• Alibaba微服务组件Sentinel

1、分布式系统遇到的问题

服务的可用性问题

当服务

我们首先

激增流量
打垮

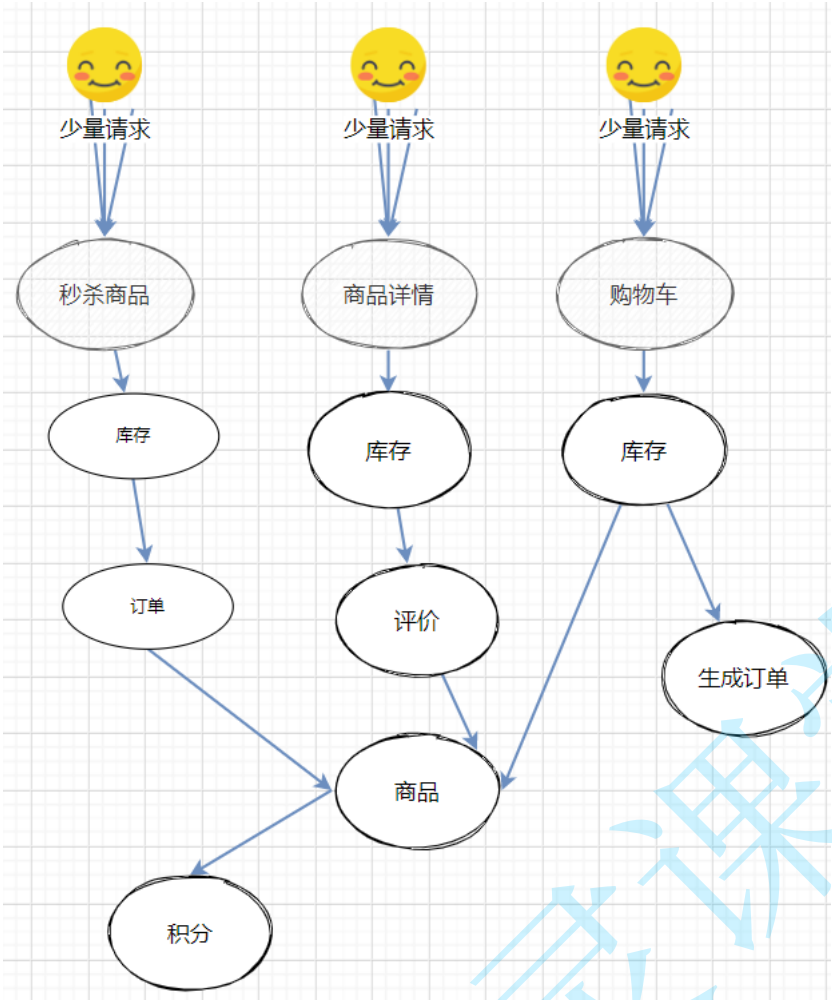
缺乏高可

尤其是

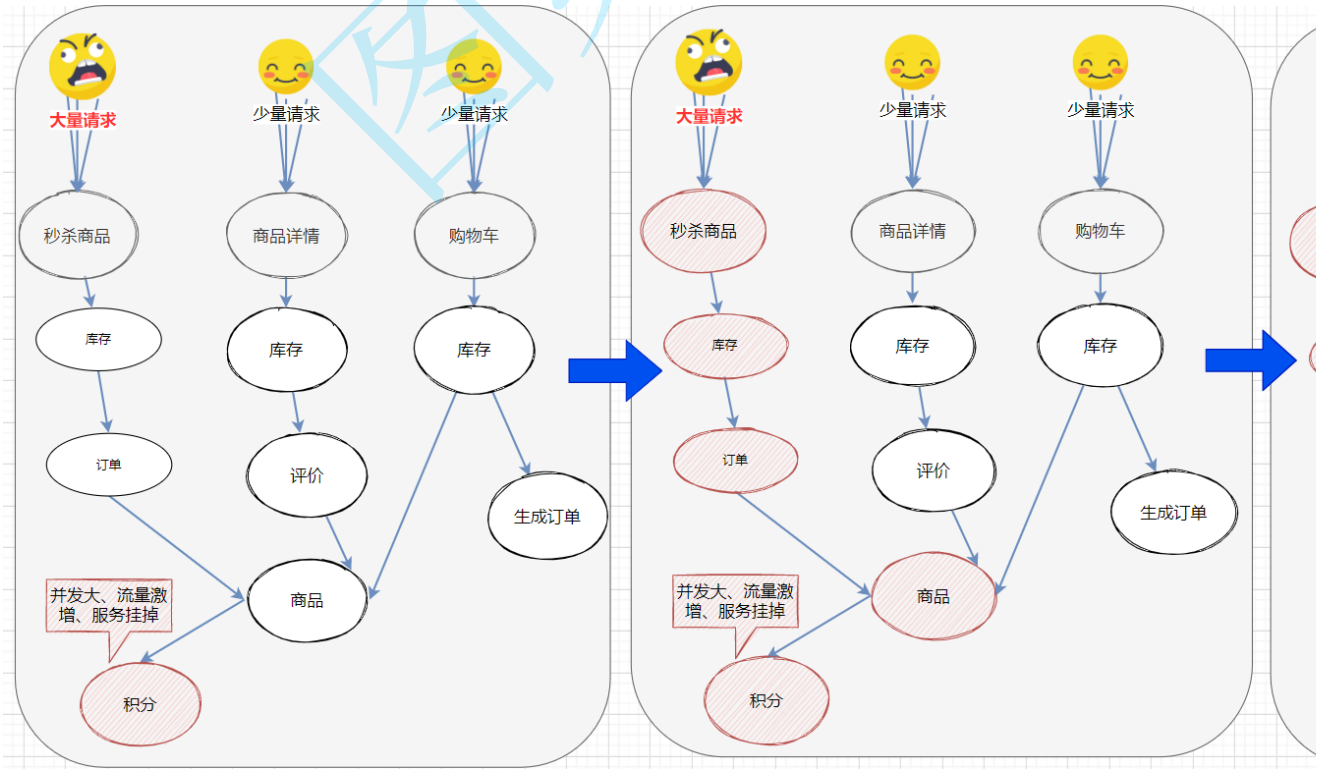
4个9 - 52.6 分钟　　5个9 - 5 分钟

