17 Resilience4j 在微服务中的应用

更新时间：2019-06-26 10:11:00

> 机会不会上门来找人，只有人去找机会。
>
> ——狄更斯

上篇文章首先和大家大致说了下断路器 Hystrix，然后详细说了下 Resilience4j 的用法，但是都是 JavaSE 环境下的用法，并没有涉及到在微服务中的使用。Resilience4j 最终我们还是要在微服务中体现它的价值，我们学习它使用它是为了解决微服务中的一些痛点，因此，本文我就来和大家分享下 Resilience4j 在微服务中的具体用法。

准备工作

首先我们需要先搭建一个测试环境，我这里创建一个名为 `Resilience4j-SpringBoot` 的父工程，然后在父工程中创建一个名为 eureka 的子项目来搭建服务注册中心，然后再创建一个名为 provider 的服务提供者，将服务提供者注册到 eureka 上，然后再在 provider 中提供一个 `/hello` 接口，内容如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(String name) {
        String s = "hello " + name + " !";
        System.out.println(s+">>>>>"+new Date());
        int i = 1 / 0;
        return s;
    }
}
```

这里接口中，我特意设置了一个异常， consumer 在调用这个接口的时候会调用失败，方便我们去测试请求重试等功能。

接下来再来创建一个 consumer。consumer 在创建的过程中只需要加入两个基本的依赖即可，即 spring-boot-starter-web 和 spring-cloud-starter-netflix-eureka-client ，其它的依赖我们在下面具体的使用过程中来说，创建完成后，也将 consumer 注册到服务注册中心上。

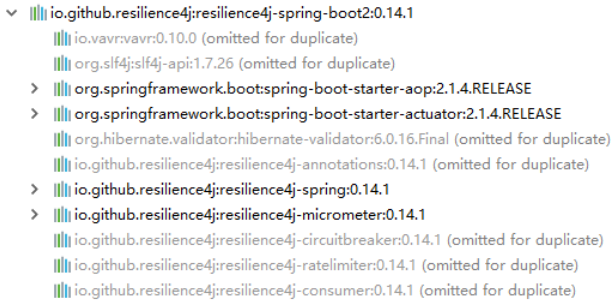
这就是我们的准备工作，由于准备工作比较简单，大家不清楚的可以参考服务注册与消费一文，我这里就不再详细介绍。

Retry

首先要和大家介绍的功能就是重试功能，开发环境一般比较稳定。微服务之间的调用，除非是逻辑错误导致调用失败，一般来说可能很少会遇到网络原因导致的调用失败，但是在生产环境中，网络原因导致的调用失败却是一个不能够忽略的问题，由于网络抖动造成的调用失败，我们一定要进行请求重试。

上文介绍的 Resilience4j 中的功能，在微服务中的使用都有两种不同的用法，一种就是借助 Spring AOP ，直接通过一个注解来实现相关的功能；还有一种就是和上文一样，通过编程的方式来实现，这里分别来和大家介绍。

不过无论哪一种，我们都是需要首先添加依赖，不同于上篇文章我们根据 Resilience4j 提供的功能挨个加依赖，在微服务中，我们可以直接引用 resilience4j-spring-boot2 依赖，这个依赖中包含了 Resilience4j 中提供的所有功能，也包含了所需要的 Spring AOP 的依赖，如下图：



依赖代码如下：

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>0.14.1</version>
</dependency>
```

AOP 式

依赖添加成功后，接下来在 application.yml 文件中配置 Retry 参数，如下：

```
resilience4j.retry:
  retryAspectOrder: 399
  backends:
    retryBackendA:
      maxRetryAttempts: 3
      waitDuration: 600
      eventConsumerBufferSize: 1
      enableExponentialBackoff: true
      exponentialBackoffMultiplier: 2
      enableRandomizedWait: false
      randomizedWaitFactor: 2
      retryExceptionPredicate: com.justdojava.consumer.RecordFailurePredicate
      retryExceptions:
        - java.io.IOException
      ignoreExceptions:
        - com.justdojava.consumer.IgnoredException
```

