# alispringcloud

4.RestTemplate 使用介绍 ，spring boot提供，参看ShareService类，学会post,get两种类型请求 get Object/Entity区别 , getEntity能获取对应的响应码

在ContentCenterApplication 中进行RestTemplate 对象Ioc容器注入，在使用的地方注入即可

5.DTO和domain对象区别在于DOMAIN多了很多注解用于持久化或表映射，而dto只作为单纯的对象使用很少加持久化相关注释

6.版本定义

<!--语义化的版本控制-->

<!--2：主版本，第几代-->

<!--1：次版本，一些功能的增加，但是架构没有太大的变化，是兼容的-->

<!--5：增量版本，bug修复-->

<!--release：里程碑，SNAPSHOT：开发版 M：里程碑 RELEASE：正式版-->

<version>2.1.5.RELEASE</version>

GREENWICH-SR1 (SERVICE-REALESE) 表示第一个bug修复版本，建议使用SR2版本间隔时间长，大部分问题得到解决

GREENWICH RELEASE 表示第一个正式发布版本

一般不用不维护版本以及M1,M2里程碑版本

7.spring cloud以及spring alibaba 整合到spring boot中，在pom.xml中加入如下代码即可，下面是整好的pom

<dependencyManagement>

<dependencies>

<!--整合spring cloud-->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Greenwich.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!--整合spring cloud alibaba-->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-alibaba-dependencies</artifactId>

<version>0.9.0.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

使用上面包中的工具依赖时，直接添加如下工具依赖即可，无需版本号

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>

</dependency>

8.参看nacos官网

https://nacos.io/zh-cn/docs/what-is-nacos.html

每个微服务实例都注册一个nacos client用于和nacos server进行通信，启动注册，停止关闭注册信息

启动nacos

Linux/Unix/Mac

启动命令(standalone代表着单机模式运行，非集群模式):

sh startup.sh -m standalone

如果您使用的是ubuntu系统，或者运行脚本报错提示[[符号找不到，可尝试如下运行：

bash startup.sh -m standalone

Windows

启动命令：

cmd startup.cmd

或者双击startup.cmd运行文件。

添加nacos依赖，启动注解不要了，自动添加

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>

</dependency>

添加配置nacos服务端配置,另外需要添加服务名称

cloud:

nacos:

discovery:

# 指定nacos server的地址

server-addr: localhost:8848

application:

# 服务名称尽量用-，不要用\_，不要用特殊字符

name: user-center

9.spring cloud 子项目依赖命名

spring-cloud-starter-alibaba{子项目名}-sentinel{模块名}

10.TestController 测试服务发现接口DiscoveryClient, 若有个服务实例下线了，再次查询就差不到了

@GetMapping("test2")

public List<ServiceInstance> getInstances() {

// 查询指定服务的所有实例的信息

// consul/eureka/zookeeper...

return this.discoveryClient.getInstances("user-center");

}

11.了解stream 以及lamda表达式使用，functional函数式编程

12.nacos的领域划分， namespace > group > service > cluster > instance , namespace用于划分dev,test,prod实现隔离, group用于划分多个微服务为一组， service用于区分微服务业务领域, cluster 作为微服务集群,cluster可以建设两个集群用于容灾，比如一个南京机房，一个北京机房，服务可以指定访问哪个机房集群服务，instance为单个微服务实例

13.在微服务中配置如上领域，在配置文件中进行元数据设置，也可以在nacos页面进行元数据设置

cloud:

nacos:

discovery:

# namespace: 56116141-d837-4d15-8842-94e153bb6cfb // 需要再nacos上新建再配置此处uuid

# NJ

# 指定集群名称

cluster-name: BJ // 设置该微服务属于南京机房的集群，在ribbon时设置调用

metadata: // 只能设置实例元数据，另外也可以通过nacos界面进行设置

instance: c

haha: hehe

version: v1

14.客户端负载均衡 ribbon，它会自动从nacos上获取注册的负载均衡列表实例信息 ,而不是服务端nginx负载均衡,添加ribbon依赖添加

spring-cloud-starter-alibaba-nacos-discovery已经带了ribbon包

添加注释@LoadBalanced 支持客户端负载均衡ribbon功能

@LoadBalanced

public RestTemplate restTemplate() {

RestTemplate template = new RestTemplate();

template.setInterceptors(

Collections.singletonList(

new TestRestTemplateTokenRelayInterceptor()

)

);

return template;

}

请求代码如下，会自动将user-center转化为对应的负载均衡分配的实例ip地址以及端口号

ResponseEntity<String> forEntity = restTemplate.getForEntity(

"http://user-center/users/{id}",

String.class, 2

);

15.ribbon 默认规则配置，通过接口实现自定义配置，其中负载均衡策略默认使用zoneavoidanceRule，在没有

zone时使用的是roundrobinrule规则，另外还有其他规则可以选择使用

方式一:

// 先定义ribbon规则，另外该类必须放在主类之外，避免被主启动程序扫描到因为它也是@Component，若放在主类中被扫描到，会出现该规则将作为全局的ribbon规则使用，其它Iping等接口也可以在如下类中进行注册实现

@Configuration

public class RibbonConfiguration {

@Bean

public IRule ribbonRule() {

return new RandomRule();

}

}

// 再将该规则设置到ribbonclient上， 可以通过设置name="user-center"指定访问对应的微服务使用对应的负载均衡方式

@Configuration

@RibbonClient(name="user-center", configuration = RibbonConfiguration.class)

public class UserCenterRibbonConfiguration {

}

方式二: 配置文件方式配置负载均衡 ，使用application.yml ,属性配置方式优先级比上面代码方式高，先用属性配置，实现不了再用代码配置

user-center: // 这里可以配置对应的微服务名使用如下负载均衡方法

ribbon:

