10.1 Resilience4j 简介

Resilience4j 是 Spring Cloud Greenwich 版推荐的容错解决方案，相比 Hystrix，Resilience4j 专为 Java8 以及函数式编程而设计。

Resilience4j 主要提供了如下功能：

1. 断路器
2. 限流
3. 基于信号量的隔离
4. 缓存
5. 限时
6. 请求重试

10.2 基本用法

首先搭建一个简单的测试环境。

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

10.2.1 断路器

Resilience4j 提供了很多功能，不同的功能对应不同的依赖，可以按需添加。

使用断路器，则首先添加断路器的依赖：

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-circuitbreaker</artifactId>
    <version>0.13.2</version>
</dependency>
```

一个正常执行的例子：

```
@Test
public void test1() {
    //获取一个CircuitBreakerRegistry实例，可以调用ofDefaults获取一个
CircuitBreakerRegistry实例，也可以自定义属性。
    CircuitBreakerRegistry registry = CircuitBreakerRegistry.ofDefaults();
    CircuitBreakerConfig config = CircuitBreakerConfig.custom()
            //故障率阈值百分比，超过这个阈值，断路器就会打开
            .failureRateThreshold(50)
            //断路器保持打开的时间，在到达设置的时间之后，断路器会进入到 half open 状态
            .waitDurationInOpenState(Duration.ofMillis(1000))
            //当断路器处于half open 状态时，环形缓冲区的大小
            .ringBufferSizeInHalfOpenState(2)
            .ringBufferSizeInClosedState(2)
            .build();
    CircuitBreakerRegistry r1 = CircuitBreakerRegistry.of(config);
```

```
    CircuitBreaker cb1 = r1.circuitBreaker("javaboy");
    CircuitBreaker cb2 = r1.circuitBreaker("javaboy2", config);
    CheckedFunction0<String> supplier =
CircuitBreaker.decorateCheckedSupplier(cb1, () -> "hello resilience4
    Try<String> result = Try.of(supplier)
            .map(v -> v + " hello world");
    System.out.println(result.isSuccess());
    System.out.println(result.get());
}
```

一个出异常的断路器：

```
@Test
public void test2() {
    CircuitBreakerConfig config = CircuitBreakerConfig.custom()
            //故障率阈值百分比，超过这个阈值，断路器就会打开
            .failureRateThreshold(50)
            //断路器保持打开的时间，在到达设置的时间之后，断路器会进入到 half open 状态
            .waitDurationInOpenState(Duration.ofMillis(1000))
            //当断路器处于half open 状态时，环形缓冲区的大小
            .ringBufferSizeInClosedState(2)
            .build();
    CircuitBreakerRegistry r1 = CircuitBreakerRegistry.of(config);
    CircuitBreaker cb1 = r1.circuitBreaker("javaboy");
    System.out.println(cb1.getState());//获取断路器的一个状态
    cb1.onError(0, new RuntimeException());
    System.out.println(cb1.getState());//获取断路器的一个状态
    cb1.onError(0, new RuntimeException());
    System.out.println(cb1.getState());//获取断路器的一个状态
    CheckedFunction0<String> supplier =
CircuitBreaker.decorateCheckedSupplier(cb1, () -> "hello resilience4j");
    Try<String> result = Try.of(supplier)
            .map(v -> v + " hello world");
    System.out.println(result.isSuccess());
    System.out.println(result.get());
}
```

注意，由于 ringBufferSizeInClosedState 的值为 2，表示当有两条数据时才会去统计故障率，所以，下面的手动故障测试，至少调用两次 onError ，断路器才会打开。

10.2.2 限流

RateLimiter 本身和前面的断路器很像。

首先添加依赖：

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-ratelimiter</artifactId>
    <version>0.13.2</version>
</dependency>
```

限流测试：

```
@Test
public void test3() {
```

```
    RateLimiterConfig config = RateLimiterConfig.custom()
            .limitRefreshPeriod(Duration.ofMillis(1000))
            .limitForPeriod(4)
            .timeoutDuration(Duration.ofMillis(1000))
            .build();
    RateLimiter rateLimiter = RateLimiter.of("javaboy", config);
    CheckedRunnable checkedRunnable =
RateLimiter.decorateCheckedRunnable(rateLimiter, () -> {
        System.out.println(new Date());
    });
    Try.run(checkedRunnable)
            .andThenTry(checkedRunnable)
            .andThenTry(checkedRunnable)
            .andThenTry(checkedRunnable)
            .onFailure(t -> System.out.println(t.getMessage()));
}
```

