Spring Cloud Hystrix：服务容错保护

- 当某个服务单元发生故障之后，通过断路器的故障监控，向调用方返回一个错误响应，而不是长时间等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延
- 该框架的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具备服务降级、 服务熔断、 线程和信号隔离、 请求缓存、 请求合并以及服务监控等强大功能。

入门

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

- 在ribbon-consumer工程的主类上使用@EnableCircuitBreaker注解开启断路器功能:
  - @SpringCloudApplication 包含了 `@SpringBootApplication`、`@EnableDiscoveryClient`、`@EnableCircuitBreaker`
- 改造服务消费方式， 新增 HelloService 类， 注入 RestTemplate 实例
  - 在 ConsumerController 中对 RestTemplate 的使用迁移到 helloService 函数中
  - 在 helloService 函数上增加 @HystrixCornrnand 注解来指定回调方法

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

  @HystrixCommand(fallbackMethod = "helloFallback")
  public String helloService() {
    return restTemplate.getForEntity("http://HELLO-SERVICE/hello",
String.class) .getBody();
  }

  public String helloFallback () (
    return "error";
  }
}
```

- 修改 ConsumerController 类， 注入上面实现的 HelloService 实例， 并在helloConsumer 中进行调用

```
@RestController
public class ConsumerController {
    @Autowired
    private HelloService helloService;

    @RequestMapping(value = "/ribbon-consumer", method =
RequestMethod.GET)
    public String helloConsumer() {
        return helloService.helloService();
    }
}
```

- 访问`http://localhost:9000/ribbon-consumer`
- 模拟一下服务阻塞(长时间未响应)的情况。 对 HELLO-SERVICE 的/hello 接口做一些修改

```
@RequestMapping(value= "/hello", method= RequestMethod.GET)
public String hello() throws Exception {
  Serviceinstance instance= client.getLocalServiceinstance();
  //让处理线程等待几秒钟
  //  Hystrix 默认超时时间为 2000 毫秒
  int sleepTime= new Random().nextint(3000);
  logger.info("sleepTime:" + sleepTime); Thread.sleep{sleepTime);
  logger.info("/hello, host:" + instance.getHost() + ", service id:" +
instance.getServiceid());
  return "Hello World";
}
```

原理分析

- 1.创建HystrixCommand或HystrixObservableCommand对象
  - 首先，构建 一 个HystrixCommand或是HystrixObservableCommand对象，用来表示对依赖服务的操作请求， 同时传递所有需要的参数
    - HystrixCommand: 用在依赖的服务返回单个操作结果的时候
    - HystrixObservableCommand: 用在依赖的服务返回多个操作结果的时候
  - 命令模式， 将来自客户端的请求封装成一个对象， 从而让你可以使用不同的请求对客户端进行参数化。 它可以被用于实现 “ 行为请求者 ” 与 “ 行为实现者 ” 的解耦， 以便使两者可以适应变化

```
//接收者  Receiver: 接收者， 它知道如何处理具体的业务逻辑。
public class Receiver {
  public void action(){
    //真正的业务逻辑
  }
}

//抽象命令
```

```
//抽象命令， 它定义了 一 个命令对象 应具备的 一系列命令操作， 比如 execute()、 undo()、 redo()等。 当命令操作被调用的时候就会触发接收者去做具体命令对应的业务逻辑
interface Corrunand {
  void execute();
}

//具体命令实现
//具体的命令实现， 在这里它绑定了命令操作与接收者之间的关系， execute()命令的实现委托给了 Receiver的 ac巨on()函数。
public class ConcreteCorrunand implements Corrunand {
  private Receiver receiver;
  public ConcreteCorrunand(Receiver receiver){
    this.receiver = receiver;
    public void execute() {
      this.receiver.action();
  }
}

