# 后台服务系统框架预研

## 1 系统整体架构

#### 1.1 模块功能划分

后台服务项目采用spring cloud框架，整个系统由云服务框架以及业务服务集群两部分组成，其中，云服务框架主要作用是提供整个系统层面的架构支撑，包括动态服务注册与发现（Eureka），反向代理（Nginx），链路追踪（Zipkin），日志管理（ELK），熔断器(Hystrix)，数据总线（RabbitMQ/Redis）等；业务服务集群主要作用是提供支撑业务的各个功能，包括业务服务程序（OMS、WMS…），服务间的互相调用（Feign），数据持久化（Mybatis）等。

后台服务系统对外仅提供webapi形式的接口，无论是前端还是第三方软件，与系统的对接均需要统一访问Nginx代理服务器，由代理服务器做进一步的验证及转发到Zuul网关。

根据系统多用户，多数据库的系统特点，在进行系统设计时，我们将系统进行了小集群的划分，一定量的用户（如200个）组成一个小集群，在集群内部，用户共享所有的服务程序，及数据库连接池。

#### 1.2 整体架构图



#### 请求路由过程描述

1） Nginx接收外部的url请求，取出其中的token，根据token判断出该请求发自哪个用户，并从redis缓存中获取用户对应的集群名及zuul网关列表。通过负载均衡算法，选取一个zuul网关，并将服务名（wms）扩展为“集群名-服务名”（gxxx-wms），组成新的url并调用；

2） Zuul接收到Nginx发来的url，取出“集群名-服务名”（gxxx-wms）信息，根据从eureka服务获取的服务列表，找到对应的地址列表，转化url为新的请求，并附带上user信息和内部接口调用时需要用到的session信息；

3） 业务服务接收到请求，验证session，执行相应的操作。

业务流 URL转换流程图

## 2 系统核心组件

#### 2.1 Eureka 服务注册发现中心

对于开发者来说，将服务程序设置为Eureka客户端很简单，只需要在我们的服务配置中加上Eureka客户端相关配置，即可将自己的服务注册到Eureka服务中去，同时也可以定时从Eureka获得所有已注册的服务列表。通过自动注册和获取服务，实现了动态发现所有可用服务的功能。

在业务服务的配置文件中，主要配置以下几项：

spring.application.name 服务名（集群名-服务名 如g001-wms，表示1号集群的wms服务），同一个集群内的同一类服务，服务名的一样的，以达到负载均衡的效果

