微服务保护

解决雪崩问题

- 超时处理：设定超时时间，请求超过一定时间没有响应就会返回错误信息，不会无休止等待
- 舱壁模式：限定每个业务能使用的线程数，避免耗尽整个tomcat的资源，也叫线程隔离
- 熔断降级：由断路器统计业务执行的异常比例，如果超出阈值则会熔断业务，拦截访问该业务
- 流量控制：限制业务访问的QPS，避免服务因流量的徒增而故障（预防）

服务保护技术对比

	Sentinel	Hystrix
隔离策略	信号量隔离	线程池隔离/信号量隔离
熔断降级策略	基于慢调用比例或异常比例	基于失败比率
实时指标实现	滑动窗口	滑动窗口（基于 RxJava）
规则配置	支持多种数据源	支持多种数据源
扩展性	多个扩展点	插件的形式
基于注解的支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持
流量整形	支持慢启动、匀速排队模式	不支持
系统自适应保护	支持	不支持
控制台	开箱即用，可配置规则、查看秒级监控、机器发现等	不完善
常见框架的适配	Servlet、Spring Cloud、Dubbo、gRPC 等	Servlet、Spring Cloud Netflix

Sentinel 流控模式

在添加限流规则时，点击高级选项，可以选择三种流控模式：

- 直接：统计当前资源的请求，触发阈值时对当前资源直接限流，也是默认的模式
- 关联：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流
- 链路：统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流

- 关联模式：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流
- 使用场景：比如用户支付时需要修改订单状态，同时用户要查询订单。查询和修改操作会争抢数据库锁，产生竞争。业务需求是有限支付和更新订单的业务，因此当修改订单业务触发阈值时，需要对查询订单业务限流。

链路模式：只针对从指定链路访问到本资源的请求做统计，判断是否超过阈值。

- Sentinel默认只标记Controller中的方法为资源，如果要标记其它方法，需要利用@SentinelResource注解，示例：

```
@SentinelResource("goods")
public void queryGoods() {
    System.err.println("查询商品");
}
```

- Sentinel默认会将Controller方法做context整合，导致链路模式的流控失效，需要修改application.yml，添加配置：

```
spring:
  cloud:
    sentinel:
      web-context-unify: false # 关闭context整合
```

Sentinel 流控效果

流控效果是指请求达到流控阈值时应该采取的措施，包括三种：

- 快速失败：达到阈值后，新的请求会被立即拒绝并抛出FlowException异常。是默认的处理方式。
- warm up：预热模式，对超出阈值的请求同样是拒绝并抛出异常。但这种模式阈值会动态变化，从一个较小值逐渐增加到最大阈值。
- 排队等待：让所有的请求按照先后次序排队执行，两个请求的间隔不能小于指定时长

warm up也叫预热模式，是应对服务冷启动的一种方案。请求阈值初始值是 threshold / coldFactor，持续指定时长后，逐渐提高到threshold值。而coldFactor的默认值是3.

例如，我设置QPS的threshold为10，预热时间为5秒，那么初始阈值就是 10 / 3 ，也就是3，然后在5秒后逐渐增长到10.

当请求超过QPS阈值时，快速失败和warm up 会拒绝新的请求并抛出异常。而排队等待则是让所有请求进入一个队列中，然后按照阈值允许的时间间隔依次执行。后来的请求必须等待前面执行完成，如果请求预期的等待时间超出最大时长，则会被拒绝。

例如：QPS = 5，意味着每200ms处理一个队列中的请求；timeout = 2000，意味着预期等待超过2000ms的请求会被拒绝并抛出异常

热点参数限流

之前的限流是统计访问某个资源的所有请求，判断是否超过QPS阈值。而热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值。