服务的可用性场景

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, 如图所示:

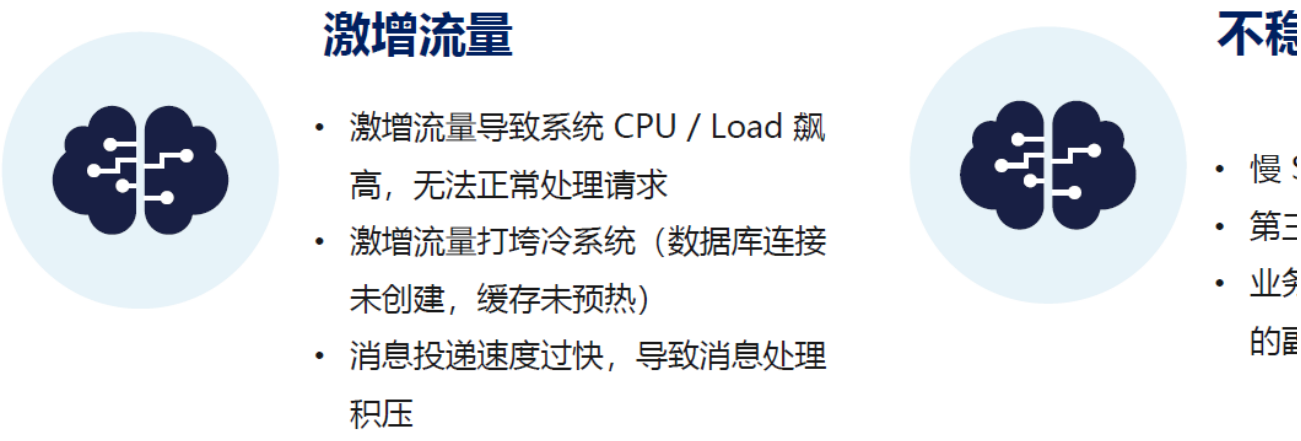


如果其中的下单服务不可用, 就会出现线程池里所有线程都因等待响应而被阻塞, 从而造成整个服务链路不可用， 进而导致整个系统的服务雪崩. 如图所示:



