Nepxion Discovery

Nepxion Discovery是一款对Spring Cloud Discovery服务注册发现、Ribbon负载均衡、Feign和RestTemplate调用、Hystrix或者阿里巴巴Sentinel熔断隔离限流降级的增强中间件，其功能包括灰度发布（包括切换发布和平滑发布）、服务隔离、服务路由（包括多机房区域路由、多版本路由和多IP和端口路由）、服务权重、黑/白名单的IP地址过滤、限制注册、限制发现等，支持Eureka、Consul、Zookeeper和阿里巴巴的Nacos为服务注册发现中间件，支持阿里巴巴的Nacos、携程的Apollo和Redis为远程配置中心，支持Spring Cloud Gateway、Zuul网关和微服务的灰度发布，支持多数据源的数据库灰度发布等客户特色化灰度发布，支持用户自定义和编程灰度路由策略（包括RPC和REST两种调用方式），支持运维调度灰度发布和路由的元数据，兼容Spring Cloud Edgware版、Finchley版和Greenwich版。现有的Spring Cloud微服务很方便引入该中间件，代码零侵入。鉴于Spring Cloud官方对Eureka和Hystrix不再做新功能的迭代，推荐用Nacos和Sentinel，它们对Spring Cloud灰度发布和路由更具出色的兼容性和友好性

:100:鸣谢

- 感谢阿里巴巴中间件Nacos和Sentinel团队，尤其是Nacos负责人@于怀，Sentinel负责人@子衿，Spring Cloud Alibaba负责人@亦盏的技术支持
- 感谢携程Apollo团队，尤其是@宋顺，特意开发OpenApi包和技术支持
- 感谢代码贡献者@esun，@JikaiSun，@HaoHuang，@Fan Yang，@Ankeway等同学，感谢为本框架提出宝贵意见和建议的同学
- 感谢使用本框架的公司和企业

使用方便，只需要如下步骤：

- 引入相关依赖到pom.xml，参考 依赖兼容
- 操作配置文件，参考 配置文件
  - 在元数据MetaData中，为微服务定义一个版本号（version），定义一个所属组名（group）或者应用名（application），定义一个所属区域（region）名
  - 根据项目实际情况，开启和关闭相关功能项或者属性值，达到最佳配置
- 规则推送，参考 规则定义
  - 通过远程配置中心推送规则
  - 通过控制平台界面推送规则
  - 通过客户端工具（例如Postman）推送

兼容性强。支持如下版本：

Spring Cloud版本	Spring Boot版本	Spring Cloud Alibaba	Nepxion Discovery版本	备注
Hoxton.-	-	-	-	敬请期待
Greenwich.SR1	2.1.4.RELEASE	0.9.0.RELEASE	5.0.8	维护中，可用
Finchley.SR3	2.0.7.RELEASE	0.2.2.RELEASE	4.8.13	维护中，可用
Edgware.SR5	1.5.18.RELEASE	0.1.2.RELEASE	3.8.13	维护中，可用
Dalston.-	-	-	2.0.11	不维护，不可用
Camden.-	-	-	1.0.1	不维护，不可用

快速开始

建议循序渐进阅读下面文章，特别是极简示例

- 极简示例
- 入门教程
- 示例演示

目录

请联系我

微信和公众号

□□

界面展示

图形化灰度发布桌面程序 □□□ 图形化灰度发布Web平台 □ 集成规则配置的Apollo配置中心 □ 集成规则配置的Nacos配置中心 □ Nacos服务注册发现中心 □ Spring Boot Admin监控平台 □ 集成Spring Boot Admin（F版或以上）监控平台，实现通过JMX向Endpoint推送规则和版本，实现灰度发布 □ 集成Spring Boot Admin（E版）监控平台，实现通过JMX向Endpoint推送规则和版本，实现灰度发布 □ 集成Sentinel熔断隔离限流降级平台 □□ 集成健康检查的Consul界面 □

现有痛点

现有的Spring Cloud微服务痛点

- 如果你是运维负责人，是否会经常发现，你掌管的测试环境中的服务注册中心，被一些不负责的开发人员把他本地开发环境注册上来，造成测试人员测试失败。你希望可以把本地开发环境注册给屏蔽掉，不让注册
- 如果你是运维负责人，生产环境的某个微服务集群下的某个实例，暂时出了问题，但又不希望它下线。你希望可以把该实例给屏蔽掉，暂时不让它被调用
- 如果你是业务负责人，鉴于业务服务的快速迭代性，微服务集群下的实例发布不同的版本。你希望根据版本管理策略进行路由，提供给下游微服务区别调用，例如访问控制快速基于版本的不同而切换，例如在不同的版本之间进行流量调拨
- 如果你是业务负责人，希望灰度发布功能可以基于业务场景特色定制，例如根据用户手机号进行不同服务器的路由
- 如果你是DBA负责人，希望灰度发布功能可以基于数据库切换上

- 如果你是测试负责人，希望对微服务做A/B测试，那么通过动态改变版本达到该目的

应用场景

- 黑/白名单的IP地址注册的过滤
  - 开发环境的本地微服务（例如IP地址为172.16.0.8）不希望被注册到测试环境的服务注册发现中心，那么可以在配置中心维护一个黑/白名单的IP地址过滤（支持全局和局部的过滤）的规则
  - 我们可以通过提供一份黑/白名单达到该效果
- 最大注册数的限制的过滤
  - 当某个微服务注册数目已经达到上限（例如10个），那么后面起来的微服务，将再也不能注册上去
- 黑/白名单的IP地址发现的过滤
  - 开发环境的本地微服务（例如IP地址为172.16.0.8）已经注册到测试环境的服务注册发现中心，那么可以在配置中心维护一个黑/白名单的IP地址过滤（支持全局和局部的过滤）的规则，该本地微服务不会被其他测试环境的微服务所调用
  - 我们可以通过推送一份黑/白名单达到该效果
- 多版本访问的灰度控制
  - A服务调用B服务，而B服务有两个实例（B1、B2），虽然三者相同的服务名，但功能上有差异，需求是在某个时刻，A服务只能调用B1，禁止调用B2。在此场景下，我们在application.properties里为B1维护一个版本为1.0，为B2维护一个版本为1.1
  - 我们可以通过推送A服务调用某个版本的B服务对应关系的配置，达到某种意义上的灰度控制，改变版本的时候，我们只需要再次推送即可
- 多版本权重的灰度控制
  - 上述场景中，我们也可以通过配给不同版本的权重（流量比例），根据需求，A访问B的流量在B1和B2进行调拨
- 多区域权重的灰度控制
  - 上述场景中，我们也可以通过配给不同区域的权重（流量比例），根据需求，A访问B的流量在B1和B2（B1和B2所属不同区域）进行调拨
- 多数据源的数据库灰度控制
  - 我们事先为微服务配置多套数据源，通过灰度发布实时切换数据源
- 动态改变微服务版本
  - 在A/B测试中，通过动态改变版本，不重启微服务，达到访问版本的路径改变
- 用户自定义和编程灰度路由策略，可以通过非常简单编程达到如下效果
  - 在业务REST调用上，在Header上传入服务名和版本对应关系的Json字符串，后端若干个服务会把请求路由到指定版本的服务器上
  - 在业务REST调用上，在Header上传入区域（region）名，后端若干个服务会把请求路由到指定区域（region）名的服务器上
  - 在业务REST调用上，在Header上传入Token，根据不同的Token查询到不同的用户，后端若干个服务会把请求路由到指定的服务器上
  - 在业务RPC调用上，根据不同的业务参数，例如手机号或者身份证号，后端若干个服务会把请求路由到指定的服务器上

