注册

设计方案:https://www.yuque.com/xiguataizao/nqpt6u/rftqhh

Control Panel

设计方案:

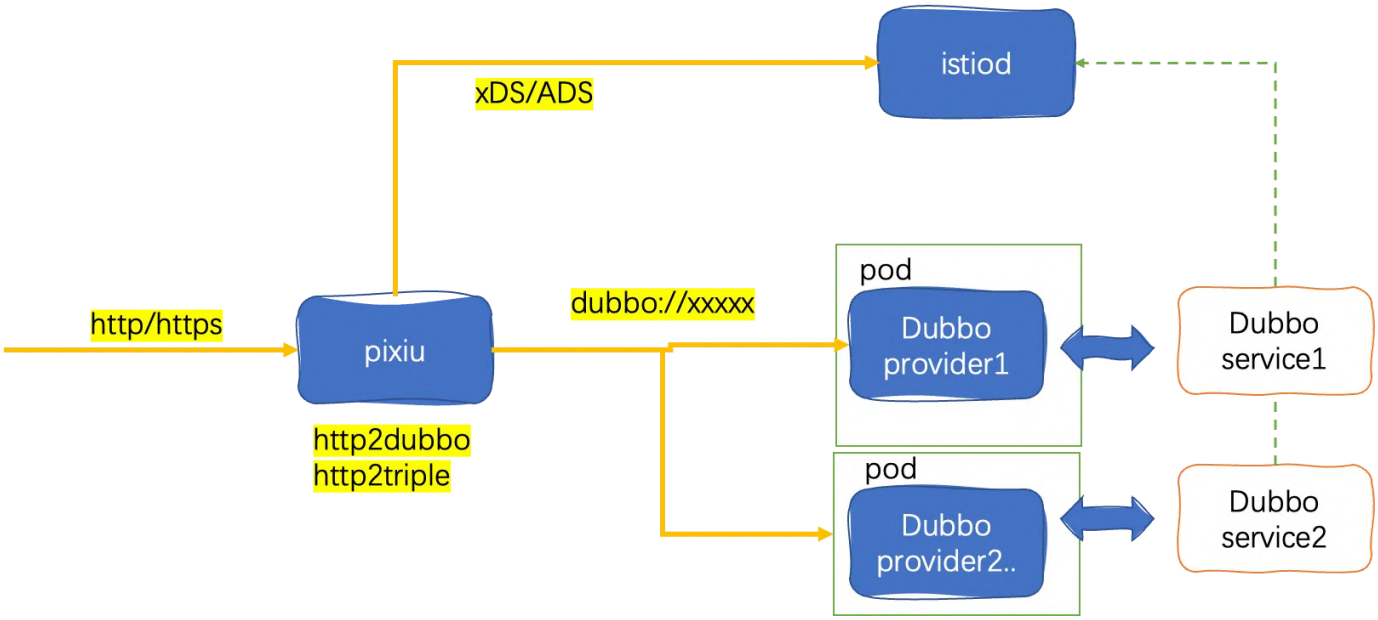
Cli工具部署

设计方案:

与istio一起工作

用例

当dubbo部署istio + k8s环境中，pixiu可以作为dubbo provider的ingress gateway，接收dubbo/triple/http的流量，并将请求转发到dubbo provider中。



在这个过程，需要pixiu，与istiod通信，基于ADS/CDS发现dubbo服务。

配置

一个典型的配置包括：

dubbo provider + service:

```yaml
dubbo:
  application:
    version: 1.0.0
  registries:
    xds:
      protocol: xds
      address: istiod.istio-system.svc.cluster.local:15010
  protocols:
    dubbo:
      name: dubbo
      port: 20000
  provider:
    services:
      GreetingService:
        protocol: dubbo
        version: 1.0.0
        group: default
        serialization: hessian2
        interface: com.dubbo.demo.GreetingService


```

```yaml
---
apiVersion: v1
kind: Service
metadata:
  annotations:
    meta.helm.sh/release-name: dubbo-go-app
    meta.helm.sh/release-namespace: default
  labels:
    app.kubernetes.io/instance: dubbo-go-app
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: dubbo-go-app
    app.kubernetes.io/version: 1.16.0
    helm.sh/chart: dubbo-go-app-0.0.1
  name: dubbo-go-app
  namespace: default
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: triple
    port: 20000
    protocol: TCP
    targetPort: triple
  selector:
    app.kubernetes.io/instance: dubbo-go-app
    app.kubernetes.io/name: dubbo-go-app
  sessionAffinity: None
  type: ClusterIP
```

---

gateway配置包括 使用dynamic_resources， 并在 dgp.filter.httpconnectionmanager 的route中使用cluster为dubbo-go-server。 dubbo-go-server并不是静态配置的cluster，而是有CDS生成的。

YAML

```yaml
node:
  id: "test-id"
  cluster: "pixiu"

dynamic_resources:
  cds_config:
    cluster_name: ["xds-server"]
    api_type: "ISTIO"
    refresh_delay: "5s"
    request_timeout: "10s"
    grpc_services:
      - timeout: "5s"
static_resources:
  clusters:
    - name: "xds-server"
      type: "Static"
      endpoints:
        - socket_address:
            address: "istiod.istio-system.svc.cluster.local"
            port: 15010
  listeners:
    - name: "net/http"
      protocol_type: "HTTP"
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8883
      filter_chains:
        filters:
          - name: dgp.filter.httpconnectionmanager
            config:
              route_config:
                routes:
                  - match:
                      prefix: "/UserService"
                    route:
                      cluster: "dubbo-go-app"
              http_filters:
                - name: dgp.filter.http.directdubboproxy
                  config:
  shutdown_config:
    timeout: "5s"
    step_timeout: "2s"
    reject_policy: "immediacy"
```

部署

完整的应用运行在istio + k8s环境中，包括

deployment: dubbo-go-app：dubbo-go provider

service :dubbo-go-app：dubbo-go-app对应的service

deployment: pixiu-gateway: pixiu 以gateway部署

service: pixiu-gateway: pixiu gateway对应service

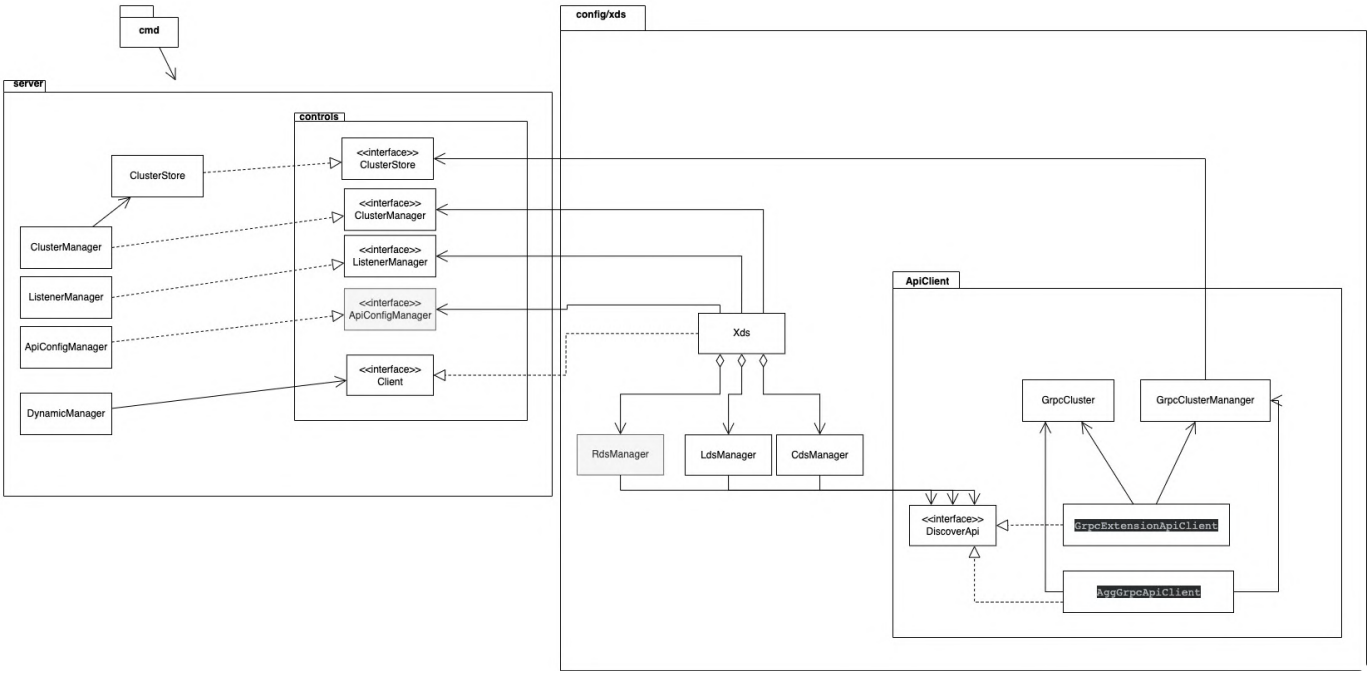
具体参考 https://github.com/apache/dubbo-go-pixiu-samples/tree/xds-istio/xds/dubbo-go-istio

编译pixiu docker image

pixiu-gateway 部署依赖pixiu的基础 image，可以自行编译。

```
dubbo-go-pixiu> make image
```

技术设计



AggGrpcApiClient：实现基于ADS xds client over gRPC. 实现与istio对接。

GrpcExtensionApiClient: 实现基于extension discover service的服务发现；实现与pixiu admin对接。

配置上的区别是 api_type = ISTIO / GRPC：

Plain Text

```
dynamic_resources:
  cds_config:
    cluster_name: ["xds-server"]
    api_type: "ISTIO"
    refresh_delay: "5s"
    request_timeout: "10s"
    grpc_services:
      - timeout: "5s"
```

部署调研和方案思考

安装

一、通过 Helm 安装

- 一个 helm 的仓库 https://github.com/dubbo-go-pixiu/pixiu-helm-chart
- helm 入门必看 https://helm.sh/zh/docs/chart_template_guide/getting_started/

参考 https://github.com/apache/apisix-helm-chart

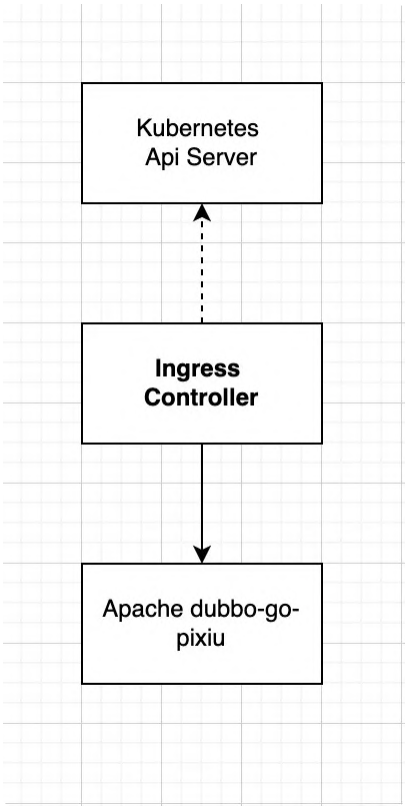
二、通过 yaml 文件安装

- 准备一个仓库 pixiu-on-kubernetes
  - `kubectl apply -f pixiu.yaml`

CRD

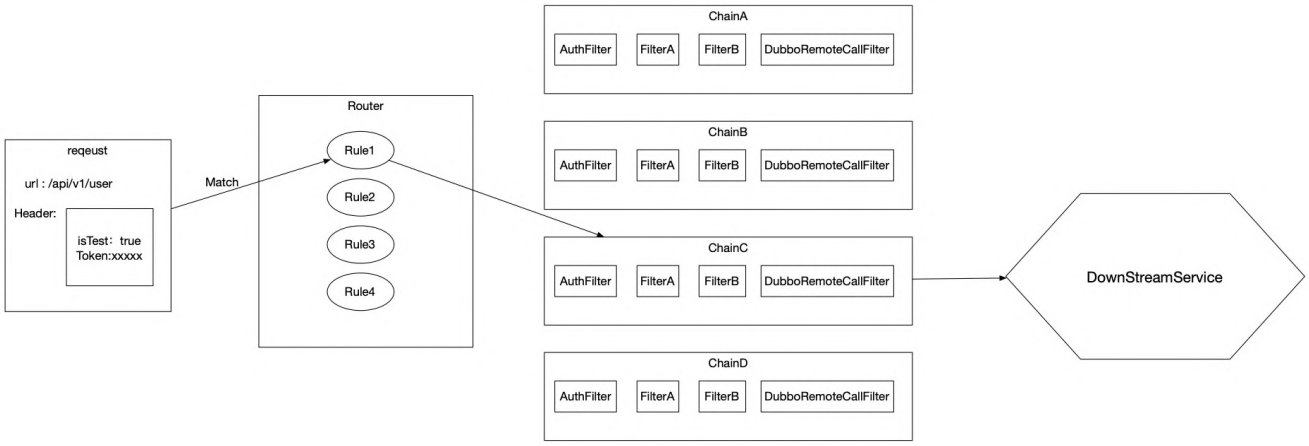
ingress-controller

图来自 apisix


Kubernetes
Api Server
Ingress
Controller
Apache dubbo-go-
pixiu

非标准 Rete

需求与背景



网关的核心之一是他的路由逻辑，决定一个请求需要经过怎样的加工，被转发到哪个下行服务。
目前基于url匹配可以完成80%的路由需求表达

例如
/test/** 开头的url 路由到测试环境集群

但不可避免的，有一些需求，需要根据header信息或者body信息来做一些路由判断。
以我们公司的实际业务场景为例
header中 env这个key 对应的value 包含 sandBox 字符串的情况下，需要路由转发到沙箱环境。
类似的需求都要求gateway 具有更强的规则表达能力。

同时网关作为所有请求的入口，每一毫秒的延时都会做用在全量的业务下，在mesh 场景下，延时还会随着调用链路的加深，被倍数放大。按照生产环境业务相应<=7毫秒的标准来看，规则匹配的性能要求也是十分苛刻的。一定不能随着规则数目的增加而性能退化。
引入rete的beta 网络来表达复杂规则，似乎成了唯一解?

rete核心思想

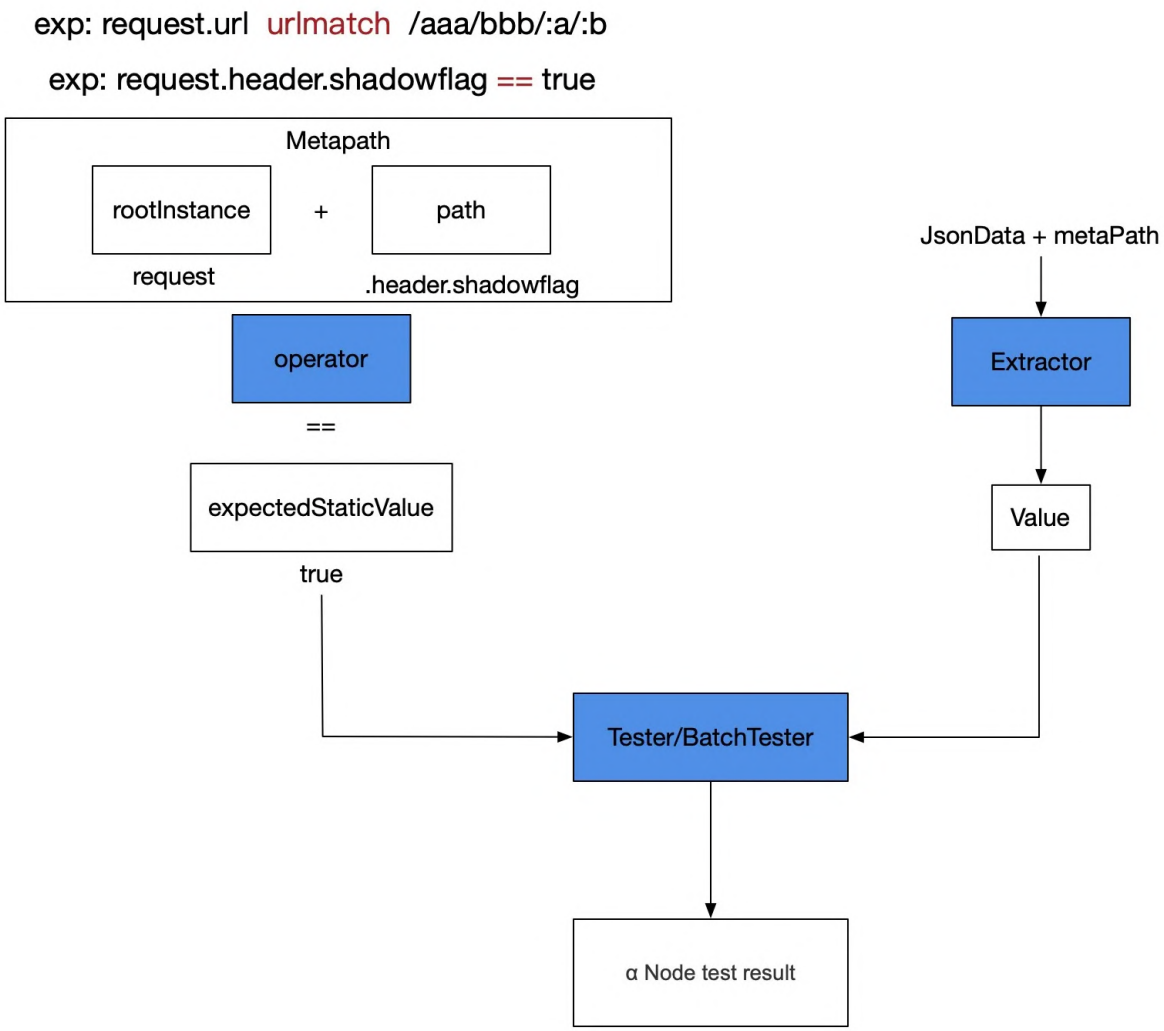
1.把所有的复杂条件拆分成最简单的子条件
2.使用根据最子条件构建alpha 网络，完成子条件的批量测试
3.根据alpha 网络的批量测试结果（alpha网络产出 所有match 的子条件集合）输入到beta网络做复杂条件匹配。
4.复杂条件匹配结果对应的rule 集合 就是所有匹配的规则

标准rete大量使用了实例数据匹配结果的缓存，取巧的通用的解决了批量规则匹配问题，比较重量级，适合动态匹配场景。我们没有动态匹配需求，非标准rete 不使用缓存，alpha网络根据oprator 做每个场景的细分优化，beta网络使用动态规划避免全量计算更轻量。理论上每个operator的batch test 逻辑都优化到O1 的情况下性能不输标准rete命中缓存的情况。

编码前的思考

1. request.header.shadowflag == request.header.flag 类似这样的配置需求不考虑，这种动态的规则匹配需要循环迭代，很影响性能，不考虑支持。
2. 由1 推出，表达式可以简化为 变量 + 操作符 + 静态值
3. 第一版本仅支持变量位于操作符左边的语法 a == 1 支持， 1==a 不支持。
4. 操作符带有方向性，少数操作符无方向性。 a==1 等价 1==a ，这种为无方向性， a > 1 不等价 1 > a 这种为有方向性。
    a. 有方向性的操作符需要有对应反向的操作符来表达对应的变量在右侧的需求，exp: 1 > a 需要由 a < 1 表达。
    b. request.header.str1 contains "abca" 属于有方向性的操作符，但暂时不考虑反向操作符。业务上没意义。
5. 考虑到需要支持结构化的树形的组织形式，变量需要描述清楚两部分内容，通过metaPath这个结构来描述
    a. 变量从哪个结构下获取 rootInstance
    b. 变量根据怎样的逻辑对应结构下获取 path
6. 暂时所有模型都不考虑持久化 上rds再考虑。

α node 核心模型如下：



目前的trie 实现的url 匹配 ，作为 urlMatch 操作符的 batch Tester 实现。
每种操作符，必定存在逐个匹配的 match 方案，但是性能不一定优秀。经过优化的batchTester 不一定容易实现，所以在没有对应的batchTester 优化方案的前提下，退化为 逐个调用singleTester，这时候匹配的效率会随着匹配规则的增长线性退化。复杂度O（n）。

以urlmatch 为例，目前已经优化到了O（1）所以按照标准的rete 实现对匹配结果做缓存没有必要，读取缓存的复杂度也可以看作是O（1），两者没有差距。

规则的拆分

单条复杂规则可以是多个小规则的 且逻辑拼接，或逻辑拼接。
拆分到最小元素不可被拆分的小规则对应一个alpha Node。
一个完成的规则输入以后，按照&&和｜｜对规则进行拆分，拆分后的子rule作为规则判断的最小单元。

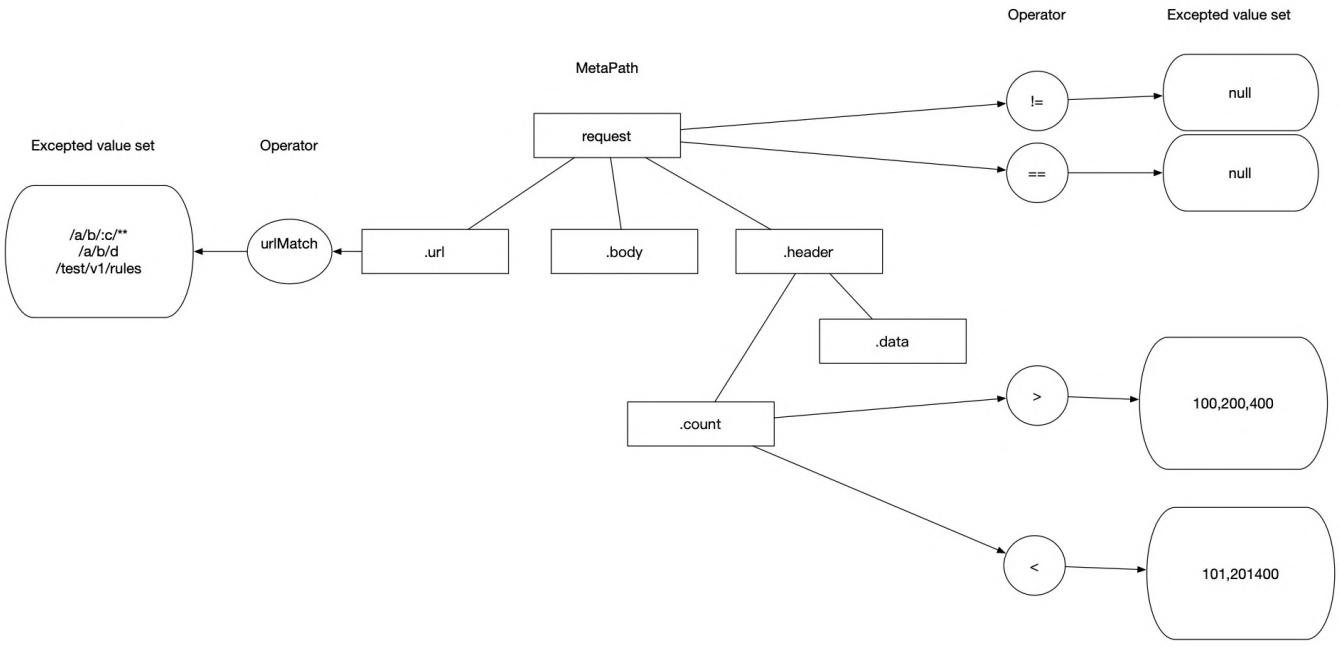
这种允许多个规则组合的需求导致单纯的树状推理结构不能满足要求，以入度大于1 的节点为区分，称之为β node。

Beta node 的描述方式决定了组网的方式，组网逻辑可以自由发挥，这次暂时只支持 && | | 两种beta node

所有最子节点不可被拆分的判断条件称之为alphaNode，都可以做批量优化，比如 == 操作符， 利用互斥关系容易得到 一旦 a==1 成立 a==2 ，a==3 等其他value 一定不成立，所有描述a 且操作符号为 ==的规则存放在一起，构建 hash table ，就能把一批条件同时做出判断。这种优化逻辑称的实现之为 batchTester。batchTester的优化思路需要按照oprator 分类讨论。

alphaNode分区

规则天然以 metapath 为分区，同一metapath 的情况下，还可继续以oprator 为分区，不同分区下的alpha node成立与否互不影响，可以并行计算。数据量足够大有必要的情况下可以做shard，目前无必要，单节点下内存存储足够了。但是并行计算可以用上。



上图是alpha node 分区后的存储结果，每个value set 里的一个value 代表了一个拆分到不可拆分状态的一条rule

上图等价于如下 rule 集合

request!= null

request==null

request.header.count>100

request.header.count>200

request.header.count>400
request.header.count<101
request.header.count<201
request.header.count<401
request.url urlMatch /a/b/:c/**
request.url urlMatch /a/b/d
request.url urlMatch /test/v1/rules

解释一下为什么可以并行计算每个value set
如下case

JSON

```
request{
  body{
    "a":"a"
  },
  header{
    isTest:"true",
    token:"asidufjasdhjfklsahdfjklasdhfldsakjhf"
  }
}
```

下面的解释是不是有必要？。。好像有点废话
这样一个请求作为待匹配的实例数据，找到他吻合哪些rule，很容易得出结论 描述header的信息一定不会影响body信息的匹配结果。如果header结构比较复杂那么需要用metapath 来描述具体是哪个属性需要满足哪个规则，metapath之间也一定不会相互影响。所以按照metapah 分区，可并发一定成立。

再来看一下不同的operator
看毛啊明显也不相互影响。
废话结束

所以匹配过程可以认为是把输入数据的每个字段的值，逐个与静态期望值的一个对比过程。

alpha 网络 匹配逻辑伪代码如下：
for each 每个对象的属性 属性的每个属性 in 对象
    for each alphaNode in excepted value set
        判断 属性 是否 match alphaNode
        if（match）

作为alpha 网络结果输出

第一层for循环可以并发处理，第二层for循环通常可以优化到O1，是在没办法的时候逐个match。

或逻辑分析 （ || ）

或逻辑在一边条件为true 的情况下，可以做出短路判断。
假设存在如下 规则描述，满足这个规则的前提下需要执行 filter1
request.header.count<401 || request.url urlMatch /a/b/:c/** ---> rule 1 then do filter1

那么根据短路原理可以等价描述成如下两个 子rule 到fillter 的映射：
request.header.count<401 ---> rule 1 then do filter1
request.url urlMatch /a/b/:c/** ----> rule 1 then do filter1

看一个略复杂的例子
a&&(b||c)&&d&& (e||f) then do filter1
其实可以拆分成等价的
a&&b&&d&&e then do filter1
a&&c&&d&&e then do filter1
a&&b&&d&&f then do filter1
a&&c&&d&&f then do filter1

可以得到推论，复杂的或逻辑可以被拆分成多条等价只包含且逻辑的子rule。

且逻辑分析 （ && ）

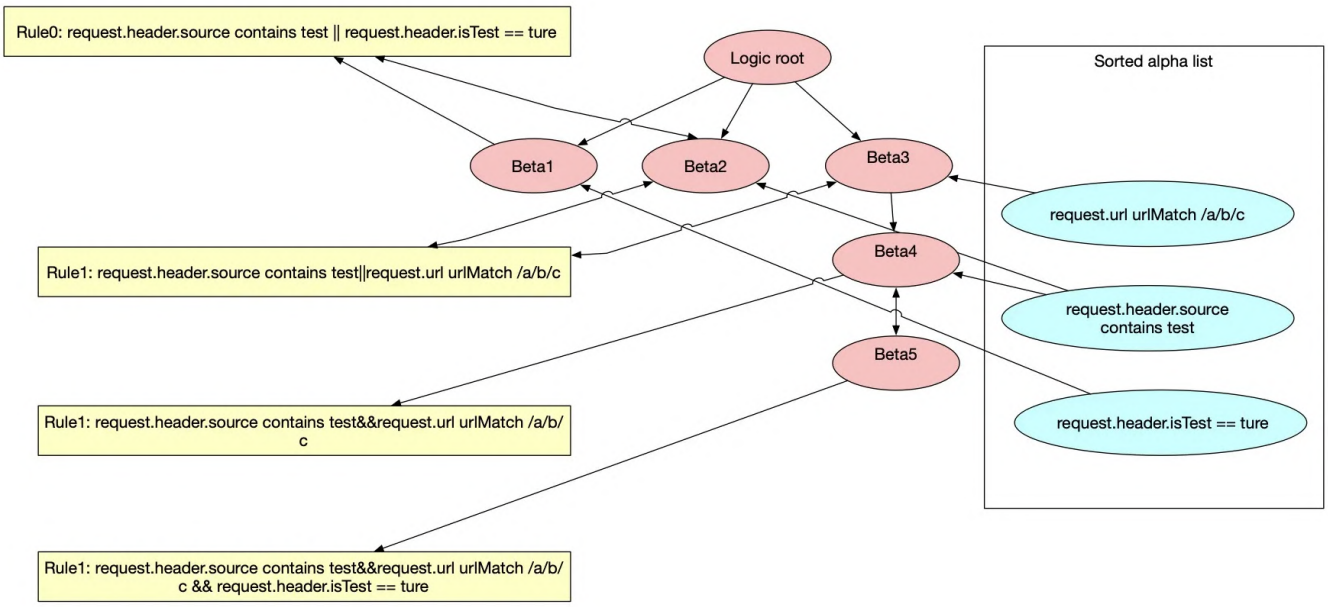
且逻辑在一边条件为false 的情况下，可以做出短路判断。
a&&b&&c&&d&&e 在a 为false 的情况下就没有必要继续判断后面的bcde条件。
是不是又联想到了字典树
以单个条件作为字典树的node，字典树的路径代表且逻辑，字典树上的路径等价于一个仅包含且逻辑的复杂rule。

值得思考的点在于 a&&b&&c&&d&&e 与 e&&d&&c&&b&&a 是完全等价的，但是放入字典树的顺序会影响短路的先后，越后面的条件被计算的概率越低，相应的一些匹配复杂度较高的rule 可以设置较低的组网优先级，使得该节点再beta网络里靠后，整个排序算法可以自定。目前一个rule 必须关联一个

batch tester ，batch tester 上维护一个排序优先级，url match 之类的 产出唯一的值，优先级最高，优先级相同情况下字母序排序。

beta 组网

如上一段落说明， 按照一定规则排序后 假定 三个规则排序顺序如下（蓝色node）

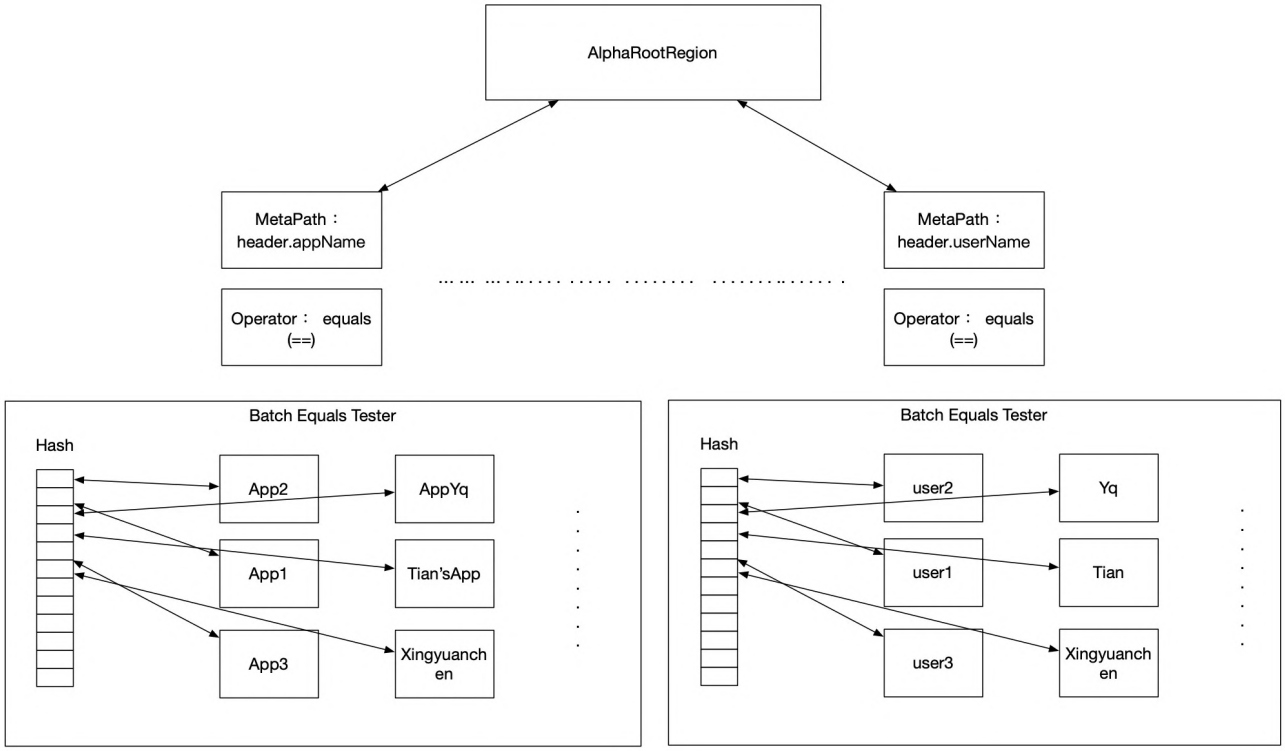


构造字典树后形成红色node 的beta 网络，可以由多个betaNode 指向同一个rule，来表达或逻辑。

Alpha 组网

按照 operator 和 metapath 分区后 ，区内按照batch tester 优化逻辑不同，对应多种组网方式。

1. equals 等 基于hash table 优化的操作符　平铺组“网”



在batchTester.test("yq") 调用时，本质上去hashTable 看有没有value 为yq 的元素，match到的情况下把对应的alpha node 作为输出，然后执行beta 网络匹配

2. urlMatch 等 基于非标准trie 的组“网”

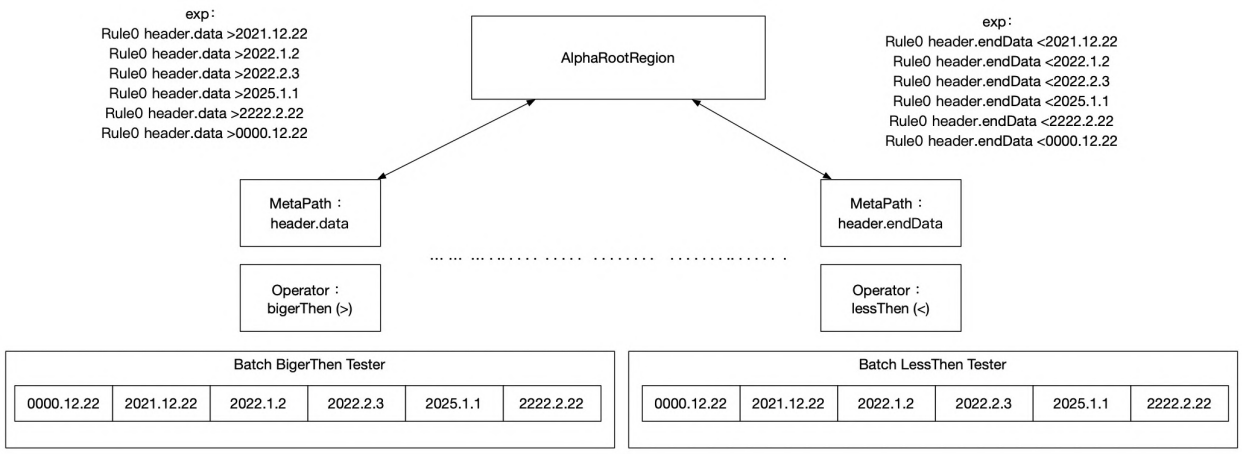
详见 https://dubbogoproxy.yuque.com/dubbogoproxy/vld1hq/cyxixw

3. 基于标准trie的 contains 操作符 组网

详细见 https://dubbogoproxy.yuque.com/dubbogoproxy/vld1hq/apfc9k

4. > ， < 操作符 组网

d. 基于有序队列的 > < 组网（比对等读操作OlogN ， 写入等维护操作OlogN））



e. 基于线段树的 > < 组网 （读O1， 写入触发resize情况下Onlogn ）

详见

5. in 同 equals
6. ！= ， not in 同 equals
7. 暂时不考虑其他操作符了

基于离散线段树的BatchNumberRangeIn优化

数字范围类型的规则描述通常也是常见的一种匹配操作符。

MatchContext.startTime > toLong(前一天 0点0分) and MatchContext.endTime < toLong(今天 0点0分)

如上，一组描述生效时间范围的规则通常会被这样表达。本质上代表了一组区间。

区间表达 按照数学描述 分为如下几类

1. 前开后开 (A,B)
2. 前开后闭 (A,B]
3. 前闭后开 [A,B)
4. 前闭后闭 [A,B]

直接用于Beta 网络面临的问题

目前beta网络设计初衷在于优化规则匹配的性能到与存量规则数量无关。但是一些细节上并不能完全做到这个设计初衷。根本原因在于 如果 阿尔法node 产出的结果集非常的大，那么是要逐一添加命中节点到bfsQueue下的。本质上是一次匹配到条件的全量遍历。所以在设置规则

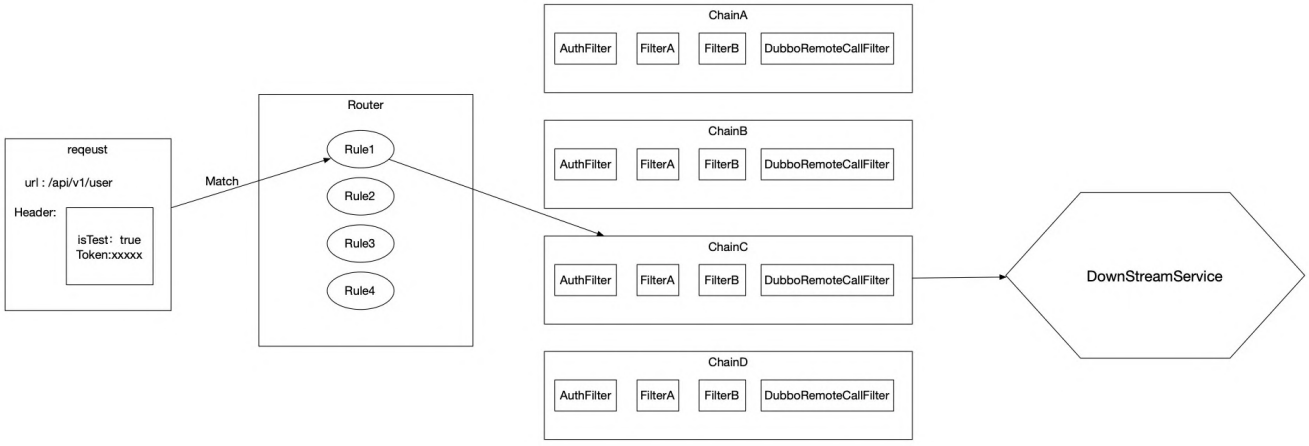
那么这类区间问题会被表达为 两个条件的and 拼接。假设区间数为N。节点数为2N，单次节点做判断的时候，总会有一半的节点在

用

容易想到 先进行一次排序后根据

Pixiu路由原理简介

简介



网关的核心之一是路由逻辑，决定一个请求需要经过怎样的加工，被转发到哪个下行服务。
其中 80% 的路由需求表达都以 URL 为基础。需要描述清楚具有某个特征的 URL 或者 URL 集合对应怎样的一系列下游处理策略。

例如，'/test/**' 开头的 URL 路由到测试环境集群，'/release/user/**' 开头的 URL 会被路由到正式环境的 user 服务集群。

同时网关作为所有请求的入口，每一毫秒的延时都会做用在全量的业务下，在 mesh 场景下，延时还会随着调用链路的加深，被倍数放大。按照生产环境业务相应 <=7 毫秒的标准来看，规则匹配的性能要求也是十分苛刻的。一定不能随着规则数目的增加而性能退化。

使用介绍

仅从使用方的角度阐述 pixiu 的配置文件如何描述 URL 相关的路由规则。（下面，我们介绍一下如何配置 URL 路由规则）

如下是一份 pixiu 的 api 配置文件，这份配置文件会被解析后生成一份对应的内存模型，作为 pixiu 路由相关配置的初始状态。之后由 RDS 协议修改解析后得到的内存模型，实现路由逻辑动态生效的效果。RDS 协议（RDS：xDS 协议下描述路由规则的部分）相关内容是后话不详细阐述。我们把注意力聚焦到resource部分。

resource 下 path 部分就是上文阐述的，URL 相关的路由描述。意思是满足 path 描述特征的 URL 会被成功匹配。

JSON

```yaml
name: server
description: server sample
resources:
  - path: '/api/v1/test-dubbo/user/name/:name'
    type: restful
    description: user
    methods:
      - httpVerb: GET
        enable: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
          uri:
            - name: name
              required: true
        integrationRequest:
          requestType: dubbo
          mappingParams:
            - name: uri.name
              mapTo: 0
              mapType: "string"
          applicationName: "UserProvider"
          interface: "com.dubbogo.server.UserService"
          method: "GetUserByName"
          group: "test"
          version: 1.0.0
          clusterName: "test_dubbo"


```

被匹配后的请求会被转化成 dubbo 协议转发到 test_dubbo 集群调用 com.dubbogo.server.UserService 下的 GetUserByName 服务。
我们继续聚焦到如下范围：

```JSON
path: '/api/v1/test-dubbo/user/name/:name'
```

为了描述清楚一个 URL 或者一组 URL，路由引擎需要拥有以下能力：

1. URL 可以包含变量，'/api/v1/test-dubbo/user/name/:name' 代表 URL 用“/”分割后，第六个部分的值作为变量 name 的值，向下游 filter 传递供filter使用。
2. 需要有通配符
   a. * 代表一个层级任意字符的通配 '/api/*/test-dubbo/user/name/:name' 这样的一个 path 描述代表了可能并不关心具体的版本，不论什么版本下的 URL 只要匹配都使用相同的逻辑加工数据并转发。
   b. ** 代表多个层级的通配，从这个层级以后，子层级也可以是任意字符，**只可能存在于 URL 的末尾，不然会有二义性。'/api/v1/**' 这样的 path 表达了所有 V1 版本下的 URL 都采用相同的逻辑。

为了正确的使用 pixiu 您可能还需要了解如下内容。

优先级

并非是独创的，类似 java 下的 spring 以及其他框架统一具有的优先级逻辑：

1. 通配的优先级低于特指 。 '/api/v1/**' 低于 '/api/v1/test-dubbo/user/name/:name' 的优先级，假设有两个 resource 分别采用了如上两个path 配置，request 为 '/api/v1/test-dubbo/user/name/yqxu' 的请求到达pixiu 后应该生效哪个 resource？按照通配低于特指的原则，'/api/v1/test-dubbo/user/name/:name' 这条规则会生效。
2. 深度更深的，优先级更高 。 '/api/v1/**' 对比 /api/v1/test-dubbo/**' ，如果请求同时满足如上两个描述， '/api/v1/test-dubbo/**' 深度更深，会生效。
3. 通配符之间 '/*' 优先级高于 '/**'
4. 变量等同于通配。

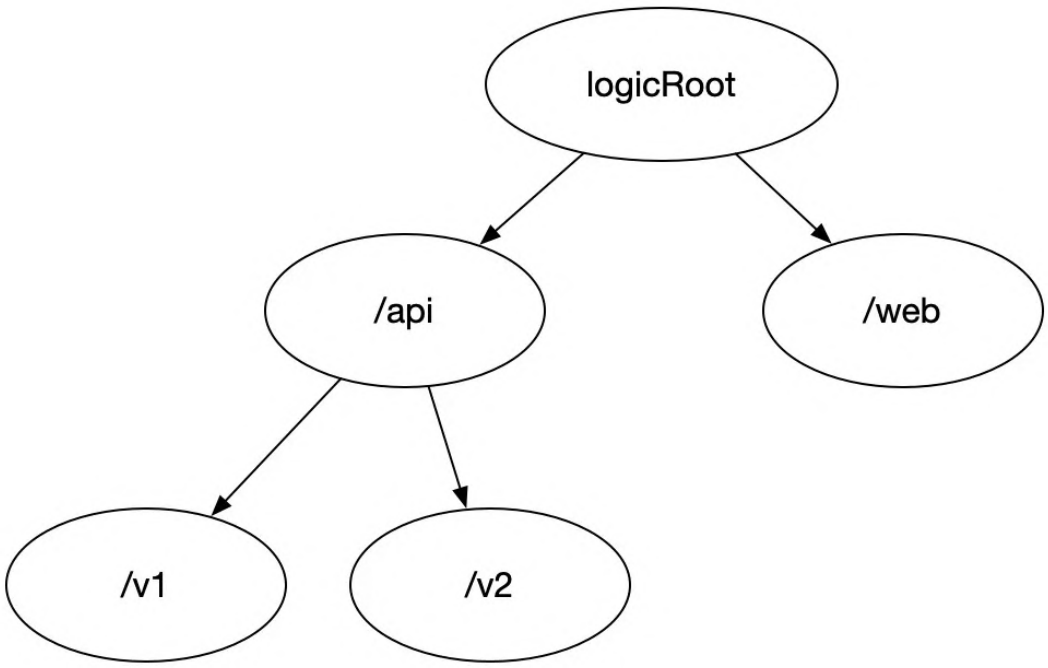
冲突处理

优先级规则只是冲突解决策略的一种，才同时匹配多个url描述时，优先级更高的那一种将会生效，然而优先级策略并不能涵盖所有的情况。
如果强行配置两条 resource path 完全相同，但是转发到不同的下游服务，这时候就会冲突。pixiu 下应对冲突的方案是 failfast，在 pixiu 初始化阶段，发现配置文件中有冲突的两项规则，则启动失败，让开发者今早发现问题并处理。

原理介绍

技术选型之初，以及确定使用pixiu后为了处理一些突发情况，以及应付一些pixiu自身可能存在的bug，开发者需要对pixiu 的路由原理有更深刻的了解。
下面，我们将详细介绍路由引擎的相关原理和实现，供感兴趣的同学了解。
相信阅读这部分内容的同学一定会有人下意识联想到字典树这个结构。使用字典树这个结构能实现存量规则数无关的匹配性能优化。

一个存放字符串作为node的字典树，具有表达url 的能力。



如上图描述等价于URL集合 '/api/v1' ，'/api/v2' ，'/web'

维护一个标准字典树有几个关键的操作

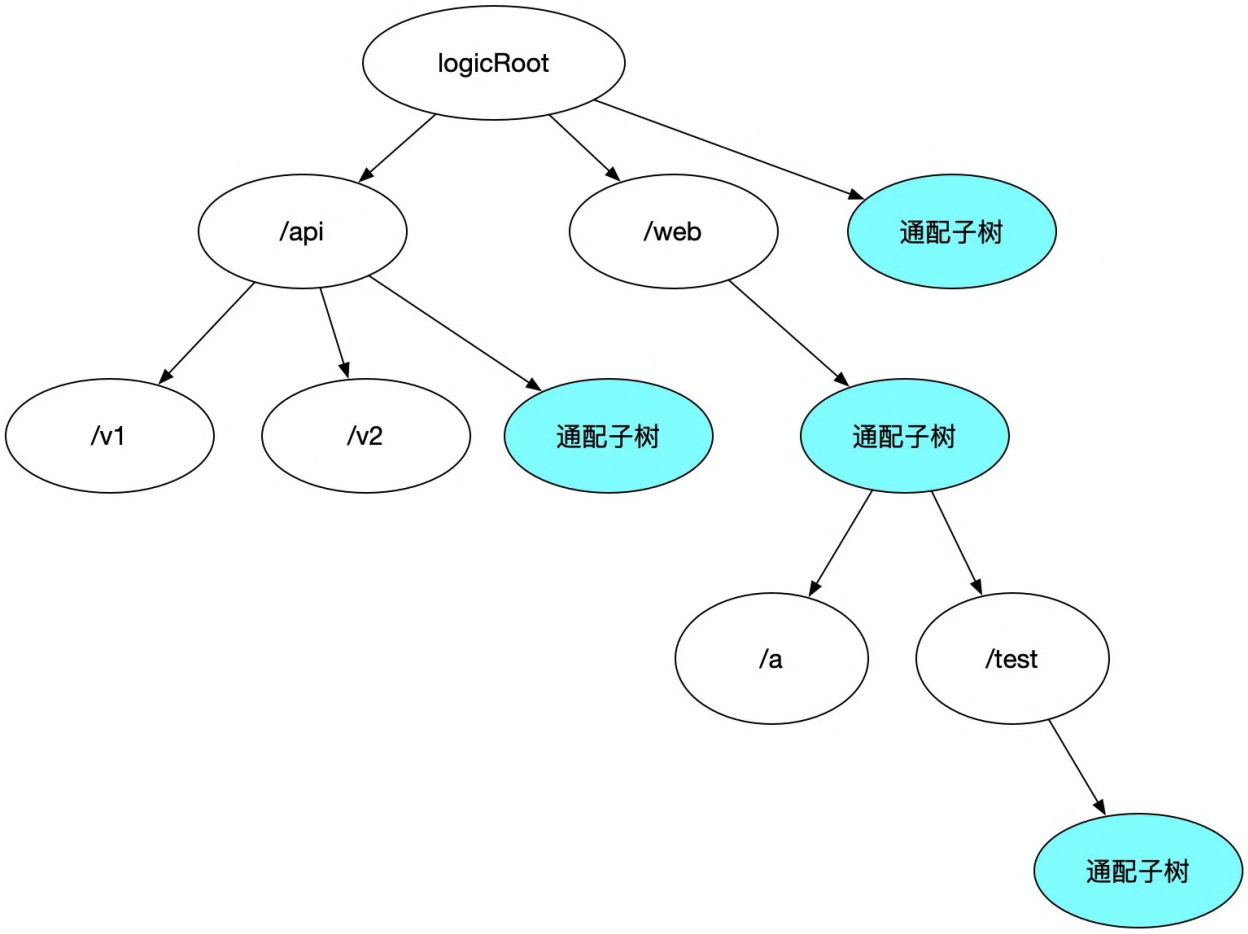
1. 字典树指定节点的查找（find）：从root 开始遍历字典书，'/api/v2' 称之为路径，在当前层级寻找指定路径，如果存在就继续在子树下完成剩下的路径匹配。'/api/v2' 先从 logic root 找到 '/api' ，并在 '/api' 的子树下继续查找剩下的路径 '/v2' 。
2. 字典树节点的添加（add）： 尝试查找指定节点，如果指定节点不存在则新建一个节点。假设一个空树状态下添加 '/api/v1' ，因为是空树那么logic root 下查找 '/api' 一定不存在，则在 root 下创建 '/api' ，继续在创建的 '/api' 节点下查找 '/v1' 因为 '/api' 是新建的 v1 一定也不存在，则继续创建v1
3. 字典树url匹配（match）： 在这个最简单的版本下，匹配逻辑与指定节点的查找逻辑没有区别。

还有一些不涉及递归或者复用上面逻辑递归操作的简单操作

4. 修改字典树节点（modify）： 通过 find 逻辑找到指定节点，调用 set 方法或者直接赋值的方式修改节点内容。
5. 删除字典树节点（delete）：通过 modify 逻辑修改 isdeleted 标为 true，并把节点内容 modify 为空。节点本身的内存不释放。
6. 重建字典树（rebuild）： 遍历所有节点，添加到新树，如果 isdeleted为 true 则不添加到新树，通过rebuild 操作创建副本。

由上可知，标准字典树结构距离通用的路由引擎底层数据结构能力还有一定差距，缺乏统配描述能力，缺乏变量表达的能力，下面我们来看一下如何进行改进。

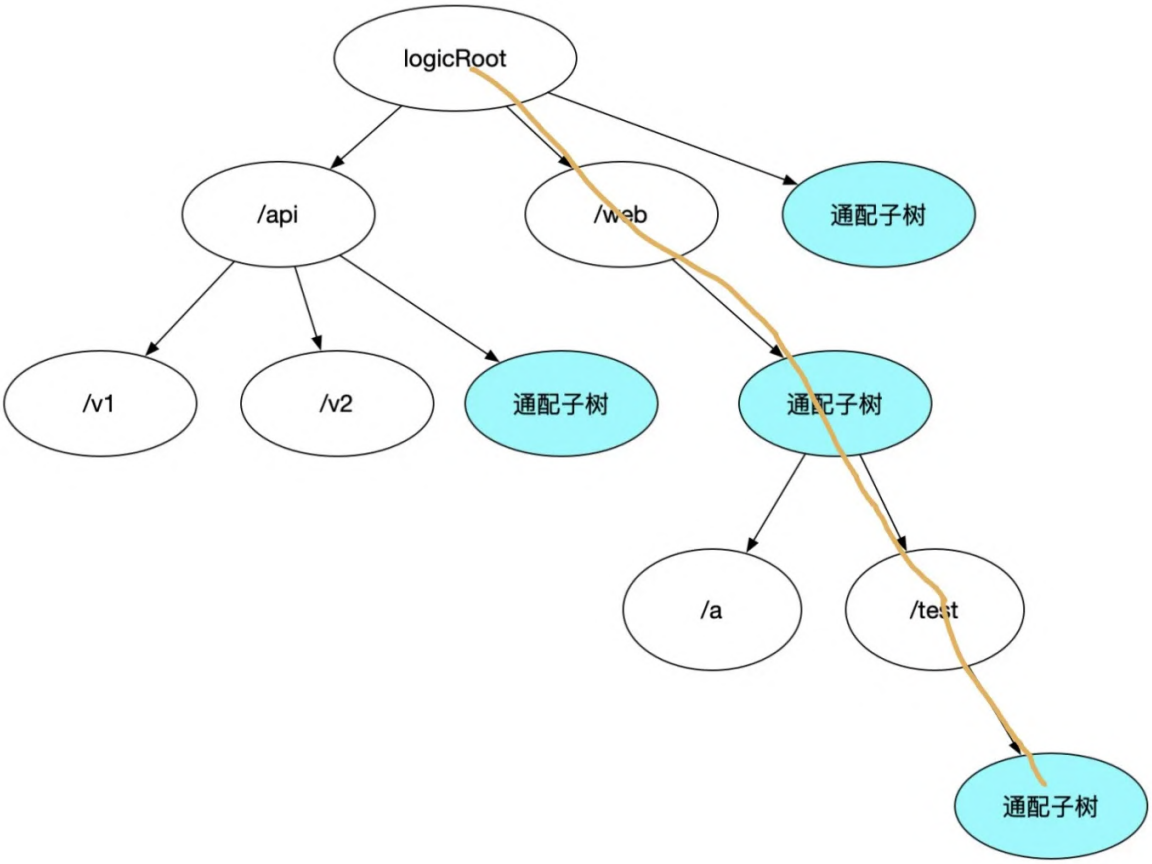
添加 描述统配逻辑的子树，作为子树中默认存在的一部分



现在我们的变种字典树多了变量表达能力

'/web/:appname/test/*' 这样的url 在图中应该怎么表达?