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：



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。

2、解决方案

稳定性、恢复性

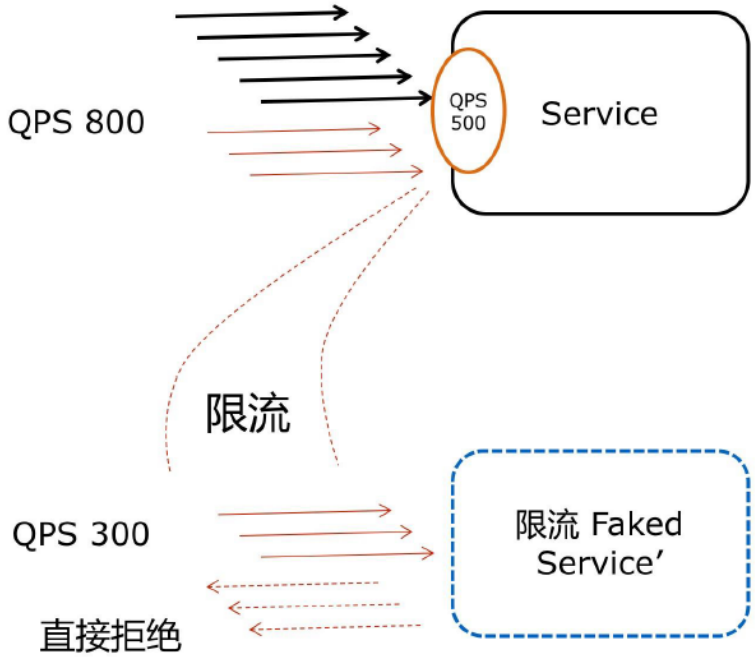
Reliability && Resilience

常见的容错机制：

- **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。

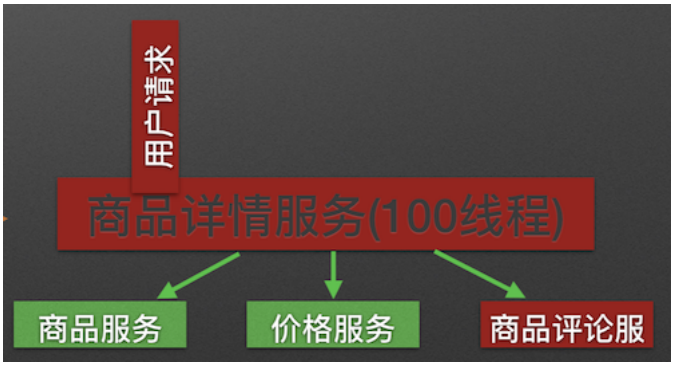
- **服务限流**



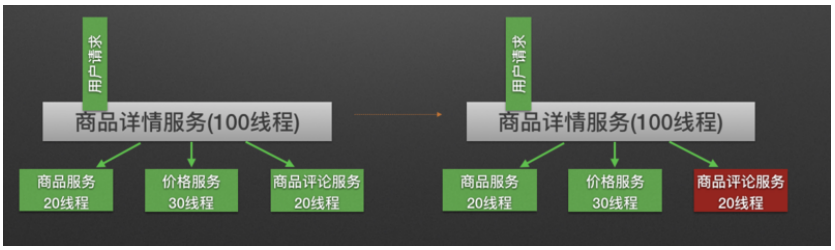
- **隔离**

原理：用户的请求将不再直接访问服务，而是通过线程池中的空闲线程来访问服务，如果线程池已满，则会进行降级处理，用户的请求不会被阻塞，至少可以看到一个执行结果（例如返回友好的提示信息），而不是无休止的等待或者看到系统崩溃。