功能简介

- 基于Spring Cloud的微服务和Spring Cloud Gateway和Zuul网关实现下述功能，它具有几个特性
  - 具有极大的灵活性 - 支持在任何环节（微服务和两个网关），多种方式（REST和RPC）做过滤控制和灰度发布
  - 具有极小的限制性 - 只要开启了服务注册发现，程序入口加了@EnableDiscoveryClient注解
  - 具有极强的健壮性 - 当远程配置中心全部挂了，可以通过Rest方式进行灰度发布；当远程规则配置不规范，马上切换到本地规则来代替
  - 具有极简的易用性 - 只需要引入相关的包同时规则里含有了对应的配置，该功能将自然启动
- 实现服务注册层面的控制
  - 基于黑/白名单的IP地址过滤机制禁止对相应的微服务进行注册
  - 基于最大注册数的限制微服务注册。一旦微服务集群下注册的实例数目已经达到上限，将禁止后续的微服务进行注册
- 实现服务发现层面的控制
  - 基于黑/白名单的IP地址过滤机制禁止对相应的微服务被发现
  - 基于版本号配对，通过对消费端和提供端可访问版本对应关系的配置，在服务发现和负载均衡层面，进行多版本访问控制
  - 基于版本权重配对，通过对消费端和提供端版本权重（流量）对应关系的配置，在服务发现和负载均衡层面，进行多版本流量调拨访问控制
  - 基于区域权重配对，通过对消费端和提供端所属区域的权重（流量）对应关系的配置，在服务发现和负载均衡层面，进行多区域流量调拨访问控制
- 实现用户业务层面的控制
  - 使用者可以通过订阅业务参数的变化，实现特色化的灰度发布，例如，多数据源的数据库切换的灰度发布
- 实现灰度发布
  - 通过版本的动态改变，实现切换灰度发布
  - 通过版本访问规则的改变，实现切换灰度发布
  - 通过版本权重规则的改变，实现平滑灰度发布
  - 通过区域权重规则的改变，实现平滑灰度发布
- 实现通过XML或者Json进行上述规则的定义
- 实现通过事件总线机制（EventBus）的功能，实现发布/订阅功能
  - 对接远程配置中心，集成Nacos和Redis，异步接受远程配置中心主动推送规则信息，动态改变微服务的规则
  - 结合Spring Boot Actuator，异步接受Rest主动推送规则信息，动态改变微服务的规则，支持同步和异步推送两种方式
  - 结合Spring Boot Actuator，动态改变微服务的版本，支持同步和异步推送两种方式
  - 在服务注册层面的控制中，一旦禁止注册的条件触发，主动推送异步事件，以便使用者订阅
- 实现通过Listener机制进行扩展

    - 使用者可以对服务注册发现核心事件进行监听
- 实现通过策略扩展，用户自定义和编程灰度路由策略
    - 使用者可以实现跟业务有关的路由策略，根据业务参数的不同，负载均衡到不同的服务器
    - 使用者可以根据内置的版本路由策略+区域路由策略+IP和端口路由策略+自定义策略，随心所欲的达到需要的路由功能
- 实现支持Spring Boot Actuator和Swagger集成
- 实现支持Spring Boot Admin的集成
- 实现支持Sentinel熔断隔离限流降级的集成
- 实现支持未来扩展更多的服务注册中心
- 实现控制平台微服务，支持对规则和版本集中管理、推送、更改和删除
- 实现基于控制平台微服务的图形化的灰度发布功能

名词解释

- E版和F版，即Spring Cloud的Edgware和Finchley的首字母，以此类推
- 切换灰度发布（也叫刚性灰度发布）和平滑灰度发布（也叫柔性灰度发布），切换灰度发布即在灰度发布的时候，没有过渡过程，流量直接从旧版本切换到新版本；平滑灰度发布即在灰度发布的时候，有个过渡过程，可以根据实际情况，先给新版本分配低额流量，给旧版本分配高额流量，对新版本进行监测，如果没有问题，就继续把旧版的流量切换到新版本上
- IP地址，即根据微服务上报的它所在机器的IP地址。本系统内部强制以IP地址上报，禁止HostName上报，杜绝Spring Cloud应用在Docker或者Kubernetes部署时候出现问题
- 规则定义和策略定义，规则定义即通过XML或者Json定义既有格式的规则；策略定义即专指用户自定义和编程灰度路由的策略，属于编程方式的一种扩展
- 本地版本，即初始化读取本地配置文件获取的版本，也可以是第一次读取远程配置中心获取的版本。本地版本和初始版本是同一个概念
- 动态版本，即灰度发布时的版本。动态版本和灰度版本是同一个概念
- 本地规则，即初始化读取本地配置文件获取的规则，也可以是第一次读取远程配置中心获取的规则。本地规则和初始规则是同一个概念
- 动态规则，即灰度发布时的规则。动态规则和灰度规则是同一个概念
- 事件总线，即基于Google Guava的EventBus构建的组件。通过事件总线可以推送动态版本和动态规则的更新和删除
- 远程配置中心，即可以存储规则配置XML格式的配置中心，可以包括不限于Nacos，Redis，Apollo，DisConf，Spring Cloud Config
- 配置（Config）和规则（Rule），在本系统中属于同一个概念，例如更新配置，即更新规则；例如远程配置中心存储的配置，即规则XML
- 服务端口和管理端口，即服务端口指在配置文件的server.port值，管理端口指management.port（E版）值或者management.server.port（F版或以上）值