没错就是这个路径



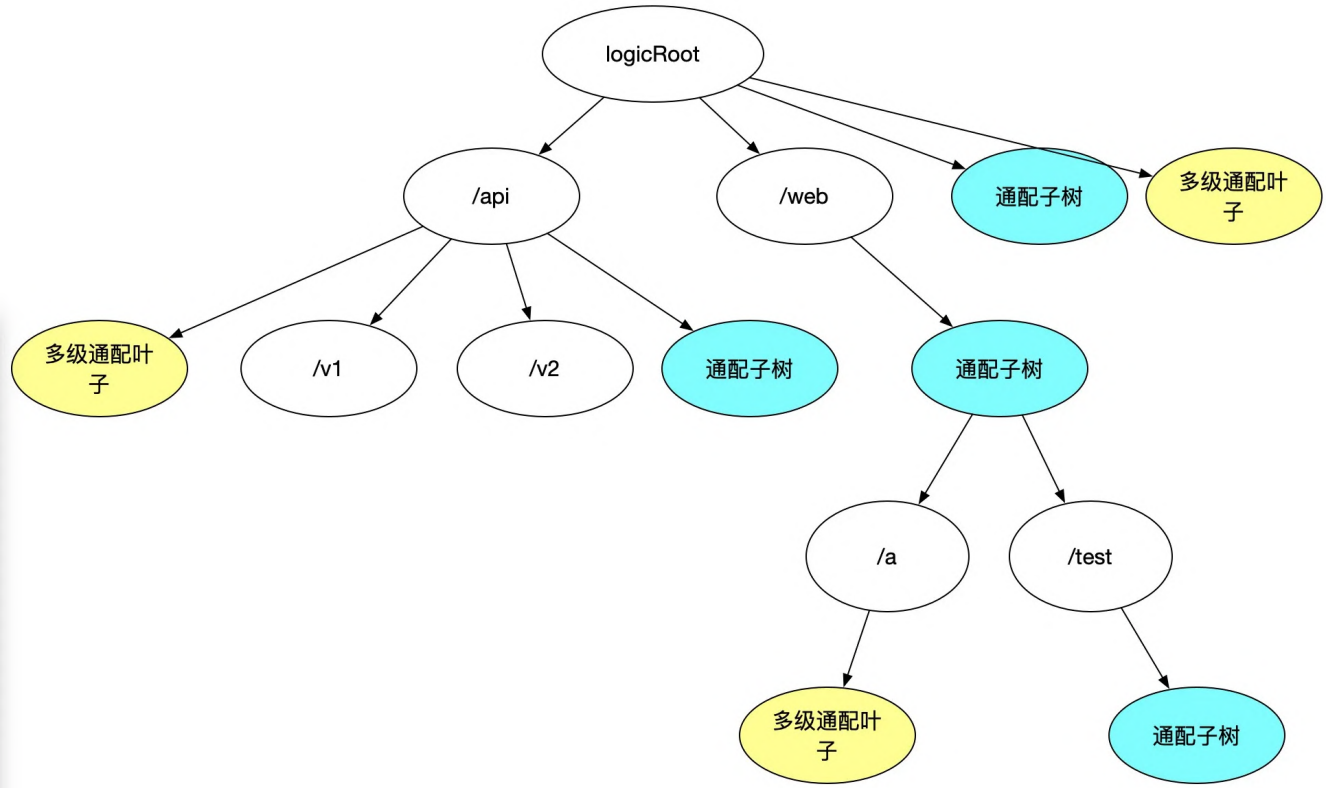
继续分析字典树几个关键的操作是否需要做变化?

1. 字典树指定节点的查找 ：
    a. 如果不改动使用前一版本逻辑在 '/*' 节点处理之前都不会有问题： 从root 开始遍历字典书，'/api/v2/*' 称之为路径，在当前层级寻找指定路径，如果存在就继续在子树下完成剩下的路径匹配。 /api/v2 先从 logic root 找到 '/api' ，并在 '/api' 的子树下继续查找剩下的路径 '/v2' 。
    b. 这版本我们加上对 '/*' 节点的处理：'/v2' 后是 '/*' ，'/*' 对应单级通配节点，继续递归查找 '/v2' 节点下一级通配节点是否为空。如果 path 是 '/api/v2/*/test2' 这样的路径则继续在统配子树下完成递归过程。
2. 字典树节点的添加 ：
    a. 在添加 '/*' 节点之前，所有逻辑上一版本就足够处理：尝试查找指定节点，如果指定节点不存在则新建一个节点。假设一个空树状态下添加 '/api/v1/*' ，因为是空树那么 logic root 下查找 '/api' 一定不存在，则在 root 下创建 '/api' ，继续在创建的 '/api' 节点下查找 '/v1' 因为 '/api' 是新建的 v1 一定也不存在，则继续创建 v1。
    b. 这版本加上 '/*' 的特殊处理 ： '/v1' 新建后，查看通配子树，通配子树不存在，则为V1 节点添

加内容为空的单级通配子树并在子树中继续递归。

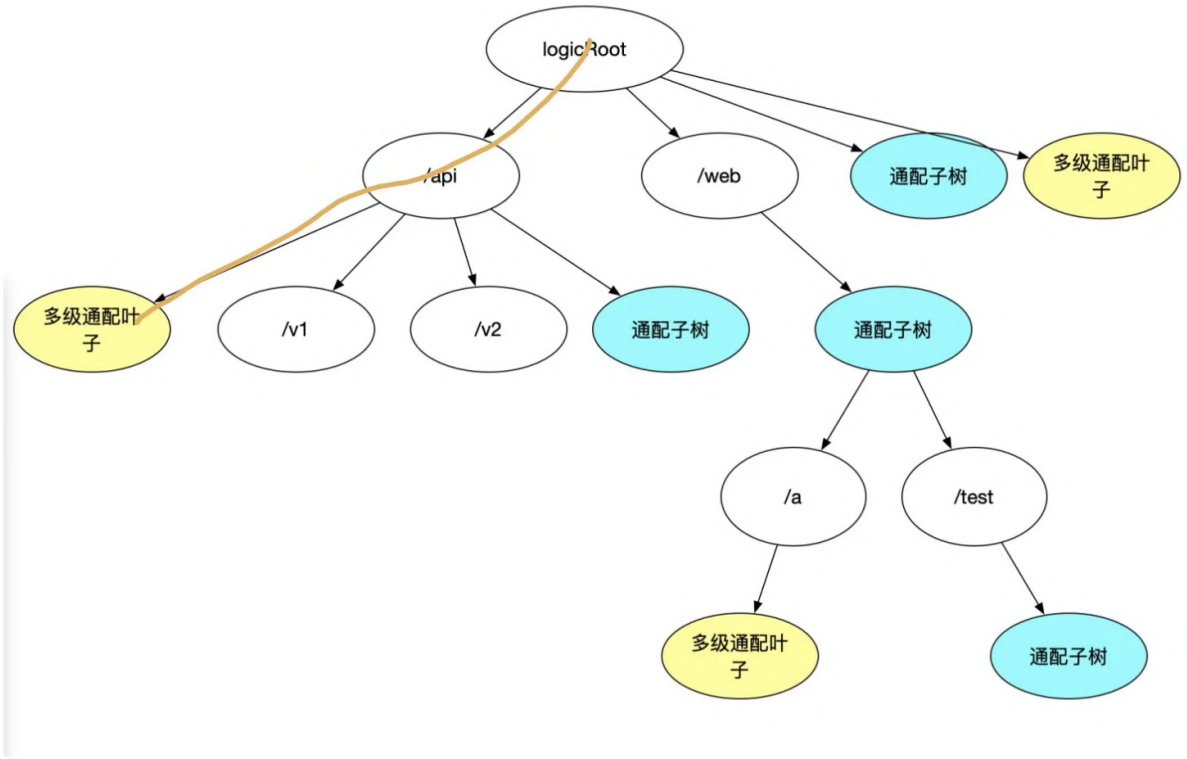
3. 字典树url匹配：在这个版本下，对比查找逻辑需要增加回朔逻辑。
    a. 在遇到通配节点前逻辑与find 依旧相同 ： 从root 开始遍历字典书，'/api/v2/*' 称之为路径，在当前层级寻找指定路径，如果存在就继续在子树下完成剩下的路径匹配。 '/api/v2' 先从 logic root 找到 '/api' ，并在 '/api' 的子树下继续查找剩下的路径 '/v2' 。
    b. 在处理统配节点的时候会与 find 逻辑有所不同：'/v2' 下普通子树无匹配节点，回朔到通配子树，查看是否能匹配，这个例子中 '/v2' 下无通配子树，查询不到节点 。值得注意的是回朔逻辑的先后顺序，是先找普通子树再回朔到通配子树还是先查找通配子树再回朔到普通子树是取决于优先级规则的，按照需求必须是先查找普通子树。

但是我们目前还是缺乏 '/**' 这种通配的表达能力代表了多级通配，可以分析需求得到结论，这种通配符，一定不存在子树，是一种特殊的叶子结点，仅用于 match 逻辑回朔时做特殊判断。继续加点特殊 node 后演化为：



好了至此，需求都能满足了。

'/api/**' 等价路径为：



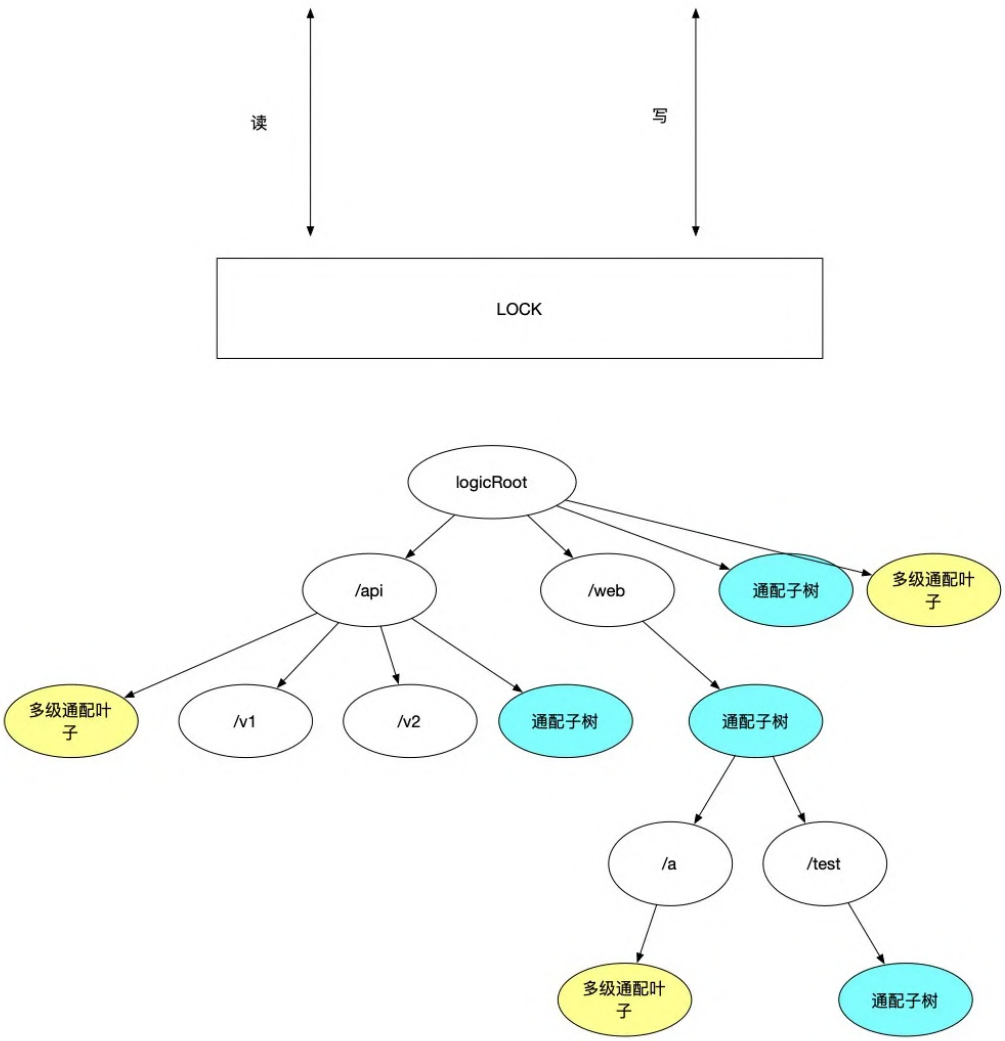
其他逻辑大同小异，match 逻辑回朔再多一级判断，如果一级通配子树也匹配不到结果，则再看一下多级通配子树是否为空（其实留一个标位就可以，为了统一模型好理解，还是用一个子树去描述）

到目前这个版本所有上文提到的能力已经都能有效支撑，回头分析一下时间复杂读。
url 被 '/' 分割出一个一个的段，容易理解在匹配一个url 过程中复杂度是 O(n) n= url 段数。与树中存有的规则数量无关。再分析 n 的范围，n 其实不是一个可以无限大的数字，一部分浏览器甚至约束 url 长度必须小于 2000，按照一个单词长度为 5 来计算，可以大概估计段数最多会在 400 左右，n 如果可以被视为一个常数，那么复杂读可以看作是 O（1）。

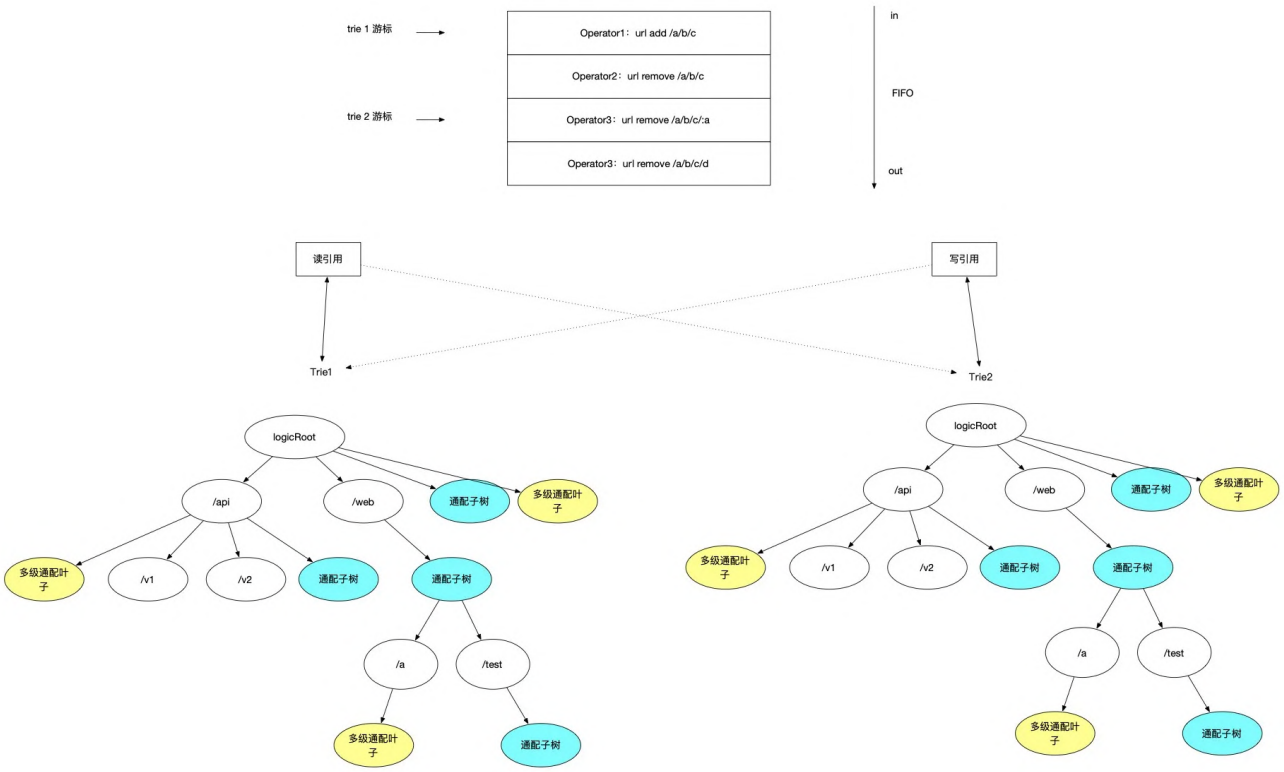
稍微解释一下find 和 match 有什么不同，为什么需要两种查找节点的方法。看下这个例子 :假设树中已经add 了 '/api/v1/:name/add' 这个 path，那么
find（"/api/v1/:name/add"），find（"/api/v1/*/add"）两个调用应该能够拿到结果，在add 的过程中用于冲突判断。
假设有请求进来url 为 '/api/v1/:name/add' 那么match（"/api/v1/:name/list"）也应该能 match 到结果且变量name 为 :name。
再假设有请求进来 url 为 '/api/v1/yq/add' 那么match（"/api/v1/yq/list"）也应该能 match 到结果且变量name 为 yq 。find（"/api/v1/yq/add" ） 则不会匹配到结果。

后续改进

目前实现在读树和写树之前，竞争一把全局锁，竞争失败后自旋直到竞争成功，然后完成读写。
解释一下为什么读都需要上锁，因为代码中大量运用了go 的 map 结构。这个结构只要并发读写直接会报如下错误：concurrent map read and map write
目前实现如下



引入 command 队列，所有对 trie 的用户写操作先入列，同时做读写分离，分为读树和写树，维护一个线程负责追 log 把 command 写入到写树，读树因为只读，没有写入线程操作读树所以可以不加锁。写树因为只有一条线程向树内写入，没有竞争问题，也可以不加锁。（写入操作并不会很频繁单线程完全能负荷）
定义一个配置延迟生效的时间，比如3s
每3秒，读树和写树角色切换，每个 trie 分别维护一个 command 队列的游标，游标代表当前这个 trie，追 log 追到了哪条记录，写入线程每3s 切换写游标引用。



如上图，最上面部分是一个先进先出的 command 队列，追 log 线程从这个队列中读取用户写操作，这个队列维护了两个游标 index1，index2，index1 代表了trie1 追 log 追到了 index1 的位置，index2 代表了 trie2 追 log 追到了 index2 的位置。追 log 线程同一时间内只会使用一个引用进行写操作，每次写完树对应的 index 游标下移一格，另一个 trie 引用将被用于读操作，一切读请求将从读引用对应的树中读取。因为追的是同一份 log ，最终一致性是能保证的。

切换逻辑：

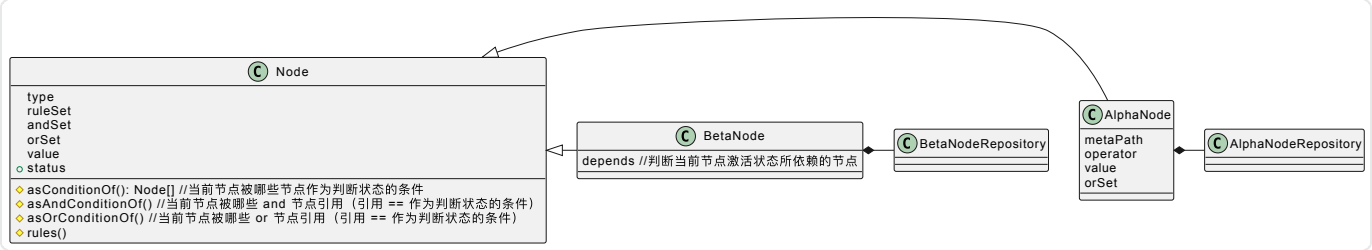
1. 先使追 log 线程空转（不挂起，避免上下文切换，因为马上要恢复）
2. 保证两个树都没有写入线程操作
3. 切换读引用到另一个树
4. 切换写引用到另一个树
5. 恢复追 log 线程

pr：

https://github.com/apache/dubbo-go-pixiu/pull/262

pkg/common/router/trie/trie.go:26

模型设计


Node
type
ruleSet
andSet
orSet
value
status
asConditionOf(): Node[] //当前节点被哪些节点作为判断状态的条件
asAndConditionOf() //当前节点被哪些 and 节点引用（引用 == 作为判断状态的条件）
asOrConditionOf() //当前节点被哪些 or 节点引用（引用 == 作为判断状态的条件）
rules()
BetaNode
depends //判断当前节点激活状态所依赖的节点
BetaNodeRepository
AlphaNode
metaPath
operator
value
orSet
AlphaNodeRepository

标准字典树做contains 批量判断

contains 是一个相对常用的操作符，“abcde”contains“cd” 结果为true

字典树做contains 批量判断

假设有如下alpha 规则集合

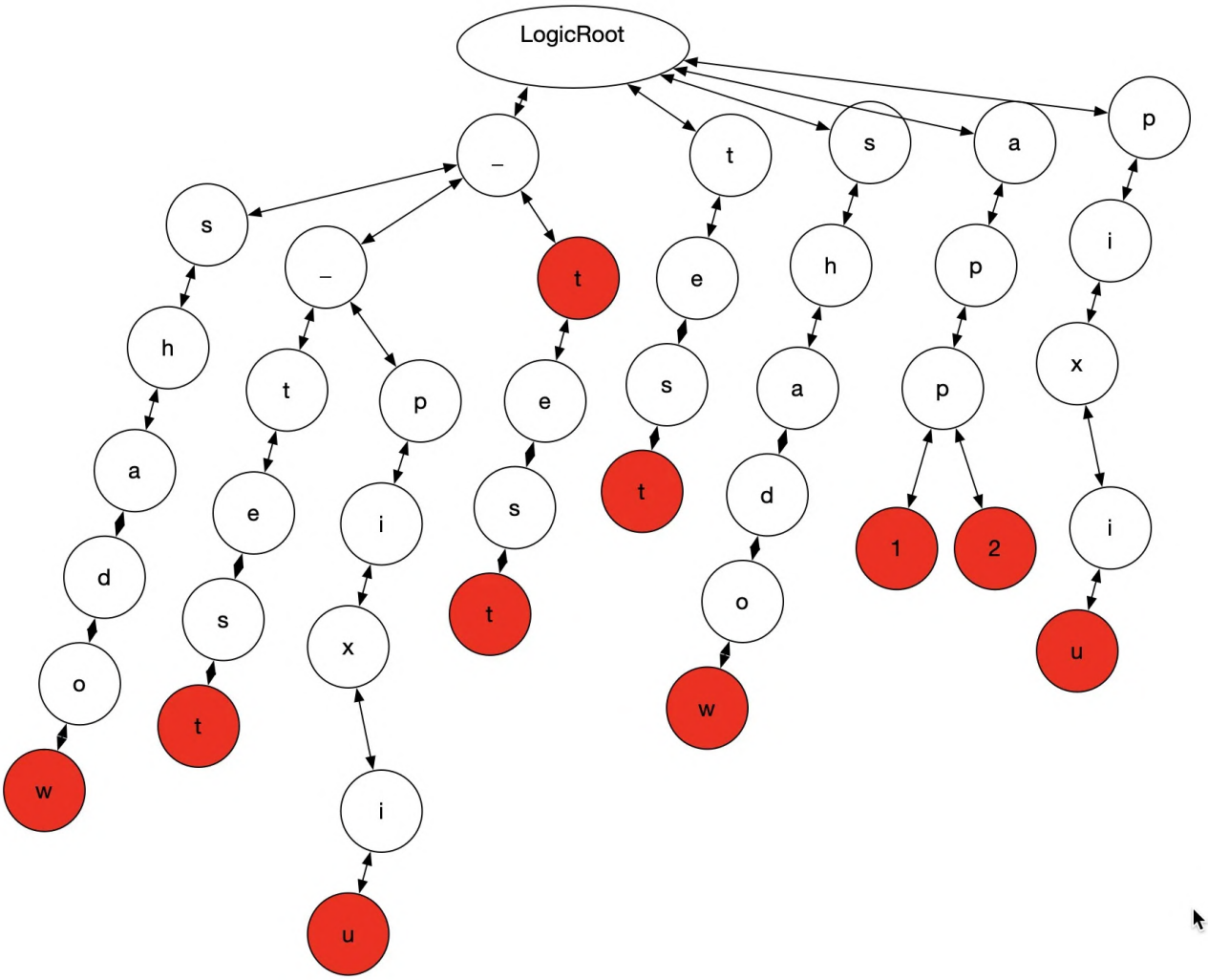
header.source contains "__test"
header.source contains "_test"
header.source contains "test"
header.source contains "shadow"
header.source contains "_shadow"
header.source contains "_t"
header.source contains "app1"
header.source contains "app2"
header.source contains "pixiu"
header.source contains "__pixiu"

分别在不同的条件下被使用，实例数据

```
header{
    source:"_tes_win_the_game___routeBy_pixiu"
}
```

进入batch tester如何复杂度较低的批量完成测试?

构建如下结构标准字典树:



构造输入传前缀集合

_tes_win_the_game___routeBy_pixiu

tes_win_the_game___routeBy_pixiu

es_win_the_game___routeBy_pixiu

s_win_the_game___routeBy_pixiu

_win_the_game___routeBy_pixiu

win_the_game___routeBy_pixiu

in_the_game___routeBy_pixiu

n_the_game___routeBy_pixiu

_the_game___routeBy_pixiu

the_game___routeBy_pixiu

he_game___routeBy_pixiu

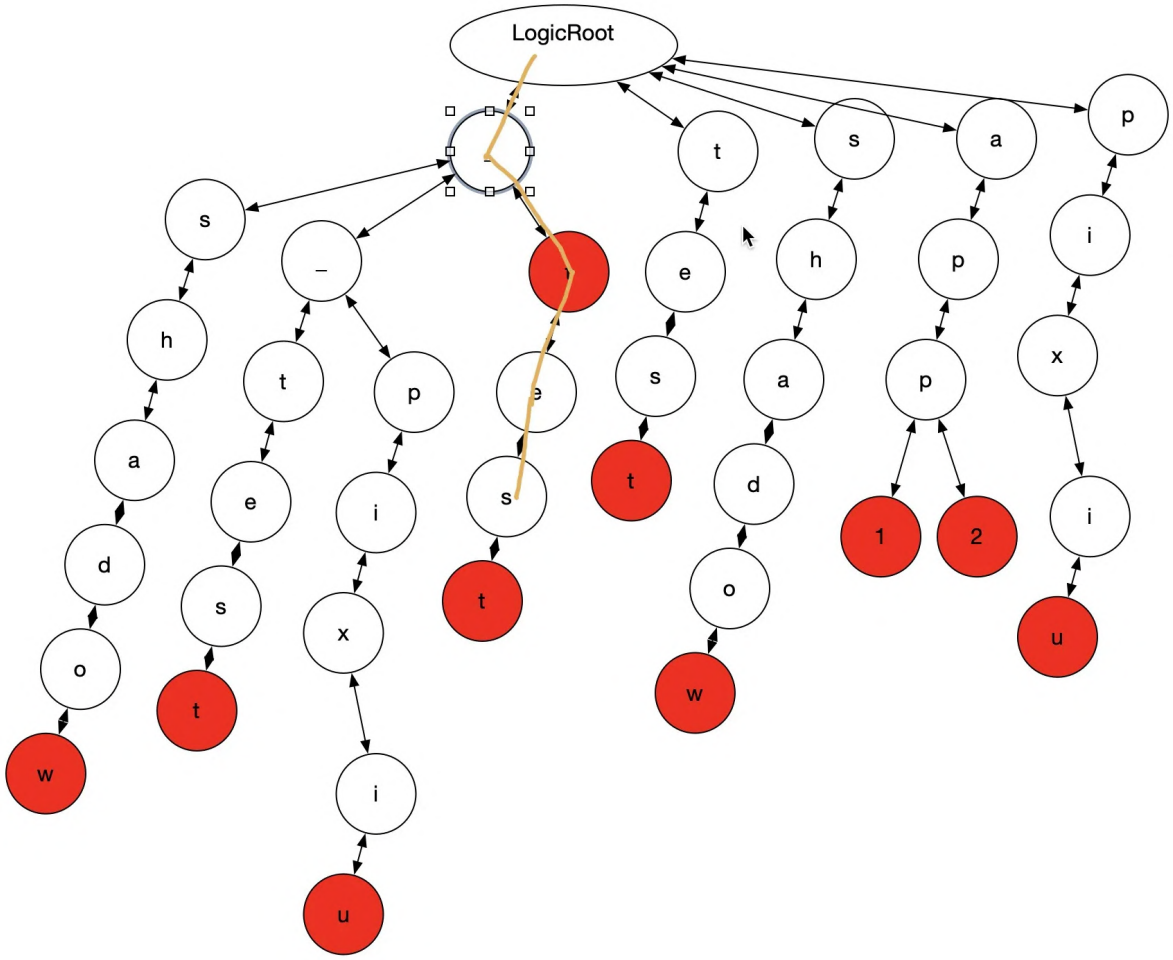
e_game___routeBy_pixiu

_game___routeBy_pixiu

game___routeBy_pixiu

ame___routeBy_pixiu
me___routeBy_pixiu
e___routeBy_pixiu
___routeBy_pixiu
__routeBy_pixiu
_routeBy_pixiu
routeBy_pixiu
outeBy_pixiu
uteBy_pixiu
teBy_pixiu
eBy_pixiu
By_pixiu
y_pixiu
_pixiu
pixiu
ixiu
xiu
iu
u

拿前缀集合中的单条记录 _tes_win_the_game___routeBy_pixiu 分析
以_tes_win_the_game___routeBy_pixiu 作为路径在字典树种寻路 能够走出如下路径



到_win开始 已经无法在字典树中找到路径了，途经了一个红点，红点代表“_t”就是这个字符串 contains 的 一个value，对应的rule 就是合法的alpha node 需要添加到输出集合。

前缀集合每个字符串都重复上述步骤可得到所有满足条件的alpha node set。
在待匹配集合恰好为输入串的前缀集合时，性能最差
复杂度O（n平方） n = 输入字符串长度，也能做到与字典树内待匹配条数无关。

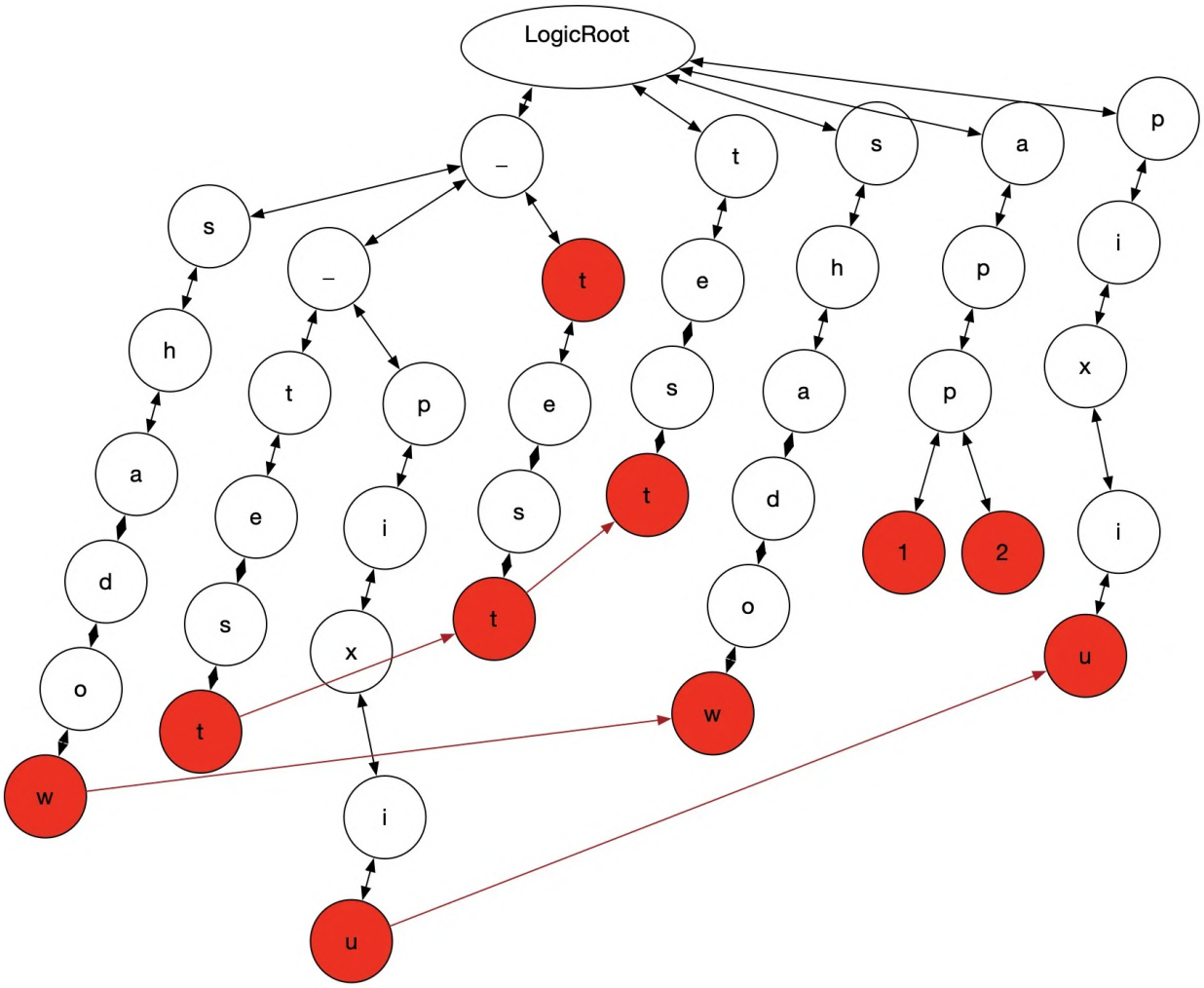
一般对象建模时字段都会有相应长度约束，在适当的建模约束的前提下复杂度可以视为O1 ，例如规定 header.source.length <= 300。

有限状态自动机

可能不是很有必要的一些优化，牺牲了易读性，读取效率从 N方 优化到 N，写入效率从ON 退化到 N方，权衡普通需要被路由的业务请求 与 通过网关RDS接口变更路由信息的比例。决定是否使用有限状态自动机。

可以发现上面的match 过程 在发现能匹配上 "_pixiu" 的前提下，居然还需要好几步计算遍历很多中间无意义结果(回朔过程)才能得出结论同时还 匹配上了 "pixiu"

参考KMP 生成模式串的思路，在构建字典树的时候添加回朔指针，形成有限状态自动机。可以进一步优化到On n=输入字符串长度，至此已经优化到了理论极限。（你不可能字符串都不遍历一遍就有足够的信息判断）



以下部分未完成

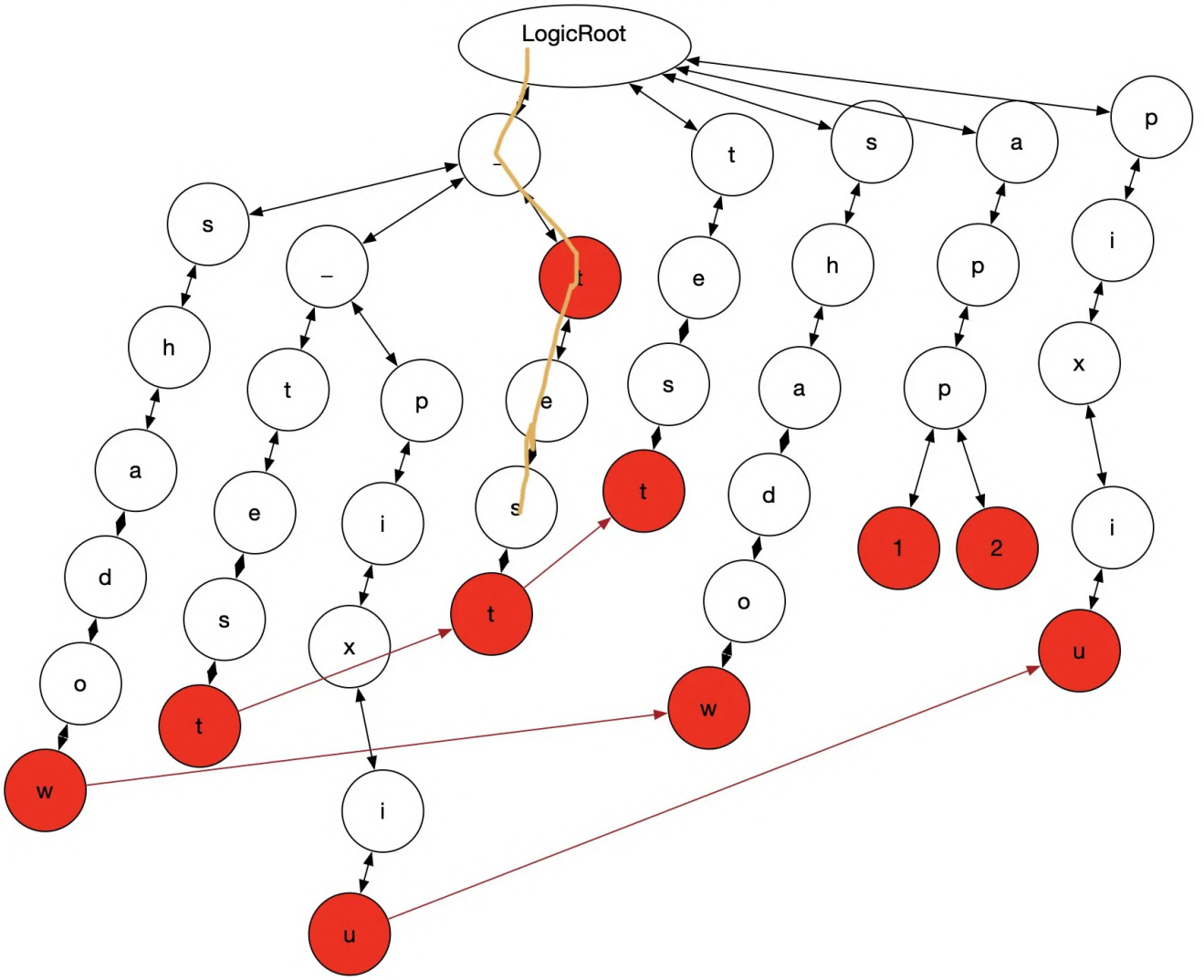
todo：

1. 回朔指针构建方式
2. 添加回朔指针后的遍历过程

回朔指针的构造方式

回朔指针的构建是一次BFS的过程，依赖

已经遍历 _tes 未遍历 __te__testCharge



到了S 节点以后发现无'_'子节点且发现无回朔指针，重新回朔到logicRoot（默认回朔到root的指针都省略维护因为root 很好获取），继续遍历输入到如下状态

已经遍历 _tes 未遍历 __te__testCharge

到如下状态后 期望的w 子状态不存在，且无回朔指针，又回到logicRoot
继续往后遍历中间步骤不画了，直到这样一个遍历状态
已经遍历 _tes_win_the_game___routeBy_pixiu 未遍历

urlMatch 的 BatchTester 实现

需求以及背景介绍

主流网关能力

1. 请求转发
2. 协议转换
3. 认证鉴权
4. 基于转发能力之上，做LB
5. 链路跟踪 （trace）
6. 服务发现
7. 流程编排?

这些能力在网关中都被抽象成 filter ，网关启动时根据配置动态组装filter 链，不同的filter 链决定了处理这个request 的流程。

因为http协议主流浏览器都支持以及用于实现http 请求的三方包成熟稳定且使用成本较低，一般网关入口流量大部分为http 请求。在这种情况下配置必须描述清楚什么特征的url，需要被什么样的filter 链处理。

同时因为网关需要对每一个请求都做处理，因此对于生产环境平均7毫秒的响应要求来说，对网关的性能要求是十分苛刻的。因此url匹配的逻辑一定不能随配置的url 数目变多而出现性能退化。

讨论后 需要实现的点包含

1. 与规则数目无关的匹配效率
2. url 可以包含变量，/a/b/:c/d 代表url 用“/”分割后，第三个部分的值作为变量c 的值，向filter 传递，
    a. request url 为 a/b/asa/d 就能匹配成功，且c = asa
3. 需要有通配符 * 代表一个层级任意字符的通配，** 代表从这个层级以后，子层级也可以是任意字符
    a. 规则为 /a/b/* 的例子下，能匹配/a/b/xxx , /a/b/bbb 等url 但不能匹配 /a/b/sd/d 这样多了一个层级的url
    b. 规则为 /a/b/** 能够匹配上 /a/b/sd/d
4. 能够根据header 等其他信息决定rule 是否匹配（下版本支持）

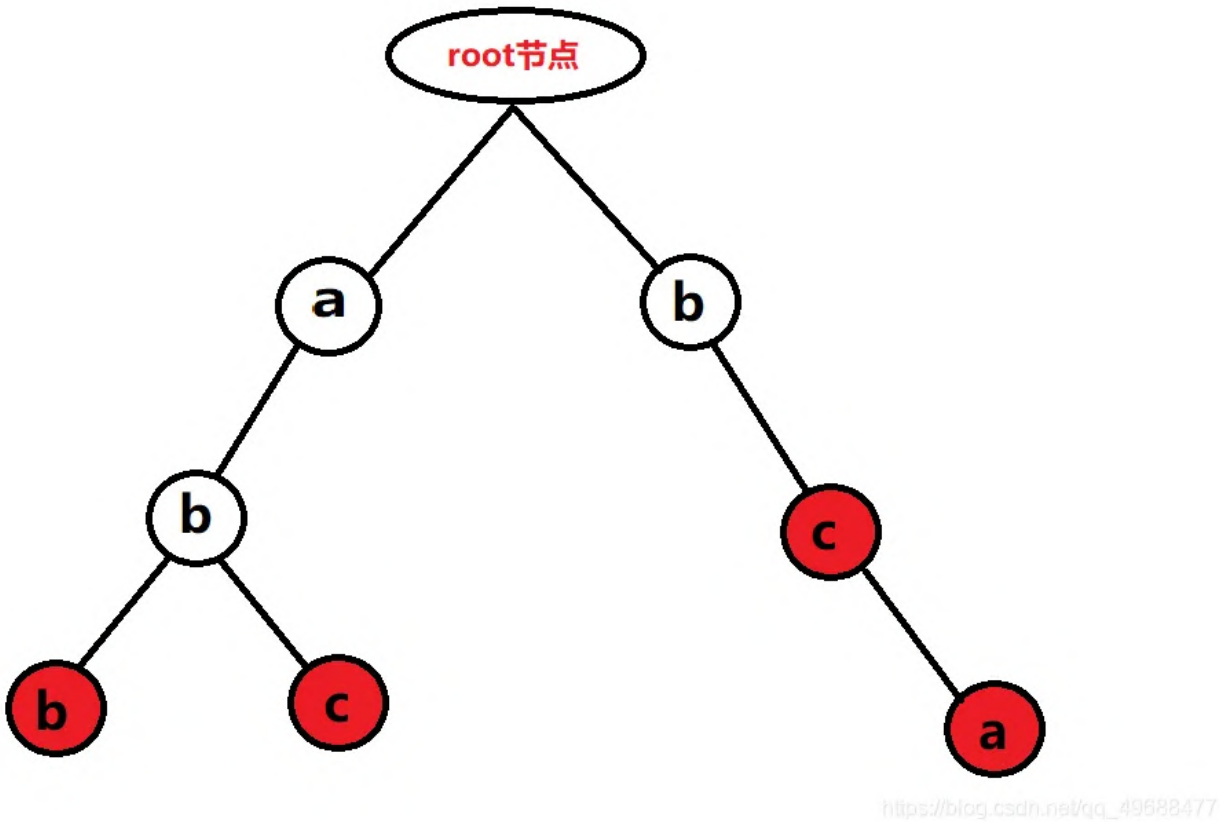
结合以上三个需求点容易联想到规则引擎，特殊的点在于规则引擎需要对url路径匹配做特殊的优化。

规则引擎可以参考rete 算法的图组织逻辑，路径匹配作为其中的一个阿尔法node ，单node下优化成字典树做路径匹配。

header信息匹配逻辑（rete 算法相关）实现会在下一迭代补齐，重点记录一下已经实现的字典树路径匹配。

字典树

普通的入门教程常见的字典树结构用于单词匹配，abb，abc，bca，bc 这四个单词组织成树是这样的一个结构



其中对字典树从根开始到每个叶子结点的路径代表了单词本身 ，每个节点存储了一个字母。遍历路径能得到hello

容易得出结论字典树的匹配逻辑算法复杂度是o（n）的但是n 指代被匹配的字符串长度（hello 的长度为5）而和一共有多少个单词被存储到了树内无关。

字典书变种

如果每个节点不存储单个字母，而存储 url path 下 两个“/”之间的内容会发生什么？没错从根到叶子遍历这样构建起来的字典树的到的就是url path，能找到这样的一个叶子结点就代表这个路径能匹配上。同理，找到和url path 相匹配的rule 的算法复杂度是O（n） n= url path 的 “/”分割后的字符串数目。在某些浏览器版本甚至会要求url 长度不超过2048字节。是一种能完全避免性能退化的算法。

变量处理

分析带变量的 rule

1.