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

16.ribbon的全局配置方式设置@RibbonClients 而不是上面@RibbonClient ，ribbon不支持全局文件配置

@Configuration

@RibbonClients(defaultConfiguration = RibbonConfiguration.class)

public class UserCenterRibbonConfiguration {

}

17. ribbon的饥饿加载方式配置，默认为懒加载造成首次ribbon访问很慢，若开启饥饿加载在程序启动时就会加载ribbon信息，初次访问url就很快

ribbon:

eager-load:

enabled: true

clients: user-center

18.自定义nacos的负载均衡权重，需要再微服务实例上设置对应的实例权重值，值越大被访问的概率越高

，下面配置好后，通过修改nacos服务实例对应的权重值进行测试，若将weight设置为0，则该服务实例将

不会再访问

首先自定义NacosWeightedRule规则

最后将该规则配置到文件或者如下java代码中

@Configuration

@RibbonClients(defaultConfiguration = RibbonConfiguration.class)

public class UserCenterRibbonConfiguration {

}

19.实现同机房优先调用，content-center属于bj机房会优先访问BJ集群下的user-center实例，若bj的user-center没有可用的实例，则会访问nj机房content-center实例

首先创建一个基于相同集群优先访问的规则类

NacosSameClusterWeightedRule

最后两个微服务实例配置如下

content-center配置BJ机房集群

cloud:

nacos:

discovery:

cluster-name: BJ // 设置cluster名称

user-center 分别配置BJ,NJ机房集群，

20 ，获取元数据信息，在元数据中设置对应微服务实例只能访问对应版本的元数据实例，在规则中进行判断

List<Instance> instances = namingService.selectInstances(name, true);

// instances.get(0).getMetadata(); 获取元数据信息

21.namespace用于资源隔离，服务只能访问同一个namespace下服务，namespace配置如下uuid,先在nacos界面进行配置

cloud:

nacos:

discovery:

namespace: 56116141-d837-4d15-8842-94e153bb6cfb

23.feign自带拦截器，支持自带的ribbon功能，缺省使用connencturl功能

24.feign的日志功能细粒度配置默认是没有开启的，开启后支持查看请求执行时间以及请求体的详细信息

指定包下的日志打印级别，这是spring-boot日志打印配置

logging:

level:

com.itmuch.contentcenter.feignclient.UserCenterFeignClient: debug

com.alibaba.nacos: error

方式一: feign日志级别是建立在配置的feign接口之上的

首先代码设置日志级别

/\*\*

\* feign的配置类

\* 这个类别加@Configuration注解了，否则必须挪到@ComponentScan能扫描的包以外,放入到ribbonconfiguration包中，参看spring 父子上下文重复扫描问题

\*/

public class GlobalFeignConfiguration {

@Bean

public Logger.Level level(){

// 让feign打印所有请求的细节，这里有四种级别，默认为NONE什么都不打印，生成用basic

return Logger.Level.FULL;

}

}

再将上面配置注册到feignclient接口上

@FeignClient(name = "user-center", configuration = GlobalFeignConfiguration.class)

public interface UserCenterFeignClient {

最后再配置文件中设置如下

logging:

level:

com.itmuch.contentcenter.feignclient.UserCenterFeignClient: debug // 接口包类 + 日志级别

方式二: 使用配置文件方式

@FeignClient(name = "user-center")

public interface UserCenterFeignClient {

/\*\*

\* http://user-center/users/{id}

\*

\* @param id

\* @return

\*/

@GetMapping("/users/{id}")

UserDTO findById(@PathVariable Integer id);

}

feign接口配置如下

feign:

client:

config:

user-center:

loggerLevel: full

25. feign的全局配置

方式一： 代码配置

/\*\*

\* feign的配置类

\* 这个类别加@Configuration注解了，否则必须挪到@ComponentScan能扫描的包以外

\*/

public class GlobalFeignConfiguration {

@Bean

public Logger.Level level(){

// 让feign打印所有请求的细节

return Logger.Level.FULL;

}

}

在启动类进行配置全局

@EnableFeignClients(defaultConfiguration = GlobalFeignConfiguration.class)

public class ContentCenterApplication {

方式二： 配置文件配置

feign:

client:

config:

# 全局配置

default:

loggerLevel: full

26.feign 的配置项可以设置超时，重试策略等，拦截器等，具体参看feign文档

27.feign接口继承 抽象出来形成父接口， controller实现该接口， feign接口继承该接口添加注解feignclient即可，把该接口单独形成一个maven模块被公共

28.feign参数传递

首先在user-center中TestController 中定义如下用户查询服务接口

// q?id=1&wxId=aaa&...

@GetMapping("/q")

public User query(User user) {

return user;

}

其次在content-center中定义如下

// 定义feign接口 ，需要加上@SpringQueryMap 注释，否则报错，但用@SpringQueryMap这种feign注释的接口参数传递是无法被继承的使用的，而@RequestParam一个个参数定义时可以进行继承使用的 ，对于post 使用的 @ResponseBody和spring 是一样使用可以被继承

@FeignClient(name = "user-center")

public interface TestUserCenterFeignClient {

@GetMapping("/q")

UserDTO query(@SpringQueryMap UserDTO userDTO);

}

// 在TestController 中测试调用feign接口

@Autowired

private TestUserCenterFeignClient testUserCenterFeignClient;

@GetMapping("test-get")

public UserDTO query(UserDTO userDTO) {

return testUserCenterFeignClient.query(userDTO);

