主讲老师：Fox

1. Sentinel控制台介绍

Sentinel 提供一个轻量级的开源控制台，它提供机器发现以及健康情况管理、监控（单机和集群），规则管理和推送的功能。

Sentinel 控制台包含如下功能:

- 查看机器列表以及健康情况：收集 Sentinel 客户端发送的心跳包，用于判断机器是否在线。
- 监控 (单机和集群聚合)：通过 Sentinel 客户端暴露的监控 API，定期拉取并且聚合应用监控信息，最终可以实现秒级的实时监控。
- 规则管理和推送：统一管理推送规则。
- 鉴权：生产环境中鉴权非常重要。这里每个开发者需要根据自己的实际情况进行定制。

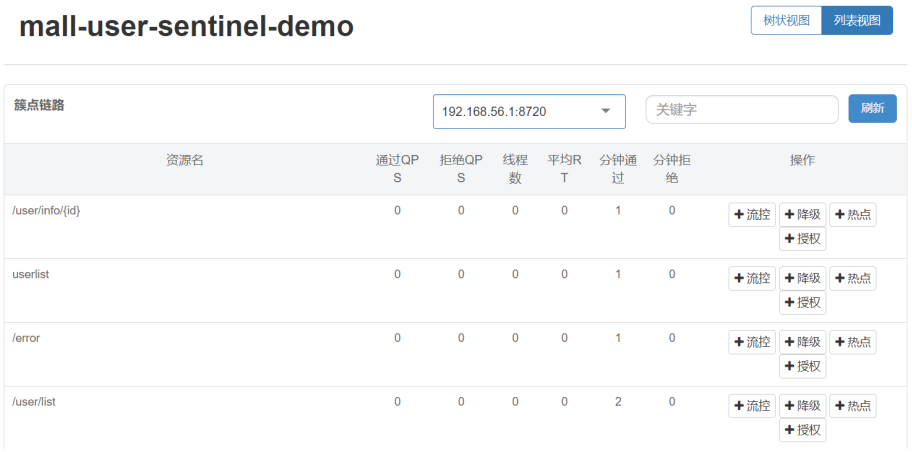
阿里云提供了 企业级的 Sentinel 控制台，**应用高可用服务 AHAS**

1.1 实时监控

监控接口的通过的QPS和拒绝的QPS

1.2 簇点链路

用来显示微服务的所监控的API



1.3 流控规则

流量控制（flow control），其原理是监控应用流量的 QPS 或并发线程数等指标，当达到指定的阈值时对流量进行控制，以避免被瞬时的流量高峰冲垮，从而保障应用的高可用性。

同一个资源可以创建多条限流规则。`FlowSlot` 会对该资源的所有限流规则依次遍历，直到有规则触发限流或者所有规则遍历完毕。一条限流规则主要由下面几个因素组成，我们可以组合这些元素来实现不同的限流效果。

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	限流阈值类型，QPS 模式（1）或并发线程数模式（0）	QPS 模式
limitApp	流控针对的调用来源	`default`，代表不区分调用来源
strategy	调用关系限流策略：直接、链路、关联	根据资源本身（直接）
controlBehavior	流控效果（直接拒绝/WarmUp/匀速+排队等待），不支持按调用关系限流	直接拒绝
clusterMode	是否集群限流	否

参考文档：

https://github.com/alibaba/Sentinel/wiki/%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6

限流阈值类型

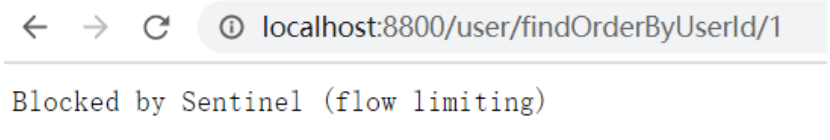
流量控制主要有两种统计类型，一种是统计并发线程数，另外一种则是统计 QPS。类型由 `FlowRule` 的 `grade` 字段来定义。其中，0 代表根据并发数量来限流，1 代表根据 QPS 来进行流量控制。

> QPS（Query Per Second）：每秒请求数，就是说服务器在一秒的时间内处理了多少个请求。

QPS

进入簇点链路选择具体的访问的API，然后点击流控按钮

测试：http://localhost:8800/user/findOrderByUserId/1




```
Blocked by Sentinel (flow limiting)
```

BlockException异常统一处理

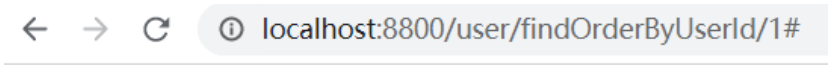
springwebmvc接口资源限流入口在**HandlerInterceptor**的实现类**AbstractSentinelInterceptor**的preHandle方法中，对异常的处理是**BlockExceptionHandler**的实现类

> **sentinel 1.7.1 引入了**sentinel-spring-webmvc-adapter.jar

自定义BlockExceptionHandler 的实现类统一处理BlockException

```
@Slf4j
@Component
public class MyBlockExceptionHandler implements BlockExceptionHandler {
 @Override
 public void handle(HttpServletRequest request, HttpServletResponse response, BlockException e) throws Exception {
  log.info("BlockExceptionHandler BlockException================"+e.getRule());

  R r = null;

  if (e instanceof FlowException) {
  r = R.error(100,"接口限流了");

  } else if (e instanceof DegradeException) {
  r = R.error(101,"服务降级了");

  } else if (e instanceof ParamFlowException) {
  r = R.error(102,"热点参数限流了");

  } else if (e instanceof SystemBlockException) {
  r = R.error(103,"触发系统保护规则了");

  } else if (e instanceof AuthorityException) {
  r = R.error(104,"授权规则不通过");
  }

  //返回json数据
  response.setStatus(500);
  response.setCharacterEncoding("utf-8");
  response.setContentType(MediaType.APPLICATION_JSON_VALUE);
  new ObjectMapper().writeValue(response.getWriter(), r);

  }
}
```

