Hystrix

学习目标

1. 什么是熔断 , 什么是降级
2. 熔断与降级两者中的关系
3. 服务雪崩效应的原因与解决思路
4. Hystrix
    1. Hystrix的使用
    2. Hystrix是什么
    3. fegin集成Hystrix
5. 熔断与降级的异常警报处理
6. 熔断隔离策略
7. 超时时间隔离
8. 断路器监控仪表盘

1. 熔断与降级

1.1.1 熔断

在了解熔断前 , 我们需要了解服务器雪崩的这个概念 , 当新的请求 进入服务器而服务器无法对请求作出相应的时候 ,这个时候就容易引发服务器雪崩的问题 , 此时我们的熔断就可以排上用场, 熔断的作用就是 , 当请求的访问达到了服务器承载极限的时候 ,这个时候服务器就会采取 , 熔断的保护措施 , 顾名思义当服务器已经无法加载请求的时候 , 服务器就会自动将这些请求当做访问失败来处理

1.1.2 降级

当并发量达到极致的时候 , 有时候我们可以忽视掉次要的业务, 来保证主要访问业务的正常执行, 这就是我们所谓的降级 , 通常情况下, 降级采用的是返回一个空的数据 /集合, 这种数据也被成为兜底数据.

2.熔断与降级的关系

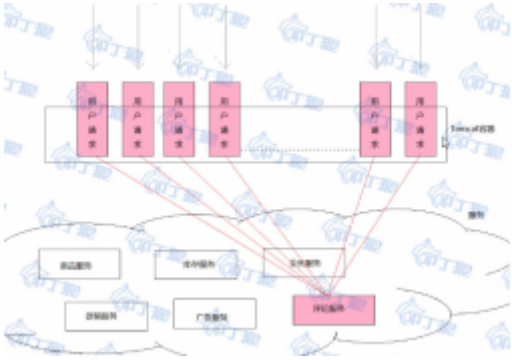
相同点 : 都是为了保证核心服务的正常运行

不同点 : 熔断是被动触发的 , 降级的主动去做的

3.服务雪崩效应的原因与解决思路

由于微服务相互之间的关系比较复杂 , 当一个请求访问的时候, 可能会因为其中一个服务的瘫痪导致请求的拥堵, 也就是意味着 , 请求积压在了服务器之间 , 因为这个微小的服务 , 导致所有的服务都不可用, 或者响应很慢 , 这就是服务器雪崩

图解:



4.Hystrix

Hystrix的存在能很大程度上解决由于请求的堆积所造成的服务器雪崩

4.1.1 什么是Hystrix

在分布式环境中，许多服务依赖项中的一些必然会失败。Hystrix是一个库，通过添加延迟容忍和容错逻辑，帮助你控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点、停止级联失败和提供回退选项来实现这一点，所有这些都可以提高系统的整体弹性。

4.1.2 Hystrix的作用

Hystrix被设计的目标是：

1. 对通过第三方客户端库访问的依赖项（通常是通过网络）的延迟和故障进行保护和控制。
2. 在复杂的分布式系统中阻止级联故障。
3. 快速失败，快速恢复。
4. 回退，尽可能优雅地降级。
5. 启用近实时监控、警报和操作控制。