/a/:b/c/d

/a/:c/c/d

这样的两个变量尽管命名不同，但是逻辑上其实无法区分应该匹配到哪条filter 链，所以应该提示冲突

得出结论：变量名不同的变量节点在树中应该是同一个node

1.
2. /a/:b/a
3. /a/:b/d/c

分析 :b 后的情况 在匹配到/a/:b 的前提下，变量后的处理逻辑与正常的字典树完全相同，/a 与 /d/c 组成的子树可以满足后续匹配

3. 单级通配处理

/a/*/a 处理逻辑其实等价于变量处理，区别在于变量需要记录下来当级url的值写到header 后向后传递，通配不需要，单级通配的逻辑是变量逻辑的简化。

4. 多级通配处理

多级通配意味着搜索到到这个node 的场景下就一定能完成匹配了，但是需要注意的是多级通配优先级应该是最低的，如果同级的变量子树和子树能match 的情况下应该返回子树match 结果。

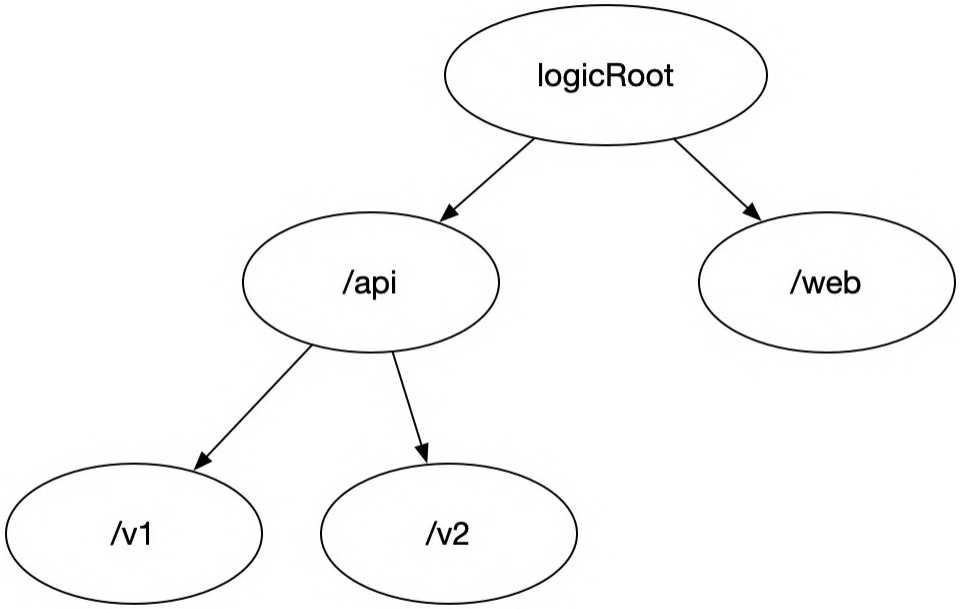
5. header 匹配处理（未实现）

计划参考rete 的组图方式，对比字典树，可以发现字典树是逻辑相对简单的一种rete，等于rete退化到只有alpha node 的情况。可以添加node 的种类加上beta node ，beta node 入度可以是2，引入beta node 后字典树会从树形结构演化成DGA。

思路演化

核心思想演变流程如下，下意识认为是个字典树表达结构。但是简单变种还缺了一些能力。

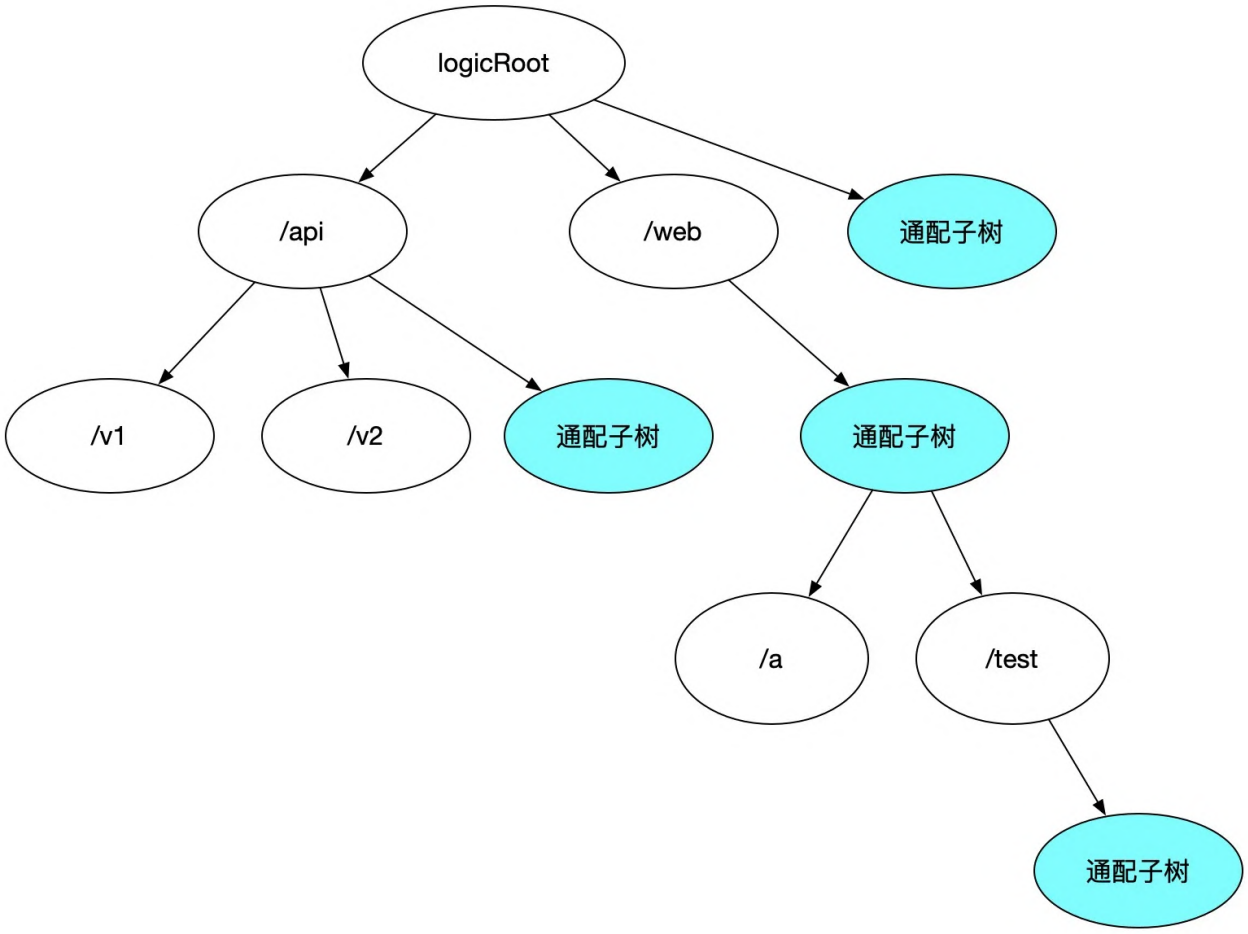
一个存放字符串作为node的变种字典树，具有表达url 的能力。



如上图描述等价于URL集合

/api/v1
/api/v2
/web

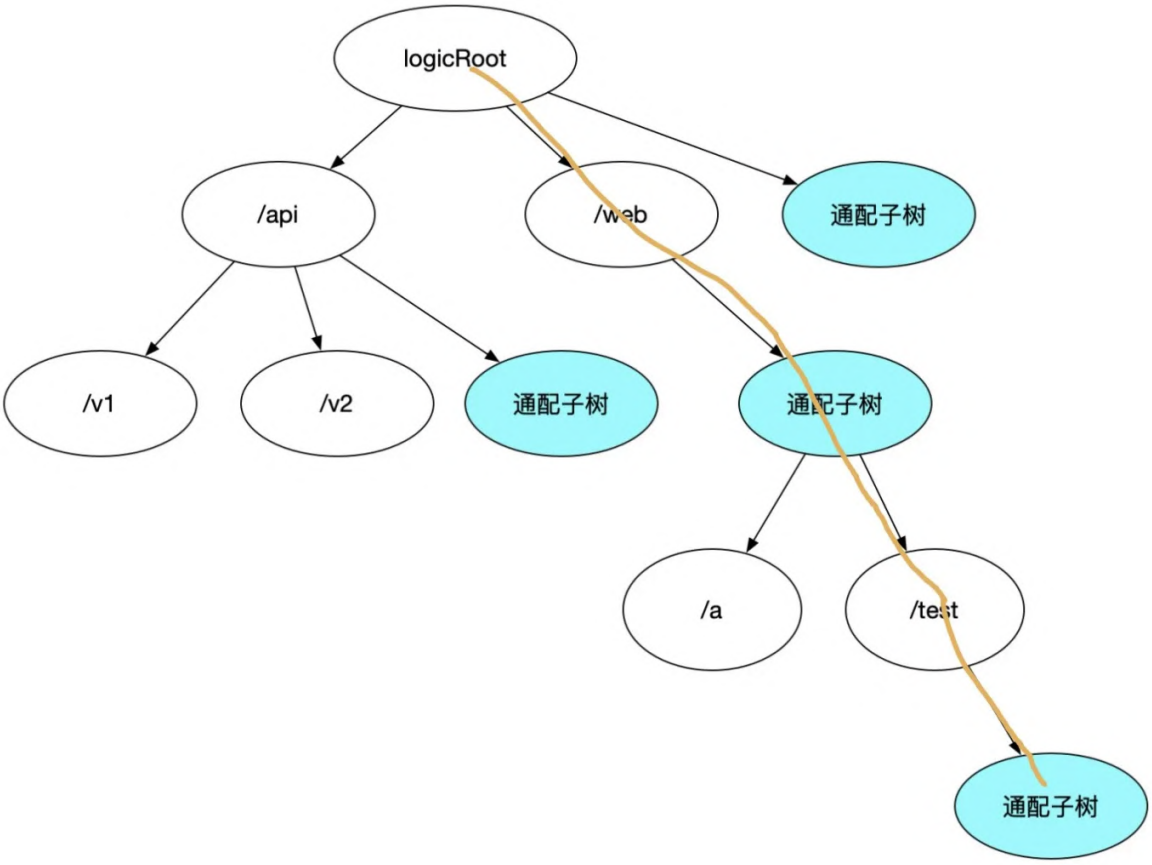
但距离我们的需求尚有一定距离，缺乏通配描述的能力，缺乏变量描述的能力，我们再加点料。
添加 描述统配逻辑的子树，作为子树中默认存在的一部分



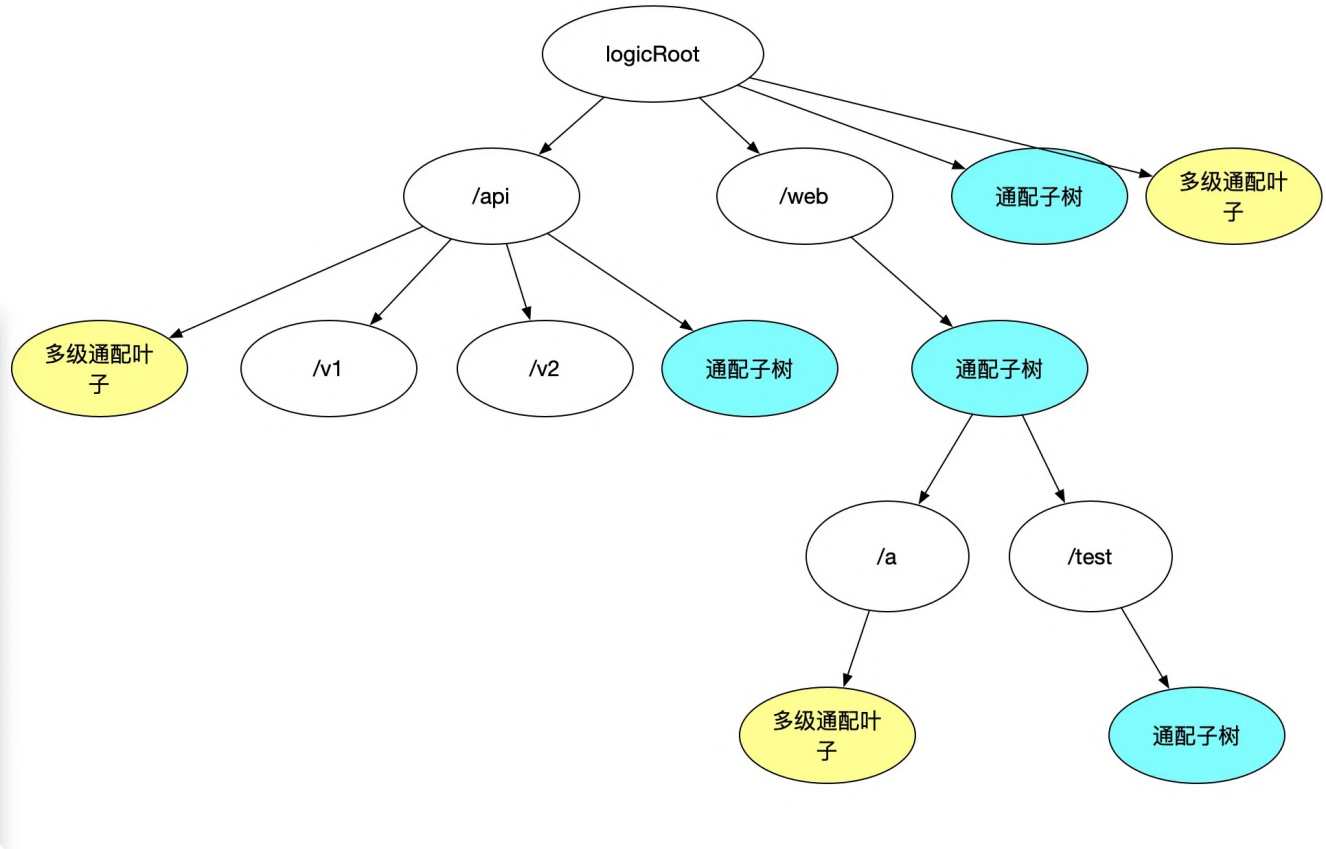
现在我们的变种字典树多了变量表达能力

/web/:appname/test/* 这样的url 在图中应该怎么表达?

没错就是这个路径

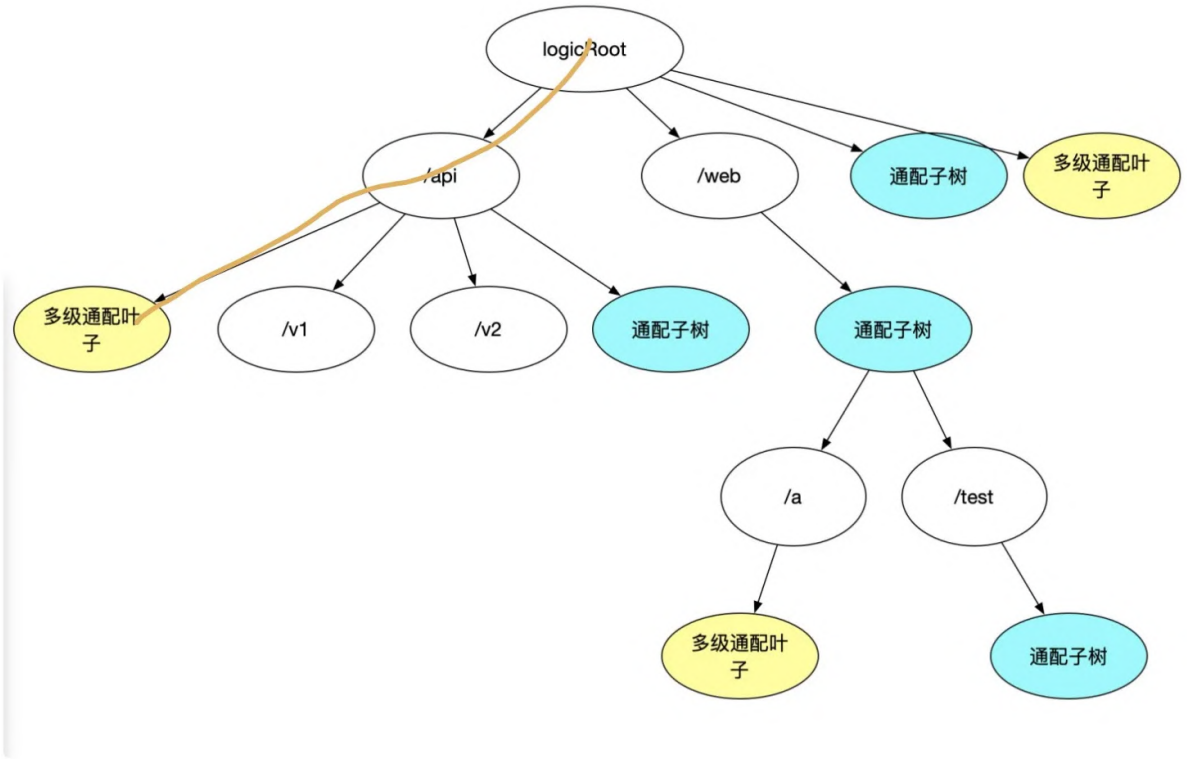


但是我们目前还是缺乏 /** 这种通配的表达能力代表了多级通配，可以分析需求得到结论，这种通配符，一定不存在子树，是一种特殊的叶子结点，用于回朔做特殊判断。继续加点特殊node后演化为：



好了至此，需求都能满足了。

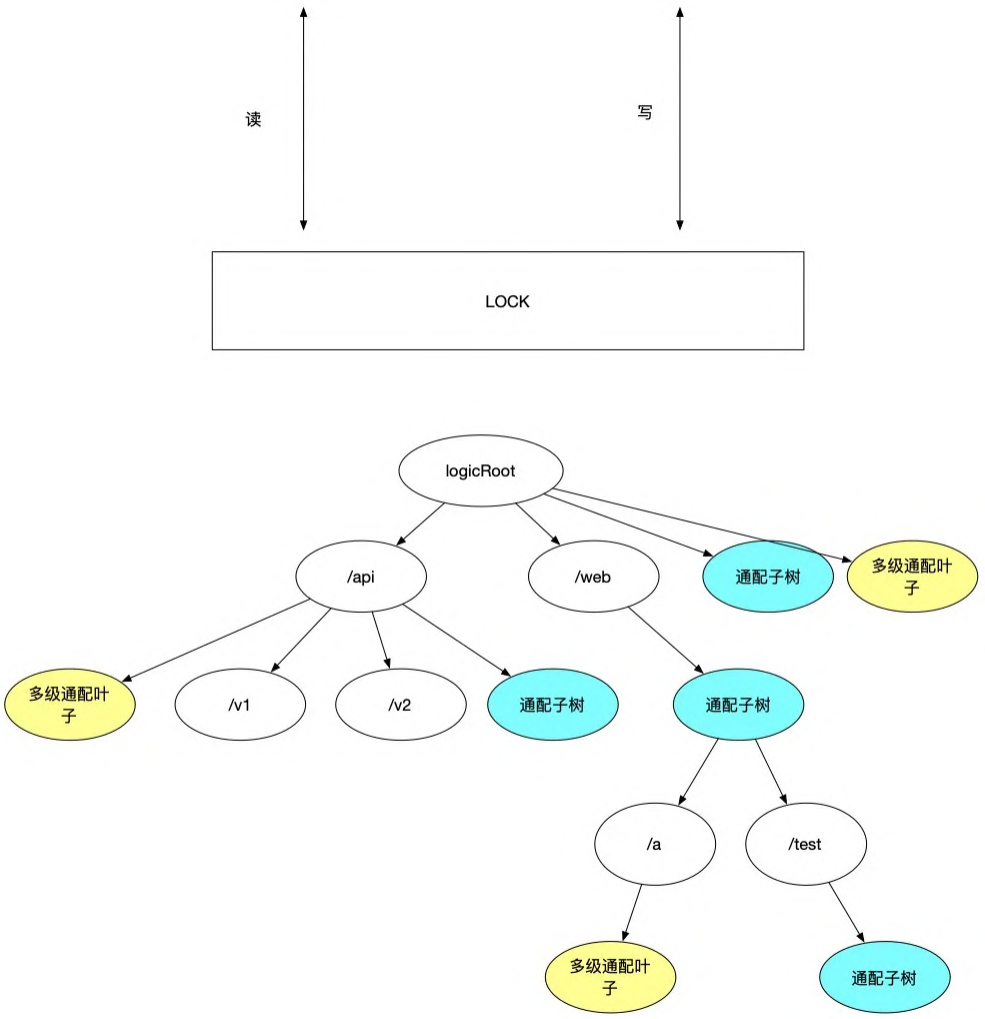
/api/** 等价路径为：



TODO 字典树无锁化改造

目前实现在读树和写树之前，竞争一把全局锁，竞争失败后自旋直到竞争成功，然后完成读写。
解释一下为什么读都需要上锁，因为代码中大量运用了go的 map 结构。这个结构在写的过程中存在 size 满后的resize 问题（确切表达是 超过某个阈值，简单理解为满），如果仅针对写加锁，读写不互斥，在写触发resize 过程的时候很可能会形成脏读。脏读在不同的使用场景下可能引发死循环等问题。

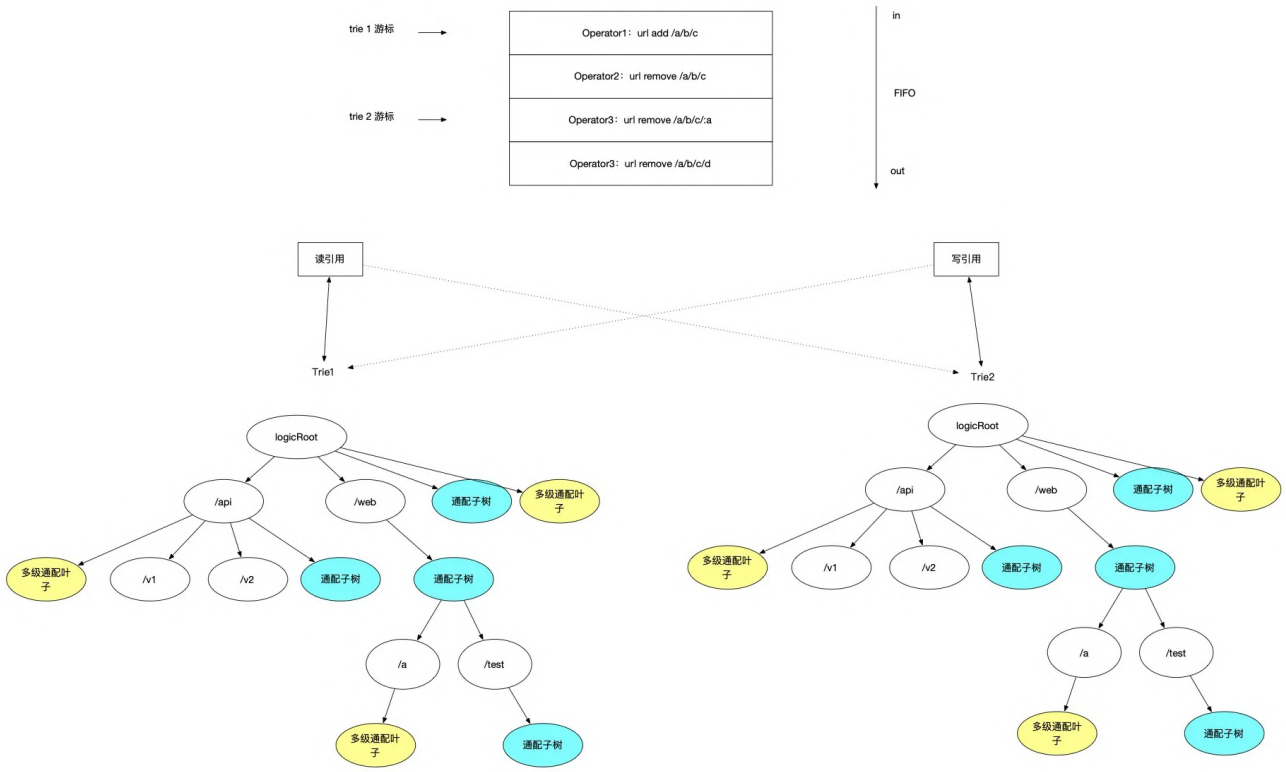
目前实现如下



引入 command 队列，做读写分离分为读树和写树，维护一个线程负责追log 把 command 写入到写树，读树因为只读，没有写入线程操作读树所以可以不加锁。写树因为只有一条线程向树内写入，没有竞争问题，也可以不加锁。

定义一个配置延迟生效的时间，比如3s

每3秒读树和写树角色切换，每个trie 分别维护一个command队列的游标，游标代表当前这个trie，追log追到了哪条记录，写入线程每3s 切换写游标引用。



切换逻辑：

1. 先使追log线程空转（不挂起，避免上下文切换，因为马上要恢复）
2. 保证两个树都没有写入线程操作
3. 切换读引用到另一个树
4. 切换写引用到另一个树
5. 恢复追log线程

pr：

https://github.com/apache/dubbo-go-pixiu/pull/262

pkg/common/router/trie/trie.go:26

WIP：开发手册

Pixiu Develop Document

一、快速启动

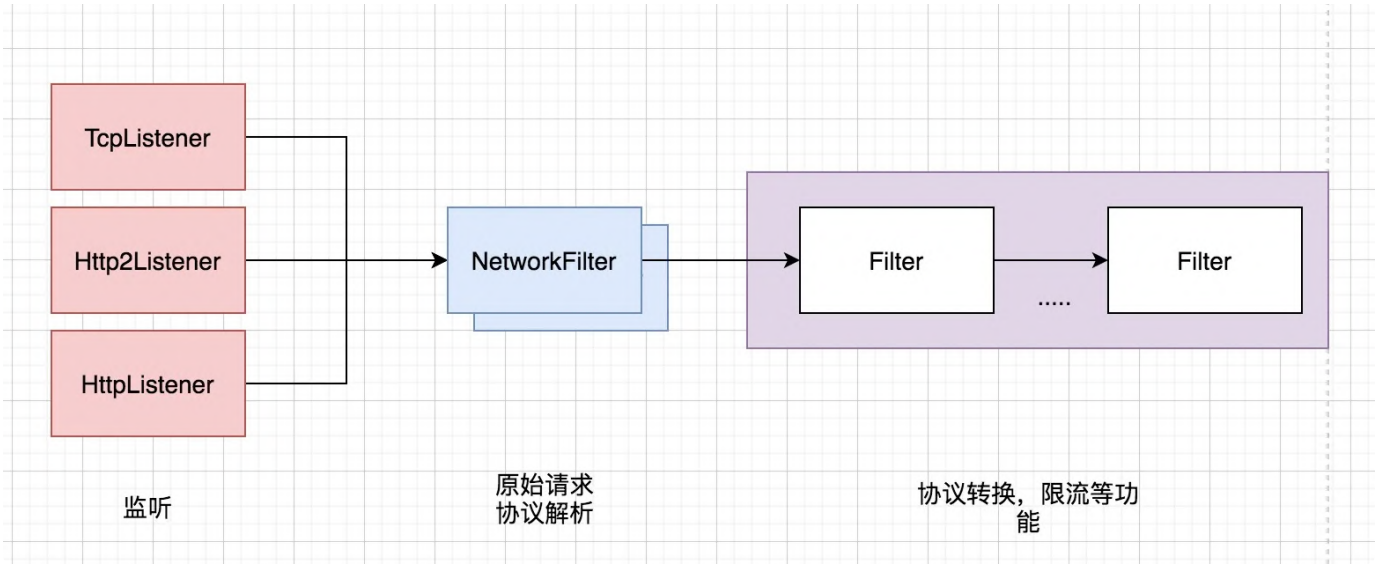
具体可以参考 readme 中的 quick start。start.sh 可以快速启动 dubbogo/simple 中的案例，其他的 samples 目前暂时只能手动启动，或者参考 start_integration_test.sh 进行启动。
较为重要的samples如下：

- samples/http/simple 常见的 Http 请求代理功能，作为常见的 API 网关；
- samples/http/grpc 将http请求转换为 grpc 请求，支持配置 proto 文件或动态从开启反射功能的 grpc server中获取 proto 信息；
- samples/springcloud http代理功能，从 spring cloud 服务注册中心中获取集群信息，动态管理 cluster 和 route 功能；
- samples/dubbogo/simple/resolve 将 http 请求转换为 dubbo 请求，按照默认http to dubbo转换规则；
- samples/dubbogo/simple/registry 从 dubbo 集群注册中心获取信息，动态配置相关路由和集群；
- samples/dubbotripleproxy dubbo2 协议和 triple 协议请求相互转换的案例（尚未merge）。

二、框架介绍

2.1 总览

pixiu 的整体目录结构和架构划分类似于 envoy，由Server、Listener、NetwrokFilter、Filter、Route 和 Cluster等组件构成，除此之外，还有 Adapter 用于和外界通信，动态获取集群信息，生成 route 和 Cluster。
其中，Listener 主要监听网络，并可能进行相关的编解码工作。NetworkFilter 是网络 Filter。主要工作是负责管理 Filter链，整合 cluster 和其他能力。Filter 则是主要提供扩展能力，提供协议转换和限流等基础功能。



2.2 源码调试

环境准备:

- dubbo-go-pixiu的源码
- dubbo-server(provider注册的样例)

ps: mac m1可能需要重新编译delve已编于创建debug进程

pixiu gateway启动需要的配置文件:

- 路由规则配置（api_config.yml)：samples/dubbogo/simple/proxy/pixiu/api_config.yaml
- pixiu各个组件的 (conf.yml)：samples/dubbogo/simple/proxy/pixiu/conf.yaml

api_config.yaml

```
name: pixiu
description: pixiu sample
resources:
    // 请求uri
  - path: '/api/v1/test-dubbo/:application/:interface'
    type: restful
    description: user
    methods:
      // 协议请求
      - httpVerb: POST
        enable: true
        // 是否上线
        onAir: true
        // 超时时间
        timeout: 1000ms
        // 入站请求
        inboundRequest:
          requestType: http
        // 转发请求
        integrationRequest:
          requestType: dubbo
           // 参数映射配置
          mappingParams:
             // url 参数名
            - name: requestBody.values
              // 映射到dubbo 接口第几个参数,不知道位置可以如下变量代替
               mapTo: opt.values
            - name: requestBody.types
               mapTo: opt.types
            //基于path中的路径进行参数匹配 /api/v1/application/interface
            - name: url.application
               mapTo: opt.application
            - name: url.interface
               mapTo: opt.interface
            - name: queryStrings.method
               mapTo: opt.method
            - name: queryStrings.group
               mapTo: opt.group
            - name: queryStrings.version
               mapTo: opt.version
          // provider 服务名
          applicationName: "UserProvider"
          // dubbo 接口类全路径名
          interface: "com.dubbogo.pixiu.UserService"
          method: "GetUserByName"
```

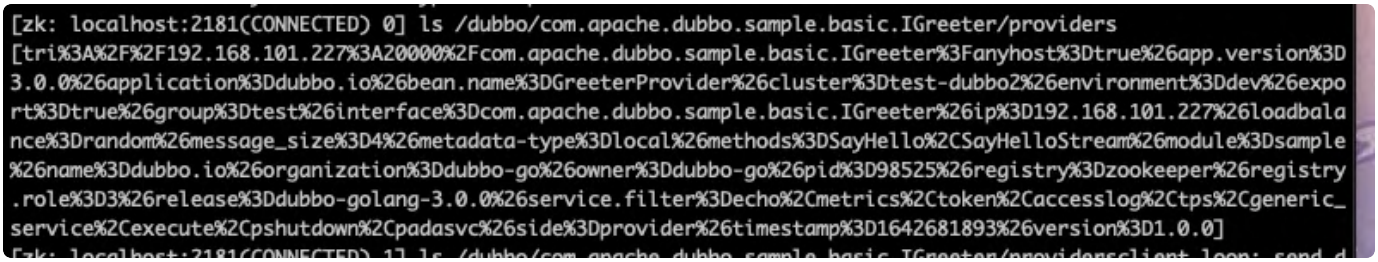
```
46              // dubbo 参数类型, 如果有多个, 按顺序一次填写(可以缺省)
47              paramTypes: [ "string" ]
48              // dubbo provider 端 group
49              group: "test"
50              // dubbo provider 接口版本
51              version: 1.0.0
52              // 集群名
53              clusterName: "test_dubbo"
```

conf.yaml

```
static_resources:
  listeners:
    - name: "net/http"
      protocol_type: "HTTP"
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8883
      filter_chains:
          filters:
            - name: dgp.filter.httpconnectionmanager
              config:
                route_config:
                  routes:
                    - match:
                        prefix: "/api/v1"
                      route:
                        cluster: "test-dubbo"
                        cluster_not_found_response_code: 505
                http_filters:
                  - name: dgp.filter.http.apiconfig
                    config:
                      path: ~/samples/dubbogo/simple/proxy/pixiu/api_confi
g.yaml
                  - name: dgp.filter.http.dubboproxy
                    config:
                      dubboProxyConfig:
                        registries:
                          "zookeeper":
                            protocol: "zookeeper"
                            timeout: "3s"
                            address: "node1:2181" //zk需要用户自己配置
                            username: ""
                            password: ""
                        timeout_config:
                          connect_timeout: 5s
                          request_timeout: 5s

                server_name: "test_http_dubbo"
                generate_request_id: false
      config:
        idle_timeout: 5s
        read_timeout: 5s
        write_timeout: 5s
  clusters:
```

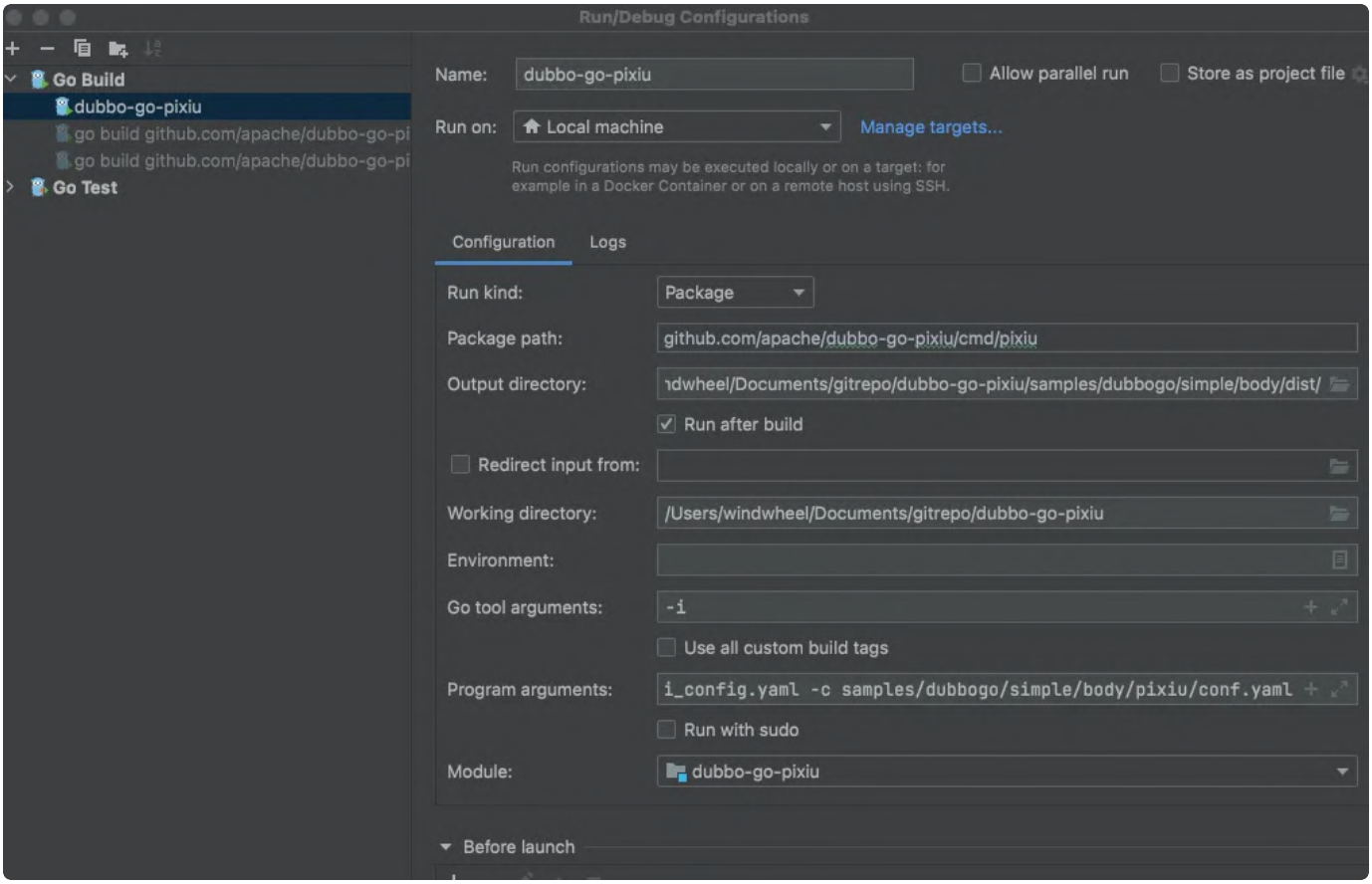
```
    - name: "test_dubbo"
      lb_policy: "RoundRobin"
      registries:
        "zookeeper":
          timeout: "3s"
          address: "node1:2181"
          username: ""
          password: ""
  shutdown_config:
    timeout: "60s"
    step_timeout: "10s"
    reject_policy: "immediacy"
```

- listeners用来匹配在api_config.yaml 当中我们设置的服务路由path，所要进行的协议转换
- adapter代表动态从注册中心比如zk,nacos获取相关服务实例
- filter 代表拦截网络请求 比如拦截从 /api/v1/test-dubbo/SayHello 发出的rpc请求会在 访问zk的相关节点 拿到/dubbo/com.apache.dubbo.sample.basic.IGreeter/providers 下的相关实例可以获取所有调用相关接口的请求 如下图所示



- cluster可以配置一些集群的负载均衡策略
- route 跟dubbo类似 条件路由，标签路由功能

接着,启动goland 在本地IDE配置如下:

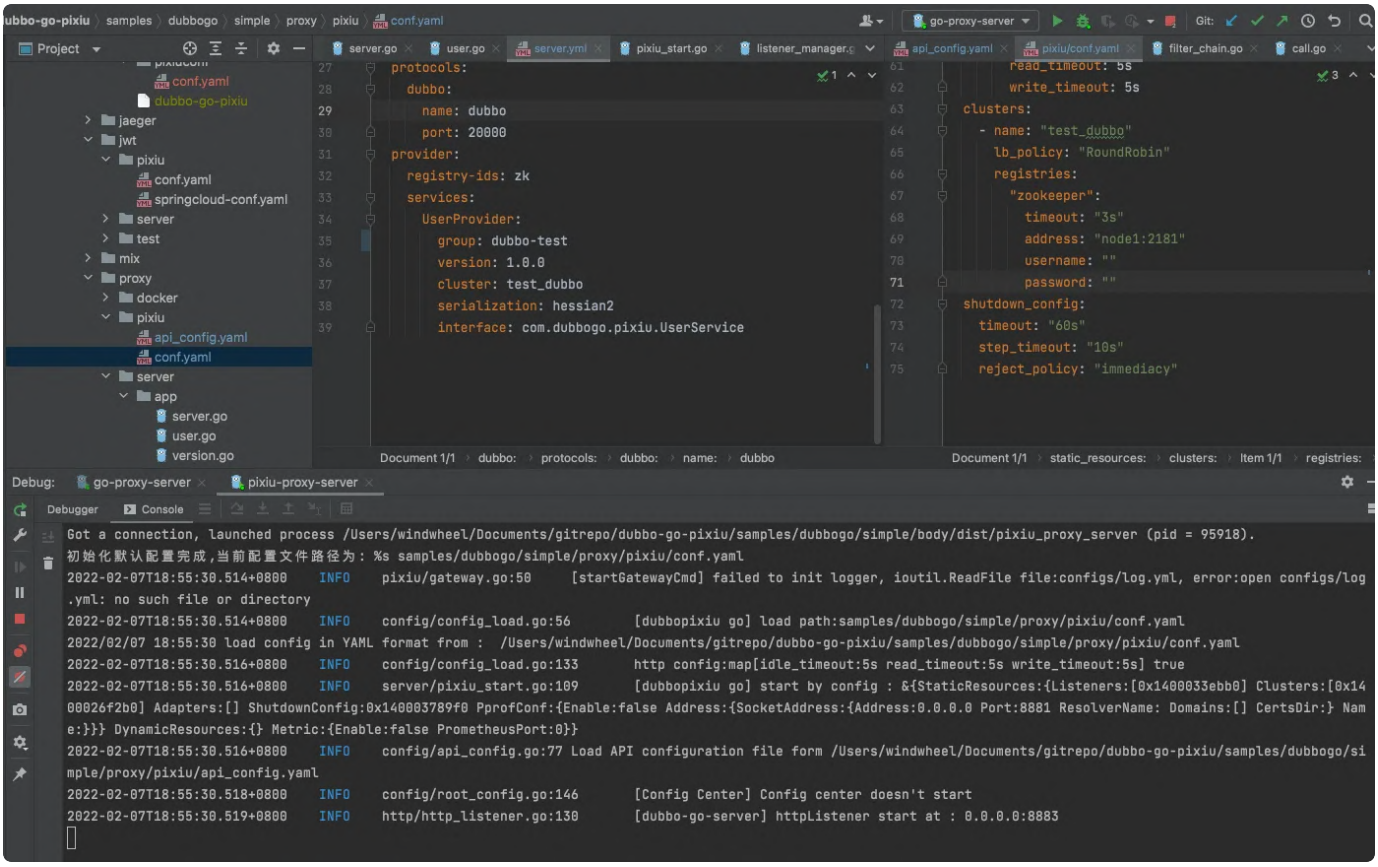


ps: $PROJECT_DIR 指的是用户配置的项目路径

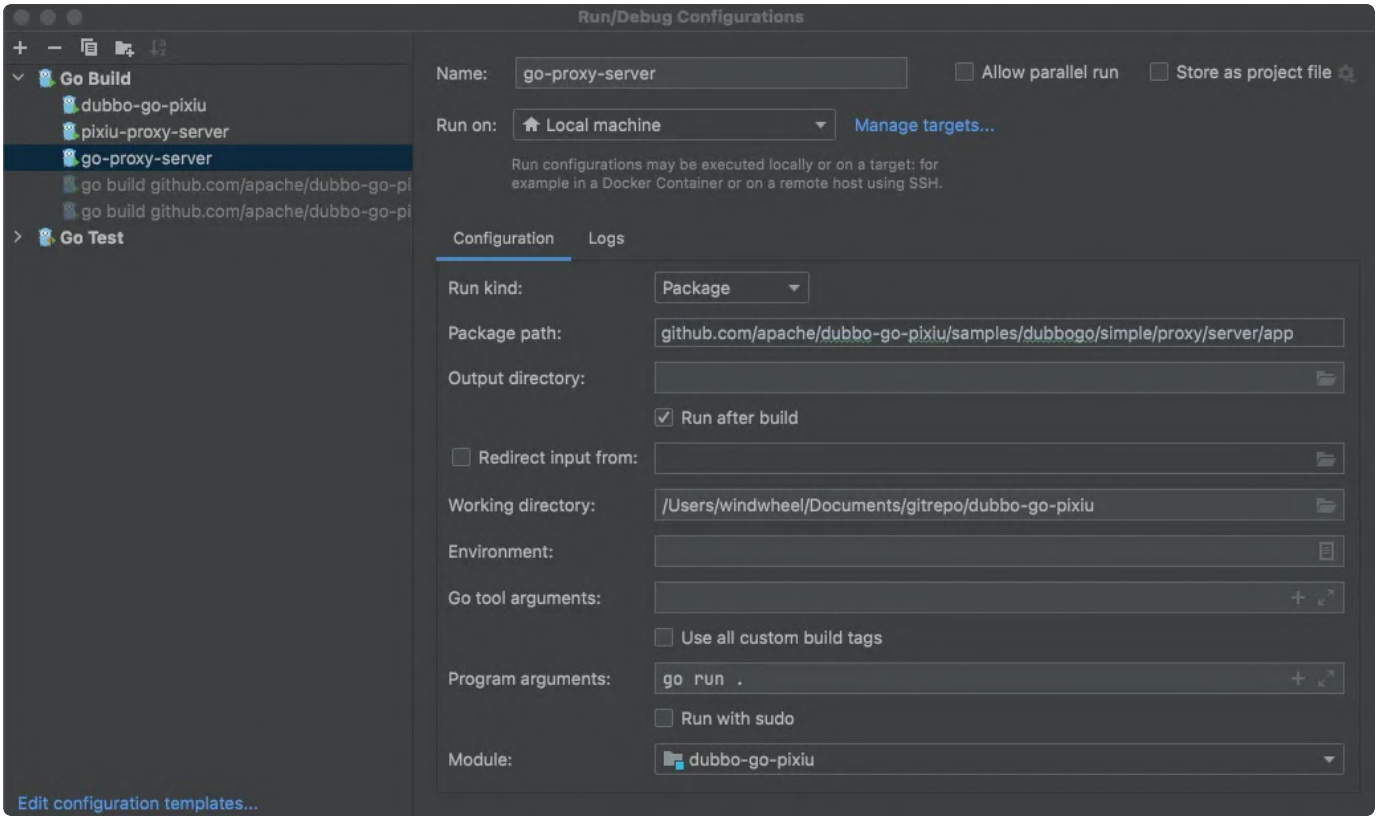
在上述配置当中 各个参数配置如下

```
Ouput Directory :

$PROJEDT_DIR/samples/dubbogo/simple/proxy/dist/

Program arguments:

gateway start -a samples/dubbogo/simple/proxy/pixiu/api_config.yaml -c samples/dubbogo/simple/proxy/pixiu/conf.yaml

```

出现如下界面说明pixiu gateway启动



接着 再建立一个注册provider实例的go-server



其中rpc接口定义

samples/dubbogo/simple/body/server/app/user.go

Plain Text

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.
0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implie
d.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package main

import (
  "context"
  "errors"
  "fmt"
  "sync"
  "time"
)

import (
  "dubbo.apache.org/dubbo-go/v3/config"
  hessian "github.com/apache/dubbo-go-hessian2"
)

func init() {
  config.SetProviderService(new(UserProvider))
  // ------for hessian2------
  hessian.RegisterPOJO(&User{})

  cache = newUserDB()

  t1, _ := time.Parse(
    time.RFC3339,
    "2021-08-01T10:08:41+00:00")

```

```
  cache.Add(&User{ID: "0001", Code: 1, Name: "tc", Age: 18, Time: t1})
  cache.Add(&User{ID: "0002", Code: 2, Name: "ic", Age: 88, Time: t1})
}

var cache *userDB

// userDB cache user.
type userDB struct {
  // key is name, value is user obj
  nameIndex map[string]*User
  // key is code, value is user obj
  codeIndex map[int64]*User
  lock      sync.Mutex
}

// userDB create func
func newUserDB() *userDB {
  return &userDB{
    nameIndex: make(map[string]*User, 16),
    codeIndex: make(map[int64]*User, 16),
    lock:      sync.Mutex{},
  }
}

// nolint
func (db *userDB) Add(u *User) bool {
  db.lock.Lock()
  defer db.lock.Unlock()

  if u.Name == "" || u.Code <= 0 {
    return false
  }

  if !db.existName(u.Name) && !db.existCode(u.Code) {
    return db.AddForName(u) && db.AddForCode(u)
  }

  return false
}

// nolint
func (db *userDB) AddForName(u *User) bool {
  if len(u.Name) == 0 {
    return false
  }

  if _, ok := db.nameIndex[u.Name]; ok {
    return false
```

```
  }

  db.nameIndex[u.Name] = u
  return true
}

// nolint
func (db *userDB) AddForCode(u *User) bool {
  if u.Code <= 0 {
    return false
  }

  if _, ok := db.codeIndex[u.Code]; ok {
    return false
  }

  db.codeIndex[u.Code] = u
  return true
}

// nolint
func (db *userDB) GetByName(n string) (*User, bool) {
  db.lock.Lock()
  defer db.lock.Unlock()

  r, ok := db.nameIndex[n]
  return r, ok
}

// nolint
func (db *userDB) GetByCode(n int64) (*User, bool) {
  db.lock.Lock()
  defer db.lock.Unlock()

  r, ok := db.codeIndex[n]
  return r, ok
}

func (db *userDB) existName(name string) bool {
  if len(name) <= 0 {
    return false
  }

  _, ok := db.nameIndex[name]
  if ok {
    return true
  }
```

```
  return false
}

func (db *userDB) existCode(code int64) bool {
  if code <= 0 {
    return false
  }

  _, ok := db.codeIndex[code]
  if ok {
    return true
  }

  return false
}

// User user obj.
type User struct {
  ID   string    `json:"id,omitempty"`
  Code int64     `json:"code,omitempty"`
  Name string    `json:"name,omitempty"`
  Age  int32     `json:"age,omitempty"`
  Time time.Time `json:"time,omitempty"`
}

// UserProvider the dubbo provider.
// like: version: 1.0.0 group: test
type UserProvider struct{}

// CreateUser new user, PX config POST.
func (u *UserProvider) CreateUser(ctx context.Context, user *User) (*Use
r, error) {
  outLn("Req CreateUser data:%#v", user)
  if user == nil {
    return nil, errors.New("not found")
  }
  _, ok := cache.GetByName(user.Name)
  if ok {
    return nil, errors.New("data is exist")
  }

  b := cache.Add(user)
  if b {
    return user, nil
  }

  return nil, errors.New("add error")
}
```

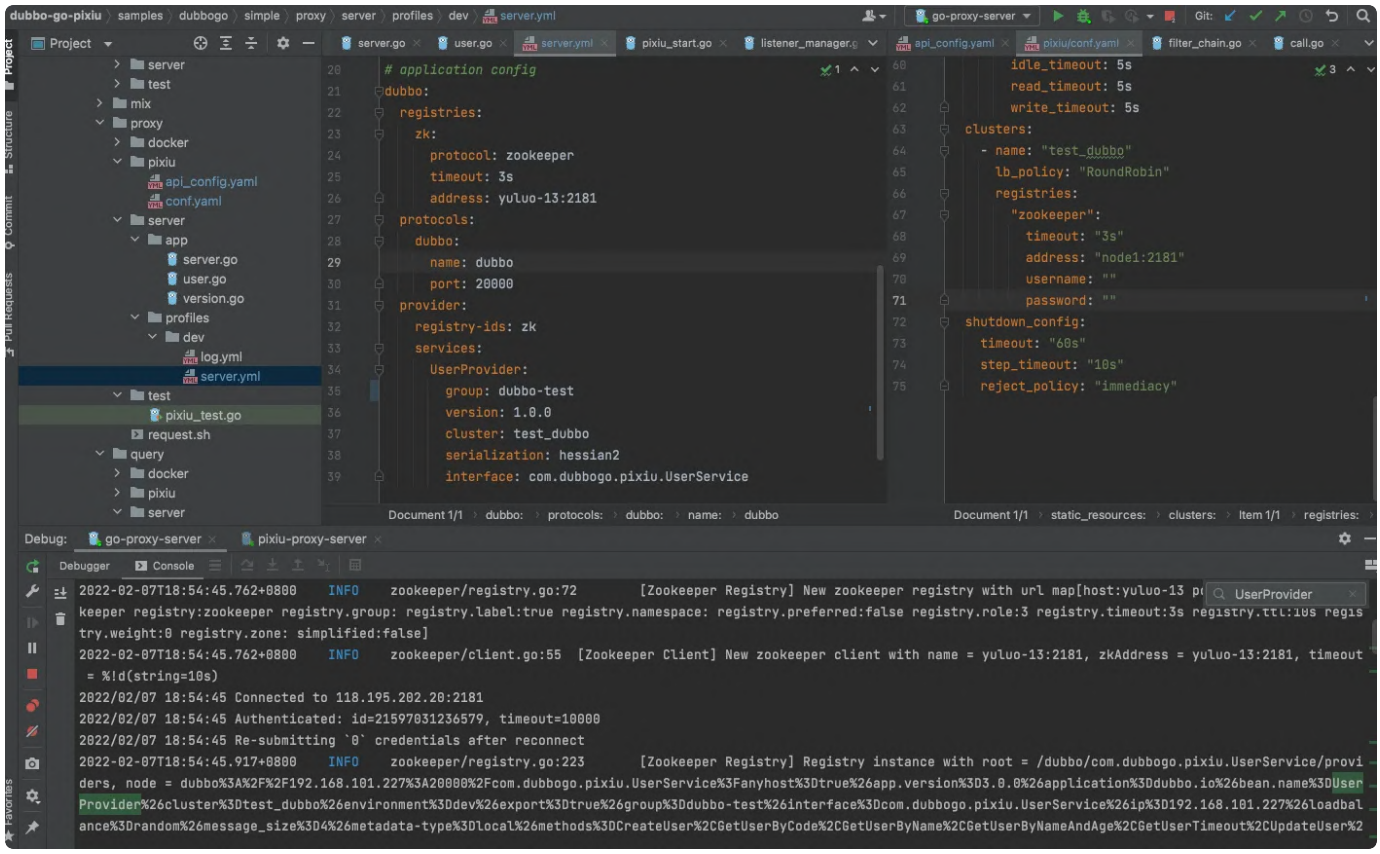
```
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
```

```
// GetUserByName query by name, single param, PX config GET.
func (u *UserProvider) GetUserByName(ctx context.Context, name string) (*User, error) {
  outLn("Req GetUserByName name:%#v", name)
  r, ok := cache.GetByName(name)
  if ok {
    outLn("Req GetUserByName result:%#v", r)
    return r, nil
  }
  return nil, nil
}