测试：



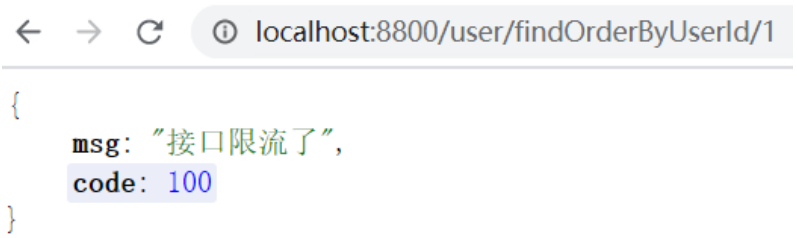

```
{
    msg: "接口限流了",
    code: 100
}
```


并发线程数

并发数控制用于保护业务线程池不被慢调用耗尽。例如，当应用所依赖的下游应用由于某种原因导致服务不稳定、响应延迟增加，对于调用者来说，意味着吞吐量下降和更多的线程数占用，极端情况下甚至导致线程池耗尽。为应对太多线程占用的情况，业内有使用隔离的方案，比如通过不同业务逻辑使用不同线程

池来隔离业务自身之间的资源争抢（线程池隔离）。这种隔离方案虽然隔离性比较好，但是代价就是线程数目太多，线程上下文切换的 overhead 比较大，特别是对低延时的调用有比较大的影响。Sentinel 并发控制不负责创建和管理线程池，而是简单统计当前请求上下文的线程数目（正在执行的调用数目），如果超出阈值，新的请求会被立即拒绝，效果类似于信号量隔离。并发数控制通常在调用端进行配置。

可以利用jmeter测试

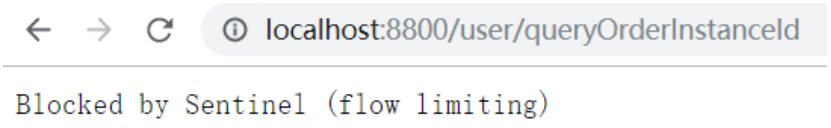


流控模式

基于调用关系的流量控制。调用关系包括调用方、被调用方；一个方法可能会调用其它方法，形成一个调用链路的层次关系。

直接

资源调用达到设置的阈值后直接被流控抛出异常



关联

当两个资源之间具有资源争抢或者依赖关系的时候，这两个资源便具有了关联。比如对数据库同一个字段的读操作和写操作存在争抢，读的速度过高会影响写得速度，写的速度过高会影响读的速度。如果放任读写操作争抢资源，则争抢本身带来的开销会降低整体的吞吐量。可使用关联限流来避免具有关联关系的资源之间过度的争抢，举例来说，`read_db` 和 `write_db` 这两个资源分别代表数据库读写，我们可以给 `read_db` 设置限流规则来达到写优先的目的：设置 `strategy` 为 `RuleConstant.STRATEGY_RELATE` 同时设置 `refResource` 为 `write_db`。这样当写库操作过于频繁时，读数据的请求会被限流。

链路

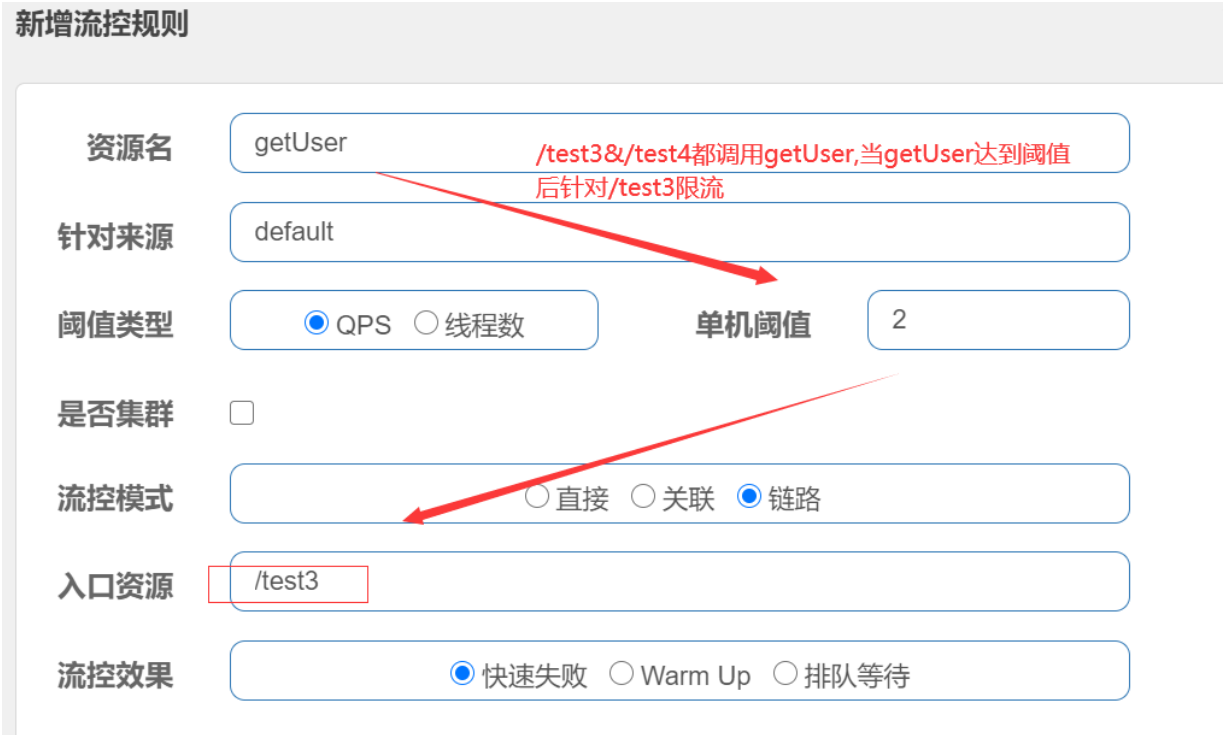
根据调用链路入口限流。

`NodeSelectorSlot` 中记录了资源之间的调用链路，这些资源通过调用关系，相互之间构成一棵调用树。这棵树的根节点是一个名字为 `machine-root` 的虚拟节点，调用链的入口都是这个虚节点的子节点。