架构工程

架构

全局架构图

从上图，可以分析出两种基于网关的灰度发布方案，您可以研究更多的灰度发布策略

:triangular_flag_on_post:基于网关版本权重的灰度发布

- 灰度发布前
    - 网关不需要配置版本
    - 网关->服务A(V1.0)，网关配给服务A(V1.0)的100%权重（流量）
- 灰度发布中
    - 上线服务A(V1.1)
    - 在网关层调拨10%权重（流量）给A(V1.1)，给A(V1.0)的权重（流量）减少到90%
    - 通过观测确认灰度有效，把A(V1.0)的权重（流量）全部切换到A(V1.1)
- 灰度发布后
    - 下线服务A(V1.0)，灰度成功

:triangular_flag_on_post:基于网关版本切换的灰度发布

- 灰度发布前
    - 假设当前生产环境，调用路径为网关(V1.0)->服务A(V1.0)
    - 运维将发布新的生产环境，部署新服务集群，服务A(V1.1)
    - 由于网关(1.0)并未指向服务A(V1.1)，所以它们是不能被调用的
- 灰度发布中
    - 新增用作灰度发布的网关(V1.1)，指向服务A(V1.1)
    - 灰度网关(V1.1)发布到服务注册发现中心，但禁止被服务发现，网关外的调用进来无法负载均衡到网关(V1.1)上
    - 在灰度网关(V1.1)->服务A(V1.1)这条调用路径做灰度测试
    - 灰度测试成功后，把网关(V1.0)指向服务A(V1.1)
- 灰度发布后
    - 下线服务A(V1.0)，灰度成功
    - 灰度网关(V1.1)可以不用下线，留作下次版本上线再次灰度发布
    - 如果您对新服务比较自信，可以更简化，可以不用灰度网关和灰度测试，当服务A(V1.1)上线后，原有网关直接指向服务A(V1.1)，然后下线服务A(V1.0)

模块结构图

工程

工程名	描述
discovery-common	通用模块
discovery-common-apollo	封装Apollo通用操作逻辑
discovery-common-nacos	封装Nacos通用操作逻辑
discovery-common-redis	封装Redis通用操作逻辑
discovery-plugin-framework	核心框架
discovery-plugin-framework-eureka	核心框架服务注册发现的Eureka实现
discovery-plugin-framework-consul	核心框架服务注册发现的Consul实现
discovery-plugin-framework-zookeeper	核心框架服务注册发现的Zookeeper实现
discovery-plugin-framework-nacos	核心框架服务注册发现的Nacos实现
discovery-plugin-config-center	配置中心实现
discovery-plugin-config-center-starter-apollo	配置中心的Apollo Starter
discovery-plugin-config-center-starter-nacos	配置中心的Nacos Starter
discovery-plugin-config-center-starter-redis	配置中心的Redis Starter
discovery-plugin-admin-center	管理中心实现
discovery-plugin-starter-eureka	Eureka Starter
discovery-plugin-starter-consul	Consul Starter
discovery-plugin-starter-zookeeper	Zookeeper Starter
discovery-plugin-starter-nacos	Nacos Starter
discovery-plugin-strategy	用户自定义和编程灰度路由策略
discovery-plugin-strategy-starter-service	用户自定义和编程灰度路由策略的Service Starter
discovery-plugin-strategy-starter-zuul	用户自定义和编程灰度路由策略的Zuul Starter
discovery-plugin-strategy-starter-gateway	用户自定义和编程灰度路由策略的Spring Cloud Gateway Starter
discovery-plugin-strategy-starter-hystrix	用户自定义和编程灰度路由策略下，Hystrix做线程模式的服务隔离必须引入的插件 Starter
discovery-console	控制平台，集成接口给UI
discovery-console-starter-apollo	控制平台的Apollo Starter
discovery-console-starter-nacos	控制平台的Nacos Starter
discovery-console-starter-redis	控制平台的Redis Starter
discovery-console-desktop	图形化灰度发布等桌面程序
discovery-springcloud-example-admin	Spring Boot Admin服务台示例
discovery-springcloud-example-console	控制平台示例

discovery-springcloud-example-eureka	Eureka服务器示例
discovery-springcloud-example-service	用于灰度发布的微服务示例
discovery-springcloud-example-zuul	用于灰度发布的Zuul示例
discovery-springcloud-example-gateway	用于灰度发布的Spring Cloud Gateway示例

依赖兼容

依赖

:exclamation:下面标注[必须引入]是一定要引入的包，标注[选择引入]是可以选择一个引入，或者不引入

核心插件引入，支持微服务端、网关Zuul端和网关Spring Cloud Gateway端，包括核心灰度发布功能，管理中心，配置中心等

```
[必须引入] 四个服务注册发现的中间件的增强插件，请任选一个引入
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-plugin-starter-eureka</artifactId>
    <artifactId>discovery-plugin-starter-consul</artifactId>
    <artifactId>discovery-plugin-starter-zookeeper</artifactId>
    <artifactId>discovery-plugin-starter-nacos</artifactId>
    <version>${discovery.version}</version>
</dependency>

[选择引入] 三个远程配置中心的中间件的扩展插件，如需要，请任选一个引入，或者也可以引入您自己的扩展
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-plugin-config-center-starter-apollo</artifactId>
    <artifactId>discovery-plugin-config-center-starter-nacos</artifactId>
    <artifactId>discovery-plugin-config-center-starter-redis</artifactId>
    <version>${discovery.version}</version>
</dependency>
```

扩展功能引入，支持微服务端、网关Zuul端和网关Spring Cloud Gateway端，包括内置版本路由、区域路由、用户自定义和编程灰度路由

```
微服务端引入
[选择引入] 用户自定义和编程灰度路由，如需要，请引入
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-plugin-strategy-starter-service</artifactId>
    <version>${discovery.version}</version>
</dependency>

网关Zuul端引入
[选择引入] 用户自定义和编程灰度路由，如需要，请引入
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-plugin-strategy-starter-zuul</artifactId>
    <version>${discovery.version}</version>
</dependency>

网关Spring Cloud Gateway端引入
[选择引入] 用户自定义和编程灰度路由，如需要，请引入
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-plugin-strategy-starter-gateway</artifactId>
    <version>${discovery.version}</version>
</dependency>
```

[选择引入] 用户自定义和编程灰度路由时候，Hystrix做线程模式的服务隔离必须引入的插件，信号量模式不需要引入 com.nepxion discovery-plugin-strategy-starter-hystrix ${discovery.version}

````
控制平台引入
```xml
[选择引入] 三个远程配置中心的中间件的扩展插件，如需要，请任选一个引入，或者也可以引入您自己的扩展
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-console-starter-apollo</artifactId>
    <artifactId>discovery-console-starter-nacos</artifactId>
    <artifactId>discovery-console-starter-redis</artifactId>
    <version>${discovery.version}</version>
</dependency>
````

:warning:特别注意：中间件的引入一定要在所有层面保持一致，绝不允许出现类似如下情况，这也是常识
````

