Hzero+微服务学习手册

Author: 刘光锐

Creation Date: 四月 11, 20200207

Last Updated: 四月 11, 20200214

Document Ref: <Document Reference Number>

Version: 1.0

1. **Title, Subject, Last Updated Date, Reference Number**, **and** **Version** are marked by a Word Bookmark so that they can be easily reproduced in the header and footer of documents. When you change any of these values, be careful not to accidentally delete the bookmark. **You can make bookmarks visible by selecting Tools->Options…View and checking the Bookmarks option in the Show region.**

**Approvals:**

|  |  |
| --- | --- |
| <Approver 1> |  |
| <Approver 2> |  |

1. To add additional approval lines, press [Tab] from the last cell in the table above.

handlogo Copy Number \_\_\_\_\_

1. You can delete any elements of this cover page that you do not need for your document. For example, Copy Number is only required if this is a controlled document and you need to track each copy that you distribute.

## Document Control

**Change Record**

6

| Date | Author | Version | Change Reference |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Reviewers**

| Name | Position |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Distribution**

| Copy No. | Name | Location |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

1. The copy numbers referenced above should be written into the **Copy Number** space on the cover of each distributed copy. If the document is not controlled, you can delete this table, the Note To Holders, and the **Copy Number** label from the cover page.

**Note To Holders:**

If you receive an electronic copy of this document and print it out, please write your name on the equivalent of the cover page, for document control purposes.

If you receive a hard copy of this document, please write your name on the front cover, for document control purposes.

Contents

Document Control ii

1. 后端深入 2

1.1. 通用导入 2

1.2. 调度任务 6

1.3. 邮箱配置 7

1.4. Eureka学习 8

1.5. Eureka配置 9

1.6. Http请求和RestTemplate 12

1.7. Spring Cloud Ribbon客户端负载均衡 13

1.8. 服务容错保护Spring Cloud Hystrix 19

1.9. 声明试服务调用Spring Cloud Feign 23

1.10. Api网关服务Spring Cloud Zuul 24

Open and Closed Issues for this Deliverable 31

Open Issues 31

Closed Issues 31

1. To update the table of contents, put the cursor anywhere in the table and press [F9]. To change the number of levels displayed, select the menu option Insert‑>Index and Tables, make sure the Table of Contents tab is active, and change the Number of Levels to a new value.

## 1. 后端深入

### 通用导入

#### 1.1.1组件介绍

通用导入是客户端组件，通过添加依赖，配合系统配置的模板就可以使用。

配置系统模板

路径：通用导入>导入模板管理

第一步，新建导入模板sheet页

第二步，新建模板、列



新建导入列的时候可以设置字段编码，列名，类型，格式掩码，长度，最大最小值，值集，正则表达式，实例数据，是否为空等信息。



#### 1.1.2后端开发

通过添加通用导入依赖包，可以自动解析excel中的列，首先对模板配置的规则进行自动校验，之后校验客户端自定义校验，最后执行插表操作。最后两步是可以进行客户化开发的。没个导入模板每次导入都会生成一个唯一的流水号，以便对同一批数据进行校验和导入。

首先添加依赖包

|  |
| --- |
| <dependency>  <groupId>org.hzero.boot</groupId>  <artifactId>hzero-boot-import</artifactId>  <version>${hzero.boot.version}</version>  </dependency> |

配置参数

|  |
| --- |
| # 服务启动是否执行表初始化脚本，默认值false  hzero.import.initTable  # 是否开启数据导入到正式表的事务控制，默认值true  hzero.import.transactionControl  # 数据操作的批次量，默认值3000  hzero.import.batchSize |

编写校验逻辑。校验逻辑只能按照单挑数据编写，每次只校验一条数据，如果向上抛出异常或者使用getContext().addErrorMsg("这条数据有错"); 来抛出错误，最终会显示在界面上。

用 @ImportValidators 注解来表明这是一个校验器，用@ImportValidator 来表明对某个模板进行校验，这里可以添加多个模板，用“,”隔开。

每一条数据进入校验器的时候都是以json字符串的形式，所以在进行取值的时候可以转换为实体。

如果最后校验成功，返回true，否则抛出异常，或者返回false。

注：模板选择的时候可能会使用值集，解决办法有两种：1.使用**lovAdapter** 查询出值集，反向将 meaning 转换为 value 2.在模板配置界面打开值集转换

|  |
| --- |
| @ImportValidators({  @ImportValidator(templateCode = **"GOMT.CREDIT\_MANAGE"**) }) **public class** CreditImportValidator **extends** ValidatorHandler {   @Autowired  **private** ObjectMapper **objectMapper**;  @Autowired  **private** SalesRelationHeaderRepository **salesRelationHeader**;  @Autowired  **private** CreditManageRepository **creditManageRepository**;  @Autowired  **private** CustomerInfoRepository **customerInfoRepository**;  @Autowired  **private** LegalEntityInfoRepository **legalEntityInfoRepository**;  @Autowired  **private** LovAdapter **lovAdapter**;   **private** Logger **logger** = LoggerFactory.*getLogger*(CreditImportValidator.**class**);    @Override  **public boolean** validate(String data) {  *//获取值集* List<LovValueDTO> lovControl = **lovAdapter**.queryLovValue(Constants.LovCode.***CREDIT\_CONTRAL***, BaseConstants.***DEFAULT\_TENANT\_ID***);  **logger**.info(**"开始导入信用管理校验"**);  CreditManage creditManage = **null**;  **try** {  creditManage = JSONObject.*parseObject*(data, CreditManage.**class**);  *//此处设置默认值,默认为类型二，账期起算为10* creditManage.setCreditType(Constants.CreditType.***TYPE\_TWO***);  creditManage.setPaymentDaysStart(**"10"**);  *//管控方式* creditManage.setManageWay(queryLovValue(lovControl, creditManage.getManageWayMeaning()));  } **catch** (Exception e) {  **context**.addErrorMsg(**"转换参数失败："** + e.getMessage());  **return false**;  }  *//区分供应商类型和非供应商类型* Assert.*notNull*(creditManage.getCreditType(), **"信用类型不存在"**);  **switch** (creditManage.getCreditType()) {  **case** Constants.CreditType.***TYPE\_ONE***:  validateTypeOne(creditManage);  **break**;  **case** Constants.CreditType.***TYPE\_TWO***:  validateTypeTwo(creditManage);  **break**;  **default**:  **context**.addErrorMsg(**"类型不存在"**);  }  *//校验公共参数* validateCommon(creditManage);  **return true**;  }  } |

编写导入逻辑，用@ImportService 注解来标注导入类，templateCode =模板编码，导入类有两种，一种是单条数据导入，继承BatchImportHandler 类，另一种是批量导入（有异常会回滚），实现IBatchImportService。

注：若相同模板编码同时存在IDoImportService和IBatchImportService的不同实现，只会执行IBatchImportService的doImport方法。

|  |
| --- |
| @ImportService(templateCode = **"GOMT.CREDIT\_MANAGE"**) **public class** CreditImportServiceImpl **implements** IBatchImportService {   @Autowired  **private** ObjectMapper **objectMapper**;   @Autowired  **private** SalesRelationHeaderRepository **salesRelationHeader**;  @Autowired  **private** CreditManageRepository **creditManageRepository**;  @Autowired  **private** CustomerInfoRepository **customerInfoRepository**;  @Autowired  **private** LegalEntityInfoRepository **legalEntityInfoRepository**;   **private** Logger **logger** = LoggerFactory.*getLogger*(CreditImportServiceImpl.**class**);   @Autowired  **private** LovAdapter **lovAdapter**;   @Override  @Transactional(rollbackFor = Exception.**class**)  **public** Boolean doImport(List<String> data) {  **logger**.info(**"开始导入信用管理"**);  Set<String> saleRelation = **new** HashSet<>();  Map<String, String> unique = **new** HashMap<>();   List<LovValueDTO> lovContral = **lovAdapter**.queryLovValue(Constants.LovCode.***CREDIT\_CONTRAL***, BaseConstants.***DEFAULT\_TENANT\_ID***);data.forEach(item -> {  **try** {  CreditManage creditManage = **objectMapper**.readValue(item, CreditManage.**class**); *//信用管理类型，默认为类型二* creditManage.setCreditType(Constants.CreditType.***TYPE\_TWO***);creditManage.setPaymentDaysStart(**"10"**);  *//管控方式* creditManage.setManageWay(queryLovValue(lovContral, creditManage.getManageWayMeaning()));  **if** (creditManage.getEffectiveDate() != **null**) {  creditManage.setStartDate(Date.*from*(creditManage.getEffectiveDate().atStartOfDay(ZoneId.*systemDefault*()).toInstant()));  }  **if** (creditManage.getExpiryDate() != **null**){  creditManage.setEndDate(Date.*from*(creditManage.getExpiryDate().atStartOfDay(ZoneId.*systemDefault*()).toInstant()));  }  validateSaleRelation(saleRelation, creditManage.getSalesContractNum());  validateUnique(unique, creditManage.getCustomerName() + creditManage.getOrgName(), creditManage.getManageWay() + creditManage.getCreditQuota());  saveCreditManage(creditManage);  } **catch** (Exception e) {  **throw new** CommonException(**"导入失败："** + e);  }  });  **return true**;  }  } |