eureka.client.service-url.defaultZone=http://ip:port/eureka eureka服务器的地址（整个系统唯一）

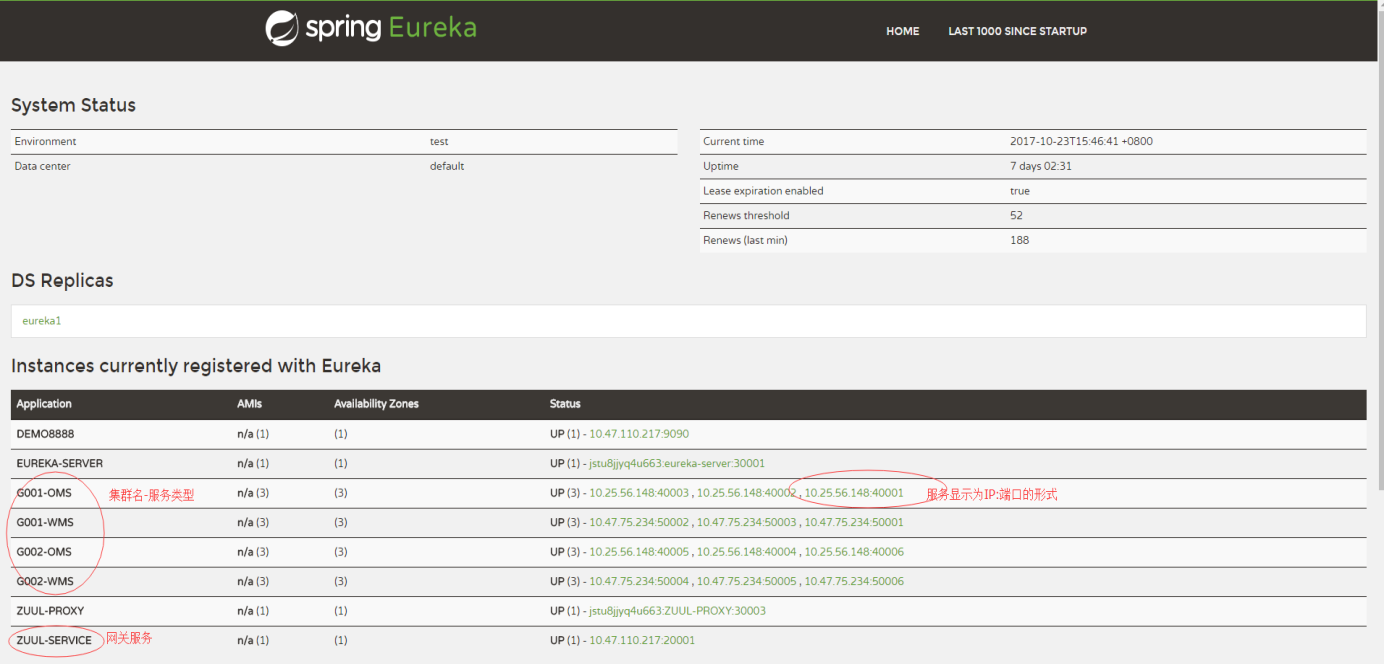
eureka.instance.lease-renewal-interval-in-seconds=10 心跳间隔

eureka.instance.lease-expiration-duration-in-seconds=30 心跳超时时间

eureka.instance.prefer-ip-address=true 显示ip地址（zuul、feign使用时也用这个，不支持通过计算机名访问时，这里必须填true，以配置ip和端口）

eureka.instance.instance-id 默认${spring.cloud.client.ipAddress}:${server.port}，即服务的ip和端口

eureka.client.registry-fetch-interval-seconds 客户端从eureka上拉去服务列表的时间间隔



Eureka监控界面

#### 2.2 Nginx 转发服务

Nginx服务的作用是，接收外部系统请求，并根据请求中携带的验证参数（token），判断该请求的合法性，再根据token对应的用户，找到用户对应的zuul网关集群，使用负载均衡算法，将请求转发到该zuul网关集群中的某一个zuul网关中去。网关集群及其对应的用户关系，存储于nginx服务的redis缓存中。

#### 2.3 Zuul 网关

Zuul网关在本系统内的作用，是接收nginx服务转发的请求，并根据请求参数，做安全验证，将通过安全验证的请求，转发到对应的业务服务器上去；将未通过安全验证的服务，直接拦截并回复。

Zuul网关在系统内，是以网关集群的形式出现的，一个网关集群负责多个业务服务集群的验证及转发工作。Zuul网关不需要判断请求的用户是否处于自己所管辖的集群范围内（这个工作已经由Nginx完成），只需要根据通过Nginx转发来的url，得到目标业务集群的服务名，并根据从Eureka中获取的服务列表，得到的对应业务服务集群的地址列表，然后通过Ribbon负载均衡，将命令交由其中一个业务服务程序执行即可。

Zuul网关可自定义Filter，对http调用的各个阶段进行验证，如使用PreRequestVerifyFilter，对进入网关的数据进行安全性验证等。

#### 2.4 Group 业务服务集群

为了使得业务服务能根据不同的负载情况，动态的改变负载能力，需要有能够支持动态增减的一组服务来共同承担一系列的功能，这块功能的核心就是Eureka服务发现，以及业务服务器的数据无关性设计。业务服务器对于其承担的每一个功能，都只负责逻辑部分的操作，命令数据、操作结果，包括中间状态的存储，都由数据库及缓存来承担。这样，一个业务服务集群中的每个业务服务程序，对于调用者来说，都是相同的，调用者（Zuul或Feign）使用负载均衡算法调用服务。