10.2.3 请求重试

首先第一步还是加依赖：

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-retry</artifactId>
    <version>0.13.2</version>
</dependency>
```

案例：

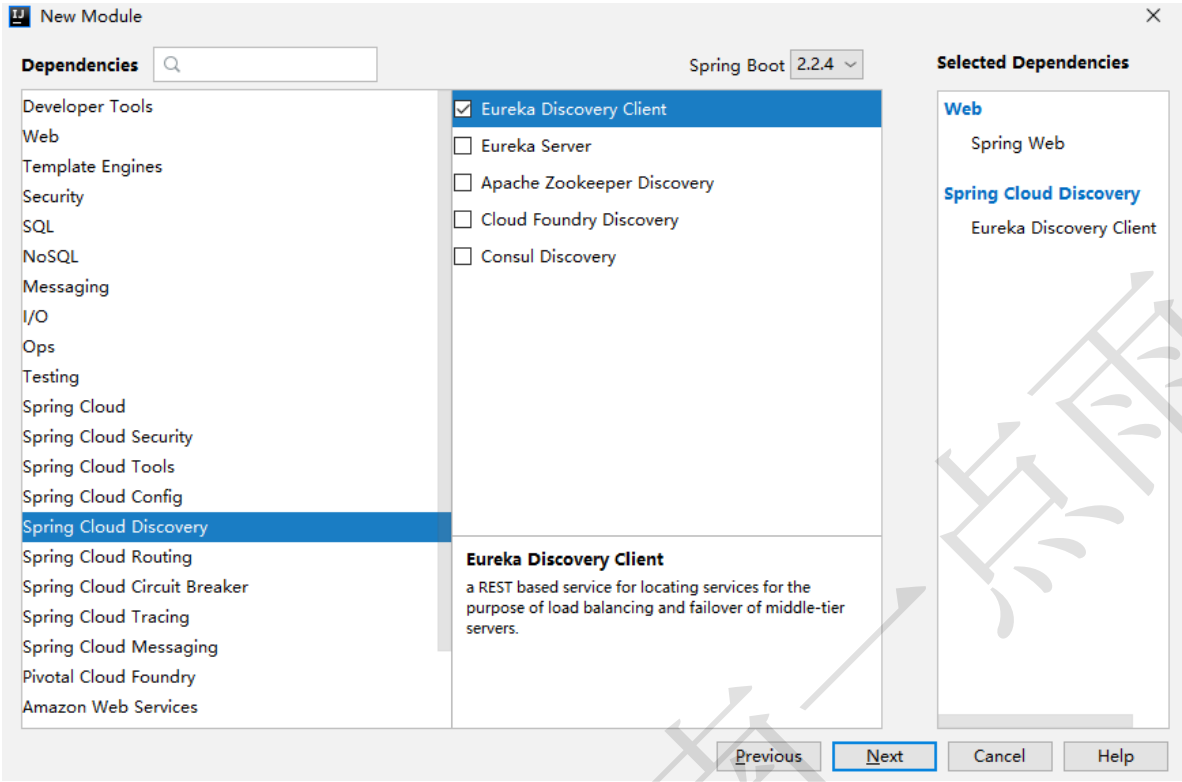
```
@Test
public void test4() {
    RetryConfig config = RetryConfig.custom()
            //重试次数
            .maxAttempts(2)
            //重试间隔
            .waitDuration(Duration.ofMillis(500))
            //重试异常
            .retryExceptions(RuntimeException.class)
            .build();
    Retry retry = Retry.of("javaboy", config);
    Retry.decorateRunnable(retry, new Runnable() {
        int count = 0;
        //开启了重试功能之后，run 方法执行时，如果抛出异常，会自动触发重试功能
        @Override
        public void run() {
            if (count++ < 3) {
                throw new RuntimeException();
            }
        }
    }).run();
}
```

10.3 结合微服务

Retry、CircuitBreaker、RateLimiter

10.3.1 Retry

首先创建一个 Spring Boot 项目，创建时，添加 eureka-client 依赖，使之能够注册到 eureka 上。



项目创建成功后，手动添加 Resilience4j 依赖：

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>1.3.1</version>
    <exclusions>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-circuitbreaker</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-ratelimiter</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-bulkhead</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-timelimiter</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

resilience4j-spring-boot2 中包含了 Resilience4j 的所有功能，但是没有配置的功能无法使用，需要将之从依赖中剔除掉。

接下来，在 application.yml 中配置 retry：