关于这一段的配置，解释如下：

1. retryAspectOrder 表示 Retry 的一个优先级。默认情况下， Retry 的优先级高于 bulkhead 、 Circuit breaker 以及 rateLimiter ，即 Retry 会先于另外三个执行。 Retry、 bulkhead 、 Circuit breaker 以及 rateLimiter 的优先级数值默认分别是 Integer.MAX_VALUE-3、Integer.MAX_VALUE-2、Integer.MAX_VALUE-1 以及 Integer.MAX_VALUE ，即数值越小，优先级越高；
2. backends 属性中我们可以配置不同的 Retry 策略，给不同的策略分别取一个名字， retryBackendA 就是一个 Retry 策略的名字。在 Java 代码中，我们将直接通过指定 Retry 策略的名字来使用某一种 Retry 方案；
3. maxRetryAttempts 表示最大重试次数；
4. waitDuration 表示下一次重试等待时间，最小为100 ms ；
5. eventConsumerBufferSize 表示重试事件缓冲区大小；
6. enableExponentialBackoff 表示是否开启指数退避抖动算法，当一次调用失败后，如果在相同的时间间隔内发起重试，有可能发生连续的调用失败，因此可以开启指数退避抖动算法；
7. exponentialBackoffMultiplier 表示时间间隔乘数；
8. enableRandomizedWait 表示下次重试的时间间隔是否随机， enableRandomizedWait 和 enableExponentialBackoff 默认为 false ，并且这两个不可以同时开启；
9. retryExceptionPredicate 类似于我们上文所说的什么样的异常会被认定为请求失败，这里的 RecordFailurePredicate是一个自定义的类；
10. retryExceptions 表示需要重试的异常；
11. ignoreExceptions 表示忽略的异常。

RecordFailurePredicate 类的定义如下：

```
public class RecordFailurePredicate implements Predicate<Throwable> {
    @Override
    public boolean test(Throwable throwable) {
        return true;
    }
}
```

方便起见，我这里未做判断，直接返回 true 。

还有一个自定义的异常 IgnoredException ，如下：

```
public class IgnoredException extends Exception {
}
```

配置完成后，接下来再在项目启动类中配置一个 RestTemplate ， 如下：

```
@SpringBootApplication
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

然后创建一个 HelloService 来进行远程调用，如下：

```
@Service
@Retry(name = "retryBackendA")
public class HelloService {
    @Autowired
    RestTemplate restTemplate;

    public String hello(String name) {
        return restTemplate.getForObject("http://provider/hello?name={1}", String.class, name);
    }
}
```

HelloService 本身是一个非常常规的类，RestTemplate 相信大家也是再熟悉不过了，唯一和我们前面学过的不同的地方是这里类上多了一个 @Retry(name = “retryBackendA”) 注解，这个注解表示在当前所有类中开启请求失败重试功能，请求失败重试策略就是 retryBackendA ，当然这个注解也可以加在某一个具体的方法上，表示只有该方法开启请求失败重试功能。

做完这些之后，我们就可以分别启动 eureka、provider 以及 consumer 了，然后在浏览器中访问 consumer 接口，观察 provider 的日志输出，就可以看到日志一共打印了三次，这里发生了请求失败重试。

编程式

在框架中，通过这种面向切面编程的方式来引用 Resilience4j 中的 Retry 功能固然很方便。不过， Resilience4j 也支持编程式引用，编程式引用的方式就和我们上篇文章介绍的用法差不多了，举例如下：

```
@RestController
public class UseHelloController {
    @Autowired
    HelloService helloService;
    @GetMapping("/hello2")
    public String hello2(String name) {
        RetryConfig config = RetryConfig.custom()
                .maxAttempts(3)
                .waitDuration(Duration.ofMillis(5000))
                .build();
        Retry retry = Retry.of("id", config);
        Try<String> result = Try.ofSupplier(Retry.decorateSupplier(retry, () -> helloService.hello(name)));
        return result.get();
    }
}
```

通过编程式来引用 Resilience4j 中的 Retry 功能，则不再需要 application.properties 中的 Retry 相关配置，也不需要在 HelloService 类上添加 @Retry 注解，所有关于重试的配置都是通过 Java 代码来实现，至于 Java 代码配置中的含义，则和上文介绍的一致，这里不再赘述。

CircuitBreaker

Resilience4j 中断路器的用法和上文也是基本一致，分为两种，可以通过 AOP 的方式使用，也可以通过编程式使用，我们分别来看。

AOP 式

由于在 Retry 中已经添加了 `resilience4j-spring-boot2` 依赖，这里我就不再重复说添加依赖的事了。在上个案例的基础上，我们继续在 application.yml 文件中添加如下配置：