业务服务集群中的每个业务服务程序，都需要在配置文件中定义自己的服务类型，以及服务所处的集群名称，这两个信息将被用于注册到Eureka服务中，方便其他服务的发现和调用。

spring.application.name=${spring.application.group}-oms 拼装后的完整服务名，“服务集群-服务类型”

spring.application.group=gxxx 服务集群名，部署服务时需要配置

#### 2.5 Service 业务服务程序

业务服务程序，是后台系统中真正与业务模块相关的部分，根据功能将服务划分为不同业务服务，如仓储服务（WMS），订单服务（OMS）等，每个服务都有对应的数据库，从业务上、数据上，对不同业务进行拆分解耦。服务间的调用，分为同步调用和异步调用两种情况：同步调用适用于逻辑关系紧密，并且执行时间较短的业务，同步调用采用Feign客户端来实现；异步调用适用于可并发执行，或者执行时间较长的业务，异步调用采用RabbitMQ及Redis来实现。

#### 2.6 Service-Service(Feign) 业务服务程序间的调用

Feign的作用是简化服务之间远程接口调用的复杂度。在本系统中，我们利用Feign来简化业务服务集群之间的调用，并利用Feign、Eureka、Ribbon的结合，实现服务集群调用的负载均衡。

Feign的实现逻辑：首先，使用Feign客户端中自带的Ribbon客户端，收集从Eureka服务器上得到的服务列表（所以业务服务程序需要既是Eureka客户端，也是Feign客户端），然后根据要调用的接口上声明的目标服务名，在服务列表中找到服务地址集合，使用Ribbon负载均衡算法，获得其中一个服务地址，然后通过rest风格的远程调用，实现不同服务集群间相互调用的功能。

在后台系统中，我们将服务集群间的接口封装成了interface，调用方继承并定义指定服务名的声明，被调用方则继承并实现接口。

Feign的安全验证功能：在发送端，采取重写FeignBuilder的方法，在Feign组织http报文的时候，在报文头中插入session等验证字段；在接收端，添加拦截器，拦截Feign调用，并取出http头中相关验证字段进行安全验证，通过才允许继续操作。

#### 2.7 Sleuth + Zipkin 链路追踪

Zipkin链路追踪服务主要的作用是，利用sleuth客户端，将ID注入到webapi调用的报文中，并根据这个ID，收集分布式系统中各个环节的调用日志，通过restful接口（同步，默认）、RabbitMQ或Redis缓存（异步）的方式，发送给Zipkin服务，由Zipkin进行统一整理和展示。

通过Zipkin的监控界面，我们能很容易的统计内外部接口调用的整个链路情况，及各个环节的执行结果，耗时统计等。

在后台系统中，我们需要在Zuul网关和业务服务程序的配置中加上Zipkin客户端相关配置。关键配置如下：

spring.zipkin.base-url=http://10.25.56.148:30002/ zipkin服务的地址，这里的采用同步restful调用发送数据，如果采用异步方式，则需要配置rabbitmq或redis的连接信息

spring.sleuth.sampler.percentage=1.0 采样频率，1.0表示100%采样，在系统数据量大，调用频繁的情况下，降低采样频率，能够减少由于zipkin带来的系统压力。

#### 2.8 Hystrix 熔断器

Hystrix 熔断器的作用是，在系统某些接口和环节出现并发数过大，或是调用产生阻塞，或是抛出异常后，能够使用预定的处理逻辑对异常进行处理：如果并发过大，则将多余的调用直接熔断返回；如果调用阻塞，则将超时的调用熔断返回；如果抛出异常，则用之前定义的降级函数进行处理。

Springboot中熔断器在客户端的使用非常方便，使用由javanica提供的一个组件，只需要在想要引入熔断器功能的接口前面加入@HystrixCommand注释即可。

Hystrix的降级函数：

使用Hystrix注解@HystrixCommand(fallbackMethod="testApiHystrixFallback")，在注解中添加fallbackMethod，定义降级函数，当调用接口出现异常时，会自动调用降级函数进行降级处理，比如数据库操作异常，可以降级到从缓存中读取数据等。使用ignoreExceptions属性，可以定义不进行降级处理的异常类型，默认忽略HystrixBadRequestException。