- 例如，网关用Eureka做服务注册发现，微服务用Consul做服务注册发现
- 例如，控制平台用Nacos做远程配置中心，微服务用Redis做远程配置中心

:star:如果只想要“用户自定义和编程灰度路由”功能，而不想要灰度发布功能

- “用户自定义和编程灰度路由”是不需要接入远程配置中心的，所以建议去除远程配置中心包的引入

```
<dependency>
    <groupId>com.nepxion</groupId>
    <artifactId>discovery-plugin-config-center-starter-xxx</artifactId>
    <version>${discovery.version}</version>
</dependency>
```

- “用户自定义和编程灰度路由”是不会对服务注册发现等逻辑产生影响，所以建议下面两项配置改为false

```
# 开启和关闭服务注册层面的控制。一旦关闭，服务注册的黑/白名单过滤功能将失效，最大注册数的限制过滤功能将失效。缺失则默认为true
spring.application.register.control.enabled=false
# 开启和关闭服务发现层面的控制。一旦关闭，服务多版本调用的控制功能将失效，动态屏蔽指定IP地址的服务实例被发现的功能将失效。缺失则默认为true
spring.application.discovery.control.enabled=false
```

兼容

版本兼容情况

- Spring Cloud F版或以上，请采用4.x.x版本，具体代码参考master分支
- Spring Cloud E版，请采用3.x.x版本，具体代码参考Edgware分支
- 4.x.x版本和3.x.x版本功能完全一致，但在Endpoint的URL使用方式上稍微有个小的区别。例如
  - 3.x.x的Endpoint URL为http://localhost:5100/config/view
  - 4.x.x的Endpoint URL为http://localhost:5100/actuator/config/config/view，注意，路径中config为两个，前面那个是Endpoint Id，Spring Boot 2.x.x规定Endpoint Id必须指定，且全局唯一

中间件兼容情况

- Consul
  - Consul服务器版本不限制，推荐用最新版本，从https://releases.hashicorp.com/consul/获取
- Zookeeper
  - Spring Cloud F版或以上，必须采用Zookeeper服务器的3.5.x服务器版本（或者更高），从http://zookeeper.apache.org/releases.html#download获取
  - Spring Cloud E版，Zookeeper服务器版本不限制
- Eureka
  - 跟Spring Cloud版本保持一致，自行搭建服务器
- Apollo
  - Apollo服务器版本，推荐用最新版本，从https://github.com/ctripcorp/apollo/releases获取
- Nacos
  - Nacos服务器版本，推荐用最新版本，从https://github.com/alibaba/nacos/releases获取
- Redis
  - Redis服务器版本，推荐用最新版本，从https://redis.io/获取

规则定义

规则是基于XML或者Json为配置方式，存储于本地文件或者远程配置中心，可以通过远程配置中心修改的方式达到规则动态化。其核心代码参考discovery-plugin-framework以及它的扩展、discovery-plugin-config-center以及它的扩展和discovery-plugin-admin-center等