隔离前：



隔离后：



b）信号隔离：

信号隔离也可以用于限制并发访问，防止阻塞扩散, 与线程隔离最大不同在于执行依赖代码的线程依然是请求线程（该线程需要通过信号申请, 如果客户端是可信的且可以快速返回，可以使用信号隔离替换线程隔离,降低开销。信号量的大小可以动态调整, 线程池大小不可以。

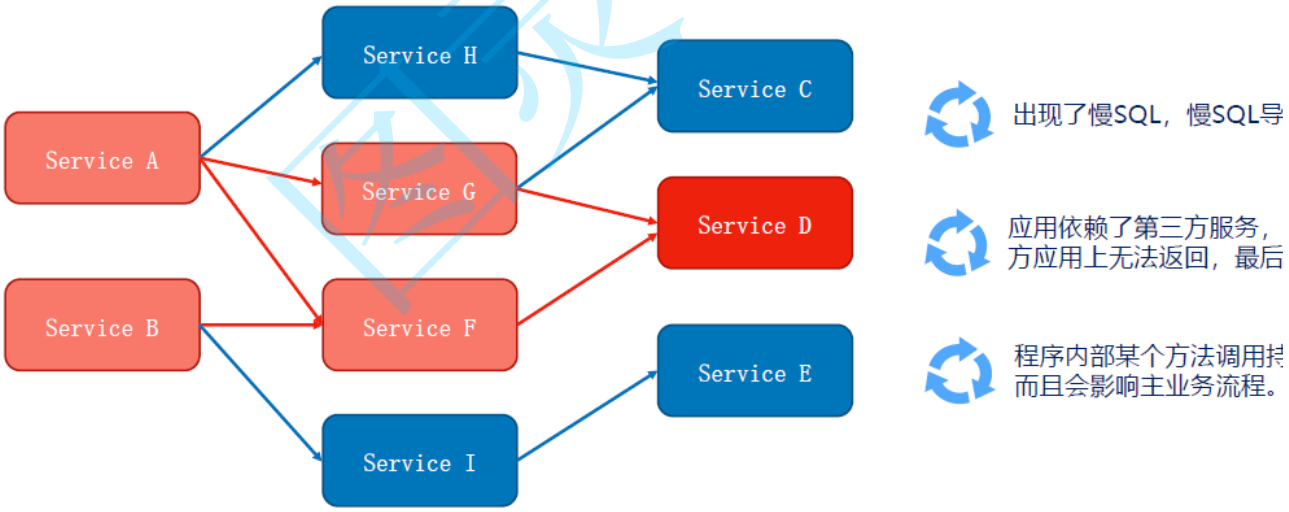
- **服务熔断**

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。



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。

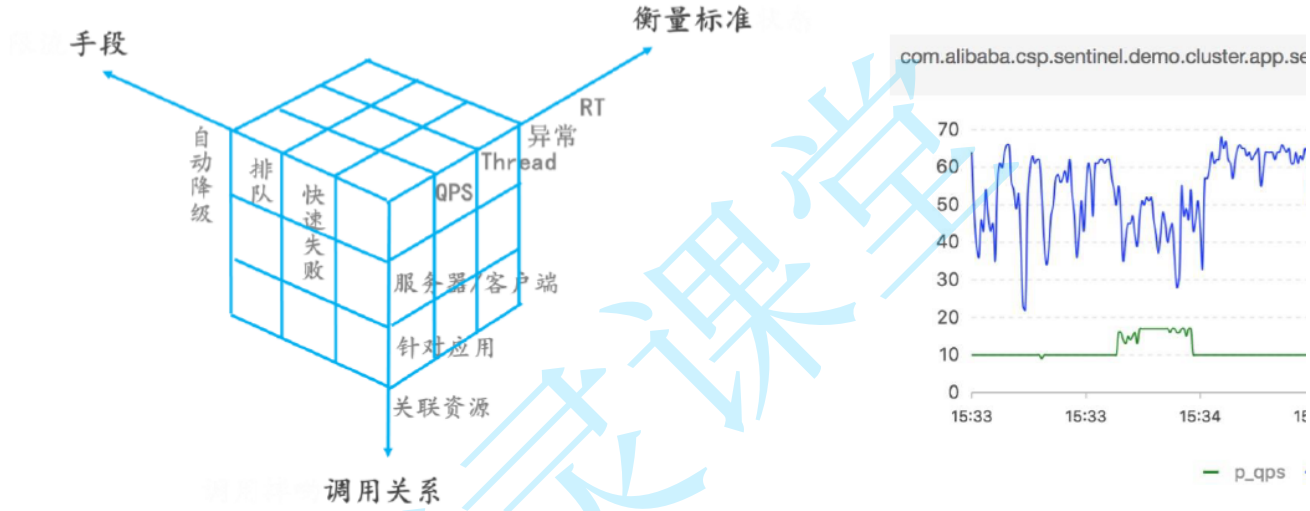
How to keep resiliency?

