16 Resilience4j 基本用法详解

更新时间：2019-06-24 10:31:26

> 书是人类进步的阶梯。
>
> ——高尔基

和传统架构相比，微服务有一个非常严重的问题常常被人诟病。什么问题呢？就是故障率。在传统架构中，我们可以将故障率降到很低，但是在微服务中，这一点却不太容易做到，因为单个微服务的故障率即使降低了，但假如有1000台机器，那么整个系统正常运行的概率就是 (1-故障率)^1000 ，经过这样一个幂运算之后，你会发现系统正常运行的概率低得吓人，而且如果处理不好的话还会发生服务雪崩(故障蔓延)，什么是服务雪崩呢？看如下一张调用图：

假设我们的客户端现在调用订单微服务去执行一个下单操作，在订单微服务中调用了支付微服务，假设现在支付微服务挂了，订单调用支付迟迟没有响应，在高并发环境下，订单微服务上积累的请求越来越多，本来订单微服务没有问题，现在也被拖挂了，这就是服务雪崩，也叫故障蔓延。那么我们能做的就是在某一个服务挂掉的时候，不能发生故障蔓延，同时整个系统还能以某种形式正常运行。这是我们的目标，为了实现这些目标，我们一般有如下几种解决方案。

Netflix Hystrix

Netflix Hystrix 断路器是 Spring Cloud 中最早就开始支持的一种服务调用容错解决方案，但是目前的 Hystrix 已经处于维护模式了，虽然这并不影响已经上线的项目，并且在短期内，你甚至也可以继续在项目中使用 Hystrix 。但是长远来看，处于维护状态的 Hystrix 走下历史舞台只是一个时间问题，特别是在 Spring Cloud Greenwich 版中，官方已经给出了 Hystrix 的建议替代方案（参见Spring Cloud Greenwich.RELEASE is now available一文），如下图：

CURRENT	REPLACEMENT
Hystrix	Resilience4j
Hystrix Dashboard / Turbine	Micrometer + Monitoring System
Ribbon	Spring Cloud Loadbalancer
Zuul 1	Spring Cloud Gateway
Archaius 1	Spring Boot external config + Spring Cloud Config

在 Spring Cloud Greenwich 版中，对于 Hystrix 以及 Hystrix Dashboard 官方都给出了替代方案。我们整个教程虽然基于最新的 Spring Cloud Greenwich 版，但是考虑到现实情况，本文中我还是先向大家大致介绍一下 Hystrix 的功能，后面我们会详细介绍 Resilience4j 的用法。

服务熔断

Hystrix 提供的第一个功能就是熔断。当然这里说的熔断不是股市的熔断，是指在当 A 服务调用 B 服务时，如果迟迟没有得到响应，就终止当前请求，而不是傻傻地等下去，避免 A 服务上出现大量的请求阻塞导致故障蔓延，这就是熔断，当然一般来说熔断器的功能还不止这些。当系统发生熔断之后，熔断器还要负责监控 B 服务，当发现 B 服务可以使用时，则再次发起调用。

服务降级

当服务调用发生熔断之后，接下来要做的就是服务降级了。例如当 A 服务调用 B 服务，没有调用成功，发生了熔断，那么此时 A 服务退而求其次，可能先从一个缓存微服务上先拿一个缓存数据顶着（我这里只是举一个简单的例子），避免给用户响应一个错误页面，这个就是服务降级。

请求缓存

通过对请求接口进行缓存，也能有效降低服务提供者的压力，当然请求缓存使用场景是那种数据更新频率较低但是访问又比较频繁的数据。

注意这里说的缓存是指 Hystrix 自带的缓存，在实际开发中，我们可能还需要配合自己的 Redis 缓存来实现更好的数据缓存效果。

请求合并

不同于 Dubbo ， Spring Cloud 中微服务之间的调用都是通过 HTTP 来实现的。由于 HTTP 协议本身的特点，在微服务调用时，如果是高并发小数据量的话，效率并不高，此时可以通过请求合并来实现，即将客户端的多个请求合并成一个，发送一个 HTTP 请求，拿到请求结果后，再将请求结果分发到不同的请求中，这样可以高效率传输数据。