一棵典型的调用树如下图所示：

```
machine-root
/ \
/ \
Entrance1 Entrance2
/ \
/ \
DefaultNode(nodeA) DefaultNode(nodeA)
```

上图中来自入口 `Entrance1` 和 `Entrance2` 的请求都调用到了资源 `NodeA`，Sentinel 允许只根据某个入口的统计信息对资源限流。



测试会发现链路规则不生效

注意，高版本此功能直接使用不生效，如何解决？

> 从1.6.3版本开始，Sentinel Web filter默认收敛所有URL的入口context，导致链路限流不生效。
> 从1.7.0版本开始，官方在CommonFilter引入了WEB_CONTEXT_UNIFY参数，用于控制是否收敛context，将其配置为false即可根据不同的URL进行链路限流。

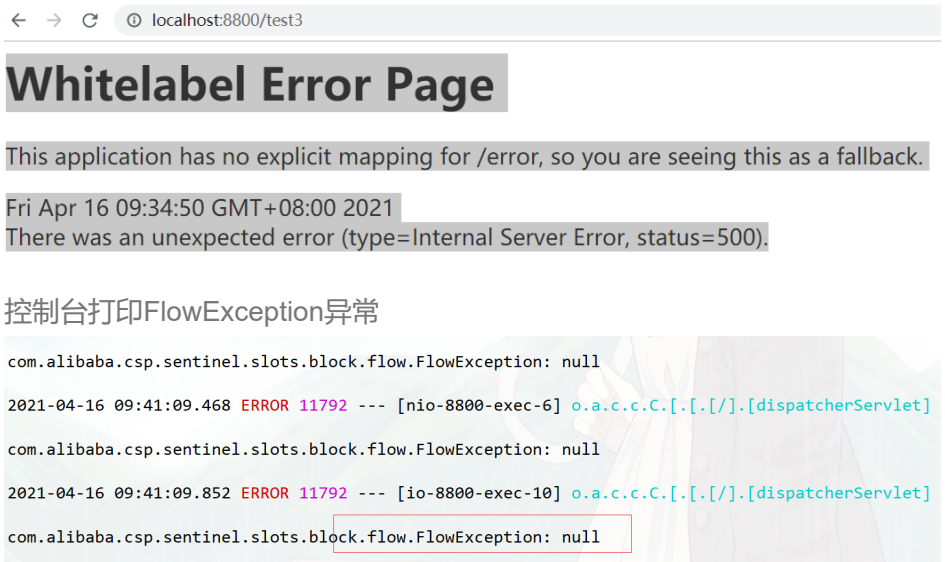
1.8.0 需要引入sentinel-web-servlet依赖

```
<!--- 解决流控链路不生效的问题-->
<dependency>
 <groupId>com.alibaba.csp</groupId>
 <artifactId>sentinel-web-servlet</artifactId>
</dependency>
```

添加配置类，配置CommonFilter过滤器，指定WEB_CONTEXT_UNIFY=false，禁止收敛URL的入口context

```
@Configuration
public class SentinelConfig {
 @Bean
 public FilterRegistrationBean sentinelFilterRegistration() {
 FilterRegistrationBean registration = new FilterRegistrationBean();
 registration.setFilter(new CommonFilter());
 registration.addUrlPatterns("/*");
 // 入口资源关闭聚合 解决流控链路不生效的问题
 registration.addInitParameter(CommonFilter.WEB_CONTEXT_UNIFY, "false");
  registration.setName("sentinelFilter");
  registration.setOrder(1);
  return registration;
  }
}
```

再次测试链路规则，链路规则生效，但是出现异常



控制台打印FlowException异常

```
com.alibaba.csp.sentinel.slots.block.flow.FlowException: null
2021-04-16 09:41:09.468 ERROR 11792 --- [nio-8800-exec-6] o.a.c.c.C.[.[.[/].[dispatcherServlet]
com.alibaba.csp.sentinel.slots.block.flow.FlowException: null
2021-04-16 09:41:09.852 ERROR 11792 --- [io-8800-exec-10] o.a.c.c.C.[.[.[/].[dispatcherServlet]
com.alibaba.csp.sentinel.slots.block.flow.FlowException: null
```

原因分析：

1. Sentinel流控规则的处理核心是 **FlowSlot**, 对getUser资源进行了限流保护，当请求QPS超过阈值2的时候，就会触发流控规则抛出**FlowException**异常

2. 对getUser资源保护的方式是@SentinelResource注解模式，会在对应的**SentinelResourceAspect**切面逻辑中处理BlockException类型的**FlowException**异常

（解决方案： 在@SentinelResource注解中指定blockHandler处理**BlockException**）

```
// UserServiceImpl.java

```

```java
@Override
@SentinelResource(value = "getUser",blockHandler = "handleException")
public UserEntity getUser(int id){
 UserEntity user = baseMapper.selectById(id);
 return user;
}

public UserEntity handleException(int id, BlockException ex) {
 UserEntity userEntity = new UserEntity();
 userEntity.setUsername("===被限流降级啦===");
 return userEntity;
}
```

如果此过程没有处理FlowException, AOP就会对异常进行处理，核心代码在CglibAopProxy.CglibMethodInvocation#proceed中，抛出**UndeclaredThrowableException**异常，属于RuntimeException

```java
public Object proceed() throws Throwable {
    try {
        return super.proceed();
    }
    catch (RuntimeException ex) {
        throw ex;
    }
    catch (Exception ex) {  ex: "com.alibaba.csp.sentinel.slo
        if (ReflectionUtils.declaresException(getMethod(), ex
            throw ex;
        }
        else {
            throw new UndeclaredThrowableException(ex);  ex:
        }
    }
```