You NEED Sentinel

3. Sentinel: 分布式系统的流量防卫兵

3.1 Sentinel 是什么

Sentinel 是阿里巴巴开源的，面向分布式服务架构的高可用



多维度的流控降级能力 **秒级实时监**

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 是面向分布式服务架构的流量控制组件，主要以流量为切入点，从限流、流量整形、熔断降级、系统负载保护、热点防护等多个维度来帮助开发者保障微服务的稳定性。

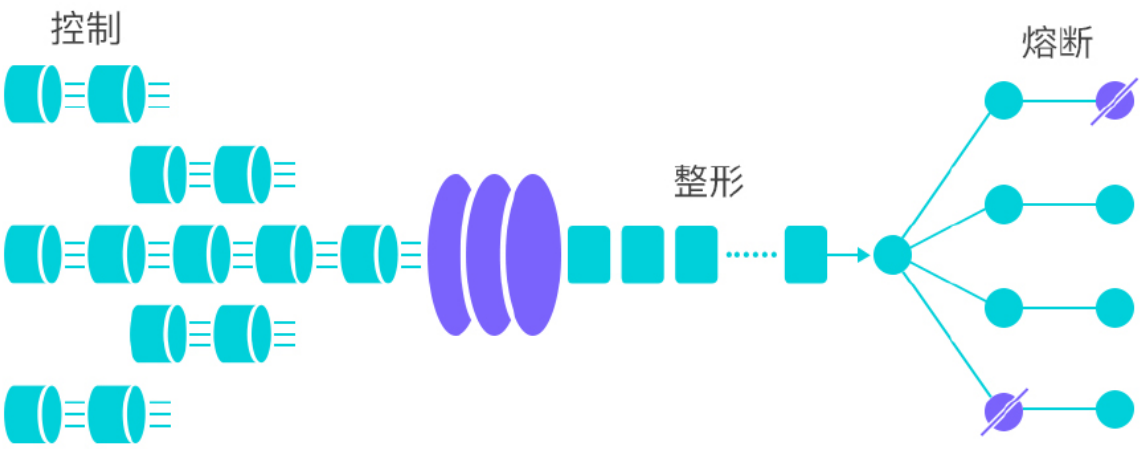
源码地址：https://github.com/alibaba/Sentinel

官方文档：https://github.com/alibaba/Sentinel/wiki

Sentinel具有以下特征:

- **丰富的应用场景**： Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、实时熔断下游不可用应用等。
- **完备的实时监控**： Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
- **广泛的开源生态**： Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。
- **完善的 SPI 扩展点**： Sentinel 提供简单易用、完善的 SPI 扩展点。您可以通过实现扩展点，快速的定制逻辑。例如定制规则管理、适配数据源等。

阿里云提供了 企业级的 Sentinel 服务，**应用高可用服务 AHAS**



Sentinel和Hystrix对比

https://github.com/alibaba/Sentinel/wiki/Sentinel-%E4%B8%8E-Hystrix-%E7%9A%84%E5%AF%B9%E6%AF%94

	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	基于响应时间或失败比率	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、匀速器模式	不支持
系统负载保护	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