//客户端调用者
// 调用者， 它持有一个命令对象， 并且可以在需要的时候通过命令对象完 成具体的业务逻辑
public class Invoker {
  private Command command;
  public void setCommand(Command command) {
    this.command= command;
  }
  public void action(){
    this. command.execute();
  }
}

public class Client {
    public static void main(String[] args) {
      Receiver receiver = new Receiver();
      Command command = new ConcreteCommand(receiver);
      Invoker invoker = new Invoker();
      invoker.setCommand(command);
      invoker.ac巨on(); //客户端通过调用者来执行命令
    }
}
```

- 2.命令执行
  - Hystrix在执行 时 会根据创建的Command对象以及具体的情况来选择一个执行
    - HystrixComrnand实现了下面两个执行方式。
      - execute(): 同步执行，从依赖的服务返回一个单一的结果对象， 或是在发生错误的时候抛出异常
      - queue(): 异步执行，直接返回 一个Future对象， 其中包含了服务执行结束时要返回的单一结果对象

        - HystrixObservableCommand实现了另外两种 执行方式
            - observe(): 返回Observable对象，它代表了操作的多个结果，它是 一个Hot Observable。
            - toObservable(): 同样会返回Observable对象， 也代表了操作的多个结果， 但它返回的是 一个Cold Observable
- 3.结果是否被缓存
    - 若当前命令的请求缓存功能是被启用的， 并且该命令缓存命中， 那么缓存的结果会立即以Observable 对象的形式 返回
- 4.断路器是否打开
    - 在命令结果没有缓存命中的时候， Hystrix在执行命令前需要检查断路器是否为打开状态:
        - 如果断路器是打开的，那么Hystrix不会执行命令，而是转接到fallback处理逻辑(对应下面第8步)
        - 如果断路器是关闭的， 那么Hystrix跳到第5步， 检查是否有可用资源来 执行命令
- 5.线程池/请求队列/信号量是否占满
    - 如果与命令相关的线程池和请求队列，或者信号量(不使用线程池的时候)已经被占满，那么Hystrix也不会执行命令， 而是转接到fallback处理逻辑(对应下面第8步)。
        - 这里Hystrix所判断的线程池并非容器的线程池，而是每个依赖服务的专有线程池。Hystrix为了保证不会因为某个依赖服务的间题影响到其他依赖服务而采用了“舱壁模式"(Bulkhead Pattern)来 隔离每个依赖的服务
- 6.HystrixObservableCommand.construct()或HystrixCommand.run()
    - Hystrix会根据我们编写的方法来决定采取什么样的方式去请求依赖服务
        - HystrixCommand.run(): 返回一个单一 的结果，或者抛出异常
        - HystrixObservableCommand.construct(): 返回一个Observable对象来发射多个结果，或通过onError发送错误通知
        - 如果run()或construet()方法的执行时间超过了命令设置的超时阙值， 当前处理线程将会抛出 一个TimeoutException (如果该命令不在其自身的线程中执行， 则会通过单独的计时线程来 抛出)。在这种情况下，Hystrix会转接到fallback处理逻辑(第8步)。 同时，如果当前命令没有被取消或中断， 那么它最终会忽略run()或者construct ()方法的返回
        - 如果命令没有抛出异常并返回了结果，那么Hystrix在记录 一些日志并采集监控报告之后将该结果返回。 在使用 run()的情况下，Hystrix 会返回一 个Observable, 它发射单个结果并产生onCompleted的结束通知; 而在使用construct ()的情况下，Hystrix会直接返回该方法产生的Observable对象
- 7.计算断路器的健康度
    - Hystrix会将“成功”、 “失败”、 “拒绝”、 “ 超时” 等信息报告给断路器，而断路器会维护一组计数器来统计这些数据
    - 断路器会使用这些统计数据来决定是否要将断路器打开， 来对某个依赖服务的请求进 行 “熔断/短路”，直到恢复期结束。 若在恢复期结束后，根据统计数据判断如果还是未达到健康指标，就再次 “熔断/短路”
