24 Spring Cloud Config 服务化、动态刷新、重试

更新时间：2019-07-12 15:35:52

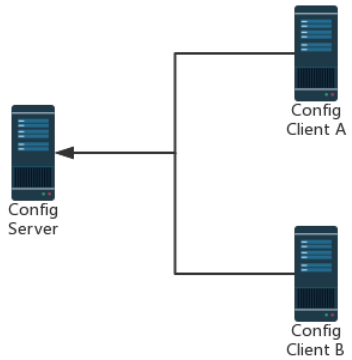
> 你若要喜爱你自己的价值，你就得给世界创造价值。
>
> ——歌德

上篇文章和大家分享了 Spring Cloud Config 中的配置文件安全问题，一方面是配置文件本身要加密，另一方面是 Spring Cloud Config 这个服务要有相应的安全机制，做好这两点，我们就不必担心数据安全问题了。

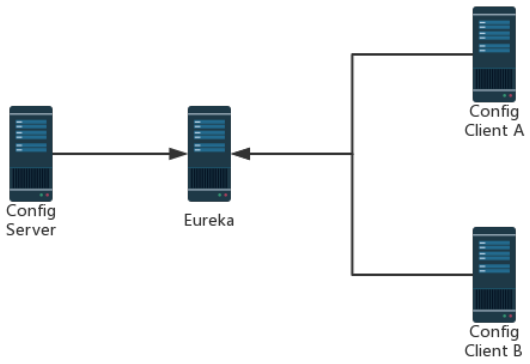
到目前为止我们的 Spring Cloud Config 中的所有案例都还是单服务的，没有服务化。另外，当配置文件刷新后，Spring Cloud Config 中的 Client 也不能及时感知到（8-1 小节向大家演示的是当配置文件变化之后，Spring Cloud Config Server 能够及时感知到），另外也没有失败重试功能，那么本文我将带领大家，来把这几个问题搞清楚。

服务化

我们在前面的配置中，当 Config Client 需要从 Config Server 上获取配置数据时，我们都是直接在 Config Client 的配置文件中写上 Config Server 的地址，类似下面这种架构：



这种写法相当于将 Config Client 和 Config Server 绑定死了，以后 Config Server 的地址不能变，Config Server 也不能挂，否则 Config Client 就获取不到信息了，而且这种方式也破坏了我们微服务的整体架构，即服务之间互相调用，获取对方的信息都是去服务注册中心上获取，所以我们要对这种结构进行改造，改造成下面这种结构：



即当 Config Server 启动时，将自己注册到服务注册中心 Eureka 上，所有 Config Client 都从 Eureka 上去获取 Config Server 的信息，这样我们就成功将 Config Server 和 Config Client 解耦了，Eureka 在这里依然扮演了数据中心的角色。

好了，那么接下来我们就来看看如何实现服务化。

本文的案例在 8-2 小节的案例上继续完成，大家可以直接在 8-2 小节的案例上继续完善。

首先我们在 CloudConfig 项目中添加一个名为 eureka 的 module ，这便是我们的注册中心了。注册中心的添加与配置，读者可以参考前面的文章，这里我就不再赘述，注册中心创建成功后，启动注册中心 eureka。

eureka 启动之后，我们要对之前的 Config Server 和 Config Client 进行服务化改造，首先给这两个模块分别加上 eureka client 依赖，如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

添加完成后，再修改配置，在 Config Server 和 Config Client 中分别添加如下配置，表示将这两个服务注册到 eureka 上面：

```
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

好了，上面这两条是 Config Server 和 Config Client 共同的配置。接下来，我们还需要在 Config Client 中再额外修改一些配置，修改后的 Config Client 的 bootstrap.properties 的配置内容如下：

```
spring.application.name=client1
server.port=8002
spring.cloud.config.profile=dev
spring.cloud.config.label=master
#spring.cloud.config.uri=http://localhost:8001/
spring.cloud.config.discovery.service-id=config-server
spring.cloud.config.discovery.enabled=true

spring.cloud.config.username=javaboy
spring.cloud.config.password=123

eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

这里其实就是使用 spring.cloud.config.discovery.service-id=config-server 和 spring.cloud.config.discovery.enabled=true 两个配置代替了原来的 spring.cloud.config.uri=http://localhost:8001/ 。其中 spring.cloud.config.discovery.service-id=config-server 表示配置 Config Server 的实例 id ，Config Client 将根据这个 id 去 eureka 上面查找 Config Server 的信息， spring.cloud.config.discovery.enabled=true 则表示开启通过 eureka 获取 Config Server 的功能。