### 调度任务

#### 1.2.1界面配置

调度任务是基于Quartz 2.3.0 开发的，是对客户端服务的任务进行调度。多个调度任务可以共用一个执行器，执行器是以微服务的维度设置的，可以在执行器中设置负载，任务触发之后可以调用服务端接口，这个触发可以是手动触发的，也可以是按照一定周期触发的。以下是执行器配置界面。如果有多个服务节点，可以在机器地址中用逗号隔开。配置界面可以设置节点的权重。

调度任务创建界面：

#### 1.2.2后端开发

其中，JobHandler 的编码是在代码中需要标明的方法入口。代码中如下：

|  |
| --- |
| @JobHandler(**"supplementaryOrderJob"**) **public class** SupplementaryOrderJob **implements** IJobHandler {  @Override  **public** ReturnT execute(Map<String, String> map, SchedulerTool tool) {  }  } |

其中的map参数是在界面中配置的任务参数，可以帮助实现任务的可控性，SchedulerTool 中可以设置当前任务的执行进度和结果。

### 邮箱配置

#### 1.3.1 界面配置

Hzero中的邮箱配置在Message服务中，主要配置邮箱账号和邮件服务器，端口信息等。配置如下

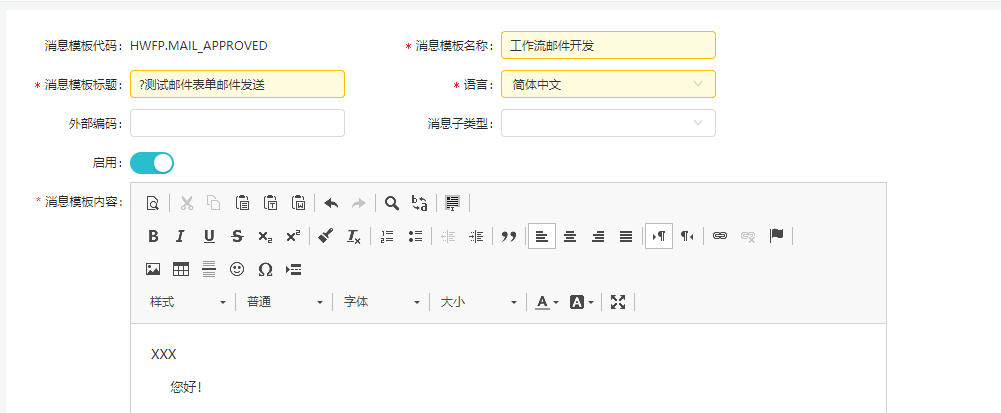
#### 1.3.2 代码写法

如果在程序中想发送邮件，可以使用Message的客户端。

首先添加Message客户端

|  |
| --- |
| <dependency>  <groupId>org.hzero.boot</groupId>  <artifactId>hzero-boot-message</artifactId>  <version>${hzero.boot.version}</version>  </dependency> |

配置消息模板



发送邮件代码：

|  |
| --- |
| @Service  **public class** MessageServiceImpl **implements** MessageService {  **private final** MessageClient **messageClient**;  @Autowired  **public** MessageServiceImpl(MessageClient messageClient) {   **this**.**messageClient** = messageClient;   }  @Override **public void** sendEmail()  {  **long** tenantId = 0L;  String serverCode = **"HZERO"**;  String messageTemplateCode = **"HWFP.REMIND"**;  Receiver receiver = **new** Receiver().setEmail(**"hzero@hand-china.com"**);  Map<String, String> args = **new** HashMap<>(16);  args.put(**"processName"**, **"测试消息"**);  args.put(**"processDescription"**, **"测试消息"**);} } |

### Eureka学习

以下内容参考网络博文，地址：https://zhuanlan.zhihu.com/p/84659866

1. **简介：**

Eureka 是 Netflix 出品的用于实现服务注册和发现的工具。 Spring Cloud 集成了 Eureka，并提供了开箱即用的支持。其中， Eureka 又可细分为 Eureka Server (服务端)和 Eureka Client(客户端)。​

1. **基本原理：**

服务启动后向Eureka注册，Eureka Server会将注册信息向其他Eureka Server进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。

当服务注册中心Eureka Server检测到服务提供者因为宕机、网络原因不可用时，则在服务注册中心将服务置为DOWN状态，并把当前服务提供者状态向订阅者发布，订阅过的服务消费者更新本地缓存。

服务提供者在启动后，周期性（默认30秒）向Eureka Server发送心跳，以证明当前服务是可用状态。Eureka Server在一定的时间（默认90秒）未收到客户端的心跳，则认为服务宕机，注销该实例。

1. **Eureka的自我保护机制**

在默认配置中，Eureka Server在默认90s没有得到客户端的心跳，则注销该实例，但是往往因为微服务跨进程调用，网络通信往往会面临着各种问题，比如微服务状态正常，但是因为网络分区故障时，Eureka Server注销服务实例则会让大部分微服务不可用，这很危险，因为服务明明没有问题。

为了解决这个问题，Eureka 有自我保护机制，通过在Eureka Server配置如下参数，可启动保护机制

|  |
| --- |
| eureka.server.enable-self-preservation=true |

它的原理是，当Eureka Server节点在短时间内丢失过多的客户端时（可能发送了网络故障），那么这个节点将进入自我保护模式，不再注销任何微服务，当网络故障恢复后，该节点会自动退出自我保护模式。

自我保护模式的架构哲学是宁可放过一个，决不可错杀一千。

1. **作为注册中心，Eureka比Zookeeper好在哪里**

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP**（疑点：如何保证CP?，参考博文：https://blog.51cto.com/welcomeweb/2103292?utm\_source=oschina-app）**, 而Eureka则是AP。

1. **Zookeeper保证CP**

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

1. **Eureka保证AP**

Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)

3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中。因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

### Eureka配置

1. **Eureka 服务端**

添加依赖

|  |
| --- |
| <dependency>                            <groupId>org.springframework.cloud</groupId>                            <artifactId>spring-cloud-starter-eureka-server</artifactId>                    </dependency>                    <dependency>                            <groupId>org.springframework.cloud</groupId>                            <artifactId>spring-cloud-starter-config</artifactId>                    </dependency>                    <dependency>                            <groupId> org.mybatis.spring.boot</groupId>                            <artifactId> mybatis-spring-boot-starter</artifactId>                    </dependency> |

配置文件

|  |
| --- |
| server:  port: 8008  eureka:  instance: # eureak实例定义  hostname: org.eurekaDemo.com # 定义 Eureka 实例所在的主机名称 |

修改host文件

|  |
| --- |
| 127.0.0.1 org.eurekaDemo.com |

创建启动类

|  |
| --- |
| Import org.springframework.boot.SpringApplication;  Import org.springframework.boot.autoconfigure.SpringBootApplication;  Import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  @SpringBootApplication  @EnableEurekaServer  publicclass EurekaDemoApplication {           public static void main(String[] args){                    SpringApplication.run(EurekaDemoApplication .class,args);           }  } |

