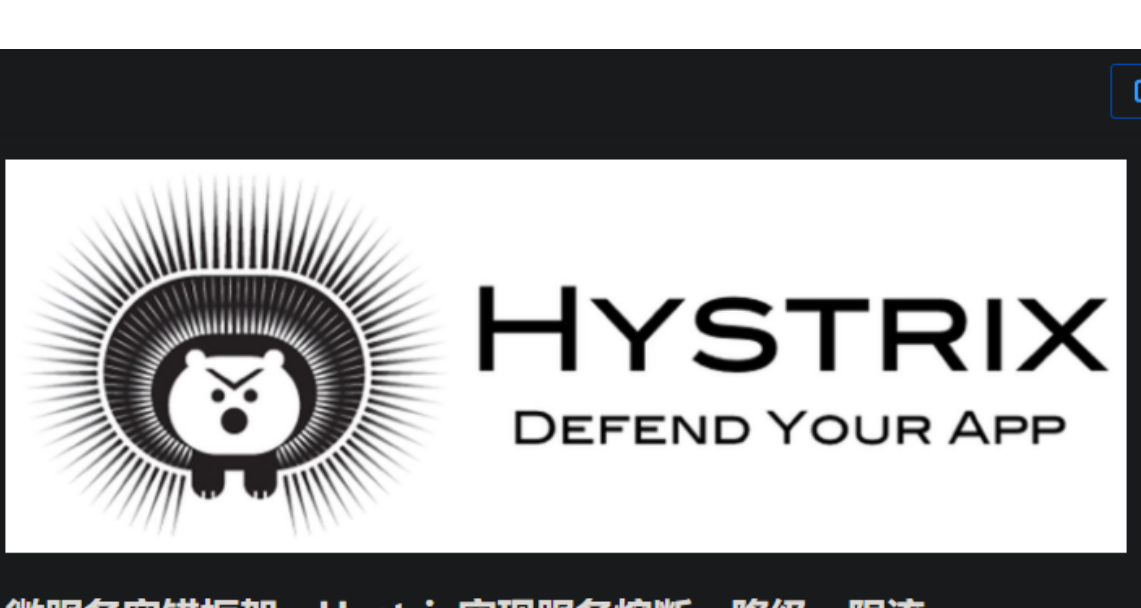


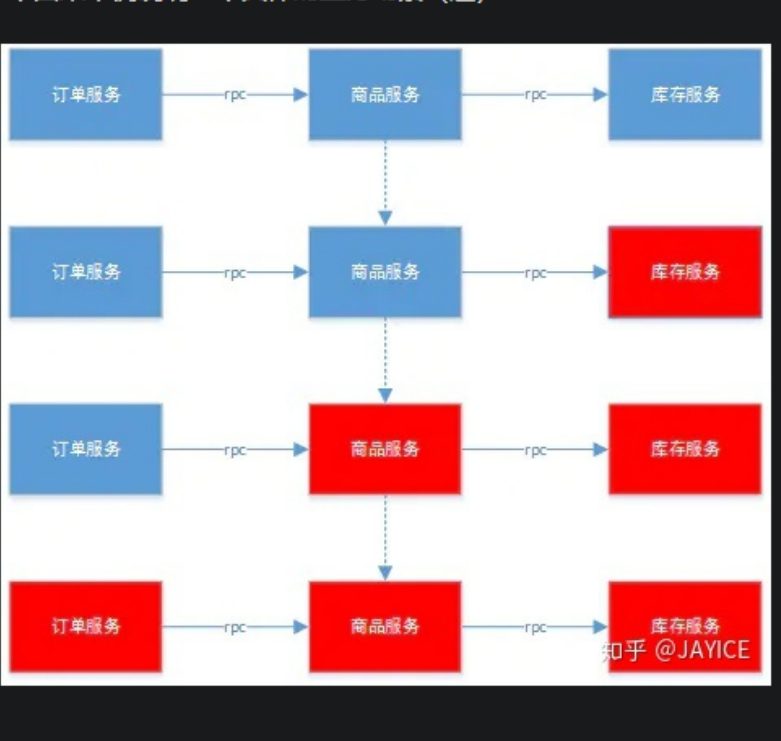
微服务容错框架：Hystrix实现服务熔断、降级、限流

JAYICE
喜欢追剧和闲逛，伪宅，会写点代码
＋ 关注他

13 人赞同了该文章

业务背景

在微服务架构体系下，服务间不可避免地会发生依赖关系，一般来说会通过REST Api来进行通信，这里先盗一个图来举例说明一个具体的业务场景（逃）：



比如一个商城系统的微服务化结构，订单、商品、库存这三个服务是紧密依赖的，在理想情况下，什么问题都不发生当然是最好的。但服务运行期间难免会出现各种问题，如网络阻塞，延迟过高（比如因为内存泄露导致的Full GC次数飙高），甚至服务直接挂掉（比如流量激增把服务打挂了）等情况都是很有可能发生的。倘若库存服务挂掉了，那对于所有对库存服务有依赖关系的服务都会受到很大影响，最终甚至会扩散到整个微服务体系，这种就称之为雪崩效应。

因此，在某一个服务发生故障时，我们要及时对该服务的故障进行隔离，不能让其扩散到整个微服务体系中。因为，为了搭建一个稳定且可靠的微服务系统，我们就需要给系统加上自我保护，出现故障自动隔离的能力。而**Hystrix**就能做到这一点

什么是Hystrix

Hystrix是Netflix开源的一款分布式容错框架，Netflix旗下还有Eureka,Zuul等优秀的分布式开源项目，Spring Cloud也提供了对Netflix中部门项目的支持，成为了SpringCloud下的一些子项目
。

Hystrix的功能：

- 阻止故障的连锁反应，实现熔断
- 快速失败，实现优雅降级
- 提供实时的监控和告警

Hystrix简单实现

```
rIdCommand extends HystrixCommand<Integer> {
tic Logger logger = LoggerFactory.getLogger(QueryUserIdCommand.class);
ceProvider userServiceProvider;

geCommand(UserServiceProvider userServiceProvider) {
withGroupKey(HystrixCommandGroupKey.Factory.asKey("userService"))
ommandKey(HystrixCommandKey.Factory.asKey("queryByUserId"))
ommandPropertiesDefaults(HystrixCommandProperties.Setter()
   .withCircuitBreakerRequestVolumeThreshold(10)//至少有10个请求，熔断器才会开始进行错误率计
