主讲老师： fox

课程内容：

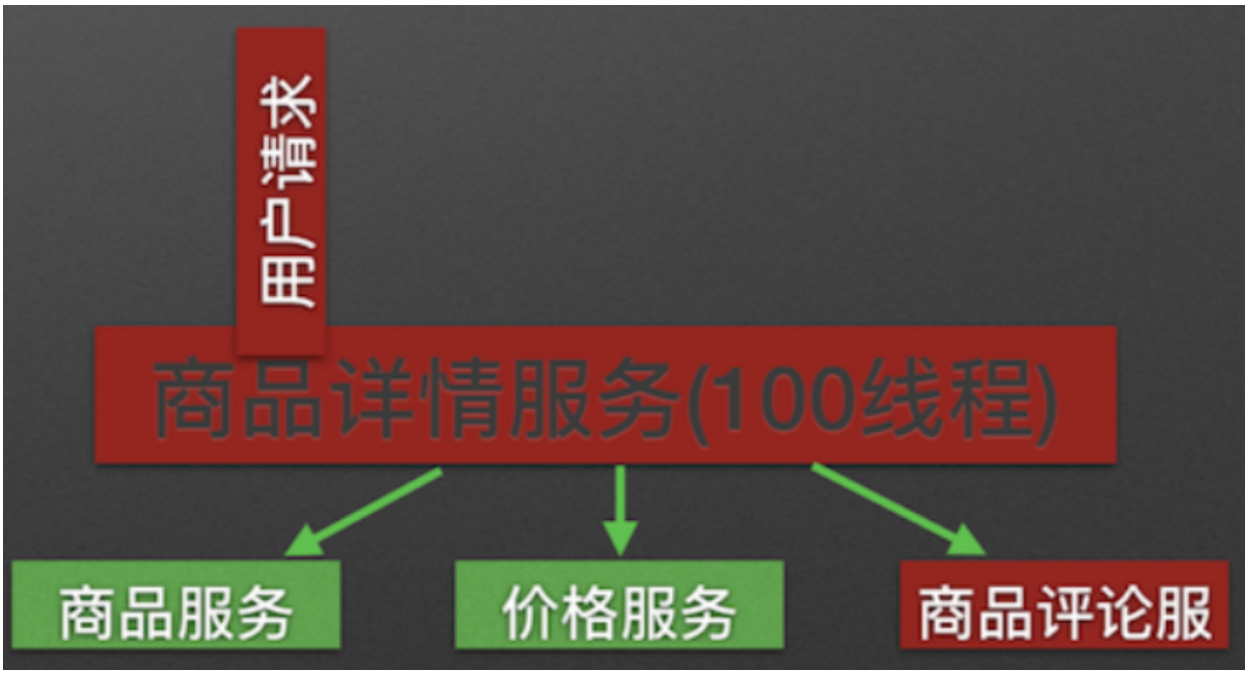
1. 服务调用组件feign源码分析
2. 服务雪崩的原因及其解决方案详解，
3. Hystrix使用及其原理剖析，熔断器跳闸机制分析，内部调用逻辑剖析

1. 前言

1. 1） 分布式系统遇到的问题

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

导致服务不可用的原因： 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。

在服务提供者不可用的时候，会出现大量重试的情况：用户重试、代码逻辑重试，这些重试最终导致：进一步加大请求流量。所以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。

1.2）解决方案

1. 超时机制
2. 服务限流(资源隔离)
3. 服务熔断
4. 服务降级

1.2.1） 超时机制

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。加入超时机制，一旦超时，就释放资源。由于释放资源速度较快，一定程度上可以抑制资源耗尽的问题。

RestTemplate设置超时时间

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
 //设置restTemplate的超时时间
 SimpleClientHttpRequestFactory requestFactory = new SimpleClientHttpRequestFactory();
 requestFactory.setReadTimeout(2000);
 requestFactory.setConnectTimeout(2000);
 RestTemplate restTemplate = new RestTemplate(requestFactory);
 return restTemplate;
 }

@Bean
@LoadBalanced
public RestTemplate restTemplate() {
 //设置restTemplate的超时时间
 HttpComponentsClientHttpRequestFactory httpRequestFactory = new HttpComponentsClientHttpRequestFactory();
 httpRequestFactory.setConnectionRequestTimeout(2000);
 httpRequestFactory.setConnectTimeout(2000);
 httpRequestFactory.setReadTimeout(2000);
 return new RestTemplate(httpRequestFactory);
 }

```

全局异常设置