// GetUserByCode query by code, single param, PX config GET.
func (u *UserProvider) GetUserByCode(ctx context.Context, code int64) (*User, error) {
  outLn("Req GetUserByCode name:%#v", code)
  r, ok := cache.GetByCode(code)
  if ok {
    outLn("Req GetUserByCode result:%#v", r)
    return r, nil
  }
  return nil, nil
}

// GetUserTimeout query by name, will timeout for pixiu.
func (u *UserProvider) GetUserTimeout(ctx context.Context, name string) (*User, error) {
  outLn("Req GetUserByName name:%#v", name)
  // sleep 10s, pixiu config less than 10s.
  time.Sleep(10 * time.Second)
  r, ok := cache.GetByName(name)
  if ok {
    outLn("Req GetUserByName result:%#v", r)
    return r, nil
  }
  return nil, nil
}

// GetUserByNameAndAge query by name and age, two params, PX config GET.
func (u *UserProvider) GetUserByNameAndAge(ctx context.Context, name string, age int32) (*User, error) {
  outLn("Req GetUserByNameAndAge name:%s, age:%d", name, age)
  r, ok := cache.GetByName(name)
  if ok && r.Age == age {
    outLn("Req GetUserByNameAndAge result:%#v", r)
    return r, nil
  }
```

```
  return r, nil
}

// UpdateUser update by user struct, my be another struct, PX config POST or PUT.
func (u *UserProvider) UpdateUser(ctx context.Context, user *User) (bool, error) {
  outLn("Req UpdateUser data:%#v", user)
  r, ok := cache.GetByName(user.Name)
  if ok {
    if user.ID != "" {
      r.ID = user.ID
    }
    if user.Age >= 0 {
      r.Age = user.Age
    }
    return true, nil
  }
  return false, errors.New("not found")
}

// UpdateUserByName update by user struct, my be another struct, PX config POST or PUT.
func (u *UserProvider) UpdateUserByName(ctx context.Context, name string, user *User) (bool, error) {
  outLn("Req UpdateUserByName data:%#v", user)
  r, ok := cache.GetByName(name)
  if ok {
    if user.ID != "" {
      r.ID = user.ID
    }
    if user.Age >= 0 {
      r.Age = user.Age
    }
    return true, nil
  }
  return false, errors.New("not found")
}

// nolint
func (u *UserProvider) Reference() string {
  return "UserProvider"
}

// nolint
func (u User) JavaClassName() string {
  return "com.dubbogo.pixiu.UserService"
}
```

```
275
276
277   // nolint
278   func outLn(format string, args ...interface{}) {
279     fmt.Printf("\033[32;40m"+format+"\033[0m\n", args...)
280   }
```

启动后日志输出如下:



另外启动一个demo工程，新建单元测试如下

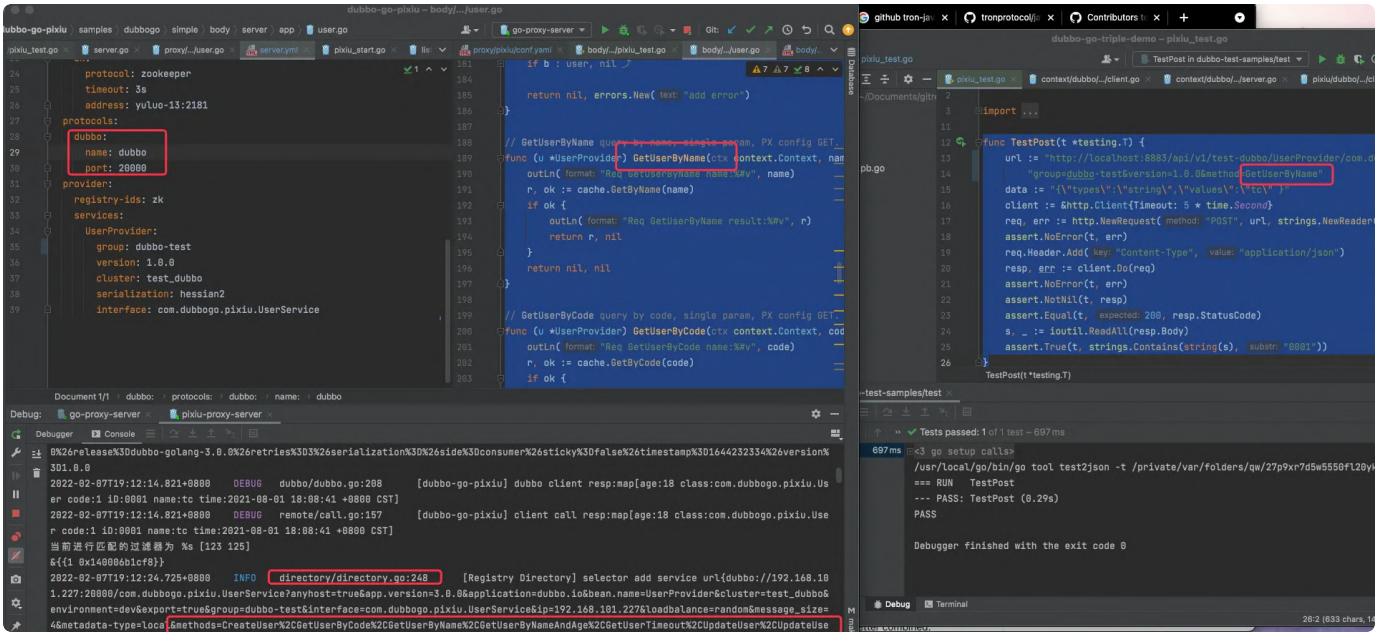
```
Plain Text
```

```
func TestPost(t *testing.T) {
  url := "http://localhost:8883/api/v1/test-dubbo/UserProvider/com.dubbogo.pixiu.UserService?" +
    "group=dubbo-test&version=1.0.0&method=GetUserByName"
  data := "{\"types\":\"string\",\"values\":\"tc\" }"
  client := &http.Client{Timeout: 5 * time.Second}
  req, err := http.NewRequest("POST", url, strings.NewReader(data))
  assert.NoError(t, err)
  req.Header.Add("Content-Type", "application/json")
  resp, err := client.Do(req)
  assert.NoError(t, err)
  assert.NotNil(t, resp)
  assert.Equal(t, 200, resp.StatusCode)
  s, _ := ioutil.ReadAll(resp.Body)
  assert.True(t, strings.Contains(string(s), "0001"))
}
```

执行效果图如下：

红框代表user.go文件中注册provider实例的全部接口，都注册到了Directory服务接口列表当中

黄框代表这一次http请求所需要调用的方法，由于在server.yml中配置为dubbo协议.所以在api_config.yaml配置文件中需要配置转入协议inboundRequest为http,以及转出协议为integrationRequest为dubbo



2.2 启动

cmd/pixiu/pixiu.go 文件是主要启动的入口，获取命令行参数，然后 pixiu 目前有 sidecar 和 gateway 两个字命令，目前只有 gateway 可用，详见 gateway.go。

gateway.go 中 startGatewayCmd 中 initApiConfig 首先读取配置文件，然后 server.Start(bootstrap) 来调用 server 进行启动。

pixiu_start.go 中有 server 启动的相关代码。主要分为 initialize 和 Start 两个阶段，会包含各类资源的 manager 的初始化，如：ListenerManager，ClusterManager， AdapterManager 和 RouterManager 等。

下面，我们以最为复杂的 ListenerManager 为例进行讲解，因为它也涉及了 Networkfilter 和 Filter 的初始化。

CreateDefaultListenerManager 是创建LisnterManager 的函数，为配置文件中的每一个 Listner 调用 CreateListenerService 创建一个 ListenerService。ListenerService 有多个类型的实现，分别是 http， http2，和 tcp，详情可以查看 listener 文件夹。

server 的 start 函数则会调用 ListenerManager 的 StartListen 函数，让每个 ListenerService 进行各自的启动监听操作。我们以 HttpListenerService 为例子，进行后续介绍。其 start函数中区分了 http 和 https 两种情况，分别启动了对应的http.Server，并进行监听。

在 HttpListenerService 初始化时，会进行其下 NetworkFilterChain，具体可看 filterchain.CreateNetworkFilterChain。在NetworFilter中，我们以 HttpConnectionManager 为例，进行介绍。

CreateHttpConnectionManager 用以创建 HttpConnectionManager，其中会创建 filterManager 和 routerCoordinator。并调用 Load 函数进行 Filter 的初始化。

Filter 的初始化依赖于 Plugin 机制。具体定义详见 common.extension.filter中对于 HttpFilterPlugin ， HttpFilterFactory ， HttpDecodeFilter 和 HttpEncodeFilter ` 的定义。

插件的各个函数及其初始化过程需要详细讲解，具体见 FilterManager的 Load 函数 @梦超

需要注意自定义插件需要在 `pluginregistry` 中进行import。

2.3 请求处理

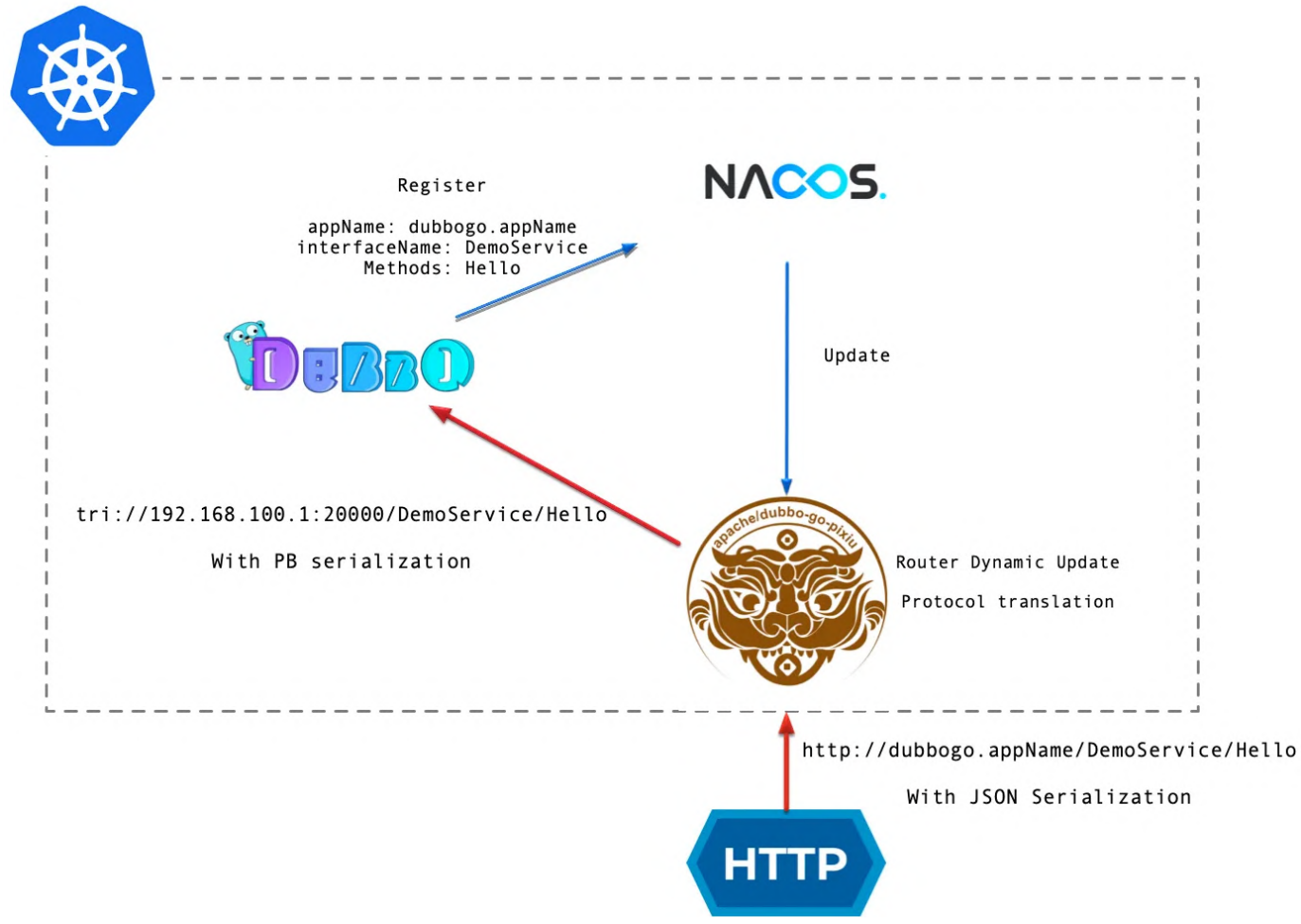
以 http 为例，首先是 http_listener 的 ServerHttp 函数，它直接将 http 请求的 request 和 response 交给 NetworkFilterChain 处理，然后按照配置，chain 中第一个为 HttpConnectionManager。则交由其 ServeHttp 函数处理。

在 HttpConnectionManager 的 ServeHttp 函数中，将 request 和 response 封装成 HttpContext，交由 Handle 函数处理。

Handle 函数首先调用 findRoute 来获取路由信息，然后交给其 handleHTTPRequest 函数处理。分别由 filterchain 的 onDecode 和 onEncode 分别在pre 和 post 两个阶段进行处理。

后续涉及到具体 Filter 的实现，可以查看 filter 包下的各种 filter 实现。

2.4 Adapter 逻辑



如上图，Adapter的作用在于当用户设置不同的组件作为配置中心(nacos,zk)，可以通过Adapter动态抓取实例信息推送到pixiu的gateway,目前支持两种类型:

- dgp.adapter.springcloud

  用户使用springcloud作为配置中心,动态推送到pixiu gateway

- dgp.adapter.dubboregistrycenter

  用户使用zk,nacos等其他一些注册中心

在conf.yml当中配置如下

Plain Text

```
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
---
static_resources:
  listeners:
    - name: "net/http"
      protocol_type: "http"
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8883
      filter_chains:
        filters:
          - name: dgp.filter.httpconnectionmanager
            config:
              route_config:
                routes:
                  - match:
                      prefix: "/api/v1"
              http_filters:
                - name: dgp.filter.http.apiconfig
                  config:
                    path: ~/simple/proxy/pixiu/api_config.yaml
                - name: dgp.filter.http.dubboproxy
                  config:
                    dubboProxyConfig:
                      registries:
                        "zookeeper":
                          protocol: "zookeeper"
```

```
                            timeout: "3s"
                            address: "node1:2181" //需要用户自己配置zk
                            username: ""
                            password: ""
                        timeout_config:
                          connect_timeout: 5s
                          request_timeout: 5s
                server_name: "test_http_dubbo"
                generate_request_id: false
        config:
          idle_timeout: 5s
          read_timeout: 5s
          write_timeout: 5s

    #配置dubbogo的负载均衡策略
    clusters:
      - name: "test-grpc"
        lb_policy: "RoundRobin"
        endpoints:
          - socket_address:
              address: 127.0.0.1
              port: 50001
              protocol_type: "GRPC"
    adapters:
      - id: "zookeeper"
        name: "dgp.adapter.dubboregistrycenter"
        config:
          freshInterval: 60s
          registry:
            name: zookeeper
            protocol: zookeeper
            timeout: 3s
            address: node1:2181
```

三、代码工具

3.1 import 拆分

社区推荐代码的 import 块按照所属进行拆分，有一个工具可以自动将import拆分

imports-formatter -path . -module github.com/apache/dubbo-go-pixiu -bl false

3.2 证书检查

社区代码必须有 apache的 license，可以通过下列命令自动判断检查或增加license
https://github.com/apache/skywalking-eyes

3.3 ci 相关命令

排版检查
go fmt ./... && git status && [[-z `git status -s`]]

golint 静态检查
GO111MODULE=on golangci-lint run --timeout=10m -v --disable-all --enable=govet --enable=staticcheck --enable=ineffassign --enable=misspell

单测
go mod vendor && go test ./pkg/... -coverprofile=coverage.txt -covermode=atomic

集成测试

chmod +x start_integrate_test.sh && chmod +x integrate_test.sh && ./start_integrate_test.sh

Dubbo-go-pixiu性能测试

机器配置：

芯片: Apple M1

核数:8 core

内存 16GB

系统盘 560G

根据开发手册进行部署环境

https://dubbogoproxy.yuque.com/docs/share/e7aca6d0-1957-4f85-af4e-306104c2bbe4?#

性能产出报告需要参照: https://www.jianshu.com/p/b6a588affd96

以下测试基于http->dubbo进行协议转换

- 配置lua脚本

Plain Text

```
local counter = 1
local threads  = {}

--local json = require("json")

local req  = {

    --group = "dubbo-test",
    --version = "1.0.0",
    --method= "GetUserByName",
    types = "types",
    values = "tccccccccc"
}

function setup(thread)
    -- 给每个线程设置一个id参数
    thread:set("id",counter)
    table.insert(threads,thread)
    counter = counter + 1

end

function init(args)
    -- 初始化两个参数 每个线程都有独立的 requests, response 参数
    requests = 0
    response = 0
    -- 打印线程被创建的消息，打印完后，线程正式开始运行
    local msg = " thread %d created"
    print(msg.format(counter))
end

function request()

    wrk.headers["Content-Type"] = "application/x-www-form-urlencoded; char
set=UTF-8"
    wrk.headers["User-Agent"] = "wrk"
    wrk.headers["Connection"] = "keep-alive"
    wrk.body = "{\"types\":\"string\",\"values\":\"tc\" }"

    print("本次请求包体为: %s",wrk.body)
    return wrk.format("POST",wrk.path,wrk.headers,wrk.body)
end

```

```
function response(status,headers,body)
    if status ~= 200 then
        print(body)
    return
    end

    -- 打印响应体
    local resp = json.decode(body)
    print(json.encode(body)..'-->'..body)

end


--function done(summary,latency,requests)
--    print("99 latency:"..latency:percentile(99.0))
--    for index,thread in ipairs(threads) do
--        local id = thread:get("id")
--        local requests = thread:get("requests")
--        local responses = thread:get("responses")
--        -- 打印每个线程发起了多少请求 得到了多少响应
--        print(wrk:format(id,requests,responses))
--    end
--    print(latency)
--end
```

性能测试部分: (consumer请求是否经过pixiu网关)

不经过pixiu网关的测试方式: 采用dubbo-go-benchmark的测试方式
https://github.com/dubbogo/dubbo-go-benchmark/tree/feat/adasvc/3.0/adaptivesvc
经过pixu网关的测试方式: 启动一个httpserver注册provider,再启动pixiu gateway设置路由规则拦截consumer请求

- 对照组1:在不同连接数下的性能测试
-

压力测试部分:
分别在不同运行时间，不同连接数，不同线程数，不同包体的情况下做测试

- 对照组1: 在不同连接数下的性能测试

case1:
运行时间: 3s　连接数: 10　线程数: 1 包体: 33字节

Plain Text

```
Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     9.41ms   26.34ms 214.80ms   95.53%
    Req/Sec     2.76k   684.49     3.55k    79.31%
  Latency Distribution
     50%    2.88ms
     75%    6.28ms
     90%   11.55ms
     99%  166.03ms
  8038 requests in 3.01s, 1.53MB read
Requests/sec:   2671.24
Transfer/sec:    521.73KB
```

结果:　每个线程的平均延迟在166.03ms, 每秒请求数 2.76k个/s，延迟分布在t99的 主要延迟时长是166.03ms

接口对总体请求的平均处理时长为 2671.24 /s

case2:

运行时间: 3s　连接数: 20　线程数: 1 包体: 33字节

```
Thread Stats   Avg      Stdev     Max   +/- Stdev
              (平均值)    (标准差)  (最大值)    (正负一个标准差所占比例)
    Latency   285.15ms  152.23us 285.28ms   66.67%
    (延迟)
    Req/Sec     9.00      0.00     9.00    100.00%
   (每秒请求数 tps)
  Latency Distribution
       (延迟分布)
     50%  285.18ms
     75%  285.28ms
     90%  285.28ms
     99%  285.28ms
  3 requests in 3.09s, 600.00B read  (3.09s内处理了3个请求，耗费流量600.00B)
Requests/sec:      0.97     (QPS 0.97 即平均每秒数处理请求数0.97  )
Transfer/sec:     194.20B   (平均每秒流量194.20B )
```

结果： 每个线程的平均延迟在285.15ms, 每秒请求数 9个/s，延迟分布在t99的 主要延迟时长是285.28ms

接口对总体请求的平均处理时长为 0.97 /s

对照结论: 在连接数增加的情况下平均延迟增大,每秒接受的请求数变少，平均处理时长降低

- 对照组2: 在不同线程下的性能测试:

case1:

运行时间: 3s 连接数: 20 线程数: 1 包体: 33字节

```
Thread Stats   Avg      Stdev     Max   +/- Stdev
              (平均值)    (标准差)  (最大值)    (正负一个标准差所占比例)
    Latency   285.15ms  152.23us 285.28ms   66.67%
    (延迟)
    Req/Sec     9.00      0.00     9.00    100.00%
   (每秒请求数 tps)
  Latency Distribution
       (延迟分布)
     50%  285.18ms
     75%  285.28ms
     90%  285.28ms
     99%  285.28ms
  3 requests in 3.09s, 600.00B read  (3.09s内处理了3个请求，耗费流量600.00B)
Requests/sec:      0.97     (QPS 0.97 即平均每秒数处理请求数0.97  )
Transfer/sec:     194.20B   (平均每秒流量194.20B )
```

结果： 每个线程的平均延迟在285.15ms, 每秒请求数 9个/s，延迟分布在t99的 主要延迟时长是285.28ms

接口对总体请求的平均处理时长为 0.97 /s

case2:

运行时间: 3s　连接数: 20　线程数: 5 包体: 33字节 (不稳定 pixiu网关很长一段时间连不到注册中心)

```
Thread Stats   Avg     Stdev     Max   +/- Stdev
               (平均值) (标准差) (最大值)  (正负一个标准差所占比例)
    Latency     6.81ms    5.90ms  58.48ms   83.17%
    (延迟)
    Req/Sec   582.34    127.02     0.95k    74.12%
(每秒请求数 tps)
  Latency Distribution
       (延迟分布)
     50%    4.91ms
     75%    9.68ms
     90%   14.63ms
     99%   27.15ms
  4941 requests in 3.05s, 0.94MB read  (3.05s内处理了4941个请求，耗费流量0.94MB)
Requests/sec:   1621.19 (QPS 163.76即平均每秒数处理请求数163.76 )
Transfer/sec:    316.64KB (平均每秒流量316.64KB)
```

结果： 每个线程的平均延迟在6.81ms，每秒平均请求数 582.34个/s，延迟分布在t99的 主要延迟时长是27.15ms

接口对总体请求的平均处理时长为 1621.19

对照结论：在仅提升处理线程数的情况下，每个线程的平均延迟降低，每秒平均请求数提升.t99的延迟时长降低

- 对照组3: 在不同运行时间的情况下测试

case1:
运行时间: 3s 连接数: 20 线程数: 5 包体: 33字节 (不稳定 pixiu网关很长一段时间连不到注册中心)

```
Thread Stats   Avg     Stdev     Max   +/- Stdev
               (平均值) (标准差) (最大值)  (正负一个标准差所占比例)
    Latency     6.81ms    5.90ms  58.48ms   83.17%
    (延迟)
    Req/Sec   582.34    127.02     0.95k    74.12%
(每秒请求数 tps)
  Latency Distribution
       (延迟分布)
     50%    4.91ms
     75%    9.68ms
     90%   14.63ms
     99%   27.15ms
  4941 requests in 3.05s, 0.94MB read  (3.05s内处理了4941个请求，耗费流量0.94MB)
Requests/sec:   1621.19 (QPS 163.76即平均每秒数处理请求数163.76 )
Transfer/sec:    316.64KB (平均每秒流量316.64KB)
```

结果： 每个线程的平均延迟在6.81ms，每秒平均请求数 582.34个/s，延迟分布在t99的 主要延迟时长是27.15ms

接口对总体请求的平均处理时长为 1621.19

case2:
运行时间: 60s 连接数: 20 线程数: 5 包体: 33字节 (不稳定 pixiu网关很长一段时间连不到注册中心)

```
wrk -d60s -c20 -t5 --latency  -s test.lua "http://localhost:8883/api/v1/test-dubbo/UserProvider/com.dubbogo.pixiu.UserService?group=dubbo-test&version=1.0.0&method=GetUserByName"
```

```
Plain Text
```

```
Thread Stats   Avg      Stdev     Max   +/- Stdev
              （平均值）  （标准差）（最大值）  （正负一个标准差所占比例）
    Latency     6.29ms    5.38ms  53.31ms   81.72%
      （延迟）
    Req/Sec   618.23    142.44     0.91k    70.00%
 （每秒请求数 tps）
  Latency Distribution
      （延迟分布）
     50%    4.45ms
     75%    8.94ms
     90%   13.46ms
     99%   23.92ms
  6172 requests in 1.00m, 1.18MB read（一分钟内处理了6172个请求，耗费流量1.18MB）
Requests/sec:    102.74  （QPS 102.74即平均每秒数处理请求数102.74 ）
Transfer/sec:     20.07KB  （平均每秒流量20.07KB ）
```

结果： 每个线程的平均延迟在6.29ms，每秒平均请求数 618.23个/s，延迟分布在t99的 主要延迟时长是23.92ms

接口对总体请求的平均处理时长为 102.74

对照结论：在不同运行时长的情况下，运行时间越久，平均处理的请求数越多，接口对总体请求的平均处理时长越短，其余参数没有变化

- 对照组4: 在不同请求包体的情况下测试

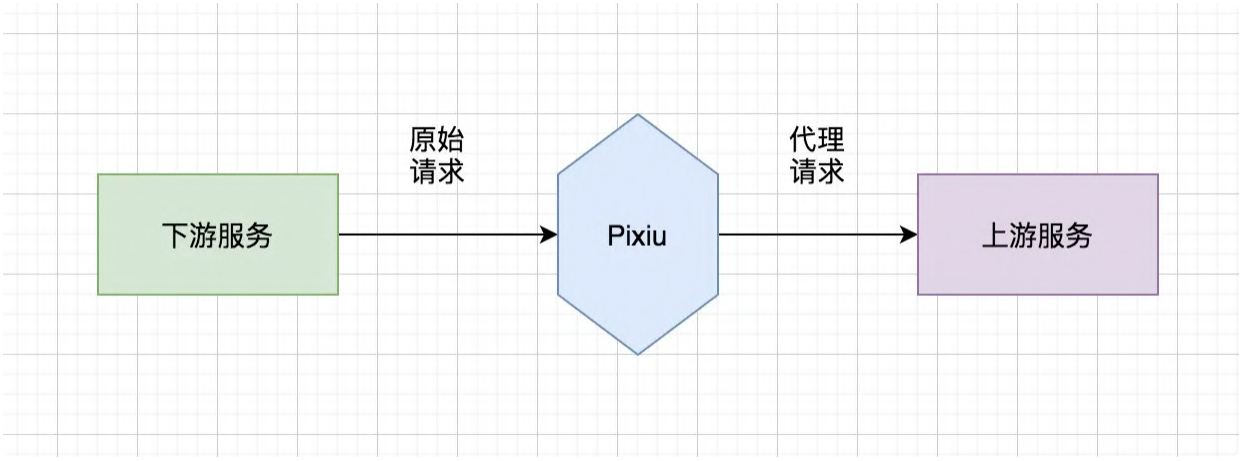
dubbo多协议转换机制

一、背景

背景是这样的，用户想上云或者跨环境互通，他们的已有服务或云上服务已经是 grpc 的，但是迁移过程很困难，涉及依赖的应用很多，没办法统一先迁移 provider 到 grpc 协议，所以需要先通过协议转换让云上的能互通

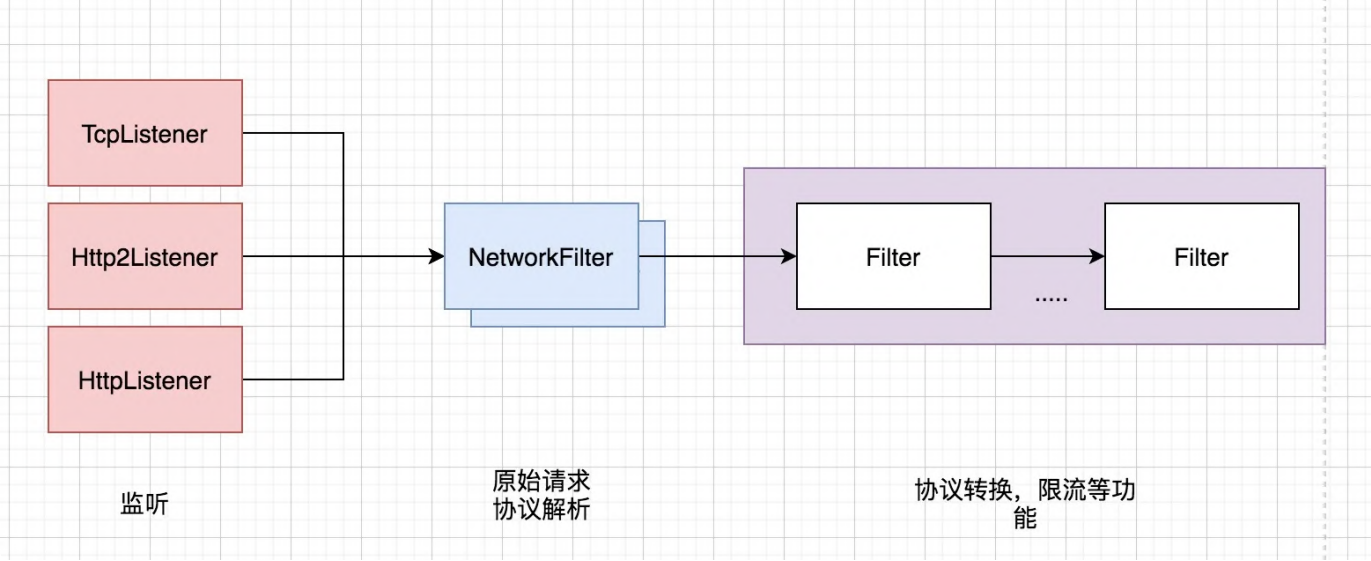
client (grpc+ wrapper + dubbo 泛化) --> 网关 --> upstream (dubbo)
client(dubbo 泛化) -> 网关 -> upstream(grpc)

此外，希望有一套机制来抽象整个协议转换过程，方便后续扩展



二、机制

利用 Pixiu 原有的 Listener, NetworkFilter 和 Filter(原HttpFilter) 三层抽象，分别对应网络监听，原始请求解析和代理请求转换能力。



比如说，目前 Pixiu 的 Listener 支持 http、http2和tcp协议。其中(名称是代指，未和源码一一对应)：

- HttpListener 用于提供 HTTP 网络监听能力，和 HttpConnectionManager 搭配，选择不同的 UpStreamFilter(原HttpFilter)可以进行不同的协议转换：
  - dubbo2Filter：将原始 http 请求转换成代理 dubbo2 协议请求；
  - tripleFilter: 将原始 http 请求转换为代理 triple hessian序列化的协议请求
  - grpcFilter: 将原始 http 请求转换为代理 grpc 请求；
  - httpFilter: 将原始 http 请求进行代理 http 请求，
- Http2Listener 用于提供 HTTP2 网络监听能力，和 GrpcConnectionManager 进行配合，转发对应的请求，从而实现 grpc 请求代理；
- Http2Listener 和 TripleConnectionManager 进行配合(待开发)：
  - dubbo2Filter：将原始的 triple 请求转换为代理 dubbo2 请求；
- TcpListener 用于提供 TCP 网络监听能力，和 Dubbo2ConnectionManager 进行配合，可以解析出 Dubbo2 协议请求，后续提供多种 UpStreamFilter 进行协议转换：
  - tripleFilter: 将原始 dubbo2请求转换为 triple hessian 序列化请求(待开发)；
  - httpFIlter：将原始 dubbo2 请求按照默认转换规则转换为 http 请求

三、需要沟通

- 网关模式 or sidecard模式
- 路由场景
- 服务发现模式

四、实现问题

不同协议栈是使用在多个 Filter 进行传递的抽象结构无法进行通用定义。
比如说 HttpConnectionManager 是处理和传递 request, response，而 Tcp + dubboConnectionManager 是处理和传递 RPCInvocation 和 RPCResult。
所以 http to dubbo 和 triple to dubbo，需要将不同结构转换为 dubbo 请求
而 dubbo to http 和 dubbo to triple 则需要将 dubbo 请求转换为不同结构体

需要一种多通道或者转接口适配机制

- dubbo 多个版本，要切两个分支?
- pixiu 转发的路由如何获取，要对接内部的注册中心?
- 性能损耗测试

Pixiu & gRPC：基于 Reflection 调用技术方案

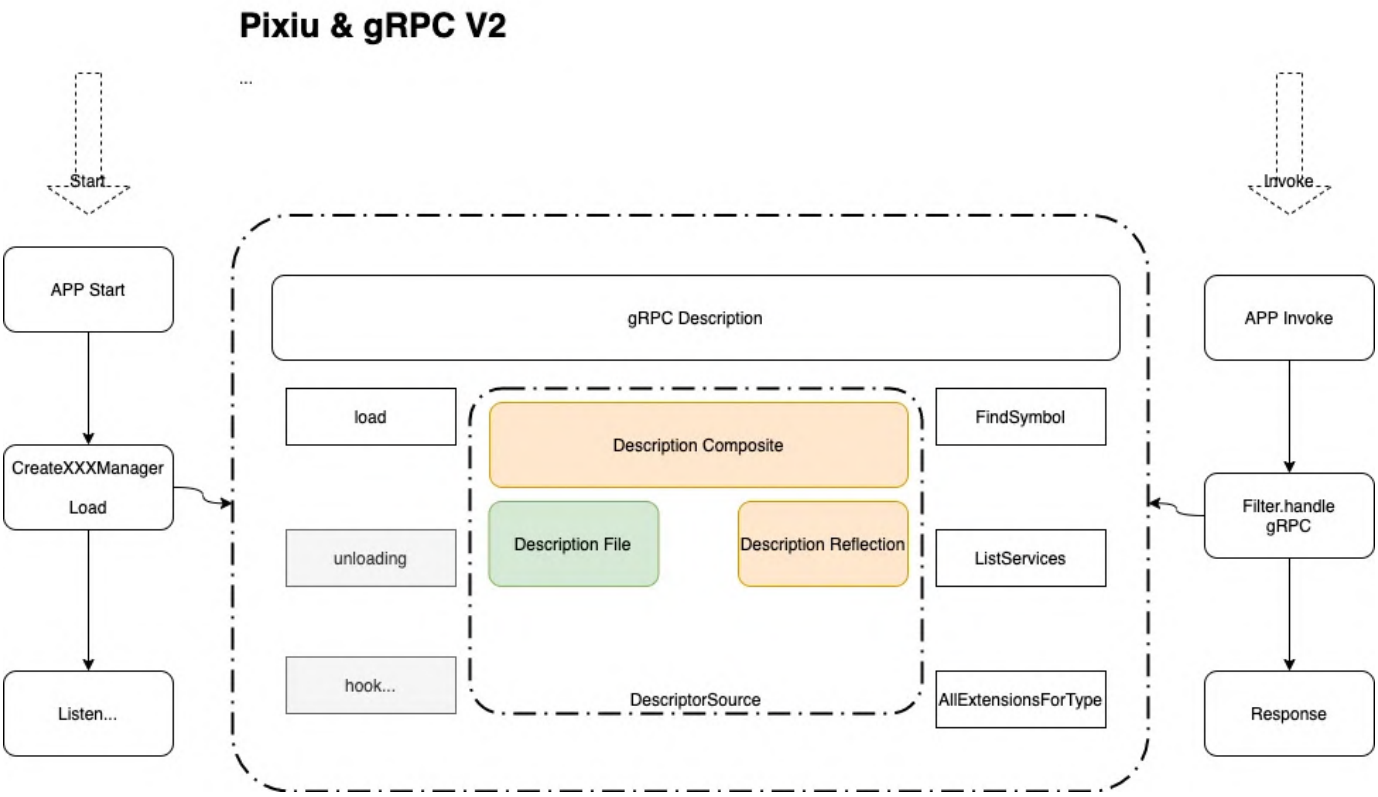
概述

背景&目的

一期由 蔡俊铭 实现了 Pixiu（HTTP）调用 gRPC能力，基于本地 proto 文件实现 Pixiu & gRPC 的通信。

本期继续迭代，基于 **GRPC Server Reflection Protocol** 实现动态获取gRPC对象的Proto文件。

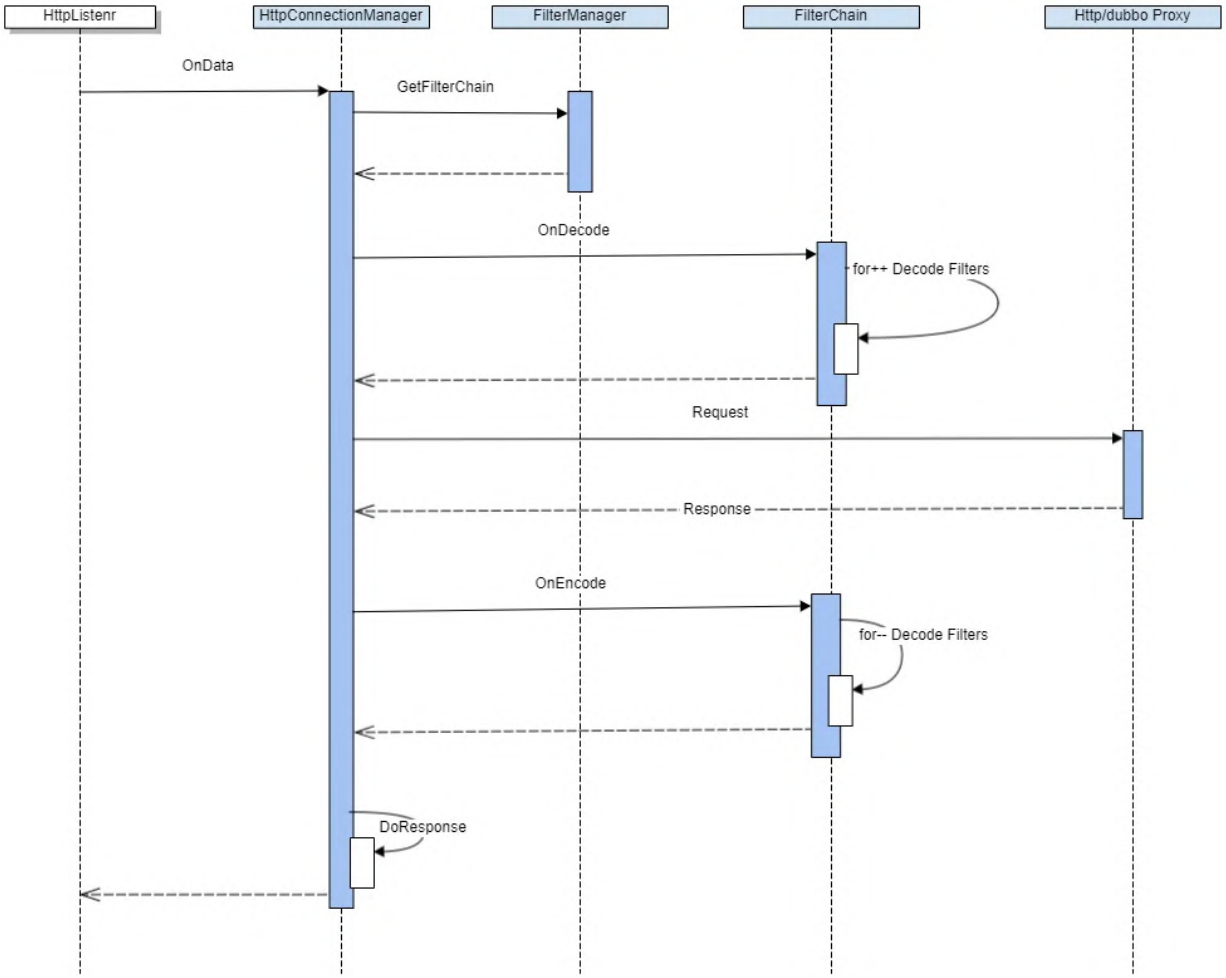
方案



Ref

- 将 HTTP/JSON 转码为 gRPC https://cloud.google.com/endpoints/docs/grpc/transcoding
- 根据配置加载proto文件到内存中形成参考：
  https://github.com/fullstorydev/grpcurl/blob/master/desc_source.go 的
  DescriptorSourceFromProtoFiles函数
- 发起调用参考：
  https://github.com/fullstorydev/grpcurl/blob/cd242fe1ed8a8b34cf71cb84c452864a8cfd55a5/invoke.go 的InvokeRPC 函数

Http Connection Manager & Filter Chain



Go

```go
func (hcm *HttpConnectionManager) handleHTTPRequest(c *pch.HttpContext) {
    chain := filter.Chain{}
    chain.OnEncode(c)
    chain.OnDecode(c)
    onRespone(c)
    // TODO redirect
}
```

Go

```go
package filter

import (
    "github.com/apache/dubbo-go-pixiu/pkg/context/http"
)

type HttpEncodeFilter interface {
    Encode(*http.HttpContext) FilterStatus
}

type HttpDecodeFilter interface {
    Decode(*http.HttpContext) FilterStatus
}

type chain struct {
    decodeFilters      []HttpDecodeFilter
    decodeFiltersIndex int

    encodeFilters      []HttpEncodeFilter
    encodeFiltersIndex int
}

func (c *chain) appendDecodeFilters(f ...HttpDecodeFilter) {
    c.decodeFilters = append(c.decodeFilters, f...)
}

func (c *chain) appendEncodeFilters(f ...HttpEncodeFilter) {
    //append encode filters in reverse order
    for i := len(f) - 1; i >= 0; i-- {
        c.encodeFilters = append(c.encodeFilters, f[i])
    }
}

func (c *chain) Decode(ctx *http.HttpContext) {
    for c.decodeFiltersIndex < len(c.decodeFilters) {
        filterStatus := c.decodeFilters[c.decodeFiltersIndex].Decode(ctx)
        if filterStatus == Continue {
            continue
        }
        return
    }
}

func (c *chain) Encode(ctx *http.HttpContext) {
    for c.encodeFiltersIndex < len(c.encodeFilters) {
```

```
            filterStatus := c.encodeFilters[c.encodeFiltersIndex].Encode(ctx)
            if filterStatus == Continue {
                continue
            }
            return
        }
    }
```

```
```

调研中|Pixiu: Dubbo与SpringCloud互通方案

前言

标题即主题，该主题“Pixiu: Dubbo与SpringCloud互通”是 天哥 调研后发起的 Pixiu 下一个阶段的产品功能方向，并提供了一些参考资料方便调研。

背景

当前 Pixiu 已经支持多种 RPC 协议，HTTP to Dubbo，HTTP to SpringCloud， 但市面上还有一些诉求，希望 Dubbo 服务 与 SpringCloud 服务之间互通，而当前很少有产品可以满足.....

目的

Pixiu 实现 Dubbo 与 SpringCloud 服务互通：

- 调用过程对用户透明
- 与 Pixiu 其他网关能力集成
- 最小接入成本

概述

功能说明

跨注册中心，满足哪些通信场景?

- [x] Dubbo to SpringCloud
- [ ] ~~Dubbo to Dubbo~~
- [x] SpringCloud to Dubbo
- [ ] ~~SpringCloud to SpringCloud~~

Dubbo-SpringCloud 通信方案（完善中）

Dubbo-go 与 SpringCloud 互通:

- 方案简述
  - Pixiu 通过 Mock 将跨注册中心的 Dubbo 或 SpringCloud 应用服务信息注册到当前需要互通的注册中心
  - Dubbo 与 SpringCloud 之间通过各自的注册中心，正常订阅消费服务，真实请求由 Pixiu 代理，该过程对服务提供者与消费者完全透明
- 应用级服务注册与发现
  - Pixiu 支持多注册中心
  - Pixiu Mock SpringCloud 应用支持 Dubbo 协议
  - Pixiu Mock Dubbo 应用支持 SpringCloud 协议
  - 集成当前 Pixiu Filter、Router 等公共配置

https://www.processon.com/view/link/617be10f0e3e7416bdf22ee2

问题

通信协议设计

Dubbo 协议

基于 dubbo3.0

SpringCloud

todo

Pixiu 互通协议

与 Pixiu 现有能力集成

todo

其他

Ref

- 成本直降50% | 阿里云发布云原生网关，开启下一代网关新进程

鉴权

JWT

目标：Pixiu仅负责通过配置好的**公钥JWK认证JWT， 支持local/remote两种方式配置JWKS**

参考组件： https://github.com/golang-jwt/jwt

https://mkjwk.org/?spm=a2c4g.11186623.0.0.2aed20e4YpLZqa

路由

提取

验证

解析

一个JWKS公钥的例子

JSON

```
{"keys":
  [
    {"kty":"EC",
     "crv":"P-256",
     "x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
     "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
     "use":"enc",
     "kid":"1"},
    {"kty":"RSA",
     "n": "lalallalaalalalalla",
     "e":"AQAB",
     "alg":"RS256",
     "kid":"2011-04-29"}
  ]
}
```

```
```

YAML

```yaml
filter_chains:
  - filter_chain_match:
    domains:
      - api.dubbo.com
      - api.pixiu.com
    filters:
      - name: dgp.filter.httpconnectionmanager
        config:
          route_config:
            dynamic: true
            dynamic_apdter: "springcloud"
            routes:
          http_filters:
            - name: dgp.filter.http.auth.jwt
              config:
                # 配置路由匹配规则
                rules:
                  # Not jwt verification is required for /health path
                  - match:
                      prefix: /health
                  # Jwt verification for provider1 is required for path pr
efixed with "prefix"
                  - match:
                      prefix: /prefix
                    #指定路由对应的providers
                    requires:
                      requires_any:
                        requirements:
                          - provider_name: provider1
                          - provider_name: provider2
                  - match:
                      prefix: /prefix
                    requires:
                      requires_all:
                        requirements:
                          - provider_name: provider1
                          - provider_name: provider2
                #配置providers
                providers:
                  provider1:
                    #可选的,如不为空，则必须与JWT的iss匹配
                    issuer: issuer1
                    #可选的,如不为空，则必须与JWT的aud匹配
                    audiences:
                      - audience1
```

```
                    - audience2
                  #远程fetch jwks第一阶段可不实现
                  remote_jwks:
                    http_uri:
                      uri: https://example.com/.well-known/jwks.json
                      cluster: example_jwks_cluster
                      timeout: 1s
                provider2:
                  issuer: issuer2
                  #从header的某个字段获取token，默认 Authorization: Bearer <token>
                  from_headers:
                    - name: Authorization
                      value_prefix: Bearer
                  forward_payload_header: "user-data"
                  local_jwks:
                    inline_string: "{ \"keys\":[ {\"e\":\"AQAB\",\"kid\":\"DHFbpoIUqrY8t2zpA2qXfCmr5VO5ZEr4RzHU_-envvQ\",\"kty\":\"RSA\",\"n\":\"xAE7eB6qugXyCAG3yhh7pkDkT65pHymX-P7KfIupjf59vsdo91bSP9C8H07pSAGQO1MV_xFj9VswgsCg4R6otmg5PV2He95lZdHtOcU5DXIg_pbhLdKXbi66GlVeK6ABZOUW3WYtnNHD-91gVuoeJT_DwtGGcp4ignkgXfkiEm4sw-4sfb4qdt5oLbyVpmW6x9cfa7vs2WTfURiCrBoUqgBo_-4WTiULmmHSGZHOjzwa8WtrtOQGsAFjIbno85jp6MnGGGZPYZbDAa_b3y5u-YpW7ypZrvD8BgtKVjgtQgZhLAGezMt0ua3DRrWnKqTZ0BJ_EyxOGuHJrLsn00fnMQ\"}]}"
```

DBMesh 实现计划

DBMesh

路线图

1. 数据库单库代理
2. 集成分布式事务使代理层具备 TransactionMesh 能力
3. 集成 Sharding 能力使代理层进化为 DBMesh

完成一阶段工作 todo list

1. 监听端口解析 mysql 协议（优先支持 mysql，适时考虑其他数据库）
2. 代理 mysql 请求到后端 DB，具备基本的数据分析能力，比如接入 apm 系统监控 sql 执行耗时等。

架构抽象

|----Application

|----Listener

|----ProtocalResolver（解析 mysql、pgsql 等协议）

|----Pipeline（责任链模式、监控等）

|----Executor（执行 Sql 语句）

Mysql 协议解析

首先识别出下面的操作