}

29.feign脱离ribbon的使用方式，如下定义

@FeignClient(name = "baidu", url = "http://www.baidu.com")

public interface TestBaiduFeignClient {

@GetMapping("")

String index();

}

// 测试使用如下方式

@Autowired

private TestBaiduFeignClient testBaiduFeignClient;

@GetMapping("baidu")

public String baiduIndex() {

return this.testBaiduFeignClient.index();

}

30.feign的性能优化，1.连接池配置， feign的底层默认使用urlconnect, 另外支持apache http以及okhttp替换，2.日志级别优化，这些优化可以提升15%性能，若要使用最高性能直接使用template即可

首先添加如下依赖

<dependency>

<groupId>io.github.openfeign</groupId>

<artifactId>feign-httpclient</artifactId>

</dependency>

其次文件属性配置,配置httpclient相关属性配置

feign:

httpclient:

# 让feign使用apache httpclient做请求；而不是默认的urlconnection

enabled: true

# feign的最大连接数

max-connections: 200

# feign单个路径的最大连接数

max-connections-per-route: 50

31.雪崩效应， 一次请求访问就占用一个线程，线程长时间等待会占用cpu,内存等资源使用，若不做处理，则该服务所在的所有线程资源会被耗尽,容灾方案如下

超时：最简单的方式，只要超过时间点自动断开连接释放资源

限流：给服务实例配置阈值，例如800qps,当请求超过该值则会被拒绝，譬如你给我三碗，我只吃一碗

仓壁模式： 给每个controller分配单独的线程池资源，其中一个资源耗尽不会影响其它请求，譬如不把鸡蛋放在一个篮子里

断路器模式: 最高端的容错方式，设置指定时间内发生错误次数，或者直接设置发生错误阈值，达到阈值就不访问远程api,进入半开状态，隔断时间允许一次访问，若成功则关闭断路器，若还不行继续关闭

32.sential是流量控制，服务降级以及，熔断的库，整合sentinal,结合actuator进行测试

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

配置如下，开启显示所有actuator 端点服务查看

management:

endpoints:

web:

exposure:

include: '\*'

endpoint:

health:

show-details: always

访问http://localhost:8080/actuator/sentinel 即可查看端点信息

33.查看项目sentinel所依赖pom的包，找到对应的sentinel服务版本使用,保证兼容性问题,查看sentinel-core对应的版本号

整合sentinel控制台

cloud:

sentinel:

transport:

# 指定sentinel 控制台的地址

dashboard: localhost:8080

测试必须进行访问后，才会在sentinel控制台进行显示，说明sentinel是懒加载的，ribbon也是懒加载的

启动

java -Dserver.port=8088 -jar sentinel-dashboard-1.6.3.jar 指定端口

访问：http://localhost:8080 , 用户名和密码：sentinel

启动rocket

start mqnamesrv.cmd --启动命名服务

start mqbroker.cmd -n 127.0.0.1:9876 autoCreateTopicEnable=true --启动borker

启动zipkin

java -jar zipkin-server-2.12.9-exec.jar

访问http://localhost:9411/zipkin/

测试访问，带token

使用idea restservice 插件

X-Token: eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwid3hOaWNrbmFtZSI6IuWkp-ebriIsInJvbGUiOiJhZG1pbiIsImlhdCI6MTU3MDc3NDEzOCwiZXhwIjoxNTcxOTgzNzM4fQ.mxDGa0JXQfQ2wWClv3knXxgO1uU3SnPCDgpAvg13c-g

id : 1

http://localhost:8010/shares/{id}

34.sentinel中操作

流控： 资源名：唯一标识(默认是路径，可以自行定义名称) ，针对来源: 设置对应的微服务名，默认default标识不针对来源全部控制， 阈值类型: 线程或者qps, 达到多少时限流 ，测试设置限流值为1， 请求http://localhost:8010/shares/1 就会提示限流

返回

{

"status": 100,

"msg": "限流了"

}

流控模式关联: 表示关联的资源达到阈值就限流自己

测试时可以关联一个端点资源/actuator/sentinel ，启动content-center中SentinelTest类，应用场景为针对同一个表查询和修改的接口，若查询很多势必会影响修改，此时要根据业务判断时读优先还是写优先，若设置关联资源是修改，表示修改到达一定赋值，则写接口资源就会关闭，用如下main1测试启动

public static void main1(String[] args) throws InterruptedException {

RestTemplate restTemplate = new RestTemplate();

for (int i = 0; i < 10000; i++) {

String object = restTemplate.getForObject("http://localhost:8010/actuator/sentinel", String.class);

Thread.sleep(500);

}

}

测试访问http://localhost:8010/shares/{id} 限流

链路资源：配置入口资源，只记录链路上流程，表示从入口资源上进来的流量就会限流

测试查看content-center: TestController

@GetMapping("test-a")

public String testA() {

this.testService.common();

return "test-a";

}

@GetMapping("test-b")

public String testB() {

this.testService.common();

return "test-b";

}

TestService类

@Slf4j

@Service

public class TestService {

@SentinelResource("common") 添加监控细粒度资源名

public String common() {

log.info("common....");

return "common";

}

}

选择链路流控模式(细粒度的api控制模式)，在入口资源设置/test-a(此时sentinel就会统计该端点资源使用情况，不管/test-b)，此时只有/test-a访问qps超过1就会限流，而/test-b则不会限流，

测试访问http://localhost:8010/test-a 会限流

warn up 预热因子方式，阈值/预热因子(默认是3)，经过过长时间才开始恢复正常的qps， 若设置qps:100, 加热时长10秒，那么会设置从 100/3作为初始阈值--》 经过10s后达到 --> 100 qps, 场景秒杀活动，若一开始就流量很大会直接打死对应的微服务，需要慢慢的过程再达到阈值效果