规则示例

XML示例（Json示例见discovery-springcloud-example-service下的rule.json）

:warning:特别注意：服务名大小写规则

- 在配置文件（application.properties、application.yaml等）里，定义服务名（spring.application.name）不区分大小写
- 在规则文件（XML、Json）里，引用的服务名必须小写
- 在Nacos、Apollo、Redis等远程配置中心的Key，包含的服务名必须小写

```
<?xml version="1.0" encoding="UTF-8"?>
<rule>
    <!-- 如果不想开启相关功能，只需要把相关节点删除即可，例如不想要黑名单功能，把blacklist节点删除 -->
    <register>
        <!-- 服务注册的黑/白名单注册过滤，只在服务启动的时候生效。白名单表示只允许指定IP地址前缀注册，黑名单表示不允许指定IP地址前缀注册。每个服务只能同时开启要么白名单，要么黑名单 -->
        <!-- filter-type，可选值blacklist/whitelist，表示白名单或者黑名单 -->
        <!-- service-name，表示服务名 -->
        <!-- filter-value，表示黑/白名单的IP地址列表。IP地址一般用前缀来表示，如果多个用“;”分隔，不允许出现空格 -->
        <!-- 表示下面所有服务，不允许10.10和11.11为前缀的IP地址注册（全局过滤） -->
        <blacklist filter-value="10.10;11.11">
```

```
            <!-- 表示下面服务，不允许172.16和10.10和11.11为前缀的IP地址注册 -->
            <service service-name="discovery-springcloud-example-a" filter-value="172.16"/>
        </blacklist>

        <!-- <whitelist filter-value="">
            <service service-name="" filter-value=""/>
        </whitelist>  -->

        <!-- 服务注册的数目限制注册过滤，只在服务启动的时候生效。当某个服务的实例注册达到指定数目时候，更多的实例将无法注册 -->
        <!-- service-name，表示服务名 -->
        <!-- filter-value，表示最大实例注册数 -->
        <!-- 表示下面所有服务，最大实例注册数为10000（全局配置） -->
        <count filter-value="10000">
            <!-- 表示下面服务，最大实例注册数为5000，全局配置值10000将不起作用，以局部配置值为准 -->
            <service service-name="discovery-springcloud-example-a" filter-value="5000"/>
        </count>
    </register>

    <discovery>
        <!-- 服务发现的黑/白名单发现过滤，使用方式跟“服务注册的黑/白名单过滤”一致 -->
        <!-- 表示下面所有服务，不允许10.10和11.11为前缀的IP地址被发现（全局过滤） -->
        <blacklist filter-value="10.10;11.11">
            <!-- 表示下面服务，不允许172.16和10.10和11.11为前缀的IP地址被发现 -->
            <service service-name="discovery-springcloud-example-b" filter-value="172.16"/>
        </blacklist>

        <!-- 服务发现的多版本灰度访问控制 -->
        <!-- service-name，表示服务名 -->
        <!-- version-value，表示可供访问的版本，如果多个用“;”分隔，不允许出现空格 -->
        <version>
            <!-- 表示网关g的1.0，允许访问提供端服务a的1.0版本 -->
            <service consumer-service-name="discovery-springcloud-example-gateway" provider-service-name="discovery-springcloud-example-a"
consumer-version-value="1.0" provider-version-value="1.0"/>
            <!-- 表示网关g的1.1，允许访问提供端服务a的1.1版本 -->
            <service consumer-service-name="discovery-springcloud-example-gateway" provider-service-name="discovery-springcloud-example-a"
consumer-version-value="1.1" provider-version-value="1.1"/>
            <!-- 表示网关z的1.0，允许访问提供端服务a的1.0版本 -->
            <service consumer-service-name="discovery-springcloud-example-zuul" provider-service-name="discovery-springcloud-example-a"
consumer-version-value="1.0" provider-version-value="1.0"/>
            <!-- 表示网关z的1.1，允许访问提供端服务a的1.1版本 -->
            <service consumer-service-name="discovery-springcloud-example-zuul" provider-service-name="discovery-springcloud-example-a"
consumer-version-value="1.1" provider-version-value="1.1"/>
            <!-- 表示消费端服务a的1.0，允许访问提供端服务b的1.0版本 -->
            <service consumer-service-name="discovery-springcloud-example-a" provider-service-name="discovery-springcloud-example-b"
consumer-version-value="1.0" provider-version-value="1.0"/>
            <!-- 表示消费端服务a的1.1，允许访问提供端服务b的1.1版本 -->
            <service consumer-service-name="discovery-springcloud-example-a" provider-service-name="discovery-springcloud-example-b"
consumer-version-value="1.1" provider-version-value="1.1"/>
            <!-- 表示消费端服务b的1.0，允许访问提供端服务c的1.0和1.1版本 -->
            <service consumer-service-name="discovery-springcloud-example-b" provider-service-name="discovery-springcloud-example-c"
consumer-version-value="1.0" provider-version-value="1.0;1.1"/>
            <!-- 表示消费端服务b的1.1，允许访问提供端服务c的1.2版本 -->
            <service consumer-service-name="discovery-springcloud-example-b" provider-service-name="discovery-springcloud-example-c"
consumer-version-value="1.1" provider-version-value="1.2"/>
        </version>

        <!-- 服务发现的多版本权重灰度访问控制 -->
        <!-- service-name，表示服务名 -->
        <!-- version-value，表示版本对应的权重值，格式为"版本值=权重值"，如果多个用“;”分隔，不允许出现空格 -->
        <weight>
            <!-- 权重流量配置有如下三种方式，粒度由细到粗，优先级分别是由高到底，即先从第一种方式取权重流量值，取不到则到第二种方式取值，
再取不到则到第二种方式取值，再取不到则忽略。使用者按照实际情况，选择一种即可 -->
            <!-- 表示消费端服务b访问提供端服务c的时候，提供端服务c的1.0版本提供90%的权重流量，1.1版本提供10%的权重流量 -->
            <service consumer-service-name="discovery-springcloud-example-b" provider-service-name="discovery-springcloud-example-c"
provider-weight-value="1.0=90;1.1=10"/>
            <!-- 表示所有消费端服务访问提供端服务c的时候，提供端服务c的1.0版本提供80%的权重流量，1.1版本提供20%的权重流量 -->
            <service provider-service-name="discovery-springcloud-example-c" provider-weight-value="1.0=80;1.1=20"/>
            <!-- 表示外界调用进来后，区域为dev的服务提供85%的权重流量，区域为qa的服务提供15%的权重流量 -->
            <!-- <region provider-weight-value="dev=85;qa=15"/> -->
        </weight>
    </discovery>

    <strategy>
        <!-- <version>{"discovery-springcloud-example-a":"1.0", "discovery-springcloud-example-b":"1.0", "discovery-springcloud-example-
c":"1.0;1.2"}</version> -->
        <!-- <version>1.0</version> -->
        <!-- <region>{"discovery-springcloud-example-a":"qa;dev", "discovery-springcloud-example-b":"dev", "discovery-springcloud-example-
c":"qa"}</region> -->
        <!-- <region>dev</region> -->
        <!-- <address>{"discovery-springcloud-example-a":"192.168.43.101:1100", "discovery-springcloud-example-b":"192.168.43.101:1201",
"discovery-springcloud-example-c":"192.168.43.101:1300"}</address> -->
    </strategy>

    <!-- 客户定制化控制，由远程推送客户化参数的改变，实现一些特色化的灰度发布，例如，基于数据库的灰度发布 -->
    <customization>
        <!-- 服务a和c分别有两个库的配置，分别是测试数据库（database的value为qa）和生产数据库（database的value为prod） -->
        <!-- 上线后，一开始数据库指向测试数据库，对应value为qa，然后灰度发布的时候，改对应value为prod，即实现数据库的灰度发布 -->
```

```
        <service service-name="discovery-springcloud-example-a" key="database" value="qa"/>
        <service service-name="discovery-springcloud-example-c" key="database" value="prod"/>
    </customization>
</rule>
```

黑/白名单的IP地址注册的过滤规则

微服务启动的时候，禁止指定的IP地址注册到服务注册发现中心。支持黑/白名单，白名单表示只允许指定IP地址前缀注册，黑名单表示不允许指定IP地址前缀注册。规则如何使用，见示例说明

- 全局过滤，指注册到服务注册发现中心的所有微服务，只有IP地址包含在全局过滤字段的前缀中，都允许注册（对于白名单而言），或者不允许注册（对于黑名单而言）
- 局部过滤，指专门针对某个微服务而言，那么真正的过滤条件是全局过滤+局部过滤结合在一起

最大注册数的限制的过滤规则

微服务启动的时候，一旦微服务集群下注册的实例数目已经达到上限（可配置），将禁止后续的微服务进行注册。规则如何使用，见示例说明

- 全局配置值，只下面配置所有的微服务集群，最多能注册多少个
- 局部配置值，指专门针对某个微服务而言，那么该值如存在，全局配置值失效

黑/白名单的IP地址发现的过滤规则

微服务启动的时候，禁止指定的IP地址被服务发现。它使用的方式和“黑/白名单的IP地址注册的过滤规则”一致

版本访问的灰度发布规则

```
1. 标准配置，举例如下
   <service consumer-service-name="a" provider-service-name="b" consumer-version-value="1.0" provider-version-value="1.0,1.1"/> 表示消费端1.0版本，允许访问提供端1.0和1.1版本
2. 版本值不配置，举例如下
   <service consumer-service-name="a" provider-service-name="b" provider-version-value="1.0,1.1"/> 表示消费端任何版本，允许访问提供端1.0和1.1版本
   <service consumer-service-name="a" provider-service-name="b" consumer-version-value="1.0"/> 表示消费端1.0版本，允许访问提供端任何版本
   <service consumer-service-name="a" provider-service-name="b"/> 表示消费端任何版本，允许访问提供端任何版本
3. 版本值空字符串，举例如下
   <service consumer-service-name="a" provider-service-name="b" consumer-version-value="" provider-version-value="1.0,1.1"/> 表示消费端任何版本，允许访问提供端1.0和1.1版本
   <service consumer-service-name="a" provider-service-name="b" consumer-version-value="1.0" provider-version-value=""/> 表示消费端1.0版本，允许访问提供端任何版本
   <service consumer-service-name="a" provider-service-name="b" consumer-version-value="" provider-version-value=""/> 表示消费端任何版本，允许访问提供端任何版本
4. 版本对应关系未定义，默认消费端任何版本，允许访问提供端任何版本

特殊情况处理，在使用上需要极力避免该情况发生
1. 消费端的application.properties未定义版本号，则该消费端可以访问提供端任何版本
2. 提供端的application.properties未定义版本号，当消费端在xml里不做任何版本配置，才可以访问该提供端
```

