SpringCloud

programmer-DD

1. Spring Cloud Eureka

Spring Cloud Eureka是Spring Cloud Netflix项目下的服务治理模块

Eureka Server的自我保护模式

解决如下警告：

> EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE L ESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

2. Spring Cloud Ribbon

负载均衡工具

3. Spring Cloud Feign

服务调用客户端

Feign文件传输

- 需要添加插件支持

```
<dependency>
   <groupId>io.github.openfeign.form</groupId>
   <artifactId>feign-form</artifactId>
   <version>3.0.3</version>
</dependency>
<dependency>
   <groupId>io.github.openfeign.form</groupId>
   <artifactId>feign-form-spring</artifactId>
   <version>3.0.3</version>
</dependency>
<dependency>
   <groupId>commons-fileupload</groupId>
   <artifactId>commons-fileupload</artifactId>
   <version>1.3.3</version>
</dependency>
```

- 并且进行代码配置

```
@Configuration
class MultipartSupportConfig {
  @Bean
  public Encoder feignFormEncoder() {
    return new SpringFormEncoder();
  }
}
```

- 最后调用文件上传服务

```
@PostMapping(value = "/uploadFile", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
String handleFileUpload(@RequestPart(value = "file") MultipartFile file);
```

使用中问题及解决

1. 明明是get方法，却总是报错“Request method 'POST' not supported”。

```
@RequestMapping(value="/user/name", method=RequestMethod.GET)
User findByUsername(final String userName, @RequestParam("address") final String address);
```

> 原因是当userName没有被@RequestParam注解修饰时，会自动被当做request body来处理。只要有body，就会被feign认为是post请求，所以整个服务是被当作带有request parameter和body的post请求发送出去的

Feign的扩展

> 由于Feign是基于Ribbon实现的，所以它自带了客户端负载均衡功能，也可以通过Ribbon的IRule进行策略扩展。另外，Feign还整合的Hystrix来实现服务的容错保护，在Dalston版本中，Feign的Hystrix默认是关闭的。

4. spring-cloud-config

可以研究一下git的权限管理，考虑spring-cloud-config是否能够实现apollo那样细致的权限管理

加密/解密

- 需要注意：`不限长度的JCE版本`

有效访问地址

- `/encrypt/status`：查看加密功能状态的端点
- `/key`：查看密钥的端点
- `/encrypt`：对请求的body内容进行加密的端点，POST请求
- `/decrypt`：对请求的body内容进行解密的端点，POST请求

对称加密

- 步骤（以`spring-cloud-config`为基础）
  1. 安装不限长度的JCE版本
  2. 在项目配置文件中添加配置：`encrypt.key=password`
     - 也可以通过环境变量设置此配置：`ENCRYPT_KEY`

非对称加密

- 步骤（以`spring-cloud-config`为基础）
  1. `keytool`生成密钥对文件`config-server.keystore`
  2. 配置boot项目

```
encrypt.key-store.location=file://${user.home}/config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=111111
encrypt.key-store.secret=222222
```

不过这些配置也可以通过环境变量来设置：

```
ENCRYPT_KEY_STORE_LOCATION
ENCRYPT_KEY_STORE_ALIAS
ENCRYPT_KEY_STORE_PASSWORD
```

```
ENCRYPT_KEY_STORE_SECRET
```

加密/解密使用

- 无论是对称还是非对称加密，都是在加密后的字符串前加上`{cipher}`标识，`spring-cloud-config`会自动解密

```
spring.datasource.password={cipher}3b6e65af8c10d2766dba099a590496a18cfd816ef9190c983bb56249595ae3f0
spring.datasource.password=
{cipher}AQCActlsAycDFYRsGHzZ8Jw2S6GO9oeqJSCcm//HenrqiuO7zSo3/vg9BeXL8xwiyIXtKcp2JN8hnrM4NTyyJDIjxhcCbJMju
GrrFJ2FdO5oJWmksymkP5EOXE6MjgxVqHh/tc+06TMBQj2xqEcfCO3jBDPxcR88Ci+VXe63xDIVgvAV9IYmCxlfXOCH31bBlK7j5FXJ8p
PLUKgXwaDGzaA5QfqMCGduOfC0AQ+iA0QEW7SdDnwChLNwCHEBfQceWAE7qt6zasiRFZeZt+waOp8rI1u+4CYcTjnV1iSdXwN5j1lhcso
iIpViNx8kbsxhcmpCzdg3bGrS1e/Pzq8CjHmV7IRRS9BfgR6K7wuyjue4SO2ZUtMbZAE5V2NHb3XsqeY=
```