4、 Sentinel快速开始

https://github.com/alibaba/Sentinel/wiki/%E5%A6%82%E4%BD%95%E4%BD%BF%E7%94%A8

在官方文档中，定义的Sentinel进行资源保护的几个步骤：

1. 定义资源
2. 定义规则
3. 检验规则是否生效

```
Entry entry = null;
// 务必保证 finally 会被执行
try {
 // 资源名可使用任意有业务语义的字符串 开启资源的保护
 entry = SphU.entry("自定义资源名");
 // 被保护的业务逻辑 method
 // do something...
} catch (BlockException ex) {
 // 资源访问阻止，被限流或被降级 Sentinel定义异常 流控规则，降级规则，热点参数规则。。。。 服务降级(降级规则)
```

```
 // 进行相应的处理操作
} catch (Exception ex) {
 // 若需要配置降级规则，需要通过这种方式记录业务异常 RuntimeException 服务降级 mock feign:fallback
 Tracer.traceEntry(ex, entry);
} finally {
 // 务必保证 exit，务必保证每个 entry 与 exit 配对
 if (entry != null) {
 entry.exit();
 }
```

Sentinel资源保护的方式

API实现

1. 引入依赖

```
<dependency>
 <groupId>com.alibaba.csp</groupId>
 <artifactId>sentinel-core</artifactId>
 <version>1.8.0</version>
</dependency>
```

2. 编写测试逻辑

```
@RestController
@Slf4j
public class HelloController {

 private static final String RESOURCE_NAME = "hello";

 @RequestMapping(value = "/hello")
 public String hello() {

 Entry entry = null;
 try {
 // 资源名可使用任意有业务语义的字符串，比如方法名、接口名或其它可唯一标识的字符串。
 entry = SphU.entry(RESOURCE_NAME);
 // 被保护的业务逻辑
 String str = "hello world";
 log.info("====="+str);
 return str;
 } catch (BlockException e1) {
 // 资源访问阻止，被限流或被降级
 //进行相应的处理操作
 log.info("block!");
 } catch (Exception ex) {
 // 若需要配置降级规则，需要通过这种方式记录业务异常
 Tracer.traceEntry(ex, entry);
 } finally {
 if (entry != null) {
 entry.exit();
 }
 }
 return null;
 }

 /**
 * 定义流控规则
 */
 @PostConstruct
 private static void initFlowRules(){
 List<FlowRule> rules = new ArrayList<>();
 FlowRule rule = new FlowRule();
 //设置受保护的资源
 rule.setResource(RESOURCE_NAME);
```

```
42  // 设置流控规则 QPS
43  rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
44  // 设置受保护的资源阈值
45  // Set limit QPS to 20.
46  rule.setCount(1);
47  rules.add(rule);
48  // 加载配置好的规则
49  FlowRuleManager.loadRules(rules);
50  }
51 }
```

测试效果：

```
er.HelloController        : =====hello world
er.HelloController        : =====hello world
er.HelloController        : block!
er.HelloController        : =====hello world
er.HelloController        : block!
er.HelloController        : block!
er.HelloController        : block!
er.HelloController        : block!
er.HelloController        : =====hello world
```

缺点：

- 业务侵入性很强，需要在controller中写入非业务代码.
- 配置不灵活 若需要添加新的受保护资源 需要手动添加 init方法来添加流控规则

@SentinelResource注解实现

> @SentinelResource 注解用来标识资源是否被限流、降级。
> blockHandler：定义当资源内部发生了BlockException应该进入的方法（捕获的是Sentinel定义的异常）
> fallback：定义的是资源内部发生了Throwable应该进入的方法
> exceptionsToIgnore：配置fallback可以忽略的异常
> 源码入口：com.alibaba.csp.sentinel.annotation.aspectj.SentinelResourceAspect

1.引入依赖

```
1 <dependency>
2  <groupId>com.alibaba.csp</groupId>
3  <artifactId>sentinel-annotation-aspectj</artifactId>
4  <version>1.8.0</version>
5 </dependency>
```

2.配置切面支持

```
1 @Configuration
2 public class SentinelAspectConfiguration {
3
4  @Bean
5  public SentinelResourceAspect sentinelResourceAspect() {
6  return new SentinelResourceAspect();
7  }
8 }
```