1. **客户端配置**

添加依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency> |

修改 application.yml 配置文件，在这个配置文件之中主要是定义要进行注册的 Eureka 服务的地址，而这个地址就是 Eureka 的客户端配置。

|  |
| --- |
| eureka:  client: # 客户端进行Eureka注册的配置  service-url:  defaultZone: http://org.eurekaDemo.com:8008/eureka |

启动类启用服务发现

|  |
| --- |
| import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  @SpringBootApplication  @EnableEurekaClient  public class ClientDemoApplication {  public static void main(String[] args) {  SpringApplication.run(ClientDemoApplication .class,args);  }  } |

1. **设置多少秒没有收到心跳时间清理服务**

|  |
| --- |
| **eureka**:  **server**:  # 若40秒内eureka服务端没有收到某一个客户端的心跳，则将其剔除  **evictionIntervalTimerInMs**: 4000 |

1. **保护模式**

它默认支持有保护模式的概念，所谓的保护模式指的是即便现在某一个微服务不可用了，eureka 不会清理，依然会进行该微服务信息的保存。

|  |
| --- |
| **eureka**:  **server**:  **enable-self-preservation**: false |

1. **客户端心跳**

微服务的客户端每过一定时间需要给Eureka返回一个心跳信息，以代表该服务的注册状态是正常的，但是作为客户端，这个心跳间隔是可以进行设置的。

|  |
| --- |
| eureka:   client: # 客户端进行Eureka注册的配   service-url:   defaultZone:http://org.eurekaDemo.com:8008/eureka   instance:   lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）   lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）   instance-id: cloud-provider-client.com # 在信息列表时显示主机名称   prefer-ip-address: true # 访问的路径变为 IP 地址 |

### Http请求和RestTemplate

#### 1.6.1 Http理解

HTTP是基于客户/服务器模式,且面向连接的。客户与服务器之间的HTTP连接是一种一次性连接,它限制每次连接只处理一个请求,当服务器返回本次请求的应答后便立即关闭连接,下次请求再重新建立连接。典型的HTTP事务处理有如下的过程:

1. 客户与服务器建立连接;
2. 客户向服务器提出请求;
3. 服务器接受请求,并根据请求返回相应的文件作为应答;
4. 客户与服务器关闭连接。

|  |  |
| --- | --- |
| Http1.0 | 无状态、无连接 |
| HTTP1.1 | 持久连接  请求管道化  增加缓存处理（新的字段如cache-control）  增加Host字段、支持断点传输等（把文件分成几部分） |
| HTTP2.0 | 二进制分帧  多路复用（或连接共享）  头部压缩  服务器推送 |

1. Http1.0：

浏览器的每次请求都需要与服务器建立一个TCP连接，服务器处理完成后立即断开TCP连接（无连接），服务器不跟踪每个客户端也不记录过去的请求（无状态）。

1. Http1.1:

HTTP/1.0中默认使用Connection: close。在HTTP/1.1中已经默认使用Connection: keep-alive，避免了连接建立和释放的开销，但服务器必须按照客户端请求的先后顺序依次回送相应的结果，以保证客户端能够区分出每次请求的响应内容。通过Content-Length字段来判断当前请求的数据是否已经全部接收。不允许同时存在两个并行的响应。

1. Http2.0:

HTTP/2引入二进制数据帧和流的概念，其中帧对数据进行顺序标识，这样浏览器收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况。同样是因为有了序列，服务器就可以并行的传输数据，这就是流所做的事情。

#### 1.6.2 RestTemplate理解

**Rest:**

表述性状态转移,是一种架构风格。是面向资源的，这个概念非常重要，而资源是通过 URI 进行暴露。URI 的设计只要负责把资源通过合理方式暴露出来就可以了。对资源的操作与它无关，操作是通过 HTTP动词来体现，所以REST 通过 URI 暴露资源时，会强调不要在 URI 中出现动词。REST 是面向资源的，强调描述应用程序的事物和名词。

|  |  |
| --- | --- |
| 请求方法 | 含义 |
| GET | 申请获取资源，不会对服务器产生影响 |
| HEAD | 和Get类似，不过只获取头信息，不进行实际操作 |
| POST | 向服务器提交数据，这种操作可能会导致服务器新增数据或更新数据 |
| PUT | 上传某个资源 |
| DELETE | 删除某个资源 |
| TRACE | 请求目标服务器返回原始HTTP请求的内容，可以用来查看中间服务器对Http的影响 |
| OPTIONS | 查看服务器对某个特定URL都支持哪些请求方法 |
| CONNECT | 用于某些代理服务器，他们能把请求链接转换为一个安全隧道 |
| PATCH | 对某个资源做部分修改 |

### Spring Cloud Ribbon客户端负载均衡

#### 1.7.1 负载均衡

负载均衡建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。负载均衡其意思就是分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

#### 1.7.2 使用Nginx实现负载均衡和负载均衡策略

俄罗斯人开发的一个高性能的 HTTP和反向代理服务器，由于Nginx 超越 Apache 的高性能和稳定性。Nignx是服务端实现负载均衡，下面是几种负载均衡策略

|  |  |  |
| --- | --- | --- |
| 源地址哈希法 | 根据获取客户端的IP地址，通过哈希函数计算得到一个数值，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。 | weight |
| 轮询法 | 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。 | ip\_hash |
| 随机法 | 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。 | fair |
| 加权轮询法 | 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。 |  |
| 加权随机法 | 与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。 |  |
| 最小连接数法 | 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，最小连接数法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。 | least\_conn |

配置举例：

|  |
| --- |
| #user nobody;  worker\_processes 4;  events {  # 最大并发数  worker\_connections 1024;  }  http{  # 待选服务器列表  upstream myserver{  # ip\_hash指令，将同一用户引入同一服务器。  ip\_hash;  server 125.219.42.4 fail\_timeout=60s; #max\_fails失败后暂定时间60s  server 172.31.2.183;  }  server{  # 监听端口  listen 80;  # 根目录下  location / {  # 选择哪个服务器列表  proxy\_pass http://myserver;  }  }  }  max\_fails 允许请求失败的次数默认为1  fail\_timeout=60s fail\_timeout=60s失败暂停的时间  down 表示当前的服务器暂时不参与负载  backup 所有非backup机器忙的时候会请求backup，因此其压力会最轻 |

#### 1.7.3 使用Ribbon实现负载均衡

Spring Cloud Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用。Ribbon是客户端实现负载均衡的。即在每一个微服务中都适用。Ribbon 的负载均衡策略和Nginx类似，其中RandomRule表示随机策略、RoundRobinRule表示轮询策略、WeightedResponseTimeRule表示加权策略、BestAvailableRule表示请求数最少策略等等。顶级接口 IRule 中的choose用来按照均衡策略选择服务。下面分别是几种基本策略的核心代码：

**随机策略：**

|  |
| --- |
| public Server choose(ILoadBalancer lb, Object key) {  if (lb == null) {  return null;  }  Server server = null;  while (server == null) {  if (Thread.interrupted()) {  return null;  }  List<Server> upList = lb.getReachableServers();  List<Server> allList = lb.getAllServers();  int serverCount = allList.size();  if (serverCount == 0) {  return null;  }  //随机获取服务索引  int index = rand.nextInt(serverCount);  server = upList.get(index);  if (server == null) {  Thread.yield();  continue;  }  if (server.isAlive()) {  return (server);  }  server = null;  Thread.yield();  }  return server;  } |

**轮询策略：**

|  |
| --- |
| public Server choose(ILoadBalancer lb, Object key) {  if (lb == null) {  log.warn("no load balancer");  return null;  }  Server server = null;  int count = 0;  while (server == null && count++ < 10) {  List<Server> reachableServers = lb.getReachableServers();  List<Server> allServers = lb.getAllServers();  int upCount = reachableServers.size();  int serverCount = allServers.size();  if ((upCount == 0) || (serverCount == 0)) {  log.warn("No up servers available from load balancer: " + lb);  return null;  }  // 每次请求过后选择下一个服务 ex: 1,2,3,4  int nextServerIndex = incrementAndGetModulo(serverCount);  server = allServers.get(nextServerIndex);  if (server == null) {  Thread.yield();  continue;  }  if (server.isAlive() && (server.isReadyToServe())) {  return (server);  }  server = null;  }  if (count >= 10) {  log.warn("No available alive servers after 10 tries from load balancer: "  + lb);  }  return server;  } |