版本权重的灰度发布规则

```
1. 标准配置，举例如下
    <service consumer-service-name="a" provider-service-name="b" provider-weight-value="1.0=90;1.1=10"/> 表示消费端访问提供端的时候，提供端的1.0版本提供90%的权重流量，1.1版本提供10%的权重流量
    <service provider-service-name="b" provider-weight-value="1.0=90;1.1=10"/> 表示所有消费端访问提供端的时候，提供端的1.0版本提供90%的权重流量，1.1版本提供10%的权重流量
2. 局部配置，即指定consumer-service-name，专门为该消费端配置权重。全局配置，即不指定consumer-service-name，为所有消费端配置相同情形的权重。当局部配置和全局配置同时存在的时候，以局部配置优先
3. 尽量为线上所有版本都赋予权重值
```

区域权重的灰度发布规则

```
1. 标准配置，举例如下
    <region provider-weight-value="dev=85;qa=15"/> 表示区域为dev的服务提供85%的权重流量，区域为qa的服务提供15%的权重流量
2. 区域权重可以切换整条调用链的权重配比
3. 尽量为线上所有区域都赋予权重值
```

全链路路由策略的灰度发布规则

```
1. 标准配置，举例如下
    <strategy>
        <!-- <version>{"discovery-springcloud-example-a":"1.0", "discovery-springcloud-example-b":"1.0", "discovery-springcloud-example-c":"1.0;1.2"}</version> --> 表示全链路调用中，按照配置的版本号的对应服务去调用
        <!-- <version>1.0</version> --> 表示全链路调用中，所有服务都调用1.0版本
        <!-- <region>{"discovery-springcloud-example-a":"qa;dev", "discovery-springcloud-example-b":"dev", "discovery-springcloud-example-c":"qa"}</region> --> 表示全链路调用中，按照配置的区域的对应服务去调用
        <!-- <region>dev</region> --> 表示全链路调用中，所有服务都调用dev区域的服务
```

```
        <!-- <address>{"discovery-springcloud-example-a":"192.168.43.101:1100", "discovery-springcloud-example-b":"192.168.43.101:1201",
"discovery-springcloud-example-c":"192.168.43.101:1300"}</address> --> 表示全链路调用中，按照配置的地址的对应服务去调用
    </strategy>
2. 用法和基于Http Header头部传路由参数一致。前置是通过前端或者网关传入，后者是配置在配置文件里。让两者全部启用的时候，以前端或者网关传入Header方式优先
```

:triangular_flag_on_post:注意 路由策略的入口有三个（以{"discovery-springcloud-example-a":"1.0", "discovery-springcloud-example-b":"1.0", "discovery-springcloud-example-c":"1.0;1.2"}）为例：

1. 从外界传入（例如：Postman），在Header上加入n-d-version={"discovery-springcloud-example-a":"1.0", "discovery-springcloud-example-b":"1.0", "discovery-springcloud-example-c":"1.0;1.2"}
2. 在网关Zuul或者Spring Cloud Gateway的Filter中指定
3. 全链路路由策略的灰度发布规则，在配置中心或者本地rule.xml配置

其作用的优先级为外界传入>网关Filter指定>配置中心或者本地rule.xml配置

用户自定义的灰度发布规则

通过订阅业务参数的变化，实现特色化的灰度发布，例如，多数据源的数据库切换的灰度发布

```
1. 标准配置，举例如下
   <service service-name="discovery-springcloud-example-a" key="database" value="prod"/>
2. 上述示例，是基于多数据源的数据库切换的灰度发布
   服务a有两个库的配置，分别是测试数据库（database的value为qa）和生产数据库（database的value为prod）
   上线后，一开始数据库指向测试数据库，对应value为qa，然后灰度发布的时候，改对应value为prod，即实现数据库的灰度发布
```

动态改变规则

微服务启动的时候，由于规则（例如：rule.xml）已经配置在本地，使用者希望改变一下规则，而不重启微服务，达到规则的改变

- 规则分为本地规则和动态规则
- 本地规则是通过在本地规则（例如：rule.xml）文件定义的，也可以从远程配置中心获取，在微服务启动的时候读取
- 动态规则是通过POST方式动态设置，或者由远程配置中心推送设置
- 规则初始化的时候，如果接入了远程配置中心，先读取远程规则，如果不存在，再读取本地规则文件
- 多规则灰度获取规则的时候，先获取动态规则，如果不存在，再获取本地规则
- 规则可以持久化到远程配置中心，一旦微服务死掉后，再次启动，仍旧可以拿到灰度规则，所以动态改变规则策略属于永久灰度发布手段
- 规则推送到远程配置中心可以分为局部推送和全局推送
  - 局部推送是基于Group+ServiceId来推送的，就是同一个Group下同一个ServiceId的服务集群独立拥有一个规则配置，如果采用这种方式，需要在每个微服务集群下做一次灰度发布。优点是独立封闭，本服务集群灰度发布失败不会影响到其它服务集群，缺点是相对繁琐
  - 全局推送是基于Group来推送的（接口参数中的ServiceId由Group来代替），就是同一个Group下所有服务集群共同拥有一个规则配置，如果采用这种方式，只需要做一次灰度发布，所有服务集群都生效。优点是非常简便，缺点具有一定风险，因为这个规则配置掌握着所有服务集群的命运。全局推送用于全链路灰度发布
  - 如果既执行了全局推送，又执行了局部推送，那么，当服务运行中，优先接受最后一次推送的规则；当服务重新启动的时候，优先读取局部推送的规则

动态改变版本

微服务启动的时候，由于版本已经写死在application.properties里，使用者希望改变一下版本，而不重启微服务，达到访问版本的路径改变

- 版本分为本地版本和动态版本
- 本地版本是通过在application.properties里配置的，在微服务启动的时候读取
- 动态版本是通过POST方式动态设置
- 多版本灰度获取版本值的时候，先获取动态版本，如果不存在，再获取本地版本
- 版本不会持久化到远程配置中心，一旦微服务死掉后，再次启动，拿到的还是本地版本，所以动态改变版本策略属于临时灰度发布手段