配置完成后，我们就可以启动 Config Client 了。启动成功后，我们再次访问 Config Client 中的 /hello 接口，获取到的数据和前面的一样，说明配置成功。

配置成功之后，以后我们所有的 Config Client 都按照上面的配置信息来配置，即开启通过注册中心来访问 Config Server 的功能，同时指定 Config Server 的实例 id ，而不用在 Config Client 中硬编码 Config Server 的地址。

动态刷新

接下来我们再来看一下配置文件动态刷新的问题，在 8-1 小节中，我们向大家演示过，当 GitHub 仓库中配置文件发生改变后，如果我们刷新 Config Server 中的请求地址，会发现数据也跟着变化了，即 Config Server 是能够及时感知到配置文件的变化的，但是这种感知却不能够传递到 Config Client 中去，即 Config Client 是无法及时感知到配置文件的变化的，默认情况下，只有 Config Client 重启，才能够加载到最新的配置文件数据，如何让 Config Client 也能动态刷新配置数据呢？

想要让 Config Client 动态刷新配置数据，其实很容易，首先我们需要在 Config Client 中引入 actuator 依赖，如下：

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

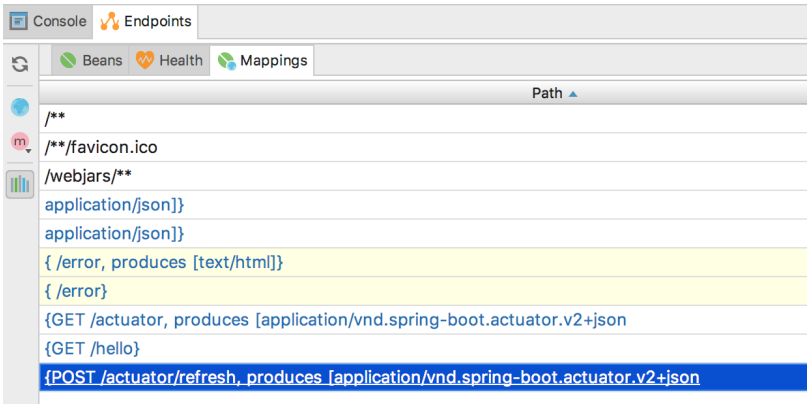
如果是 Spring Cloud Finchley 版（不含）之前的版本，那么直接添加依赖就完事了，不需要做额外配置，但是在 Finchley 版之后，因为采用的 Spring Boot 版本是 2.0.x ，Spring Boot 2 之后，考虑到数据安全，actuator 默认只开放了两个接口 health 和 info 。因此，在 Greenwich 版中，除了添加依赖外，我们还需要手动添加如下配置，表示暴露 refresh 接口：

```
management.endpoints.web.exposure.include=refresh
```

另外，我们还需要在 Config Client 中的 HelloController 上添加一个 @RefreshScope 注解，表示当调用 /refresh 接口时，动态更新容器中的数据，如下：

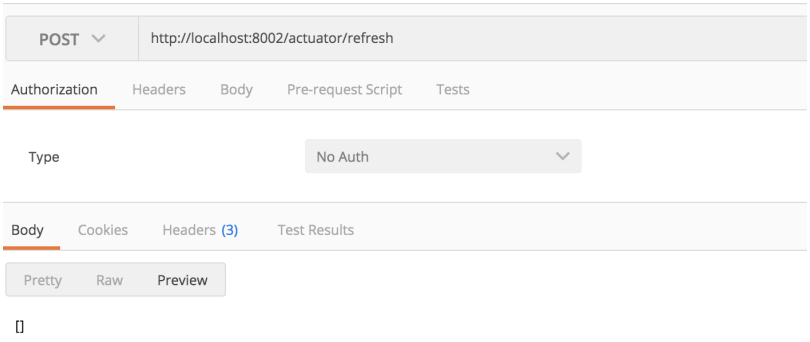
```
@RestController
@RefreshScope
public class HelloController {
    @Value("${javaboy}")
    String hello;
    @GetMapping("/hello")
    public String hello() {
        return hello;
    }
}
```

配置完成后，我们再重新启动 Config Client ，启动成功后，我们在 IntelliJ IDEA 控制台的 Endpoints 中就可以看到 /actuator/refresh 接口已经暴露出来了，如下：



一会儿，我们将通过调用这个接口实现配置文件的动态刷新。

接下来，我们分别启动 Config Server 和 Config Client ，然后修改本地仓库中的配置文件。修改完成后，提交到远程仓库，此时我们直接访问 Config Client 的 `/hello` 接口，发现数据并未发生变化，别急，我们先调用 Config Client 的 `/actuator/refresh` 接口，注意这个接口的调用是一个 POST 请求，如下：