- 8.fallback处理
    - 当命令执行失败的时候， Hystrix会进入fallback尝试回退处理， 我们通常也称该操作 为 “ 服务降级”。而能够引起服务降级处理的清况有下面几种:
        - 第4步， 当前命令处于 “ 熔断l短路 ” 状态， 断路器是打开的时候
        - 第5步， 当前命令的线程池、 请求队列或 者信号量被占满的时候

        - 第6步，HysrixObservableCommandc.onstruct()或HystrixCommand.run() 抛出异常的时候
    - 在服务降级逻辑中， 我们需要实现一个通用的响应结果， 并且该结果的处理逻辑应当是从缓存或是根据一些静态逻辑来获取，而不是依赖网络请求获取。
        - 如果一定要在降级逻辑中包含网络请求，那么该请求也必须被包装在HystrixCommand或是HystrixObservableCommand中， 从而形成级联的降级策略， 而最终的降级逻辑一定不是一个依赖网络请求的处理， 而是一个能够稳定地返回结果的处理逻辑
        - 在HystrixCommand和HystrixObservableCommand中实现降级逻辑时还略有不同:
            - 当使用HystrixCommand的时候， 通过实现HystrixCommand.getFallback() 来实现服务降级逻辑
            - 当使用 HystrixObservableCommand 的时候， 通过 HystixObservable-Command.resumeWithFallback()实现服务降级逻辑， 该方法会返回一个Observable对象来发射 一个或多个降级结果
            - 当命令的降级逻辑返回结果之后， Hystrix 就将该结果返回给调用者。 当使用HystrixCommand.getFallback()的时候， 它会返回 一个Observable对象， 该对象会发射 getFallback()的处理结果 。 而使 用 HystrixObservableCommand.resumeWithFallback()实现的时候， 它会将Observable对象直接返回
    - 如果我们没有为命令实现降级逻辑或者降级处理逻辑中抛出了异常， Hystrix 依然会返回一个Observable对象， 但是它不会发射任何结果数据， 而是通过onError 方法通知命令立即中断请求，并通过onError()方法将引起命令失败的异常发送给调用者
    - 如果降级执行发现失败的时候，Hystrix会 根据不同的执行方法做出不同的处理
        - execute(): 抛出异常
        - queue(): 正常返回Future对象，但是当调用get()来获取结果的时候会抛出异常
        - observe(): 正常返回Observable 对象， 当订阅它的时候， 将立即通过调用订阅者的onError方法来通知中止请求
        - toObservable(): 正常返回Observable对象， 当订阅它的时候， 将通过调用订阅者的onError方法来通知中止请求
- 9.返回成功的响应
    - 当Hystrix命令执行成功之后， 它会将处理结果直接返回或是以Observable 的形式返回。 而具体以哪种方式返回取决于之前第2步中我们所提到的对命令的4种不同执行方式

断路器原理

- 断路器在 HystrixCommand 和 HystrixObservableCommand 执行过程中起到了举足轻重的作用，它是 Hystrix 的核心部件
- 断路器`HystrixCircuitBreaker`是如何决策熔断和记录信息
    - 三个抽象方法
        - allowRequest(): 每个 Hystrix 命令的请求都通过它判断是否被执行
        - isOpen(): 返回当前断路器是否打开
        - markSuccess(): 用来闭合断路器
    - 三个静态类

        - 静态类Factory中维护了一个Hystrix命令与HystrixCircuitBreaker的关系 集合。其中 String 类型的 key 通过 HystrixCommandKey 定义， 每一个 Hystrix 命令需要有一个 key 来标识， 同时一个 Hystrix 命令也会在该集合中 找到它对应的断路器 HystrixCircuitBreaker 实例
        - 静态类NoOpC江cuitBreaker定义了一个什么都不做的断路器实现，它允许所有请求，并且断路器状态始终闭合
        - 静态类HystrixC江cuitBreaker工mpl是断路器接口HystrixCircuitBreaker的实现类， 在该类中定义了断路器的 4 个核心对象
            - HystrixCommandProperties properties: 断路器对应 HystrixCommand实例的属性对象