```yaml
resilience4j.circuitbreaker:
  backends:
    backendA:
      ringBufferSizeInClosedState: 5
      ringBufferSizeInHalfOpenState: 3
      waitInterval: 5000
      failureRateThreshold: 50
      eventConsumerBufferSize: 10
      registerHealthIndicator: true
      recordFailurePredicate: com.justdojava.consumer.RecordFailurePredicate
      recordExceptions:
        - org.springframework.web.client.HttpServerErrorException
      ignoreExceptions:
        - org.springframework.web.client.HttpClientErrorException
```

这里配置也很好理解，大部分参数和我们上篇文章介绍的一致：

1. backendA 是断路器策略的命名，和 Retry 类似，一会也是通过注解来引用这个策略；
2. ringBufferSizeInClosedState 表示断路器关闭状态下，环形缓冲区的大小；
3. ringBufferSizeInHalfOpenState 表示断路器处于 HalfOpen 状态下，环形缓冲区的大小；
4. waitInterval 表示断路器从 open 切换到 half closed 状态时，需要保持的时间；
5. failureRateThreshold 表示故障率阈值百分比，超过这个阈值，断路器就会打开；
6. eventConsumerBufferSize 表示事件缓冲区大小；
7. registerHealthIndicator 表示开启健康检测。

和 Retry 类似，在 Circuit Breaker 中，我们也可以通过 circuitBreakerAspectOrder 属性来修改 Circuit Breaker 的执行优先级。

配置完成后，接下来我们来定义一个名为 HelloServiceCircuitBreaker 的类，在这个类中，来定义服务请求方法：

```java
@Service
@CircuitBreaker(name = "backendA")
public class HelloServiceCircuitBreaker {
    @Autowired
    RestTemplate restTemplate;

    public String hello(String name) {
        return restTemplate.getForObject("http://provider/hello?name={1}", String.class, name);
    }
}
```

这里通过 @CircuitBreaker 注解来启用断路器。最后，我们在 UseHelloController 中调用这个方法即可。

但是这种写法有一个问题，就是没法进行服务容错降级，如果希望进行服务容错降级，那么还是需要我们上篇文章提到的通过编程实现断路器功能。

编程式

通过编程实现断路器功能，就不再需要 application.yml 中的配置了，也不需要在类上添加 @CircuitBreaker(name = “backendA”) 注解，所有的相关配置都是在 Java 代码中完成，和上篇文章基本一样，如下：

```
public String hello2(String name) {
    CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
            .failureRateThreshold(50)
            .waitDurationInOpenState(Duration.ofMillis(1000))
            .ringBufferSizeInHalfOpenState(20)
            .ringBufferSizeInClosedState(20)
            .build();
    io.github.resilience4j.circuitbreaker.CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker("backendA", circuitBreakerConfig);
    Try<String> supplier = Try.ofSupplier(io.github.resilience4j.circuitbreaker.CircuitBreaker
            .decorateSupplier(circuitBreaker,
                    () -> restTemplate.getForObject("http://provider/hello?name={1}", String.class, name)))
            .recover(Exception.class, "有异常，访问失败!");
    return supplier.get();
}
```

我来和大家捋一捋上面这段代码的思路：

1. 首先利用 Java 代码创建一个 CircuitBreakerConfig 出来，然后配置一下故障率阈值，等待时间以及环形缓冲区大小等；
2. 根据第一步创建出来的 CircuitBreakerConfig ，再去创建一个 CircuitBreaker 对象；
3. 通过 Try.ofSupplier 方法去发送一个请求，如果请求抛出异常，则在 recover 方法中进行服务降级处理，recover 可以写多个。

最后，在 UseHelloController 中调用这里的 hello2 方法去访问 provider 中的接口。接口调用失败后， consumer 中自动进行服务降级，最终返回字符串为 `有异常，访问失败!` 。

RateLimiter

接下来我们再来说一说限流工具 RateLimiter ，限流工具的用法基本上和前两个差不多，可以通过 AOP 的方式使用，也可以通过编程式来使用，下面分别来介绍。

AOP 式

通过 AOP 的方式来使用限流工具，首先在 application.yml 配置文件中添加 RateLimiter 相关配置，如下：