```
@ControllerAdvice
public class MyExcpetionHandler {

 @ExceptionHandler(value = MyTimeoutException.class)
 @ResponseBody
 public Object handleException() {
 UserInfoVo userInfoVo = new UserInfoVo();
 userInfoVo.setUsername("超时异常");
 userInfoVo.setOrderList(null);
 return userInfoVo;
```

```
 }
}
@Data
@AllArgsConstructor
public class MyTimeoutException extends RuntimeException {

 private Integer code;

 private String msg;
}
```

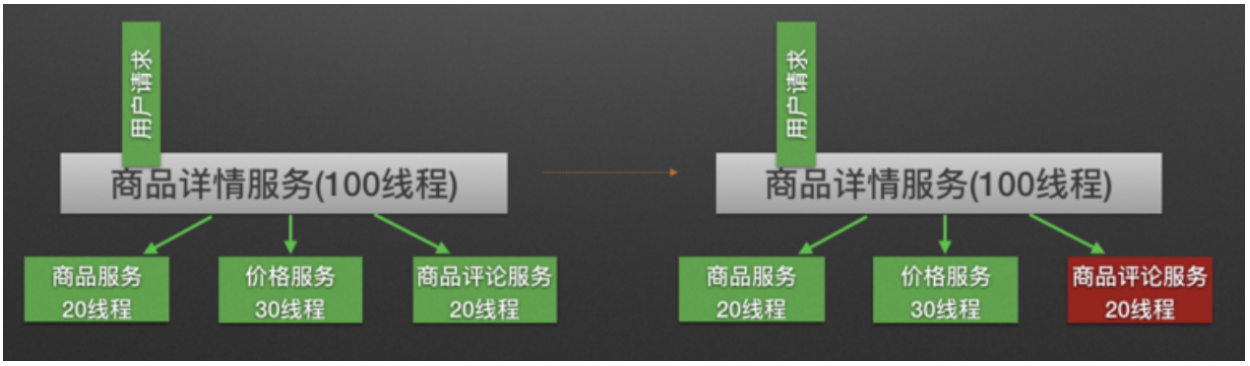
调用时捕获异常

```
String url = "http://service-order/order/findOrderByUserId/"+id;
ResponseEntity<List> responseEntity = null;
try {
 responseEntity = restTemplate.getForEntity(url, List.class);
} catch (RestClientException e) {
 throw new MyTimeoutException(-1,"调用超时");
}
List<Order> orderList = responseEntity.getBody();
```

1.2.2） 服务限流(资源隔离)

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？例如可以通过线程池+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。



1.1.3) 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。

现实世界的断路器大家肯定都很了解，断路器实时监控电路的情况，如果发现电路电流异常，就会跳闸，从而防止电路被烧毁。

软件世界的断路器可以这样理解：实时监测应用，如果发现在一定时间内失败次数/失败率达到一定阈值，就“跳闸”，断路器打开——此时，请求直接返回，而不去调用原本调用的逻辑。跳闸一段时间后（例如10秒），断路器会进入半开状态，这是一个瞬间态，此时允许一次请求调用该调的逻辑，如果成功，则断路器关闭，应用正常调用；如果调用依然不成功，断路器继续回到打开状态，过段时间再进入半开状态尝试——通过”跳闸“，应用可以保护自己，而且避免浪费资源；而通过半开的设计，可实现应用的“自我修复“。

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断路器避免资源浪费。

1.1.4) 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。

2. Hystrix快速开始

Hystrix（豪猪）是由Netflix开源的一个延迟和容错库，提供超时机制，限流，熔断，降级最全面的实现，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。

2.1）引入依赖

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

2.2）启动类加@EnableCircuitBreaker注解

```
@SpringBootApplication
@EnableCircuitBreaker
public class HystrixConsumerApplication {

 public static void main(String[] args) {
 SpringApplication.run(HystrixConsumerApplication.class,args);
```

```
    }
}
```

2.3） 服务降级方式

2.3.1）自定义命令行调用方式

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)

```
@Slf4j
public class MyHystrixCommand extends HystrixCommand<List<Order>> {

 private Integer userId;

 private RestTemplate restTemplate;

 public MyHystrixCommand(String commandGroupKey, Integer userId, RestTemplate restTemplate) {
 super(HystrixCommandGroupKey.Factory.asKey(commandGroupKey));
  this.userId = userId;
  this.restTemplate = restTemplate;
  }