            - HystrixCommandMetrics metrics: 用来让 HystrixCommand 记录各类度量指标的对象
            - AtomicBoolean circuitOpen: 断路器是否打开的标志， 默认为 false
            - AtomicLong cricuitOpenedOrLastTestedTime: 断路器打开或是上一次测试的时间戳
- HystrixCricuitBreakermpl 对 HystrixCricuitBreaker 接口的各个方法实 现如下所示。
    - isOpen (): 判断断路器的打开/关闭状态
        - 如果断路器打开标识为true, 则直接返回true, 表示断路器处千打开状态
        - 否则，就从度量指标对象 metrics 中获取 HealthCounts 统计对象做进 一 步判断(该 对象记录了一个滚动时间窗内的请求信息快照，默认时间窗为10秒)
            - 如果它的请求总数(QPS)在预设的阙值范围内就返回 false, 表示断路器处于未 打开状态。该阙值的配置参数为 circuitBreakerRequestVolumeThreshold, 默认值为 20。
            - 如果错误百分比在阑值范围内就返回 false, 表示断路器处于未打开状态。该 阙 值的配置参数为circuitBreakerErrorThresholdPercen七age, 默认值 为50 。
            - 如果上面的两个条件都不满足，则将断路器设置为打开状态 (熔断/短路)。 同 时，如果是从关闭状态切换到打开状态的话，就将当前时间记录到上面提到的 circuitOpenedOrLastTestedTirne 对象中。
    - allowReques七(): 判断请求是否被允许
        - 先根据配置对象properties中的断路器判断强制打开或关闭属性是否被设置。
        - 如果强制打开，就直接返回false, 拒绝请求。
        - 如果强制关闭，它会允许所有请求 ，但是同时也会调用 isOpen()来执行断路器的计算逻辑， 用来模拟断路器打开/关闭的行为
        - 在默认情况下，断路器并不会进入这两个强制打开或关闭的分支中去，而是通过! isOpen () I I allowSingleTest()来判断是否允许请求访问
            - allowSingleTest()：通过 circuitBreakerSleepWindowinMilliseconds 属性设置了一个断路器打开之后的休眠时间(默认为5秒)，在该休眠时间到达之后，将再次允许请求尝试访问，此时断路器处于 “ 半开 ” 状态， 若此时请求继续失败， 断路器又进入打开状态， 并继续等待下一个休眠窗口过去之后再次尝试;若请求成功， 则将断路器重新置于关闭状态。所以通过 allowSingleTest()与isOpen ()方法的配合，实现了断路器打开和关闭状态的切换
    - markSuccess(): 该函数用来在 “ 半开路 ” 状态时使用。若 Hystrix 命令调用成功，通过调用它将打开的断路器关闭， 并重置度量指标对象

依赖隔离

- Hystrix则使用舱壁模式实现线程池的隔离，它会为每一个依赖服务创建一个独立的线程池， 这样就算某个依赖服务出现延迟过高的情况， 也只是对该依赖服务的调用产生影响， 而不会拖慢其他的依赖服务
- 通过对依赖服务实现线程池隔离， 可让我们的应用更加健壮， 不会因为个别依赖服务出现问题而引起非相关服务的异常。 同时， 也使得我们的应用变得更加灵活， 可以在不停止服务的情况下，配合动态配置刷新实现性能配置上的调整
- 性能影响测试
    - NetflixHystrix官方提供的一个Hystrix命令的性能监控图， 该命令以每秒60个请求的速度(QPS) 对一个单服务实例进行访问， 该服务实例每秒运行的线程数峰值为350个
    - 在 99%的情况下， 使用线程池隔离的延迟有9ms, 对千大多数需求来说这样的消耗是微乎其微的， 更何况可为系统在稳定性和灵活性上带来巨大的提升
    - 9ms 的延迟开销非常昂贵的话：信号量
        - 可以使用信号量来控制单个依赖服务的并发度， 信号量的开销远比线程池的开销小， 但是它不能设置超时和实现异步访问。 所以， 只有在依 赖 服 务是足够可靠的情况下才使用信号量
        - 在 HystrixCommand和HystrixObservableCommand中有两处支持信号量的使用。
            - 命令执行:如果将隔离策略参数execu巨on.isolation.strategy设置为SEMAPHORE' Hysn·ix 会使用信号量替代线程池来控制依赖服务的并发
            - 降级逻辑:当 Hystrix 尝试降级逻辑时， 它会在调用线程中使用信号量。
            - 信号量的默认值为10
