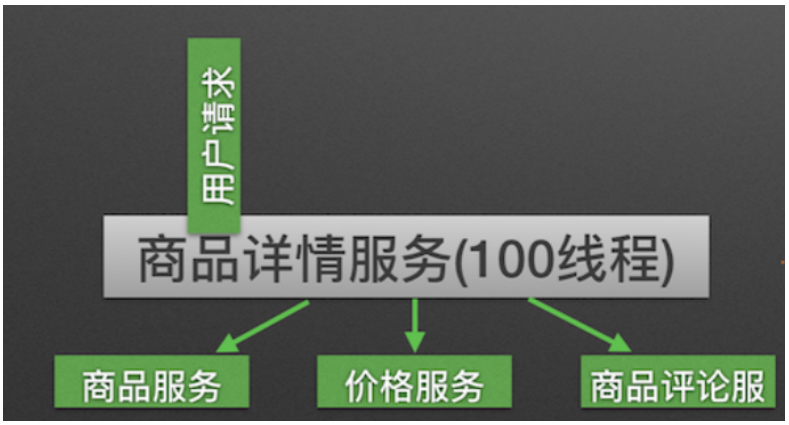
分布式系统中，会出现哪些问题？

分布式系统中一定会遇到的一个问题：服务雪崩效应 或者叫级联效应

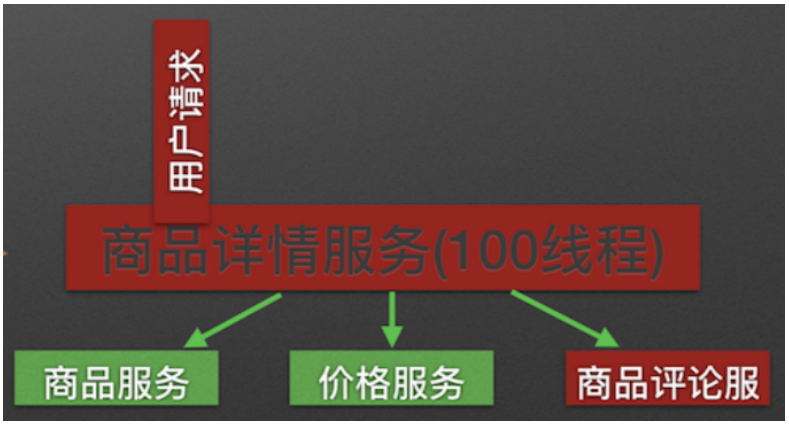
那么什么是服务雪崩效应呢？

在一个高度服务化的系统中，我们实现的一个业务逻辑通常会依赖多个服务，比如：

商品详情展示服务会依赖商品服务，价格服务，商品评论服务．如图所示：



调用三个依赖服务会共享商品详情服务的线程池．如果其中的商品评论服务不可用，就会出现线程池里所有线程都因等待响应而被阻塞，从而造成服务雪崩．如图所示：



服务雪崩效应：因服务提供者的不可用导致服务调用者的不可用,并将不可用逐渐放大的过程，就叫服务雪崩效应

讲了什么是服务雪崩效应，那么同学们知道为什么服务不可用吗？导致服务不可用的原因老师总结有几点： 程序Bug，大流量请求，硬件故障，缓存击穿

【大流量请求】：在秒杀和大促开始前,如果准备不充分,瞬间大量请求会造成服务提供者的不可用.

【硬件故障】：可能为硬件损坏造成的服务器主机宕机, 网络硬件故障造成的服务提供者的不可访问.

【缓存击穿】：一般发生在缓存应用重启, 缓存失效时高并发， 所有缓存被清空时,以及短时间内大量缓存失效时. 大量的缓存不命中, 使请求直击后端,造成服务提供者超负荷运行,引起服务不可用.

而且，同学们，在服务提供者不可用的时候，会出现重试的情况：用户重试、代码逻辑重试
用户重试：在服务提供者不可用后, 用户由于忍受不了界面上长时间的等待,会不断刷新页面甚至提交表单.
代码逻辑重试：服务调用端的会存在大量服务异常后的重试逻辑.

这些重试最终导致：进一步加大请求流量.