- 注意：`yml`文件是不能使用加密/解密的

5. Spring Cloud Bus

可实现更改配置后自动应用到应用

6. Spring Cloud Hystrix

实现了线程隔离、断路器

注解

- `@EnableCircuitBreaker`
- `@HystrixCommand(fallbackMethod = "fallback")`
- 此处可以了解一下注解：`@SpringCloudApplication`

Hystrix-服务降级

Hystrix-依赖隔离

原理

> “舱壁模式”对于熟悉Docker的读者一定不陌生，Docker通过“舱壁模式”实现进程的隔离，使得容器与容器之间不会互相影响。而Hystrix则使用该模式实现线程池的隔离，它会为每一个Hystrix命令创建一个独立的线程池，这样就算某个在Hystrix命令包装下的依赖服务出现延迟过高的情况，也只是对该依赖服务的调用产生影响，而不会拖慢其他的服务。

> 执行依赖代码的线程与请求线程(比如Tomcat线程)分离，请求线程可以自由控制离开的时间，这也是我们通常说的异步编程，Hystrix是结合`RxJava`来实现的异步编程。通过设置线程池大小来控制并发访问量，当线程饱和的时候可以拒绝服务，防止依赖问题扩散。

博客中也对**打消Hystrix线程池隔离技术对性能影响的顾虑**进行了解释；也有关于`信号量`的介绍；

- 线程池方式下`业务请求线程`和`执行依赖服务的线程`不是同一个线程
- 信号量方式下`业务请求线程`和`执行依赖服务的线程`是同一个线程

> 如果通过信号量来控制系统负载，将不再允许设置超时和异步化，这就表示在服务提供者出现高延迟，其调用线程将会被阻塞，直至服务提供者的网络请求超时，如果对服务提供者有足够的信心，可以通过信号量来控制系统的负载。

打开`信号量模式`的方式：将属性execution.isolation.strategy设置为SEMAPHORE

线程隔离的缺点：

- 线程池的主要缺点就是它增加了计算的开销
  - 对于不依赖网络访问的服务，比如只依赖内存缓存这种情况下，就不适合用线程池隔离技术

Hystrix断路器

启动熔断器的三个重要参数：

- **快照时间窗**：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，`默认为最近的10秒`。
- **请求总数下限**：在快照时间窗内，必须满足请求总数下限才有资格根据熔断。`默认为20`，意味着在10秒内，如果该hystrix命令的调用此时不足20次，即时所有的请求都超时或其他原因失败，断路器都不会打开。
- **错误百分比下限**：当请求总数在快照时间窗内超过了下限，比如发生了30次调用，如果在这30次调用中，有16次发生了超时异常，也就是超过50%的错误百分比，在`默认设定50%`下限情况下，这时候就会将断路器打开。

那么当断路器打开之后会发生什么呢？

> 我们先来说说断路器未打开之前，对于之前那个示例的情况就是每个请求都会在当hystrix超时之后返回`fallback`，每个请求时间延迟就是近似hystrix的超时时间，如果设置为5秒，那么每个请求就都要延迟5秒才会返回。当熔断器在10秒内发现请求总数超过20，并且错误百分比超过50%（*这些请求情况的指标信息都是HystrixCommand和HystrixObservableCommand实例在执行过程中记录的*），这个时候熔断器打开。打开之后，再有请求调用的时候，将不会调用主逻辑，而是直接调用降级逻辑，这个时候就不会等待5秒之后才返回fallback。通过断路器，实现了自动地发现错误并将降级逻辑切换为主逻辑，减少响应延迟的效果。

断路后，主逻辑要如何恢复？

> 对于这一问题，hystrix也为我们实现了自动恢复功能。当断路器打开，对主逻辑进行熔断之后，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的成为主逻辑，当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的主逻辑上，如果此次请求正常返回，那么断路器将继续闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时

Hystrix Dashboard

7. 服务网关zuul

zuul除了路由功能外，还有负载均衡和权限控制

- 通过服务网关实现权限控制的目的是：将权限控制这种非业务功能剥离出来

网关路由功能

- 通过`http://zuul-host:zuul-port/服务ID/controller方法`形式进行路由

路由配置

服务网关(过滤器)

服务接口文档汇总

深入阅读

- Spring Cloud源码分析（四）Zuul：核心过滤器
- Spring Cloud实战小贴士：Zuul处理Cookie和重定向
- Spring Cloud实战小贴士：Zuul统一异常处理（一）
- Spring Cloud实战小贴士：Zuul统一异常处理（二）
- Spring Cloud实战小贴士：Zuul统一异常处理（三）【Dalston版】