3.异常继续向上抛出，引入CommonFilter后，CommonFilter添加了对异常的处理机制，所以会在CommonFilter中进行处理。

（注意此处对BlockException异常的处理是UrlBlaockHandler的实现类，而在AbstractSentinelInterceptor拦截器中是使用BlockExceptionHandler的实现类处理）

```java
                urlEntry = SphU.entry(pathWithHttpMethod, ResourceTypeConstants
.COMMON_WEB, EntryType.IN);
            } else {
                urlEntry = SphU.entry(target, ResourceTypeConstants.COMMON_WEB,
EntryType.IN);
            }
        }
        chain.doFilter(request, response);
    } catch (BlockException e) {
        HttpServletResponse sResponse = (HttpServletResponse) response;
        // Return the block page, or redirect to another URL.
        WebCallbackManager.getUrlBlockHandler().blocked(sRequest, sResponse, e);
    } catch (IOException | ServletException | RuntimeException e2) {
        Tracer.traceEntry(e2, urlEntry);
        throw e2;
```

受保护的资源

BlockException的处理是UrlBlockHandler的实现类

执行的是RuntimeException类型的异常

会抛出一个RuntimeException类型的**UndeclaredThrowableException**异常，然后打印到控制台显示

```
result = {NestedServletException@11175} "org.springframework.web.util.NestedServletException: Request processing failed; nested exception is java.lang.reflect.UndeclaredThrowableException"
  detailMessage = "Request processing failed"
  cause = {UndeclaredThrowableException@11181} "java.lang.reflect.UndeclaredThrowableException"
    undeclaredThrowable = {FlowException@11164} "com.alibaba.csp.sentinel.slots.block.flow.FlowException"
    detailMessage = null
    cause = null
    stackTrace = {StackTraceElement[63]@11278}
    suppressedExceptions = {Collections$UnmodifiableRandomAccessList@11071} size = 0
  stackTrace = {StackTraceElement[45]@11184}
  suppressedExceptions = {Collections$UnmodifiableRandomAccessList@11071} size = 0
```

此处又是有坑： FlowException不会被BlockException异常机制处理，因为FlowException已经被封装为RuntimeException类型的UndeclaredThrowableException异常

测试：

自定义CommonFilter对BlockException异常处理逻辑，用于处理经过CommonFilter处理的spring webmvc接口的BlockException

```
// SentinelConfig.java
@Bean
public FilterRegistrationBean sentinelFilterRegistration() {
 FilterRegistrationBean registration = new FilterRegistrationBean();
 registration.setFilter(new CommonFilter());
 registration.addUrlPatterns("/*");
 // 入口资源关闭聚合 解决流控链路不生效的问题
 registration.addInitParameter(CommonFilter.WEB_CONTEXT_UNIFY, "false");
 registration.setName("sentinelFilter");
 registration.setOrder(1);

 //CommonFilter的BlockException自定义处理逻辑
 WebCallbackManager.setUrlBlockHandler(new MyUrlBlockHandler());

 return registration;
}

// UrlBlockHandler的实现类
@Slf4j
public class MyUrlBlockHandler implements UrlBlockHandler {
 @Override
 public void blocked(HttpServletRequest request, HttpServletResponse response, BlockException e) thr
ows IOException {
 log.info("UrlBlockHandler BlockException================"+e.getRule());

 R r = null;

 if (e instanceof FlowException) {
 r = R.error(100,"接口限流了");

 } else if (e instanceof DegradeException) {
 r = R.error(101,"服务降级了");

 } else if (e instanceof ParamFlowException) {
 r = R.error(102,"热点参数限流了");

 } else if (e instanceof SystemBlockException) {
 r = R.error(103,"触发系统保护规则了");

 } else if (e instanceof AuthorityException) {
 r = R.error(104,"授权规则不通过");
 }

 //返回json数据
 response.setStatus(500);
 response.setCharacterEncoding("utf-8");
 response.setContentType(MediaType.APPLICATION_JSON_VALUE);
```

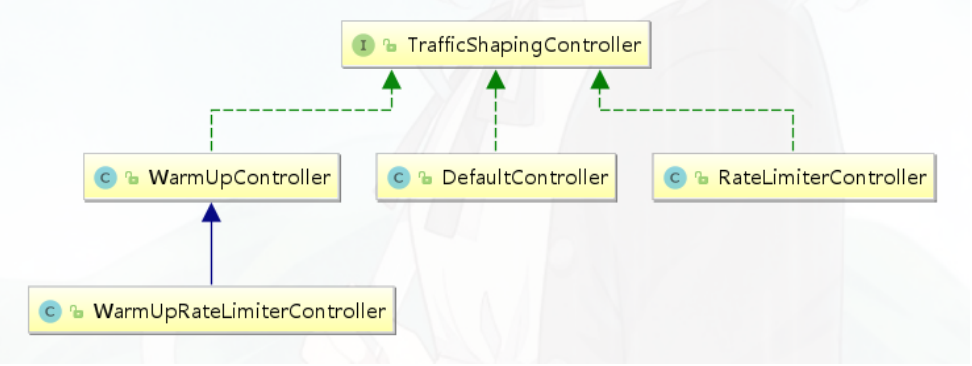
```
47    new ObjectMapper().writeValue(response.getWriter(), r);
48    }
49  }
50
51
```

测试，此场景拦截不到BlockException，对应@SentinelResource指定的资源必须在@SentinelResource注解中指定blockHandler处理BlockException

总结： 为了解决链路规则引入ComonFilter的方式，除了此处问题，还会导致更多的问题，不建议使用ComonFilter的方式。 流控链路模式的问题等待官方后续修复，或者使用AHAS。