排队等待：让请求以匀速方式一一通过，但阀值类型必须设成qps,其它线程模式则该无效，应用场景，有时服务请求比较空闲，有时比较繁忙，所以可以考虑用队列方式

测试类SentinelTest，使用如下方式测试，循环执行不会被限流，控制台打印很匀速

public static void main(String[] args) {

RestTemplate restTemplate = new RestTemplate();

for (int i = 0; i < 10000; i++) {

String object = restTemplate.getForObject("http://localhost:8010/test-a", String.class);

System.out.println("-----" + object + "-----");

}

}

35.服务降级(官网说明降级策略描述不准确，以degrade降级类代码为准)

降级规则策略：

RT策略(秒级系统统计)：平均响应时间超过阀值设置的RT响应时间(默认4900ms,即使设置最大也以4900ms计算,若要修改设置-Dcsp.sentinel参数才有效) 或者在窗口内的请求次数超过>=5次，就会触发降级，断路器就会打开，默认5s时间后断路器窗口结束

RT平均响应时间，时间窗口

异常数(分钟系统统计) 设置的异常数阀值为10,表示一分钟内统计的异常数超过时就会出现服务降级，并且在设置的时间窗口时间到了关闭服务降级，若窗口时间到了发现异常数依然大于10就会继续服务降级，因此若时间窗口设置 < 60s 可能会出现问题，时间窗口必须设置大于60s以上

异常比列(秒级系统统计) qps> 5 && 异常比列超过阀值

总结断路器一般有三种状态，打开，半开，关闭，而sentinel目前版本只有打开和关闭两种状态

36.热点规则设置，需要代码支持，应用场景为对单独api端点中指定热点参数进行流量控制，而其它参数无需控制的情况，该热点参数资源必须为基本类型

设置端点接口参数(根据索引指定)在指定窗口时间内达到指定阈值就开始进行限流，非指定的参数不会限制

content-center的类为TestController定义如下

@GetMapping("test-hot")

@SentinelResource("hot") // 需要设置该端点热点规则名

public String testHot(

@RequestParam(required = false) String a,

@RequestParam(required = false) String b

) {

return a + " " + b;

}

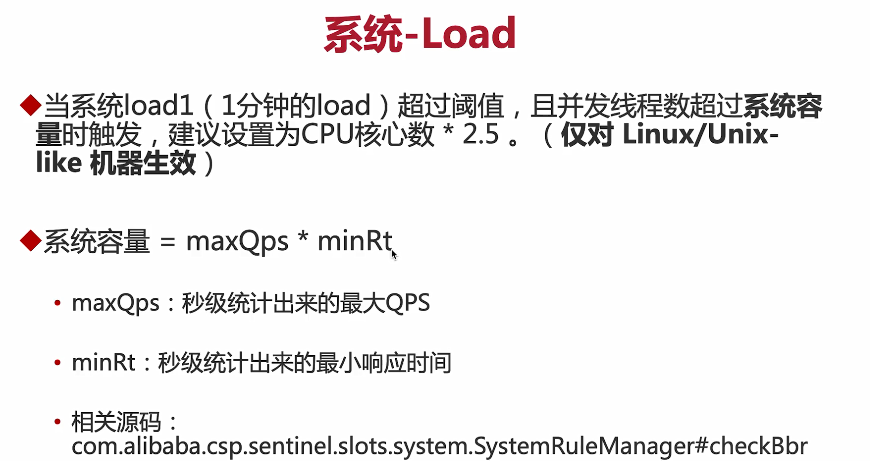
测试访问http://localhost:8010/test-hot ，在sentinel控制台出现hot端点，对其进行热点规则设置，另外在高级选项中设置参数类型string，参数值5，限流阈值1000 表示对5参数值单独例外设置