3.UserController中编写测试逻辑，添加@SentinelResource，并配置blockHandler和fallback

```
1 @RequestMapping(value = "/findOrderByUserId/{id}")
2 @SentinelResource(value = "findOrderByUserId",
3  fallback = "fallback",fallbackClass = ExceptionUtil.class,
4  blockHandler = "handleException",blockHandlerClass = ExceptionUtil.class
5  )
6 public R findOrderByUserId(@PathVariable("id") Integer id) {
7  //ribbon实现
8  String url = "http://mall-order/order/findOrderByUserId/"+id;
9  R result = restTemplate.getForObject(url,R.class);
10
```

```
 if(id==4){
 throw new IllegalArgumentException("非法参数异常");
 }

 return result;
}
```

4.编写ExceptionUtil，注意如果指定了class，方法必须是static方法

```
public class ExceptionUtil {

 public static R fallback(Integer id,Throwable e){
 return R.error(-2,"===被异常降级啦===");
 }

 public static R handleException(Integer id, BlockException e){
 return R.error(-2,"===被限流啦===");
 }
}
```

5.流控规则设置可以通过Sentinel dashboard配置

客户端需要引入 Transport 模块来与 Sentinel 控制台进行通信。

```
<dependency>
 <groupId>com.alibaba.csp</groupId>
 <artifactId>sentinel-transport-simple-http</artifactId>
 <version>1.8.0</version>
</dependency>
```

```
-Dcsp.sentinel.dashboard.server=consoleIp:port
```

5. 启动 Sentinel 控制台

下载控制台 jar 包并在本地启动：可以参见 此处文档
https://github.com/alibaba/Sentinel/releases

sentinel-dashboard-1.8.0.jar

```
#启动控制台命令
java -jar sentinel-dashboard-1.8.0.jar
```

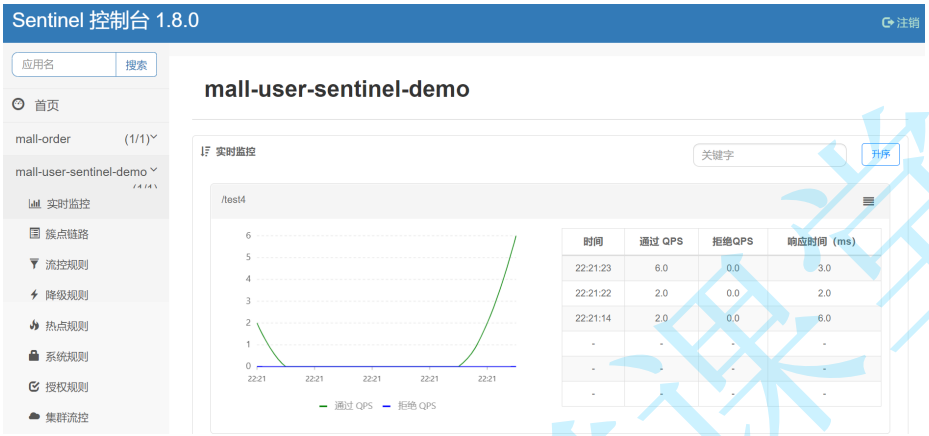
> 用户可以通过如下参数进行配置：
> `-Dsentinel.dashboard.auth.username=sentinel` 用于指定控制台的登录用户名为 `sentinel`；
> `-Dsentinel.dashboard.auth.password=123456` 用于指定控制台的登录密码为 `123456`；如果省略这两个参数，默认用户和密码均为 `sentinel`；
> `-Dserver.servlet.session.timeout=7200` 用于指定 Spring Boot 服务端 session 的过期时间，如 `7200` 表示 7200 秒；`60m` 表示 60 分钟，默认为 30 分钟；
>
> java -Dserver.port=8858 -Dsentinel.dashboard.auth.username=xushu -Dsentinel.dashboard.auth.password=123456 -jar sentinel-dashboard-1.8.0.jar

为了方便快捷启动可以在桌面创建.bat文件

```
java -Dserver.port=8858 -Dsentinel.dashboard.auth.username=xushu -Dsentinel.dashboard.auth.password=123456 -jar D:\server\sentinel-dashboard-1.8.0.jar
pause
```