  @Override
  protected List<Order> run() throws Exception {
  String url = "http://service-order/order/findOrderByUserId/"+userId;
  ResponseEntity<List> responseEntity =
  restTemplate.getForEntity(url, List.class);
  List<Order> orderList = responseEntity.getBody();
  log.info("查询orderList:"+ orderList);
  return orderList;
  }

  @Override
  protected List<Order> getFallback() {
  log.info("===调用降级方法====");
  Order order = new Order();
  order.setId(-1);
  order.setUserId(-1);
  order.setAmount(0);

  List<Order> orderList = new ArrayList<>();
  orderList.add(order);
```

```
 return orderList;
 }
}

//UserController.java

@RequestMapping(value = "/getById/{id}")
public UserInfoVo getUserById(@PathVariable("id") Integer id) {
 User user = userService.getById(id);

 //调用命令模式
 MyHystrixCommand myHystrixCommand =
 new MyHystrixCommand("orderGroupKey",id,restTemplate);
 List<Order> orderList =myHystrixCommand.execute();

 UserInfoVo userInfoVo = new UserInfoVo();
 userInfoVo.setOrderList(orderList);
 userInfoVo.setUsername(user.getUsername());

 return userInfoVo;
}
```

2.3.2) 通过@HystrixCommand指定降级方法

```
@HystrixCommand(fallbackMethod ="queryUserInfoFallBack")
@RequestMapping("/getById/{id}")
public UserInfoVo getUserById(@PathVariable("id") Integer id) {
 User user = userService.getById(id);

 String url = "http://service-order/order/findOrderByUserId/"+id;
 ResponseEntity<List> responseEntity =
 restTemplate.getForEntity(url, List.class);
 List<Order> orderList = responseEntity.getBody();

 UserInfoVo userInfoVo = new UserInfoVo();
 userInfoVo.setOrderList(orderList);
 userInfoVo.setUsername(user.getUsername());

 return userInfoVo;
}
```

测试： 服务宕机，超时，服务异常

```
@RequestMapping("/findOrderByUserId/{userId}")
public List<Order> findOrderByUserId(@PathVariable("userId") Integer user
Id) {
// try {
// // 测试熔断，模拟超时
// Thread.sleep(2000);
// } catch (InterruptedException e) {
// e.printStackTrace();
// }

 //测试熔断，模拟服务异常
 if(userId == 2){
 throw new NullPointerException();
 }
 List<Order> list = orderService.findOrderByUserId(userId);
 return list;
}
```

2.5）属性配置

ribbon+hystrix超时问题：开启超时配置

```
#设置接口的超时时间（默认1秒） 参考： HystrixCommandProperties
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=
000
#指定特定接口的超时时间
hystrix.command.queryUserInfoFallBack.execution.isolation.thread
.timeoutInMilliseconds=3000
```

2.6）hystrix整合feign

启动类上配置feign的注解

```
@SpringBootApplication
@EnableCircuitBreaker
@EnableFeignClients(basePackages = "bat.ke.qq.com")
public class FeignHystrixConsumerApplication {

 public static void main(String[] args) {
 SpringApplication.run(FeignHystrixConsumerApplication.class,args);
 }
}
```

修改配置文件

```
#设置接口的超时时间（默认1秒）
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=000

#开启feign支持hystrix 默认是关闭的
feign.hystrix.enabled=true

# ribbon全局超时时间设置（使用feign需要设置）
ribbon.ReadTimeout=8000
ribbon.ConnectTimeout=2000
```

2.7）监控

2.7.1） 访问hystrix.stream的监控端点

引入依赖

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

加入监控端点配置

```
@Bean
public ServletRegistrationBean getServlet(){
 HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();
 ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);
 registrationBean.setLoadOnStartup(1);
 registrationBean.addUrlMappings("/hystrix.stream");
 registrationBean.setName("HystrixMetricsStreamServlet");
 return registrationBean;
}
```

测试，访问端点： http://localhost:8200/hystrix.stream

2.7.2）Hystrix+dashboard监控

引入依赖

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

启动类上添加@EnableHystrixDashboard注解，开启仪表盘功能