那么，归根结底导致雪崩效应的最根本原因是：
大量请求线程同步等待造成的资源耗尽
当服务调用者使用 同步调用 时, 会产生大量的等待线程占用系统资源. 一旦线程资源被耗尽, 服务调用者提供的服务也将处于不可用状态, 于是服务雪崩效应产生了.

那么知道了分布式系统中的服务雪崩效应，以及产生的原因。那么，问题来了，怎么解决？

解决方案

那么解决方案有很多，老师根据自己的经验列举一下：

1. 超时机制
2. 服务限流
3. 服务熔断
4. 服务降级

当然还有其他的，老师不一一列举

超时机制

刚刚老师分析了，服务级联失败（服务雪崩效应）的最根本原因是：大量请求线程同步等待造成的资源耗尽
那么，在不做任何处理的情况下，服务提供者不可用会导致消费者请求线程强制等待，而造成系统资源耗尽，而且，既然服务提供者已经不可用了，还在作死的请求的话，是毫无意义

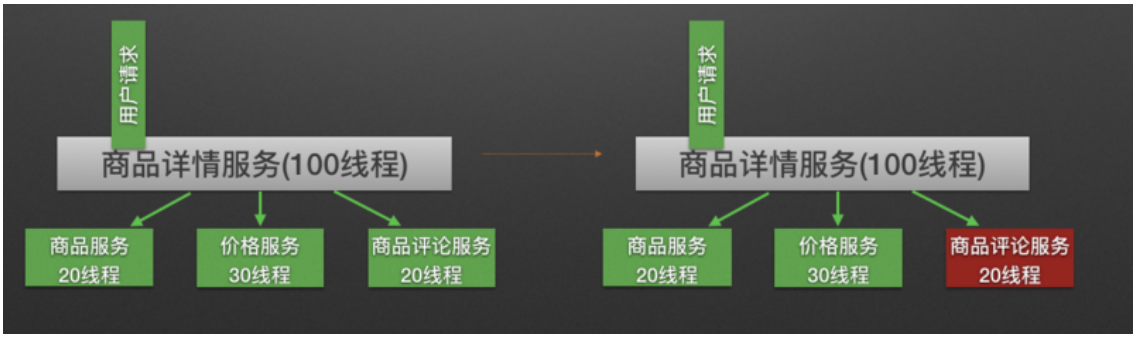
的。那么，如果我们加入超时机制，例如2s，那么超过2s就会直接返回了，那么这样是不是一定程度上可以抑制消费者资源耗尽的问题。

服务限流（资源隔离）

也就是限制请求核心服务提供者的流量，使大流量拦截在核心服务之外，这样可以更好的保证核心服务提供者不出问题，对于一些出问题的服务可以限制流量访问，只分配固定线程资源访问，这样能使整体的资源不至于被出问题的服务耗尽，进而整个系统雪崩

那么服务之间怎么限流，怎么资源隔离了？例如通过线程池+队列的方式，通过信号量的方式。

如下图所示，当商品评论服务不可用时，即使商品服务独立分配的20个线程全部处于同步等待状态，也不会影响其他依赖服务的调用.



服务熔断

远程服务不稳定或网络抖动时暂时关闭，就叫服务熔断。

可能有些同学还是不懂，举个通俗易懂的例子，就跟我们现实生活中的“跳闸”一样，跳闸同学们应该都听说过吧，比如说家里有点短路了，那是不是闸会跳掉，等你把短路的问题找到并且修复后，然后你把这个闸一送，是不是整个家庭的电路又恢复了正常。这就是**熔断器**。

所以，同样的道理，当依赖的服务有大量超时时，在让新的请求去访问根本没有意义，只会无畏的消耗现有资源。比如我们设置了超时时间为1s,如果短时间内有大量请求在1s内都得不到响应，就意味着这个服务出现了异常，此时就没有必要再让其他的请求去访问这个依赖了，这个时候就应该使用熔断器避免资源浪费。

服务降级

有服务熔断，必然要有服务降级。

所谓降级，就是当某个服务熔断之后，服务将不再被调用，此时客户端可以自己准备一个本地的fallback（回退）回调，返回一个缺省值。 例如：(备用接口/缓存/mock数据)