**加权策略:**

|  |
| --- |
| public Server choose(ILoadBalancer lb, Object key) {  if (lb == null) {  return null;  }  Server server = null;  while (server == null) {  List<Double> currentWeights = accumulatedWeights;  if (Thread.interrupted()) {  return null;  }  List<Server> allList = lb.getAllServers();  int serverCount = allList.size();  if (serverCount == 0) {  return null;  }  int serverIndex = 0;  double maxTotalWeight = currentWeights.size() == 0 ? 0 : currentWeights.get(currentWeights.size() - 1);  if (maxTotalWeight < 0.001d || serverCount != currentWeights.size()) {  server = super.choose(getLoadBalancer(), key);  if(server == null) {  return server;  }  } else {  double randomWeight = random.nextDouble() \* maxTotalWeight;  int n = 0;  //根据权重计算概率  for (Double d : currentWeights) {  if (d >= randomWeight) {  serverIndex = n;  break;  } else {  n++;  }  }  server = allList.get(serverIndex);  }  if (server == null) {  Thread.yield();  continue;  }  if (server.isAlive()) {  return (server);  }  server = null;  }  return server;  } |

**请求数最少策略：**

|  |
| --- |
| public Server choose(Object key) {  if (loadBalancerStats == null) {  return super.choose(key);  }  List<Server> serverList = getLoadBalancer().getAllServers();  int minimalConcurrentConnections = Integer.MAX\_VALUE;  long currentTime = System.currentTimeMillis();  Server chosen = null;  //循环找出最小链接的服务  for (Server server: serverList) {  ServerStats serverStats = loadBalancerStats.getSingleServerStat(server);  if (!serverStats.isCircuitBreakerTripped(currentTime)) {  int concurrentConnections = serverStats.getActiveRequestsCount(currentTime);  if (concurrentConnections < minimalConcurrentConnections) {  minimalConcurrentConnections = concurrentConnections;  chosen = server;  }  }  }  if (chosen == null) {  return super.choose(key);  } else {  return chosen;  }  } |

#### 1.7.4 Ribbon自动化配置

* IClientConfig：Ribbon的客户端配置，默认采用DefaultClientConfigImpl实现。
* IRule：默认采用ZoneAvoidanceRule实现，该策略能够在多区域环境下选出最佳区域的实例。
* IPing：默认采用DummyPing实现，该检查策略是一个特殊的实现，实际上并不会检查实例是否可用，而是始终返回true,默认认为所有实例都可用。
* ServerList<Server>:服务实例清单的维护机制，默认采用ConfigurationBasedServerList实现。
* ServerlistFilter<Server>:服务实例清单过滤机制，默认采用ZonePreferenceServerListFilter，该策略能够优先过滤出于请求方处于同区域的服务实例。
* ILoadBalancer:负载均衡器，默认采用ZoneAwareLoadBalancer，它具备区域感知的能力。

|  |
| --- |
| ribbon.ReadTimeout=1000 //处理请求的超时时间，默认为1秒  ribbon.ConnectTimeout=1000 //连接建立的超时时长，默认1秒  ribbon.MaxAutoRetries=1 //同一台实例的最大重试次数，但是不包括首次调用，默认为1次  ribbon.MaxAutoRetriesNextServer=0 //重试负载均衡其他实例的最大重试次数，不包括首次调用，默认为0次  ribbon.OkToRetryOnAllOperations=false //是否对所有操作都重试，默认false |

### 服务容错保护Spring Cloud Hystrix

参考博客：https://www.jianshu.com/p/f49b5628e8e7

#### 1.8.1 为什么需要熔断器？

在分布式中，我们会更具业务功能的不同将服务拆分成很多个，各个服务之间通过服务注册和订阅来实现相互依赖和调用功能。随着业务不断拓展，各个服务单元越来越多，相互之间依赖关系也越来越复杂，这时候，当某个服务单元出现问题或者网络因素导致调用延迟，此时如果请求不断增加，该调用服务就会出现故障，如果连锁反应越来越多，servlet的线程也会被消耗完毕，导致整个服务瘫痪。服务与服务的依赖会导致服务之间故障传播，从而迎来雪崩效应。为了解决这个点或者多个点的故障，熔断器就出现了。

#### 1.8.2 什么是熔断器？

熔断器就相当于电路中的保险丝，当检测到一段时间内服务出现很多类似的错误时，熔断器就会断开，对服务进行自我保护，使其在一段时间内将不再可用，防止某个服务不断请求导致类似错误，它会使服务继续执行而不用等待修正错误或者浪费cpu去等到长时间超时产生，从而进入回路方法。熔断器也可以使服务诊断服务是否已经可用，如果已经修正了，那么就会恢复服务使用。

#### 1.8.2 依赖隔离

资源隔离有线程池模式和信号量模式，hystrix默认使用线程池模式；ribbon发起对依赖服务的请求时，由hystrix的HystrixCommand代理实现，并初始化线程池由线程池创建新的线程来处理本次请求；之后的请求在所有核心线程工作饱和情况下先入线程池队列，否则线程池继续创建新的线程处理请求。这种方式要为每个依赖服务申请线程池，有一定的资源消耗，好处是可以应对突发流量（流量洪峰来临时，处理不完可将数据存储到线程池队列里慢慢处理）；信号量模式是每次对依赖服务的请求会创建新的线程处理，每创建新的线程信号量+1，请求返回后信号量-1，无法应对流量洪峰的发生。

* 资源隔离-信号量模式

在该模式下，接收请求和执行依赖服务在同一个线程内完成，不存在线程上下文切换所带来的性能开销，所以大部分场景应该选择信号量模式，但是在下面这种情况下，信号量模式并非是一个好的选择。比如一个接口中依赖了3个下游：serviceA、serviceB、serviceC，且这3个服务返回的数据互相不依赖，这种情况下如果针对A、B、C的熔断降级使用信号量模式，那么接口耗时就等于请求A、B、C服务耗时的总和，无疑这不是好的方案。

资源隔离-线程池模式

在该模式下，用户请求会被提交到各自的线程池中执行，把执行每个下游服务的线程分离，从而达到资源隔离的作用。当线程池来不及处理并且请求队列塞满时，新进来的请求将快速失败，可以避免依赖问题扩散。

在信号量模式提到的问题，对所依赖的多个下游服务，通过线程池的异步执行，可以有效的提高接口性能。

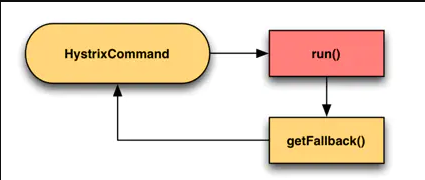
#### 1.8.3 Hzero中的依赖隔离

Hzero 默认的隔离策略是THREAD（线程级别）,线程级别的隔离导致在Feign拦截器中无法获取线程变量,解决方法：1.调整Hystrix隔离级别为信号量级别。2.自定义并发策略。源码如下：

|  |
| --- |
| private HystrixConcurrencyStrategy delegate;  public RequestAttributeHystrixConcurrencyStrategy() {  try {  this.delegate = HystrixPlugins.getInstance().getConcurrencyStrategy();  if (this.delegate instanceof RequestAttributeHystrixConcurrencyStrategy) {  // Welcome to singleton hell...  return;  }  HystrixCommandExecutionHook commandExecutionHook = HystrixPlugins.getInstance().getCommandExecutionHook();  HystrixEventNotifier eventNotifier = HystrixPlugins.getInstance().getEventNotifier();  HystrixMetricsPublisher metricsPublisher = HystrixPlugins.getInstance().getMetricsPublisher();  HystrixPropertiesStrategy propertiesStrategy = HystrixPlugins.getInstance().getPropertiesStrategy();  this.logCurrentStateOfHystrixPlugins(eventNotifier, metricsPublisher, propertiesStrategy);  HystrixPlugins.reset();  HystrixPlugins.getInstance().registerConcurrencyStrategy(this);  HystrixPlugins.getInstance().registerCommandExecutionHook(commandExecutionHook);  HystrixPlugins.getInstance().registerEventNotifier(eventNotifier);  HystrixPlugins.getInstance().registerMetricsPublisher(metricsPublisher);  HystrixPlugins.getInstance().registerPropertiesStrategy(propertiesStrategy);  } catch (Exception e) {  log.error("Failed to register Sleuth Hystrix Concurrency Strategy", e);  } } |