```yaml
resilience4j:
  retry:
    retry-aspect-order: 399 # 表示Retry的优先级
    backends:
      retryA:
        maxRetryAttempts: 5 # 重试次数
        waitDuration: 500 # 重试等待时间
        exponentialBackoffMultiplier: 1.1 # 间隔乘数
        retryExceptions:
          - java.lang.RuntimeException
spring:
  application:
    name: resilience4j
server:
  port: 5000
eureka:
  client:
    service-url:
      defaultZone: http://localhost:1111/eureka
```

最后，创建测试 RestTemplate 和 HelloService：

```java
@SpringBootApplication
public class Resilience4j2Application {

    public static void main(String[] args) {
        SpringApplication.run(Resilience4j2Application.class, args);
    }

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

@Service
@Retry(name = "retryA")//表示要使用的重试策略
public class HelloService {
    @Autowired
    RestTemplate restTemplate;
    public String hello() {
        return restTemplate.getForObject("http://localhost:1113/hello",
String.class);
    }
}

@RestController
public class HelloController {
    @Autowired
    HelloService helloService;
    @GetMapping("/hello")
    public String hello() {
        return helloService.hello();
    }
}
```

10.3.2 CircuitBreaker

首先从依赖中删除排除 CircuitBreaker。

然后在 application.yml 中进行配置：

```
resilience4j:
  retry:
    retry-aspect-order: 399 # 表示Retry的优先级
    backends:
      retryA:
        maxRetryAttempts: 5 # 重试次数
        waitDuration: 500 # 重试等待时间
        exponentialBackoffMultiplier: 1.1 # 间隔乘数
        retryExceptions:
          - java.lang.RuntimeException
  circuitbreaker:
    instances:
      cbA:
        ringBufferSizeInClosedState: 5
        ringBufferSizeInHalfOpenState: 3
        waitInterval: 5000
        recordExceptions:
          - org.springframework.web.client.HttpServerErrorException
    circuit-breaker-aspect-order: 398
```

配置完成后，用 @CircuitBreakder 注解标记相关方法：

```
@Service
@CircuitBreaker(name = "cbA", fallbackMethod = "error")
public class HelloService {
    @Autowired
    RestTemplate restTemplate;

    public String hello() {
        return restTemplate.getForObject("http://localhost:1113/hello",
String.class);
    }

    public String error(Throwable t) {
        return "error";
    }
}
```

@CircuitBreaker 注解中的 name 属性用来指定 circuitbreaker 配置，fallbackMethod 属性用来指定服务降级的方法，需要注意的是，服务降级方法中，要添加异常参数。

10.3.3 RateLimiter

RateLimiter 作为限流工具，主要在服务端使用，用来保护服务端的接口。

首先在 provider 中添加 RateLimiter 依赖：

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>1.2.0</version>
    <exclusions>
        <exclusion>
```

```
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-circuitbreaker</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-bulkhead</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-timelimiter</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

接下来，在 provider 的 application.properties 配置文件中，去配置 RateLimiter：

```
# 这里配置每秒钟处理一个请求
resilience4j.ratelimiter.limiters.rlA.limit-for-period=1
resilience4j.ratelimiter.limiters.rlA.limit-refresh-period=1s
resilience4j.ratelimiter.limiters.rlA.timeout-duration=1s
```

为了查看请求效果，在 provider 的 HelloController 中打印每一个请求的时间：

```
@Override
@RateLimiter(name = "rlA")
public String hello() {
    String s = "hello javaboy:" + port;
    System.out.println(new Date());
    return s;
}
```

这里通过 @RateLimiter 注解来标记该接口限流。

配置完成后，重启 provider。

然后，在客户端模拟多个请求，查看限流效果：