这样做，虽然服务水平下降，但好歹可用，比直接挂掉要强，当然这也要看适合的业务场景。

好，同学们又有疑问了，老师讲了这么多理论，那么**到底怎么来实际落地**呢？
好，那么接下来将实际落地的实现。

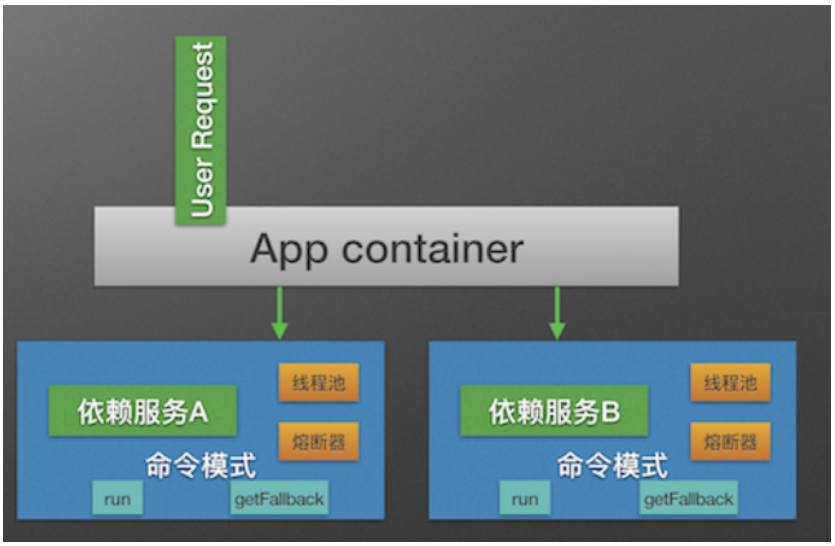
实战Hystrix-降级，超时

在整个SpringCloud构建微服务的体系中，有一个提供超时机制，限流，熔断，降级最全面的实现：Hystrix(豪猪) 翻译过来表示：自身带刺，有自我保护的意思，外国人起名字还是很有意思滴。当然Hystrix并不是Spring的，而是NetFlix公司开源的。那么Spring只是把它拿过来，在他的基础上面做了一些封装，然后加入到了SpringCloud中，实现高可用的分布式微服务架构
Hystrix博大精深，功能齐全

直接看代码示例：06-ms-consumer-order-ribbon-hystrix-customizing
1、引入Springcloud Hystrix**依赖，那么在哪里引入呢？**
一定是在调用方来做降级，所以需要在消费者这边引入Hystrix，也就是我们的**订单微服务方**

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

2、那么Hystrix是怎么做到降级的呢？
他通过一种叫做“命令模式”的方式，继承(HystrixCommand类)来包裹具体的服务调用逻辑(run方法), 并在命令模式中添加了服务调用失败后的降级逻辑(getFallback)，见(OrderServiceCommand类)



3、用Hystrix的注解@HystrixCommand可以更简单的实现上面的降级逻辑

见代码示例：06-ms-consumer-order-ribbon-hystrix-fallback

直接在接口调用方的方法上增加注解@HystrixCommand(fallbackMethod = "findByIdFallback")

```
@HystrixCommand(fallbackMethod = "findByIdFallback")
@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
  logger.info("================请求用户中心接口==============");
  return this.restTemplate.getForObject("http://microservice-provider-user/" + id, User.class);
}

public User findByIdFallback(Long id) {
  User user = new User();
  user.setId(-1L);
  user.setName("默认用户");
  return user;
}
```

4、超时回退怎么实现？

在用户微服务工程(06-ms-provider-user)里将UserController的findById接口增加执行等待时间，让该接口的执行时间变长，Hystrix调用接口默认两秒超时，超时后会自动执行降级方法

```
@GetMapping("/{id}")
public User findById(@PathVariable Long id) throws Exception {
    //测试超时触发降级
    int sleepTime = new Random().nextInt(3000);
    logger.info("sleepTime:" + sleepTime);
    Thread.sleep(sleepTime);

    User findOne = userRepository.findOne(id);
    return findOne;
}
```

实战Hystrix-熔断，限流

熔断怎么实现？见示例：06-ms-consumer-order-ribbon-hystrix-fusing

在用户微服务工程(06-ms-provider-user)里将UserController的findById接口增加模拟报错代码

```
@GetMapping("/{id}")
public User findById(@PathVariable Long id) throws Exception {
    /*//测试超时触发降级
    int sleepTime = new Random().nextInt(3000);
    logger.info("sleepTime:" + sleepTime);
    Thread.sleep(sleepTime); */

    //测试熔断，传入不存在的用户id模拟异常情况
    if (id == 10) {
      throw new NullPointerException();
    }

    User findOne = userRepository.findOne(id);
    return findOne;
}
```