Hystrix的线程隔离和Group分组：

同处于一个服务程序内的不同接口，正常情况下是共享同一个线程池的，当其中某个接口出现调用堵塞等情况，导致大量调用堆积，就会有可能出现耗尽线程池，导致整个服务不可用的后果。Hystrix提供了线程池分组技术，可以将不同的接口分成若干线程池组，当某个接口出问题时，只会影响该接口的线程池，服务中其他线程池相关接口还可以正常调用，这种方式提高了服务的可用性和稳定性。使用该功能举例，在方法前的注释里加上属性@HystrixCommand(groupKey="Key1", threadPoolKey="threadPoolKey1")。

熔断器默认配置的调用超时时间是1000ms，但是由于spring boot的懒加载机制，第一次调用接口的时候需要进行很多对象的生成初始化，所以往往第一次调用会抛出熔断的错误，通常需要在配置文件中添加修改熔断器的默认超时配置：

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000

#### 2.9 Turbine 接口调用状态监控平台

Hystrix内置dashboard，对服务内的依赖关键指标提供实时监控，包括单位时间内的调用次数，调用总数，成功、出错、降级数及百分比，以及断路器开关状态，调用延时时间指标等。但是dashboard只支持查看单个服务的运行情况，如果要查看服务集群中某个或多个接口的实时调用指标，就需要用到Hystrix的插件Turbine。

Turbine客户端配置：被注入Hystrix的服务程序，要使用Turbine，需要在配置文件中加上在Eureka注册metadata-map的配置，eureka.instance.metadata-map.cluster=SERVERNAME,

这个SERVERNAME将会被用于Turbine在eureka上寻找需要监控的服务。

Turbine服务端配置：

服务端引入

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-netflix-turbine</artifactId>

</dependency>

并在Application前加上注解@EnableHystrixDashboard @EnableTurbine。

配置文件中添加核心配置：

turbine.aggregator.clusterConfig=g001-WMS,g001-OMS metadata对应的SERVERNAME

turbine.appConfig=g001-wms,g001-oms 监控的服务在eureka上注册的服务名

turbine.clusterNameExpression=metadata['cluster'] 匹配表达式

使用时，先打开dashboard的监控中心<http://localhost:port/hystrix>，然后在里面填入需要监控的集群服务url：http://turbine-hostname:port/turbine.stream?cluster=[clusterName]

#### 2.10 CloudConfig 分布式配置中心

Spring Cloud Config为服务端和客户端提供了分布式系统的外部化配置支持。配置服务器为各应用的所有环境提供了一个中心化的外部配置。它实现了对服务端和客户端对Spring Environment和PropertySource抽象的映射，所以它除了适用于Spring构建的应用程序，也可以在任何其他语言运行的应用程序中使用。作为一个应用可以通过部署管道来进行测试或者投入生产，我们可以分别为这些环境创建配置，并且在需要迁移环境的时候获取对应环境的配置来运行。

在构建分布式配置中心时，我们将配置中心作为一个集群服务注册到了eureka上，访问配置中心可以通过配置服务的服务名来进行。

客户端使用配置：

客户端要使用配置服务，需要2点操作，首先，在pom文件中引入

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

然后，在项目的resources中加上bootstrap.properties配置文件，修改spring.cloud.config.name选项为cloud-服务类型。（bootstrap.properties已写好，共用一份）

配置中心服务端配置：

服务端pom引入jar包

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

在Application前加上@EnableConfigServer @EnableDiscoveryClient注解，启用配置服务和注册客户端。

在配置文件中加上配置文件存放的git的相关信息

spring.cloud.config.label=master

spring.cloud.config.server.git.username=username // git登录用户名

spring.cloud.config.server.git.password=password // git登录密码

spring.cloud.config.server.git.uri=git url // git项目url

spring.cloud.config.server.git.searchPaths=filepath // git项目文件夹名

启动时，需要先启动配置服务，后启动客户端，否则客户端无法读取到git上的配置文件。