测试http://localhost:8010/test-hot? a=5?b=demoData则不会限流而报错了

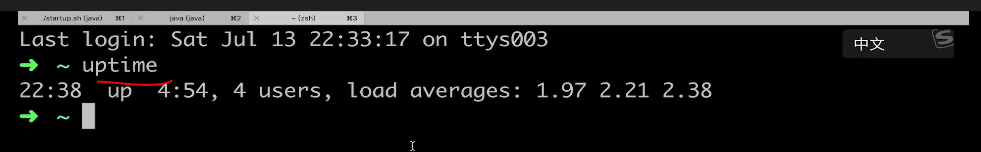
36.系统保护规则类

阀值类型包括四种: load ,rt,线程数, 入口qps

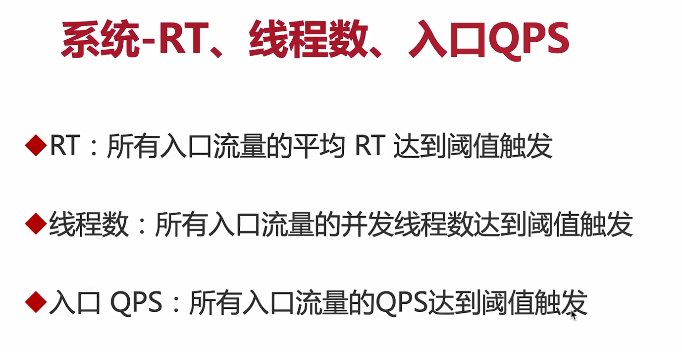
若核数为4 ，则设置load阈值为10



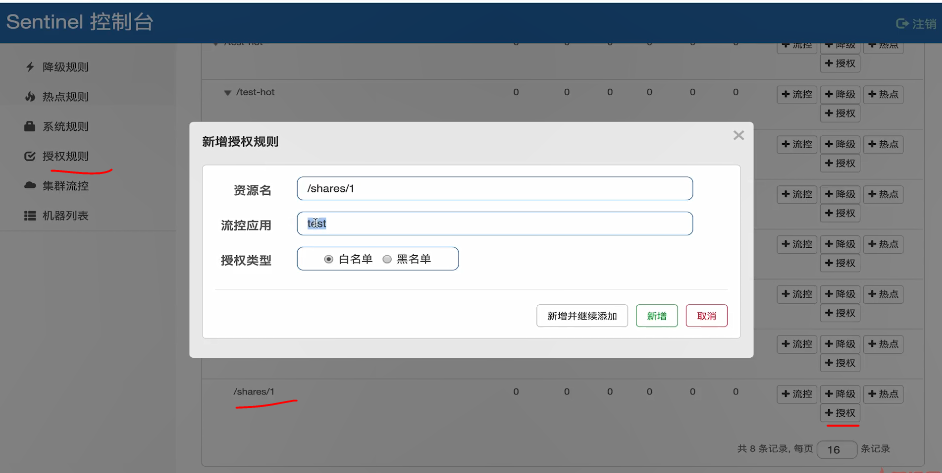
查看load负载，一分钟，5分钟，15分钟 三种平均负载值



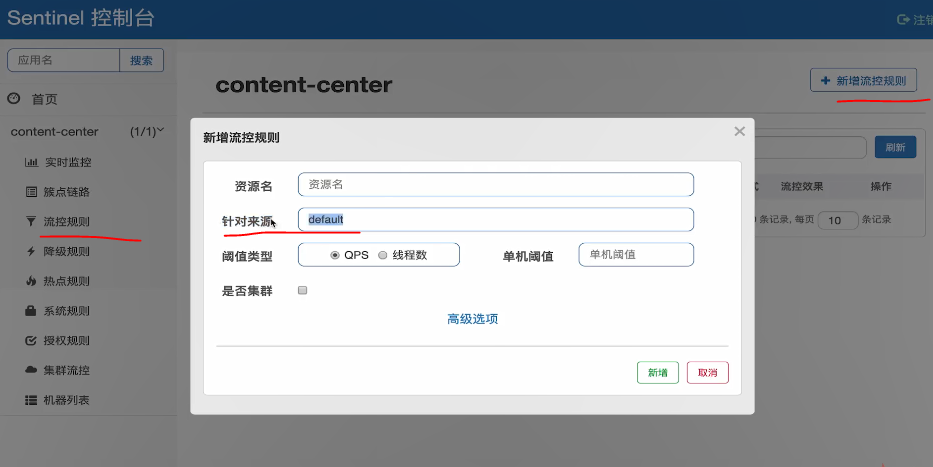
记住是所有入口资源总量



授权规则表示如下白名单该端点资源只能被微服务test 所访问，黑名单表示该资源不能被test微服务所访问



这与新增流控规则针对来源是相同的，需要扩展的



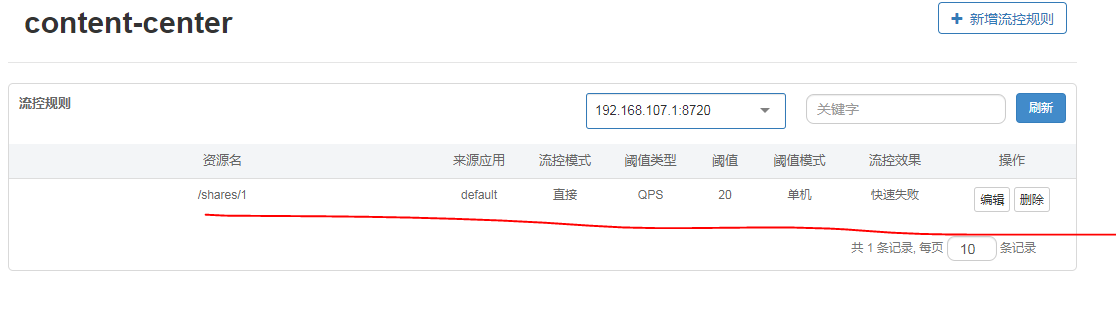
37.通过代码配置规则，参看如下文档,一般都通过界面控制



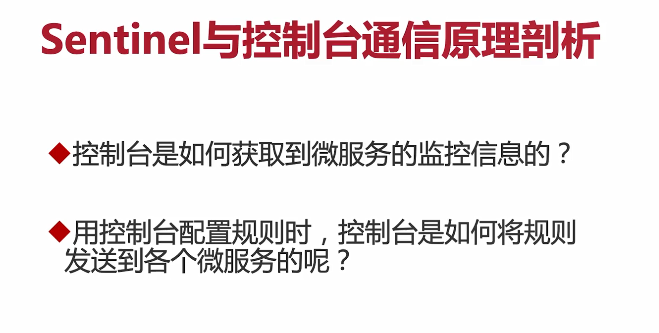
参看content-center的TestController类

@GetMapping("test-add-flow-rule")  
public String testHot() {  
 this.initFlowQpsRule();  
 return "success";  
}  
  
private void initFlowQpsRule() {  
 List<FlowRule> rules = new ArrayList<>();  
 FlowRule rule = new FlowRule("/shares/1");  
 // set limit qps to 20  
 rule.setCount(20);  
 rule.setGrade(RuleConstant.*FLOW\_GRADE\_QPS*);  
 rule.setLimitApp("default");  
 rules.add(rule);  
 FlowRuleManager.*loadRules*(rules);  
}

