在微服务架构中，我们将系统拆分成了一个个的服务单元，各单元应用间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会出现因等待出现故障的依赖方响应而形成任务积压，线程资源无法释放，最终导致自身服务的瘫痪，进一步甚至出现故障的蔓延最终导致整个系统的瘫痪。如果这样的架构存在如此严重的隐患，那么相较传统架构就更加的不稳定。为了解决这样的问题，因此产生了断路器等一系列的服务保护机制。

针对上述问题，在Spring Cloud Hystrix中实现了线程隔离、断路器等一系列的服务保护功能。它也是基于Netflix的开源框架 Hystrix实现的，该框架目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix具备了服务降级、服务熔断、线程隔离、请求缓存、请求合并以及服务监控等强大功能。
接下来，我们就从一个简单示例开始对Spring Cloud Hystrix的学习与使用。
实战练习
在开始使用Spring Cloud Hystrix实现断路器之前，我们先拿之前实现的一些内容作为基础，其中包括：

- eureka-server工程：服务注册中心，端口：1001
- eureka-client工程：服务提供者，两个实例启动端口分别为2001

下面我们可以复制一下之前实现的一个服务消费者：eureka-consumer-ribbon，命名为eureka-consumer-ribbon-hystrix。下面我们开始对其进行改在：
第一步：pom.xml的dependencies节点中引入spring-cloud-starter-hystrix依赖：

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

第二步：在应用主类中使用@EnableCircuitBreaker或@EnableHystrix注解开启Hystrix的使用：

```
@EnableCircuitBreaker
@EnableDiscoveryClient
@SpringBootApplication
public class EurekaConsumerRibbonHystrixApplication {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        new
SpringApplicationBuilder(EurekaConsumerRibbonHystrixApplication.class).web(true).run(args);
    }
}
```

注意：这里我们还可以使用Spring Cloud应用中的@SpringCloudApplication注解来修饰应用主类，该注解的具体定义如下所示。我们可以看到该注解中包含了上我们所引用的三个注解，这也意味着一个Spring Cloud标准应用应包含服务发现以及断路器。

```
@Target({ElementType.TYPE})
```

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public @interface SpringCloudApplication {
}
```

第三步：改造服务消费方式，新增ConsumerService类，在为具体执行逻辑的函数上增加@HystrixCommand注解来指定服务降级方法，如下所示：

```
@Service
public class ConsumerService {
    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "fallback")
    public String consumer() {
        return restTemplate.getForObject("http://eureka-client/dc", String.class);
    }

    public String fallback() {
        return "发生异常";
    }
}
```

然后将在Controller中的逻辑迁移出去：

```
@RestController
public class DcController {
    @Autowired
    ConsumerService consumerService;

    @GetMapping("/consumer")
    public String dc() {
        return consumerService.consumer();
    }
}
```

第四步：修改application.properties

```
spring.application.name=eureka-consumer-ribbon-hystrix
server.port=2301
eureka.client.serviceUrl.defaultZone=http://localhost:1001/eureka/
```

下面我们来验证一下上面Hystrix带来的一些基础功能。我们先把涉及的服务都启动起来，然后访问localhost:2301/consumer，此时可以获取正常的返回，比如：Services: [eureka-consumer-ribbon-hystrix, eureka-client]。

为了触发服务降级逻辑，我们将服务提供者eureka-client的逻辑加一些延迟，代码如下：

```
@GetMapping("/dc")
public String dc() throws InterruptedException {
    Thread.sleep(5000L);
```

```
    String services = "Services: " + discoveryClient.getServices() + "---端口号：" + port;
    System.out.println(services);
    return services;
}
```

重启eureka-client之后，再尝试访问localhost:2301/consumer，此时我们将获得的返回结果为：fallback。我们从eureka-client的控制台中，可以看到服务提供方输出了原本要返回的结果，但是由于返回前延迟了5秒，而服务消费方触发了服务请求超时异常，服务消费者就通过HystrixCommand注解中指定的降级逻辑进行执行，因此该请求的结果返回了fallback。这样的机制，对自身服务起到了基础的保护，同时还为异常情况提供了自动的服务降级切换机制。

具体工程说明如下：

- eureka的服务注册中心：eureka-server
- eureka的服务提供方：eureka-client
- eureka的服务消费者：eureka-consumer-ribbon-hystrix