流控效果

当 QPS 超过某个阈值的时候，则采取措施进行流量控制。流量控制的效果包括以下几种：**快速失败（直接拒绝）**、**Warm Up**（预热）、**匀速排队（排队等待）**。对应 `FlowRule` 中的 `controlBehavior` 字段。



快速失败

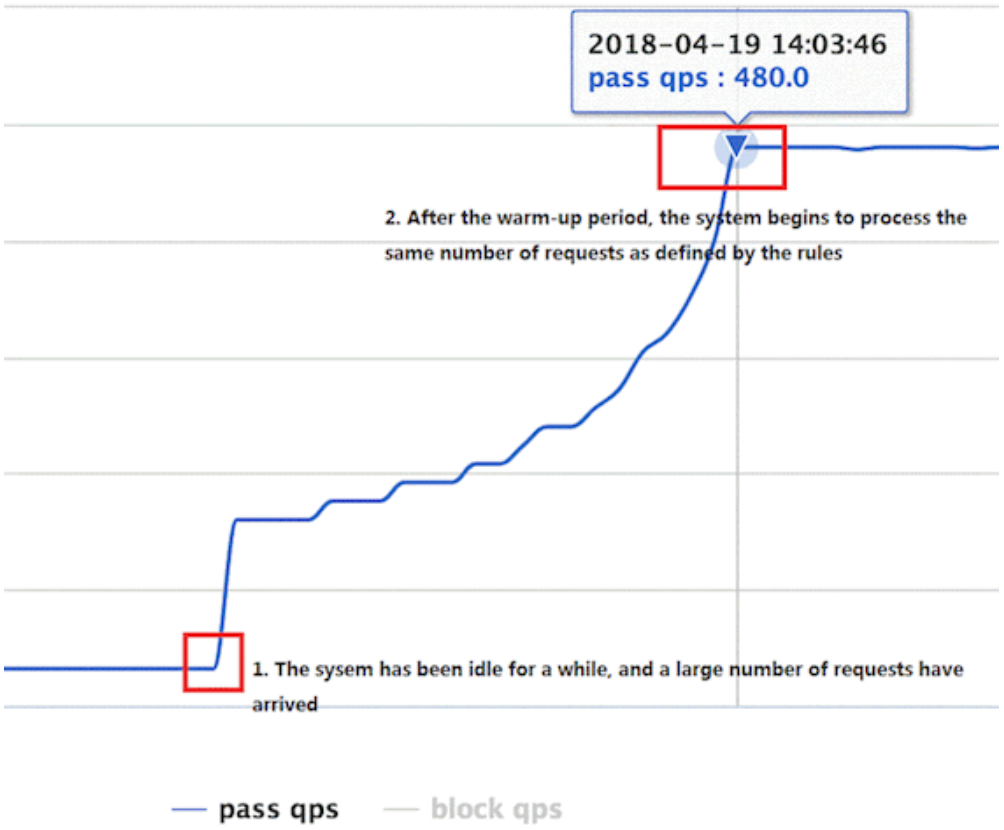
（`RuleConstant.CONTROL_BEHAVIOR_DEFAULT`）方式是默认的流量控制方式，当QPS超过任意规则的阈值后，新的请求就会被立即拒绝，拒绝方式为抛出`FlowException`。这种方式适用于对系统处理能力确切已知的情况下，比如通过压测确定了系统的准确水位时。

Warm Up

Warm Up（`RuleConstant.CONTROL_BEHAVIOR_WARM_UP`）方式，即预热/冷启动方式。当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。通过"冷启动"，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，给冷系统一个预热的时间，避免冷系统被压垮。

冷加载因子: codeFactor 默认是3，即请求 QPS 从 threshold / 3 开始，经预热时长逐渐升至设定的 QPS 阈值。

通常冷启动的过程系统允许通过的 QPS 曲线如下图所示



测试用例

```
@RequestMapping("/test")
public String test() {
 try {
 Thread.sleep(100);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 return "========test()========";
}
```

编辑流控规则

jmeter测试

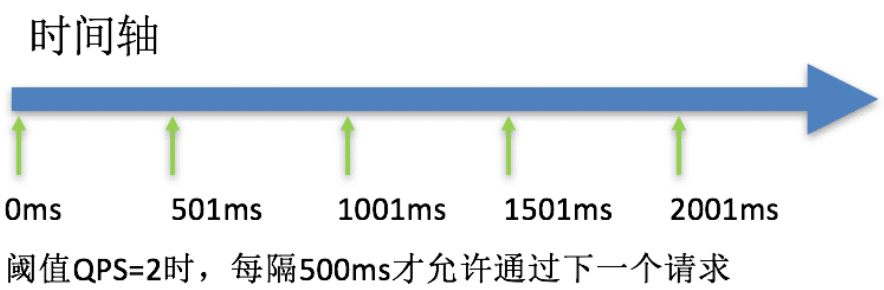
查看实时监控，可以看到通过QPS存在缓慢增加的过程

时间	通过 QPS	拒绝QPS	响应时间（ms）
11:26:00	10.0	15.0	500.0
11:25:59	10.0	15.0	500.0
11:25:58	9.0	16.0	445.0
11:25:57	6.0	17.0	500.0
11:25:56	5.0	18.0	400.0
11:25:55	4.0	18.0	501.0

匀速排队

匀速排队（`RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER`）方式会严格控制请求通过的间隔时间，也即是让请求以均匀的速度通过，对应的是漏桶算法。

该方式的作用如下图所示：



这种方式主要用于处理间隔性突发的流量，例如消息队列。想象一下这样的场景，在某一秒有大量的请求到来，而接下来的几秒则处于空闲状态，我们希望系统能够在接下来的空闲期间逐渐处理这些请求，而不