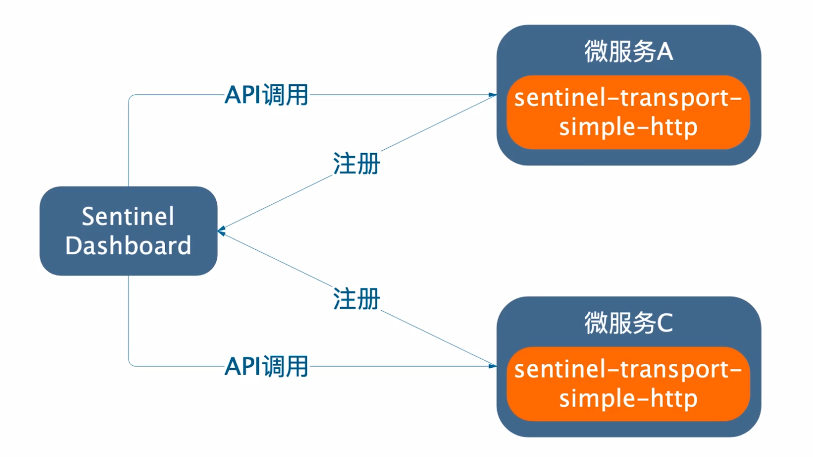
访问<http://localhost:8010/test-add-flow-rule> 进行规则初始化，就会多出如下信息



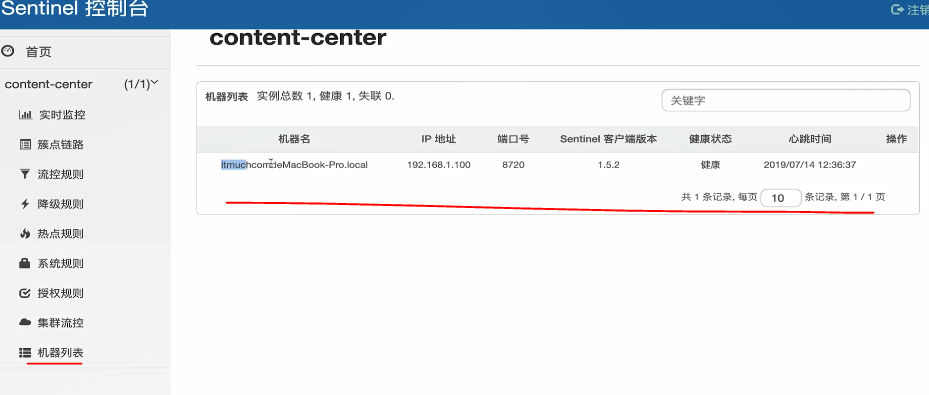
38.sentinel控制台原理



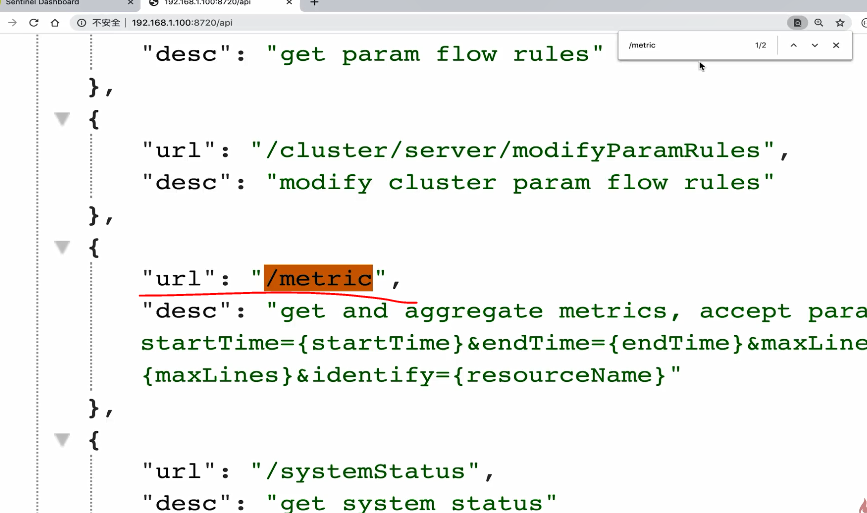
Sentinel类似服务发现服务端，将集成sentinel的客户端的微服务注册到上面