                - 可以通过动态刷新配置的方式来控制并发线程的数量。 对于信号量大小的估算方法与线程池并发度的估算类似。 仅访问内存数据的请求一般耗时 在 lms 以内，性能可以达到 5000rps （rps 指每秒的请求数)，这样级别的请求可以将信号量置为 1 或者 2, 我们可以按此标准并根据实际请求耗时来设置信号量

使用详解

- 创建请求命令
    - Hystrix 命令就是HystrixCommand, 它用来封装具体的依赖服务调
    - 继承方式实现
    - 注解方式实现：方法上标注@HystrixCommand注解
        - 一般是同步执行实现，还有异步、响应式实现
- 定义服务降级
    - fallback是Hystrix命令执行失败时使用的后备方法， 用来实现服务的降级处理逻辑
    - 注解方式
        - 只需要使用@HystrixCommand 中的 fallbackMethod参数来指定具体的服务降级实现方法
        - 需要将具体的 Hystrix 命令与 fallback 实现函数定义在同 一 个类中， 并且

fallbackMethod 的值必须与实现 fallback 方法的名字相同。 由于必须定义在一个类中， 所以对于 fallback 的访问修饰符没有特定的要求， 定义为private、 protected、 public 均可

    - 在实际使用时，我们需要为大多数执行过程中可能会失败的Hystrix命令实现服务降级逻辑，但是也有一些情况可以不去实现降级逻辑
        - 执行写操作的命令：当Hystrix命令是用来执行写操作而不是返回一些信息的时候，通常情况下这类操作的返回类型是 void 或是为空的 Observable, 实现服务降级的意义不是很大。 当写入操作失败的时候， 我们通常只需要通知调用者即可。
        - 执行批处理或离线计算的命令：当Hystrix命令是用来执行批处理程序生成一份报告或是进行任何类型的离线计算时， 那么通常这些操作只需要将错误传播给调用者，然后让调用者稍后重试而不是发送给调用者 一 个静默的降级处理响应。
- 异常处理
    - 注解方式，只需要在 fallback 实现方法的参数中增加 Throwable e 对象的定义， 这样在方法内部就可以获取触发服务降级的具体异常内容了
- 命令名称、分组以及线程池划分
    - 通过设置命令组，Hystrix会根据组来组织和统计命令的告警、 仪表盘等信息。
    - 那么为什么一定要设置命令组呢?因为除了根据组能实现统计之外， Hystrix 命令默认的线程划分也是根据命令分组来实现的。默认情况下，Hystrix 会让相同组名的命令使用同一个线程池，所以我们需要在创建 Hystrix 命令时为其指定命令组名来实现默认的线程池划分
    - 如果 Hystrix 的线程池分配仅仅依靠命令组来划分， 那么它就显得不够灵活了， 所以Hystrix 还提供了 HystrixThreadPoolKey 来对线程池进行设置， 通过它我们可以实现更细粒度的线程池划分
    - 注解方式划分，只需设置@HystrixCommand 注解的 commandKey、 groupKey 以及 threadPoolKey 属性即可， 它们分别表示了命令名称、 分组以及线程池划分
- 请求缓存
    - 在高并发的场景之下， Hystrix 中提供了请求缓存的功能， 我们可以方便地开启和使用请求缓存来优化系统， 达到减轻高并发时的请求线程消耗、 降低请求响应时间的效果
    - 尝试获取请求缓存以及将请求结果加入缓存
    - 注解方式缓存
        - 设置请求缓存
            - 只需方法上添加@CacheResult 注解即可，缓存 Key 值会使用所有的参数
        - 定义缓存Key
            - 使用@CacheKey 注解在方法参数中指定用于组装缓存 Key 的元素，它的优先级比 cacheKeyMethod 的优先级低`(@CacheKey("id") Long id)`。 它还允许访问参数对象 的内部属性作为缓存 Key`(@CacheKey("id") User user)`
            - 或者使用@CacheResult和@CacheRemove注解的cacheKeyMethod 方法来指定具体的生成函数`@CacheResult(cacheKeyMethod = "getUserByidCacheKey")`,`private Long getUserByidCacheKey(Long id) {return id;}`
        - 缓存清理
            - 通过@CacheRemove 注解来实现失效缓存的清理
            - @CacheRemove 注解的 commandKey 属性是必须要指定的（设置缓存方法的名称）， 它用来指明需要使用请求缓存的请求命令， 因为只有通过该属性的配置，

Hystrix 才能找到正确的请求命令缓存位置

- 请求合并
  - Hystrix 提供了 HystrixCollapser 来实现请求的合并， 以减少通信消耗和线程数的占用
  - 注解方式