#### 1.8.4 服务降级

hystrix熔断器打开后，调用依赖服务时会尝试进行服务的降级处理，保证依赖服务的可用性。另外，线程池队列排满/信号量阈值情况下都会引起对依赖服务的服务降级处理，保证依赖服务的可用性。使用getFallback进行降级策略。



#### 1.8.5 请求实例

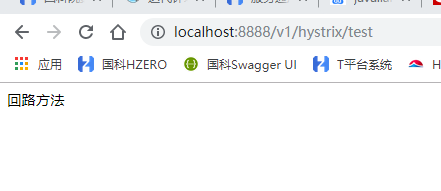
* 添加依赖

|  |
| --- |
| <!--熔断器-->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  <version>1.4.7.RELEASE</version>  </dependency> |

* 创建请求和回路方法

|  |
| --- |
| @GetMapping("/test")  @HystrixCommand(fallbackMethod = "fallBack")  public ResponseEntity<?> testMethod(){  System.out.println("开始调用");  try {  Thread.sleep(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("调用成功");  return ResponseEntity.ok("调用成功111");  }  //回路方法  public ResponseEntity<?> fallBack(){  System.out.println("回路方法");  return ResponseEntity.ok("回路方法");  } |

启动类中加注解“@EnableHystrix”

* 返回值：

这种情况是由于熔断器监测到方法返回时间过长，fallback到回路方法，而不是返回原来接口值。

#### 1.8.6 配置熔断器(分组，隔离，默认配置)

|  |
| --- |
| public class HystrixFallback4TimeoutTest extends HystrixCommand<String> {  private final String name;  public HystrixFallback4TimeoutTest(String name) {  super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HystrixFallback4TimeoutTest"))  .andCommandKey(HystrixCommandKey.Factory.asKey("testCommandKey"))// 按服务分组+命令粒度进行线程池隔离  .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("ThreadPoolTest"))// 线程池分组标识，不设置为默认分组  .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()  //设置为线程隔离  .withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrategy.THREAD)  //设置最大请求时间  .withFallbackIsolationSemaphoreMaxConcurrentRequests(100)  //打开短路器  .withExecutionIsolationThreadInterruptOnFutureCancel(true)  //发生超时是否中断  .withExecutionIsolationThreadInterruptOnTimeout(true)  .withFallbackEnabled(true)// 是否进行降级  .withExecutionTimeoutEnabled(true)  .withExecutionTimeoutInMilliseconds(500)));/\* 配置依赖超时时间,500毫秒\*/  this.name = name;  }  // 采用无限循环或sleep来模拟超时  @Override  protected String run() throws Exception {  // 超时异常，就算使用try-catch捕获也会触发fallback  try {  TimeUnit.MILLISECONDS.sleep(1200);  } catch(Exception e) { // note：就算此处被捕获也会触发fallback  e.printStackTrace();  }  System.out.println("after sleep");  return name;  }  @Override  protected String getFallback() {  return "fallback: " + name;  }  public static void main(String[] args) {  System.out.println(new HystrixFallback4TimeoutTest("Hlx").execute());  }  } |

### 声明试服务调用Spring Cloud Feign

#### 1.9.1 介绍

前面使用了Ribbon做客户端负载均衡，使用Hystrix做容错保护，这两者被作为基础工具类框架被广泛地应用在各个微服务的实现中。SpringCloudFeign是将两者做了更高层次的封装以简化开发。它基于Netfix Feign实现，整合了SpringCloudRibbon和SpringCloudHystrix，除了提供这两者的强大功能外，还提供了一种声明式的Web服务客户端定义的方式。SpringCloudFeign在NetFixFeign的基础上扩展了对SpringMVC注解的支持，在其实现下，我们只需创建一个接口并用注解的方式来配置它，即可完成对服务提供方的接口绑定。简化了SpringCloudRibbon自行封装服务调用客户端的开发量。

#### 1.9.2 配置及使用

1. **添加依赖**

|  |
| --- |
| <!-- eureka-->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency> <!-- feign-->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency>  <!-- spring- web-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> |

1. **启动类**

|  |
| --- |
| @SpringBootApplication  @EnableFeignClients  @EnableDiscoveryClient  public class KwrceTestApplication {  public static void main(String[] args) {  SpringApplication.run(KwrceTestApplication.class, args);  }  } |

1. **feign-consumer**

接口中使用注解表明其他服务接口地址，FeignClient 注解中注明服务名称、fallbck类，如果在调用成功的情况下并不会进入fallback实现类，直接返回调用结果，但是如果因为网络因素、熔断器熔断或其他因素导致调用失败，则会进入服务降级的方法中，做后续处理。

|  |
| --- |
| @FeignClient(value = "hzero-iam",fallback = IamClientFallback.class)  public interface IamRemoteService {  /\*\*  \* 查询用户信息  \*  \* @param organizationId 用户信息  \* @param userId 用户Id  \* @return JSONObject  \*/  @RequestMapping(method = RequestMethod.GET, path = "/hzero/v1/{organizationId}/users/{userId}/info")  JSONObject queryUser(@PathVariable("organizationId") Long organizationId, @PathVariable("userId") Long userId);  } |

1. **Fallback**

|  |
| --- |
| @Component  public class IamClientFallback implements IamRemoteService {  @Override  public JSONObject queryUser(Long organizationId, Long userId) {  throw new RuntimeException("查询用户信息失败，Iam服务异常");  }  } |

1. **配置文件**

|  |
| --- |
| spring:  application:  name: feign-consumer  server:  port: 88888  eureka:  client::  service-url:  # 注册中心地址  defaultZone: http://localhost:8888/eureka/ |

#### 1.9.3 Feign 启用 Hystrix 及 Ribbon 配置

|  |
| --- |
| feign:  hystrix:  enabled: true  #hystrix 配置和上面一致，此处省略  ribbon:  # 客户端读取超时时间，超时时间要小于Hystrix的超时时间，否则重试机制就无意义了  ReadTimeout: ${RIBBON\_READ\_TIMEOUT:30000}  # 客户端连接超时时间  ConnectTimeout: ${RIBBON\_CONNECT\_TIMEOUT:3000}  # 访问实例失败(超时)，允许自动重试，设置重试次数，失败后会更换实例访问，请一定确保接口的幂等性，否则重试可能导致数据异常。  OkToRetryOnAllOperations: true  MaxAutoRetries: 1  MaxAutoRetriesNextServer: 1 |

### Api网关服务Spring Cloud Zuul

参考博客：*https://blog.csdn.net/niugang0920/article/details/80548602*

#### 1.10.1 介绍

Zuul是Netflix开源的微服务网关，可以和Eureka、Ribbon、Hystrix等组件配合使用。 Zuul的主要功能是路由转发和过滤器。路由功能是微服务的一部分，比如／demo/test转发到到demo服务。zuul默认和Ribbon结合实现了负载均衡的功能。

从运维的角度来看，当客户端单机某个功能的时候往往会发出一些请求到后端，这些请求通过F5，Nginx等设施的路由和负载均衡分配后，被转发到各个不同的实例上，而为了让这些设施能够正确的路由与分发请求，运维人员需要手动维护这些实例列表，当系统规模增大的时候，这些看似简单的维护回变得越来越不可取。 从开发的角度来看，为了保证服务的安全性，我们需要在调用内部接口的时候，加一层过滤的功能，比如权限的校验，用户登陆状态的校验，服务的聚合等；同时为了防止客户端在请求时被篡改等安全方面的考虑，还会有一些签名机制的存在。