复杂分布式体系结构中的应用程序有许多依赖项，每个依赖项在某些时候都不可避免地会失败。如果主机应用程序没有与这些外部故障隔离，那么它有可能被他们拖垮。

例如，对于一个依赖于30个服务的应用程序，每个服务都有99.99%的正常运行时间，你可以期望如下：

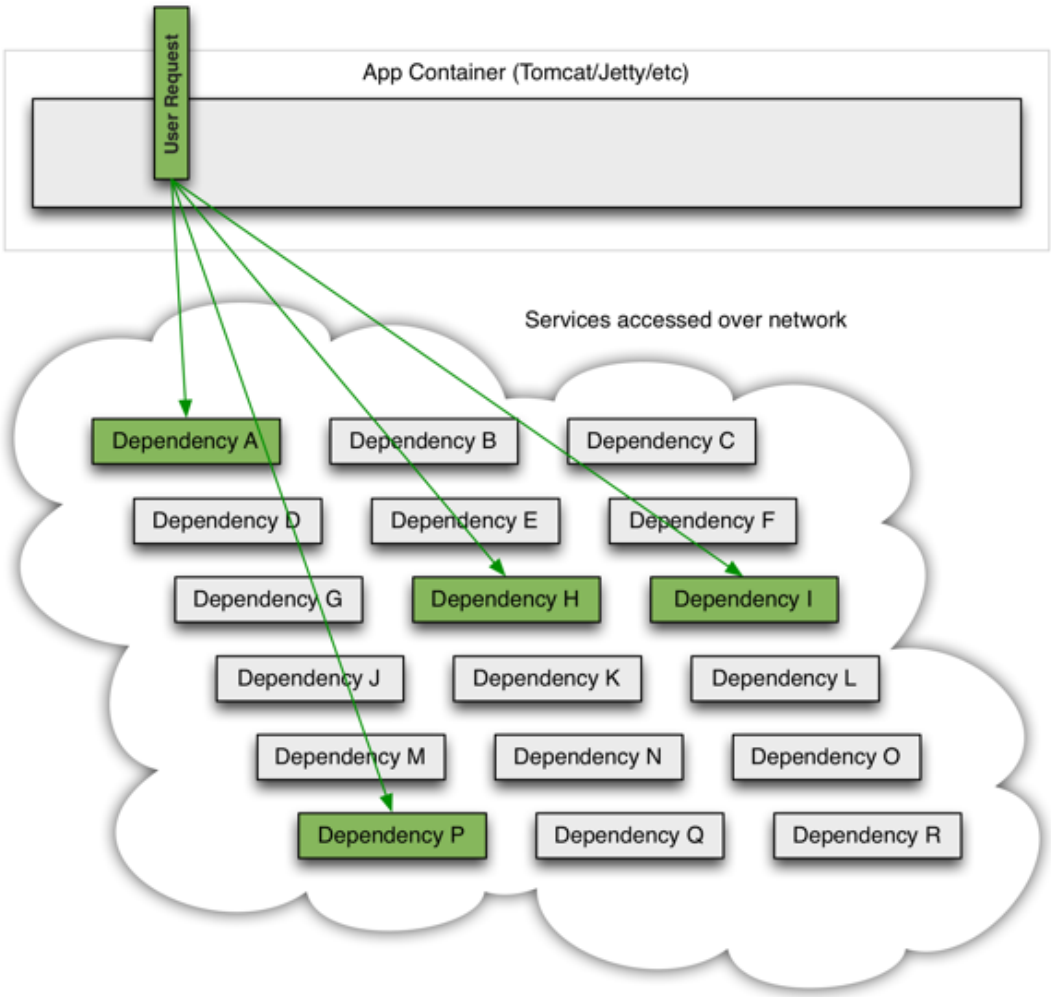
99.9930 = 99.7% 可用

也就是说一亿个请求的0.03% = 3000000 会失败

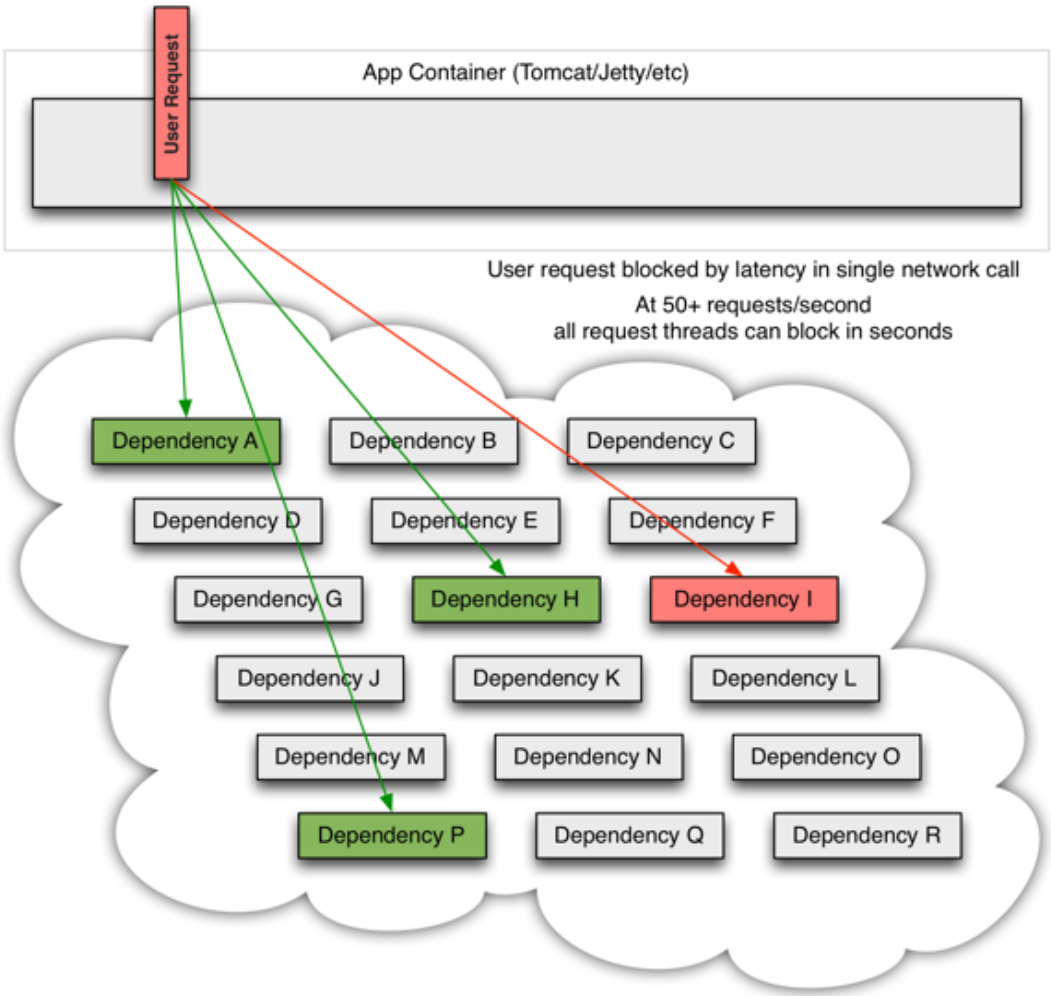
如果一切正常，那么每个月有2个小时服务是不可用的