策略定义

策略是通过REST或者RPC调用传递Header或者参数，达到用户自定义和编程灰度路由的目的。使用者可以实现跟业务有关的路由策略，根据业务参数的不同，负载均衡到不同的服务器，其核心代码参考discovery-plugin-strategy以及它的扩展

服务端的编程灰度路由策略

基于服务端的编程灰度路由，实现DiscoveryEnabledStrategy，通过RequestContextHolder（获取来自网关的Header参数）和ServiceStrategyContext（获取来自RPC方式的方法参数）获取业务上下文参数，进行路由自定义，见示例演示的“用户自定义和编程灰度路由的操作演示”

Zuul端的编程灰度路由策略

基于Zuul端的编程灰度路由，实现DiscoveryEnabledStrategy，通过Zuul自带的RequestContext（获取来自网关的Header参数）获取业务上下文参数，进行路由自定义，见示例演示的“用户自定义和编程灰度路由的操作演示”

Gateway端的编程灰度路由策略

基于Spring Cloud Gateway端的编程灰度路由，实现DiscoveryEnabledStrategy，通过GatewayStrategyContext（获取来自网关的Header参数）获取业务上下文参

数，进行路由自定义，见示例演示的“用户自定义和编程灰度路由的操作演示”

REST调用的内置多版本灰度路由策略

基于Feign/RestTemplate的REST调用的多版本灰度路由，在Header上传入服务名和版本对应关系的Json字符串，如下表示，如果REST请求要经过a，b，c三个服务，那么只有a服务的1.0版本，b服务的1.1版本，c服务的1.1或1.2版本，允许被调用到 Header的Key为"n-d-version"，value为：

```
{"discovery-springcloud-example-a":"1.0", "discovery-springcloud-example-b":"1.1", "discovery-springcloud-example-c":"1.1;1.2"}
```

见示例演示的“用户自定义和编程灰度路由的操作演示”

如果，链路调用中所有的服务都是指定某个版本（例如1.1），那么value的格式可以简化，不需要Json字符串，直接是

```
1.1
```

多版本灰度路由架构图

REST调用的内置多区域灰度路由策略

基于Feign/RestTemplate的REST调用的多区域灰度路由，在Header上传入服务名和版本对应关系的Json字符串，如下表示，如果REST请求要经过a，b，c三个服务，那么只有dev区域的a服务，qa区域的b服务，dev和qa区域c服务，允许被调用到 Header的Key为"n-d-region"，value为：

```
{"discovery-springcloud-example-a":"dev", "discovery-springcloud-example-b":"qa", "discovery-springcloud-example-c":"dev;qa"}
```

见示例演示的“用户自定义和编程灰度路由的操作演示”

如果，链路调用中所有的服务都是指定某个区域（例如dev），那么value的格式可以简化，不需要Json字符串，直接是

```
dev
```

多区域灰度路由架构图

:warning:特别注意：Spring Cloud内置zone的策略，功能跟region策略很相似，但zone策略不能跟用户自定义路由组合使用，故提供了更友好的region策略

REST调用的内置多IP和端口灰度路由策略

基于Feign/RestTemplate的REST调用的多版本灰度路由，在Header上传入服务名和版本对应关系的Json字符串，如下表示，如果REST请求要经过a，b，c三个服务，那么只需要指定三个服务所给定的IP（或者IP和端口组合），允许被调用到 Header的Key为"n-d-address"，value为：

```
{"discovery-springcloud-example-a":"192.168.43.101:1101", "discovery-springcloud-example-b":"192.168.43.101:1201", "discovery-springcloud-example-c":"192.168.43.101:1302"}
```

见示例演示的“用户自定义和编程灰度路由的操作演示”

多IP和端口灰度路由架构图

REST调用的编程灰度路由策略

基于Feign/RestTemplate的REST调用的自定义路由，见示例演示的“用户自定义和编程灰度路由的操作演示”

RPC调用的编程灰度路由策略

基于Feign/RestTemplate的RPC调用的自定义路由，见示例演示的“用户自定义和编程灰度路由的操作演示”

规则和策略

规则和策略的区别

属性	规则	策略
方式	通过XML或者Json配置	通过REST或者RPC调用传递Header或者参数
频率	灰度发布期间更新，频率低	每次调用时候传递，频率高
扩展性	内置，有限扩展，继承三个AbstractXXXListener	内置，完全扩展，实现DiscoveryEnabledStrategy
作用域	运行前，运行期	运行期
依赖性	依赖配置中心或者本地配置文件	依赖每次调用

规则和策略的关系

- 规则和策略，可以混合在一起工作，也关闭一项，让另一项单独工作
- 规则和策略，一起工作的时候，先执行规则过滤逻辑，再执行策略过滤逻辑
- 规则和策略关闭
  - 规则关闭，spring.application.register.control.enabled=false和spring.application.discovery.control.enabled=false
  - 策略关闭，spring.application.strategy.control.enabled=false

外部元数据

外部系统（例如运维发布平台）在远程启动制定微服务的时候，可以通过参数传递来动态改变元数据或者增加运维特色的参数，最后注册到远程配置中心。有两种方式，如下图：

- 通过Program arguments来传递，它的用法是前面加“--”。支持Eureka、Zookeeper和Nacos（增量覆盖），Consul支持的不好（全量覆盖）
- 通过VM arguments来传递，它的用法是前面加“-D”。支持上述所有的注册组件，它的限制是变量前面必须要加“ext.”
- 两种方式尽量避免同时用

配置文件

- 基础属性配置 不同的服务注册发现组件对应的不同的配置值（region配置可选），请仔细阅读

```
# Eureka config for discovery
eureka.instance.metadataMap.group=xxx-service-group
eureka.instance.metadataMap.version=1.0
eureka.instance.metadataMap.region=dev

# 奇葩的Consul配置（参考https://springcloud.cc/spring-cloud-consul.html - 元数据和Consul标签）
# Consul config for discovery
spring.cloud.consul.discovery.tags=group=xxx-service-group,version=1.0,region=dev

# Zookeeper config for discovery
spring.cloud.zookeeper.discovery.metadata.group=xxx-service-group
spring.cloud.zookeeper.discovery.metadata.version=1.0
spring.cloud.zookeeper.discovery.metadata.region=dev

# Nacos config for discovery
spring.cloud.nacos.discovery.metadata.group=example-service-group
spring.cloud.nacos.discovery.metadata.version=1.0
spring.cloud.nacos.discovery.metadata.region=dev

# Admin config
# E版配置方式
# 关闭访问Rest接口时候的权限验证
management.security.enabled=false
management.port=5100

# F版或以上配置方式
management.server.port=5100
```