当然， Hystrix 提供的功能不止这些，还包括资源隔离、对依赖服务分类等，这里就不再一一细说了。

Resilience4j

Resilience4j 是 Spring Cloud Greenwich 版推荐的容错解决方案，它是一个轻量级的容错库，受 Netflix Hystrix 的启发而设计，它专为 Java 8 和函数式编程而设计。Resilience4j 非常轻量级，因为它的库只使用 Vavr （以前称为 Javaslang ），它没有任何其他外部库依赖项。相比之下， Netflix Hystrix 对Archaius 具有编译依赖性，这导致了更多的外部库依赖，例如 Guava 和 Apache Commons 。而如果使用Resilience4j，你无需引用全部依赖，可以根据自己需要的功能引用相关的模块即可。

Resilience4j 也提供了一系列增强微服务可用性的功能，主要功能如下：

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

接下来，我们就先来看看 Resilience4j 中这几个功能的基本用法。

介绍 Resilience4j 的基本用法，我们需要首先搭建一个测试环境。这里我们先来说 Resilience4j 的基本用法，下篇文章再来说在 Spring Boot 中的用法，因此这里我只需要创建一个名为 Resilience4j 的普通的 Maven 项目即可。上面提到的 Resilience4j 中的功能，每一个功能都对应了一个依赖，这些依赖在下面的讲解中，我会分别添加进来，这样大家就能知道到底哪个依赖对应哪个功能。另外，由于我使用单元测试来演示 Resilience4j 的用法，因此创建好 Maven 项目后，再加入单元测试依赖，如下：

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

好了，工程创建好了准备工作就算 OK 了。

断路器

断路器初始化

使用 Resilience4j 提供的断路器功能，需要我们首先加入如下依赖：

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

这个库提供了一个基于 ConcurrentHashMap 的 CircuitBreakerRegistry ， CircuitBreakerRegistry 是线程安全的，并且是原子操作。开发者可以使用 CircuitBreakerRegistry 来创建和检索 CircuitBreaker 的实例 ，开发者可以直接使用默认的全局CircuitBreakerConfig 为所有 CircuitBreaker 实例创建 CircuitBreakerRegistry ，如下所示：

```
CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.ofDefaults();
```

当然开发者也可以提供自己的 CircuitBreakerConfig ，然后根据自定义的 CircuitBreakerConfig 来创建一个 CircuitBreakerRegistry 实例，进而创建 CircuitBreaker 实例。如果我们使用自定义的 CircuitBreakerConfig ，可以配置如下参数：

- 故障率阈值百分比，超过这个阈值，断路器就会打开

- 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
- 当断路器处于 half open 状态时，环形缓冲区的大小
- 当断路器关闭时，环形缓冲区的大小
- 自定义断路器中的事件操作
- 自定义 Predicate 以便计算异常是否被记录为失败事件

具体定义如下：

```
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
    .failureRateThreshold(50)
    .waitDurationInOpenState(Duration.ofMillis(1000))
    .ringBufferSizeInHalfOpenState(2)
    .ringBufferSizeInClosedState(2)
    .build();
CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.of(circuitBreakerConfig);
CircuitBreaker circuitBreaker2 = circuitBreakerRegistry.circuitBreaker("otherName");
CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker("uniqueName", circuitBreakerConfig);
```

上面的代码，首先定义了一个 CircuitBreakerConfig 对象。在定义 CircuitBreakerConfig 对象时，设置了故障率阈值为 50% ，断路器保持打开时间为 2 秒。当断路器处于 half open 状态时，环形缓冲区大小为2，当对象处于关闭状态时，环形缓冲区大小也为2，然后根据创建出来的 CircuitBreakerConfig 对象创建一个 CircuitBreakerRegistry ，再根据 CircuitBreakerRegistry 创建两个断路器 CircuitBreaker 。

如果开发者不想使用 CircuitBreakerRegistry 来管理断路器，那么也可以直接创建一个 CircuitBreaker 对象，创建方式如下：

```
CircuitBreaker defaultCircuitBreaker = CircuitBreaker.ofDefaults("testName");
CircuitBreaker customCircuitBreaker = CircuitBreaker.of("testName", circuitBreakerConfig);
```