```
@SpringBootApplication
@EnableHystrixDashboard
public class ServiceHystrixDashboardApplication {

 public static void main(String[] args) {
 SpringApplication.run(ServiceHystrixDashboardApplication.class, args);
 }

}
```

配置属性

```
server.port=10000
spring.application.name=hystrix-dashboard
```

测试： http://localhost:10000/hystrix

输入http://localhost:8200/hystrix.stream

2.7.3）turbine集群监控

引入依赖

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

启动类上配置@EnableTurbine注解

```
@SpringBootApplication
@EnableTurbine
public class ServiceTurbineApplication {

 public static void main(String[] args) {
 SpringApplication.run(ServiceTurbineApplication.class, args);
 }

}
```

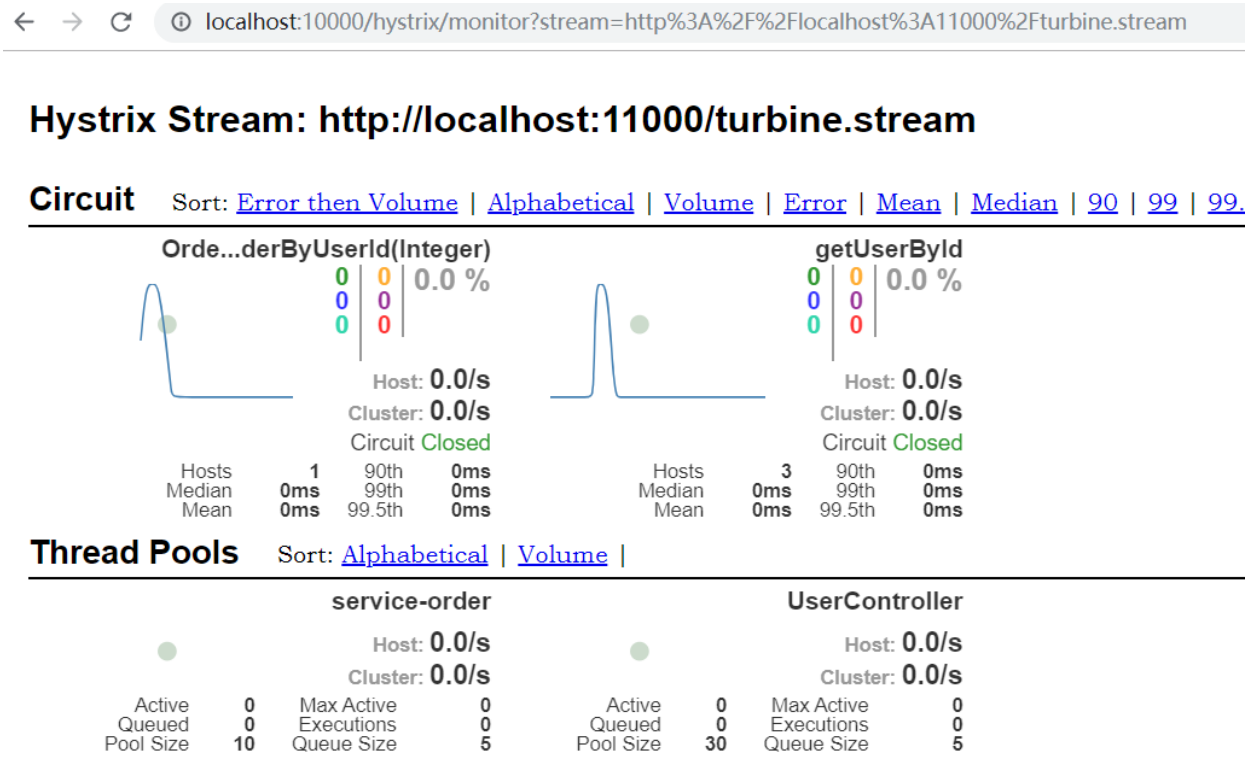
配置属性

```
server.port=11000

#注册到eureka服务端的微服务名称
spring.application.name=service-turbine

#暴露给其他eureka client 的注册地址
eureka.client.service-url.defaultZone=http://www.eureka8761.com:8761/eureka/
#将ip注册到Eureka Server上
```

```
eureka.instance.prefer-ip-address=true
#显示微服务的服务实例id
eureka.instance.instance-id=${spring.application.name}-${server.port}