是在第一秒直接拒绝多余的请求。

> 注意：匀速排队模式暂时不支持 QPS > 1000 的场景。

jemeter压测

查看实时监控，可以看到通过QPS为5，体现了匀速排队效果

时间	通过 QPS	拒绝QPS	响应时间（ms）
11:37:41	2.0	0.0	750.0
11:37:40	5.0	4.0	624.0
11:37:39	5.0	10.0	499.0
11:37:38	5.0	10.0	499.0
11:37:37	5.0	10.0	499.0
11:37:36	5.0	10.0	500.0

降级规则

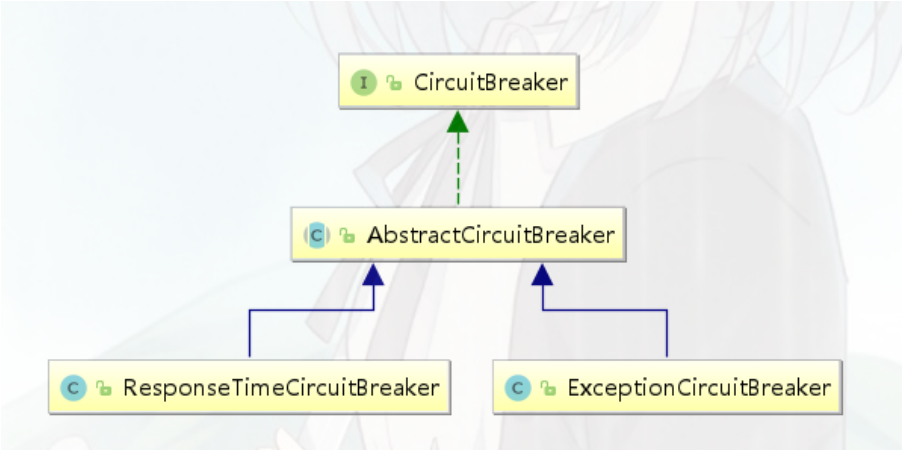
除了流量控制以外，对调用链路中不稳定的资源进行熔断降级也是保障高可用的重要措施之一。我们需要对不稳定的**弱依赖服务调用**进行熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。熔断降级作为保护自身的手段，通常在客户端（调用端）进行配置。

熔断降级规则说明

熔断降级规则（DegradeRule）包含下面几个重要的属性：

Field	说明
resource	资源名，即规则的作用对象
grade	熔断策略，支持慢调用比例/异常比例/异常数策略
count	慢调用比例模式下为慢调用临界 RT（超出该值计为慢调用）；异常比例/异常数模式下为对应的阈值
timeWindow	熔断时长，单位为 s
minRequestAmount	熔断触发的最小请求数，请求数小于该值时即使异常比率超出阈值也不会熔断（1.7.0 引入）

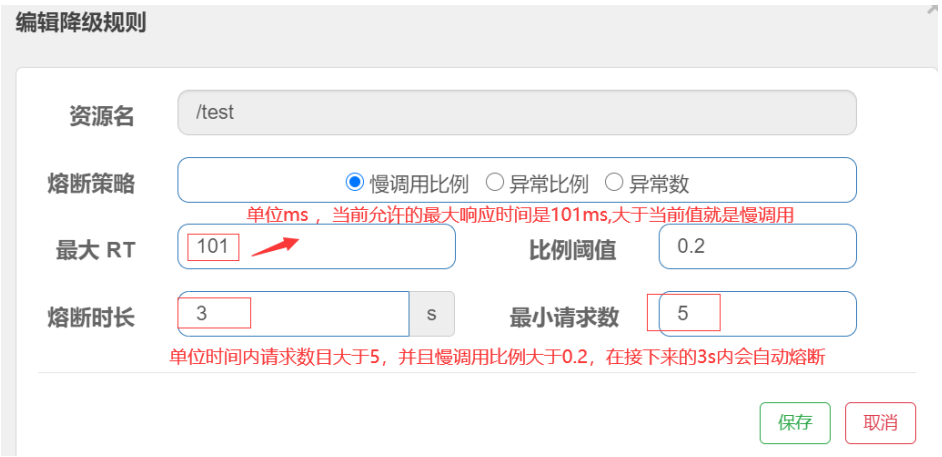
statIntervalMs	统计时长（单位为 ms），如 60*1000 代表分钟级（1.8.0 引入）
slowRatioThreshold	慢调用比例阈值，仅慢调用比例模式有效（1.8.0 引入）



熔断策略

慢调用比例

慢调用比例 (`SLOW_REQUEST_RATIO`)：选择以慢调用比例作为阈值，需要设置允许的慢调用 RT（即最大的响应时间），请求的响应时间大于该值则统计为慢调用。当单位统计时长（`statIntervalMs`）内请求数目大于设置的最小请求数目，并且慢调用的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求响应时间小于设置的慢调用 RT 则结束熔断，若大于设置的慢调用 RT 则会再次被熔断。

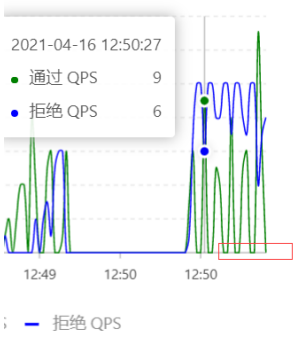


测试用例

```
@RequestMapping("/test")
public String test() {
 try {
 Thread.sleep(100);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 return "========test()========";
}
```

jemeter压测/test接口，保证每秒请求数超过配置的最小请求数

查看实时监控，可以看到断路器熔断效果



熔断效果

时间	通过 QPS	拒绝QPS	响应时间（ms）
12:50:43	0.0	8.0	0.0
12:50:42	6.0	7.0	101.0
12:50:41	13.0	4.0	91.0
12:50:40	0.0	10.0	0.0
12:50:39	0.0	10.0	0.0
12:50:38	6.0	7.0	101.0