断路器使用案例

断路器使用了装饰者模式，开发者可以使用 CircuitBreaker.decorateCheckedSupplier(), CircuitBreaker.decorateCheckedRunnable() 或者 CircuitBreaker.decorateCheckedFunction() 来装饰 Supplier / Runnable / Function 或者 CheckedRunnable / CheckedFunction，然后使用 Try.of(…) 或者 Try.run(…) 来进行调用操作，也可以使用 map、flatMap、filter、recover 或者 andThen 进行链式调用，但是调用这些方法断路器必须处于 CLOSED 或者 HALF_OPEN 状态。例如下面一个例子，创建一个断路器出来，首先装饰了一个函数，这个函数返回一段字符串，然后使用 Try.of 去执行，执行完后再进入到 map 中去执行。如果第一个函数正常执行第二个函数才会执行，如果第一个函数执行失败，那么 map 函数将不会执行：

```
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");
CheckedFunction0<String> decoratedSupplier = CircuitBreaker
      .decorateCheckedSupplier(circuitBreaker, () -> "This can be any method which returns: 'Hello");
Try<String> result = Try.of(decoratedSupplier)
      .map(value -> value + " world'");
System.out.println(result.isSuccess());
System.out.println(result.get());
```

这里两个函数使用了相同的断路器，你也可以将不同断路器的函数连接起来，如下：

```
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");
CircuitBreaker anotherCircuitBreaker = CircuitBreaker.ofDefaults("anotherTestName");
CheckedFunction0<String> decoratedSupplier = CircuitBreaker
      .decorateCheckedSupplier(circuitBreaker, () -> "Hello");
CheckedFunction1<String, String> decoratedFunction = CircuitBreaker
      .decorateCheckedFunction(anotherCircuitBreaker, (input) -> input + " world");
Try<String> result = Try.of(decoratedSupplier)
      .mapTry(decoratedFunction::apply);
System.out.println(result.isSuccess());
System.out.println(result.get());
```

断路器打开

这里创建了两个 CircuitBreaker ，装饰了两个函数，第二次使用了 mapTry 方法来连接。前面给大家演示的几种情况，都是执行成功的，即断路器一直处于关闭的状态，接下来给大家再来演示一个断路器打开的例子，如下：

```
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
      .ringBufferSizeInClosedState(2)
      .waitDurationInOpenState(Duration.ofMillis(1000))
      .build();
CircuitBreaker circuitBreaker = CircuitBreaker.of("testName", circuitBreakerConfig);
circuitBreaker.onError(0, new RuntimeException());
System.out.println(circuitBreaker.getState());
circuitBreaker.onError(0, new RuntimeException());
System.out.println(circuitBreaker.getState());
Try<String> result = Try.of(CircuitBreaker.decorateCheckedSupplier(circuitBreaker, () -> "Hello"))
      .map(value -> value + " world");
System.out.println(result.isSuccess());
System.out.println(result.get());
```

这里手动模拟错误，首先设置了断路器关闭状态下的环形缓冲区大小为 2 ，即当有两条数据时就可以去统计故障率了，这里没有设置故障率，默认的故障率是 50% ，当第一次调用 onError 方法后，打印断路器当前状态，发现断路器还是处于关闭状态，并未打开，接下来再次调用 onError 方法，然后再去查看断路器状态，此时发现断路器已经打开了，因为满足了 50% 的故障率了。

断路器重置

断路器也支持重置，重置之后数据清空，恢复到初始状态，如下：

```
circuitBreaker.reset();
```

服务请求降级

既然是断路器，当然也支持服务降级，如下：

```
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");
CheckedFunction0<String> checkedSupplier = CircuitBreaker.decorateCheckedSupplier(circuitBreaker, () -> {
   throw new RuntimeException("BAM!");
});
Try<String> result = Try.of(checkedSupplier)
      .recover(throwable -> "Hello Recovery");
System.out.println(result.isSuccess());
System.out.println(result.get());
```

如果需要使用服务降级，可以使用 Try.recover() 链接，当 Try.of() 返回 Failure 时服务降级会被触发。

状态监听

状态监听可以获取到熔断器当前的运行数据，例如：

```
CircuitBreaker.Metrics metrics = circuitBreaker.getMetrics();
// 获取故障率
float failureRate = metrics.getFailureRate();
// 获取调用失败次数
int failedCalls = metrics.getNumberOfFailedCalls();
```