```
resilience4j.ratelimiter:
  limiters:
    backendA:
      limitForPeriod: 1
      limitRefreshPeriodInMillis: 5000
      timeoutInMillis: 5000
      subscribeForEvents: true
      registerHealthIndicator: true
      eventConsumerBufferSize: 100
```

关于这段配置，我说如下几点：

1. backendA 在这里依然表示配置的名称，在 Java 代码中，我们将通过指定限流工具的名称来使用某一种限流策略；
2. limitForPeriod 表示请求频次的阈值；
3. limitRefreshPeriodInMillis 表示频次刷新的周期；
4. timeoutInMillis 许可期限的等待时间，默认为5秒；
5. subscribeForEvents 表示开启事件订阅；
6. registerHealthIndicator 表示开启健康监控；
7. eventConsumerBufferSize 表示事件缓冲区大小。

配置完成后，创建一个 HelloServiceRateLimiter 类，内容如下：

```
@Service
@RateLimiter(name = "backendA")
public class HelloServiceRateLimiter {
    @Autowired
    RestTemplate restTemplate;
    public String hello(String name) {
        return restTemplate.getForObject("http://provider/hello?name={1}", String.class, name);
    }
}
```

这里就是一个很常规的服务调用，然后在 UseHelloController 中调用该方法：

```
@GetMapping("/rl")
public void rateLimiter(String name) {
    for (int i = 0; i < 5; i++) {
        String hello = helloServiceRateLimiter.hello(name);
    }
}
```

为了测试出效果，这里使用了一个 for 循环，循环中连续发送五次请求，我们发现服务端打印日志如下：

```
hello 江南一点雨 !>>>>>Sun Apr 14 21:20:03 CST 2019
hello 江南一点雨 !>>>>>Sun Apr 14 21:20:05 CST 2019
hello 江南一点雨 !>>>>>Sun Apr 14 21:20:10 CST 2019
hello 江南一点雨 !>>>>>Sun Apr 14 21:20:15 CST 2019
hello 江南一点雨 !>>>>>Sun Apr 14 21:20:20 CST 2019
```

可以看到，限流已经生效。

编程式

当然，这里的效果也可以通过编程来实现。通过编程实现代码和上篇文章介绍的基本一致，同时，这里也不再需要在 application.yml 中添加配置，所有的条件通过 Java 代码来配置即可，如下：

```
public void hello2(String name) {
    RateLimiterConfig config = RateLimiterConfig.custom()
            .limitRefreshPeriod(Duration.ofMillis(5000))
            .limitForPeriod(1)
            .timeoutDuration(Duration.ofMillis(6000))
            .build();
    RateLimiterRegistry rateLimiterRegistry = RateLimiterRegistry.of(config);
    RateLimiter rateLimiter = RateLimiter.of("backendB", config);
    Supplier<String> supplier = RateLimiter.decorateSupplier(rateLimiter, () ->
            restTemplate.getForObject("http://provider/hello?name={1}", String.class, name)
    );
    for (int i = 0; i < 5; i++) {
        Try<String> aTry = Try.ofSupplier(supplier);
        System.out.println(aTry.get());
    }
}
```

这里的代码和我们前文所讲的基本一致，创建一个 Supplier 对象，然后使用 Try.of 方法执行调用，且调用多次。然后在 UseHelloController 中调用这个方法：

```
@GetMapping("/r2")
public void rateLimiter2(String name) {
    helloServiceRateLimiter.hello2(name);
}
```

观察 provider 中的日志输出，我们可以看到，限流已经生效了。

小结

本文主要向大家介绍了 Resilience4j 在微服务中的应用。相对于 Hystrix ， Resilience4j 更加轻便简洁，而且到处充满了 JDK8 的元素，确实非常好用，也是未来处理微服务系统稳定性的一个方向。

本文作者：纯洁的微笑、江南一点雨

16 Resilience4j 基本用法详解

18 Micrometer 实现微服务监控