```
mysql.COM_QUERY
mysql.COM_PING
mysql.COM_INIT_DB
mysql.COM_FIELD_LIST
mysql.COM_STMT_PREPARE
mysql.COM_STMT_EXECUTE
mysql.COM_STMT_CLOSE
mysql.COM_STMT_SEND_LONG_DATA
mysql.COM_STMT_RESET
mysql.COM_SET_OPTION
```

Event Mesh 实现调研 && 实现方案

面向服务容器：

- 接口协议：
  Event Mesh 实现了 TCP 和 HTTP 协议的接口
  Dapr 实现了 HTTP 和 gRPC 协议的接口
  （监听端口需要考虑接下来的协议下降到传输层，可以先实现多端口多协议）
  基于 URL 去判断使用特定功能：
  a. POST http://localhost:3500/v1.0/**invoke**/cart/method
  b. POST http://localhost:3500/v1.0/**publish**/shipping/orders
  事件监听功能：使用 websocket / gRPC 双向流 / TCP 长连接
- 支持的存储
  Event Mesh 支持了消息队列、对象存储、IMDG（内存数据网格）
  目前暂时先支持 ~~RocketMQ / Pulsar ?~~ Kafka && RocketMQ
- Connector 接口和 SPI 接口规范
  向容器请求凭证访问第三方 Key Vault （通过SPI实现）
- SDK （第一阶段暂时不实现，考虑多语言支持的难度）
  Dapr 和 Event Mesh 均实现了 SDK 来完成消息的发送和接收

实现问题：
~~基于现有 Router 和 Filter 方案，参考 proxy 的 Filter 实现与 MQ 的互通，但是目前得 HttpConnectionManager不支持对单个 Router 进行 Filter 的配置，想参考 Dapr 根据 URL 对 Filter Chains 进行配置的话会有实现上的问题~~

实现方案（第一阶段）：

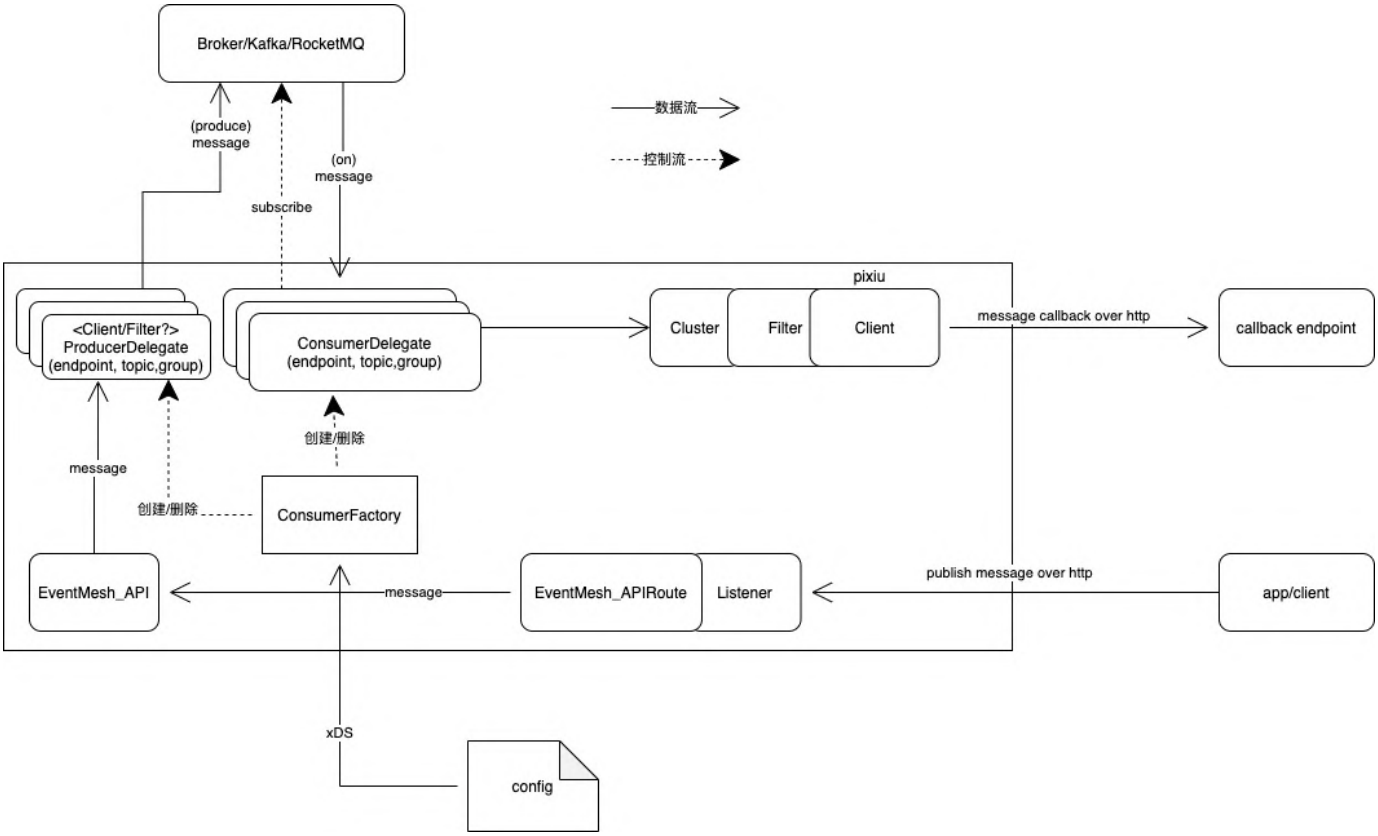
发布：

1. 匹配的请求在 Filter 中发布消息

订阅：

1. 启动时启动一个 Adapter 处理请求发来的订阅请求，请求包括 Topic、消费组 id以及回调 API 地址（消费者端点），维护起 topic => consumer-group => consumer-endporint => 回调的 web hook 的映射关系
2. 通过相关 MQ 的 SDK 去订阅相应的 Topic，然后调用 web hook 发送给真正的消费者进行消费
3. 请求过来通过 FIlter 发送给在后台运行的 Adapter

架构：


Broker/Kafka/RocketMQ
数据流
控制流
(produce)
message
(on)
message
subscribe
pixiu
<Client/Filter?>
ProducerDelegate
(endpoint, topic,group)
ConsumerDelegate
(endpoint, topic,group)
Cluster
Filter
Client
message callback over http
callback endpoint
创建/删除
message
创建/删除
ConsumerFactory
EventMesh_API
message
EventMesh_APIRoute
Listener
publish message over http
app/client
xDS
config

Filter 设计

Go

```go
const (
    // Kind is the kind of Fallback.
    Kind = constant.AccessLogFilter
)

func init() {
    extension.RegisterHttpFilter(&AccessPlugin{})
}

type (
    // AccessPlugin is http filter plugin.
    AccessPlugin struct {
    }
    // AccessFilter is http filter instance
    AccessFilter struct {
        cfg *Config
        alw *model.AccessLogWriter
        alc *model.AccessLogConfig
    }
    // Config describe the config of AccessFilter
    Config struct{}
)

func (ap *AccessPlugin) Kind() string {
    return Kind
}

func (ap *AccessPlugin) CreateFilter(hcm *http2.HttpConnectionManager, config interface{}, bs *model.Bootstrap) (extension.HttpFilter, error) {
    alc := bs.StaticResources.AccessLogConfig
    if !alc.Enable {
        return nil, errors.Errorf("AccessPlugin CreateFilter error the access_log config not enable")
    }

    accessLogWriter := &model.AccessLogWriter{AccessLogDataChan: make(chan model.AccessLogData, constant.LogDataBuffer)}
    specConfig := config.(Config)
    return &AccessFilter{cfg: &specConfig, alw: accessLogWriter, alc: &alc}, nil
}

func (af *AccessFilter) PrepareFilterChain(ctx *http.HttpContext) error {
    ctx.AppendFilterFunc(af.Handle)
    return nil
```

```
}

func (af *AccessFilter) Handle(c *http.HttpContext) {
    start := time.Now()

    c.Next()

    latency := time.Now().Sub(start)
    // build access_log message
    accessLogMsg := buildAccessLogMsg(c, latency)
    if len(accessLogMsg) > 0 {
        af.alw.Writer(model.AccessLogData{AccessLogConfig: *af.alc, AccessLogMsg: accessLogMsg})
    }
}
```

Go

```go
func Init() {
    manager.RegisterFilterFactory(constant.AccessLogFilter, newAccessLog)
}

type accessLog struct {
    conf *AccessLogConfig
    alw  *AccessLogWriter
}

func newAccessLog() filter.Factory {
    return &accessLog{
        conf: &AccessLogConfig{},
        alw: &AccessLogWriter{
            AccessLogDataChan: make(chan AccessLogData, constant.LogDataBuffer),
        },
    }
}

func (a *accessLog) Config() interface{} {
    return a.conf
}

func (a *accessLog) Apply() (filter.Filter, error) {
    // init
    a.alw.Write()

    return accessLogFunc(a.alw, a.conf), nil
}

// access log filter
func accessLogFunc(alw *AccessLogWriter, alc *AccessLogConfig) filter.Filter {
    return func(c context.Context) {

        start := time.Now()
        c.Next()
        latency := time.Now().Sub(start)
        // build access_log message
        accessLogMsg := buildAccessLogMsg(c, latency)
        if len(accessLogMsg) > 0 {
            alw.Writer(AccessLogData{AccessLogConfig: *alc, AccessLogMsg: accessLogMsg})
        }
    }
```

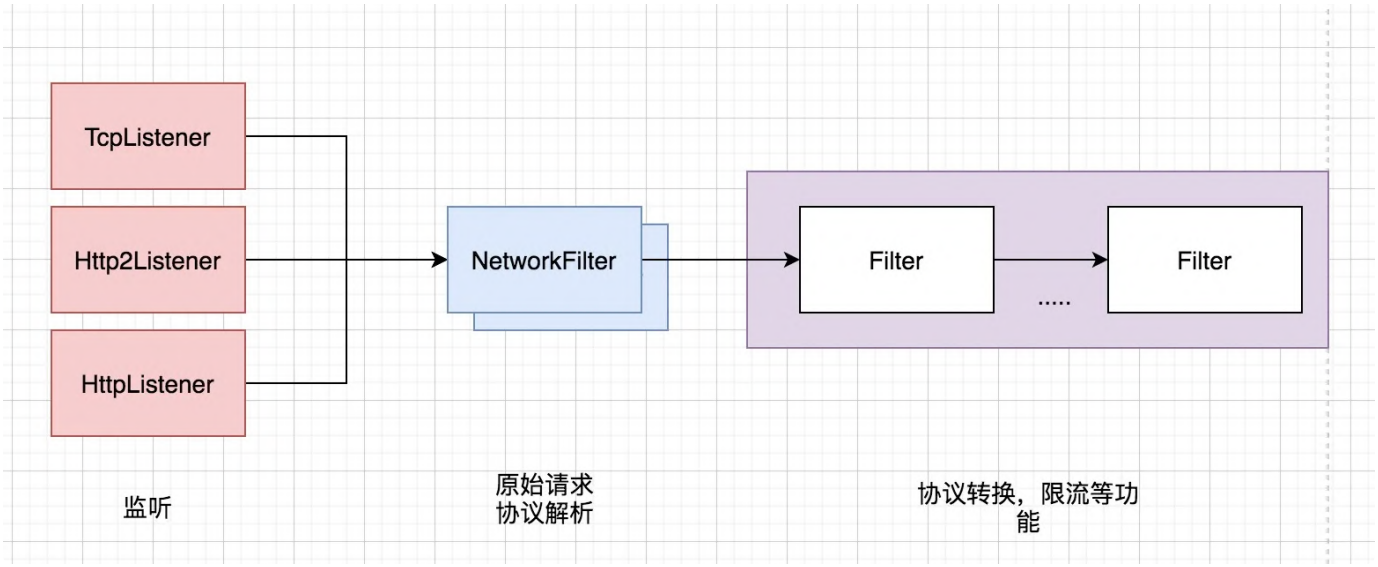
```
43  }
44
45
```

讨论问题：

- Init 改成 init，然后有一个统一的文件专门匿名import即可
- 将工厂函数和filter函数统一抽象成Plugin 和 HttpFilter（我这边还有network filter 和 spring cloud 的adapter）
    - 将注册Factory改成注册Plugin，因为Networkfilter和Adapter都是统一的Plugin模式，Plugin中有专门的factory函数
    - Filter也要抽象成interface，不只是一个Func指针，Filter中专门的Func函数

dubbo 协议转换的 Filter 接口定义问题

之前聊过，之前 Pixiu 的 Filter 分为 NetworkFilter 和 HttpFilter。
这种定义很好适配了原有以 Http 协议为主处理场景。



但是新增了对 dubbo 协议的转换功能，这套机制受到了挑战。
dubbo to triple，http 和 triple to dubbo 这三个协议转换时，最终进行处理的结构体是 RPCInvocation和 RPCResult。类似于Reuqest 和 Response。

旧的HttpFilter 中的 handle 定义无法使用。

```go
type HttpFilter interface {
    Handle(ctx *http.HttpContext)
}
```

那么，对应这类多协议情况下，Filter 的接口如何定义?

- 不同协议使用各自不同定义的 Filter；
- Filter 扩充函数定义，接收不同的参数。

第一种相当于每一种协议转换对都有一个 Filter，比如说 http to dubbo 和 triple to dubbo，虽然都是转换到dubbo，但是源协议不同，所以需要两个独立的 Filter

第二种相当于只要目标协议相同，就只有一个 Filter，比如说 http to dubbo 和 triple to dubbo，都是转换到dubbo，所以是一个 Filter，它提供了两个函数，分别两种不同的源协议 http

http -> dubbo 转换策略

一、目前 pixiu 支持的策略

1.1 静态配置

支持完全通过配置指定 dubbo 转发所需信息。

YAML

```yaml
- path: '/api/v1/test-dubbo/student/name'
    type: restful
    methods:
      - httpVerb: GET
        onAir: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
        integrationRequest:
          requestType: dubbo
          applicationName: "StudentService"
          interface: "com.dubbogo.server.StudentService"
          method: "GetStudentByName"
          group: "test"
          version: 1.0.0
          clusterName: "test_dubbo"
```

所有支持配置的数据如下所示(注册中心的地址等信息需要额外配置)

YAML

```yaml
type DubboBackendConfig struct {
  ClusterName     string `yaml:"clusterName" json:"clusterName"`
  ApplicationName string `yaml:"applicationName" json:"applicationName"`
  Protocol        string `yaml:"protocol" json:"protocol,omitempty" defaul
t:"dubbo"`
  Group           string `yaml:"group" json:"group"`
  Version         string `yaml:"version" json:"version"`
  Interface       string `yaml:"interface" json:"interface"`
  Method          string `yaml:"method" json:"method"`
  Retries         string `yaml:"retries" json:"retries,omitempty"`
}
```

1.2 按照配置从 http 请求中转换

从指定的 Path、Query、Header和 Body 位置获得 application、interface、method、group 和 version 等信息。

```yaml
name: server
description: server sample
resources:
  - path: '/api/v1/test-dubbo/:application/:interface'
    type: restful
    description: common
    methods:
      - httpVerb: POST
        onAir: true
        timeout: 100s
        inboundRequest:
          requestType: http
        integrationRequest:
          requestType: dubbo
          mappingParams:
            - name: requestBody.values
              mapTo: opt.values
            - name: requestBody.types
              mapTo: opt.types
            - name: uri.application   # 从 path variable 中获取 application
              mapTo: opt.application
            - name: uri.interface
              mapTo: opt.interface
            - name: queryStrings.method   # 从 query variable 中获取 method
              mapTo: opt.method
            - name: queryStrings.group
              mapTo: opt.group
            - name: queryStrings.version
              mapTo: opt.version
          clusterName: "test_dubbo"
```

二、待讨论问题

2.1 多协议支持

dubbo框架支持多协议，目前pixiu的只支持 dubbo 协议

2.2 默认约定策略

不需要配置，根据默认转换策略从 http 请求中获得元信息，转换为dubbo请求，例如默认将application、interface、method等放在Query中。但是这样对于调用方来讲，需要了解的信息太多。

http to dubbo规约

一、默认规约

1.1 发起泛化调用所需的信息：

- applicationName
  - 不需要，gateway统一一个即可
- interfaceName
  - queryVariable
- cluster
  - queryVariable
- registry相关的信息
  - 配置固化
- protocol
  - 配置固化
- methodName
  - queryVariable
- type
  - body
- value
  - body

YAML

```
{
  "type": ["org.apache.dubbo.User"],
  "value": [{"id":123,"name":"Tom"}]
}
```

泛化调用的返回值是一个map格式的pojo和err。

方案一：将其转换为http response的话，err相关问题转换为httpcode，而map转换为body中。

方案二：将error细分，形成相应的code和map一起放入body中，更加有利于下游对不同的场景进行差异对待。

二、泛化调用

3.0 和 1.5

YAML

```yaml
var (
  appName         = "UserConsumer"
  referenceConfig = config.ReferenceConfig{
    InterfaceName: "org.apache.dubbo.UserProvider",
    Cluster:       "failover",
    Registry:      "demoZk",
    Protocol:      dubbo.DUBBO, # 如果是triple协议，则serialization必须是hessian2
    Generic:       "true",
  }
)


func callQueryUser() {
  gxlog.CInfo("\n\n\nstart to generic invoke")
    resp, err := referenceConfig.GetRPCService().(*config.GenericService).Invoke(
      context.TODO(),
      []interface{}{
        "queryUser",
        []string{"org.apache.dubbo.User"},
        []hessian.Object{map[string]hessian.Object{
          "id":   "3213",
          "name": "panty",
          "age":  25,
          "time": time.Now(),
        }},
      },
    )
    if err != nil {
      panic(err)
    }
    gxlog.CInfo("res: %+v\n", resp)
    gxlog.CInfo("success!")
}
```

配置和整体资源抽象

一、背景

pixiu 在 0.4.0 开发过程中，包括 spring cloud 自动路由转发功能和 http to grpc 协议转换功能，配置简化功能等，涉及到差异化配置和代码组织问题，考虑到后续还会接入 MQ ，提供更多默认插件等功能，所以急需对 pixiu 的整体代码架构和配置进行统一抽象，方便各类功能在该架构和配置上进行统一，避免后续再统一时额外付出的重构成本。

二、目的

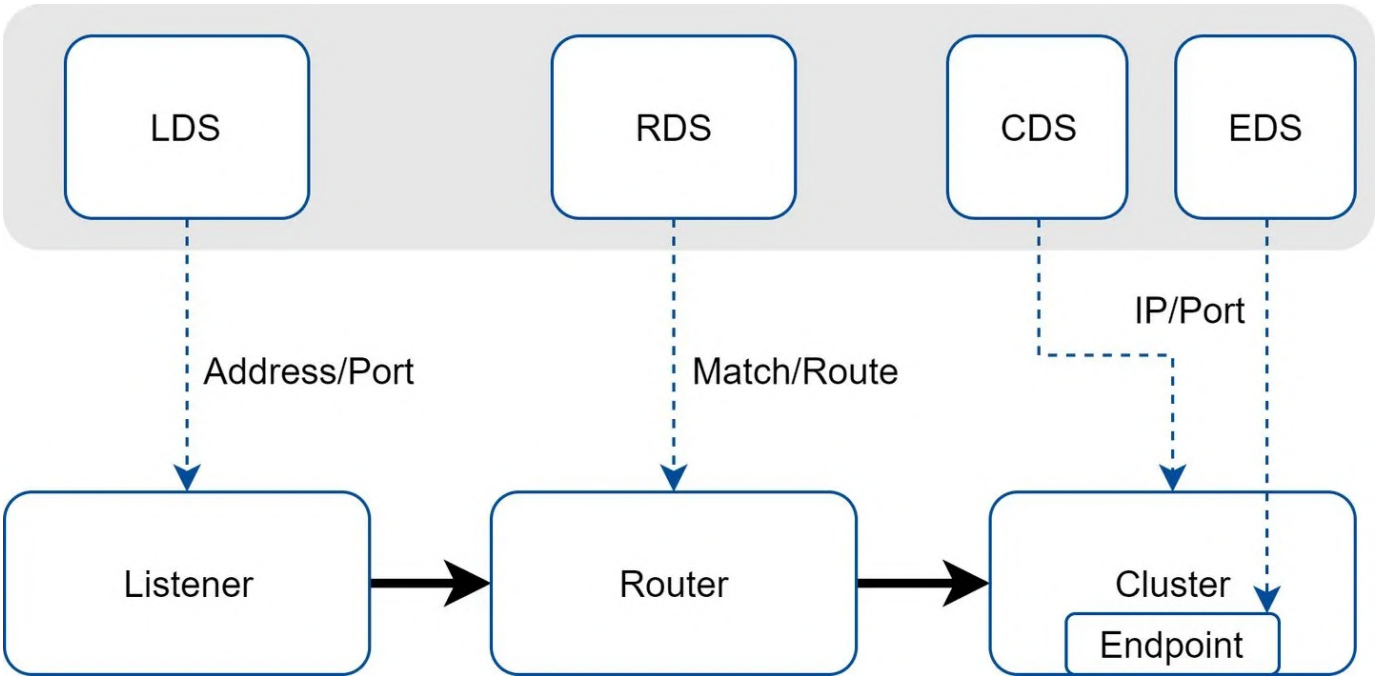
敲定 pixiu 的整体抽象架构和配置组织方式，方便后续功能不断集成。

三、资料

3.1 envoy xDS

https://zhuanlan.zhihu.com/p/108846492

- Listener(LDS)、Router(RDS)、Cluster(CDS和EDS)和Filter(嵌入上述配置)抽象



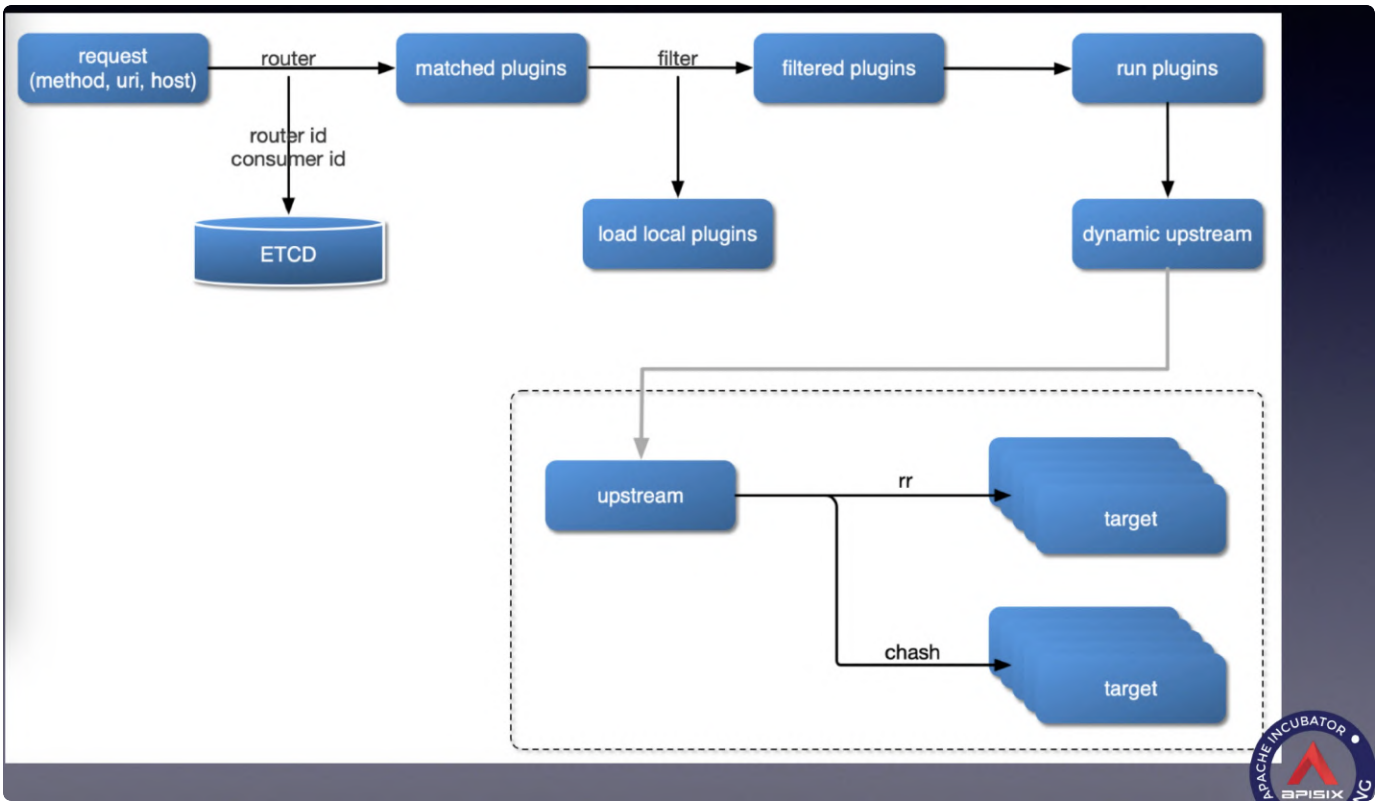
3.2 apisix

https://opentalk-blog.b0.upaiyun.com/prod/2019-10-28/9de5fe4f15bafec21b063d2e373ab7dd.pdf

摘要：

- radixtree 前缀树路由，提高路由性能
- 全流程插件体系

- 多协议接入，可以转发 http，grpc，MQTT等。



YAML

```
# upstream
curl "http://127.0.0.1:9080/apisix/admin/upstreams/1" -H "X-API-KEY: edd1c9f034335f136f87ad84b625c8f1" -X PUT -d '
{
  "type": "roundrobin",
  "nodes": {
    "httpbin.org:80": 1
  }
}'

# plugin
curl "http://127.0.0.1:9080/apisix/admin/consumers" -H "X-API-KEY: edd1c9f034335f136f87ad84b625c8f1" -X PUT -d '
{
  "username": "john",
  "plugins": {
    "key-auth": {
      "key": "key-of-john"
    }
  }
}'
# route
curl "http://127.0.0.1:9080/apisix/admin/routes/1" -H "X-API-KEY: edd1c9f034335f136f87ad84b625c8f1" -X PUT -d '
{
  "uri": "/get",
  "host": "httpbin.org",
  "plugins": {
    "key-auth": {}
  },
  "upstream_id": "1"
}'

# grpc proxy

curl http://127.0.0.1:9080/apisix/admin/routes/1 -H 'X-API-KEY: edd1c9f034335f136f87ad84b625c8f1' -X PUT -d '
{
    "methods": ["POST", "GET"],
    "uri": "/helloworld.Greeter/SayHello",
    "upstream": {
        "scheme": "grpc",
        "type": "roundrobin",
        "nodes": {
            "127.0.0.1:50051": 1
```

```
        }
    }
}'
grpcurl -insecure -import-path /pathtoprotos  -proto helloworld.proto  \
    -d '{"name":"apisix"}' 127.0.0.1:9443 helloworld.Greeter.SayHello
{
  "message": "Hello apisix"
}

# http to grpc

curl http://127.0.0.1:9080/apisix/admin/routes/111 -H 'X-API-KEY: edd1c9f0
34335f136f87ad84b625c8f1' -X PUT -d '
{
    "methods": ["GET"],
    "uri": "/grpctest",
    "plugins": {
        "grpc-transcode": {
         "proto_id": "1",
         "service": "helloworld.Greeter",
         "method": "SayHello"
        }
    },
    "upstream": {
        "scheme": "grpc",
        "type": "roundrobin",
        "nodes": {
            "127.0.0.1:50051": 1
        }
    }
}'
$ curl -i http://127.0.0.1:9080/grpctest?name=world
```

3.3 shenyu&soul

http://events.jianshu.io/p/0ce27807323a

https://shenyu.apache.org/zh/projects/shenyu/flow-control/

https://shenyu.apache.org/zh/projects/shenyu/database-design/

https://shenyu.apache.org/zh/projects/shenyu/custom-plugin/

- 插件-选择器-规则：细粒度控制
- 客户端接入体系，网关背后的各类服务通过client接入到shenyu注册中心
- 支持http->x协议，不支持其他协议proxy ？

- 自定义插件
  - 一类是单一职责的调用链，不能对流量进行自定义的筛选。
  - 一类是能对匹配的流量，执行自己的职责调用链。

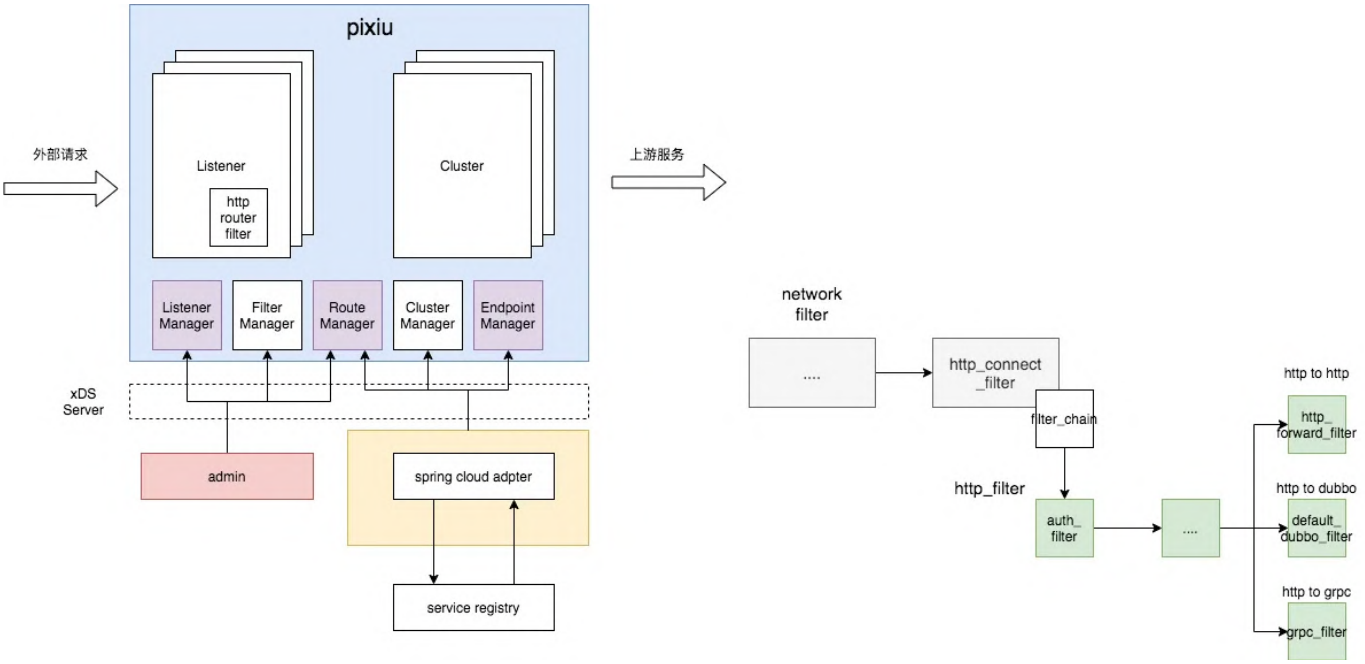
3.4 easegress

https://github.com/megaease/easegress

四、待讨论方案

4.1 总述

~~按照铁城第一个版本的 conf.yaml 和 envoy 的思路，扩充static_resource对应的动态配置 dynamic_resource，并且调整 api_config.yaml 中配置及其相关逻辑对整个结构的影响。~~



4.2 conf.yaml

目前，在conf.yaml中的配置如下所示，对应结构体详见 model 文件夹：

```yaml
---
static_resources:
  listeners:
    - name: "net/http"
      address:
        socket_address:
          protocol_type: "HTTP"
          address: "0.0.0.0"
          port: 8888
      filter_chains:
        - filter_chain_match:
          filters:
            - name: dgp.filters.http_connect_manager
                http_filters:
                  - name: dgp.filters.http.authority_filter
                    config:
                  - name: dgp.filters.remote_call // 1 原有逻辑，路由依赖api_config
                    config:
                server_name: "test_http_dubbo"
                generate_request_id: false
  clusters:
    - name: "test_dubbo"
      lb_policy: "RoundRobin"
      adapter: api_config
  adapter:
     name: api_config
     config:
        - path: conf/api_config.yaml
        - registry:
            "zookeeper":
            protocol: "zookeeper"
            timeout: "3s"
            address: "127.0.0.1:2181"
            username: ""
            password: ""
```

http to grpc

```yaml
      filter_chains:
        - filter_chain_match:
          filters:
            - name: dgp.filters.http_connect_manager
              config:
                route_config:
                  name: local_route
                  routes:
                    - match: { prefix: "/helloworld.Greeter" }
                      route: { cluster: grpc, timeout: { seconds: 60 } }
                http_filters:
                  - name: dgp.filters.grpc_translator
                    config:
                      proto_descriptor: "/tmp/proto.pb"
                      services: ["helloworld.Greeter"]
  cluster:
    - name: grpc
      address: xxx
```

spring cloud

```yaml
       filter_chains:
        - filter_chain_match:
          filters:
            - name: dgp.filters.http_connect_manager
                http_filters:
                  - name: dgp.filters.http.route
   clusters:
     - name: "spring cloud"
       lb_policy: "RoundRobin"
       adpter: spring cloud
   adpater:
     - name: "consul registry"
       plugin: dgp.plugin.adpater.consul
       config:
         address:
         xxx
```

五、详细改动

基于梦超的 Make filter configurable & Centralized management分支代码，

- pixiu/listener中routeRequest不应该直接取api_config中配置，而是应该应该经过filters的层层处理，如果达到remote_call，然后在做进一步更细致的路由和plugin操作(route和filter的先后关系？)
- listener,filter[],route[disvoeryservice?],cluster,endpoint的相关manager
- adpater 启动流程
- pixiu初始化时，api_config相关配置的初始化是否可以延后

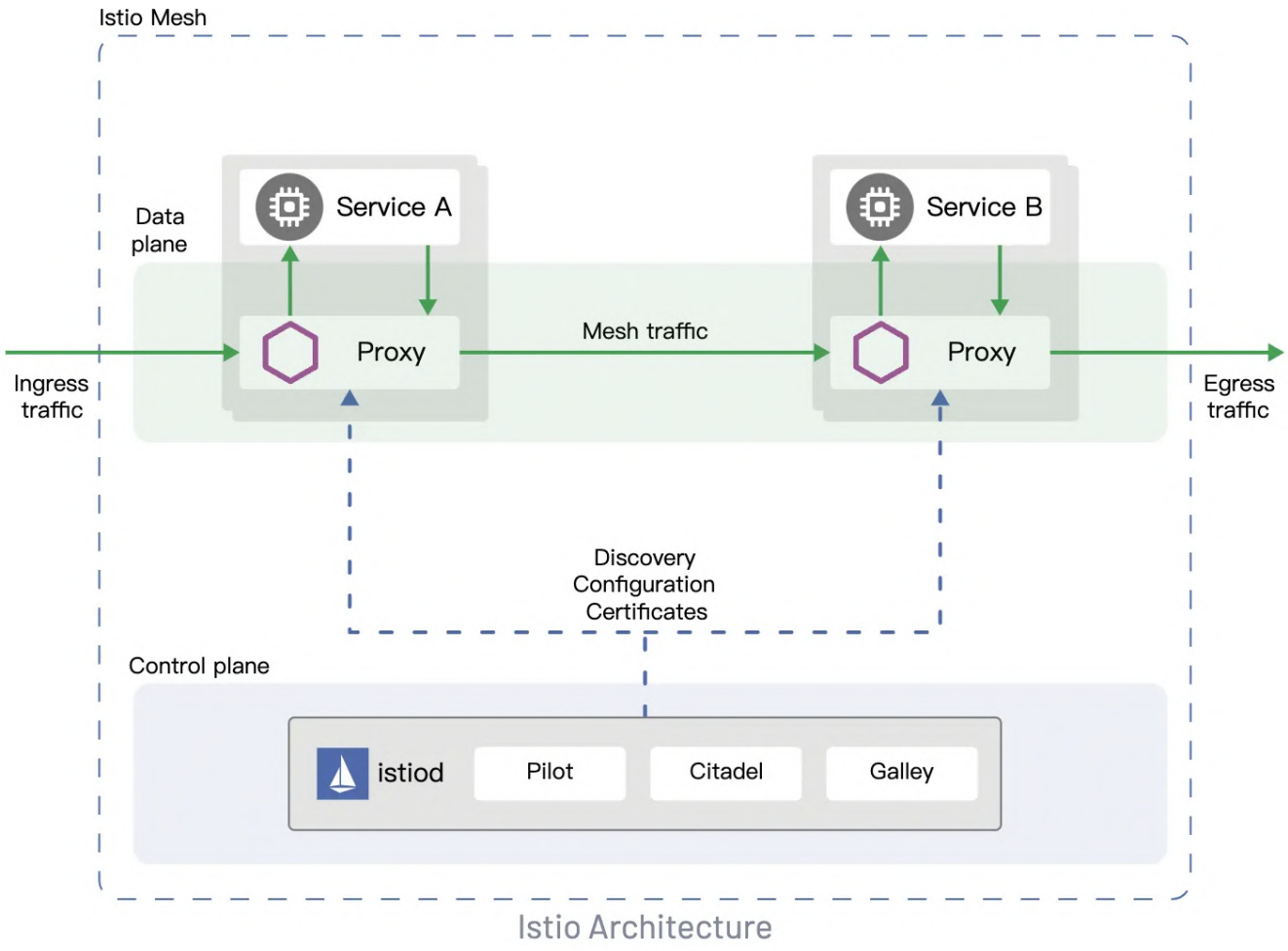
六、计划时间

xDS资料

https://www.envoyproxy.io/docs/envoy/latest/api-v3/api#envoy-v3-api-reference

https://www.servicemesher.com/istio-handbook/

istio架构



Istio Architecture

Envoy

Istio uses an extended version of the Envoy proxy. Envoy is a high-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. Envoy proxies are the only Istio components that interact with data plane traffic.

Istiod

Istiod provides service discovery, configuration and certificate management.

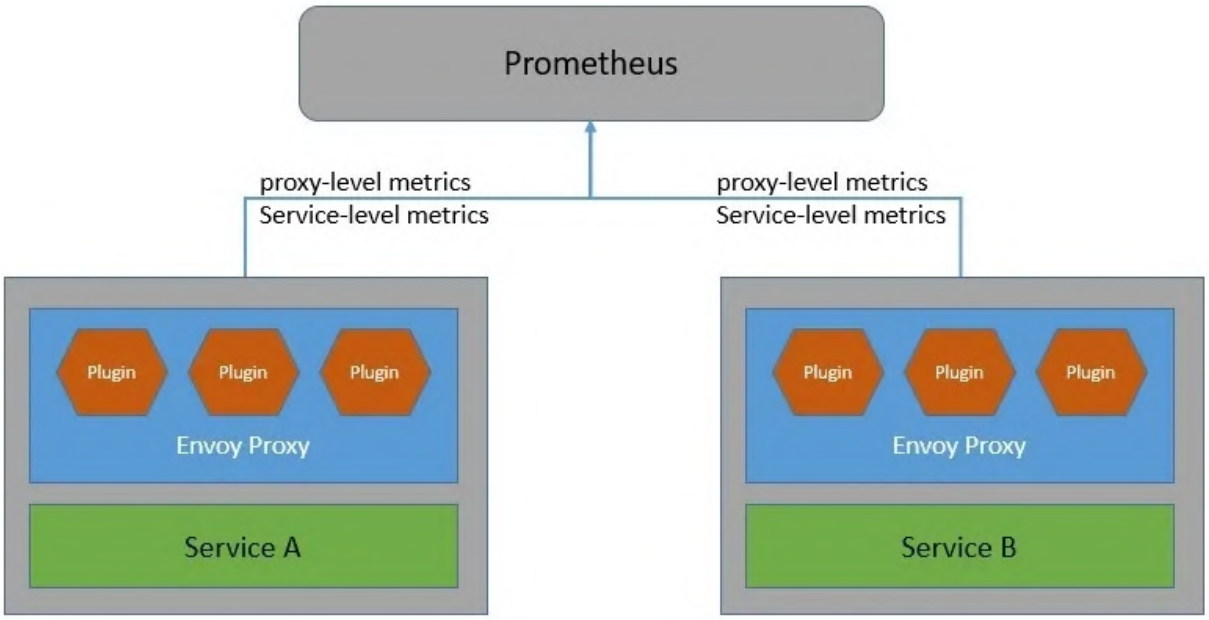
Istiod converts high level routing rules that control traffic behavior into Envoy–specific configurations, and propagates them to the sidecars at runtime. Pilot abstracts platform–specific service discovery mechanisms and synthesizes them into a standard format that any sidecar conforming with the Envoy API can consume.

istio metric

Observability

Istio generates detailed telemetry like metrics, distributed traces, and access logs for all service communication within the mesh. Istio **generates a rich set of proxy-level metrics, service-oriented metrics, and control plane metrics**.

Earlier, the Istio telemetry architecture included Mixer as a central component. But starting with Telemetry v2, features provided by Mixer were replaced with the Envoy proxy plugins:



Moreover, Istio generates distributed traces through the Envoy proxies. Istio supports a number of tracing backends like Zipkin, Jaeger, Lightstep, and Datadog. We can also control the sampling rate for trace generation. Further, Istio also generates access logs for service traffic in a configurable set of formats.

Pilot

在应用从单体架构向微服务架构演进的过程中，微服务之间的服务发现、负载均衡、熔断、限流等服务治理需求是无法回避的问题。

在 Service Mesh 出现之前，通常的做法是将这些基础功能以 SDK 的形式嵌入业务代码中，但是这种强耦合的方案会增加开发的难度，增加维护成本，增加质量风险。比如 SDK 需要新增新特性，业务侧也很难配合 SDK 开发人员进行升级，所以很容易造成 SDK 的版本碎片化问题。如果再存在跨语言应用间的交互，对于多语言 SDK 的支持也非常的低效。一方面是相当于相同的代码以不同语言重复实现，实现这类代码既很难给开发人员带来成就感，团队稳定性难以保障；另一方面是如果实现这类基础框架时涉及到了语言特性，其他语言的开发者也很难直接翻译。

而 Service Mesh 的本质则是将此类通用的功能沉淀至 sidecar 中，由 sidecar 接管服务的流量并对其进行治理。在这个思路下，可以通过流量劫持的手段，做到代码零侵入性。这样可以让业务开发人员更关心业务功能。而底层功能由于对业务零侵入，也使得基础功能的升级和快速的更新迭代成为可能。

Istio 是近年来 Service Mesh 的代表作，而 Istio 流量管理的核心组件就是 Pilot。Pilot 主要功能就是管理和配置部署在特定 Istio 服务网格中的所有 sidecar 代理实例。它管理 sidecar 代理之间的路由流量规则，并配置故障恢复功能，如超时、重试和熔断。

Pilot 架构

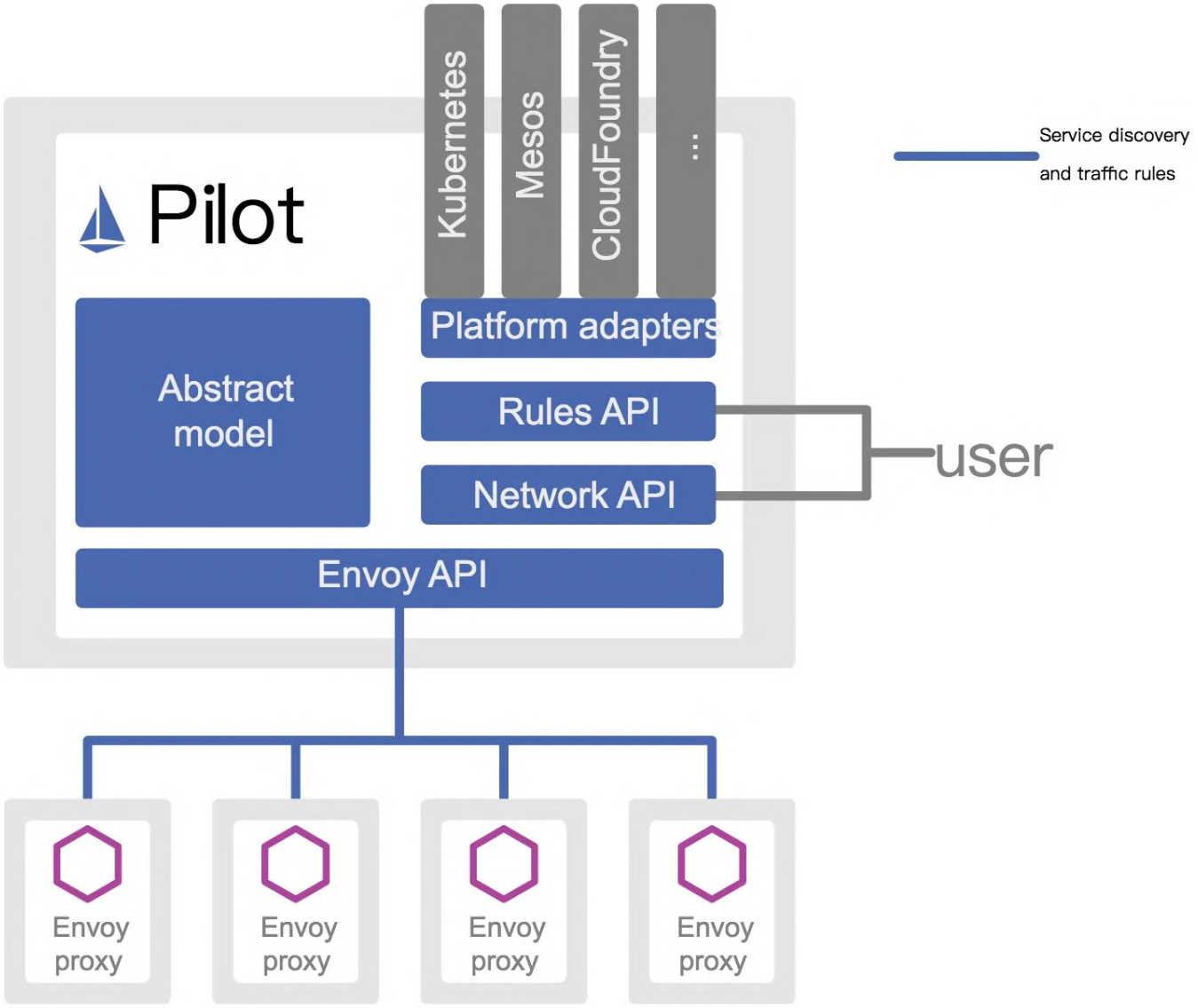


图 2.3.1.1.1：Pilot 架构（图片来自Istio官方网站）

根据上图， Pilot 几个关键的模块如下。

抽象模型 （Abstract Model）

为了实现对不同服务注册中心 （Kubernetes、consul） 的支持，Pilot 需要对不同的输入来源的数据有一个统一的存储格式，也就是抽象模型。

抽象模型中定义的关键成员包括 HostName（service 名称）、Ports（service 端口）、Address（service ClusterIP）、Resolution （负载均衡策略） 等。

平台适配器 （Platform adapters）

Pilot 的实现是基于平台适配器（Platform adapters） 的，借助平台适配器 Pilot 可以实现服务注册中心数据到抽象模型之间的数据转换。

例如 Pilot 中的 Kubernetes 适配器通过 Kubernetes API 服务器得到 Kubernetes 中 service 和 pod 的相关信息，然后翻译为抽象模型提供给 Pilot 使用。

通过平台适配器模式， Pilot 还可以从 Consul 等平台中获取服务信息，还可以开发适配器将其他提供服务发现的组件集成到 Pilot 中。

协议一览

服务简写	全称	描述
LDS	Listener Discovery Service	监听器发现服务
RDS	Route Discovery Service	路由发现服务
CDS	Cluster Discovery Service	集群发现服务
EDS	Endpoint Discovery Service	集群成员发现服务
SDS	Service Discovery Service	v1 时的集群成员发现服务，后改名为 EDS
ADS	Aggregated Discovery Service	聚合发现服务
HDS	Health Discovery Service	健康度发现服务
SDS	Secret Discovery Service	密钥发现服务
MS	Metric Service	指标发现服务
RLS	Rate Limit Service	限流发现服务

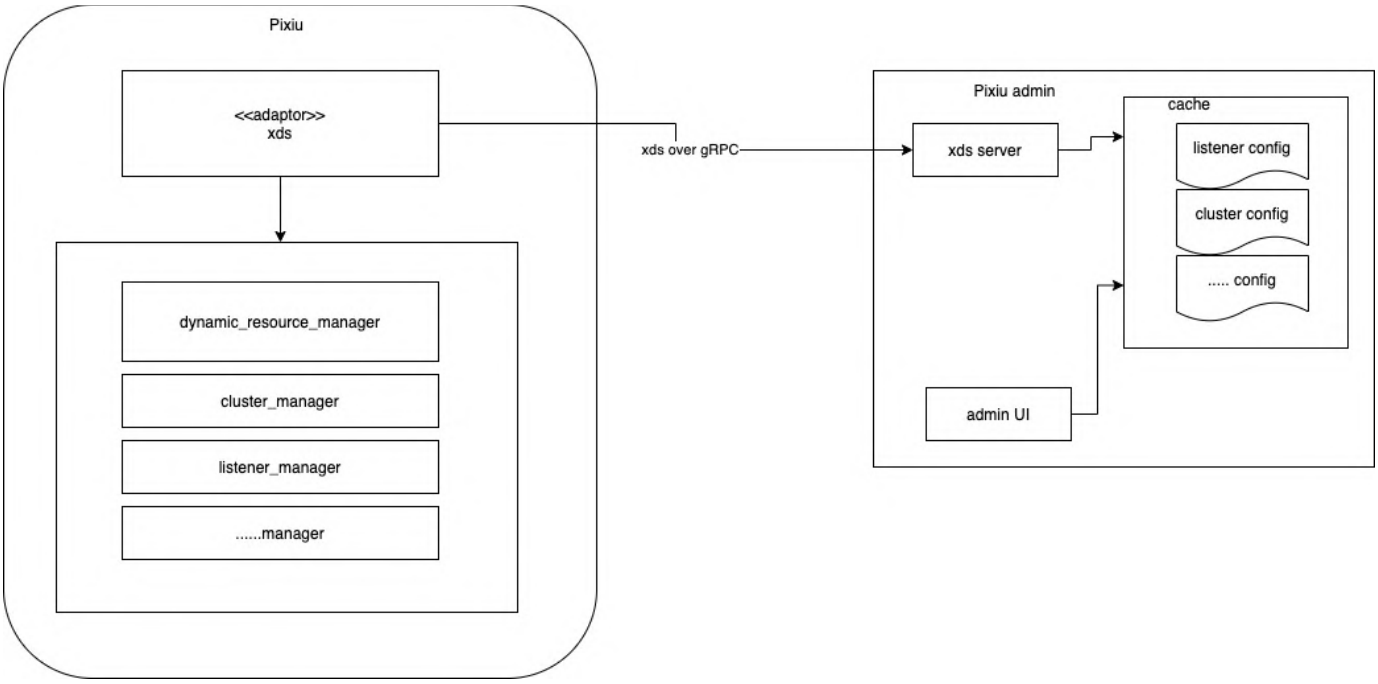
xDS API Flow

对于典型的 HTTP 路由场景，客户端配置的核心资源类型是 Listener、RouteConfiguration、Cluster 和 ClusterLoadAssignment。每个Listener资源可能指向一个RouteConfiguration资源，RouteConfiguration资源可能指向一个或多个Cluster资源，每个Cluster资源可能指向一个ClusterLoadAssignment资源。

Envoy 在启动时获取所有 Listener 和 Cluster 资源。然后它获取侦听器和集群资源所需的任何 RouteConfiguration 和 ClusterLoadAssignment 资源。实际上，每个 Listener 或 Cluster 资源都是 Envoy 配置树的一部分的根。

非代理客户端（例如 gRPC）可能首先只获取它感兴趣的特定侦听器资源。然后它获取那些侦听器资源所需的 RouteConfiguration 资源，然后是那些 RouteConfiguration 资源所需的任何集群资源，然后是集群资源所需的 ClusterLoadAssignment 资源。实际上，原始侦听器资源是客户端配置树的根。

pixiu 集成xds技术设计



1. dynamic_resource_manager: 加载管理 dynamic resource的配置信息。
2. cluster_manager/listener_manager....实现对pixiu cluster/listener的动态修改。
3. xds adaptor: 是插件化形式，初始化时，load LDS/CDS/....配置，创建xds client(control-panel)，同步pixiu admin 中配置的各种资源。
    a. xds client：通过xds Delta(gRPC streaming)API，动态获取资源配置的变化。
    b. xds adaptor通过cluster manager / listener manager等，动态修改pixiu的cluster和listener.
4. xds server(data-panel) 维护所有pixiu节点的config的snapshot，跟踪节点配置版本。
    a. 可以通过admin UI来维护cache中的config。
    b. xds server可以管理每一个pixiu节点的配置。
5. pixiu admin cache中的配置是基于protobuf定义的配置信息。

LDS集成设计

LDS集成设计

Listener 动态配置能力调研

pixiu根据主要组件listener用来接收client端流量，对于listener是否需要动态配置，参考对比其他API gateway的listener 配置能力，对比结果只有envoy支持动态配置能力：

nginx	Config reload	
traefik	在静态配置中指定 entrypoint, middlewares可以动态配置	图片加载失败
apisix	By default, the Admin API listens to port 9080 (9443 for HTTPS) when APISIX is launched. This can be changed by modifying your configuration file (conf/config.yaml).	图片加载失败

envoy	Generally we recommend running a single Envoy per machine regardless of the number of configured listeners. Listeners can also be fetched dynamically via the listener discovery service (LDS). The listener discovery service (LDS) is an optional API that Envoy will call to dynamically fetch listeners. Envoy will reconcile the API response and add, modify, or remove known listeners depending on what is required.	https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/lds#config-listeners-lds

go-control-plane 调研

go-control-panel使用golang实现了envoy xDS API server.
项目地址：https://github.com/envoyproxy/go-control-plane

运行demo

1. 安装envoy
   Mac: brew install envoy
2. 下载代码
   git clone https://github.com/envoyproxy/go-control-plane
3. 运行demo
   make example
4. 测试：
   curl -v http://localhost:10000

curl -v localhost:10000

- Trying ::1...
- TCP_NODELAY set
- Connection failed
- connect to ::1 port 10000 failed: Connection refused
- Trying 127.0.0.1...
- TCP_NODELAY set
- Connected to localhost (127.0.0.1) port 10000 (#0)

> GET / HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.1
> Accept: /

< HTTP/1.1 301 Moved Permanently
< cache-control: public, max-age=0, must-revalidate

< content-length: 42
< content-security-policy: frame-ancestors 'self';
< content-type: text/plain
< date: Fri, 05 Nov 2021 05:00:48 GMT
< age: 33911
< location: https://www.envoyproxy.io/
< x-nf-request-id: 01FKR7VXRSC2QMTTECDZA1MDK6
< server: envoy
< x-envoy-upstream-service-time: 1439

Envoy对于control panel的配置方式

参考： https://www.envoyproxy.io/docs/envoy/latest/start/quick-start/configuration-dynamic-control-plane

通过静态资源static_resource配置API Server的位置。
dynamic_resource的各部分（cds|lds|...)通过的api_config_source配置到到对应的cluster(static_resource)。
服务端提供API 向envoy提供配置：