配置示例：

代表的含义是：对hot这个资源的0号参数（第一个参数）做统计，每1秒**相同参数值**的请求数不能超过5

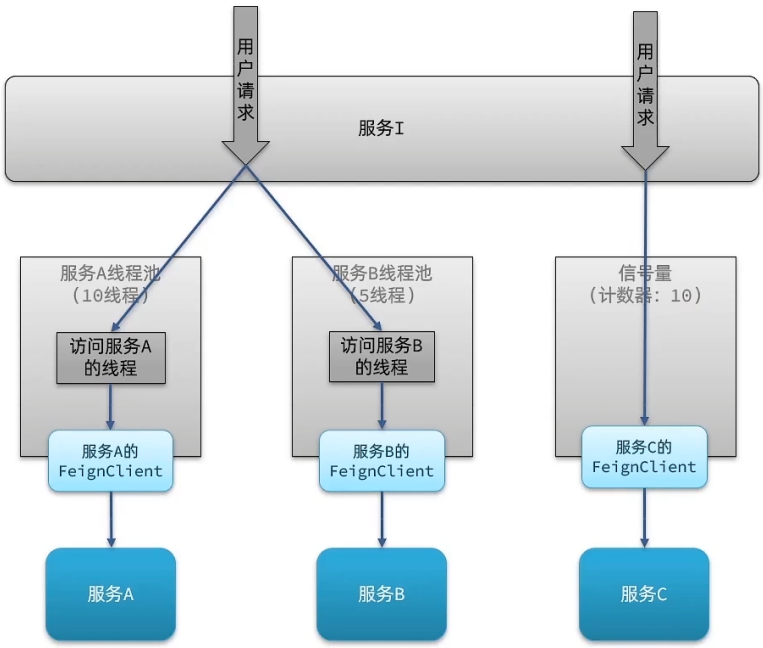
> 热点参数限流对默认的Spring MVC的资源无效

虽然限流可以尽量避免因高并发而引起的服务故障，但服务还会因为其它原因而故障。而要将这些故障控制在一定范围，避免雪崩，就要靠线程隔离（舱壁模式）和熔断降级手段了。

线程隔离

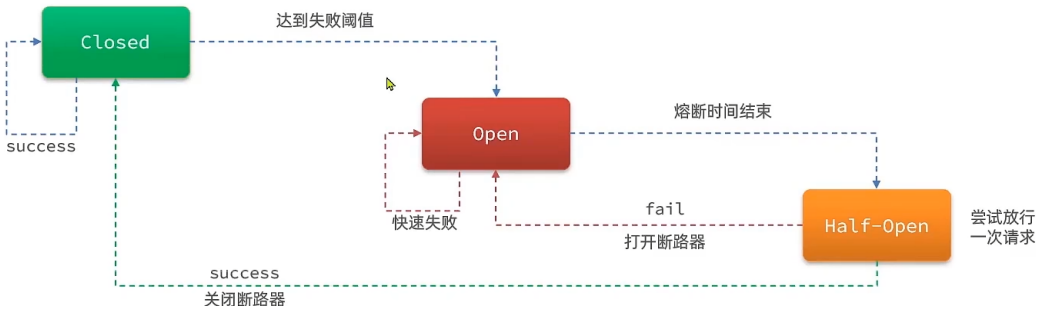
线程隔离有两种方式实现：

- 线程池隔离
- 信号量隔离（Sentinel默认采用）



- QPS：就是每秒的请求数，在快速入门中已经演示过
- 线程数：是该资源能使用用的tomcat线程数的最大值。也就是通过限制线程数量，实现舱壁模式。

熔断降级是解决雪崩问题的重要手段。其思路是由**断路器**统计服务调用的异常比例、慢请求比例，如果超出阈值则会**熔断**该服务。即拦截访问该服务的一切请求；而当服务恢复时，断路器会放行访问该服务的请求。



断路器熔断策略有三种：慢调用、异常比例、异常数

- 慢调用：业务的响应时长（RT）大于指定时长的请求认定为慢调用请求。在指定时间内，如果请求数量超过设定的最小数量，慢调用比例大于设定的阈值，则触发熔断。例如：

解读：RT超过500ms的调用是慢调用，统计最近10000ms内的请求，如果请求量超过10次，并且慢调用比例不低于0.5，则触发熔断，熔断时长为5秒。然后进入half-open状态，放行一次请求做测试。

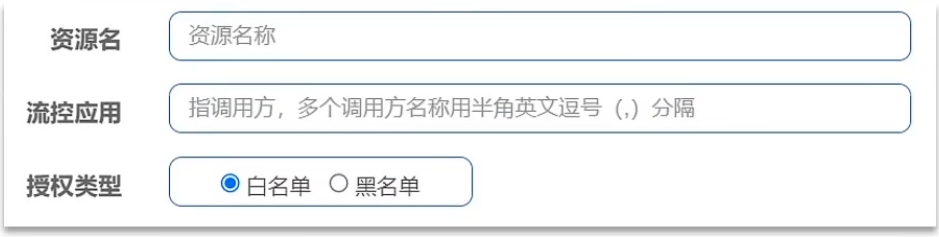
断路器熔断策略有三种：慢调用、异常比例或异常数

- 异常比例或异常数：统计指定时间内的调用，如果调用次数超过指定请求数，并且出现异常的比例达到设定的比例阈值（或超过指定异常数），则触发熔断。例如：

解读：统计最近1000ms内的请求，如果请求量超过10次，并且异常比例不低于0.5，则触发熔断，熔断时长为5秒。然后进入half-open状态，放行一次请求做测试。