限流

RateLimiter 和我们前面提到的断路器实际上非常类似，它也有一个基于内存的 RateLimiterRegistry 和 RateLimiterConfig 可以配置，我们可以配置如下一些参数：

- 限流之后的冷却时间
- 阈值刷新时间
- 阈值刷新频次

使用限流工具，我们当然需要首先引入限流工具的依赖，如下：

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

基本用法

例如，想限制某个请求的频率为 2QPS（每秒处理两个请求），为什么给一个这样的频率呢？主要是为了大家一会儿测试方便，代码如下：

```
RateLimiterConfig config = RateLimiterConfig.custom()
        .limitRefreshPeriod(Duration.ofMillis(1000))
        .limitForPeriod(2)
        .timeoutDuration(Duration.ofMillis(1000))
        .build();
RateLimiterRegistry rateLimiterRegistry = RateLimiterRegistry.of(config);
RateLimiter rateLimiterWithDefaultConfig = rateLimiterRegistry.rateLimiter("backend");
RateLimiter rateLimiterWithCustomConfig = rateLimiterRegistry.rateLimiter("backend#2", config);
RateLimiter rateLimiter = RateLimiter.of("NASDAQ :-)", config);
```

和前面的一样，我们也可以使用 RateLimiterRegistry 来统一管理 RateLimiter ，也可以通过 RateLimiter.of 方法来直接创建一个 RateLimiter。创建好了，就可以直接使用了，代码如下：

```
CheckedRunnable restrictedCall = RateLimiter
        .decorateCheckedRunnable(rateLimiter,()->{
            System.out.println(new Date());
        });
Try.run(restrictedCall)
        .andThenTry(restrictedCall)
        .andThenTry(restrictedCall)
        .andThenTry(restrictedCall)
        .onFailure(throwable -> System.out.println(throwable.getMessage()));
```

执行结果如下：

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java ...
Fri Apr 05 19:59:52 CST 2019
Fri Apr 05 19:59:52 CST 2019
Fri Apr 05 19:59:53 CST 2019
Fri Apr 05 19:59:53 CST 2019

Process finished with exit code 0
```

可以看到，因为限流，一次只执行了两个方法，另外两个方法是 1s 后执行的。限流参数也可以随时修改，修改之后，本次限流周期内不起作用，下次限流时会生效，修改方式如下：

```
rateLimiter.changeLimitForPeriod(100);
rateLimiter.changeTimeoutDuration(Duration.ofMillis(100));
```

事件监听

限流中，我们也可以获取所有允许和拒绝执行的事件信息，获取方式如下：

```
rateLimiter.getEventPublisher()
        .onSuccess(event -> {
            System.out.println(new Date()+">>>"+event.getEventType()+">>>"+event.getCreationTime());
        })
        .onFailure(event -> {
            System.out.println(new Date()+">>>"+event.getEventType()+">>>"+event.getCreationTime());
        });
```

请求隔离

不同于 Hystrix、Resilience4j 中提供的请求隔离，主要是基于信号量的请求隔离，不包含基于线程的请求隔离，具体用法和前面两个类似，不过在使用之前，需要先添加请求隔离相关的依赖，如下：

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

可以基于默认配置创建一个 BulkheadRegistry：

```
BulkheadRegistry bulkheadRegistry = BulkheadRegistry.ofDefaults();
```

也可以自定义最大并行数和进入饱和态 Bulkhead 时线程的最大阻塞时间，如下：

```
BulkheadConfig config = BulkheadConfig.custom()
        .maxConcurrentCalls(150)
        .maxWaitTime(100)
        .build();
BulkheadRegistry registry = BulkheadRegistry.of(config);
Bulkhead bulkhead1 = registry.bulkhead("foo");
BulkheadConfig custom = BulkheadConfig.custom()
        .maxWaitTime(0)
        .build();
Bulkhead bulkhead2 = registry.bulkhead("bar", custom);
```

这里实例的创建方式和前面两个类似，我就不一一解释了。另外，开发者如果不想通过 BulkheadRegistry 来管理 Bulkhead ，也可以直接创建 Bulkhead 的实例，如下：