```
const (
  ClusterName  = "example_proxy_cluster"
  RouteName    = "local_route"
  ListenerName = "listener_0"
  ListenerPort = 10002
  UpstreamHost = "www.envoyproxy.io"
  UpstreamPort = 80
)

func makeCluster(clusterName string) *cluster.Cluster {
  return &cluster.Cluster{
     Name:                 clusterName,
     ConnectTimeout:       durationpb.New(5 * time.Second),
     ClusterDiscoveryType: &cluster.Cluster_Type{Type: cluster.Cluster_LOGICAL_DNS},
     LbPolicy:             cluster.Cluster_ROUND_ROBIN,
```

```
    LoadAssignment:        makeEndpoint(clusterName),
    DnsLookupFamily:       cluster.Cluster_V4_ONLY,
  }
}

func makeEndpoint(clusterName string) *endpoint.ClusterLoadAssignment {
  return &endpoint.ClusterLoadAssignment{
    ClusterName: clusterName,
    Endpoints: []*endpoint.LocalityLbEndpoints{{
      LbEndpoints: []*endpoint.LbEndpoint{{
        HostIdentifier: &endpoint.LbEndpoint_Endpoint{
          Endpoint: &endpoint.Endpoint{
            Address: &core.Address{
              Address: &core.Address_SocketAddress{
                SocketAddress: &core.SocketAddress{
                  Protocol: core.SocketAddress_TCP,
                  Address:  UpstreamHost,
                  PortSpecifier: &core.SocketAddress_PortValue{
                    PortValue: UpstreamPort,
                  },
                },
              },
            },
          },
        },
      }},
    }},
  }
}

func makeRoute(routeName string, clusterName string) *route.RouteConfiguration {
  return &route.RouteConfiguration{
    Name: routeName,
    VirtualHosts: []*route.VirtualHost{{
      Name:    "local_service",
      Domains: []string{"*"},
```

```
        Routes: []*route.Route{{
          Match: &route.RouteMatch{
            PathSpecifier: &route.RouteMatch_Prefix{
              Prefix: "/",
            },
          },
          Action: &route.Route_Route{
            Route: &route.RouteAction{
              ClusterSpecifier: &route.RouteAction_Cluster{
                Cluster: clusterName,
              },
              HostRewriteSpecifier: &route.RouteAction_HostRewriteLiteral{
                HostRewriteLiteral: UpstreamHost,
              },
            },
          },
        }},
      }},
    }
}

func makeHTTPListener(listenerName string, route string) *listener.Listener {
  // HTTP filter configuration
  manager := &hcm.HttpConnectionManager{
    CodecType:  hcm.HttpConnectionManager_AUTO,
    StatPrefix: "http",
    RouteSpecifier: &hcm.HttpConnectionManager_Rds{
      Rds: &hcm.Rds{
        ConfigSource:    makeConfigSource(),
        RouteConfigName: route,
      },
    },
    HttpFilters: []*hcm.HttpFilter{{
      Name: wellknown.Router,
    }},
  }
```

```
  pbst, err := anypb.New(manager)
  if err != nil {
    panic(err)
  }

  return &listener.Listener{
    Name: listenerName,
    Address: &core.Address{
      Address: &core.Address_SocketAddress{
        SocketAddress: &core.SocketAddress{
          Protocol: core.SocketAddress_TCP,
          Address:  "0.0.0.0",
          PortSpecifier: &core.SocketAddress_PortValue{
            PortValue: ListenerPort,
          },
        },
      },
    },
    FilterChains: []*listener.FilterChain{{
      Filters: []*listener.Filter{{
        Name: wellknown.HTTPConnectionManager,
        ConfigType: &listener.Filter_TypedConfig{
          TypedConfig: pbst,
        },
      }},
    }},
  }
}

func makeConfigSource() *core.ConfigSource {
  source := &core.ConfigSource{}
  source.ResourceApiVersion = resource.DefaultAPIVersion
  source.ConfigSourceSpecifier = &core.ConfigSource_ApiConfigSource{
    ApiConfigSource: &core.ApiConfigSource{
      TransportApiVersion:       resource.DefaultAPIVersion,
      ApiType:                  core.ApiConfigSource_GRPC,
```

```
        SetNodeOnFirstMessageOnly: true,
        GrpcServices: []*core.GrpcService{{
          TargetSpecifier: &core.GrpcService_EnvoyGrpc_{
            EnvoyGrpc: &core.GrpcService_EnvoyGrpc{ClusterName: "xds_cluster"},
          },
        }},
      },
    }
    return source
}

func GenerateSnapshot() cache.Snapshot {
    snap, _ := cache.NewSnapshot("1",
      map[resource.Type][]types.Resource{
        resource.ClusterType:  {makeCluster(ClusterName)},
        resource.RouteType:    {makeRoute(RouteName, ClusterName)},
        resource.ListenerType: {makeHTTPListener(ListenerName, RouteName)},
      },
    )
    return snap
}
```

Envoy的配置：

```
# Base config for a split xDS management server on 18000, admin port on 19000
admin:
 access_log_path: /dev/null
 address:
   socket_address:
     address: 127.0.0.1
     port_value: 19000
dynamic_resources:
 cds_config:
   resource_api_version: V3
   api_config_source:
     api_type: GRPC
```

```
      transport_api_version: V3
      grpc_services:
      - envoy_grpc:
          cluster_name: xds_cluster
      set_node_on_first_message_only: true
  lds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
      grpc_services:
      - envoy_grpc:
          cluster_name: xds_cluster
      set_node_on_first_message_only: true
node:
  cluster: test-cluster
  id: test-id
static_resources:
  clusters:
  - connect_timeout: 1s
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: 127.0.0.1
                port_value: 18000
    http2_protocol_options: {}
    name: xds_cluster
  - connect_timeout: 1s
    load_assignment:
      cluster_name: als_cluster
      endpoints:
      - lb_endpoints:
```

```
      - endpoint:
          address:
            socket_address:
              address: 127.0.0.1
              port_value: 18090
  http2_protocol_options: {}
  name: als_cluster
layered_runtime:
 layers:
  - name: runtime-0
    rtds_layer:
      rtds_config:
        resource_api_version: V3
        api_config_source:
          transport_api_version: V3
          api_type: GRPC
          grpc_services:
            envoy_grpc:
              cluster_name: xds_cluster
      name: runtime-0
```

动态修改xDS server的配置, sync到envoy。

xDS server 可以通过SnapshotCache.SetSnapshot 方法修改动态把新的配置sync到envoy。

```
type SnapshotCache interface {
  Cache

  // SetSnapshot sets a response snapshot for a node. For ADS, the snapshots
  // should have distinct versions and be internally consistent (e.g. all
  // referenced resources must be included in the snapshot).
  //
  // This method will cause the server to respond to all open watches, for which
```

```
  // the version differs from the snapshot version.
  SetSnapshot(ctx context.Context, node string, snapshot Snapshot) error
```

源码解读

```
// SetSnapshotCacheContext updates a snapshot for a node.
func (cache *snapshotCache) SetSnapshot(ctx context.Context, node string, snapshot
Snapshot) error {
  cache.mu.Lock()
  defer cache.mu.Unlock()

  // update the existing entry
  cache.snapshots[node] = snapshot

  // trigger existing watches for which version changed
  if info, ok := cache.status[node]; ok {
     info.mu.Lock()
     defer info.mu.Unlock()
     for id, watch := range info.watches {
        version := snapshot.GetVersion(watch.Request.TypeUrl)
        if version != watch.Request.VersionInfo {
           cache.log.Debugf("respond open watch %d%v with new version %q", id,
watch.Request.ResourceNames, version)

           resources := snapshot.GetResourcesAndTTL(watch.Request.TypeUrl)
           err := cache.respond(ctx, watch.Request, watch.Response, resources, version, false)
           if err != nil {
              return err
           }

           // discard the watch
           delete(info.watches, id)
        }
     }
```

```
    // We only calculate version hashes when using delta. We don't
    // want to do this when using SOTW so we can avoid unnecessary
    // computational cost if not using delta.
    if len(info.deltaWatches) > 0 {
      err := snapshot.ConstructVersionMap()
      if err != nil {
        return err
      }
    }

    // process our delta watches
    for id, watch := range info.deltaWatches {
      res, err := cache.respondDelta(
        ctx,
        &snapshot,
        watch.Request,
        watch.Response,
        watch.StreamState,
      )
      if err != nil {
        return err
      }
      // If we detect a nil response here, that means there has been no state change
      // so we don't want to respond or remove any existing resource watches
      if res != nil {
        delete(info.deltaWatches, id)
      }
    }
  }

  return nil
}
```

xDS server扩展

xDS 服务分层

服务栈	
server	go-control-panel/pkg/server --- sotw (SoTW (State of The World)) --- rest (grpc gateway) --- grpc (v3)

API	endpointservice.EndpointDiscoveryServiceServer clusterservice.ClusterDiscoveryServiceServer routeservice.RouteDiscoveryServiceServer routeservice.ScopedRoutesDiscoveryServiceServer listenerservice.ListenerDiscoveryServiceServer discoverygrpc.AggregatedDiscoveryServiceServer secretservice.SecretDiscoveryServiceServer runtimeservice.RuntimeDiscoveryServiceServer extensionconfigservice.ExtensionConfigDiscoveryServiceServer -------------- type (xxxxxxx|Endpoint)DiscoveryServiceServer interface { // The resource_names field in DiscoveryRequest specifies a list of clusters // to subscribe to updates for. StreamEndpoints(EndpointDiscoveryService_StreamEndpointsServer) error DeltaEndpoints(EndpointDiscoveryService_DeltaEndpointsServer) error FetchEndpoints(context.Context, *v3.DiscoveryRequest) (*v3.DiscoveryResponse, error) }
Impl **--------** **Cache** **SnapShot**	// Cache is a generic config cache with a watcher. type Cache interface { ConfigWatcher ConfigFetcher } // ConfigFetcher fetches configuration resources from cache type ConfigFetcher interface { // Fetch implements the polling method of the config cache using a non-empty request. Fetch(context.Context, *Request) (Response, error) }

	`` ----------------- func (cache *snapshotCache) Fetch(ctx context.Context, request *Request) (Response, error) { nodeID := cache.hash.ID(request.Node) cache.mu.RLock() defer cache.mu.RUnlock() if snapshot, exists := cache.snapshots[nodeID]; exists { // Respond only if the request version is distinct from the current snapshot state. // It might be beneficial to hold the request since Envoy will re-attempt the refresh. version := snapshot.GetVersion(request.TypeUrl) if request.VersionInfo == version { cache.log.Warnf("skip fetch: version up to date") return nil, &types.SkipFetchError{} } resources := snapshot.GetResourcesAndTTL(request.TypeUrl) out := createResponse(ctx, request, resources, version, false) return out, nil } return nil, fmt.Errorf("missing snapshot for %q", nodeID) } ```
resource (Data) Config Model	go-control-panel/envoy/config/(accesslog|bootstrap|cluster.....)

方案选择

1. 使用envoy的Extension Resource

使用envoy/core/v2/TypedExtensionConfig 对自定义的pb模型进行封装

```
type TypedExtensionConfig struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields



    // The name of an extension. This is not used to select the extension, instead
    // it serves the role of an opaque identifier.
    Name string `protobuf:"bytes,1,opt,name=name,proto3"         json:"name,omitempty"`
    // The typed config for the extension. The type URL will be used to identify
    // the extension. In the case that the type URL is *xds.type.v3.TypedStruct*
    // (or, for historical reasons, *udpa.type.v1.TypedStruct*), the inner type
    // URL of *TypedStruct* will be utilized. See the
    // :ref:`extension configuration overview
    // <config_overview_extension_configuration>` for further details.
    TypedConfig *any.Any
`protobuf:"bytes,2,opt,name=typed_config,json=typedConfig,proto3"
json:"typed_config,omitempty"`
}
```

示例代码 （server）

```
func GenerateSnapshot2() cache.Snapshot {
any, _ := anypb.New(&envoy_extensions_common_tap_v3.AdminConfig{ConfigId:
"pixiu.config.test"})
	snap, _ := cache.NewSnapshot("2",
	map[resource.Type][]types.Resource{
		resource.ClusterType:  {makeCluster2(ClusterName2)},
		resource.RouteType:    {makeRoute2(RouteName2, ClusterName2)},
		resource.ListenerType: {makeHTTPListener2(ListenerName2, RouteName2)},
		resource.ExtensionConfigType: {&core.TypedExtensionConfig{
			Name:        "pixiu.cluster",
			TypedConfig: any,
			}},
		},
	)
	return snap
}
```

示例代码 (client)

```
cdsClient :=
extensionpb.NewExtensionConfigDiscoveryServiceClient(conn)//clusterpb.NewClusterDiscove
ryServiceClient(conn)
clsRsp, err := cdsClient.FetchExtensionConfigs(context.Background(),
&discoverypb.DiscoveryRequest{
        VersionInfo: "",
        Node: &envoy_config_core_v3.Node{
          Id:            "test-id",
          Cluster:       "test-cluster",
          UserAgentName: "ma-test-agent",
        },
        //ResourceNames: []string {"pixiu.cluster"},
        TypeUrl: resource.ExtensionConfigType,

      })

if err != nil {
  panic(err)
}
log.Printf("==> result %v\n", clsRsp)
```

2. 增加pixu API,修改 Server，增加 Model

3. 完全直接做grpc server.

xDS-data-panel方案调研+POC

data-panel访问xDS-control-panel(xDS management server)，通过gRPC（早期也有restful）协议，获取CDS,LDS等配置，根据配置设置data-panel的服务。

本文主要内容：

1. 收集汇总相关技术资料
    a. data-panel-api: envoy的api定义。
    b. grpc-xds: grpc官方xDS library。
    c. go-control-panel: envoy提供golang实现xDS server。
2. POC：实现xDS-data-panel的POC，与go-control-panel通讯完成配置读取（模拟envoy）。
3. pixiu 实现xDS-data-panel 和 xDS-control-panel(admin)需要做哪些工作。

POC

POC 1: 创建client 获取xDS server的配置

1. 使用go-data-panel的example创建简单的sample。

创建的resource

```
func GenerateSnapshot2() cache.Snapshot {
snap, _ := cache.NewSnapshot("2",
map[resource.Type][]types.Resource{
resource.ClusterType:  {makeCluster2(ClusterName2)},
resource.RouteType:    {makeRoute2(RouteName2, ClusterName2)},
resource.ListenerType: {makeHTTPListener2(ListenerName2,
RouteName2)},
},
)
return snap
}
```

创建server (:18000)

```
.....
.....
snapshot2 := example.GenerateSnapshot2()
if err := snapshot.Consistent(); err != nil {
l.Errorf("snapshot inconsistency: %+v\n%+v", snapshot2, err)
os.Exit(1)
}


// Add the snapshot to the cache
if err := cache.SetSnapshot(context.Background(), nodeID, snapshot2); err
!= nil {
l.Errorf("snapshot error %q for %+v", err, snapshot2)
os.Exit(1)
}


.....
.....
ctx := context.Background()
cb := &test.Callbacks{Debug: l.Debug}
srv := server.NewServer(ctx, cache, cb)
example.RunServer(ctx, srv, port)
```

1. 使用envoy xDS API(gRPC)创建client，获取cluster的配置。

通过基本通过的方式连接grpc

```
creds := insecure.NewCredentials()
if *xdsCreds {
log.Println("Using xDS credentials...")
var err error
if creds, err =
xdscreds.NewClientCredentials(xdscreds.ClientOptions{FallbackCreds:
insecure.NewCredentials()}); err != nil {
log.Fatalf("failed to create client-side xDS credentials: %v", err)
}
}
conn, err := grpc.Dial(*target, grpc.WithTransportCredentials(creds))
if err != nil {
log.Fatalf("grpc.Dial(%s) failed: %v", *target, err)
}
defer conn.Close()
```

查询cluster 配置，并进行读取信息

```
cdsClient := clusterpb.NewClusterDiscoveryServiceClient(conn)
clsRsp, err := cdsClient.FetchClusters(context.Background(),
&discoverypb.DiscoveryRequest{
VersionInfo:   "",
Node:          &envoy_config_core_v3.Node{
Id:                "test-id",
Cluster:             "test-cluster",
UserAgentName:        "ma-test-agent",


},
TypeUrl:       resource.ListenerType,


})
if err != nil {
panic(err)
}
log.Printf("==> result %v\n", clsRsp)
for _, _resource := range clsRsp.Resources {
clsConfig := cluster.Cluster{}
err := _resource.UnmarshalTo(&clsConfig)
if err != nil {
panic(err)
}
log.Printf("==> resource %v\n", clsConfig.LoadAssignment)
}
```

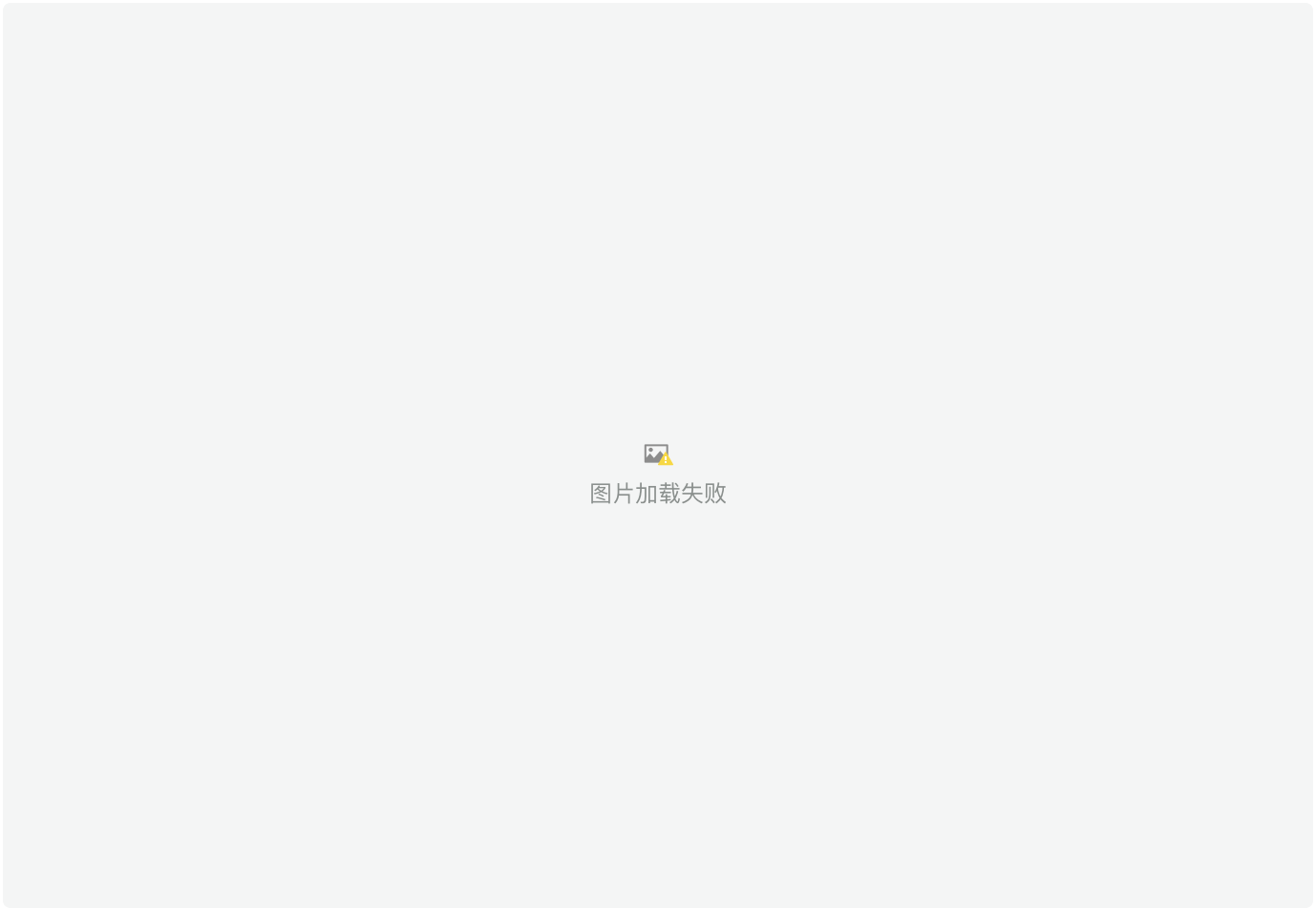
POC 2: watch Cluster配置，动态更新client配置（todo).

使用xDS需要完成的工作：

1. 使用xDS 要对齐 envoy的xDS API。
2. 要扩展xDS API：有些配置是标准API无法支持的,如dubbo的endpoint配置(todo, 需要调研一下扩展）。
3. pixiu各种资源动态配置。
4. admin使用go-control-panel lib，创建xDS server，对接界面。

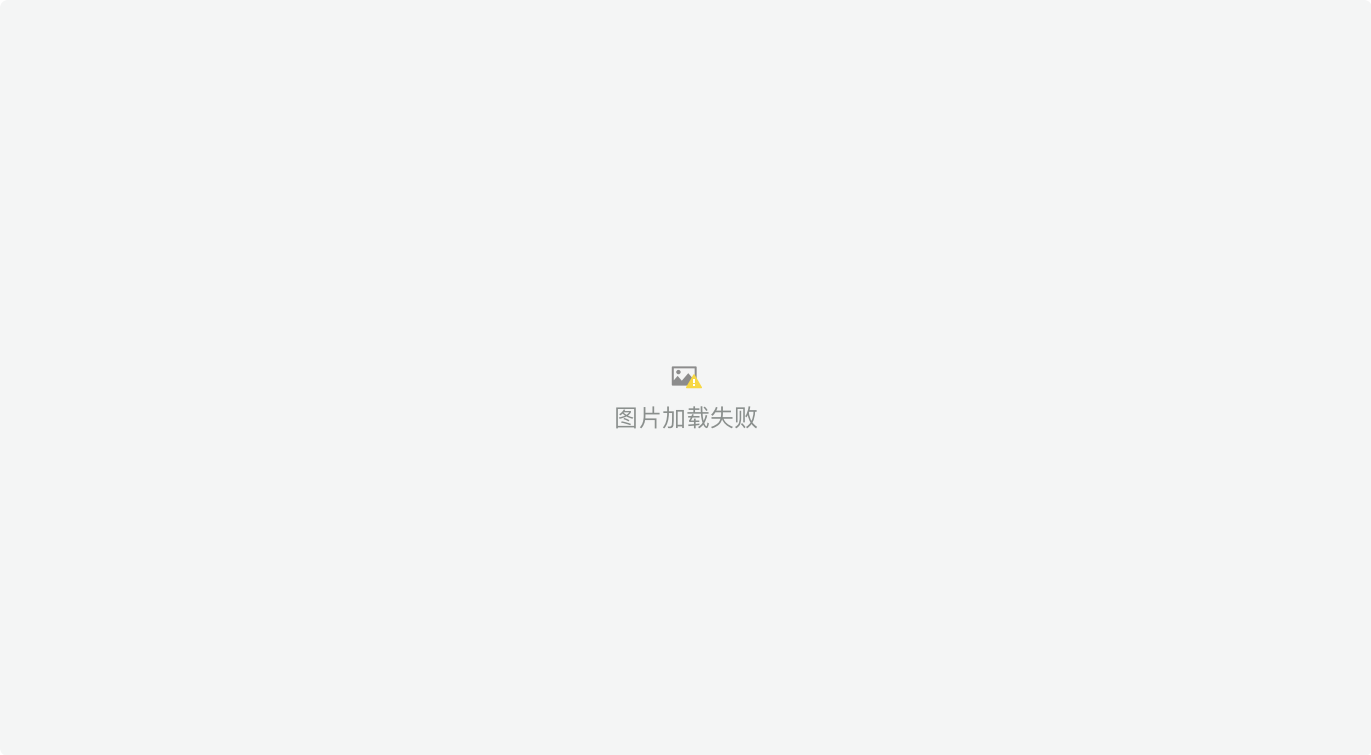
参考资料：

1. https://github.com/grpc/grpc-go/tree/master/examples/features/xds **grpc-go的xDS实现, 为了帮助应用层使用xDS 进行服务调用，而不是通过envoy来建立mesh。**
2. https://github.com/grpc/proposal/blob/master/A27-xds-global-load-balancing.md#xdsclient-and-bootstrap-file **grpc-xDS 技术架构**



1. https://github.com/envoyproxy/data-plane-api **v2 API definitions as a standalone repository**

2. https://blog.envoyproxy.io/the-universal-data-plane-api-d15cec7a **xDS怎么来的。**
3. https://www.youtube.com/watch?v=IbcJ8kNmsrE **grpc-xds视频介绍**



图：基于sidecar的mesh支持（envoy）

图： 基于gRPC library的mesh支持

相关项目

- https://github.com/envoyproxy/envoy/tree/main/api – canonical read/write home for the APIs.
- https://github.com/envoyproxy/data-plane-api – read-only mirror of https://github.com/envoyproxy/envoy/tree/main/api, providing the ability to consume the data plane APIs

pixiu 支持类 xDS 协议远程配置

一、目标

admin 提供 grpc server 用于远程配置管理，pixiu 监听消息进行对应配置变更，类似于 pilot 和 envoy 的关系

二、现状

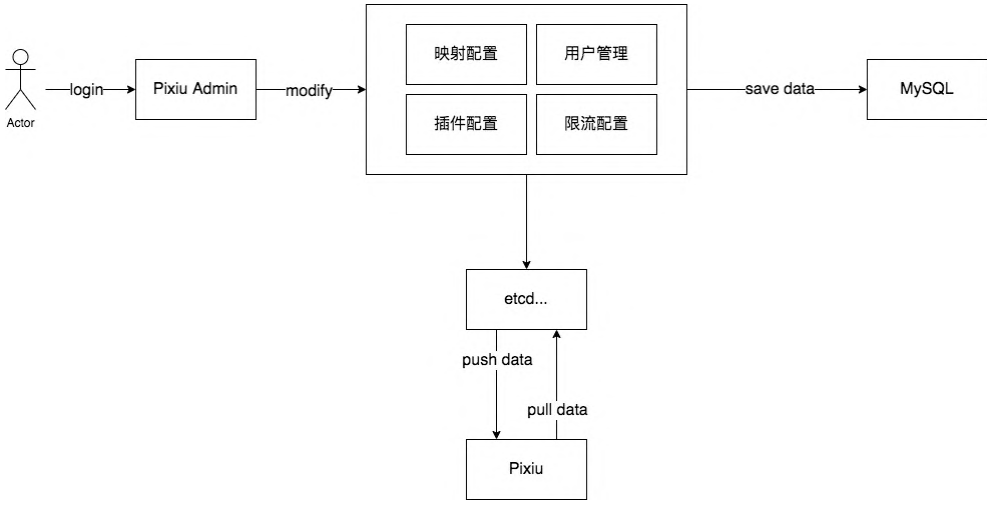
2.1 pixiu

pixiu 目前本地yaml文件配置，类似于 envoy 配置，但是细节有些区别

```yaml
static_resources:
  listeners:
    - name: "net/http"
      address:
        socket_address:
          protocol_type: "HTTP"
          address: "0.0.0.0"
          port: 8888
      filter_chains:
        - filter_chain_match:
          domains:
            - api.dubbo.com
            - api.pixiu.com
          filters:
            - name: dgp.filter.httpconnectionmanager
              config:
                route_config:
                  routes:
                    - match:
                        prefix: "/user"
                      route:
                        cluster: "user"
                        cluster_not_found_response_code: 505
                http_filters:
                  - name: dgp.filter.http.httpproxy
                    config:
                  - name: dgp.filter.http.response
                    config:

  clusters:
    - name: "user"
      lb_policy: "lb"
      endpoints:
        - id: 1
          socket_address:
            address: 127.0.0.1
            port: 1314

```

目前基础配置不支持远程，只有dubbo 协议解析部分的api_config配置支持admin的远程配置。

2.2 admin

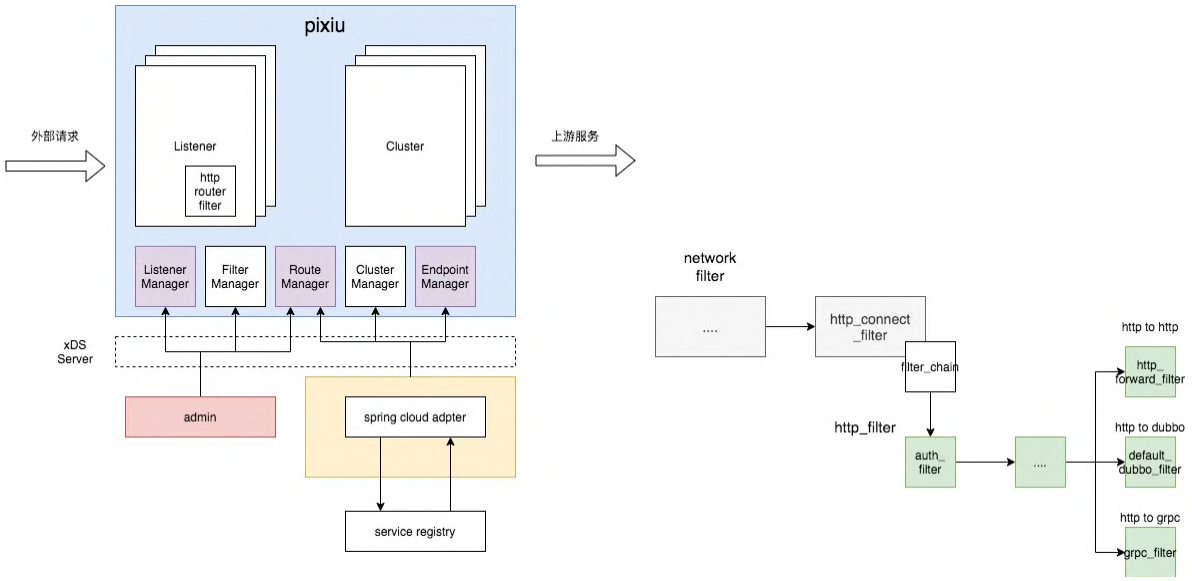


目前是将配置数据存放在 etcd，将用户等数据存放在 mysql，pixiu 从 etcd中获取key-value 数据并监听对应路径数据。

遇到一些问题：

- key-value不容易满足复杂数据格式
- etcd 的 event 只有set 和 rm，编码上不容易处理

三、方案



类似上图，不过 admin 和 xDS server 可以放在一起。pixiu 从 xDS server 中获取配置数据，并且 spring cloud adapter 和 dubbo registry 后续可以迁移到admin处

类似 envoy, static_resource 的cluster 定义 xDS server 信息，然后dynamic_resource 中配置对应的监听对象

YAML

```yaml
# envoy configuration
dynamic_resources:
  ads_config:
    api_type: GRPC
    transport_api_version: V3
    grpc_services:
    - _grpc:
        cluster_name: xds_cluster
  cds_config:
    resource_api_version: V3
    ads: {}
  lds_config:
    resource_api_version: V3
    ads: {}

static_resources:
  clusters:
  - type: STRICT_DNS
    typed_extension_protocol_options:
      envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
        "@type": type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProtocolOptions
        explicit_http_config:
          http2_protocol_options: {}
    name: xds_cluster
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: my-control-plane
                port_value: 18000
```

https://www.envoyproxy.io/docs/envoy/latest/start/sandboxes/dynamic-configuration-control-plane

分工：

- admin 侧，提供 xDS api，并提供增删改查配置的api接口
- pixiu 侧，引入static_resource，监听xDS server 进行对应的配置变更

四、问题

- 如何和api_config进行统一
- 统一的API抽象和多样性的功能之间的冲突
  - listener
  - 后续多协议，filter可能会有几套，估计是不同的抽象，则需要不同的配置

五、参考

待解决问题

一、Listener监听

直接使用 http server 监听端口，所以只支持七层协议的HTTP，在request和response层面进行处理，是否应该处理四层tcp协议，处理帧，方便后续扩展其他七层协议

改为第四层上进行监听，需要socket 编程。虽然工作量可能会较大，但是如果想要在后期可以支持各种grpc，amqp等协议，还是需要做出这个改动。现阶段需要定一下做的时间点；
Owner：张天

二、整体热更新机制

梦超的filter的reload提供了更新能力，但是整体配置 listener、cluster、router(api_config目前支持)不支持配置远程配置和更新替换能力，后续流量动态管理功能都依赖于此。

冯振宇先把api_config 作为全局变量搞个pr跟大家商量看看

三、可选Filter-chain，Filter实例的数量问题，Filter的共用配置

目前同一个httpconnectionmanager下的所有请求都会走相同的http-filter-chain，是否需要根据route走不同的filter。
如果需要，则这些 filter 是多个实例还是一个实例，这些filter的配置是共用的，还是要分别配置

```
func addFilter(ctx *h.HttpContext, api router.API) {
        ctx.AppendFilterFunc(extension.GetMustFilterFunc(constant.MetricFilter),
                extension.GetMustFilterFunc(constant.RecoveryFilter), extension.GetMustFilterFunc(constant.TimeoutFilter))
        alc := config.GetBootstrap().StaticResources.AccessLogConfig
        if alc.Enable {
                ctx.AppendFilterFunc(extension.GetMustFilterFunc(constant.AccessLogFilter))
        }
        ctx.AppendFilterFunc(extension.GetMustFilterFunc(constant.RateLimitFilter))

        switch api.Method.IntegrationRequest.RequestType {
        // TODO add some basic filter for diff protocol
        case fc.DubboRequest:

        case fc.HTTPRequest:
                httpFilter(ctx, api.Method.IntegrationRequest)
        }
        // load plugins
        pluginsFilter := plugins.GetAPIFilterFuncsWithAPIURL(ctx.Request.URL.Path)
        ctx.AppendFilterFunc(pluginsFilter.Pre...)

        ctx.AppendFilterFunc(header.New().Do(), extension.GetMustFilterFunc(constant.RemoteCallFilter))
        ctx.BuildFilters()

        ctx.AppendFilterFunc(pluginsFilter.Post...)
        ctx.AppendFilterFunc(extension.GetMustFilterFunc(constant.ResponseFilter))
}
```

四、filter更清晰的细分

目前有一些隐含的规则需要实际用户知晓，比如：

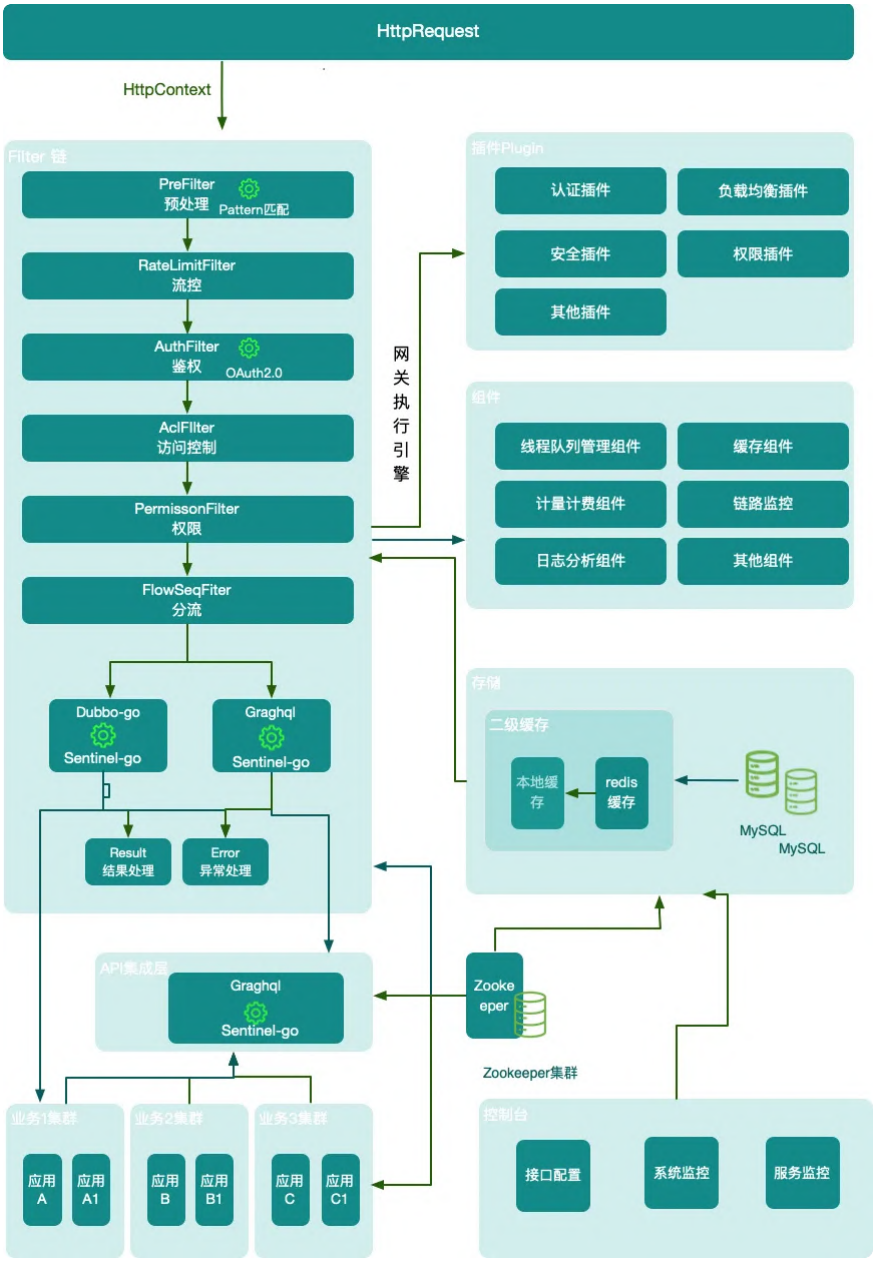
- 目前httpproxy、grpcproxy、dubboproxy分别对应不同协议，但是都需要再加一个responsefilter负责返回响应。
- 如果如果不配置httpproxy、grpcproxy、dubboproxy等协议，实际无法进行转发
- 单一的Handle抽象，必须显示调用next，以next的前后区分pre还是post 处理。

owner: 吕梦超

五、route 的优化

振宇大佬提议使用字典树来对route匹配path进行优化

参考


HttpRequest
HttpContext
Filter 链
PreFilter
预处理
Pattern匹配
RateLimitFilter
流控
AuthFilter
鉴权
OAuth2.0
AclFilter
访问控制
PermissonFilter
权限
FlowSeqFiter
分流
Dubbo-go
Sentinel-go
Graghql
Sentinel-go
Result
结果处理
Error
异常处理
网关执行引擎
插件Plugin
认证插件
负载均衡插件
安全插件
权限插件
其他插件
组件
线程队列管理组件
缓存组件
计量计费组件
链路监控
日志分析组件
其他组件
存储
二级缓存
本地缓存
redis缓存
MySQL
MySQL
API集成层
Graghql
Sentinel-go
Zooke eper
Zookeeper集群
业务1集群
应用A
应用A1
业务2集群
应用B
应用B1
业务3集群
应用C
应用C1
控制台
接口配置
系统监控
服务监控

🚀 Dubbo-go-pixiu 与 SpringCloud 服务发现自动映射路由

项目排期

	功能点	Owner	开始时间	
项目框架改在设计	pixiu接入 SpringCloud 插件化设计方案	岳枫博		
	启动加载流程插件化解耦设计：使用引导，初始化加载各个实现（饥饿加载）	岳枫博		
	请求处理流程插件化解耦设计：服务发现Service实现统一放到服务引导启动阶段初始化配置；流程主要负责handle request；特殊懒加载诉求除外	岳枫博		
貔貅服务启动阶段	sc获取所有服务实例信息接口			
	sc获取指定服务实例信息根据serviceId接口			
	sc服务注册-将自己注册到注册中心			
	sc服务心跳实现，刷新缓存服务实例信息			

	自动隐射本地配置的服务信息、过滤器等			
貔貅请求代理阶段	服务发现，兼容本地配置的services信息及sc 服务发现			
	~~Canary 灰度流程逻辑，根据权重获取一个serverInfo，再从服务信息中过滤得到符合的 service instance 集合~~			
	负载均衡实现，轮询负载均衡获取一个 instance			
	protocol: invoke remote，HTTP 远程调用			

Java

```java
public interface DiscoveryClient extends Ordered {

    /**
     * Default order of the discovery client.
     */
    int DEFAULT_ORDER = 0;

    /**
     * A human-readable description of the implementation, used in HealthIndicator.
     * @return The description.
     */
    String description();

    /**
     * Gets all ServiceInstances associated with a particular serviceId.
     * @param serviceId The serviceId to query.
     * @return A List of ServiceInstance.
     */
    List<ServiceInstance> getInstances(String serviceId);

    /**
     * @return All known service IDs.
     */
    List<String> getServices();

    /**
     * Can be used to verify the client is valid and able to make calls.
     * <p>
     * A successful invocation with no exception thrown implies the client is able to make
     * calls.
     * <p>
     * The default implementation simply calls {@link #getServices()} - client
     * implementations can override with a lighter weight operation if they choose to.
     */
    default void probe() {
        getServices();
    }

    /**
     * Default implementation for getting order of discovery clients.
     * @return order
```

```
     */
    @Override
    default int getOrder() {
        return DEFAULT_ORDER;
    }

}
```

🌏 项目介绍

概述

该项目最初是因为涂鸦有场景需要 网关通过服务发现自动映射调用 HTTP Server 服务，社区当时也希望做 SpringCloud 的支持，所以刚好也能支持涂鸦的诉求。后社区多人与涂鸦张永红等拉会确认后，了解涂鸦的现状与目的：业务方自己申请了域名来暴露http服务，想要把服务迁移到网关统一管理。简单来说，想要网关去管理服务域名实现路由即可。

所以，经与社区 张训 确认，本项目核心聚焦于对 SpringCloud 的支持上，实现 SpringCloud 服务发现自动映射路由。

项目边界：解决 Pixiu 与 SpringCloud 的支持能力，只需要解决 Pixiu 的服务注册发现。区别于 Dubbo-go 与 SpringCloud 打通（这需要另起一个项目）。

项目背景

1. ~~涂鸦真实使用场景的需要，调用链有 Client(http) → Serve(http)，也有 Client(http) → Pixiu-Gateway →Serve(dubbo) 的应用场景~~。
2. Pixiu 当前对 SpringCloud 服务发现并没有很好的支持。
3. 有 Issure 提到 Dubbogo 对 SpringCloud 的支持。
   – 替代springcloud gateway中的一个微服务 #684
   替代springcloud gateway中的一个微服务 · Issue #684 · apache/dubbo-go
   – How to call the service of spring cloud #833
   How to call the service of spring cloud · Issue #833 · apache/dubbo-go
   @zouyx ：There is not support call the spring cloud service .Because there is some

problems in go-client of eureka .
@AlexStocks ：dubbogo spring cloud client 无法保证数据一致性

项目目标

~~**实现 Pixiu-Gateway 与 Eureka、Nacos、Consul、Zookeeper 的 服务发现自动映射路由，**支持与 SpringCloud 互相调用。同时也可以支持一些最佳实践，比如基于网关的能力，Server 层 Dubbogo 服务可直接替换 SpringCloud 服务或其他最佳实践场景。~~

实现 Pixiu 与原生 SpringCloud 项目打通，基于 Pixiu 的服务发现自动映射路由调用 SpringCloud 服务。

🧭 项目方案

- @ChengJiapeng @PhilYue

项目现状

Pixiu 网关仓库地址：https://github.com/apache/dubbo-go-pixiu
Pixiu Admin地址：dubbogo/pixiu-admin

项目结构

```
.
├── CHANGE.md
├── LICENSE
├── Makefile
├── NOTICE
├── README.md
├── README_CN.md
├── before_ut.sh
├── before_validate_license.sh
├── build
├── cmd
│   └── pixiu
├── configs
│   ├── api_config.yaml
│   ├── conf.yaml
│   └── log.yml
├── docs
├── dubbo-go-pixiu
├── go.mod
├── go.sum
├── integrate_test.sh
├── logs
├── pkg
│   ├── client
│   ├── common
│   ├── config
│   ├── context
│   ├── filter
│   ├── initialize
│   ├── logger
│   ├── model
│   ├── pixiu
│   ├── pool
│   ├── registry
│   ├── router
│   └── service
├── samples
│   ├── admin
│   ├── dubbogo
│   ├── http
│   └── plugins
├── start.sh
└── start_integrate_test.sh
```

当前有注册中心的代码实现

有注册中心的代码实现，待确认当前情况 @张天 @张训

pkg/registry/load.go

```go
// Loader this interface defined for load services from different kinds registry, such as nacos,consul,zookeeper.
type Loader interface {
    // LoadAllServices load all services registered in registry
    LoadAllServices() ([]*common.URL, error)
    // GetCluster get the registry name
    GetCluster() (string, error)
}
```

项目功能点

- [ ] **服务发现自动映射路由：网关注册，获取服务信息，自动映射路由**
- [ ] 服务发现配置化手动映射路由
- [ ] **服务自动映射开关：关闭从注册中心获取到的服务路由（屏蔽单个、多个、或全部应用）**
- [ ] 服务网关路由通过前缀 `/api` 区分API路由还是内容路由
- [x] 静态URL手动映射路由
- [ ] 动态重新加载路由配置
- [ ] 服务超时
- [ ] **远程调用协议支持：rest**
- [ ]

🔧 技术方案

Pixiu 整体架构

https://www.processon.com/view/link/60db20b61e08532a43c572f4

请求流程

https://www.processon.com/view/link/60e17cb00e3e745b08a608bd

处理请求流程（现状）

https://www.processon.com/view/link/60e2f889f346fb04d2db5fd8

RD视角方案TODO

https://www.processon.com/view/link/60f6d766e401fd09d480bbc9

自动映射路由信息

todo

项目核心功能

本项目优先基于 Zookeeper 注册中心，打通与 SpringCloud 服务的依赖。

实现与 SpringCloud 服务发现自动映射路由能力，分为几个阶段：

1、注册。Pixiu 作为Client，注册到 与SpringCloud应用统一的注册中心

2、发现。Pixiu 作为调用方，订阅SpringCloud应用信息、地址

3、调用&管理。Pixiu 作为网关，自动映射 SpringCloud 应用路由信息，实现 rest 协议调用，同时支持网关的其他特性能力（Loadbalance，Ratelimit ...）

服务注册

todo 技术方案细化

注册能力基本能力：

- Register：服务注册，提供自身的元数据，比如IP地址、端口，运行状况指示符URL，主页
- Renew：服务续约，每隔30秒发送一次心跳来续约，通过续约来告知注册中心，该服务仍然存在，没有出现问题。
- Fetch Registries：获取注册列表信息
- Cancel：服务下线，客户端实例信息将从服务器的实例注册表中删除
- Eviction：服务剔除，心跳超时（未送达），注册中心将该服务实例从服务注册列表删除，即服务剔除

SpringCloud 服务注册接口类：

Java

```java
public interface ServiceRegistry<R extends Registration> {

    /**
     * Registers the registration. A registration typically has information about an
     * instance, such as its hostname and port.
     * @param registration registration meta data
     */
    void register(R registration);

    /**
     * Deregisters the registration.
     * @param registration registration meta data
     */
    void deregister(R registration);

    /**
     * Closes the ServiceRegistry. This is a lifecycle method.
     */
    void close();

    /**
     * Sets the status of the registration. The status values are determined by the
     * individual implementations.
     * @param registration The registration to update.
     * @param status The status to set.
     * @see org.springframework.cloud.client.serviceregistry.endpoint.ServiceRegistryEndpoint
     */
    void setStatus(R registration, String status);

    /**
     * Gets the status of a particular registration.
     * @param registration The registration to query.
     * @param <T> The type of the status.
     * @return The status of the registration.
     * @see org.springframework.cloud.client.serviceregistry.endpoint.ServiceRegistryEndpoint
     */
    <T> T getStatus(R registration);

}
```

服务发现

todo 技术方案细化

实现 SpringCloud 基于注册中心 **Zookeeper**，Consul，Nacos，Eureka 的服务发现能力

- 多注册中心注册订阅，优先实现 **Zookeeper** 的服务注册发现。
- 调用协议扩展，rest

SpringCloud 服务发现接口类：

```java
public interface DiscoveryClient extends Ordered {

    /**
     * Default order of the discovery client.
     */
    int DEFAULT_ORDER = 0;

    /**
     * A human-readable description of the implementation, used in HealthIndicator.
     * @return The description.
     */
    String description();

    /**
     * Gets all ServiceInstances associated with a particular serviceId.
     * @param serviceId The serviceId to query.
     * @return A List of ServiceInstance.
     */
    List<ServiceInstance> getInstances(String serviceId);

    /**
     * @return All known service IDs.
     */
    List<String> getServices();

    /**
     * Default implementation for getting order of discovery clients.
     * @return order
     */
    @Override
    default int getOrder() {
        return DEFAULT_ORDER;
    }

}
```

自动映射路由

todo

注册中心功能对比

	Nacos	Eureka	Consul	Zookeeper
一致性协议	CP+AP	AP	CP	CP
健康检查	TCP/HTTP/MYSQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	Keep Alive
负载均衡策略	权重/metadata/Selector	Ribbon	Fabio	—
雪崩保护	有	有	无	无
自动注销实例	支持	支持	不支持	支持
访问协议	HTTP/DNS	HTTP	HTTP/DNS	TCP
监听支持	支持	支持	支持	支持
多数据中心	支持	支持	支持	不支持
跨注册中心同步	支持	不支持	支持	不支持
SpringCloud集成	支持	支持	支持	支持
Dubbo集成	支持	不支持	不支持	支持
K8S集成	支持	不支持	支持	不支持

Eureka 服务注册信息

JSON