访问http://localhost:8080/#/login ,默认用户名密码： sentinel/sentinel

Sentinel 会在客户端首次调用的时候进行初始化，开始向控制台发送心跳包，所以要确保客户端有访问量；



6、 Spring Cloud Alibaba整合Sentinel

1.引入依赖

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

2.添加yml配置，为微服务设置sentinel控制台地址
添加Sentinel后，需要暴露/actuator/sentinel端点,而Springboot默认是没有暴露该端点的，所以需要设置，测试http://localhost:8800/actuator/sentinel

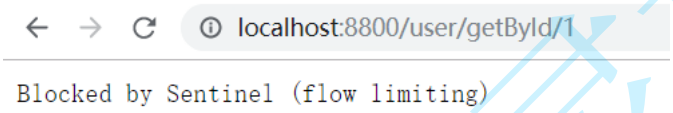
```
server:
 port: 8800

spring:
 application:
 name: mall-user-sentinel-demo
 cloud:
 nacos:
 discovery:
 server-addr: 127.0.0.1:8848

 sentinel:
 transport:
 # 添加sentinel的控制台地址
 dashboard: 127.0.0.1:8080
 # 指定应用与Sentinel控制台交互的端口，应用本地会起一个该端口占用的HttpServer
```

```
17  # port: 8719
18
```

3.在sentinel控制台中设置流控规则

- **资源名**：接口的API
- **针对来源**：默认是default，当多个微服务都调用这个资源时，可以配置微服务名来对指定的微服务设置阈值
- **阈值类型**：分为QPS和线程数 假设阈值为10
- **QPS类型**：只得是每秒访问接口的次数>10就进行限流
- **线程数**：为接受请求该资源分配的线程数>10就进行限流

测试： 因为QPS是1，所以1秒内多次访问会出现如下情形：

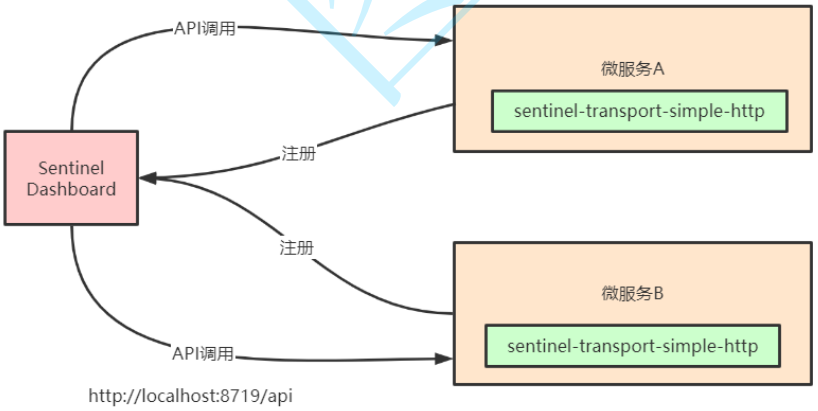


访问http://localhost:8800/actuator/sentinel， 可以查看flowRules

```
{
    blockPage: null,
    appName: "service-sentinel-consumer",
    consoleServer: "localhost:8888",
    coldFactor: "3",
  - rules: {
        systemRules: [ ],
        authorityRule: [ ],
        paramFlowRule: [ ],
      - flowRules: [
          - {
                resource: "/user/getById/1",
                limitApp: "default",
                grade: 1,
                count: 1,
                strategy: 0,
                refResource: null,
                controlBehavior: 0,
                warmUpPeriodSec: 10,
                maxQueueingTimeMs: 500,
                clusterMode: false,
              - clusterConfig: {
                    flowId: null,
                    thresholdType: 0,
                    fallbackToLocalWhenFail: true,
                    strategy: 0,
                    sampleCount: 10,
                    windowIntervalMs: 1000
                }
            }
        ],
        degradeRules: [ ]
    }
```


微服务和Sentinel Dashboard通信原理

Sentinel控制台与微服务端之间，实现了一套服务发现机制，集成了Sentinel的微服务都会将元数据传递给Sentinel控制台，架构图如下所示：



流控针对privoder　熔断降级 针对consumer