此时浏览器访问会出现服务降级结果

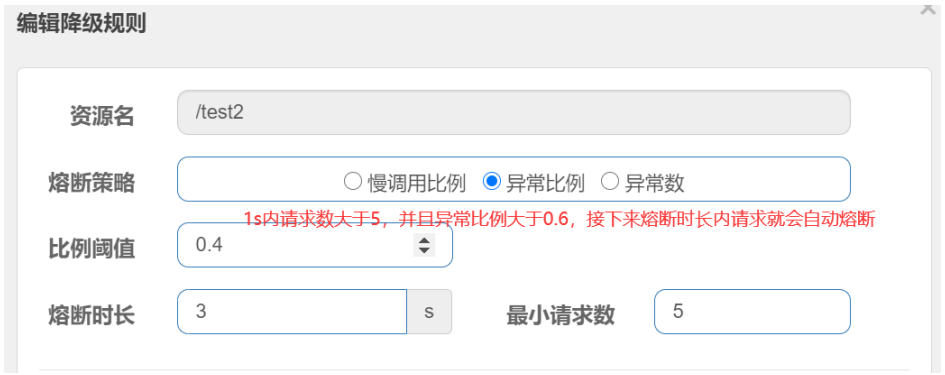
异常比例

异常比例 (ERROR_RATIO)：当单位统计时长（statIntervalMs）内请求数目大于设置的最小请求数目，并且异常的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。

测试用例

```
@RequestMapping("/test2")
public String test2() {
 atomicInteger.getAndIncrement();
 if (atomicInteger.get() % 2 == 0){
 //模拟异常和异常比率
 int i = 1/0;
 }

 return "========test2()========";
}
```

配置降级规则



查看实时监控，可以看到断路器熔断效果

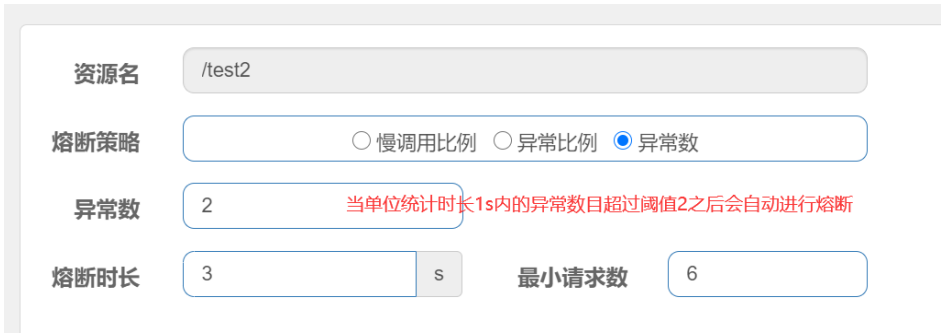
时间	通过 QPS	拒绝QPS	响应时间（ms）
16:13:05	4.0	0.0	0.0
16:13:04	1.0	5.0	0.0
16:13:03	0.0	6.0	0.0
16:13:02	0.0	6.0	0.0
16:13:01	6.0	0.0	0.0
16:13:00	2.0	0.0	0.0

异常数

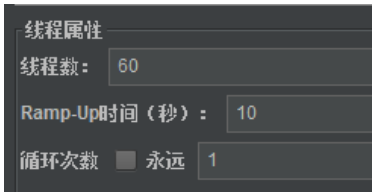
异常数 (ERROR_COUNT)：当单位统计时长内的异常数目超过阈值之后会自动进行熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。

注意：异常降级仅针对业务异常，对 Sentinel 限流降级本身的异常（BlockException）不生效。

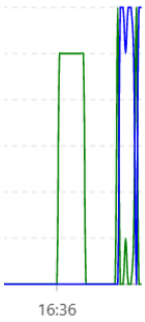
配置降级规则



jemeter测试



查看实时监控，可以看到断路器熔断效果



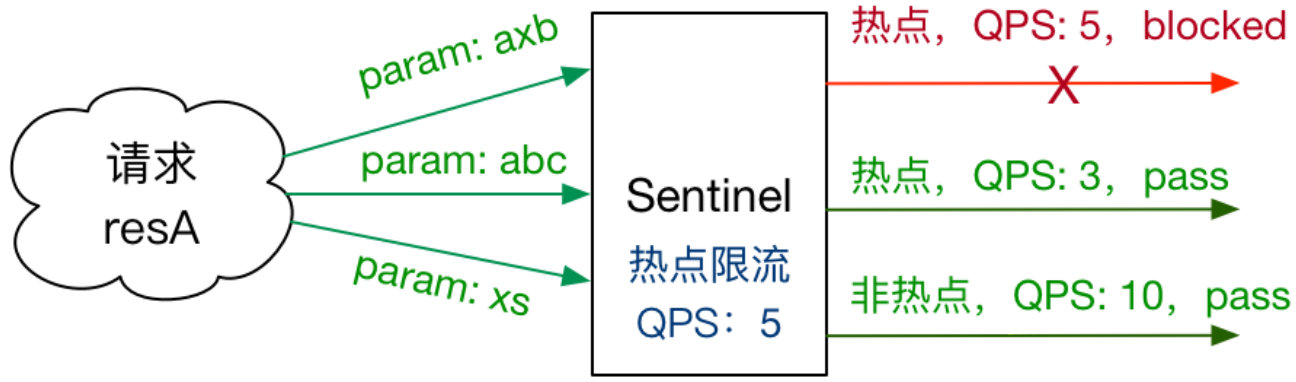
时间	通过 QPS	拒绝QPS	响应时间（ms）
16:37:16	0.0	6.0	0.0
16:37:15	0.0	6.0	0.0
16:37:14	6.0	0.0	0.0
16:37:13	1.0	5.0	0.0
16:37:12	0.0	6.0	0.0
16:37:11	0.0	6.0	0.0