测试报错和正常的情况，我们可以看到当报错达到一定阈值时，会自动熔断，阈值可以配置，如下:

一个rolling window内最小的请求数。如果设为20，那么当一个rolling window的时间内（比如说1个rolling window是10秒）收到19个请求，即使19个请求都失败，也不会触发circuit break。默认20

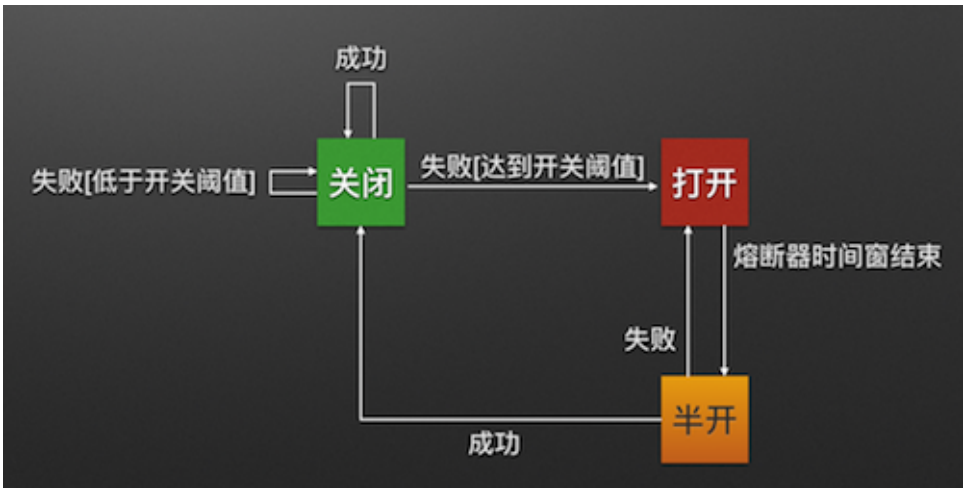
hystrix.command.default.circuitBreaker.requestVolumeThreshold

触发短路的时间值，当该值设为5000时，则当触发circuit break后的5000毫秒内都会拒绝request，也就是5000毫秒后才会关闭circuit。默认5000

hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds

熔断器原理

熔断器模式定义了熔断器开关相互转换的逻辑:



服务的健康状况 = 请求失败数 / 请求总数.

熔断器开关由关闭到打开的状态转换是通过当前服务健康状况和设定阈值比较决定的.

1、当熔断器开关关闭时，请求被允许通过熔断器. 如果当前健康状况高于设定阈值，开关继续保持关闭. 如果当前健康状况低于设定阈值，开关则切换为打开状态.

2、当熔断器开关打开时，请求被禁止通过.

3、当熔断器开关处于打开状态，经过一段时间后，熔断器会自动进入半开状态，这时熔断器只允许一个请求通过. 当该请求调用成功时，熔断器恢复到关闭状态. 若该请求失败，熔断器继续保持打开状态，接下来的请求被禁止通过.

熔断器的开关能保证服务调用者在调用异常服务时，快速返回结果，避免大量的同步等待. 并且熔断器能在一段时间后继续侦测请求执行结果，提供恢复服务调用的可能.

限流，线程资源隔离怎么实现？见示例：06-ms-consumer-order-ribbon-hystrix-thread-isolation

在用户微服务工程(06-ms-provider-user)里将UserController的findById接口模拟执行等待的代码

```
@GetMapping("/{id}")
public User findById(@PathVariable Long id) throws Exception {
    logger.info("用户中心接口：查询用户"+ id +"信息");
    /*//测试超时触发降级
    int sleepTime = new Random().nextInt(3000);
    logger.info("sleepTime:" + sleepTime);
    Thread.sleep(sleepTime); */

    /*//测试熔断，传入不存在的用户id模拟异常情况
    if (id == 10) {
      throw new NullPointerException();
    }*/

    //测试限流，线程资源隔离，模拟系统执行速度很慢的情况
    Thread.sleep(3000);

    User findOne = userRepository.findOne(id);
    return findOne;
}
```