```
public String hello() {
    for (int i = 0; i < 5; i++) {
        restTemplate.getForObject("http://localhost:1113/hello", String.class);
    }
    return "success";
}
```

10.4 服务监控

微服务由于服务数量众多，所以出故障的概率很大，这种时候不能单纯的依靠人肉运维。

早期的 Spring Cloud 中，服务监控主要使用 Hystrix Dashboard，集群数据库监控使用 Turbine。

在 Greenwich 版本中，官方建议监控工具使用 Micrometer。

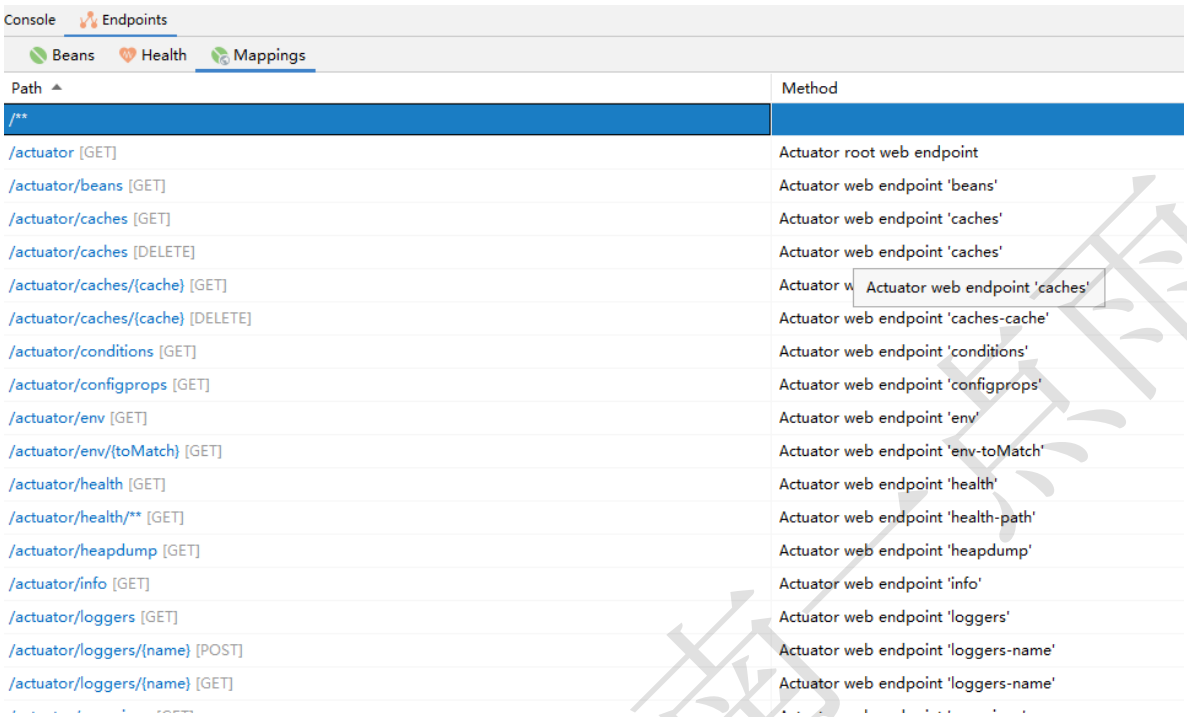
Micrometer：

1. 提供了度量指标，例如 timers、counters
2. 一揽子开箱即用的解决方案，例如缓存、类加载器、垃圾收集等等

新建一个 Spring Boot 项目，添加 Actuator 依赖。项目创建成功后，添加如下配置，开启所有端点：

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

然后就可以在浏览器查看项目的各项运行数据，但是这些数据都是 JSON 格式。



我们需要一个可视化工具来展示这些 JSON 数据。这里主要和大家介绍 Prometheus。

10.4.1 Prometheus

安装

```
wget
https://github.com/prometheus/prometheus/releases/download/v2.16.0/prometheus-
2.16.0.linux-amd64.tar.gz
tar -zxvf prometheus-2.16.0.linux-amd64.tar.gz
```

解压完成后，配置一下数据路径和要监控的服务地址：

```
cd prometheus-2.16.0.linux-amd64/
vi prometheus.yml
```

修改 prometheus.yml 配置文件，主要改两个地方，一个是数据接口，另一个是服务地址：

接下来，将 Prometheus 整合到 Spring Boot 项目中。

首先加依赖：

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

然后在 application.properties 配置中，添加 Prometheus 配置：

```
management.endpoints.web.exposure.include=*
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
management.endpoint.metrics.enabled=true
```

接下来启动 Prometheus。

启动命令：

```
./prometheus --config.file=prometheus.yml
```

启动成功后，浏览器输入 http://192.168.91.128:9090 查看 Prometheus 数据信息。

Grafana：https://grafana.com/grafana/download?platform=linux