```
@Service
public class UserService {
  @Autowired
  private RestTemplate restTemplate;
  //通过batchMethod 属性指定了批量请求的实现方法为巨ndAll 方法, 同时通过
collapserProperties属性为合并请求器 设置了相关属性,将合并时间窗设置为100毫秒
  @HystrixCollapser(batchMethod= "findAll", collapserProperties= {
    @HystrixProperty(name="timerDelayinMilliseconds", value = "100")
  })
  public User find(Long id){
    return null;
  }

  @HystrixCommand
  public List<User> fndAll (List<Long> ids) {
    return restTemplate.getForObject("http://USER-SERVICE/users?ids
{l}",
List. class, StringUtils.join(ids, ", "));
  }
}
```

  - 是否使用请求合并器需要 根据依赖服务调用的实际情况来选择， 主要考虑下面两个方面。
    - 请求命令本身的延迟。 如果依赖服务的请求命令本身是一个高延迟的命令， 那么可以使用请求合并器， 因为延迟时间窗的时间消耗显得微不足道了
    - 延迟时间窗内的并发量。 如果一个时间窗内只有1-2个请求， 那么这样的依赖服务不适合使用请求合并器。这种情况不但不能提升系统性能， 反而会成为系统瓶颈，因为每个请求都需要多消耗一个时间窗才响应。 相反， 如果一个时间窗内具有很高的并发量，并且服务提供方也实现了批量处理接口， 那么使用请求合并器可以有效减少网络连接数量并极大提升系统吞吐量， 此时延迟时间窗所增加的消耗就可以忽略不计了。

属性详解

- 当通过注解的方法实现时， 只需使用 @HystrixCommand 中的 commandProperties 属性来设置