在订单微服务工程(06-ms-consumer-order-ribbon-hystrix-thread-isolation)中修改接口超时配置为20秒：

```yaml
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 20000
```

然后用注解配置线程池大小：

```java
@RestController
public class OrderController {
    private static final Logger logger = LoggerFactory.getLogger(OrderController.class);
    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "findByIdFallback",
                    groupKey = "orderGroup",
                    threadPoolKey = "orderThreadPool",
                    threadPoolProperties = {
                        @HystrixProperty(name = "coreSize", value = "2")/*,
                        @HystrixProperty(name = "maxQueueSize", value = "1"),
                        @HystrixProperty(name = "queueSizeRejectionThreshold", value = "6")*/ })
    @GetMapping("/user/{id}")
    public User findById(@PathVariable Long id) {
        logger.info("================请求用户中心接口，用户id:" + id + "==============");
        return restTemplate.getForObject("http://microservice-provider-user/" + id, User.class);
    }
```

部分注解意思如下：

CommandGroupKey：配置全局唯一标识服务分组的名称，比如，库存系统就是一个服务分组。当我们监控时，相同分组的服务会聚合在一起，必填选项。

CommandKey：配置全局唯一标识服务的名称，比如，库存系统有一个获取库存服务，那么就可以为这个服务起一个名字来唯一识别该服务，如果不配置，则默认是简单类名。

ThreadPoolKey：配置全局唯一标识线程池的名称，相同线程池名称的线程池是同一个，如果不配置，则默认是分组名，此名字也是线程池中线程名字的前缀。

ThreadPoolProperties：配置线程池参数，coreSize配置核心线程池大小和线程池最大大小，keepAliveTimeMinutes是线程池中空闲线程生存时间（如果不进行动态配置，那么是没有任何作用的），maxQueueSize配置线程池队列最大大小，queueSizeRejectionThreshold限定当前队列大小，即实际队列大小由这个参数决定，通过改变queueSizeRejectionThreshold可以实现动态队列大小调整。

CommandProperties：配置该命令的一些参数，如executionIsolationStrategy配置执行隔离策略，默认是使用线程隔离，此处我们配置为THREAD，即线程池隔离。