现实通常是更糟糕

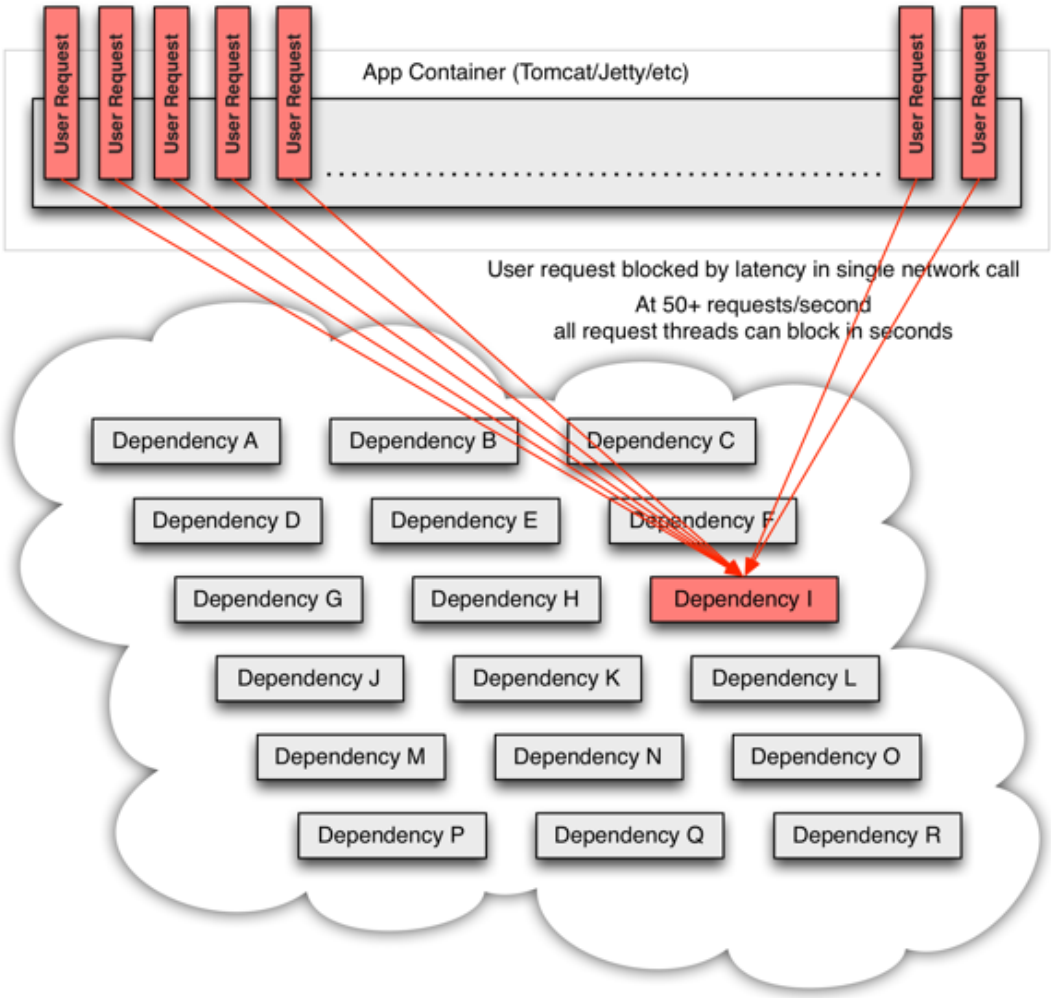
当一切正常时，请求看起来是这样的：



当其中一个系统有延迟或者挂掉之后，它可能阻塞整个用户的请求



在高并发/高流量的情况下，一个后端依赖项的延迟可能导致所有服务器上的所有资源在数秒内饱和（PS：意味着后续再有请求将无法立即提供服务）



4.1.3 Hystrix的设计原则

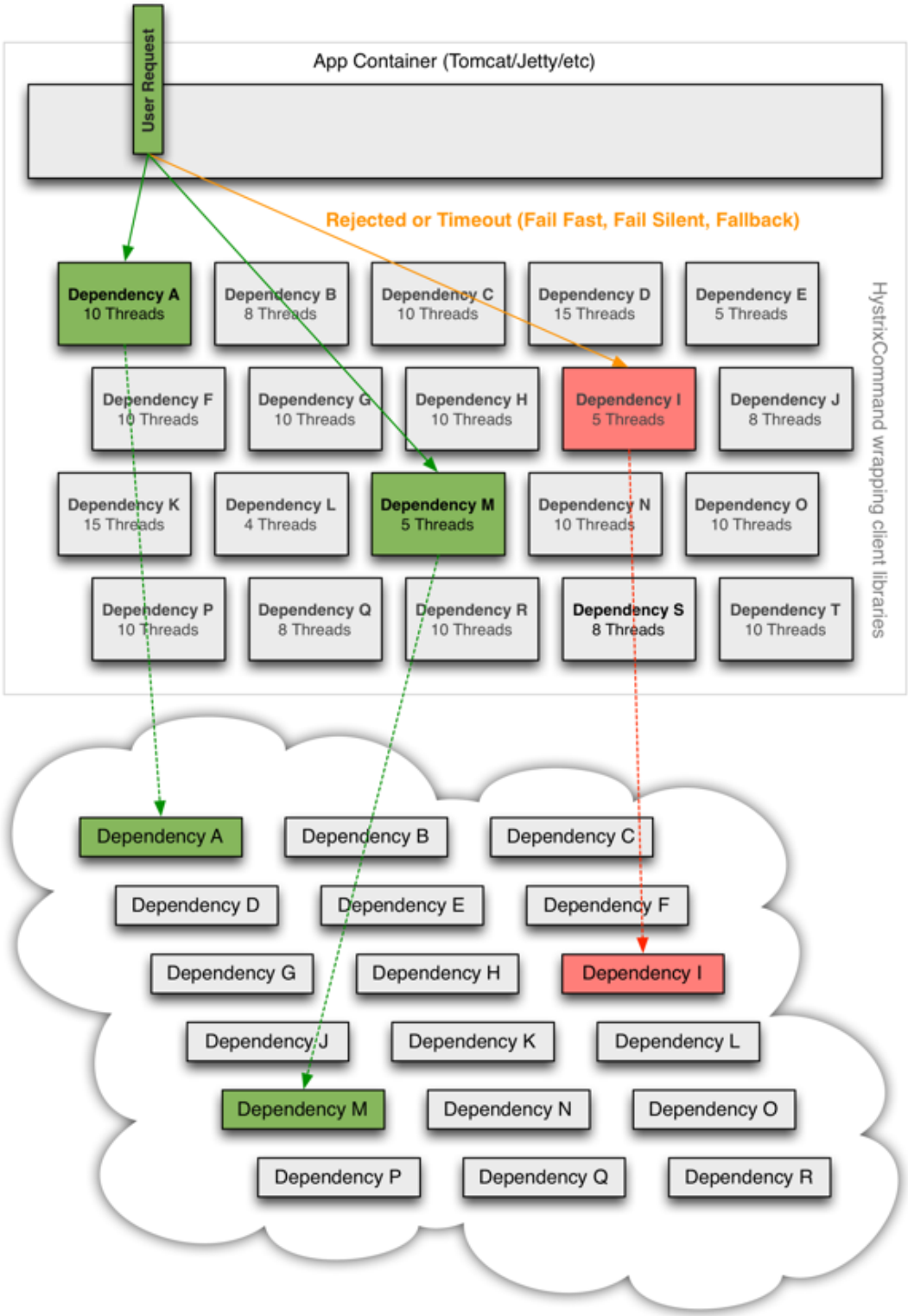
- 防止任何单个依赖项耗尽所有容器（如Tomcat）用户线程。
- 甩掉包袱，快速失败而不是排队。