```
@HystrixConunand(conunandKey = "helloKey", comnandProperties = {

@HystrixProperty(name="execution.is0lation.thread.timeoutinMilliseconds",
value = "5000")})
public User getUserByid(Long id) {
  return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
}
```

- 配置优先级别（低-高）
  - 全局默认值，该属性通过代码定义， 所以对于这个级别，主要关注它在代码中定义的默认值即可
  - 全局配置属性，通过在配置文件中定义全局属性值，在应用启动时或在与 SpringCloud Config 和 Spring Cloud Bus 实现的动态刷新配置功能配合下， 可以实现对 “ 全局默认值” 的覆盖以及在运行期对 “全局默认值” 的动态调整
  - 实例默认值，通过代码为实例定义的默认值。 通过代码的方式为实例设置属性值来覆盖默认的全局配置
  - 实例配置属性，通过配置文件来为指定的实例进行属性配置， 以覆盖前面的三个默 认值。 它也可用Spring Cloud Config和Spring Cloud Bus实现的动态刷新配置功能实现对具体实例配置的动态调整
- Command属性
  - 主要用来控制HystrixCommand命令的行为，它主要有5种不同类型的属性配置。
    - execution配置，控制的是HystrixCommand.run()的执行
      - execution.isolation .strategy: 该属性用来设置Hys七rixCommand.run()执行的隔离策略
        - THREAD: 通过线程池隔离的策略。 它在独立的线程上执行， 并且它的并发限制受线程池中线程数量的限制
        - SEMAPHORE: 通过信号量隔离的策略。 它在调用线程上执行， 并且它的并发限制受信号量计数的限制
      - execution.isolation. thread.timeoutinMilliseconds: 该属性用来配置HystrixCommand执行的超时时间（毫秒）。 当HystrixCommand执行时间超过该配置值之后， Hystrix会将该执行命令标记为TIMEOUT并进入服务降级处理逻辑。
      - execution.timeout.enabled: 该属性用来配置HystrixCommand.run()的执行是否启用超时时间
      - execution.isolation.thread.interruptOnTimeout: 该属性用来配置当HystrixCommand.run()执行超时的时候是否要将它中断
      - execution.isolation. thread.interruptOnCancel: 该属性用来配置当HystrixCommand.run()执行被取消的时候是否要将它中断
      - execution.isolation.semaphore.maxConcurrentRequests: 当HystrixCommand的隔离策略使用信号量的时候， 该属性用来配置信号量的大小(并发请求数)。 当最大并发请求数达到该设置值时， 后续的请求将会被拒绝
- fallback配置

    - 控制HystrixComrnand.getFallback ()的执行。 这些属性同时适用于线程池的信号量的隔离策略
        - fallback.isolation.semaphore.maxConcurrentRequests: 该属性用来设置从调用线程中允许HystrixComrnand.get Fallback()方法执行的最大并发请求数。 当达到最大并发请求数时， 后续的请求将会被拒绝并抛出异常(因为它已经没有后续的fallback可以被调用了)
        - fallback.enabled: 该属性用来设置服务降级策略是否启用， 如果设置为false, 那么当请求失败或者拒绝发生时， 将不会调用HystrixComrnand.getFallback () 来执行服务降级逻辑
- circuitBreaker配置
    - 用来控制HystrixCircuitBreaker的行为。
        - circuitBreaker.enabled: 该属性用来确定当服务请求命令失败时， 是否使用断路器来跟踪其健康指标和熔断请求
        - circuitBreaker .requestVolumeThreshold: 该属性用来设置在滚动时间窗中，断路器熔断的最小请求数。 例如，默认该值为 20 的时候，如果滚动时间窗(默 认10秒)内仅收到了19个请求， 即使这19个请求都失败了，断路器也不会打开。
        - circuitBreaker.sleepWindowinMilliseconds: 该属性用来设置当断路器打开之后的休眠时间窗。 休眠时间窗结束之后，会将断路器置为 “ 半开 ” 状态， 尝试熔断的请求命令，如果依然失败就将断路器继续设置为 “ 打开 ” 状态，如果成功就设置为“关闭” 状态。
        - circuitBreaker.errorThresholdPercentage: 该属性用来设置断路器打开的错误百分比条件。 例如，默认值为 5000 的情况下，表示在滚动时间窗中，在请求 数量超过circuitBreaker.requestVolumeThreshold阅值的前提下，如果 错误请求数的百分比超过50, 就把断路器设置为 “ 打开 ” 状态， 否则就设置为 “ 关 闭” 状态。
        - circui七Breaker.forceOpen: 如果将该属性设置为 true, 断路器将强制进入 “ 打开 ” 状态， 它会拒绝所有请求。 该属性优先于 circuitBreaker.forceClosed属性
        - circuitBreaker.forceClosed: 如果将该属性设置为 true, 断路器将强制进入 “ 关闭 ” 状态， 它会接收所有请求。 如果 c江cuitBreaker.forceOpen 属性为true, 该属性不会生效
- metrics配置
    - 执行中捕获的指标信息有关
        - metrics.rollingStats.timeinMillseconds: 该属性用来设置滚动时间窗的长度， 单位为毫秒。 该时间用于断路器判断健康度时需要收集信息的持续时间。断路器在收集指标信息的时候会根据设置的时间窗长度拆分成多个 “ 桶 ” 来累计各度量值， 每个 “ 桶 ” 记录了一 段时间内的采集指标。 例如， 当采用默认值10000毫秒时， 断路器默认将其拆分成10个桶(桶的数量也可通过metrics.rollingStats.numBuckets参数设置)， 每个桶记录1000毫秒内的指标信息
        - metrics.rollingstats.numBuckets: 该属性用来设置滚动时间窗统计指标信息时划分 “ 桶 ” 的数量
        - metrics.rollingPercentile. enabled: 该属性用来设置对命令执行的延迟是否使用百分位数来跟踪和计算。 如果设置为false, 那么所有的概要统计都将返回 -1
        - metrixcs.rollingPercentile.timeinMilliseconds: 该属性用来设置百分位统计的滚动窗口的持续时间， 单位为毫秒
        - metrics.rollingPercentile.numBuckets: 该属性用来设置百分位统计滚动窗口中使用 “ 桶 ”的数量。
        - metrics.rollingPercentile.bucketSize: 该属性用来设置在执行过程中 每个 “ 桶 ” 中保留的最大执行次数。 如果在滚动时间窗内发生超过该设定值的执行次数， 就从最初的位置

开始重写。 例如， 将该值设置为100, 滚动窗口为10秒， 若 在10秒内一个 “ 桶 ” 中发生了500次执行， 那么该 “ 桶 ” 中只保留 最后的100次执 行的统计。 另外， 增加该值的大小将会增加内存量的消耗， 并增加排序百分位数所 需的计算时间

        - metrics.healthSnapshot.intervalinMilliseconds: 该属性用来设置采集影响断路器状态的健康快照(请求的成功、 错误百分比)的间隔等待时间

- requestContext配置
    - requestCache.enabled: 此属性用来配置是否开启请求缓存
    - requestLog.enabled: 该属性用来设置Hys立ixCommand的执行和事件是否打印日志到HystrixRequestLog中
- collapser属性
    - 属性除了在代码中用set和配置文件配置之外， 也可使用注解进行配置。 可使用@HystrixCollapser中的collapserProperties属性来设置