   .withCircuitBreakerSleepWindowInMilliseconds(5000)//熔断器中断请求，5秒后会进入一个半打
   .withCircuitBreakerErrorThresholdPercentage(50)//错误率达到50%就开启熔断保护
   .withExecutionTimeoutEnabled(true))
hreadPoolPropertiesDefaults(HystrixThreadPoolProperties
   .Setter().withCoreSize(10)));
iceProvider = userServiceProvider;



 run() {
rviceProvider.queryByUserId();



 getFallback() {
```

发起请求

```
Integer res = new QueryUserIdCommand(userServiceProvider).execute();
log.info("result:{}", res);
```

访问接口，正常情况下，会返回正确的信息，当把UserServiceProvider所依赖的服务的接口改为直接抛出一个异常，就会发现总是返回-1了。这样就做到了对错误进行隔离。

Hystrix容错

那接下来从三个角度来聊一下Hystrix提供的容错功能，分别是**资源隔离**，**熔断**和**降级**

资源隔离

我们之前也讨论过，微服务体系中，各个服务之间都通过REST Api来进行调用，从而建立依赖关系。倘若该服务调用和业务代码在同一个线程会中执行的话，如果 api在调用的时候出现了网络堵塞等情况，那么不仅会对业务代码进行阻塞，也会对后面的请求造成阻塞，因为线程池的线程数是额定的。所以，Hystrix也提供了资源隔离的机制，主要是线程隔离和信号量隔离

资源隔离-线程池

刚才我们简单地应用了一下Hystrix，很明显看到我们得先实现一个自己的HystrixCommand，然后把服务调用的操作封装在这个类里面。而实际上，线程级别的资源隔离就是在HystrixCommand中实现。Hystrix会给每一个Command分配一个单独的线程池，这样在进行单个服务调用的时候，就可以在独立的线程池里面进行，而不会对其他线程池造成影响。

Hystrix通过一个ConcurrentHashMap来维护这些线程池：

```
final static ConcurrentHashMap<String, HystrixThreadPool> threadPools = new Concurrent

//其他代码

if (!threadPools.containsKey(key)) {
    threadPools.put(key, new HystrixThreadPoolDefault(threadPoolKey, propertiesBuilder
}
```