- 功能开关配置 请注意，如下很多配置项，如果使用者不想做特色化的处理，为避免繁琐，可以零配置（除了最底下，但一般也不会被用到）

```
# Plugin core config
# 开启和关闭服务注册层面的控制。一旦关闭，服务注册的黑/白名单过滤功能将失效，最大注册数的限制过滤功能将失效。缺失则默认为true
# spring.application.register.control.enabled=true
# 开启和关闭服务发现层面的控制。一旦关闭，服务多版本调用的控制功能将失效，动态屏蔽指定IP地址的服务实例被发现的功能将失效。缺失则默认为true
# spring.application.discovery.control.enabled=true
# 开启和关闭通过Rest方式对规则配置的控制和推送。一旦关闭，只能通过远程配置中心来控制和推送。缺失则默认为true
# spring.application.config.rest.control.enabled=true
# 规则文件的格式，支持xml和json。缺失则默认为xml
# spring.application.config.format=xml
# spring.application.config.format=json
# 本地规则文件的路径，支持两种方式：classpath:rule.xml（rule.json） - 规则文件放在resources目录下，便于打包进jar；
file:rule.xml（rule.json） - 规则文件放在工程根目录下，放置在外部便于修改。缺失则默认为不装载本地规则
# spring.application.config.path=classpath:rule.xml
# spring.application.config.path=classpath:rule.json
# 为微服务归类的Key，一般通过group字段来归类，例如eureka.instance.metadataMap.group=xxx-group或者
eureka.instance.metadataMap.application=xxx-application。缺失则默认为group
# spring.application.group.key=group
# spring.application.group.key=application
# 内置Rest调用路径的前缀，当配置了server.context-path或者server.servlet.context-path时候，需要同步配置下面的值，务必保持一致
# spring.application.context-path=${server.servlet.context-path}

# Plugin strategy config
# 开启和关闭策略扩展功能的控制。一旦关闭，用户自定义和编程灰度路由策略功能将失效。缺失则默认为true
# spring.application.strategy.control.enabled=true
# 开启和关闭Ribbon默认的ZoneAvoidanceRule负载均衡策略。一旦关闭，则使用RoundRobin简单轮询负载均衡策略。缺失则默认为true
# spring.application.strategy.zone.avoidance.rule.enabled=true
# 启动和关闭用户自定义和编程灰度路由策略的时候，对RPC方式的调用拦截。缺失则默认为false
spring.application.strategy.rpc.intercept.enabled=true
# 用户自定义和编程灰度路由策略的时候，对RPC方式调用拦截的时候，需要指定对业务Controller类的扫描路径，以便传递上下文对象。该项配置只对服务有效，对网关无效
spring.application.strategy.scan.packages=com.nepxion.discovery.plugin.example.service.feign
# 启动和关闭用户自定义和编程灰度路由策略的时候，对REST方式的调用拦截。缺失则默认为false
```

```
spring.application.strategy.rest.intercept.enabled=true
# 用户自定义和编程灰度路由策略的时候，对REST方式调用拦截的时候（支持Feign或者RestTemplate调用），需要把来自外部的指定Header参数传递到服务里，如果多个用“;”分隔，不允许出现空格。该项配置只对服务有效，对网关无效
spring.application.strategy.request.headers=token
# 启动和关闭用户自定义和编程灰度路由策略的时候日志打印，注意每调用一次都会打印一次，会对性能有所影响，建议压测环境和生产环境关闭。缺失则默认为false
spring.application.strategy.intercept.log.print=true
# 开启服务端实现Hystrix线程隔离模式做服务隔离时，必须把spring.application.strategy.hystrix.threadlocal.supported设置为true，同时要引入discovery-plugin-strategy-starter-hystrix包，否则线程切换时会发生ThreadLocal上下文对象丢失
# spring.application.strategy.hystrix.threadlocal.supported=true
```

监听扩展

使用者可以继承如下类

- AbstractRegisterListener，实现服务注册的监听，用法参考discovery-springcloud-example-service下MyRegisterListener
- AbstractDiscoveryListener，实现服务发现的监听，用法参考discovery-springcloud-example-service下MyDiscoveryListener。注意，在Consul下，同时会触发service和management两个实例的事件，需要区别判断，见上图“集成了健康检查的Consul界面”
- AbstractLoadBalanceListener，实现负载均衡的监听，用法参考discovery-springcloud-example-service下MyLoadBalanceListener

配置中心

- 默认集成
  - 本系统跟Apollo集成，如何安装使用，请参考https://github.com/ctripcorp/apollo
  - 本系统跟Nacos集成，如何安装使用，请参考https://github.com/alibaba/nacos
  - 本系统跟Redis集成
- 扩展集成
  - 使用者也可以跟更多远程配置中心集成
  - 参考三个跟Nacos或者Redis有关的工程

管理中心

:exclamation:PORT端口号为服务端口或者管理端口都可以

- 配置接口
- 版本接口
- 路由接口 参考Swagger界面，如下图

:exclamation:Swagger默认不支持多个Swagger包路径下的实现，如果业务系统有自己的Swagger功能，那么只需要在配置文件里面加上swagger.service.base.package={路径1},{路径2},{路径3}

控制平台

为UI提供相关接口，包括

- 一系列批量功能
- 跟Nacos、Apollo和Redis集成，实现配置拉取、推送和清除

:exclamation:PORT端口号为服务端口或者管理端口都可以

- 控制平台接口 参考Swagger界面，如下图

监控平台

基于Spring Boot Actuator技术的Spring Boot Admin监控平台

参考https://github.com/codecentric/spring-boot-admin

熔断隔离限流降级平台

基于Alibaba Sentinel的熔断隔离限流降级平台

参考https://github.com/alibaba/Sentinel

界面工具

请参考入门教程的“界面操作”

基于图形化桌面程序的灰度发布

- 见入门教程的“运行图形化灰度发布桌面程序”

基于图形化Web程序的灰度发布

- 见入门教程的“运行图形化灰度发布Web程序”

基于Apollo界面的灰度发布

- 见入门教程的“运行Apollo配置界面”

基于Nacos界面的灰度发布

- 见入门教程的“运行Nacos配置界面”

基于Rest方式的灰度发布

- 见入门教程的“运行Swagger或者Postman方式”

性能分析

在我的电脑上，做了如下性能测试：

应用	耗时	内存
空的Spring Cloud启动	9秒	400M
带有Discovery Plugin的Spring Cloud启动	13秒	480M

启动项	耗时
Spring Boot预热启动	2秒
Discovery Plugin预热启动	2秒
Spring上下文扫描加载	2秒
RestController和Swagger扫描加载	1秒
determine local hostname	1.5秒
连本地配置中心，并打印本地规则和远程规则	1.5秒
Actuator Endpoint扫描加载	1秒
连本地服务注册中心，并启动结束	2秒

Star走势图