```
[
  {
    "uri": "http://172.18.153.43:9000",
    "secure": false,
    "metadata": {

    },
    "instanceInfo": {
      "instanceId": "172.18.153.43:spring-cloud-producer:9000",
      "app": "SPRING-CLOUD-PRODUCER",
      "appGroupName": null,
      "ipAddr": "172.18.153.43",
      "sid": "na",
      "homePageUrl": "http://172.18.153.43:9000/",
      "statusPageUrl": "http://172.18.153.43:9000/info",
      "healthCheckUrl": "http://172.18.153.43:9000/health",
      "secureHealthCheckUrl": null,
      "vipAddress": "spring-cloud-producer",
      "secureVipAddress": "spring-cloud-producer",
      "countryId": 1,
      "dataCenterInfo": {
        "@class": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo"
,
        "name": "MyOwn"
      },
      "hostName": "172.18.153.43",
      "status": "UP",
      "leaseInfo": {
        "renewalIntervalInSecs": 30,
        "durationInSecs": 90,
        "registrationTimestamp": 1625476088459,
        "lastRenewalTimestamp": 1625477588519,
        "evictionTimestamp": 0,
        "serviceUpTimestamp": 1625476087664
      },
      "isCoordinatingDiscoveryServer": false,
      "metadata": {

      },
      "lastUpdatedTimestamp": 1625476088459,
      "lastDirtyTimestamp": 1625476087596,
      "actionType": "ADDED",
      "asgName": null,
      "overriddenStatus": "UNKNOWN"
    },
```

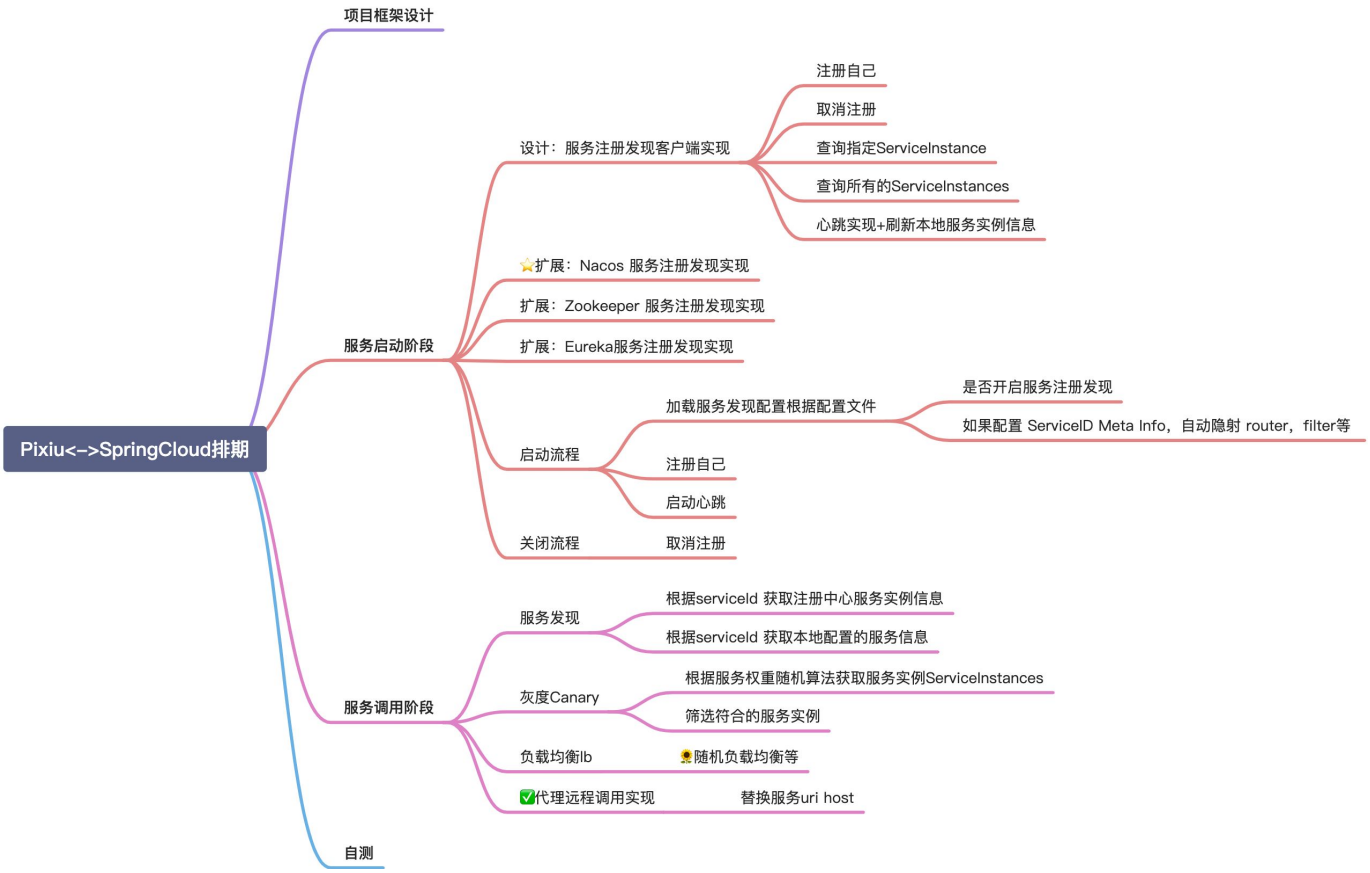
```
        "host": "172.18.153.43",
        "port": 9000,
        "serviceId": "SPRING-CLOUD-PRODUCER"
    }
]
```

📈项目排期

TODO:

- SpringCloud 集成插件化 shenyu ==》岳枫博
- 启动阶段，监听注册中心目录变化 ==》岳枫博
- Pixiu框架设计插件化 ==》张天主R调研，周会讨论

从设计上，流程阶段边界清晰，可以在不同的流程处挂载钩子，兼容多个注册中心：



	功能点	Owner	开始时间	
项目框架设计				
貔貅服务启动阶段	服务注册-将自己注册到注册中心，兼容多个			
	服务心跳实现（周期性任务，考虑加锁，或者任务队列实现）			
	缓存所有的注册中心的服务到本地注册表，周期性任务与心跳一起			
	服务发现接口提供：获取所有的服务，获取指定的服务 By ServiceID			
貔貅请求代理阶段				
	服务发现- discovery by serviceId ，获取所有的服务信息			
	Canary 灰度流程逻辑，根据权重获取一个serverInfo ，再从服务信息中过滤得到符合的 service instance 集合			
	负载均衡实现，轮询负载均衡获取一个 instance			

	protocol: invoke remote , HTTP 远程调用			

参考资料

- Dubbo Spring Cloud https://github.com/alibaba/spring-cloud-alibaba/wiki/Dubbo-Spring-Cloud
- go-eureka-client https://github.com/ArthurHlt/go-eureka-client

grpc proxy

目标

pixiu 支持代理 grpc 协议，为后续功能组件提供基础协议功能

问题

一、listener 目前仅支持 http协议，需要将其进行扩充，支持多种协议。

- 监听 socket 接口，其他协议在 tcp 基础上进行包解析
- 提供 protocl 字段，根据字段值使用不同API进行接口监听

二、filter 等抽象如何复用
目前 filter 使用 httpcontext，是否要换回顶层 context 抽象。

三、stream模式
unary ， post filter 是否要对 stream 模式的 server 返回进行处理

案例

可以参考开源项目 https://github.com/mwitkow/grpc-proxy

https://stackoverflow.com/questions/52002623/golang-tcp-server-how-to-write-http2-data
https://undertow.io/blog/2015/04/27/An-in-depth-overview-of-HTTP2.html

pixiu 支持 http to grpc 特性

一、需求

1. 本阶段实现外部http/json请求通过 pixiu 调用内部的 grpc server unary服务
2. 【后续】外部 grpc client 和 内部 grpc server 的proxy，支持 unary，stream等

二、原理

参考：

- https://github.com/apache/apisix/blob/baf843403461883c1334e63d15a6bb3622c31940/docs/zh/latest/plugins/grpc-transcode.md apisix 的http to grpc 功能，需要上传 proto 配置
- https://github.com/fullstorydev/grpcurl 类似curl的grpccurl命令行工具，主要参考，从 grpc reflect server 中获取 proto 配置
- https://github.com/Windfarer/dynamic-protobuf-example/blob/master/main.go 代码构造 proto 配置
- https://github.com/jhump/protoreflect 主要使用库
- https://github.com/mwitkow/grpc-proxy grpc proxy

根据调研，需要 proto 定义才能发送 grpc 请求的，只不过获取 proto 定义的方式有三种：

- 在网关映射配置时，直接上传 proto 定义，比如 apisix
- grpc server 开启了反射功能，可以通过请求从 server 处获取 proto 定义，比如 grpccurl
- 可以通过代码构造 proto 定义，所以可以通过 api_config 中约定特殊配置进行生成。

实现的 demo

https://github.com/ztelur/dubbo-go-pixiu/blob/feature-support-grpc/pkg/client/grpc/grpc_test.go

和 dubbo-go grpc 泛化调用关系

dubbo-go 的 grpc 泛化调用目前是指 POJO 不同协议序列化场景，和 pixiu 支持 http 转换为 grpc 并不合拍。

三、pixiu 代码实现

TODO

3.1 api_config 配置

```yaml
  - path: '/api/v1/test-dubbo/user'
    type: restful
    description: user
    timeout: 100ms
    plugins:
      pre:
        pluginNames:
          - rate limit
          - access
      post:
        groupNames:
          - group2
    methods:
      - httpVerb: GET
        onAir: true
        timeout: 1000ms
        requestType: grpc
        destination: /pb.UserService/checkPassword
        inputType: LoginRequest
        outputType: LoginResponse
        queryStrings:
        - name: name
            required: true
        - password: password
            required: true

```

3.2 client/dubbo

类似网关系统

自动发现服务

服务组流控鉴权配置

- 上下线设置服务是否暴露出去
- 访问级别, 设置鉴权方式
- 流控保护, 接口限流

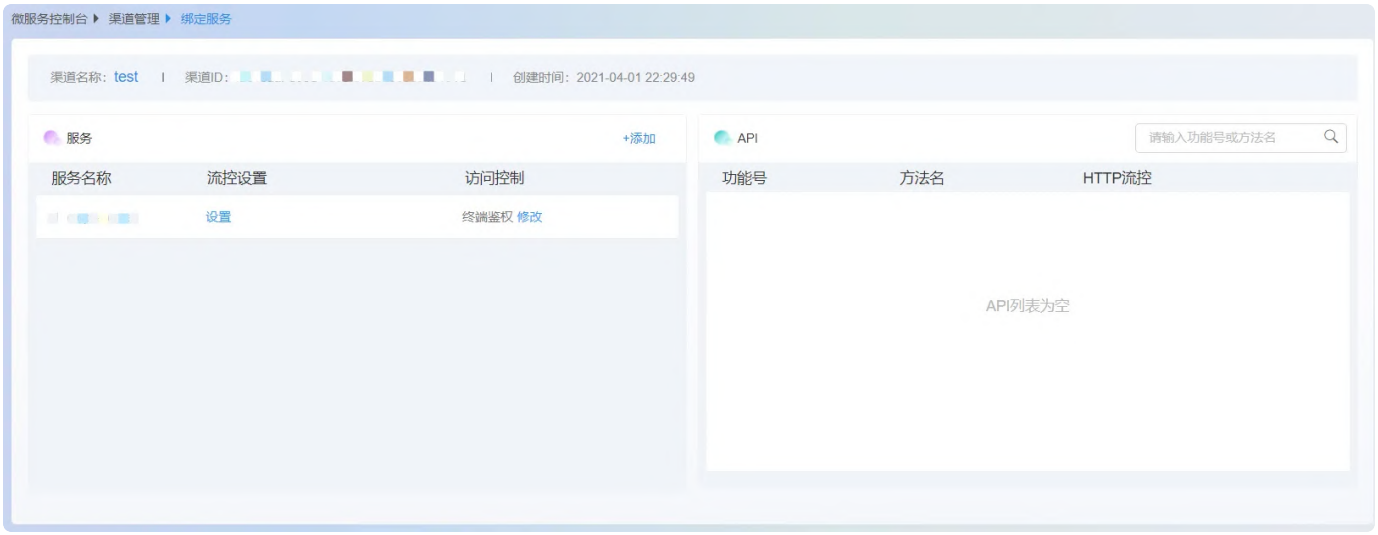
单接口流控设置

添加单接口虚拟URL 映射

- url 映射分自动映射和手动映射
- 自动映射会根据服务 版本、组、接口信息， 自动拼装一个URL

appKey 管理

appKey 服务授权


微服务控制台 ▸ 渠道管理 ▸ 绑定服务
渠道名称：test
渠道ID：
创建时间：2021-04-01 22:29:49
服务
+添加
服务名称
流控设置
访问控制
设置
终端鉴权 修改
API
请输入功能号或方法名
功能号
方法名
HTTP流控
API列表为空

pixiu 功能简介和问题

一、功能

1.1 协议代理或转换

- http to dubbo 目前支持 http 协议和 dubbo 协议，支持接口级别服务注册模式，使用 api_config 配置映射关系
- http to grpc unary 模式，使用 google cloud http to grpc transcode 默认规约进行参数转换
- http to http http 代理

1.2 filter 功能

- cors 跨域
- csrf 跨站
- metric 数据聚合 opentelemetry
- ratelimit 限流
- proxy rewrite http 请求修改(删除 path 前缀)
- tracing tracing 分布式链路追踪 jaeger
- seata 分布式事务

1.3 生态集成

- 通过 dubbo 服务注册中心自动生成路由
  - zk
- 通过 springcloud 服务注册中心自动生成路由和集群信息
  - nacos

1.4 eventmesh

- 通过 webhook 进行消息的接收和发送

1.5 dbmesh

- mysql 协议解析

二、问题

2.1 api_config 和 route 问题

现状

api_config 用于 http to dubbo 功能，pixiu 的 filter 配置如下所示。

```yaml
- name: dgp.filter.http.apiconfig
    config:
    path: $PROJECT_DIR/pixiu/api_config.yaml
- name: dgp.filter.http.dubboproxy
    config:
    dubboProxyConfig:
        registries:
        "zookeeper":
            protocol: "zookeeper"
            timeout: "3s"
            address: "127.0.0.1:2181"
            username: ""
            password: ""
        timeout_config:
        connect_timeout: 5s
        request_timeout: 5s
- name: dgp.filter.http.response
    config:
```

- 其中 dpg.filter.http.apiconfig 是用于 http to dubbo 的路由和元信息映射规则
- dgp.filter.http.dubboproxy 则是用于最终发送 dubbo 泛化调用请求(在dubbo协议时)，其下为 dubbo 注册中心的相关配置

apiconfig filter 需要指定本地路径下的 api_config 配置文件

```yaml
resources:
  - path: '/api/v1/test-dubbo/user'
    type: restful
    description: user
    methods:
      - httpVerb: POST
        enable: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
        integrationRequest:
          requestType: dubbo
          mappingParams:
            - name: requestBody._all
              mapTo: 0
              mapType: "object"
          applicationName: "UserProvider"
          interface: "com.dubbogo.pixiu.UserService"
          method: "CreateUser"
          group: "test"
          version: 1.0.0
          clusterName: "test_dubbo"
```

如上配置，表示了路径为 /api/v1/test-dubbo/user 且 method 为 POST 的http请求要转换成调用 com.dubbogo.pixiu.UserServic 的 CreateUser 方法的 dubbo 调用，还包括 mappingParams 等参数转换规则。

对 http to dubbo 的 samples 可以参考 /samples/dubbogo/simple下的例子，有更多高级别用法。

而更加通用的 route 则用于 http to http 和 http to grpc。由 routes 给出路由规则，然后 grpcproxy 或者 httpproxy 进行实际转发。

```yaml
config:
  route_config:
    routes:
    - match:
        prefix: "/api/v1"
        route:
        cluster: "test-grpc"
        cluster_not_found_response_code: 505
  http_filters:
    - name: dgp.filter.http.grpcproxy
    config:
        path: $PROJECT_DIR/proto
    - name: dgp.filter.http.response
    config:
....

  clusters:
    - name: "test-grpc"
      lb_policy: "RoundRobin"
      endpoints:
        - socket_address:
            address: 127.0.0.1
            port: 50001
            protocol_type: "GRPC"
```

如上配置，routes 表示将 path 前缀为 /api/v1 的 http 请求转发到 test-grpc 集群。而 test-grpc 集群下有一地址为 127.0.0.1的 endpoint。

更加具体的 samples 可以参考 samples/http 下的例子

问题

- api_config 和 route 导致有两套路由规则，诸如前缀树路由性能优化等工作要分别为二者进行适配；
- api_config 其实缺少了对 cluster 信息的维护，交由 dubbo 自身机制去路由到具体 upstream 和负载；
- route 缺少参数转换功能；

解决方案?

- 引入 http to dubbo 默认规约 https://www.yuque.com/docs/share/4f82b71e-fa2b-4e65-8007-ba3afe2a1740?# 后，是否可以将 api_config 去掉?
- 增加 filter，进行对应的 request 参数转换功能

- 是否可以不使用 dubbo 的泛化调用
  - 原因
    - refereconfig 初始化时用时去查询服务注册中心获取 upstream 的数据，是每次调用都再次查询，upstream 无法精确控制，效率上可能会下降
    - 治理功能全部依赖 dubbo 机制，pixiu 无法深入
  - 反驳
    - 自己维护一套和dubbo兼容的请求生成机制的复杂性

2.2 filter 的定义

现状

目前模仿 envoy，pixiu 包括了 network filter 和 http filter，具体接口定义可参考 pkg/common/extension/filter/filter.go

- network filter 是指入度网络协议相关的 filter，比如专门接受 http 请求的 httpconnectionmanager，它会挂在 listener 的 filter_chains 下；
- http filter 单指和 http 请求相关的 filter，可以看做是 httpconnectionmanager 下的子 filter 种类。

二者的关系和配置案例如下所示

YAML

```yaml
listeners:
  - name: "net/http"
    address:
      socket_address:
          ....
    filter_chains:
      - filter_chain_match:
        filters:
          - name: dgp.filter.httpconnectionmanager
              http_filters:
                - name: dgp.filter.http.grpcproxy
                  config:
                    path: $PROJECT_DIR/proto
                - name: dgp.filter.http.response
                  config:
```

对于 http_filters 的定义如下

```go
// HttpFilter describe http filter
    HttpFilter interface {

        // 处理请求相关

        // PrepareFilterChain add filter into chain
        // 请求将要过来时调用，用来判断filter是否处理此次调用，或者是否可以进行其他操作
        // 可以调用接口将自身加入到filter-chain，表示需要处理此次请求
        PrepareFilterChain(ctx *http.HttpContext) error

        // Handle filter hook function
        // 真正处理请求
        Handle(ctx *http.HttpContext)

        // 初始化阶段

        // filter 配置数据填充后，pixiu 会调用该函数，让filter 进行初始化
        Apply() error

        // pixiu 获取 filter 的config，会自动进行配置数据填充
        Config() interface{}
    }
```

比如说对于分布式链路追踪的 tracing filter 来讲。

在初始化阶段：

```go
// 返回 config 变量，让 pixiu 自动注入配置
func (m *TraceFilter) Config() interface{} {
    return m.cfg
}
// 进行自身初始化，此处是根据tc.URL生成 Jaeger 相关结构体
func (m *TraceFilter) Apply() error {
    // init
    tc := m.cfg
    switch tc.Type {
    case TracingType_Jaeger:
        tp, err := newTracerProvider(tc.URL)
        if err != nil {
            log.Fatal(err)
        }
        otel.SetTracerProvider(tp)
    default:
        panic("unsupported tracing")
    }
    return nil
}
```

在请求处理阶段：

```go
func (mf *TraceFilter) PrepareFilterChain(ctx *contexthttp.HttpContext) error {
    ctx.AppendFilterFunc(mf.Handle)
    return nil
}

func (f TraceFilter) Handle(hc *contexthttp.HttpContext) {
    // http 处理的 pre 阶段，生成 span
    spanName := "HTTP " + hc.Request.Method
    tr := otel.Tracer(traceName)
    ctx := extractTraceCtxRequest(hc.Request)
    ctxWithTid, span := tr.Start(ctx, spanName, trace.WithSpanKind(trace.SpanKindServer))

    hc.Request = hc.Request.WithContext(ctxWithTid)
    // 需要filter 调用next，让下一个filter处理
    hc.Next()
    // 处理的 post 阶段，完成 span
    span.End()
}
```

问题：

- 需要 filter 在 handle 中调用 hc.Next 来让下一个 filter 处理；
- 以 hc.Next 的调用位置来区分 pre 和 post 阶段，无法处理 grpc stream 或者 event 等请求和响应不一一对应的场景；
- 参数为 HttpContext，导致 http-filter 几乎无法复用到其他 network-filter下。

解决方案?

- 将 pre 和 post 阶段的处理函数分开；
- 由 pixiu 控制 chain 处理链的推进；
- 优化参数定义?

2.3 listener 定义

支持多协议的 listener 如何实现：

- 根据type 分别建立对应的 http、http2或tcp server 进行分配监听；
- 监听 tcp ， 接收包交由 network filter 进行协议解析处理。

目前使用第一种方案推进。

三、后续

3.1 网关功能

- Auth filter 功能
- cluster 和 route 的动态更新能力
- grpc 协议代理能力支持
- 更多 filter 能力
  - https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters

3.2 eventmesh

- 通过webhook 进行消息的接收和发送

3.3 dbmesh

- sharding 能力 ?

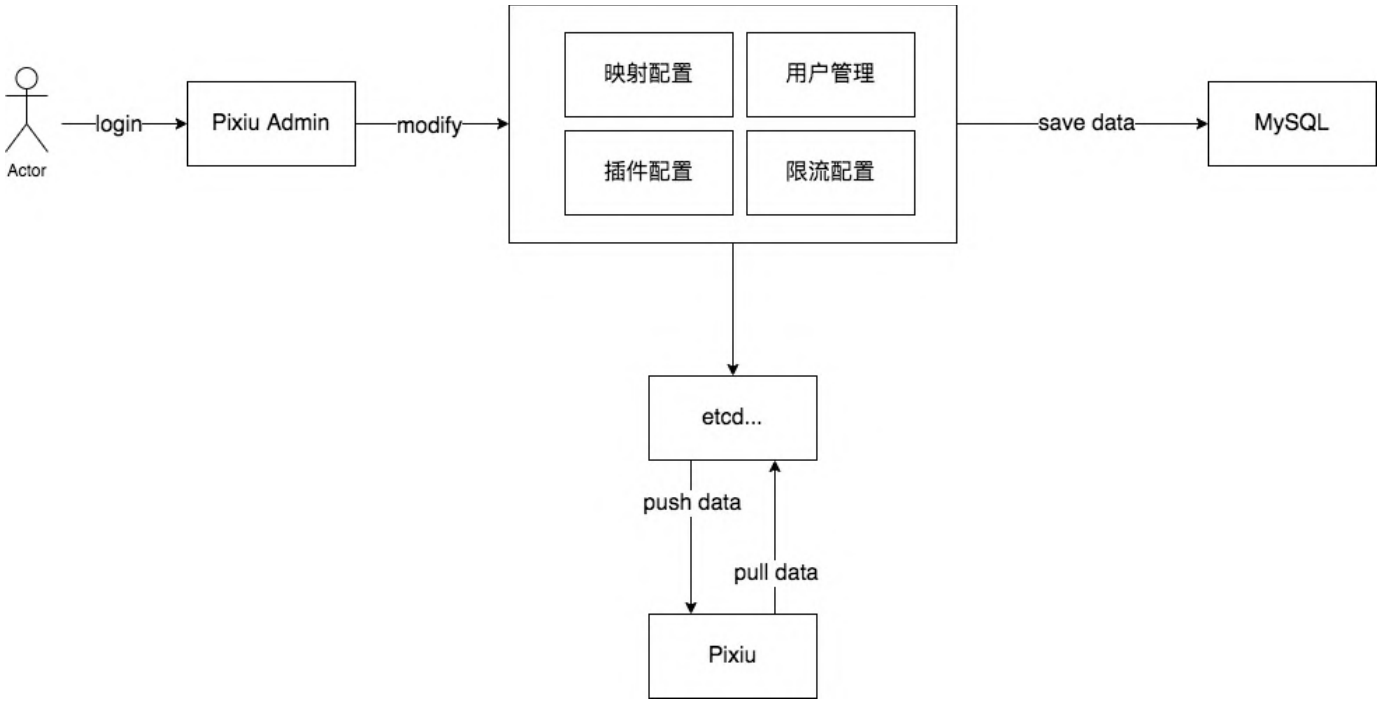
3.4 生态

- dubbo
  - 默认http to dubbo 规约支持 https://www.yuque.com/docs/share/4f82b71e-fa2b-4e65-8007-ba3afe2a1740?#
  - 3.0 支持
- springcloud 和 dubbo 互通
  - dubbo 应用级模式支持
  - 网关mock服务注册能力 https://developer.aliyun.com/article/792458?spm=a2c6h.13148508.0.0.66504f0e5Zkplj
  - dubbo to http 请求转换
-

参考：

- https://dubbogoproxy.yuque.com/docs/share/dbb648c6-af1a-4fd5-86fa-943e3ce68f05?# 《待解决问题》
- https://dubbogoproxy.yuque.com/dubbogoproxy/blxg8g/xwrmpi

roadmap



一、开源之夏天

- cli 工具替换为 cobra
- 登陆和用户管理
  - 默认 root 用户登陆
  - 用户管理界面 CRUD
  - 用户权限
- 0.4.0需求确定

配置粒度细化

细粒度配置，细粒度存储，细粒度更新

一、etcd 存储设计

root path
规则：{namespace}/{group}/config/{version}
例如：dubbo/pixiu/config/{version}

note：version暂时不处理

1.1全量存储

整体都存储在根目录key下

1.2 细粒度存储

按照 基础信息，resources 和 plugin三大块进行存储(Definitions看着目前没有用了)

{root}/base => Name,Description etc.
{root}/resources/{pathID} => path,type,description etc.
{root}/resources/{pathID}/plugins
{root}/resources/{pathID}/methods/{method}
{root}/plugin => PluginFilePath and PluginsGroup

YAML

```yaml
name: pixiu
description: pixiu sample
resources:
  - path: '/api/v1/test-dubbo/user'
    type: restful
    description: user
    plugins:
      pre:
        pluginNames:
          - rate limit
          - access
      post:
        groupNames:
          - group2
    methods:
      - httpVerb: GET
        onAir: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
          queryStrings:
            - name: name
              required: true
        integrationRequest:
          requestType: http
          host: 127.0.0.1:8889
          path: /UserProvider/GetUserByName
          mappingParams:
            - name: queryStrings.name
              mapTo: queryStrings.name
          group: "test"
          version: 1.0.0
      - httpVerb: POST
        onAir: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
          queryStrings:
            - name: name
              required: true
        integrationRequest:
          requestType: http
          host: 127.0.0.1:8889
          path: /UserProvider/CreateUser
          group: "test"
```

```
          version: 1.0.0
pluginFilePath: ""
pluginsGroup:
  - groupName: "group1"
    plugins:
      - name: "rate limit"
        version: "0.0.1"
        priority: 1000
        externalLookupName: "ExternalPluginRateLimit"
      - name: "access"
        version: "0.0.1"
        priority: 1000
        externalLookupName: "ExternalPluginAccess"
  - groupName: "group2"
    plugins:
      - name: "trace"
        version: "0.0.1"
        priority: 1000
        externalLookupName: "ExternalPluginTrace"


 dubbo/pixiu/config/current/  'name: pixiu\ndescription: pixiu sample'
 dubbo/pixiu/config/current/resources/user 'path: '/api/v1/test-dubbo/user'\ntype: restful\ndescription: user'
 dubbo/pixiu/config/current/resources/user/GET
```

二、版本和发布

默认{namespace}/{group}/config/current 路径是主路径，

- pixiu默认从该版本获取配置。
- admin设置相关配置后，需要进行发布，将最新版本数据覆盖到该路径
  - 可以直接全部覆盖
  - 也可以覆盖change部分

三、pixiu 初始化和监听配置

使用 etcd 将root目录下的所有kv全部获取，然后初始化。并且watch root 目录下的kv变更，然后根据对应的pluginID或者resourceID进行配置变更。

```go
c.rawClient.Get(c.ctx, k, clientv3.WithPrefix())
c.rawClient.Watch(c.ctx, prefix, clientv3.WithPrefix())
```

四、运行时更新

涉及discovery_service 的运行时更新，可以使用copy on write 方式进行数据变更，或者直接加锁进行变更。

五、前端对接

根据1，2小节，前端要对接的操作如下。

- 查看和编辑基础信息，名称描述等
- 查看、新增，编辑和删除插件配置列表和文件位置
- 查看、新增、编辑和删除 resource列表
    - 查看、新增、编辑和删除某一resource下的method列表
- 发布配置变更

六、pixiu watch

单一watch，然后区分add，modify和delete处理。
问题就是如何通过key来判断具体路径

Admin 测试文档

一、部署文档

1.1 下载代码

```bash
cd /root/
git clone https://github.com/dubbogo/pixiu-admin.git
```

1.2 部署etcd

```bash
docker run -d -p2379:2379 --env ALLOW_NONE_AUTHENTICATION=yes --name etcd bitnami/etcd

# m1/m1 pro 运行
# docker run -d -p2379:2379 --platform linux/amd64 --env ALLOW_NONE_AUTHENTICATION=yes --name etcd bitnami/etcd:3.5.1
```

1.3 运行admin

一、源代码运行

```bash
cd /root/pixiu-admin-master/cmd/admin/
# 直接运行
go run admin.go -c /root/pixiu-admin-master/configs/admin_config.yaml
# 后台运行
nohup go run admin.go -c /root/pixiu-admin-master/configs/admin_config.yaml &
```

admin_config.yaml的默认配置如下

```Bash
server:
  address: 127.0.0.1:8081 // 服务地址
etcd:
  address: 127.0.0.1:2379 // etcd 地址
  path: /pixiu/config/api // etcd 键值的默认路径，需要和 pixiu 中的配置对应
```

1.4 测试运行 admin-web

```Bash
cd /root/pixiu-admin-master/web/
yarn install # 安装依赖
yarn run serve # 测试运行
```

具体部署方案详见 web/README.md

admin-web的配置文件见 web 目录下的 vue.config.js，有关后端 server 地址配置片段如下所示：

```Vue
devServer:{
        host: '0.0.0.0',
        port: 8080, // web 应用的地址
        hot: true,
        https: false,
        open: false,
        disableHostCheck: true,
        proxy: {
            "/config": {
                target: "http://127.0.0.1:8081", // 后端服务地址
                ws: true, // 是否启用websockets
                changOrigin: true, //开启代理
                //将api替换为空
                // pathRewrite:{
                //     '^/api':''
                // },
            },
        }

    },
```

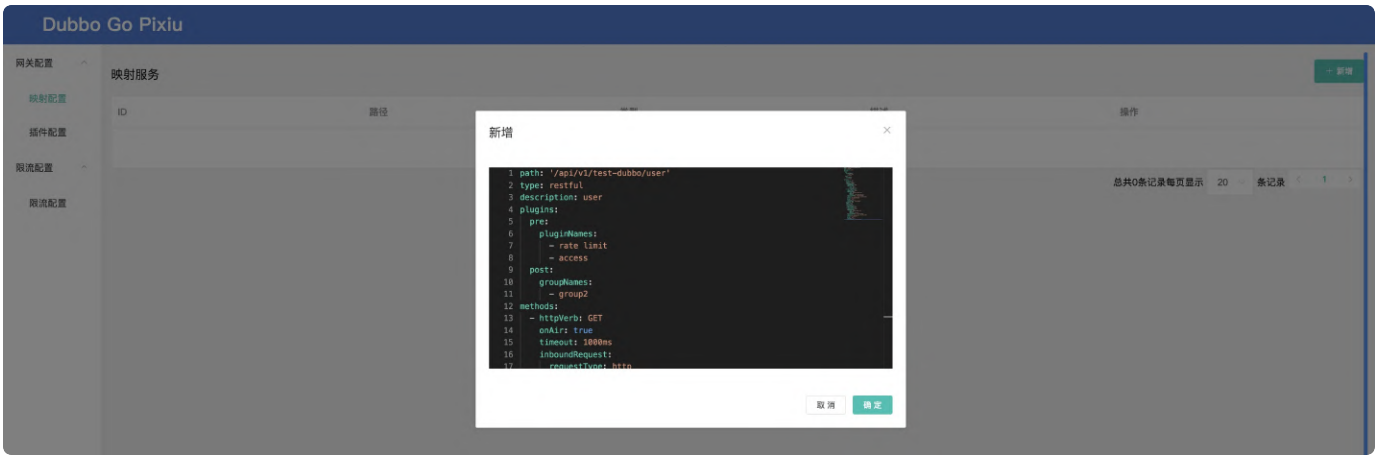
运行成功后，可以浏览器访问 http://127.0.0.1:8081/login.html#/Overview

二、相关操作

2.1 管理映射(Resource)

1 点击映射配置，进入映射配置列表界面，会展示当前配置的映射列表，点击右上角新增按钮可以创建新的映射配置。

2 在代码编辑器中键入对应的映射配置，点击确认进行创建。



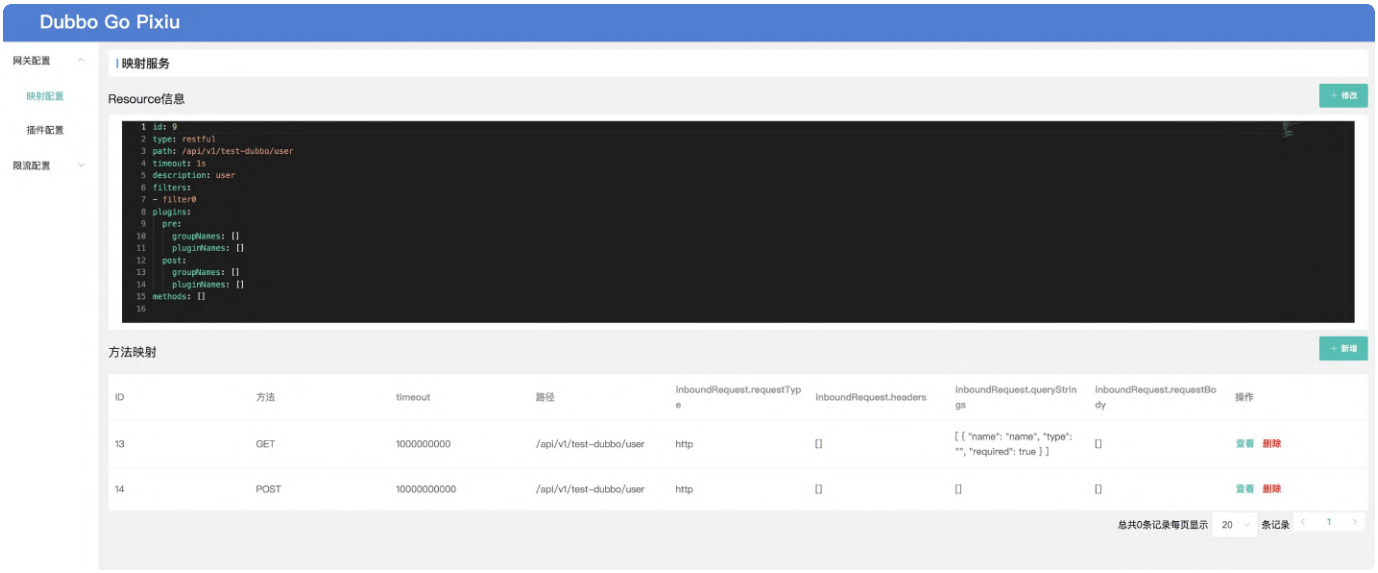
配置yaml具体如下，需要注意整体锁进格式

YAML

```yaml
path: '/api/v1/test-dubbo/user'
type: restful
description: user
filters:
  - filter0
methods:
  - httpVerb: GET
    onAir: true
    timeout: 1000ms
    inboundRequest:
      requestType: http
      queryStrings:
        - name: name
          required: true
    integrationRequest:
      requestType: dubbo
      mappingParams:
        - name: queryStrings.name
          mapTo: 0
          mapType: "java.lang.String"
      applicationName: "UserProvider"
      interface: "com.ic.user.UserProvider"
      method: "GetUserByName"
      group: "test"
      version: 1.0.0
      clusterName: "test_dubbo"
  - httpVerb: POST
    onAir: true
    timeout: 10s
    inboundRequest:
      requestType: http
    integrationRequest:
      requestType: dubbo
      mappingParams:
        - name: requestBody._all
          mapTo: 0
          mapType: "object"
      applicationName: "UserProvider"
      interface: "com.ic.user.UserProvider"
      method: "CreateUser"
      group: "test"
      version: 1.0.0
      clusterName: "test_dubbo"
```

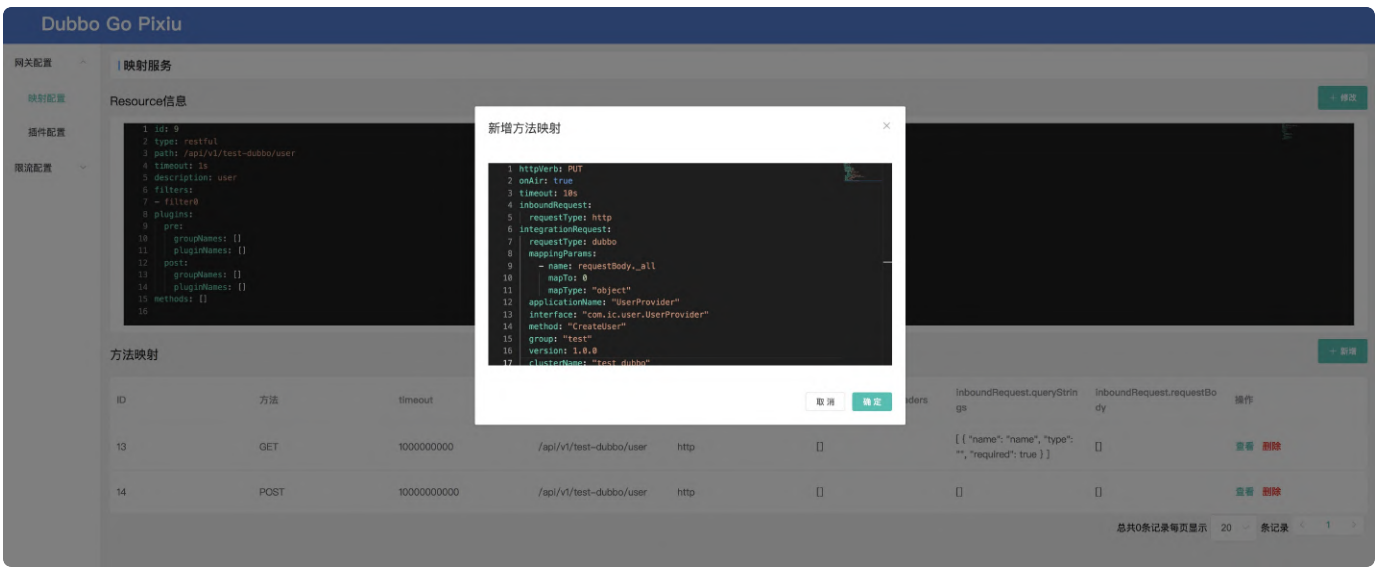
确认进行创建后，列表刷新会展示当前的映射列表，可以对其进行查看和删除操作。

点击删除会删除该映射配置，点击查看会跳转到映射配置详情界面。



可以在上方编辑区直接修改 Resource 的基础信息，然后点击右侧修改按钮进行修改

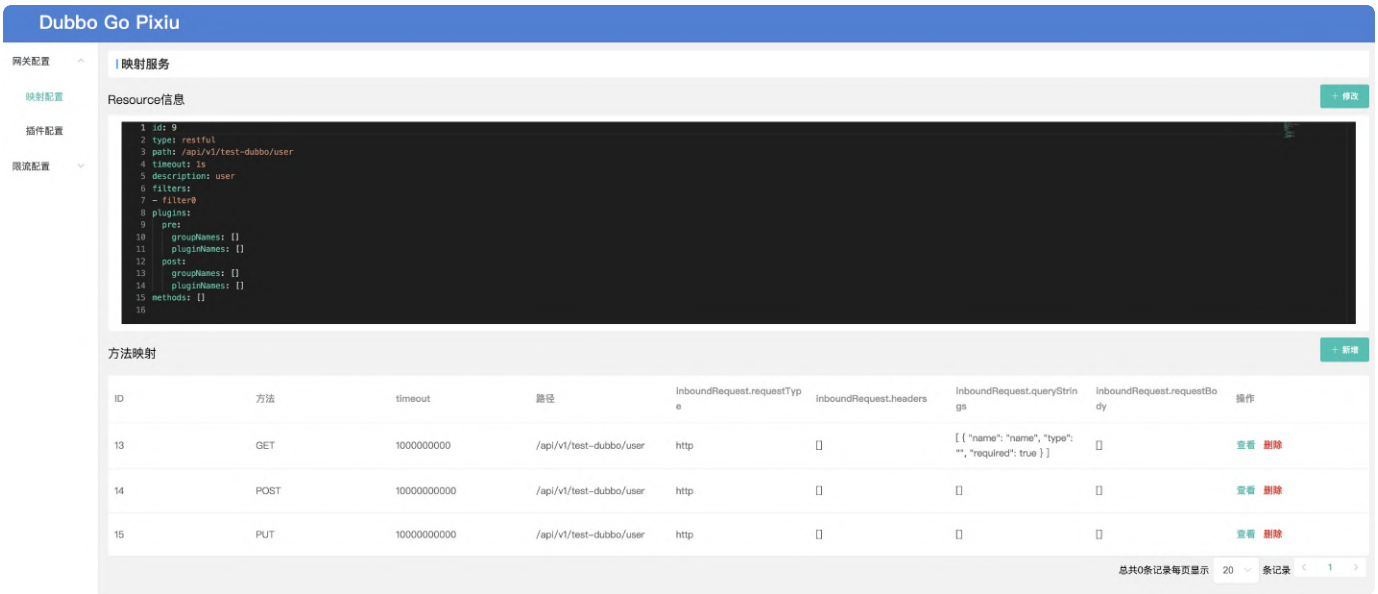
此外，可以在下方区域对 Resource 对应的方法映射进行增删改，点击右侧新增按钮，会弹出方法映射编辑框



具体的配置示例如下所示：

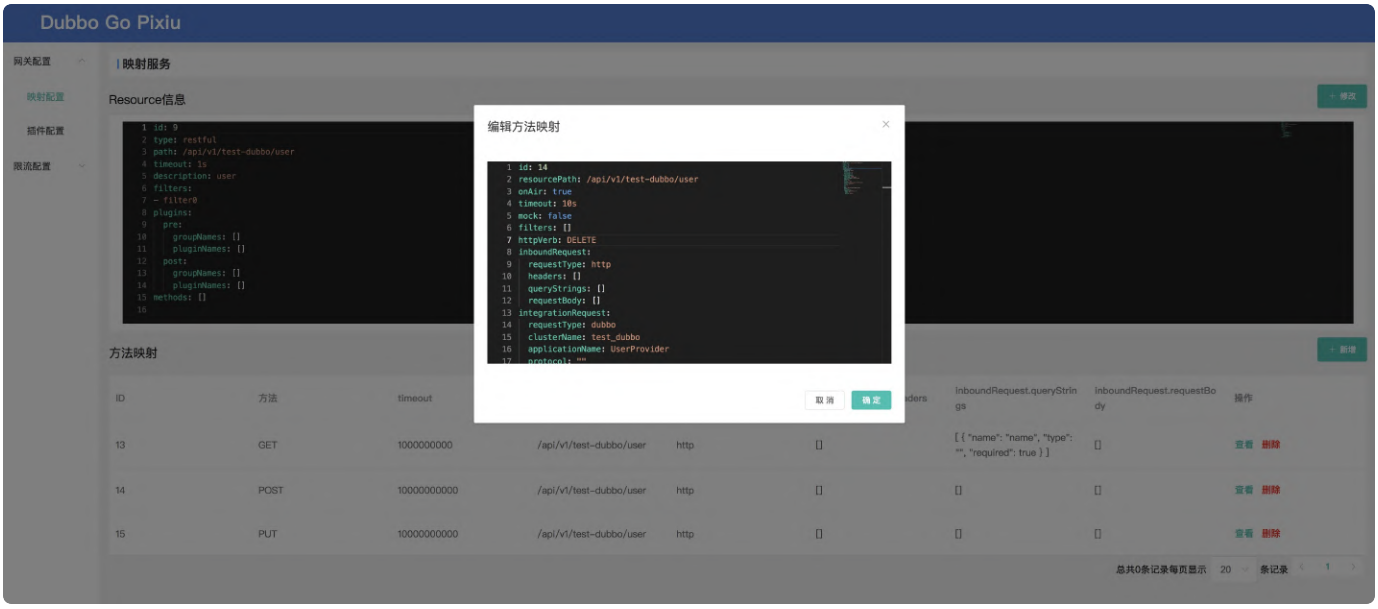
```yaml
httpVerb: PUT
onAir: true
timeout: 10s
inboundRequest:
  requestType: http
integrationRequest:
  requestType: dubbo
  mappingParams:
    - name: requestBody._all
      mapTo: 0
      mapType: "object"
  applicationName: "UserProvider"
  interface: "com.ic.user.UserProvider"
  method: "CreateUser"
  group: "test"
  version: 1.0.0
  clusterName: "test_dubbo"
```

点击确认后，会创建新的方法映射(method)，列表也会进行刷新。

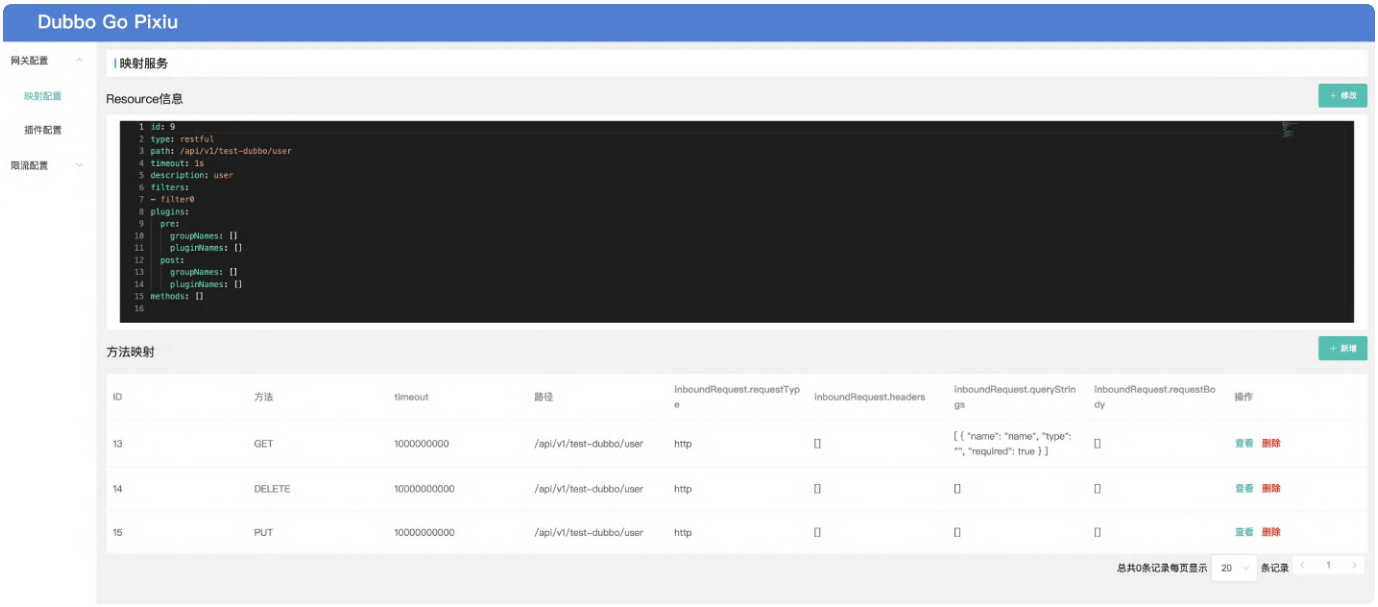


方法映射处有查看和删除两个操作，点击删除会直接删除对应的方法映射，点击查看则弹出编辑框，全量展示方法映射相关配置并可以进行修改。

比如，将第二个方法映射的 httpVerb 从 POST 修改为 DELETE。

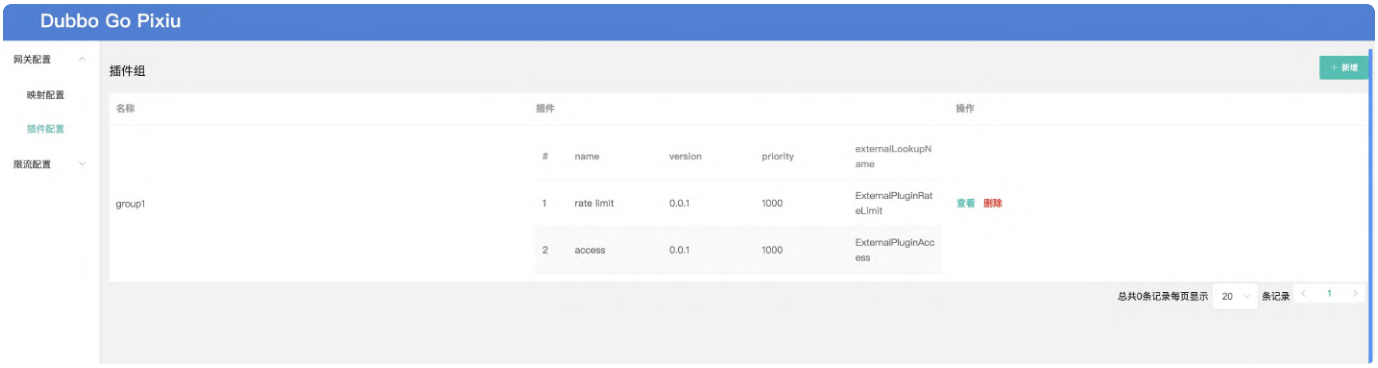


注意：id不能进行修改，即使修改保存后也会改变为旧值。点击确定后，会更新该方法映射。



2.2 管理插件组

点击左侧插件配置菜单项，可以查看插件相关配置。

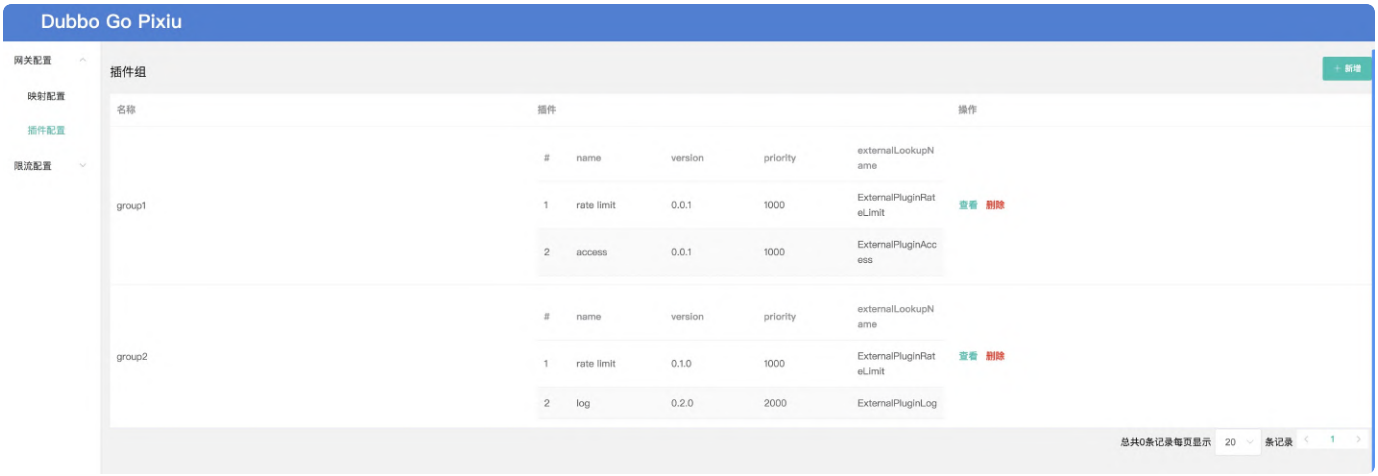


点击右上方新增，可以创建新的插件组。

具体插件组配置可以查阅插件配置相关文档，演示示例具体配置如下所示：