线程隔离的优点：

- 保护当前应用免受来自其他服务故障的影响，最终提高整个微服务体系的稳定性

- 可以对一个Command里面的线程调用参数进行单独设置，而不影响其他Command，如果使用Spring Cloud Hystrix的话，那就是：

```
@HystrixCommand(groupKey="UserGroup", commandKey = "GetUserByIdCommand",
                commandProperties = {
                    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilli
                    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", v
                    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",
                },
                threadPoolProperties = {
                        @HystrixProperty(name = "coreSize", value = "30"),
                        @HystrixProperty(name = "maxQueueSize", value = "101"),
                        @HystrixProperty(name = "keepAliveTimeMinutes", value = "2"),
                        @HystrixProperty(name = "queueSizeRejectionThreshold", value =
                        @HystrixProperty(name = "metrics.rollingStats.numBuckets", val
                        @HystrixProperty(name = "metrics.rollingStats.timeInMillisecon
        })
```

- 线程池如果处于饱和态，还往其中添加请求的话，会直接触发fallback，所以还有限流的作用

而为每一个Command配置一个线程池的缺点就是增加开销，线程一多起来，就增加了调度，上下文切换等额外的开销。但通常情况下，引入线程池的开销是很小的，相对于它带来的好处，大多数情况还是非常乐于进行这样的交换，但如果是一些耗时非常低的请求，比如直接从缓存中获取数据然后返回，引入线程池带来的影响可能会显得比较大，这种时候可以用另一种方法，就是**信号量隔离**

资源隔离-信号量隔离

信号量隔离本质上并没有做到资源隔离，只是起到了限流的作用，从而防止出现线程大面积阻塞，功能和JUC下的那个Semaphare类差不多，但区别就是，没有执行条件的线程在这里会直接调用fallback，而不是阻塞。启用方法也很简单，在实现Hystrix的时候声明一下即可：

```
public class QueryUserIdCommand extends HystrixCommand<Integer> {
    private final static Logger logger = LoggerFactory.getLogger(QueryUserIdCommand.cl
    private UserServiceProvider userServiceProvider;
        public QueryUserAgeCommand(UserServiceProvider userServiceProvider) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("userService"))
                .andCommandKey(HystrixCommandKey.Factory.asKey("queryByUserId"))
                    .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                        .withCircuitBreakerRequestVolumeThreshold(10)//至少有10个请求，
                            .withCircuitBreakerSleepWindowInMilliseconds(5000)//熔断器
                        .withCircuitBreakerErrorThresholdPercentage(50)//错误率达到50%
                            .withExecutionIsolationStrategy(HystrixCommandProperties.E
                        .withExecutionTimeoutEnabled(true))
                    .andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties
                        .Setter().withCoreSize(10)));
        this.userServiceProvider = userServiceProvider;
        }

        @Override
        protected Integer run() {
           return userServiceProvider.queryByUserId();
        }

        @Override
        protected Integer getFallback() {
           return -1;
        }
}
```

总结一下，资源隔离就大概是这两种了，大多数情况用的还是线程隔离居多，毕竟它才是真正意义上的资源隔离，而信号量隔离只是起到一个限流的作用而已

熔断

一个服务出现故障之后，防止这个故障蔓延到所有依赖它的服务，这就是熔断。一开始的时候我们也已经写过代码来简单实现熔断了，所以直接介绍几个重要的熔断相关的参数：

```
- circuitBreaker.enabled：是否启用熔断器，默认true

- circuitBreaker.forceOpen：强制打开熔断器，默认false。
- circuitBreaker.forceClosed：强制关闭熔断器，默认false。

- circuitBreaker.errorThresholdPercentage：错误率，默认50%。在一段时间内，服务调用超时或者失

- circuitBreaker.requestVolumeThreshold：默认20。意思为在一段时间内要有20个及以上的请求才会

- circuitBreaker.sleepWindowInMilliseconds：半开状态试探睡眠时间，默认为5000ms。也就是熔断
```

来写代码测试一下其中几个参数：

```
@HystrixCommand(groupKey = "productStockOpLog", commandKey = "addProductStockOpLog", f
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.thread.timeoutInMillisec
                           @HystrixProperty(name = "circuitBreaker.requestVolumeThresh
                @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", val
        }
)
```

我先设置了超时时间为400ms。然后在服务提供接口方法改成这样：

```
@Service
public class ServiceProviderImpl implements ServiceProvider {
    private int c = 0;

    @Override
    public Integer service() {

        if (c < 10) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
            }
        }
        return c;
    }
}
```