通过sentinel的机器列表进行查看，8720是sentinel之间通信端口



微服务服务调用sentine的api获取对应的规则数据，并且修改规则



下面表示内容中心用什么端口和sentinel进行通信

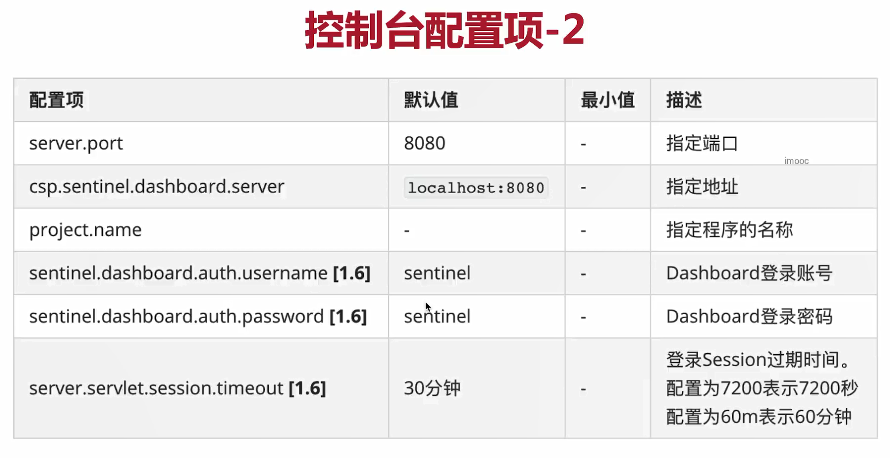




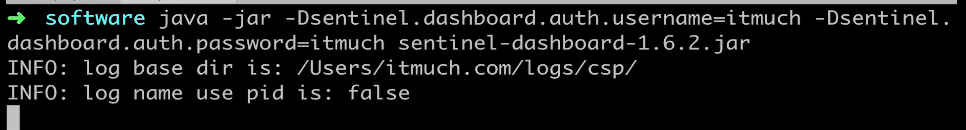
39. 应用端配置项目，该8719为控制台和该服务通信端口



下面是控制台配置项目



在启动控制台进行设置



40. sentinel的api 详解

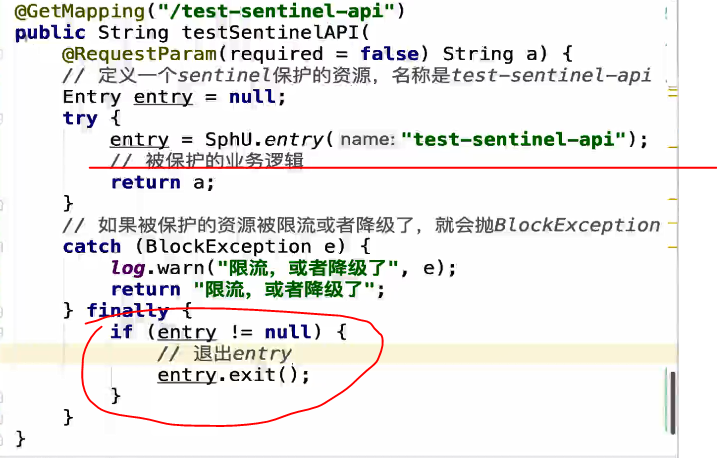
先关闭端点保护，此时sentinel就不会对其进行端点控制

**cloud**:**sentinel**:  
 **filter**:  
 *# 打开/关闭掉对Spring MVC端点的保护* **enabled**: **false**

此时访问[http://localhost:8010/shares/{id}](http://localhost:8010/shares/%7bid%7d) 不会在如下控制台显示



定义如下，用代码方式设置需要保护端点资源，名字自己定义，下面是对应的业务逻辑，用完该资源后，最后要进行释放即可，定好后sentinel就会对该端点资源的输入qps,线程数，以及调用外部出错的 RT, 异常比例，异常数进行统计控制， entry会根据sentinel的统计规则进行校验不满足则会抛出blockexception错误



测试访问<http://localhost:8010/test-sentinel-api>，查看sentinel控制台，出现如下端点资源，资源显示如下

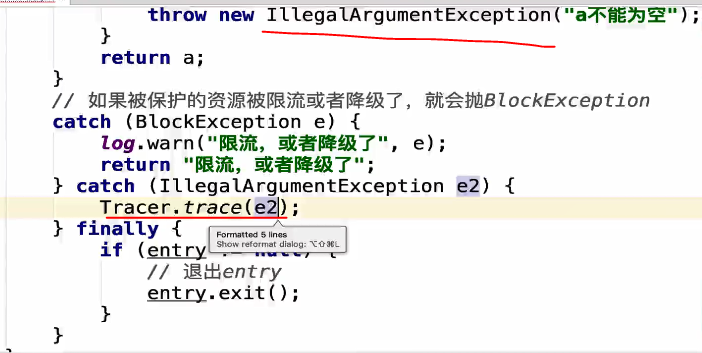


给其设置规则测试访问<http://localhost:8010/test-sentinel-api>， 显示服务降级或限流 了

测试设置异常比例为如下



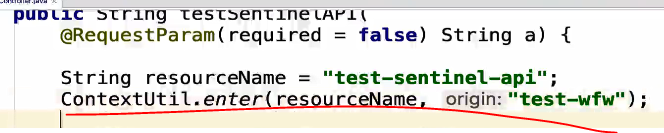
默认对illegalargumentexception 异常，sentinel是不会进行统计的，只会对blockexception进行统计，需要设置如下tracer.trace(e2)进行记录