```yaml
groupName: "group2"
plugins:
    - name: "rate limit"
      version: "0.1.0"
      priority: 1000
      externalLookupName: "ExternalPluginRateLimit"
    - name: "log"
      version: "0.2.0"
      priority: 2000
      externalLookupName: "ExternalPluginLog"
```

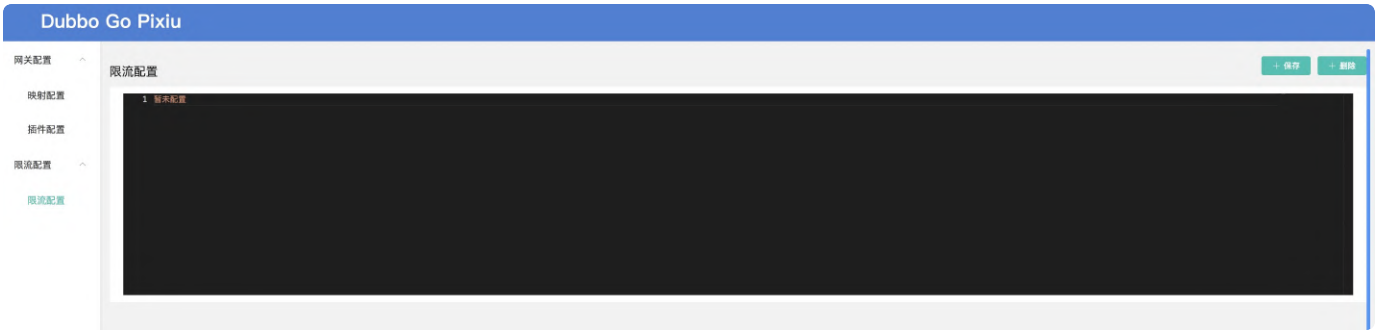
点击保存后，会创建新的插件组，并且刷新列表。



和映射配置类似，点击查看会弹出编辑框，可以对插件组配置进行修改；点击删除会删除整个插件组配置。

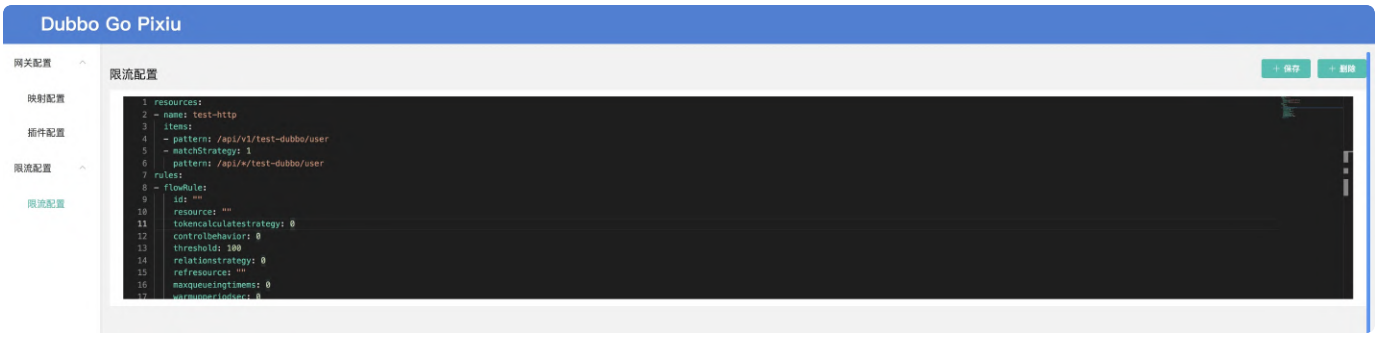
2.3 管理限流配置

点击左侧限流配置菜单项，可以进行限流组件相关配置。有关限流组件原理和配置，详见限流相关文档。



可以直接在编辑框中进行编辑，然后点击保存。点击删除则会删除当前的限流配置。

```yaml
resources:
- name: test-http
  items:
  - pattern: /api/v1/test-dubbo/user
  - matchStrategy: 1
    pattern: /api/*/test-dubbo/user
rules:
- flowRule:
    id: ""
    resource: ""
    tokencalculatestrategy: 0
    controlbehavior: 0
    threshold: 100
    relationstrategy: 0
    refresource: ""
    maxqueueingtimems: 0
    warmupperiodsec: 0
    warmupcoldfactor: 0
    statintervalinms: 1000
  enable: true

```



三、Pixiu 远程配置

3.1 启动和配置

启动 Pixiu，需要配置程序参数，指定 Pixiu 配置文件地址。

```yaml
-a /XXX/dubbo-go-proxy/samples/admin/proxy/api_config.yaml
```

具体的配置文件中需要定义对应的远程配置 etcd 地址和主路径：

```yaml
api_meta_config:
  address: "127.0.0.1:2379"
  api_config_path: "/pixiu/config/api"
```

3.2 测试

首先，我们在 admin 中新建如下 resource 配置。

YAML

```yaml
path: '/api/v1/test-dubbo/user'
type: restful
description: user
filters:
  - filter0
methods:
  - httpVerb: GET
    onAir: true
    timeout: 1000ms
    inboundRequest:
      requestType: http
      queryStrings:
        - name: name
          required: true
    integrationRequest:
      requestType: dubbo
      mappingParams:
        - name: queryStrings.name
          mapTo: 0
          mapType: "java.lang.String"
      applicationName: "UserProvider"
      interface: "com.ic.user.UserProvider"
      method: "GetUserByName"
      group: "test"
      version: 1.0.0
      clusterName: "test_dubbo"
  - httpVerb: POST
    onAir: true
    timeout: 10s
    inboundRequest:
      requestType: http
    integrationRequest:
      requestType: dubbo
      mappingParams:
        - name: requestBody._all
          mapTo: 0
          mapType: "object"
      applicationName: "UserProvider"
      interface: "com.ic.user.UserProvider"
      method: "CreateUser"
      group: "test"
      version: 1.0.0
      clusterName: "test_dubbo"
```

根据上文中的配置，可以执行如下命令查看 Pixiu 是否可以正常转发。

```yaml
curl "http://127.0.0.1:8888/api/v1/test-dubbo/user?name=tc"
curl -XPOST "http://127.0.0.1:8888/api/v1/test-dubbo/user?name=tc"
```

此时，pixiu 可以找到对应的映射配置，会尝试转发该请求，如果能找到对应服务，则返回服务返回值；如果未找到对应服务的返回值如下所示

```yaml
curl "http://127.0.0.1:8888/api/v1/test-dubbo/user?name=tc"

{"message":"Failed to invoke the method $invoke. No provider available for the service dubbo://:@:/?interface=com.ic.user.UserProvider\u0026group=test\u0026version=1.0.0 from registry zookeeper://127.0.0.1:2181?group=\u0026registry=zookeeper\u0026registry.label=true\u0026registry.preferred=false\u0026registry.role=0\u0026registry.timeout=3s\u0026registry.ttl=\u0026registry.weight=0\u0026registry.zone=\u0026simplified=false on the consumer 192.168.1.84 using the dubbo version 1.5.6 .Please check if the providers have been started and registered."}
```

而当我们用 PUT 指令时，网关找不到对应转发配置则会返回如下值

```yaml
curl -XPUT "http://127.0.0.1:8888/api/v1/test-dubbo/user?name=tc"

404 page not found%
```

为了让上述转发上述命令，我们可以在 admin 中将 httpverb 从 post 改为 put，再一次执行上述命令，发现情况发生变化。
接着我们再将 put 方法映射删除，再次执行 curl -XPUT，则返回值又变回了 404。
然后，我们新建一个 put 方法映射，再次执行 curl -XPUT，则可以进行转发。

后端API接口文档

配置postman 导入文件一起使用 一起使用

返回值：

```
{
    "code":
    "data": 一般为yaml格式
}
```

code：

10001 成功

10002 未找对对应的数据

10003 并发操作，刷新页面重试

一、基础信息

1、获取基础信息

```
GET /config/api/base HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: deb6b451-f211-41fd-8cdb-0801b13bab69
```

可能返回值：

```
{
    "code": "10001",
    "data": "name: pixiu\ndescription: pixiu111 sample\npluginFilePath: \"\"\n"
}
```

2、创建或者修改基础信息

```
POST /config/api/base HTTP/1.1
Host: 127.0.0.1:8080
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW
cache-control: no-cache
Postman-Token: a5867037-5bc0-4b9f-aef2-57980ddc9dbf
```

Content-Disposition: form-data; name="content"

name: pixiu
description: pixiu111 sample
------WebKitFormBoundary7MA4YWxkTrZu0gW--

二、Resource

2.1 获取 Resource 列表

GET /config/api/resource/list HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 19ffe51c-8193-4722-b94b-88d2502e3046

2.2 获取 Resource 详情

GET /config/api/resource/detail?resourceId=1 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 6b97b70d-70fc-4633-9ba7-dee3e08d7468

2.3 创建 Resource

需要注意：methods中的resourcePath要和resource的path相同

POST /config/api/resource/ HTTP/1.1
Host: 127.0.0.1:8080
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW
cache-control: no-cache
Postman-Token: 4e7b85b1-a969-46a4-b3d2-3df5d2f2073e

Content-Disposition: form-data; name="content"

    path: '/api/v1/test-dubbo/friend2'
    type: restful

```
    description: user
    timeout: 100ms
    plugins:
      pre:
        pluginNames:
          - rate limit
          - access
      post:
        groupNames:
          - group2
    methods:
      - httpVerb: GET
        resourcePath: '/api/v1/test-dubbo/friend2'
        onAir: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
          queryStrings:
            - name: name
              required: true
        integrationRequest:
          requestType: http
          host: 127.0.0.1:8889
          path: /UserProvider/GetUserByName
          mappingParams:
            - name: queryStrings.name
              mapTo: queryStrings.name
          group: "test"
          version: 1.0.0
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

2.3 修改 Resource

```
PUT /config/api/resource? HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
```

```
Postman-Token: 67d3fb19-bb66-4e92-be1b-419fdd3bcb28
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="content"

    id: 1
    path: '/api/v1/test-dubbo/friend'
    type: restful
    description: update
    timeout: 1000ms
    plugins:
      pre:
        pluginNames:
          - rate limit
          - access
      post:
        groupNames:
          - group2
    methods:
      - httpVerb: GET
        onAir: true
        timeout: 1000ms
        inboundRequest:
          requestType: http
          queryStrings:
            - name: name
              required: true
        integrationRequest:
          requestType: http
          host: 127.0.0.1:8889
          path: /UserProvider/GetUserByName
          mappingParams:
            - name: queryStrings.name
              mapTo: queryStrings.name
          group: "test"
          version: 1.0.0
```

------WebKitFormBoundary7MA4YWxkTrZu0gW--

2.4 删除 Resource

DELETE /config/api/resource/?resourceId=2 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: faed927d-2857-40a2-9cfb-1a3ce2f704e4

三、Method 相关

3.1 查询某个 Resource 下的 Method 列表

GET /config/api/resource/method/list?resourceId=1 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 90c3df1d-02b4-42af-8846-f4c74a36fdae

3.2 查询 method 详情

GET /config/api/resource/method/detail?resourceId=1& methodId=2 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 6d0909cc-1585-46c5-975f-e4a0d0b2f490

3.3 创建 method

POST /config/api/resource/method/?resourceId=1 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 0d5e2885-67e6-41cb-84e8-777a8d077384
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="content"

httpVerb: PUT
resourcePath: '/api/v1/test-dubbo/friend'

```
onAir: true
timeout: 1000ms
inboundRequest:
    requestType: http
    queryStrings:
    - name: name
      required: true
integrationRequest:
    requestType: http
    host: 127.0.0.1:8889
    path: /UserProvider/GetUserByName
    mappingParams:
    - name: queryStrings.name
      mapTo: queryStrings.name
    group: "test"
    version: 1.0.0
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

3.4 修改 method

```
PUT /config/api/resource/method/?resourceId=1 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 9a635d6d-b52a-4cf5-8f1a-a7d450cc3552
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="content"

id: 2
httpVerb: PUT
resourcePath: '/api/v1/test-dubbo/friend'
onAir: true
timeout: 300ms
inboundRequest:
    requestType: http
    queryStrings:
    - name: name
      required: true
```

```
integrationRequest:
    requestType: http
    host: 127.0.0.1:8889
    path: /UserProvider/GetUserByName
    mappingParams:
    - name: queryStrings.name
      mapTo: queryStrings.name
    group: "test"
    version: 1.0.0
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

3.5 删除 method

```
DELETE /config/api/resource/method/?resourceId=1&amp;amp; methodId=2 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 83930da3-7ab5-47b4-a47e-ef0ceeb6e9af
```

四、PluginGroup和plugin相关

4.1 查看 PluginGroup 列表

```
GET /config/api/plugin_group/list HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: f48aa74f-38c9-4ee9-87a3-c4a35a2350f9
```

4.2 查看详情

```
GET /config/api/plugin_group/list HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 739ee1d2-7801-4005-90a3-cc2dfb03b5e7
```

4.3 创建

```
POST /config/api/plugin_group/ HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 1e41964b-989b-4f03-8512-9dc7e9b7635d
```

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="content"

groupName: "group1"
plugins:
    - name: "rate limit"
      version: "0.0.1"
      priority: 1000
      externalLookupName: "ExternalPluginRateLimit"
    - name: "access"
      version: "0.0.1"
      priority: 1000
      externalLookupName: "ExternalPluginAccess"
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

4.4 修改

```
PUT /config/api/plugin_group/ HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: 5166a7df-b838-4941-add3-5073a6e430f0
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="content"

groupName: "group1"
plugins:
    - name: "rate limit"
      version: "0.0.2"
      priority: 1000
      externalLookupName: "ExternalPluginRateLimit"
    - name: "access"
      version: "0.0.1"
      priority: 1000
      externalLookupName: "ExternalPluginAccess"
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

4.5 删除

```
DELETE /config/api/plugin_group/?name=group1 HTTP/1.1
Host: 127.0.0.1:8080
cache-control: no-cache
Postman-Token: d761e1fa-02c4-4693-b765-b16ce389c9b2
```

五、其他

5.1 梦超的限流配置 TODO

六、postman导出文件

可以将其保存为 admin.json，然后导入postman

JSON


```
{
  "info": {
    "_postman_id": "cb35ad41-ccb3-4c7c-a3a8-ebbd2f60f7da",
    "name": "pixiu admin",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
  },
  "item": [
    {
      "name": "设置基础信息",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "name": "Content-Type",
            "value": "application/json",
            "type": "text"
          }
        ],
        "body": {
          "mode": "formdata",
          "formdata": [
            {
              "key": "content",
              "value": "name: pixiu\ndescription: pixiu111 sample",
              "type": "text"
            }
          ]
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/base",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "base"
          ]
```

```
        }
      },
      "response": []
    },
    {
      "name": "获取基础信息",
      "request": {
        "method": "GET",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": ""
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/base",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "base"
          ]
        }
      },
      "response": []
    },
    {
      "name": "获取resource列表",
      "request": {
        "method": "GET",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": ""
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/resource/list",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
```

```
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "resource",
            "list"
          ]
        }
      },
      "response": []
    },
    {
      "name": "创建resource",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "name": "Content-Type",
            "value": "application/json",
            "type": "text"
          }
        ],
        "body": {
          "mode": "formdata",
          "formdata": [
            {
              "key": "content",
              "value": "    path: '/api/v1/test-dubbo/friend2'\n    type: restful\n    description: user\n    timeout: 100ms\n    plugins:\n      pre:\n        pluginNames:\n          - rate limit\n          - access\n      post:\n        groupNames:\n          - group2\n    methods:\n      - httpVerb: GET\n        resourcePath: '/api/v1/test-dubbo/friend2'\n        onAir: true\n        timeout: 1000ms\n        inboundRequest:\n          requestType: http\n          queryStrings:\n            - name: name\n              required: true\n        integrationRequest:\n          requestType: http\n          host: 127.0.0.1:8889\n          path: /UserProvider/GetUserByName\n          mappingParams:\n            - name: queryStrings.name\n              mapTo: queryStrings.name\n              group: \"test\"\n          version: 1.0.0",
              "type": "text"
            }
          ]
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/resource/",
```

```
            "protocol": "http",
            "host": [
              "127",
              "0",
              "0",
              "1"
            ],
            "port": "8080",
            "path": [
              "config",
              "api",
              "resource",
              ""
            ]
          }
        },
        "response": []
      },
      {
        "name": "获取resource配置详情(yaml格式)",
        "request": {
          "method": "GET",
          "header": [],
          "body": {
            "mode": "raw",
            "raw": ""
          },
          "url": {
            "raw": "http://127.0.0.1:8080/config/api/resource/detail?resourceId=1",
            "protocol": "http",
            "host": [
              "127",
              "0",
              "0",
              "1"
            ],
            "port": "8080",
            "path": [
              "config",
              "api",
              "resource",
              "detail"
            ],
            "query": [
              {
                "key": "resourceId",
                "value": "1"
```

```
            }
          ]
        }
      },
      "response": []
    },
    {
      "name": "修改资源",
      "request": {
        "method": "PUT",
        "header": [],
        "body": {
          "mode": "formdata",
          "formdata": [
            {
              "key": "content",
              "value": "    id: 1\n    path: '/api/v1/test-dubbo/friend'\n    type: restful\n    description: update\n    timeout: 1000ms\nplugins:\n      pre:\n        pluginNames:\n          - rate limit\n          - access\n      post:\n        groupNames:\n          - group2\nmethods:\n      - httpVerb: GET\n        onAir: true\n        timeout: 1000ms\n        inboundRequest:\n          requestType: http\n          queryStrings:\n            - name: name\n              required: true\n    integrationRequest:\n          requestType: http\n          host: 127.0.0.1:8889\n          path: /UserProvider/GetUserByName\n          mappingParams:\n            - name: queryStrings.name\n              mapTo: queryStrings.name\n          group: \"test\"\n          version: 1.0.0",
              "type": "text"
            }
          ]
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/resource",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "resource"
          ],
          "query": [
            {
```

```
                "key": "content",
                "value": "",
                "disabled": true
              }
            ]
          }
        },
        "response": []
      },
      {
        "name": "删除resource",
        "request": {
          "method": "DELETE",
          "header": [],
          "body": {
            "mode": "raw",
            "raw": ""
          },
          "url": {
            "raw": "http://127.0.0.1:8080/config/api/resource/?resourceId=2",
            "protocol": "http",
            "host": [
              "127",
              "0",
              "0",
              "1"
            ],
            "port": "8080",
            "path": [
              "config",
              "api",
              "resource",
              ""
            ],
            "query": [
              {
                "key": "resourceId",
                "value": "2"
              }
            ]
          }
        },
        "response": []
      },
      {
        "name": "获取某个资源下的method列表",
        "request": {
```

```
        "method": "GET",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": ""
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/resource/method/list?resourceId=1",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "resource",
            "method",
            "list"
          ],
          "query": [
            {
              "key": "resourceId",
              "value": "1"
            }
          ]
        }
      },
      "response": []
    },
    {
      "name": "创建新的method",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "formdata",
          "formdata": [
            {
              "key": "content",
              "value": "httpVerb: PUT\nresourcePath: '/api/v1/test-dubbo/friend'\nonAir: true\ntimeout: 1000ms\ninboundRequest:\n    requestType: http\n    queryStrings:\n    - name: name\n      required: true\nintegrationRequest:\n    requestType: http\n    host: 127.0.0.1:8889\n    path: /
```

```
    UserProvider/GetUserByName\n    mappingParams:\n    - name: queryStrings.
    name\n      mapTo: queryStrings.name\n    group: \"test\"\n    version:
    1.0.0",
306                 "type": "text"
307               }
308             ]
309           },
310           "url": {
311             "raw": "http://127.0.0.1:8080/config/api/resource/method/?resou
    rceId=1",
312             "protocol": "http",
313             "host": [
314               "127",
315               "0",
316               "0",
317               "1"
318             ],
319             "port": "8080",
320             "path": [
321               "config",
322               "api",
323               "resource",
324               "method",
325               ""
326             ],
327             "query": [
328               {
329                 "key": "resourceId",
330                 "value": "1"
331               }
332             ]
333           }
334         },
335         "response": []
336       },
337       {
338         "name": "获取method详情",
339         "request": {
340           "method": "GET",
341           "header": [],
342           "body": {
343             "mode": "raw",
344             "raw": ""
345           },
346           "url": {
347             "raw": "http://127.0.0.1:8080/config/api/resource/method/detai
    l?resourceId=1&methodId=2",
348             "protocol": "http",
```

```
            "host": [
              "127",
              "0",
              "0",
              "1"
            ],
            "port": "8080",
            "path": [
              "config",
              "api",
              "resource",
              "method",
              "detail"
            ],
            "query": [
              {
                "key": "resourceId",
                "value": "1"
              },
              {
                "key": "methodId",
                "value": "2"
              }
            ]
          }
        },
        "response": []
      },
      {
        "name": "修改method",
        "request": {
          "method": "PUT",
          "header": [],
          "body": {
            "mode": "formdata",
            "formdata": [
              {
                "key": "content",
                "value": "id: 2\nhttpVerb: PUT\nresourcePath: '/api/v1/test-dubbo/friend'\nonAir: true\ntimeout: 300ms\ninboundRequest:\n    requestType: http\n    queryStrings:\n    - name: name\n      required: true\nintegrationRequest:\n    requestType: http\n    host: 127.0.0.1:8889\n    path: /UserProvider/GetUserByName\n    mappingParams:\n    - name: queryStrings.name\n      mapTo: queryStrings.name\n    group: \"test\"\n    version: 1.0.0",
                "type": "text"
              }
            ]
```

```
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/resource/method/?resou
rceId=1",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "resource",
            "method",
            ""
          ],
          "query": [
            {
              "key": "resourceId",
              "value": "1"
            }
          ]
        }
      },
      "response": []
    },
    {
      "name": "删除method",
      "request": {
        "method": "DELETE",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": ""
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/resource/method/?resou
rceId=1&methodId=2",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
```

```
          "port": "8080",
          "path": [
            "config",
            "api",
            "resource",
            "method",
            ""
          ],
          "query": [
            {
              "key": "resourceId",
              "value": "1"
            },
            {
              "key": "methodId",
              "value": "2"
            }
          ]
        }
      },
      "response": []
    },
    {
      "name": "获取pluginGroupList",
      "request": {
        "method": "GET",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": ""
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/plugin_group/list",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "plugin_group",
            "list"
          ]
        }
```

```
      },
      "response": []
    },
    {
      "name": "创建新的pluginGroup",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "formdata",
          "formdata": [
            {
              "key": "content",
              "value": "groupName: \"group1\"\nplugins:\n    - name: \"rate limit\"\n      version: \"0.0.1\"\n      priority: 1000\n      externalLookupName: \"ExternalPluginRateLimit\"\n    - name: \"access\"\n      version: \"0.0.1\"\n      priority: 1000\n      externalLookupName: \"ExternalPluginAccess\"",
              "type": "text"
            }
          ]
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/plugin_group/",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "plugin_group",
            ""
          ]
        }
      },
      "response": []
    },
    {
      "name": "修改pluginGroup",
      "request": {
        "method": "PUT",
        "header": [],
        "body": {
```

```
          "mode": "formdata",
          "formdata": [
            {
              "key": "content",
              "value": "groupName: \"group1\"\nplugins:\n    - name: \"rate limit\"\n      version: \"0.0.2\"\n      priority: 1000\n      externalLookupName: \"ExternalPluginRateLimit\"\n    - name: \"access\"\n      version: \"0.0.1\"\n      priority: 1000\n      externalLookupName: \"ExternalPluginAccess\"",
              "type": "text"
            }
          ]
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/plugin_group/",
          "protocol": "http",
          "host": [
            "127",
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "plugin_group",
            ""
          ]
        }
      },
      "response": []
    },
    {
      "name": "删除pluginGroup",
      "request": {
        "method": "DELETE",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": ""
        },
        "url": {
          "raw": "http://127.0.0.1:8080/config/api/plugin_group/?name=group1",
          "protocol": "http",
          "host": [
            "127",
```

```
            "0",
            "0",
            "1"
          ],
          "port": "8080",
          "path": [
            "config",
            "api",
            "plugin_group",
            ""
          ],
          "query": [
            {
              "key": "name",
              "value": "group1"
            }
          ]
        }
      },
      "response": []
    }
  ]
}
```

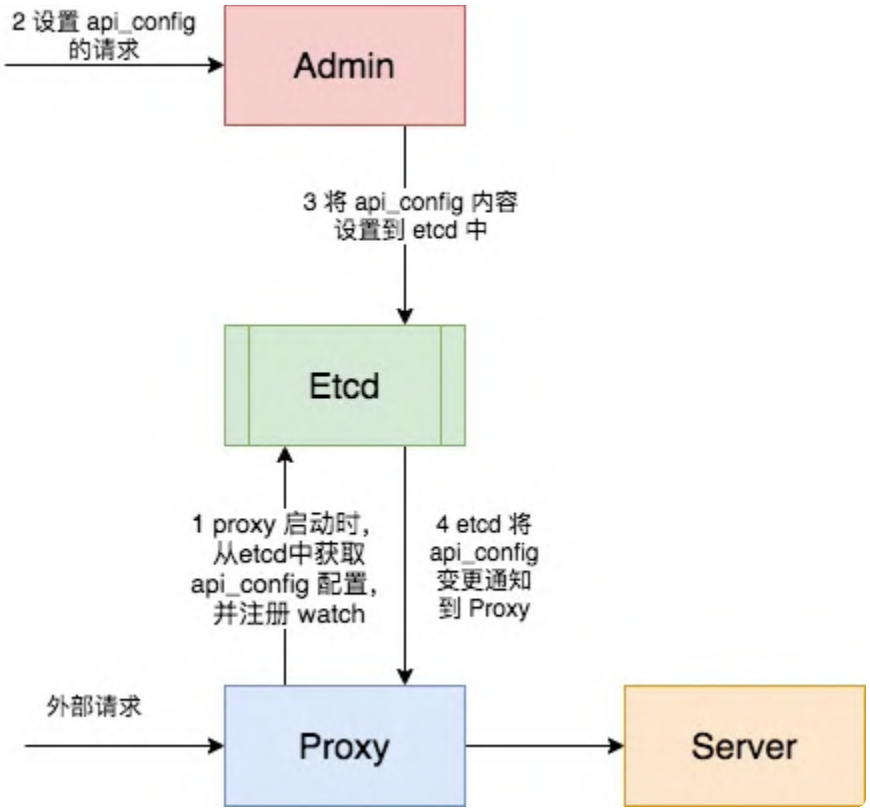
七、 登录注册

https://www.yuque.com/docs/share/7668bb7c-3901-4b3e-9a04-73ef3525e9b5?# 《设计文档 - 前端》

问题反馈

具体实现

一、整体示意图



二、具体源码实现

2.1 Proxy 启动加载远程配置

在 control.go 启动时，会根据 bootstrap 的配置来决定是从远程加载 还是从本地文件读取 api_config。

```
if bootstrap.GetAPIMetaConfig() != nil {
    if _, err := config.LoadAPIConfig(bootstrap.GetAPIMetaConfig()); err != nil {
        logger.Errorf( fmt: "load api config from etcd error:%+v", err)
        return err
    }
} else {
    if _, err := config.LoadAPIConfigFromFile(apiConfigPath); err != nil {
        logger.Errorf( fmt: "load api config error:%+v", err)
        return err
    }
}
```

config.LoadAPIConfig 函数会根据 APIMetaConfig 来初始化 etcdclient ，从 etcd 读取对应的 api_config 内容，并注册监听的 watch

```go
// LoadAPIConfig load the api config from config center
func LoadAPIConfig(metaConfig *model.APIMetaConfig) (*APIConfig, error) {

    client = etcdv3.NewConfigClient(
        etcdv3.WithName(etcdv3.RegistryETCDV3Client),
        etcdv3.WithTimeout(10*time.Second),
        etcdv3.WithEndpoints(strings.Split(metaConfig.Address,  sep: ",")...),
    )

    go listenAPIConfigNodeEvent(metaConfig.APIConfigPath)

    content, err := client.Get(metaConfig.APIConfigPath)

    if err != nil {
        return nil, perrors.Errorf( format: "Get remote config fail error %v", err)
    }

    initAPIConfigFromString(content)

    return apiConfig, nil
}
```

discovery_service 初始化时会向 config 注册监听 api_config 变化的 listener

```go
// InitAPIsFromConfig inits the router from API config and to local cache
func InitAPIsFromConfig(apiConfig config.APIConfig) error {
    localAPIDiscSrv := extension.GetMustAPIDiscoveryService(constant.LocalMemoryApiDiscoveryService)
    if len(apiConfig.Resources) == 0 {
        return nil
    }
    // register config change listener
    config.RegisterConfigListener(localAPIDiscSrv)
    // load pluginsGroup
    plugins.InitPluginsGroup(apiConfig.PluginsGroup, apiConfig.PluginFilePath)
    // init plugins from resource
    plugins.InitAPIURLWithFilterChain(apiConfig.Resources)
    return loadAPIFromResource( parentPath: "", apiConfig.Resources,  parentHeaders: nil, localAPIDiscSrv)
}
```

2.2 Proxy 感知变更修改配置

当 etcd 中的 api_config 内容发生变化时，会通知到 proxy 的 watch 监听中，然后 proxy 会通过 listener 通知相关的涉众，也就是 discovery_service，discovery_service就重新进行初始化。

```go
func listenAPIConfigNodeEvent(key string) bool {
    for {
        wc, err := client.Watch(key)
        if err != nil {
            logger.Warnf( fmt: "Watch api config {key:%s} = error{%v}", key, err)
            return false
        }

        select {

        // client stopped
        case <-client.Done():
            logger.Warnf( fmt: "client stopped")
            return false
        // client ctx stop
        // handle etcd events
        case e, ok := <-wc:
            if !ok {
                logger.Warnf( fmt: "watch-chan closed")
                return false
            }

            if e.Err() != nil {
                logger.Errorf( fmt: "watch ERR {err: %s}", e.Err())
                continue
            }
            for _, event := range e.Events {
                switch event.Type {
                case mvccpb.PUT:
                    initAPIConfigFromString(string(event.Kv.Value))
                    listener.APIConfigChange(GetAPIConf())
                case mvccpb.DELETE:
                    logger.Warnf( fmt: "get event (key{%s}) = event{EventNodeDeleted}", event.Kv.Key)
                    return true
                default:
                    return false
                }
            }
```

2.3 Admin 操作

admin 是一个简单的网络服务器，监听特定端口，提供 get/set两个功能。内部具体实现则是从 etcd 中读取和设置对应数据。

使用文档

详见 dubbo-go/dubbo-go-proxy/samples/admin文件夹

一、准备etcd环境

使用docker或者本地启动 etcd，然后初始化 etcd 中的 api_config 配置
– 注意 etcdctl 的api 的版本，export ETCDCTL_API=3
– 如果使用 docker 启动的 etcd， 可以提前将 api_config.yaml 文件拷贝到 docker 中。使用如下命令 *`docker cp api_config.yaml mycontainer:/path`*
– 在 api_config.yaml 所在路径下，执行如下命令 *`cat api_config.yaml | etcdctl put "/proxy/config/api"` 设置 etcd 中目标路径的值，也就是 api_config 的值*

二、启动 Server

设置如下环境变量，注意 XXX 路径为自己环境下的路径
– CONF_PROVIDER_FILE_PATH=/XXX/dubbo-go-proxy/samples/admin/server/config/server.yml
– APP_LOG_CONF_FILE=/XXX/dubbo-go-proxy/samples/admin/server/config/log.yml

运行 samples/admin/server/app/server.go

三、启动 dubbo-go-proxy

配置如下 program arguments：
– -c /XXX/dubbo-go-proxy/samples/admin/proxy/conf.yaml

注意注意，在 conf.yaml 文件中，定义了 api_meta_config 相关的配置，即配置 etcd 的地址和对应 api_config的path。供 proxy 使用，向 etcd 中查询对应的值。

```*yaml*
  *api_meta_config:*
    *address: "127.0.0.1:2379"*
    *api_config_path: "/proxy/config/api"*
```

```

执行 cmd/proxy/proxy.go

## 四、第一次验证

**执行如下 curl 命令** curl "http://127.0.0.1:8888/api/v1/test-dubbo/user?name=tc" ， 可以得到如下结果

```json
{
    "age": 18,
    "id": "0001",
    "name": "tc"
}
```

五、启动 admin

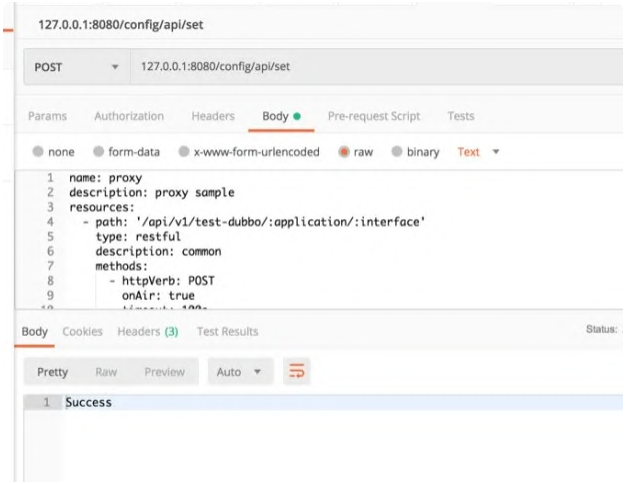
配置如下 program arguments，设置 admin 自身的一些参数，比如端口，etcd path等。
- -c /XXX/dubbo-go-proxy/samples/admin/admin/admin_config.yaml

执行 cmd/admin/admin.go

六、验证 admin 的功能和 api_config 配置自动变更 功能

- **执行如下命令** *`curl 127.0.0.1:8080/config/api` 可以从admin获得当前proxy的api_config配置*
- **修改** api_config.yaml 内容, 比如将请求的path *`test-dubbo/user` 修改为`test-dubbo/user_new`*
- **执行如下命令，将修改后的 api_config.yaml 文件通过 admin 更改到 etcd 中** *`curl "127.0.0.1:8080/config/api/set" -X POST --data-binary "@/xx/xx/dubbo-go-proxy/samples/admin/proxy/api_config.yaml"`*

！注意，是将文件的内容作为request body 传入，postman配置如下图示。
```



– **再次执行如下命令**，*`curl "http://127.0.0.1:8888/api/v1/test-dubbo/user?name=tc"`*， 会得到 404 的结果

– **执行如下命令** *`curl "http://127.0.0.1:8888/api/v1/test-dubbo/user_new?name=tc"`，则可以获得正确的返回*

Local Plugins实现方案

配置

配置文件路径：configs/api_config.yaml

pluginFilePath

此配置为.so的存放路径，例如：configs/plugins/extern_filter.so

注意：so文件加载，只能有一个文件，意味着所有的扩展函数（externalLookupName），必须都要放在同一个so文件中。

pluginsGroup

插件组配置，继续往下看。

groupName

组名称，resources中plugins配置有groupNames和pluginNames，其中groupNames可以配置整组的plugins，全局唯一，且依赖名称查找组。

plugins

plugin的集合，下面的四个配置组成一个完整的plugin。

name

plugin的名称，全局唯一，且依赖名称查找对应的plugin。

version

plugin的版本号，版本号大的优先，结合remote plugin配合使用有效，第一期未实现（跟remote plugin一起实现）。

priority

plugin的优先级，值越大，优先级越高，结合remote plugin配合使用有效，第一期未实现（跟remote plugin一起实现）。

externalLookupName

需要加载的扩展插件名称，参见samples/plugins/plugin.go中的export filter。

切入点

初始化插件

从配置中读取path对应的插件集合到内存中去：

```
#pkg/service/api/discovery_service.go
func InitAPIsFromConfig(apiConfig config.APIConfig) error {
   ...
      // load pluginsGroup
      plugins.InitPluginsGroup(apiConfig.PluginsGroup, apiConfig.PluginFilePath)
      // init plugins from resource
      plugins.InitAPIURLWithFilterChain(apiConfig.Resources)
   ...
}
```

应用插件

```
#pkg/proxy/listener.go
func addFilter(ctx *h.HttpContext, api router.API) {
      ...
      // load plugins
       filterChain := plugins.GetAPIFilterFuncsWithAPIURL(ctx.Request.URL.Path)
      ctx.AppendFilterFunc(filterChain...)
      ...
}
```

实现

参见：pkg/filter/plugins/plugins.go源码

对外接口

InitPluginsGroup

输入参数：groups和filePath

filePath：对应api_config.yaml中的pluginFilePath

groups：对应api_config.yaml中的pluginsGroup

加载filePath对应的.so文件，并将配置中groups的插件函数映射到本地字典groupWithPluginsMap中。

InitAPIURLWithFilterChain

输入参数：resources

将resources中path用到扩展插件一一映射到字典apiURLWithPluginsMap中key为fullPath，value为需要要执行插件函数。

GetAPIFilterFuncsWithAPIURL

输入参数：url

根据url（fullPath）从apiURLWithPluginsMap中获取对应要执行的插件函数。

内部接口

pairURLWithFilterChain

由InitAPIURLWithFilterChain调用，解析resources中fullPath对应的插件函数。

loadExternalPlugin

由InitPluginsGroup调用，加载so文件中具体对应插件函数，主要实现如下：

```
sb, err := pl.Lookup(p.ExternalLookupName)
if nil != err {
   panic(err)
}

sbf := sb.(func() filter.Filter)
return sbf().Do()
```

getApiFilterFuncsWithPluginsGroup

由pairURLWithFilterChain调用，根据groupNames和pluginNames查找so中对应的插件函数，具体可参见代码。

测试

测试比较简单，参见源码：pkg/filter/plugins/plugins_test.go

编译SO

参见：samples/plugins/bulid.txt

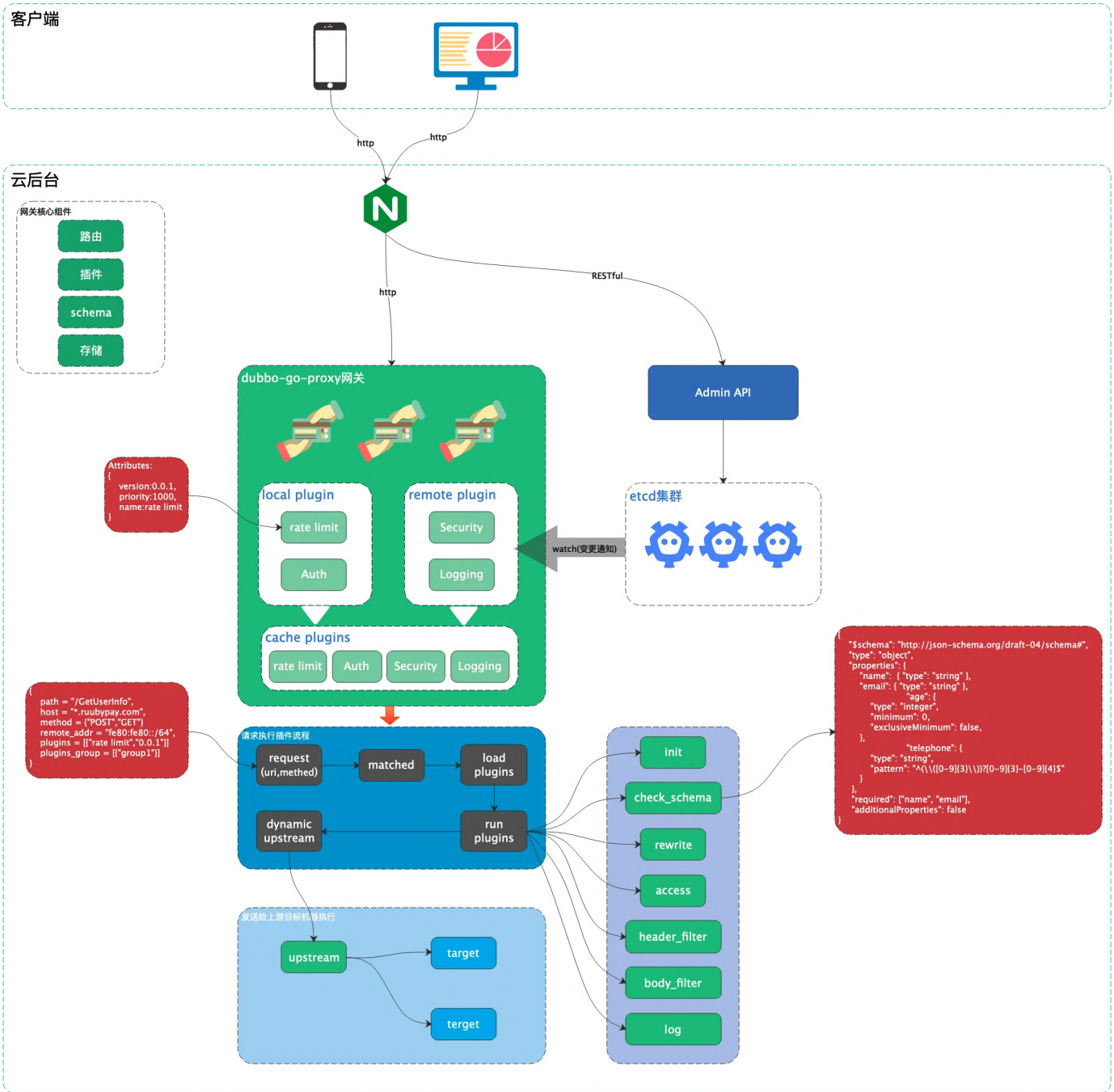
```
go build –o plugin.so –buildmode=plugin .
```

对应要编译的go文件：samples/plugins/plugin.go

Plugin调研与方案

第一次讨论

> 郑先乐提出的整体架构



第二次讨论

吕梦超同学主导

目标：实现在独立打包的可配置可动态插拔与升级的插件型Filter

Go plugin还未成熟，若要实现在Proxy项目外打包，受制于以下几个条件

- 插件实现和主应用程序都必须使用**完全相同的 Go 工具链版本和完全相同的 GOPATH 构建**

  解决方案：提供镜像用于打包插件，插件源文件需放置在对应的目录下。

- 在插件中的任何**直接**依赖项应该与主应用程序中的依赖项版本相同

  解决方案：提取公共依赖项 filter interface至https://github.com/dubbogo/dubbo-go-proxy-filter，插件引用的interface需要与Proxy引用的版本一致。

  Logger有没有必要提取出来?

- 插件不可卸载

  解决方案: 会占用内存，但不影响插拔

方案实现步骤：

- [x] 提取filter interface
- [x] 配置不可插拔的插件
- [ ] Samples
- [ ] 插件可配置参数
- [ ] 动态插拔（依赖于配置中心）

具体实现

Filter interface

https://github.com/dubbogo/dubbo-go-proxy-filter

> 原则：提取的接口，尽量少提取type，不提取不必要的Method
> 需要提取的有三大类：
> - Filter interface
> - router.API
> - api.APi

```
//直接提取即可
type Filter interface {
    Do() context.FilterFunc
}
//需要提取Context
type FilterFunc func(Context)

type Context interface {
   ...      //忽略无依赖的func
  API(router.API) // router.API 需要提取
  GetAPI() *router.API
  Api(api *api.Api) //api.API 需要提取
  GetApi() *api.Api
}

router.API
type API struct {
  URLPattern    string `json:"urlPattern" yaml:"urlPattern"`
  //config.Method需要提取
```

```
    config.Method `json:"method,inline" yaml:"method,inline"`
    Headers       map[string]string `json:"headers,omitempty" yaml:"headers,omitempty"`
}
```

config.Method

//config\api_config.go

API config 中分为两类，struct互相依赖全部提取，**初始化与加载**留在Proxy.

Proxy中剩余的api_config.go

```
var (
    apiConfig *fc.APIConfig
    once      sync.Once
)
// LoadAPIConfigFromFile load the api config from file
func LoadAPIConfigFromFile(path string) (*fc.APIConfig, error) {}

// GetAPIConf returns the initted api config
func GetAPIConf() fc.APIConfig {
    return *apiConfig
}
```

提取至fitler的，link

api.Api

```
type Api struct {
    Name          string      `json:"name" yaml:"name"`
    ITypeStr      string      `json:"itype" yaml:"itype"`
    IType         ApiType     `json:"-" yaml:"-"`
    OTypeStr      string      `json:"otype" yaml:"otype"`
    OType         ApiType     `json:"-" yaml:"-"`
    Status        Status      `json:"status" yaml:"status"`
    Metadata      interface{} `json:"metadata" yaml:"metadata"`
    Method        string      `json:"method" yaml:"method"`
    RequestMethod `json:",omitempty" yaml:"-"`
}
```

整个提取，并把base.go中所有相关type提取

pixiu精简配置技术方案

背景

目前pixiu项目的配置是用户编写yaml文件并指定文件路径进行导入，存在以下问题：

- pixiu作为网关，需要配置的选项多且杂，用户配置可能难以抓住重点，容易被劝退。
- yaml文件对填写格式有较严格的要求，用户在编写和修改配置文件时容易出错。

因此，为了让用户在配置时少走弯路，需要对pixiu的配置进行精简，同时提供更多符合用户使用习惯的配置方式。

思路

回归pixiu本质，resources模块作为必填配置，用户只需配置提供协议转换的配置，即可实现最简单的跨协议调用。其它如：listener、router、filter作为可选配置，由pixiu设置初始值，用户可自行修改。

配置规则如下：

- 若必填配置项有缺失，直接抛出panic。
- 可选配置不填则设定为默认值，默认值由pixiu指定。

必填配置（10%）

配置方式

yaml，api（TODO）

配置项

协议转换配置是必须的，下面通过“#”注释掉的部分表示可以优化掉。

api_config.yaml

```yaml
resources:
  - path: '/api/v1/test-dubbo/userByName'
    type: restful                              #可选值：restful、grpc，下面requestType就不用再选择
    #description: user
    methods:
      - httpVerb: GET
        #onAir: true
        #timeout: 100s
        #inboundRequest:
          #requestType: http
        integrationRequest:
          requestType: dubbo
          mappingParams:
            - name: queryStrings.name
              mapTo: 0
              mapType: "string"
          applicationName: "UserService"
          interface: "com.dubbogo.pixiu.UserService"    #不明白interface代表
          method: "GetUserByName"
          #group: "test"
          #version: 1.0.0
          #clusterName: "test_dubbo"          #可选，不填则默认路由到本地
```

可选配置（90%）

配置方式

yaml

配置项

listener、router、filter可选，不填为系统默认值。尤其是对于timeout、log、port这类信息，绝大多数情况下用户不会修改配置，由pixiu提供较通用的默认配置。

api_config.yaml

```yaml
rateLimit:
  resources:
    - name: test-dubbo
      items:
        #Exact
        - matchStrategy: 0
          pattern: "/api/v1/test-dubbo/user"
        #Regex
        - matchStrategy: 1
          pattern: "/api/v1/test-dubbo/user/*"
  rules:
    #qps sample At most 100 requests can be passed in 1000ms, so qps is 100
    - enable: true
      flowRule:
        #the resource's name
        resource: "test-dubbo"
        threshold: 100
        statintervalinms: 1000
```

conf.yaml

YAML

```yaml
static_resources:
  listeners:
    - name: "net/http"
      address:
        socket_address:
          protocol_type: "HTTP"
          address: "0.0.0.0"
          port: 8888
      filter_chains:
        - filter_chain_match:
            domains:
              - api.dubbo.com
              - api.pixiu.com
          filters:
            - name: dgp.filters.http_connect_manager
              config:
                route_config:
                  routes:
                    - match:
                        prefix: "/api/v1"
                        headers:
                          - name: "X-DGP-WAY"
                            value: "dubbo"
                      route:
                        cluster: "test-dubbo"
                        cluster_not_found_response_code: 505
                        cors:
                          allow_origin:
                            - "*"
                          enabled: true
                  authority_config:
                    authority_rules:
                      - strategy: "Blacklist"
                        limit: "IP"
                        items:
                          - "127.0.0.1"
                      - strategy: "Whitelist"
                        limit: "App"
                        items:
                          - "test_dubbo"
                http_filters:
                  - name: dgp.filters.http.authority_filter
                    config:
                server_name: "test_http_dubbo"
                generate_request_id: false
```

```yaml
      config:
        idle_timeout: 5s
        read_timeout: 5s
        write_timeout: 5s
  clusters:
    - name: "test_dubbo"
      lb_policy: "RoundRobin"
      registries:
        "zookeeper":
          protocol: "zookeeper"
          timeout: "3s"
          address: "127.0.0.1:2181"
          username: ""
          password: ""
  timeout_config:
    connect_timeout: "5s"
    request_timeout: "10s"
  shutdown_config:
    timeout: "60s"
    step_timeout: "10s"
    reject_policy: "immediacy"
  pprofConf:
    enable: true
    address:
      socket_address:
        address: "0.0.0.0"
        port: 6060
  accessLog:
    enable: true
    outputpath: C:\Users\60125\Desktop\dubbo-go\logs\dubbo-go-pixiu-access
metric:
  enable: true
  prometheus_port: 2222
```

log.yaml

YAML

```yaml
level: "debug"
development: true
disableCaller: false
disableStacktrace: false
sampling:
encoding: "console"

# encoder
encoderConfig:
  messageKey: "message"
  levelKey: "level"
  timeKey: "time"
  nameKey: "logger"
  callerKey: "caller"
  stacktraceKey: "stacktrace"
  lineEnding: ""
  levelEncoder: "capitalColor"
  timeEncoder: "iso8601"
  durationEncoder: "seconds"
  callerEncoder: "short"
  nameEncoder: ""

outputPaths:
  - "stderr"
errorOutputPaths:
  - "stderr"
initialFields:
```

其它配置方式（TODO）

考虑引入XML、JSON、pixiu api等配置方式，或者多种配置方式相结合。