授权规则可以对调用方的来源做控制，有白名单和黑名单两种方式。

- 白名单：来源（origin）在白名单内的调用者允许访问
- 黑名单：来源（origin）在黑名单内的调用者不允许访问



Sentinel是通过RequestOriginParser这个接口的parseOrigin来获取请求的来源的。

```
public interface RequestOriginParser {

    /**
     * 从请求request对象中获取origin，获取方式自定义
     */
    String parseOrigin(HttpServletRequest request);
}
```

例如，我们尝试从request中获取一个名为origin的请求头，作为origin的值：

```
@Component
public class HeaderOriginParser implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest request) {
        String origin = request.getHeader("origin");
        if(StringUtils.isEmpty(origin)){
            return "blank";
        }
        return origin;
    }
}
```

我们还需要在gateway服务中，利用网关的过滤器添加名为gateway的origin头：

```
spring:
  cloud:
    gateway:
      default-filters:
        - AddRequestHeader=origin,gateway # 添加名为origin的请求头，值为gateway
```

给/order/{orderId} 配置授权规则：

自定义异常结果

默认情况下，发生限流、降级、授权拦截时，都会抛出异常到调用方。如果要自定义异常时的返回结果，需要实现BlockExceptionHandler接口：

```
public interface BlockExceptionHandler {

    /**
     * 处理请求被限流、降级、授权拦截时抛出的异常：BlockException
     */
    void handle(HttpServletRequest request, HttpServletResponse response, BlockException e) throws Exception;

}
```

而BlockException包含很多个子类，分别对应不同的场景：

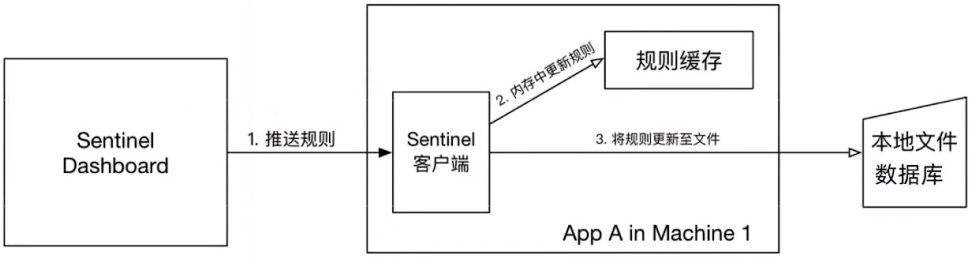
异常	说明
FlowException	限流异常
ParamFlowException	热点参数限流的异常
DegradeException	降级异常
AuthorityException	授权规则异常
SystemBlockException	系统规则异常

```
@Component
public class SentinelBlockHandler implements BlockExceptionHandler {
    @Override
    public void handle(
            HttpServletRequest httpServletRequest,
            HttpServletResponse httpServletResponse, BlockException e) throws Exception {
        String msg = "未知异常";
        int status = 429;
        if (e instanceof FlowException) {
            msg = "请求被限流了！";
        } else if (e instanceof DegradeException) {
            msg = "请求被降级了！";
        } else if (e instanceof ParamFlowException) {
            msg = "热点参数限流！";
        } else if (e instanceof AuthorityException) {
            msg = "请求没有权限！";
            status = 401;
        }
        httpServletResponse.setContentType("application/json;charset=utf-8");
        httpServletResponse.setStatus(status);
        httpServletResponse.getWriter().println("{\"message\": \"" + msg + "\", \"status\": " + status + "}");
    }
}
```

Sentinel的控制台规则管理有三种模式：

- 原始模式：Sentinel的默认模式，将规则保存在内存，重启服务会丢失。

pull模式：控制台将配置的规则推送到Sentinel客户端，而客户端会将配置规则保存在本地文件或数据库中。以后会定时去本地文件或数据库中查询，更新本地规则。



push模式：控制台将配置规则推送到远程配置中心，例如Nacos。Sentinel客户端监听Nacos，获取配置变更的推送消息，完成本地配置更新。