# 指定聚合哪些集群，多个使用","分割，默认为default。
# 可使用http://.../turbine.stream?cluster={clusterConfig之一}访问
turbine.aggregator.cluster-config=default
#指定要监控的微服务名
turbine.app-config=service-hystrix-consumer,service-feign-hystrix-consumer
#集群的名字为default
turbine.cluster-name-expression="default"
#同一主机上的服务通过host和port的组合来进行区分
turbine.combine-host-port=true
# 默认path: /actuator/hystrix.stream
turbine.instanceUrlSuffix= /hystrix.stream
```

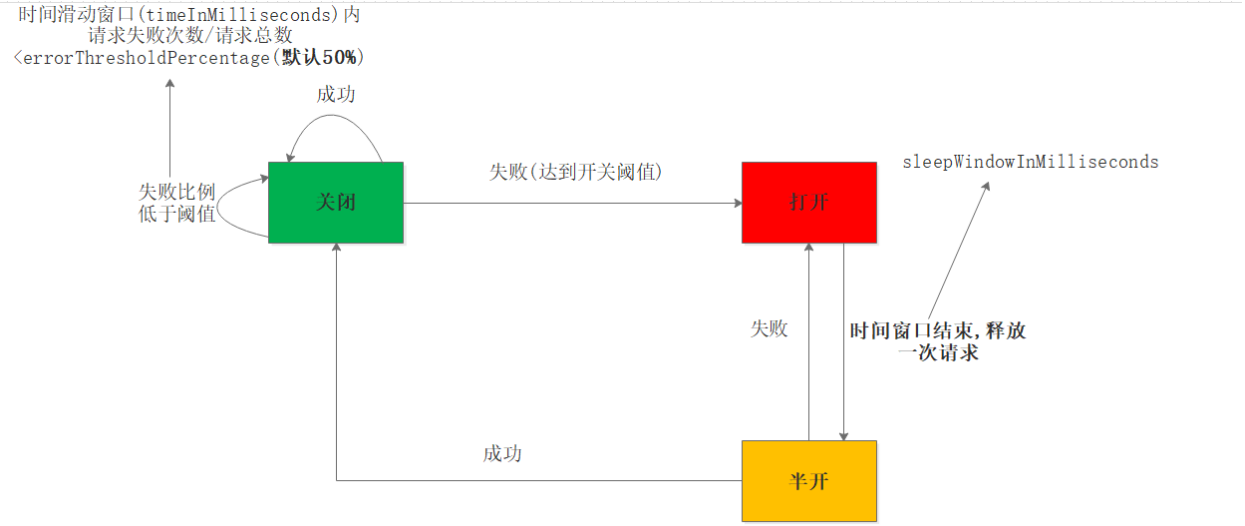
测试 http://localhost:10000/hystrix

输入http://localhost:11000/turbine.stream



3. Hystrix原理

3.1）熔断器跳闸机制三态转换图



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.metrics.rollingStats.timeInMilliseconds 设置统计的时间窗值，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

3.2） 测试熔断打开以及半开