```
Bulkhead bulkhead1 = Bulkhead.ofDefaults("foo");
Bulkhead bulkhead2 = Bulkhead.of(
        "bar",
        BulkheadConfig.custom()
                .maxConcurrentCalls(50)
                .build()
);
```

创建好了之后，使用步骤也基本上和断路器一致，举例如下：

```
BulkheadConfig config = BulkheadConfig.custom()
        .maxConcurrentCalls(1)
        .maxWaitTime(100)
        .build();
Bulkhead bulkhead = Bulkhead.of("testName", config);
CheckedFunction0<String> decoratedSupplier = Bulkhead.decorateCheckedSupplier(bulkhead, () -> "This can be any method which returns: 'Hello");
Try<String> result = Try.of(decoratedSupplier)
        .map(value -> value + " world'");
System.out.println(result.isSuccess());
System.out.println(result.get());
```

请求重试

请求失败重试也是一个常见功能，Resilience4j 中对此也提供了支持，首先引入重试相关依赖：

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

然后通过如下代码我们创建一个重试的实例：

```
RetryConfig config = RetryConfig.custom()
    .maxAttempts(3)
    .waitDuration(Duration.ofMillis(500))
    .build();
Retry retry = Retry.of("id", config);
```

在上面的配置中，我们配置了重试次数为3，重试间隔 500ms，有了 Retry 实例之后，就可以直接使用了：

```
CheckedFunction0<String> retryableSupplier = Retry.decorateCheckedSupplier(retry, ()->{
    System.out.println(new Date());
    return "hello retry";
});
Try<String> result = Try.of(retryableSupplier).recover((throwable) -> "Hello world from recovery function");
System.out.println(result.isSuccess());
System.out.println(result.get());
```

执行过程和前面的也基本类似，如果执行过程中抛出异常了，那么就会触发重试机制。

缓存

Resilience4j 中也提供了基于 JCache 的方法缓存，考虑到实际开发中以 Redis 缓存为主，这里我就不去介绍这里自带的缓存了，有兴趣的读者可以参考这里。

限时

Resilience4j 中的限时器是要结合 Future 一起来使用，开发者需要提前配置过期时间，在过期时间内要是没有获取到value，那么 Future 将会被取消，使用步骤如下：
限时首先也要加入依赖，如下：

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

示例代码如下：

```
TimeLimiterConfig config = TimeLimiterConfig.custom()
        .timeoutDuration(Duration.ofSeconds(60))
        .cancelRunningFuture(true)
        .build();
TimeLimiter timeLimiter = TimeLimiter.of(config);
ExecutorService executorService = Executors.newSingleThreadExecutor();
Supplier<Future<Integer>> futureSupplier = () -> executorService.submit(backendService::doSomething);
Callable restrictedCall = TimeLimiter
        .decorateFutureSupplier(timeLimiter, futureSupplier);
Try.of(restrictedCall::call)
        .onFailure(throwable -> System.out.println(throwable.getMessage()));
```

这里首先创建了一个 TimeLimiter，然后将任务放到线程池中，获取到一个 Supplier<Future> 对象，然后使用限时器包装该对象，当调用超时， onFailure 方法就会被触发。

也可以将限时器和断路器结合使用，当调用超时次数过多，直接熔断，如下：

```
Callable restrictedCall = TimeLimiter
    .decorateFutureSupplier(timeLimiter, futureSupplier);
Callable chainedCallable = CircuitBreaker.decorateCallable(circuitBreaker, restrictedCall);
Try.of(chainedCallable::call)
    .onFailure(throwable -> LOG.info("We might have timed out or the circuit breaker has opened."));
```

小结

本文首先向大家介绍了传统的容错方案 Hystrix 的一些大致功能，这个读者作为了解即可；然后向读者介绍了 Resilience4j 的一些基本功能，这些基本功能涵盖了请求熔断、限流、限时、缓存、隔离以及重试，这里我们只是介绍了 Resilience4j 的一些基本用法。上文中所有的案例都是在一个普通的 JavaSE 项目中写的，这里并未涉及到微服务，下篇文章我将和大家分享，这六个功能如何在微服务中使用，进而实现微服务系统的高可用。

本文作者：纯洁的微笑、江南一点雨

15 Feign 中的继承、日志与数据压缩

17 Resilience4j 在微服务中的应用