调用成功之后，再去调用 Config Client 的 `/hello` 接口，此时发现配置文件已经发生变化了。

这样，我们在不重启 Config Client 的情况下，就能够动态刷新配置了，可能有人还是觉得这样太麻烦了，因为所有的微服务都要挨个去发送 `/actuator/refresh` 请求，这个工作量也不小，那么这个工作有没有可能做进一步的简化呢？当然是可以的，我们在后面的文章会继续为大家介绍。

请求失败重试

请求失败重试这也是一个非常常见的需求，在前面的文章中为大家介绍过微服务调用过程中的请求失败重试问题，那么 Config Client 在调用 Config Server 时，一样也会发生请求失败的问题。我们平时做开发，一般来说公司的网络都是杠杠滴，但是在实际生产环境中，网络问题各种各样，我们必须考虑弱网环境下如何保证服务的高可用性，那么请求失败重试就是策略之一。

要在 Config Client 中实现请求失败重试，其实非常容易，添加如下两个依赖即可：

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

添加完成后，还需要我们在 Config Client 的配置文件中添加如下配置：

```
spring.cloud.config.fail-fast=true
```

这行配置表示开启失败快速响应。什么是失败快速响应呢？默认情况下，当 Config Client 访问 Config Server 失败时，并不会立马报错，而是会等到用这个数据时，才会抛出异常，以我们前面的代码为例，如果 Config Client 访问 Config Server 失败，并不会立马抛出异常，而是等到在 Config Client 中使用注入进来的 javaboy 这个变量时，发现没有这个变量，此时才会抛出异常。这个时候项目已经启动失败了，停止运行了，所以也不会有失败重试什么事了，因此我们要开启失败快速响应。什么是失败快速响应呢？就是当 Config Client 访问 Config Server 失败时，就不再执行后面的流程了，立马做出响应，重试 or 抛异常。

添加完这个配置之后，为了演示执行效果，接下来我们再做一点点修改，由于目前我们的 Config Server 是有安全认证的，Config Client 必须要有用户名密码才能访问到 Config Server 中的数据。我们暂时先注释掉 Config Client 中访问 Config Server 的用户名密码，即如下两行：

```
#spring.cloud.config.username=javaboy
#spring.cloud.config.password=1234
```

注释掉之后，我们再去启动 Config Client ，此时就会启动失败，我们来观察效果重试效果：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.4.RELEASE)

2019-05-12 12:58:50.225  INFO 78271 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://192.168.0.106:8001/
2019-05-12 12:58:51.277  INFO 78271 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://192.168.0.106:8001/
2019-05-12 12:58:52.391  INFO 78271 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://192.168.0.106:8001/
2019-05-12 12:58:53.615  INFO 78271 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://192.168.0.106:8001/
2019-05-12 12:58:54.961  INFO 78271 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://192.168.0.106:8001/
2019-05-12 12:58:56.440  INFO 78271 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://192.168.0.106:8001/
2019-05-12 12:58:56.459 ERROR 78271 --- [           main] o.s.boot.SpringApplication               : Application run failed
```

可以看到，请求一共发送了6次，第一次失败之后，还重试了5次。这是默认的请求重试策略，开发者也可以自定义请求重试的相关参数，如下：

```
spring.cloud.config.retry.initial-interval=1000
spring.cloud.config.retry.multiplier=1.1
spring.cloud.config.retry.max-interval=2000
```

这四个配置含义如下：

1. max-attempts 表示最大请求次数，默认值为 6 ，就是大家在上图看到的情况；
2. initial-interval 表示请求重试的初始时间间隔；
3. multiplier 表示时间的间隔乘数，由于网络抖动一般都是有规律的，为了防止请求重试时连续的冲突，我们需要一个时间间隔乘数，这里我设置了间隔乘数为 1.2 ，表示第一次重试间隔时间为 1 s，第二次间隔时间为 1.2 秒，第三次间隔时间为 1.44 秒…；
4. max-interval 表示重试的最大间隔时间。

开启了请求重试机制之后，即使在弱网环境下，我们也能有效保证服务的可用性。

小结

本文主要向大家介绍了分布式配置中心 Spring Cloud Config 中三个常见的问题，服务化、配置数据动态刷新以及请求失败重试。服务化降低了 Config Server 和 Config Client 之间的耦合度，使我们的项目架构更加规范；动态刷新则让我们在不重启 Config Client 的情况下，能够刷新配置数据；最后的请求重试则保证了弱网环境下服务的可用性，在实际生产项目中，这三个基本上也都是必配的，大家需要认真掌握。

23 Spring Cloud Config 中配置文件的加密与解密

25 Docker 简介与消息中间件安装