此处可以粗粒度实现隔离，也可以细粒度实现隔离，如下所示。

服务分组+线程池：粗粒度实现，一个服务分组/系统配置一个隔离线程池即可，不配置线程池名称或者相同分组的线程池名称配置为一样。

服务分组+服务+线程池：细粒度实现，一个服务分组中的每一个服务配置一个隔离线程池，为不同的命令实现配置不同的线程池名称即可。

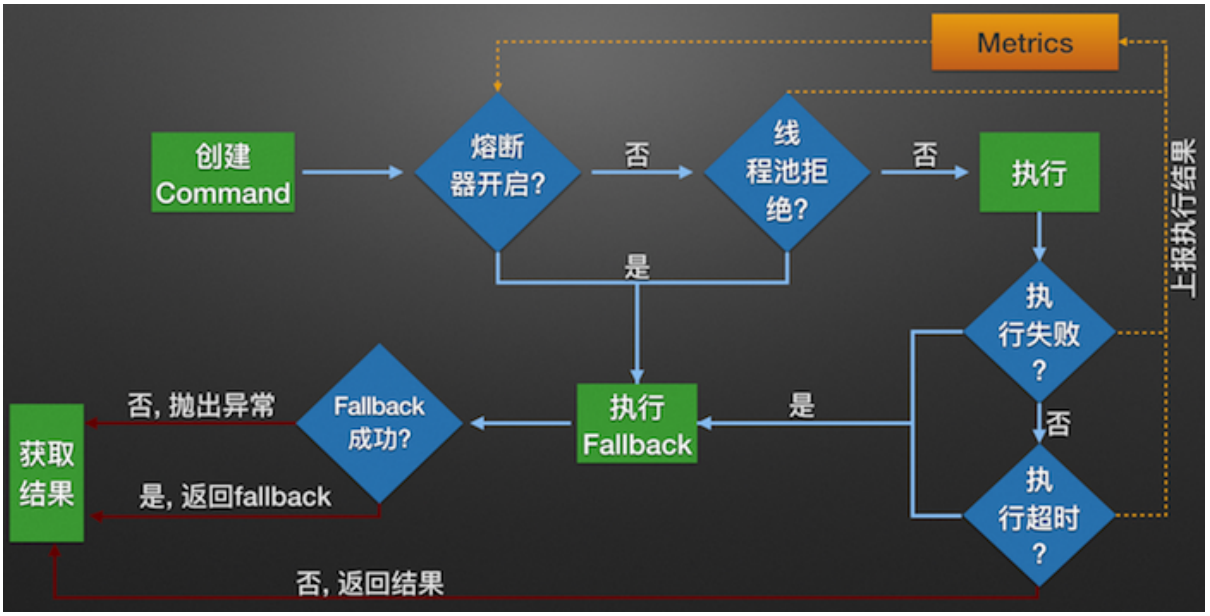
混合实现：一个服务分组配置一个隔离线程池，然后对重要服务单独设置隔离线程池。

本demo可以用Jmeter模拟多线程调用来验证结果，参看上课视频

那么，这个是Hystrix基于线程池+队列的方式实现的限流，当然，还有另外一种，基于**信号量**来实现的。同学们知道什么是信号量把，说白了就是计数器。
当 *计数器 + 1 > 设置的最大并发数* 时，就限流，或者走降级渠道

Hystrix服务调用的内部逻辑

下图为Hystrix服务调用的内部逻辑:



1. 构建Hystrix的Command对象，调用执行方法.
2. Hystrix检查当前服务的熔断器开关是否开启，若开启，则执行降级服务getFallback方法.
3. 若熔断器开关关闭，则Hystrix检查当前服务的线程池是否能接收新的请求，若线程池已满，则执行降级服务getFallback方法.
4. 若线程池接受请求，则Hystrix开始执行服务调用具体逻辑run方法.
5. 若服务执行失败，则执行降级服务getFallback方法，并将执行结果上报Metrics更新服务健康状况.
6. 若服务执行超时，则执行降级服务getFallback方法，并将执行结果上报Metrics更新服务健康状况.
7. 若服务执行成功，返回正常结果.
8. 若服务降级方法getFallback执行成功，则返回降级结果.
9. 若服务降级方法getFallback执行失败，则抛出异常.

Hystrix Metrics的实现

Hystrix的Metrics中保存了当前服务的健康状况, 包括服务调用总次数和服务调用失败次数等. 根据Metrics的计数, 熔断器从而能计算出当前服务的调用失败率, 用来和设定的阈值比较从而决定熔断器的状态切换逻辑.

Hystrix相关配置:

```
hystrix.command.default和hystrix.threadpool.default中的default为默认CommandKey

Command Properties
Execution相关的属性的配置:
hystrix.command.default.execution.isolation.strategy 隔离策略，默认是Thread， 可选Thread | Semaphore

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 命令执行超时时间，默认1000ms

hystrix.command.default.execution.timeout.enabled 执行是否启用超时，默认启用true
hystrix.command.default.execution.isolation.thread.interruptOnTimeout 发生超时是是否中断，默认true
hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests 最大并发请求数，默认10，该参数当使用ExecutionIsolationStrategy.SEMAPHORE策略时才有效。如果达到最大并发请求数，请求会被拒绝。理论上选择semaphore size的原则和选择thread size一致，但选用semaphore时每次执行的单元要比较小且执行速度快（ms级别），否则的话应该用thread。
semaphore应该占整个容器（tomcat）的线程池的一小部分。
Fallback相关的属性
这些参数可以应用于Hystrix的THREAD和SEMAPHORE策略

hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests 如果并发数达到该设置值，请求会被拒绝和抛出异常并且fallback不会被调用。默认10
hystrix.command.default.fallback.enabled 当执行失败或者请求被拒绝，是否会尝试调用hystrixCommand.getFallback() 。默认true
Circuit Breaker相关的属性
hystrix.command.default.circuitBreaker.enabled 用来跟踪circuit的健康性，如果未达标则让request短路。默认true
hystrix.command.default.circuitBreaker.requestVolumeThreshold 一个rolling window内最小的请求数。如果设为20，那么当一个rolling window的时间内（比如说1个rolling window是10秒）收到19个请求，即使19个请求都失败，也不会触发circuit break。默认20
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds 触发短路的时间值，当该值设为5000时，则当触发circuit break后的5000毫秒内都会拒绝request，也就是5000毫秒后才会关闭circuit。默认5000
hystrix.command.default.circuitBreaker.errorThresholdPercentage错误比率阀值，如果错误率>=该值，circuit会被打开，并短路所有请求触发fallback。默认50
hystrix.command.default.circuitBreaker.forceOpen 强制打开熔断器，如果打开这个开关，那么拒绝所有request，默认false
hystrix.command.default.circuitBreaker.forceClosed 强制关闭熔断器 如果这个开关打开，circuit将一直关闭且忽略circuitBreaker.errorThresholdPercentage
Metrics相关参数
hystrix.command.default.metrics.rollingStats.timeInMilliseconds 设置统计的时间窗口值的，毫秒值，circuit break 的打开会根据1个rolling window的统计来计算。若rolling window被设为10000毫秒，则rolling window会被分成n个buckets，每个bucket包含success，failure，timeout，rejection的次数的统计信息。默认10000
hystrix.command.default.metrics.rollingStats.numBuckets 设置一个rolling window被划分的数量，若numBuckets = 10，rolling window = 10000，那么一个bucket的时间即1秒。必须符合rolling window % numberBuckets == 0。默认10
hystrix.command.default.metrics.rollingPercentile.enabled 执行时是否enable指标的计算和跟踪，默认true
hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds 设置rolling percentile window的时间，默认60000
```