实现也很简单，前十次调用都睡500ms，也就是说，前十次调用肯定是超时的。

而在客户端，实现如下：

```
@Test
public void test() throws InterruptedException {
    for (int i = 1; i < 15; i++) {
        HystrixCommand<Integer> command = new QueryByUserIdCommand(userServiceProvider
        Integer r = command.execute();
        String res = r == -1 ? "fallback" : "success";
        System.out.println("第"i+"次调用，结果为"+ method);
    }
    //等待5.5s，让熔断器进入半打开状态
    Thread.sleep(5500);
    for (int i = 15; i < 20; i++) {
        HystrixCommand<Integer> command = new UserByOrderIdCommand(userServiceProvider
        Integer r = command.execute();
        String method = r == -1 ? "fallback" : "success";
        System.out.println("第"i+"次调用，结果为"+ method);
    }
}
```

查看执行结果：

```
第1次调用，结果为fallback
第2次调用，结果为fallback
第3次调用，结果为fallback
第4次调用，结果为fallback
第5次调用，结果为fallback
第6次调用，结果为fallback
第7次调用，结果为fallback
第8次调用，结果为fallback
第9次调用，结果为fallback
第10次调用，结果为fallback
第11次调用，结果为fallback
第12次调用，结果为fallback
第13次调用，结果为fallback
第14次调用，结果为fallback
第15次调用，结果为success
第16次调用，结果为success
第17次调用，结果为success
第18次调用，结果为success
第19次调用，结果为success
第20次调用，结果为success
```

分析可得，1-10次触发fallback的原因是超时

而11-14次的fallback，是因为满足了“请求次数到达10次”和“失败率超过了50%”而触发了熔断器，所以直接快速失败。

待沉睡了5.5秒后，熔断器进入了半打开状态，此时熔断器放开第15个请求去调用服务，发现成功了，此时熔断器关闭，后续都成功了。

降级

熔断和降级其实它们的原理都相似，都是服务调用失败后的进行一些快速失败措施。但它们的出发点不一样，熔断是为了防止异常不扩散，保证系统的稳定性

而降级则是人为操作。在一些流量顶峰期，为了保证某些热门接口的正常运作，有时候会牺牲一些非核心接口，把资源全都让给热点接口，这就是服务降级。每年12306抢票的时候，大家都集中抢购那几个热门车次的车票，而如果此时有其他用户去查询几天后的非热门车票，有可能会查不出来。这就是降级的表现，在秒杀期间，其他不参与秒杀的接口停止服务，把资源都让给参与秒杀的接口。

所以降级的操作其实也很简单，和前面的熔断一样，编写好调用失败的补救逻辑，然后对其他的服务直接停止运行，这样这些接口就无法正常调用，但又不至于直接报错，只是服务水平下降了。

```
@FeignClient(value = "microservicecloud-test", fallbackFactory = UserServiceFallbackFa
public interface UserService
{
    public boolean add(User user);

    public User queryUserByUserId(Long id);
}
```

这里有一个Service接口，可以看到在注解的地方给它配置了一个fallbackFactory

```
@Component
public class UserServiceFallbackFactory implements FallbackFactory<UserService>
{
    @Override
    public UserService create(Throwable throwable)
    {
        return new UserService() {
            @Override
            public boolean add(User user)
            {
                return false;
            }

            @Override
            public User queryUserByUserId(Long id)
            {
                User user = new User();
                return user;
            }
        };
    }
}
```

这样就大概完成了一个降级配置了。（其实也可以当熔断来用，是吧？没有任何问题）

Hystrix执行调用的几个方法

Hystrix一共有四种调用方法：

execute

还记得一开始的时候，写了一个简单使用Hystrix的demo，其中调用的代码是：

```
Integer res = new QueryUserIdCommand(userServiceProvider).execute();
log.info("result:{}", res);
```

调用的是execute方法，它将以同步的方式调用run方法。

那问题来了，如果我系统它以异步的方式执行怎么办？我还得自己封装Callable吗？哈，其实Hystrix已经考虑到这一点了，那就是使用queue方法

queue

这个调用也挺简单的，我直接贴代码

```
Future<Integer> future = new QueryUserIdCommand(userServiceProvider).queue();
return future.get();
```