#### 1.10.2 传统路由

使用Spring Cloud Zuul实现路由功能非常简单，只需要对api-gateway服务增加一些关于路由的配置规则，就能实现传统路由方式。

|  |
| --- |
| zuul:  routes:  my-work-lgr:  path: /my-work-lgr/\*\*  url: http://localhost:8888/ |

该配置定义了发往API网关服务的请求中，所有符合/api-a-url/ 规则的访问都将被路由转发到 http://localhost:8888 的地址上，也就是说，当我们访问http://localhost:8080/my-work-lgr/hello 的时候，API网关服务会将该请求路由到http://localhost:8888/hello 提供的微服务接口中。其中，配置属性zuul.routes.api-a-url.path 中的api-a-url部分为路由的名字，可以任意定义，但是一组path和url映射关系的路由名要相同\*\*。

传统的路由方式是通过路径转发到其对应映射服务上去，这里是一对一的关系，映射路径是服务的实际地址，当服务增多或者使用多节点使就不能很好的完成配置。

1. 创建注册中心，配置如下：

|  |
| --- |
| spring:  application:  name: eureka-server  server:  port: 8100  eureka:  instance:  hostname: localhost  client:  serviceUrl:  ##注册地址  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/  register-with-eureka: false |

1. 在注册中心中创建测试接口

|  |
| --- |
| @GetMapping("/test")  public ResponseEntity<?> testMethod() {  System.out.println("开始调用");  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("调用成功");  return ResponseEntity.ok("调用成功111");  } |

1. 启动注册中心，调用接口*http://localhost:8100/v1/ask/test*



1. 创建网关服务

* 依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  </dependency> |

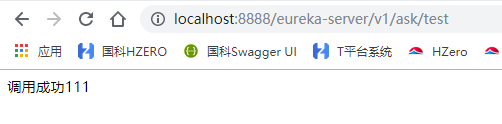
* 配置

|  |
| --- |
| spring:  application:  name: my-work-lgr  server:  port: 8888  eureka:  instance:  # 以IP注册到注册中心  preferIpAddress: ${EUREKA\_INSTANCE\_PREFER\_IP\_ADDRESS:true}  leaseRenewalIntervalInSeconds: 10  leaseExpirationDurationInSeconds: 30  # 服务的一些元数据信息  metadata-map:  VERSION: 1.0.0.RELEASE  NODE\_GROUP\_ID: ${HZERO\_NODE\_GROUP\_ID:0}  PRODUCT\_CODE: ${HZERO\_PRODUCT\_CODE:DEFAULT}  PRODUCT\_VERSION\_CODE: ${HZERO\_PRODUCT\_VERSION\_CODE:DEFAULT}  PRODUCT\_ENV\_CODE: ${HZERO\_PRODUCT\_ENV\_CODE:DEFAULT}  SERVICE\_VERSION\_CODE: ${HZERO\_SERVICE\_VERSION\_CODE:DEFAULT}  client:  serviceUrl:  # 注册中心地址  defaultZone: ${EUREKA\_DEFAULT\_ZONE:http://localhost:8100/eureka}  registryFetchIntervalSeconds: 10  disable-delta: true  zuul:  routes:  eureka-server:  path: /eureka-server/\*\*  url: http://localhost:8100/ |

启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableZuulProxy  public class KwrceTestApplication {  public static void main(String[] args) {  SpringApplication.run(KwrceTestApplication.class, args);  }  } |

1. 启动网关，访问网关代理路径：http://localhost:8888/eureka-server/v1/ask/test



至此，网关路由转发就完成了，http://localhost:8100/v1/ask/test 映射后的路径为http://localhost:8888/eureka-server/v1/ask/test

#### 1.10.3 服务路由及默认配置规则

使用传统路由对运维人员很不友好，需要大量的时间去维护路由的映射关系，需要转发的url少倒还好，多起来就会更容易配置出错。服务路由是通过指定服务名称，然后通过名称在Eureka中查找实例地址，最后转发到相关服务上去。需要注意的是服务路由需要指定注册中心地址。

|  |
| --- |
| zuul:  routes:  order-server:  path: /order-server/\*\*  service-id: order-server |

path的前缀使用了order-service,而对应的服务名称也是order-service。对于这种配置，zuul默认实现了这样的配置，当用Spring Cloud Zuul构建API网关并且整合Spring Cloud Eureka，它会为Eureka中的每个服务自动创建一个默认的路由规则，这些默认的规则的path会使用serviceId配置的服务名称作为请求前缀。

由于默认的规则会为在Eureka上注册的服务，都会被zuul自动创建映射关系来进行路由，这样会使一些不对外开放的服务也可能被访问到。zuul提供了zuul.ignored-services参数来设置一个服务名匹配表达式来定义不自动创建路由的规则。那么zuul会跳过该服务，不为其创建路由规则。

如：zuul.ignored-services=\*,Zuul将对所有的服务都不会自动创建路由规则.这样我们需要手动添加那个服务想创建映射关系。

#### 1.10.4 自定义路由规则

在实际开发中，可能会为每一个微服务起一个版本号为了迭代开发，如order-service-v1,userservice-v1,order-service-v2,userservice-v2,默认情况下，Zuul自动为服务创建的路由表达会采用服务名作为前缀，比如上面服务为产生的路由为order-service-v1/\*\*，userservice-v1/\*\*，这种不方便管理，我们可能希望产生这样的路由方式/v1/order-service/\*\*,/v1/userservice/\*\*。

|  |
| --- |
| @Bean  public PatternServiceRouteMapper serviceRouteMapper() {    //官方源码推荐的方式  return new PatternServiceRouteMapper("(?<name>^.+)-(?<version>v.+$)", "${version}/${name}");  } |

#### 1.10.5 Hystrix和ribbon支持

spring-cloud-starter-zuul依赖包括spring-cloud-starter-hystrix和spring-cloud-starter-ribbon 模块的依赖，所以zuul天生就拥有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡功能。

需要注意的事，当使用path与url的映射关系来配置路由规则的时候，对于路由转发的请求不会采用hystrixCommand来包装，所以这类请求没有线程隔离和断路器的保护，并且也不会有负载均衡的能力。因此，我们在使用zuul的时候尽量使用path和serviceId的组合来进行配置，这样不仅可以保证api网关的健壮和稳定，也能用到ribbon的客户端负载均衡功能。

我们在使用zuul搭建api网关的时候，可以通过hystrix和ribbon的参数来调整路由请求的各种超时时间等配置，比如下面这些参数的设置。

|  |
| --- |
| hystrix.command.default.execution.isolation.thread.timeoutInMillseconds:该参数可以用来设置api网关中路由转发请求hystrixCommand执行超时时间，单位为毫秒。当路由转发请求的命令执行时间超过该配置值之后，hystrix会将该执行命令标记为timeout并抛出异常，zuul会对该异常进行处理并返回如下的json信息给外部调用方。 |
| ribbon.ConnectTimeout:该参数用来设置路由转发请求的时候，创建请求连接的超时时间。当ribbon.ConnectTimeout的配置值小于hystrix.command.default.execttion.isolation.thread.timeoutInMilliseconds配置值的时候，若出现路由请求连接超时时，会自动进行重试路由请求，如果请求依然失败，zuul会返回如下json信息给外部调用方。 |
| ribbon.ConnectTimeout的配置值大于hystrix.command.default.execution.isolation.thread.timeoutInMillseconds配置值的时候，当出现路由请求连接超时时，由于此时对于路由转发的请求命令已经超时，所以不会进行重试路由请求，而是直接按请求命令超时处理，返回TIMEOUT的错误信息。 |
| ribbon.ReadTimeout:该参数用来设置路由转发请求的超时时间，它的处理与ribbon.ConnectTimeout类似，只是它的超时是对请求连接建立之后的处理时间。当ribbon.ReadTimeout的配置值小于hystrix.command.default.execttion.isolation.thread.timeoutInMilliseconds配置值的时候，若路由请求的处理时间超过该配置值并且依赖服务还没有响应的时候，会自动进行重试路由请求。如果重试后依然没有获得请求响应，zuul会返回NUMBEROF\_RETRIES\_NEXTSERVER\_EXCEEDED错误。如果ribbon.ReadTimeout的配置值大于hystrix.command.default.execttion.isolation.thread.timeoutInMilliseconds配置值，若路由请求的处理时间超过该配置值并且依赖服务还没有响应时，不会进行重试路由请求，而是直接按请求命令超时处理，返回timeout的错误信息。 |