- 在任何可行的地方提供回退，以保护用户不受失败的影响。
- 使用隔离技术（如隔离板、泳道和断路器模式）来限制任何一个依赖项的影响。
- 通过近实时的度量、监视和警报来优化发现时间。
- 通过配置的低延迟传播来优化恢复时间。
- 支持对Hystrix的大多数方面的动态属性更改，允许使用低延迟反馈循环进行实时操作修改。
- 避免在整个依赖客户端执行中出现故障，而不仅仅是在网络流量中。

4.1.4 Hystrix是如何实现它的目标的

1. 用一个HystrixCommand 或者 HystrixObservableCommand （这是命令模式的一个例子）包装所有的对外部系统（或者依赖）的调用，典型地它们在一个单独的线程中执行
2. 调用超时时间比你自己定义的阈值要长。有一个默认值，对于大多数的依赖项你是可以自定义超时时间的。
3. 为每个依赖项维护一个小的线程池(或信号量)；如果线程池满了，那么该依赖性将会立即拒绝请求，而不是排队。
4. 调用的结果有这么几种：成功、失败（客户端抛出异常）、超时、拒绝。
5. 在一段时间内，如果服务的错误百分比超过了一个阈值，就会触发一个断路器来停止对特定服务的所有请求，无论是手动的还是自动的。
6. 当请求失败、被拒绝、超时或短路时，执行回退逻辑。
7. 近实时监控指标和配置变化。