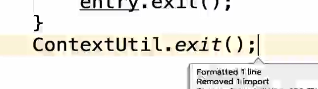
设置针对来源



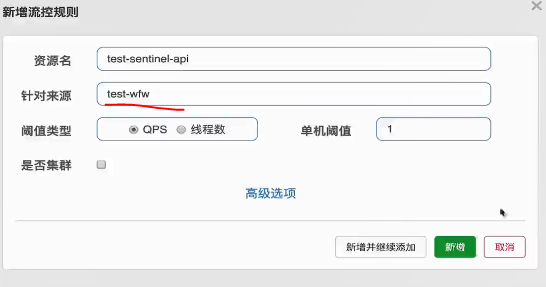
设置来源



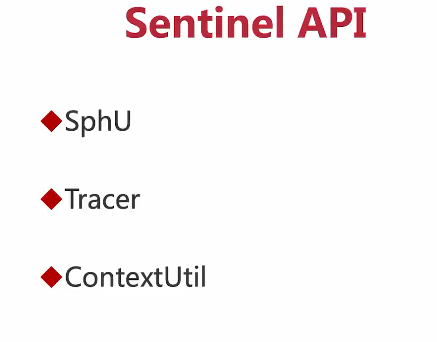
来源也要关闭



设置流控来源，对于来源如下名字的端点资源进行控制，非来源如下配置的服务不会进行规则控制，测试访问



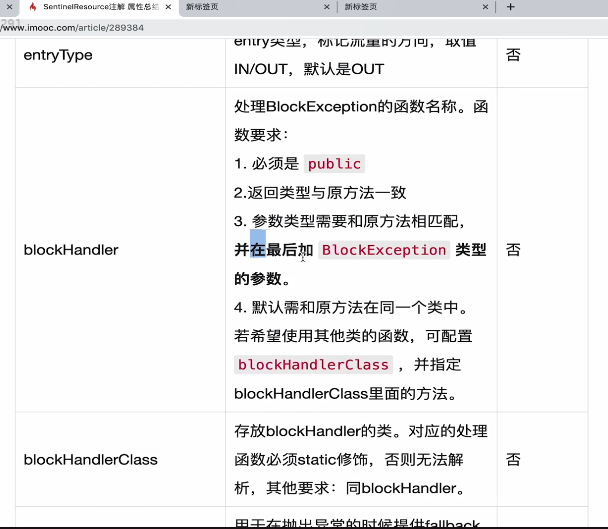
设置调用端点资源，跟踪资源点，以及设置对应来源



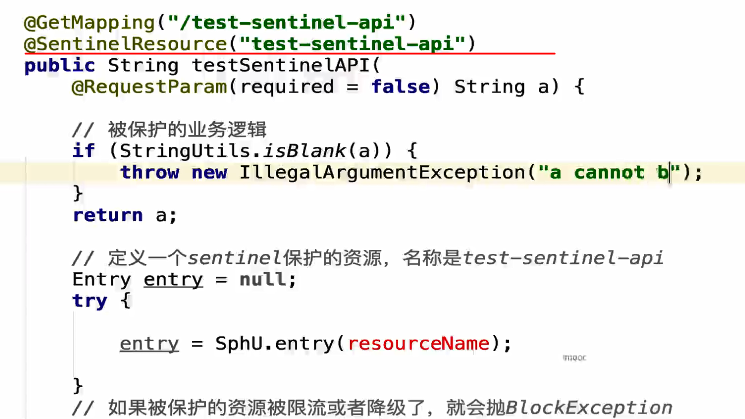
41．Sentinelresource注解使用



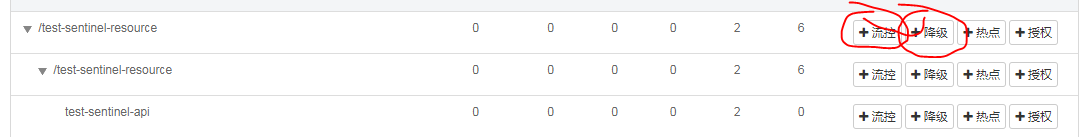
参看如下使用文档



使用注解方式设置如上代码



配置流控规则以及服务降级规则



分别对应如下配置的回掉方法调用

@SentinelResource(  
 value = "test-sentinel-api",  
 blockHandler = "block", // 与流控规则对应使用  
 blockHandlerClass = TestControllerBlockHandlerClass.class,  
 fallback = "fallback" // 与降级规则对应使用  
)

方法*需要加上static修饰符*

public class TestControllerBlockHandlerClass {  
 */\*\*  
 \* 处理限流或者降级，需要加上static修饰符  
 \*  
 \** ***@param*** *a  
 \** ***@param*** *e  
 \** ***@return*** *\*/* public static String block(String a, BlockException e) {  
 *log*.warn("限流，或者降级了 block", e);  
 return "限流，或者降级了 block";  
 }  
}

42．Resttemplate 整合sentinel



通过@SentinelRestTemplate 整合

@Bean  
@LoadBalanced  
@SentinelRestTemplate  
public RestTemplate restTemplate() {  
 RestTemplate template = new RestTemplate();  
 template.setInterceptors(  
 Collections.*singletonList*(  
 new TestRestTemplateTokenRelayInterceptor()  
 )  
 );  
 return template;  
}

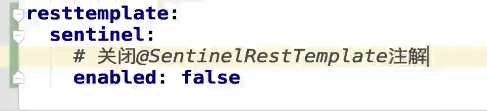
通过如下方式测试，并且给簇点GET:http://user-center 设置流控规则

[http://localhost:8010/test-rest-template-sentinel/{userId}](http://localhost:8010/test-rest-template-sentinel/%7buserId%7d)

template的sentinel的开关

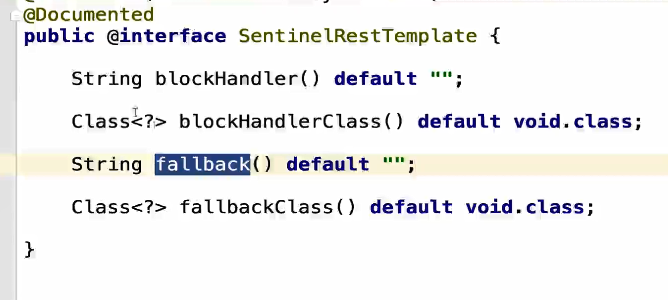


关闭注解功能, yml 配置文件，用于在开发时不希望受sentinel 干扰，需要进行关闭



**resttemplate**:  
 **sentinel**:  
 *# 设置成false，表示关闭@SentinelRestTemplate注解* **enabled**: **true**

对于template的流控与降级，如下，实现如下接口



43. feign整合sentinel,只要添加如下配置即可



**feign**:  
 **sentinel**:  
 *# 为feign整合sentinel* **enabled**: true

配置 GET:http://user-center/users/{id} 端点资源 ,会报限流错误

测试 [http://localhost:8010/shares/{id}](http://localhost:8010/shares/%7bid%7d)

给feign提供自定义限流错误处理