虽然Future.get()方法依然是阻塞的，但是服务调用总是不阻塞了嘛~

observe

接下里又来了一个新的需求了，我需要发多个请求，并且每请求得到一个结果，就要做一下相关的处理 。这个无论用queue还是execute都不太方便。那此时我们可以选择使用observe。如果使用observe的话，那么就不再是使用HystrixCommand了，而是HystrixObservableCommand

```
public class UserServiceObserveCommand extends HystrixObservableCommand<String>{

    private RestTemplate restTemplate;

    protected HelloServiceObserveCommand(String commandGroupKey, RestTemplate restTemp
        super(HystrixCommandGroupKey.Factory.asKey(commandGroupKey));
        this.restTemplate = restTemplate;
    }

    @Override
    protected Observable<String> construct() {
        return Observable.create(new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<? super String> subscriber) {
                try {
                    if (!subscriber.isUnsubscribed()){
                        System.out.println("服务调用开始！");

                        String result = restTemplate.getForEntity("http://your-url1",
                        //触发监听
                        subscriber.onNext(result);

                        String result1 = restTemplate.getForEntity("http://your-url2",
                        //触发监听
                        subscriber.onNext(result1);
                        subscriber.onCompleted();
                    }
                } catch (Exception e) {
                    subscriber.onError(e);
                }
            }
        });
    }
   //降级Fallback
    @Override
    protected Observable<String> resumeWithFallback() {
        return Observable.create(new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<? super String> subscriber) {
                try {
                    if (!subscriber.isUnsubscribed()) {
                        subscriber.onNext("fallback");
                        subscriber.onCompleted();
                    }
                } catch (Exception e) {
                    subscriber.onError(e);
                }
            }
        });
    }
}
```

可以看到，每调用完一个方法，就调用一次subscriber.onNext来触发监听。那具体的监听实现怎么写呢？

```
    @Test
    public void test() throws ExecutionException, InterruptedException {
        UserServiceObserveCommand command = new HelloServiceObserveCommand("user",rest
        //observe调用
        Observable<String> observable = command.observe();
       //注册监听
        observable.subscribe(new Observer<String>() {
                //onCompleted监听
            @Override
            public void onCompleted() {
                System.out.println("调用完毕！");
            }

            @Override
            public void onError(Throwable t) {
                t.printStackTrace();
            }

            //onNext监听
            @Override
            public void onNext(String s) {
                System.out.println("调用完了一个服务！");
            }
        });
    }
```

通过以上代码应该大概能知道observe的作用了

toObservable

observe叫做热执行，而toObservable叫做冷执行。有啥区别呢？

observe热执行，无需等客户端注册监听，就可以直接执行call方法里面的内容。相当于：你监听

不监听是你的事情，反正我该调用的都调用了，要是你不想监听我也没办法（逃）

toObservable冷执行，需要等客户端注册监听，才可以执行call方法里面的内容。相当于：你要是不监听的话，那我就不会执行call方法，直到你注册监听，我才开始执行调用

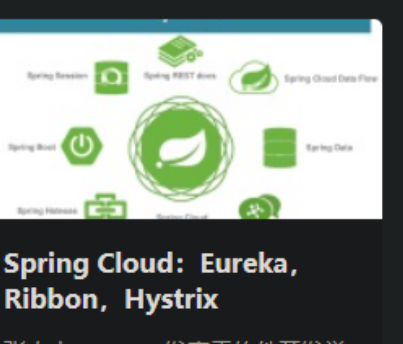
Hystrix作为熔断降级限流的主要功能和用法就是这些。其实Hystrix还能起到监控的功能：Hystrix-DashBorad，这个以后有机会再聊~

编辑于 2020-06-26

微服务架构 Spring Cloud Hystrix

推荐阅读

19张图带你梳理SpringCloud体系中的重要知识点！
小知 发表于Java知...



Spring Cloud：Eureka，Ribbon，Hystrix
张人大 发表于软件开发学...

全网最详细的一篇SpringCloud总结
老炮说Ja... 发表于炮哥和他的...

国内比较火Spring Cloud微服务开源项目
1、Cloud-Platform 简介Cloud-Platform是国内首个基于Spring Cloud微服务化开发平台，具有统一授权、认证后台管理系统，其中包含具备用户管理、资源权限管理、网关API 管理等多个模块，支...
大数据技术天涯

2 条评论 ⇄ 切换为时间排序

写下你的评论...

Garone 2020-06-27
看不懂惹[可怜]
赞

JAYICE (作者) 回复 Garone 2020-06-27
[可怜]大佬说笑了惹
赞

Captured by FireShot Pro: 17 三月 2021, 14:26:32
https://getfireshot.com