```
# 查看详细的健康检查信息
management.endpoint.health.show-details=always
```

访问：http://localhost:8200/actuator/health

查看监控端点hystrix的值：

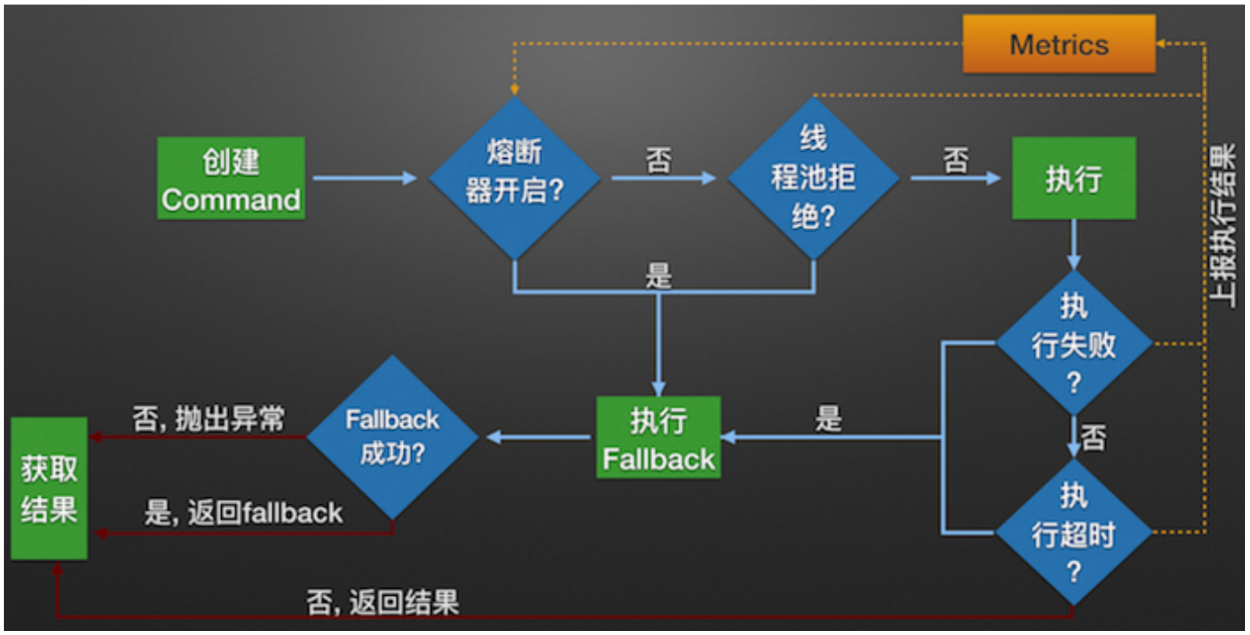
```
- hystrix: {
      status: "UP"
  }
```

在时间窗口内，连续点击失败次数,当此时达到设置的requestVolumeThreshold的阈值，就直接进入降级方法,此时再来看hystrix的监控信息：

```
- hystrix: {
    status: "CIRCUIT_OPEN",
  - details: {
      - openCircuitBreakers: [
            "UserController::getUserById"
        ]
    }
}
```

等到熔断器半开后，测试一个正确的查询，那么熔断器就会关闭，恢复正常调用

3.2 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.

3.3 Hystrix相关配置

hystrix.command.default和hystrix.threadpool.default中的default为默认
CommandKey
Command Properties

3.3.1）Execution相关的属性的配置：

1. hystrix.command.default.execution.isolation.strategy 隔离策略，默认是Thread, 可选Thread | Semaphore
2. hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 命令执行超时时
间，默认1000ms
3. hystrix.command.default.execution.timeout.enabled 执行是否启用超时，默认启用true
4. hystrix.command.default.execution.isolation.thread.interruptOnTimeout 发生超时是是否中断，默认true
5. 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thread。semaphore应该占整个容器（tomcat）的线程池的一小部分。

3.3.2） Fallback相关的属性

这些参数可以应用于Hystrix的THREAD和SEMAPHORE策略

1. hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequest

s 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

2. 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

3.3.3）Circuit Breaker相关的属性

1. 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

2. 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

3. 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

4. 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

5. 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

6. 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

3.3.4）Metrics相关参数

1. 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

2. 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

3. hystrix.command.default.metrics.rollingPercentile.enabled 执行时是否enable指标的计算和跟踪，默认true

4. hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds 设置rolling percentile window的时间，默认60000

5. hystrix.command.default.metrics.rollingPercentile.numBuckets 设置rolling percentile window的numberBuckets。逻辑同上。默认6

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

7. 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

3.3.5）Request Context 相关参数

1. hystrix.command.default.requestCache.enabled 默认true，需要重载getCacheKey()，返回null时不缓存

2. hystrix.command.default.requestLog.enabled 记录日志到HystrixRequestLog，默认true

3.3.6）Collapser Properties 相关参数

1. 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

2. 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

3. hystrix.collapser.default.requestCache.enabled 是否对HystrixCollapser.execute() and

HystrixCollapser.queue()的cache，默认true

3.3.7）ThreadPool 相关参数

1. 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，那有个基本的公式可以follow：requests per second at peak when healthy × 99th percentile latency in seconds + somebreathing 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）基本的原则是保持线程池尽可能小，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。

2. hystrix.threadpool.default.coreSize 并发执行的最大线程数，默认10

3. 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。

4. 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

5. 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.

6. hystrix.threadpool.default.metrics.rollingStats.timeInMilliseconds 线程池统计指标的时间，默

认10000

7. hystrix.threadpool.default.metrics.rollingStats.numBuckets 将rolling window划分为n个buckets，默认10