hystrix.command.default.metrics.rollingPercentile.numBuckets 设置rolling percentile window的numberBuckets。逻辑同上。默认6
hystrix.command.default.metrics.rollingPercentile.bucketSize 如果bucket size=100，window=10s，若这10s里有500次执行，只有最后100次执行会被统计到bucket里去。增加该值会增加内存开销以及排序的开销。默认100
hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds 记录health 快照（用来统计成功和错误绿）的间隔，默认500ms
Request Context 相关参数
hystrix.command.default.requestCache.enabled 默认true，需要重载getCacheKey()，返回null时不缓存
hystrix.command.default.requestLog.enabled 记录日志到HystrixRequestLog，默认true

Collapser Properties 相关参数
hystrix.collapser.default.maxRequestsInBatch 单次批处理的最大请求数，达到该数量触发批处理，默认Integer.MAX_VALUE
hystrix.collapser.default.timerDelayInMilliseconds 触发批处理的延迟，也可以为创建批处理的时间+该值，默认10
hystrix.collapser.default.requestCache.enabled 是否对HystrixCollapser.execute() and HystrixCollapser.queue()的cache，默认true

ThreadPool 相关参数
线程数默认值10适用于大部分情况（有时可以设置得更小），如果需要设置得更大，那有个基本得公式可以follow：
requests per second at peak when healthy × 99th percentile latency in seconds + some breathing room
每秒最大支撑的请求数（99%平均响应时间 + 缓存值）
比如：每秒能处理1000个请求，99%的请求响应时间是60ms，那么公式是：
1000 （0.060+0.012）

基本得原则时保持线程池尽可能小，他主要是为了释放压力，防止资源被阻塞。
当一切都是正常的时候，线程池一般仅会有1到2个线程激活来提供服务

hystrix.threadpool.default.coreSize 并发执行的最大线程数，默认10
hystrix.threadpool.default.maxQueueSize BlockingQueue的最大队列数，当设为-1，会使用SynchronousQueue，值为正时使用LinkedBlcokingQueue。该设置只会在初始化时有效，之后不能修改threadpool的queue size，除非reinitialising thread executor。默认-1。
hystrix.threadpool.default.queueSizeRejectionThreshold 即使maxQueueSize没有达到，达到queueSizeRejectionThreshold该值后，请求也会被拒绝。因为maxQueueSize不能被动态修改，这个参数将允许我们动态设置该值。if maxQueueSize == -1，该字段将不起作用
hystrix.threadpool.default.keepAliveTimeMinutes 如果corePoolSize和maxPoolSize设成一样（默认实现）该设置无效。如果通过plugin（https://github.com/Netflix/Hystrix/wiki/Plugins）使用自定义实现，该设置才有用，默认1.
hystrix.threadpool.default.metrics.rollingStats.timeInMilliseconds 线程池统计指标的时间，默认10000
hystrix.threadpool.default.metrics.rollingStats.numBuckets 将rolling window划分为n个buckets，默认10