```
@HystrixCollapser(batchMethod = "batch", collapserProperties = {
@HystrixProperty(name="timerDelayinMilliseconds", value = "20")
))
```

    - 用来控制命令合并相关的行为
        - maxRequestsinBatch: 该参数用来设置 一 次请求合并批处理中允许的最大请求数
        - timerDelayinMilliseconds: 该参数用来设置批处理过程中每个命令延迟的时间， 单位为毫秒
        - request Cache.enabled: 该参数用来设置批处理过程中是否开启请求缓存。
- thread Pool属性
    - 该属性除了在代码中用set和配置文件配置之外，还可使用注解进行配置。可使用@HystrixCommand中的threadPoolProperties属性来设置

```
@HystrixCommand(fallbackMethod= "helloFallback", commandKey= "helloKey",
threadPoolProperties = {
        @HystrixProperty(name="coreSize", value = "20")
})
```

    - 控制Hystrix 命令所属线程池的配置
        - coreSize: 该参数用来设置执行命令线程池的核心线程数，该值也就是命令执行的最大并发量
        - maxQueueSize: 该参数用来设置线程池的最大队列大小。当设置为-1时，线程池将使用SynchronousQueue实现的队列，否则将使用LinkedBlockingQueue实现的队列
        - queueSizeRejection Threshold: 该参数用来为队列设置拒绝阀值。 通过该参数， 即使队列没有达到最大值也能拒绝请求。 该参数主要是对归nkedBlockingQueue 队列的补充， 因为LinkedBlockingQueue 队列不能动态修改它的对象大小， 而通过该属性就可以调整拒绝请求的队列大小了。当 maxQueueSize 属性为-1 的时候， 该属性不会生效
        - metrics.rollingstats.timeinMilliseconds: 该参数用来设置滚动时间窗的长度， 单位为毫秒。 该滚动时间窗的长度用于线程池的指标度量， 它会被分成多个 “桶” 来统计指标
        - metrics.rollingStats.numBuckets: 该参数用来设置滚动时间窗被划分成“桶” 的数量

        - metrics.rollingStats.七irne工nM过江seconds 参数的设置必须能够被rnetrics.rolling-Stats.nurnBuckets参数整除，不然将会抛出异常

Hystrix仪表盘

- 主要用来 实时监控Hystrix的各项指标信息。通过Hystrix Dashboard反馈的实时信息，可以帮助我们快速发现系统中存在的问题，从而及时的采取对应的措施

```
<dependency>
  <groupid>org.springframework.cloud</groupid>
  <artifactid>spring-cloud-starter-hystrix</artifactid>
</dependency>
<dependency>
        <groupid>org.springframework.cloud</groupid>
  <artifactid>spring-cloud-starter-hystrix-dashboard</artifactid>
</dependency>
<dependency>
        <groupid>org.springframework.boot</groupid>
        <artifactid>spring-boot-starter-actuator</artifactid>
</dependency>
```

- 为应用主类加上@EnableHystrixDashboard, 启用 Hystrix Dashboard 功能。
- application.properties 配置文件-应用名称、端口（2001）
- 启动，访问http://localhost:2001/hystrix
- Hystrix Dashboard共支持三种不同的监控方式
    - 默认的集群监控、指定的集群监控、单体应用监控
- 为 RIBBON-CONSUMER 加入下面的配置之后重启它的实例，访问 `http://localhost:9000/hystrix.stream`
    - 新增 spring-boot-starter.actuator 监控模块以开启监控相关的端点， 并确保已经引入断路器的依赖 spring-cloud-starter-hystrix
    - 主类中已经使用@EnableC江cuitBreaker 注解， 开启了断路器功能
- 监控页面
    - 实心圆：其有两种含义。 通过颜色的变化代表了实例的健康程度，它的健康度从绿色、 黄色、 橙色、 红色递减。 该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化， 流量越大该实心圆就越大
    - 曲线：用来记录2分钟内流量的相对变化， 可以通过它来观察流量的上升和下降趋势