#### 1.10.6 Zuul过滤器的生命周期

Zuul大部分功能都是通过过滤器来实现的，Zuul定义了4种标准的过滤器类型，这些过滤器类型对应于请求的典型生命周期。

a、pre: 这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在集群中选择请求的微服务，记录调试信息等。

b、routing: 这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用apache httpclient或netflix ribbon请求微服务。

c、post: 这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的http header、收集统计信息和指标、将响应从微服务发送给客户端等。

e、error: 在其他阶段发送错误时执行该过滤器。

#### 1.10.7 自定义过滤器

一下是DebugFilter的源码。

|  |
| --- |
| public class DebugFilter extends ZuulFilter {  private static final DynamicBooleanProperty ROUTING\_DEBUG = DynamicPropertyFactory  .getInstance().getBooleanProperty(ZuulConstants.ZUUL\_DEBUG\_REQUEST, false);  private static final DynamicStringProperty DEBUG\_PARAMETER = DynamicPropertyFactory  .getInstance().getStringProperty(ZuulConstants.ZUUL\_DEBUG\_PARAMETER, "debug");  //过滤器类型  @Override  public String filterType() {  return PRE\_TYPE;  }  //过滤器顺序  @Override  public int filterOrder() {  return DEBUG\_FILTER\_ORDER;  }  //返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关  @Override  public boolean shouldFilter() {  HttpServletRequest request = RequestContext.getCurrentContext().getRequest();  if ("true".equals(request.getParameter(DEBUG\_PARAMETER.get()))) {  return true;  }  return ROUTING\_DEBUG.get();  }  //过滤器的具体逻辑。在该函数中，我们可以实现自定义的过滤逻辑，来确定是否要拦截当前的请求，不对其进行后续的路由，或是在请求路由返回结果之后，对处理结果做一些加工等。  @Override  public Object run() {  RequestContext ctx = RequestContext.getCurrentContext();  ctx.setDebugRouting(true);  ctx.setDebugRequest(true);  return null;  }  } |

比如限流：

|  |
| --- |
| public class RateLimiterFilter extends AbstractPreZuulFilter {  private static final Logger LOGGER = LoggerFactory.getLogger(RateLimiterFilter.class);  /\*\*  \* 每秒允许处理的量是50  \*/  RateLimiter rateLimiter = RateLimiter.create(50);  @Override  public int filterOrder() {  return FilterOrder.RATE\_LIMITER\_ORDER;  }  @Override  public Object doRun() {  HttpServletRequest request = context.getRequest();  String url = request.getRequestURI();  if (rateLimiter.tryAcquire()) {  return success();  } else {  LOGGER.info("rate limit:{}", url);  return fail(401, String.format("rate limit:{}", url));  }  }  } |

#### 1.10.8 动态路由

动态路由需要配置可持久化且能动态刷新路由配置。DiscoveryClientRouteLocator具备刷新功能，但是需要已有的项目将服务注册到eureka。DiscoveryClientRouteLocator 中提供了刷新路由的接口 refresh()，当服务初始化时或者刷新路由的时候则会调用此接口。

### Spring Cloud Config分布式配置中心

#### 1.11.1 为什么要统一管理微服务配置？

随着微服务不断的增多，每个微服务都有自己对应的配置文件。在研发过程中有测试环境、UAT环境、生产环境，因此每个微服务又对应至少三个不同环境的配置文件。这么多的配置文件，如果需要修改某个公共服务的配置信息，如：缓存、数据库等，难免会产生混乱，这个时候就需要引入Spring Cloud另外一个组件：Spring Cloud Config。

#### 1.11.2 有哪几种配置中心？

1.SpringCloud config

Spring Cloud Config是一个解决分布式系统的配置管理方案。它包含了Client和Server两个部分，Server提供配置文件的存储、以接口的形式将配置文件的内容提供出去，Client通过接口获取数据、并依据此数据初始化自己的应用。 其实就是Server端将所有的配置文件服务化，需要配置文件的服务实例去Config Server获取对应的数据。将所有的配置文件统一整理，避免了配置文件碎片化。 如果服务运行期间改变配置文件，服务是不会得到最新的配置信息，需要解决这个问题就需要引入Refresh。可以在服务的运行期间重新加载配置文件。

2.nacos(阿里巴巴)

Nacos 支持基于 DNS 和基于 RPC 的服务发现（可以作为springcloud的注册中心）、动态配置服务（可以做配置中心）、动态 DNS 服务。Nacos 是构建以“服务”为中心的现代应用架构(例如微服务范式、云原生范式)的服务基础设施。

3.Apollo(阿波罗)

Apollo（阿波罗）是携程框架部门研发的配置管理平台，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端。apollo项目基于springboot与springcloud，可以独立部署，git地址：<https://github.com/ctripcorp/apollo>

#### 1.11.3 搭建配置中心，配置git仓库

1. 添加依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId> </dependency> |

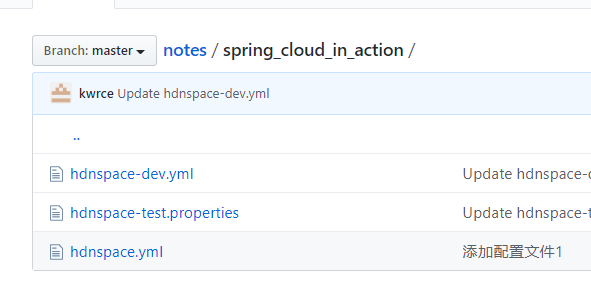
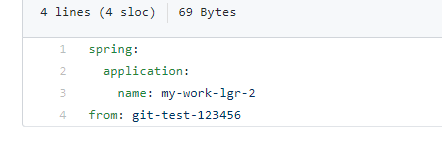
2. 启动类添加注解

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableConfigServer  public class KwrceTestApplication {  public static void main(String[] args) {  SpringApplication.run(KwrceTestApplication.class, args);  }  } |

3. 添加配置

|  |
| --- |
| spring:  application:  name: customer-config  cloud:  config:  server:  git:  uri: https://github.com/kwrce/notes  search-paths: spring\_cloud\_in\_action/  username: xxxx  password: xxxx server:  port: 8888  eureka:  instance:  # 以IP注册到注册中心  preferIpAddress: ${EUREKA\_INSTANCE\_PREFER\_IP\_ADDRESS:true}  leaseRenewalIntervalInSeconds: 10  leaseExpirationDurationInSeconds: 30  # 服务的一些元数据信息  metadata-map:  VERSION: 1.0.0.RELEASE  NODE\_GROUP\_ID: ${HZERO\_NODE\_GROUP\_ID:0}  PRODUCT\_CODE: ${HZERO\_PRODUCT\_CODE:DEFAULT}  PRODUCT\_VERSION\_CODE: ${HZERO\_PRODUCT\_VERSION\_CODE:DEFAULT}  PRODUCT\_ENV\_CODE: ${HZERO\_PRODUCT\_ENV\_CODE:DEFAULT}  SERVICE\_VERSION\_CODE: ${HZERO\_SERVICE\_VERSION\_CODE:DEFAULT}  client:  serviceUrl:  # 注册中心地址  defaultZone: ${EUREKA\_DEFAULT\_ZONE:http://localhost:8100/eureka}  registryFetchIntervalSeconds: 10  disable-delta: true |

4. 创建github仓库 notes，并且在目录spring\_cloud\_in\_action/ 下创建一下配置文件。

1. 在配置文件中添加配置信息
2. 启动配置服务，访问 <http://localhost:8888/hdnspace/dev/master>，结果如下：

#### 1.11.4 接口默认配置规则

配置中心配置了git仓库之后可以获取远程配置拉到本地，访问配置的时候默认按照以下规则访问。

|  |
| --- |
| / { 应用名 } / { 环境名 } [ / { 分支名 } ]  / { 应用名 } - { 环境名 }.yml  / { 应用名 } - { 环境名 }.properties  / { 分支名 } / { 应用名 } - { 环境名 }.yml  / { 分支名 } / { 应用名 } - { 环境名 }.properties |