热点参数限流

何为热点？热点即经常访问的数据。很多时候我们希望统计某个热点数据中访问频次最高的 Top K 数据，并对其访问进行限制。比如：

- 商品 ID 为参数，统计一段时间内最常购买的商品 ID 并进行限制
- 用户 ID 为参数，针对一段时间内频繁访问的用户 ID 进行限制

热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。



注意：

1. 热点规则需要使用@SentinelResource("resourceName")注解，否则不生效
2. 参数必须是7种基本数据类型才会生效

测试用例

```
@RequestMapping("/info/{id}")
@SentinelResource(value = "userinfo",
 blockHandlerClass = CommonBlockHandler.class,
 blockHandler = "handleException2",
 fallbackClass = CommonFallback.class,
 fallback = "fallback"
 )
public R info(@PathVariable("id") Integer id){
 UserEntity user = userService.getById(id);
  return R.ok().put("user", user);
}
```

配置热点参数规则

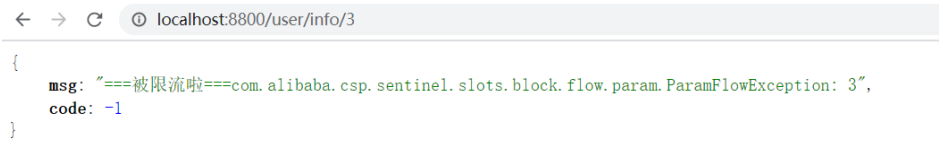
注意： 资源名必须是@SentinelResource(value="资源名")中 配置的资源名，热点规则依赖于注解

具体到参数值限流，配置参数值为3,限流阈值为1

测试：

http://localhost:8800/user/info/1 限流的阈值为3

http://localhost:8800/user/info/3 限流的阈值为1



系统规则

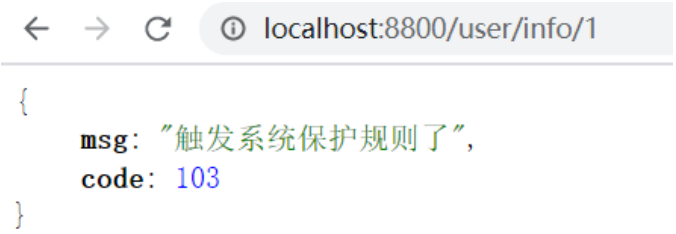
Sentinel 系统自适应限流从整体维度对应用入口流量进行控制，结合应用的 Load、CPU 使用率、总体平均 RT、入口 QPS 和并发线程数等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

- **Load 自适应**（仅对 Linux/Unix-like 机器生效）：系统的 load1 作为启发指标，进行自适应系统保护。当系统 load1 超过设定的启发值，且系统当前的并发线程数超过估算的系统容量时才会触发系统保护（BBR 阶段）。系统容量由系统的 `maxQps * minRt` 估算得出。设定参考值一般是 `CPU cores * 2.5`。
- **CPU usage**（1.5.0+ 版本）：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0），比较灵敏。
- **平均 RT**：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
- **并发线程数**：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。

编写系统规则

jemeter配置

测试结果



授权控制规则

很多时候，我们需要根据调用来源来判断该次请求是否允许放行，这时候可以使用 Sentinel 的来源访问控制（黑白名单控制）的功能。来源访问控制根据资源的请求来源（`origin`）限制资源是否通过，若配置白名单则只有请求来源位于白名单内时才可通过；若配置黑名单则请求来源位于黑名单时不通过，其余的请求通过。

来源访问控制规则（`AuthorityRule`）非常简单，主要有以下配置项：

- `resource`：资源名，即限流规则的作用对象。
- `limitApp`：对应的黑名单/白名单，不同 origin 用 `,` 分隔，如 `appA,appB`。
- `strategy`：限制模式，`AUTHORITY_WHITE` 为白名单模式，`AUTHORITY_BLACK` 为黑名单模式，默认为白名单模式。

配置授权规则

第一步：实现com.alibaba.csp.sentinel.adapter.spring.webmvc.callback.RequestOriginParser接口，在parseOrigin方法中区分来源，并交给spring管理

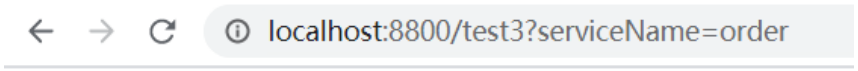
注意：如果引入CommonFilter，此处会多出一个

RequestOriginParser.java (sentinel-web-servlet-1.8.0-sources.jar\com\alibaba\csp\sentinel\adapter\servlet\callback)
RequestOriginParser.java (sentinel-spring-webmvc-adapter-1.8.0-sources.jar\com\alibaba\csp\sentinel\adapter\spring\webmvc\c

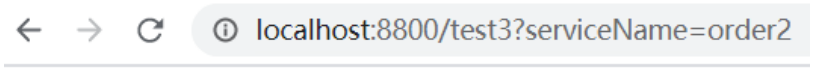
```
1 import com.alibaba.csp.sentinel.adapter.spring.webmvc.callback.RequestOriginParser;
2 import org.springframework.stereotype.Component;
3
```

```java
import javax.servlet.http.HttpServletRequest;

/**
 * @author Fox
 */
@Component
public class MyRequestOriginParser implements RequestOriginParser {
 /**
 * 通过request获取来源标识，交给授权规则进行匹配
 * @param request
 * @return
 */
 @Override
 public String parseOrigin(HttpServletRequest request) {
 // 标识字段名称可以自定义
 String origin = request.getParameter("serviceName");
// if (StringUtil.isBlank(origin)){
// throw new IllegalArgumentException("serviceName参数未指定");
// }
 return origin;
 }
}
```

测试：origin是order的请求不通过。




```
{
    msg: "授权规则不通过",
    code: 104
}
```


```
{
    id: 1,
    username: "fox",
    age: 31
}
```


文档：08-1 Sentinel控制台规则配置详解.not...
链接：http://note.youdao.com/noteshare?id=fc7d801d1a8213fc4d1b691302e82a62&sub=675ACF22A59841AFAECD58A0A9B4E151