当你使用Hystrix来包装每个依赖项时，上图中所示的架构会发生变化，如下图所示：

每个依赖项相互隔离，当延迟发生时，它会被限制在资源中，并包含回退逻辑，该逻辑决定在依赖项中发生任何类型的故障时应作出何种响应：



4.1.5 Hystrix的配置步骤

1, 导入依赖

```xml
<!--hystrix的依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

2, 在启动类中添加@EnableCircuitBreaker注解

3, 在最外层添加熔断降级的处理. 在order-server中的控制器中添加`@HystrixCommand(fallbackMethod = "saveFail")`注解 (注意fallbackMethod需要和原方法一样的签名)

demo

> 熔断处理的方法
>
> ```
>
>     @RequestMapping("/queryOrder")
>     //在需要熔断处理的地方配置
>  @HystrixCommand(fallbackMethod = "queryOrderFali")
>  public Order queryOrder(Long id, HttpServletRequest request) {
>      String cookie = request.getHeader("cookie");
>      System.out.println("请求中的cookie为" + cookie);
> //         int i = 1 / 0;
>      return orderService.findById(id);
>  }
> ```
>
> 熔断的实际降级
>
> ```
>     //熔断的降级
>     public Order queryOrderFali(Long id, HttpServletRequest request) {
>      String cookie = request.getHeader("cookie");
>      System.out.println("请求中的cookie为" + cookie);
>      System.out.println("order执行降级");
>      return orderService.findById(id);
>  }
> ```

4.1.6 Fegin集成Hystrix

1, 在ProductFeignApi的`@FeignClient(name = "PRODUCT-SERVER",fallback = ProductFeignHystrix.class)`

demo

```
@FeignClient(name = "PRODUCT-SERVER", fallback =
ProductFeginHystrix.class)
public interface ProductFeignApi {
 @RequestMapping("findById")
 Product find(@RequestParam("id") Long id);
}
```

2, 实现一个feginHystrix的类并实现api的接口 (注意方法的签名要一致)

demo

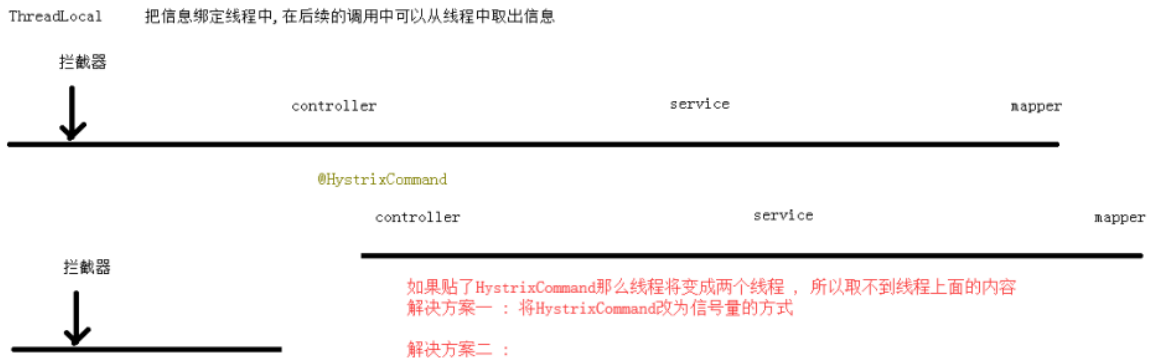
```java
@Component
public class ProductFeginHystrix implements ProductFeignApi {
 @Override
 public Product find(Long id) {
     System.out.println("默认的商品被执行了");
     return new Product(1L, "默认商品", 0);
 }
}
```

3, 配置文件中配置Hystrix开启 , 因为在默认的设置中, Hystrix是关闭的

```yaml
#配置hystrix开启
feign:
    hystrix:
        enabled: true
```

4.1.7 可能遇到的问题

在默认使用线城池的方式时线程内的对象无法获取



产生的原因

由于在通常情况下我们获取请求对象是从一个线程之中获取的 , 但是由于线城池将线程分配了出去 , 由于线程不是同一个 , 所以也就导致了线程内对象无法获取的问题

解决办法

将 HystrixCommand 该为信号量模式

在配置文件中配置不拦截

sensitiveHeaders

5.熔断与降级的异常警报处理

5.1.1 熔断与降级的异常警报处理机制

通常情况下，我们在一个系统发生了熔断或者降级的时候要发出一个警报，在发出警报的时候，要通知我们的运维人员来观察一下为什么会出现这样的情况.

但是我们不能每熔断一次就通知一下运维人员，这样运维人员的手机也许会成消息撑爆，所以我们只需要在熔断的时候通知一次，并在指定的时间在通知一次就可以了，基于以上的信息，发现我们使用Redis来存储这样一个消息通知是否有被发送是最合适的，当第一 次消息发送的时候，我们将该消息的key存放在redis中，用于鉴别我们是否发了这样一个消息，在20分钟后，超时过后自动清除Redis中的内容，然后此时还在熔断的话，就继续发送

5.1.2 代码实现

demo

```
public Order saveFail(Long userId,Long productId){
     //开启一个线程用于发送一个短信 ， 在真实的环境中 ， 使用另外一个服务或许更好一点
     new Thread(()->{
         String redisKey = "order-save";
         String value = stringRedisTemplate.opsForValue().get(redisKey);
         if(StringUtils.isEmpty(value)){
             System.out.println("order下订单服务失败，请查找原因.");
             stringRedisTemplate.opsForValue().set(redisKey,"save-order-fail",20, TimeUnit.SECONDS);
         }else{
             System.out.println("已经发送过短信");
         }
     }).start();
     return new Order();
 }

```

6 熔断隔离策略

hystrix里面，核心的一项功能，其实就是所谓的资源隔离，要解决的最最核心的问题，就是将多个依赖服务的调用分别隔离到各自自己的资源池内。避免说对某一个依赖服务的调用，因为依赖服务的接口调用的延迟或者失败，导致服务所有的线程资源全部耗费在这个服务的接口调用上。

6.1.2 熔断的隔离策略

一般情况下熔断是有两种隔离的策略 分别是 **线城池**，**信号量**

作用都是用来限流用的

Hystrix的默认采用的是线城池的熔断隔离机制

6.1.3 线城池

当一个请求进入到我们的服务中时，会给每一个线程池设定最大的访问量，当这个线程进入我们的线程池的时候，我们的请求堆中的请求是可以继续接受用户请求的，也就是说，当有15个请求进入线城池，线城池设置最大线程数量为10，通过线城池线程的分发后即使线城池满了 ，用户也是可以发送请求的，只不过此时用户的请求不会再发送到这个线程池中了

在该模式下，用户请求会被提交到各自的线程池中执行，把执行每个下游服务的线程分离，从而达到资源隔离的作用。当线程池来不及处理并且请求队列塞满时，新进来的请求将快速失败，可以避免依赖问题扩散。

优势

- 减少所依赖服务发生故障时的影响面，比如ServiceA服务发生异常，导致请求大量超时，对应的线程池被打满，这时并不影响ServiceB、ServiceC的调用。
- 如果接口性能有变动，可以方便的动态调整线程池的参数或者是超时时间，前提是Hystrix参数实现了动态调整。

缺点

- 请求在线程池中执行，肯定会带来任务调度、排队和上下文切换带来的开销。
- 因为涉及到跨线程，那么就存在ThreadLocal数据的传递问题，比如在主线程初始化的ThreadLocal变量，在线程池线程中无法获取

总结 : 线城池适合调用异步的请求，一些耗时比较久的请求 ,采用线城池是好的方案，并且线城池可以接受更多的用户请求

6.1.4 信号量

在该模式下，接受请求和执行下游在同一个线程完成，不存在线程的上下文切换所带来的性能开销，但是如果遇到了这种情况，比如一个接口 依赖了三个下游，serviceA，serviceB，serviceC 并且这三个下游之间返回的数据并不是相互依赖的, 那么此时该模式的单一线程执行耗时 就是 A + B +C 的总和，最重要的是, 在这个线程访问其中一个下游的时候，接受请求的地方是不可以在接受新的请求的，这样线程执行的时间将会非常漫长，这样造成的结果就是，一个用户在访问的时候，等待了15秒，在这15秒期间，其余的用户是不能访问的，并且当上一个用户执行完毕, 这个用户又开始等待15秒

该模式的使用处在于，内部的访问，比如在没有涉及到外部访问的时候，访问一个复杂代码的业务逻辑 ，这样的情况下该模式的执行时间是很短的

6.1.5 简单介绍

通常大部分的情况下，线城池能解决我们很多的问题

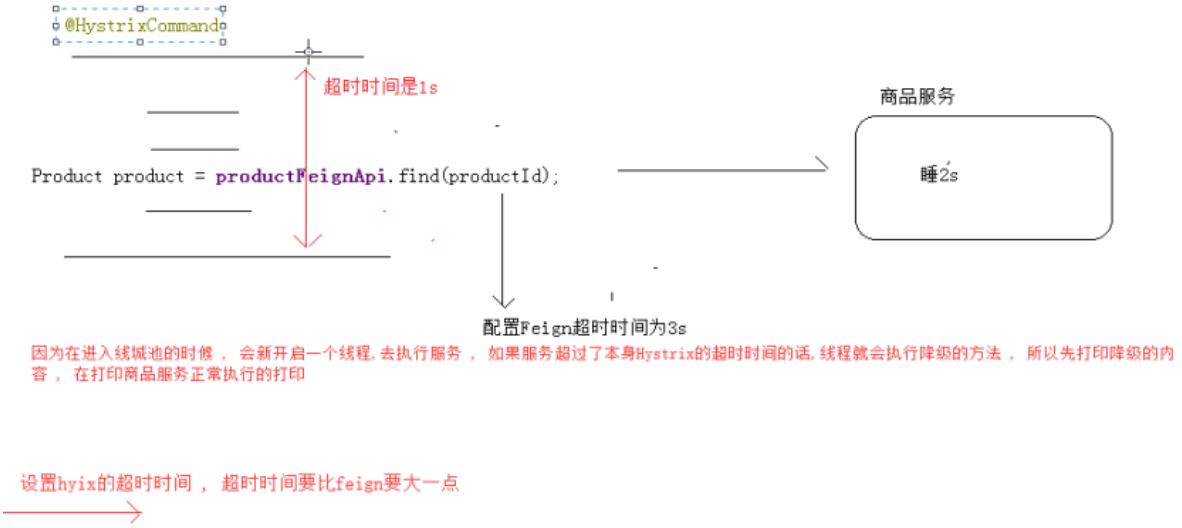
因为Hystrix默认使用了线程池模式，所以对于每个Command，在初始化的时候，会创建一个对应的线程池，如果项目中需要进行降级的接口非常多，比如有上百个的话，不太了解Hystrix内部机制的同学，按照默认配置直接使用，可能就会造成线程资源的大量浪费。

7 . 超时时间调整

HystrixCommand超时机制

明明代码没有问题，但是执行最后的结果顺序有问题

图解:



由于HystrixCommand 本身就带有超时机制，当线城池下游的服务如果执行时间过长，导致HystrixCommand 在超时时间范围内，没有收到来自线程池内部的响应的话，HystrixCommand 的主线程就不会等待分发到下游的线程，从而直接去请求的下一个目标

解决的办法

给HystrixCommand 配置超时时间

注：Hystrix的超时时间应该等于 ribbon访问远程的超时时间 + 执行时间

```
#是否开启超时限制 （一定不要禁用）
hystrix:
      command:
        default:
            execution:
                timeout:
                    enabled:

#超时时间调整

hystrix:
    command:
        default:
          execution:
            isolation:
              thread:
                timeoutInMilliseconds: 4000
```

8.断路器监控仪表盘

断路器监控仪表盘 其实就是一个可视化页面，用于监控 Hystrix 各个服务之间调用的时间，缓存，熔断 / 降级原因等问题

使用方式如下：

1. 加入一依赖

```
仪表盘的依赖
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

```
仪表盘的另一个依赖
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>


```

2 . 增加配置

```
#添加地址
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

3 . 在启动类上贴@EnableHystrixDashboard

4.访问入口：http://localhost:启动端口/hystrix，然后再地址栏上输入:http://localhost:启动端口/actuator/hystrix.stream

参数