#### 1.11.5 配置中心客户端配置

1. 添加依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency> |

2. 配置中心地址

|  |
| --- |
| spring:  application:  name: hdnspace  profiles:  active: default  cloud:  config:  profile: dev  label: master  uri: http://localhost:8888/  enabled: true  retry:  # 最大重试次数  maxAttempts: 6  multiplier: 1.1  # 重试间隔时间  maxInterval: 2000 |

3. 编写测试类

|  |
| --- |
| @Autowired  private Environment environment;  @RequestMapping("/from")  public String from(){  return environment.getProperty("from");  } |

1. 在浏览器访问 <http://localhost:8833/v1/hystrix/from>，便可以看到配置

#### 1.11.4 搭建配置中心，配置SVN仓库

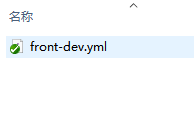
* springcloud 默认只提供了git版本实现，svn实现需要映入实现svnkit。

|  |
| --- |
| <dependency>  <groupId>org.tmatesoft.svnkit</groupId>  <artifactId>svnkit</artifactId>  <version>1.9.1</version>  </dependency> |

基础配置：

|  |
| --- |
| spring:  application:  name: front  profiles:  active: subversion  cloud:  config:  server:  svn:  uri: https://192.168.xxx.xx/svn/ERP/document/xx项目/99.其他/99-其他  username: xxxx  password: xxxx  default-label: lable  enabled: true  retry:  # 最大重试次数  maxAttempts: 6  multiplier: 1.1  # 重试间隔时间  maxInterval: 2000 |

* 在svn中添加文件 front-dev.yml

1587185905(1)

fetch-dev.properties #文件命名规则，{name}-{profle}.properties

|  |
| --- |
| "{[/{name}-{profiles}.properties],methods=[GET]}"  "{[/{name}-{profiles}.yml || /{name}-{profiles}.yaml],methods=[GET]}"  "{[/{name}/{profiles:.\*[^-].\*}],methods=[GET]}"  "{[/{label}/{name}-{profiles}.properties],methods=[GET]}"  "{[/{label}/{name}-{profiles}.json],methods=[GET]}"  "{[/{name}/{profiles}/{label:.\*}],methods=[GET]}"  "{[/{label}/{name}-{profiles}.yml || /{label}/{name}-{profiles}.yaml],methods=[GET]}"  "{[/{name}-{profiles}.json],methods=[GET]}"  "{[/{name}/{profile}/{label}/\*\*],methods=[GET],produces=[application/octet-stream]}"  "{[/{name}/{profile}/{label}/\*\*],methods=[GET]}"  "{[/{name}/{profile}/\*\*],methods=[GET],params=[useDefaultLabel]}" |

启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableConfigServer  public class KwrceTestApplication {  public static void main(String[] args) {  SpringApplication.run(KwrceTestApplication.class, args);  }} |

在浏览器访问 http://localhost:8833/front/dev/lable

#### 1.11.5 配置文件覆盖本地配置文件解决办法

在spirngcloud config 的需要不被复写的应用的配置文件（git/svn/mysql/本地）里加上下面两句话，这个应用的本地配置文件就不会被覆盖。

|  |
| --- |
| #不覆盖本地配置文件  spring.cloud.config.override-none=true  spring.cloud.config.override-system-properties=false |

#### 1.11.5 配置中心健康检查

在配置中心应用中，要实现可监控的功能，依赖的是spring-boot-stater-actuator这个组件;它提供了很多监控和管理的功能，并且用户可以有选择的开启和关闭监控功能；

|  |
| --- |
| **spring**:   **cloud**:  **config**:  **server**:  **health**:  **enabled**: true |

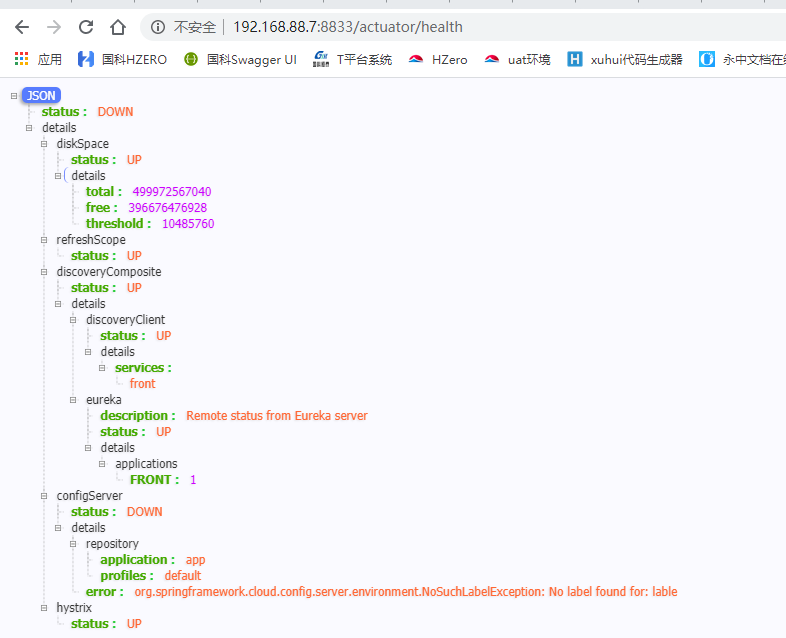
在pom文件中引入actuator组件;在实际的config-server配置中心服务中已经集成了actuator组件，不需要重复引用。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency> |

要查看更详细的信息需要配置management.endpoint.health.show-details的值为always。

|  |
| --- |
| management:  endpoint:  health:  show-details: always |

在浏览器中访问地址 [http://192.168.88.7:8833/actuator/health ，可以看到健康状态是DOWN,因为如果在所有的健康检查中如果有任何一项检查不通过，那么状态就会为DOWN。](http://192.168.88.7:8833/actuator/health，可以看到健康状态是DOWN,因为如果在所有的健康检查中如果有任何一项检查不通过，那么状态就会为DOWN。)



#### 1.11.6 失败快速响应

Spring Cloud Config的客户端会预先加载很多其他信息，然后再开始连接Config Server进行属性的注入。当我们构建的应用较为复杂的时候，可能在连接Config Server之前花费较长的启动时间，而在一些特殊场景下，我们又希望一可以快速知道当前应用是否能顺利地从Config Server获取到配置信息，这对在初期构建调试环境时，可以减少很多等待启动的时间。要实现客户端优先判断Config Server获取是否正常，并快速响应失败内容，只需要加入以下配置：

|  |
| --- |
| spring:  cloud:  config:  fail-fast: true |

#### 1.11.6 自动重试

在Config Server宕机或是客户端配置不正确导致连接不到而启动失败的情况，快速响应的配置可以发挥比较好的效果。但是，若只是因为网络波动等其他问歇性原因导致的问题，直接启动失败似乎代价有些高。所以，Config客户端还提供了自动重试的功能，在开启重试功能前，先确保己经配置了spring.cloud.config.fail-fast=true，再进行下面的操作。

|  |
| --- |
| spring:  application:  name: front  profiles:  active: subversion  cloud:  config:  server:  health:  enabled: true  default-label: lable  enabled: true  retry:  # 最大重试次数  maxAttempts: 6  multiplier: 1.1  # 重试间隔时间  maxInterval: 2000  fail-fast: true |

## Open and Closed Issues for this Deliverable

1. Add open issues that you identify while writing or reviewing this document to the open issues section. As you resolve issues, move them to the closed issues section and keep the issue ID the same. Include an explanation of the resolution.  
     
   When this deliverable is complete, any open issues should be transferred to the project- or process-level Risk and Issue Log (PJM.CR.040) and managed using a project level Risk and Issue Form (PJM.CR.040). In addition, the open items should remain in the open issues section of this deliverable, but flagged in the resolution column as being transferred.

### Open Issues

| ID | Issue | Resolution | Responsibility | Target Date | Impact Date |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### Closed Issues

| ID | Issue | Resolution | Responsibility | Target Date | Impact Date |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |