# 系统保护策略

在互联网应用中，会有很多突发性的高并发访问常见，比如双11大促、秒杀等。这些场景最大的特点就是访问量会远远超出系统所能够处理的并发数。在没有任何保护机制的情况下，如果所有的流量都进入服务器，很可能造成服务器宕机导致整个系统不可用，从而造成巨大的损失。为了保证系统在这些场景中仍然能够稳定运行，就需要采取一定的系统保护策略，常见的策略有服务降级、限流和熔断等。

## 服务限流的作用及实现

1. **什么是限流**

限流的主要目的是通过限制并发访问数或者限制一个时间窗口内允许处理的请求数量来保护系统，一旦达到限制数量则对当前请求进行处理采取对应的拒绝策略，比如跳转到错误页面拒绝请求、排队进入系统、降级等。从本质上说，限流的主要作用是损失一部分用户的可用性，为大部分用户提供稳定可靠的服务。比如系统当前能够处理的并发数是10万，如果此时来了12万用户，那么限流机制会保证为10万用户提供正常服务。



**2、应用**

在实际开发中，限流几乎无处不在：

·在Nginx层添加限流模块限制平均访问速度。

·通过设置数据库连接池、线程池的大小来限制总的并发数。

·通过Gnava提供的Ratelimiter限制接口的访问速度。

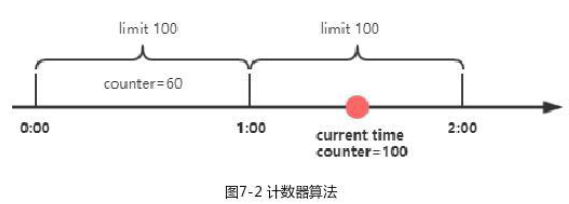
·TCP通信协议中的流量整形。

要实现限流，最重要的是限流的算法。

## 常见限流算法

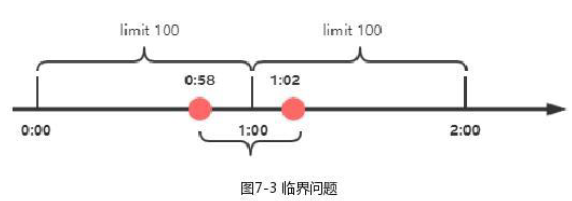
要实现限流，最重要的是限流的算法。常见的限流算法有计数器算法、滑动窗口算法、令牌桶限流算法、漏桶限流算法。

1. **计数器算法**
2. 计数器算法是一种比较简单的限流实现算法，在指定周期内累加访问次数，当访问次数达到设定的阈值时，触发限流策略，当进入下一个时间周期时进行访问次数的清零。示意图如下：



在上图中，限定了每一分钟能够处理的总的请求数为100，在第一个一分钟内，一共请求了60次。接着到第二个分钟，counter又从0开始计数，在一分半钟时，已经达到了最大限流的阈值，这个时候后续的所有请求都会被拒绝。这种算法可以应用在短信发送的频次限制上，比如限制同一个用户一分钟之内触发短信发送的次数。

2）但计数器算法存在一个临界问题，如下图所示，

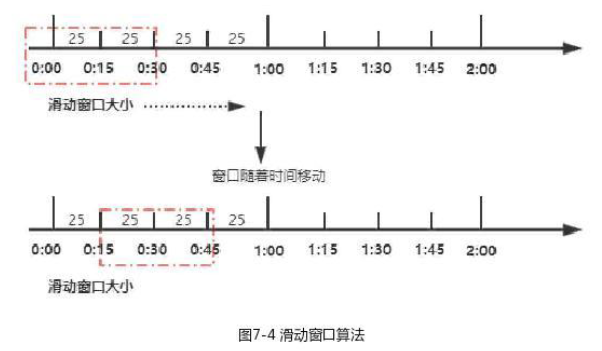


在第一个分钟的0:58和第二分钟的1:02这个时间段内，分别出现了100个请求，整体来看就会出现4秒内总的请求量达到200，超出了设置的阈值。

1. **滑动窗口算法**

为了解决计数器算法带来的临界问题，所以引入了滑动窗口算法。滑动窗口算法是一种流量控制技术，在TCP网络通信协议中，就采用了滑动窗口算法来解决网络拥堵的情况。

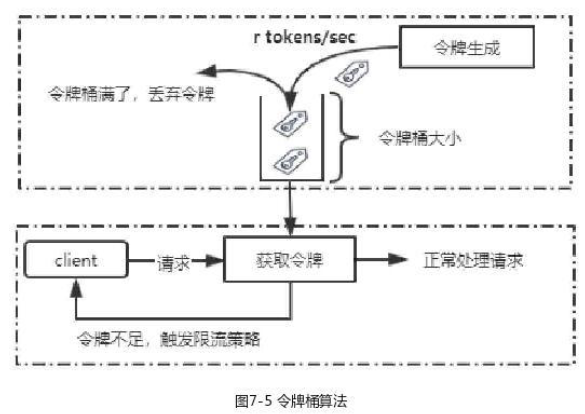
简单来说，滑动窗口算法的原理是在固定窗口中分割出多个小时窗口，分别在每个小时间窗口中记录访问次数，然后根据时间将窗口往前滑动并删除过期的小时间，最终只需要统计滑动窗口范围内所有小时间窗口总的计数即可。如下示意图：



我们将一分钟拆分为4个小时间窗口，每个小时间窗口最多能够处理25个请求。并且通过虚线框表示滑动窗口的大小（当前窗口的大小是2，也就是在这个窗口内最多能够处理50个请求）。同时滑动窗口会随着时间往前移动，比如前面15s结束之后，窗口会滑倒15~45s这个范围，然后在新的窗口中重新统计数据。这种方式很好地解决了固定窗口算法的临界值问题。

Sentinel就是采用滑动窗口算法来实现限流的。

1. **令牌桶限流算法**
2. 令牌桶是网络流量整形（Traffic Shapping）和速度限制（Rate Limiting）中最常使用的一种算法。对于每一个请求，都需要从令牌桶中获得一个令牌，如果没有获得令牌，则需要触发限流策略。如下图所示：



系统会以一个恒定速度（r tokens/sec）往固定容量的令牌桶中放入令牌，如果

此时有客户端请求过来，则需要先从令牌桶中拿到令牌已获得访问资格。

1. 假设令牌生成速度是每秒10个，也就等同于QPS=10，此时在请求获取令牌的时候，会存在三种情况：

·请求速度大于令牌生成速度：那么令牌会很快被取完，后续再进来的请

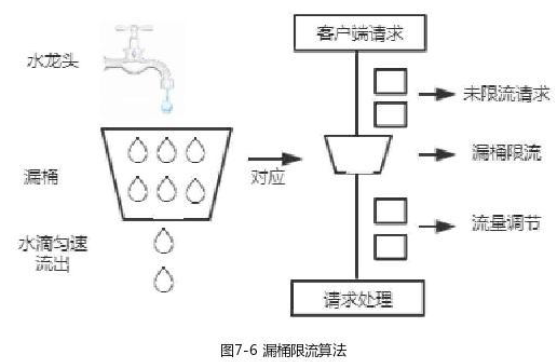
求会被限流。

·请求速度小于令牌生成速度：此时流量处于平稳状态

·请求速度小于令牌生成速度：说明此时系统的并发数并不高，能正常处

理请求。

1. 由于令牌桶有固定的大小，当请求速度小于令牌生成速度时，令牌桶会被填满。所以令牌桶能够处理突发流量，也就是在短时间内新增的流量系统能够正常处理，这是令牌桶的特性。
2. **漏桶限流算法**
3. 漏桶限流算法的主要作用是控制数据注入网络的速度，平滑网络上的突发流量。漏桶限流算法的原理图如下：



在漏桶算法内部同样维护一个容器，这个容器会以恒定速度出水，不管上面的水

流速度多快，漏桶水滴的流出速度始终保持不变。实际上消息中间件就使用了漏

桶限流的思想，不管生产者的请求量有多大，消息的处理能力取决于消费者。

1. 在漏桶限流算法中，存在以下几种可能的情况：

·请求速度大于漏桶流出水滴的速度：请求数超过当前服务所能处理的极限，

将会触发限流策略。

·请求速度小于或者等于漏桶流出水滴的速度：服务端的处理能力正好满足

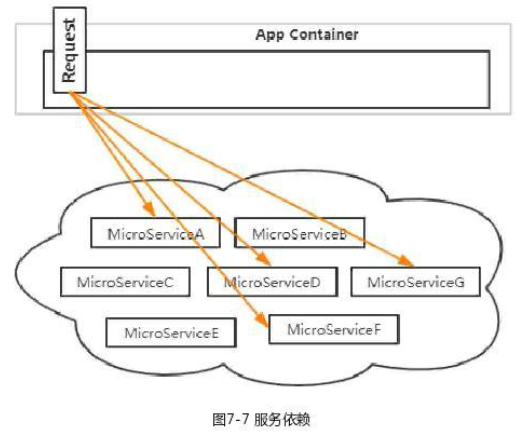
客户端的请求量，将正常执行。

1. 漏桶限流算法和令牌桶限流算法的实现原理相差不大，最大的区别是漏桶无法处理短时间内的突发流量，漏桶限流算法是一种恒定速度的限流算法。

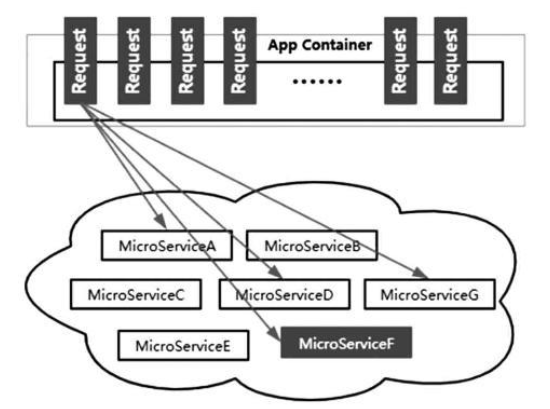
## 服务熔断与降级

1. **什么是服务熔断**

在微服务架构中，由于服务粒度较细，会出现请求链路较长的情况。如下图所示，用户发起一个请求，需要调用多个微服务才能完成。



在高并发常见中，这些依赖服务的稳定性对系统的影响非常大，比如某个服务因为网络延迟或者请求超时等原因不可用时，就会导致当前请求阻塞，如下图所示，一旦某个链路上被依赖的服务不可用，很多能出现请求堆积从而导致出现雪崩效应（业务系统崩溃）。



所以服务熔断就是用来解决这个问题的方案。服务熔断是指当某个服务提供者无法正常为服务调用者提供服务时，比如请求超时、服务异常等，为了防止整个系统出现雪崩效应，暂时将出现故障的接口隔离出来，断绝与外部接口的联系，当触发熔断之后，后续一段时间内该服务调用者的请求都会直接失败，直到目标服务恢复正常。

1. **服务降级的参考标准**

服务降级需要有一个参考指标，一般来说有以下几种常见方案：

·平均响应时间：比如1s内持续进入5个请求，对应时刻的平均响应时间均超过

阈值，那么接下来在一个固定的时间窗口内，对这个方法的访问都会自动熔断。

·异常比例：当某个方法每秒调用所获得的异常总数的比例超过设定的阈值时，该

资源会自动进行降级状态，也就是在接下来的一个固定时间窗口中，对这个方

法的调用都会自动返回。

·异常数量：和异常比例类似，当某个方法在指定时间窗口内获得的异常数量超过

阈值时会触发熔断。

Sentinel也提供了熔断功能。

# 分布式限流框架Sentinel

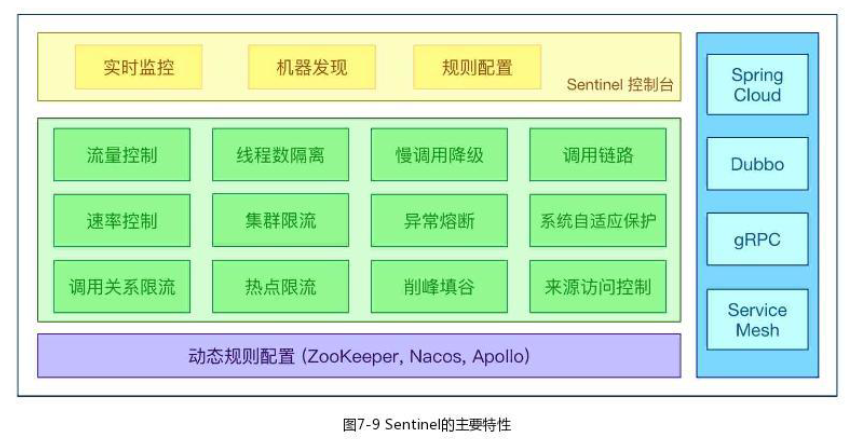
## 什么是Sentinel

Sentinel是面向分布式服务架构的轻量级流量控制组件，主要以流量为切入点，从限流、流量整形、服务降级、系统负载保护等多个维度来帮助我们保障微服务的稳定性。

在阿里巴巴内部有一句口号“稳定压倒一切”，稳定性是系统的基础能力，稳定性差的系统会出现服务超时或服务不可用，给用户带来不好的体验，从而对业务造成影响。所以系统稳定向是一条“红线”，任何业务需求或技术架构升级都不应该越过它。目前，Sentinel在阿里内部被广泛使用，为多年双11等大促活动保驾护航，并且，Sentinel开源以后也被很多互联网企业采用。

## Sentinel的特性

如下图所示，Sentinel的特性非常多。



·丰富的应用场景：几乎涵盖所有的应用场景，例如秒杀（即突发性流量控制在系统容

量可以承受的范围）、消息削峰填谷、集群流量控制等。

·实时监控：Sentinel提供了实时监控功能。开发者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台以下规模的集群汇总运行情况。

·开源生态支持：Sentine提供开箱即用的与其他开源框架/库的整合，如与Spring Cloud

Dubbo、gRPC的整合。开发者只需要引入相应的依赖并进行简单的配置即可快速接入

Sentinel。

·SPI扩展支持：Sentinel提供了SPI扩展点支持，开发者可以通过扩展点来定制化限流

规则，动态数据源适配等需求。

## Sentinel的组成

Sentinel分为两个部分：

·核心库（Java客户端）：不依赖任何框架/库，能够运行于所有Java运行时环境，

同时对Dubbo、Spring Cloud等框架也有较好的支持（通常是一个项目或服务）。

·控制台（Dashboard）：基于Spring Boot开发，打包后可以直接运行，不需要额

外的Tomcat等应用容器。

## Sentinel Dashboard的部署

**1、什么是Sentinel Dashboard**

Sentinel Dashboard是Sentinel提供的一个轻量级的开源控制台，它支持机器发现，以及健康情况管理、监控（单击和集群）、规则管理和推送的功能。

1. **Sentinel Dashboard安装步骤**

Sentinel Dashboard安装步骤如下：

·直接下载源码通过mvn clean package自己构建

·直接在Release页面下载已经构建号的Jar

1. **启动Sentinel Dashboard**

通过以下命令启动控制台：

java -Dserver.port=7777 -Dcsp.sentinel.dashboard.server=localhost:7777 -Dproject.name=sentinel-dashboard -jar sentinel-dashboard-1.7.1.jar

其中，启动参数如下：

·-Dserver.port：指定Sentinel控制台的访问端口，默认是8080

·-Dcsp.sentinel.dashboard.server：指定Sentinel Dashboard控制台的IP地址和

端口，这是进行设置的目的是把自己的限流数据暴露到监控平台。

·-Dproject.name：设置该控制台项目名称

启动之后，就可以在浏览器中访问Sentinel控制台了，从Sentinel 1.6.0开始，Sentinel控制台引入了基本的登录功能，默认用户名和密码都是sentinel。

# Sentinel的基本应用

## 限流实现步骤

使用Sentinel的核心库来实现限流，主要分以下几个步骤。

·定义资源

·定义限流规则

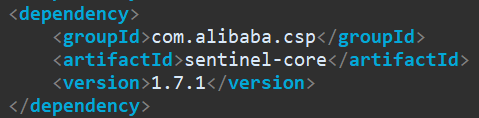
·检验规则是否生效

所谓的资源，就是需要通过限流保护的最基本元素，比如一个方法。有了需要保护的资源之后，就可以针对该资源设置流量控制规则了。

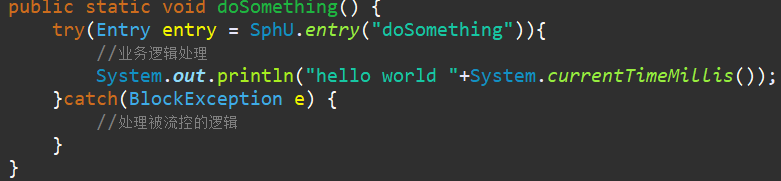
## Sentinel实现限流

通过一个简单的案例来演示一下Sentinel的基本使用方法。

1. 首先，引入Sentinel的核心库

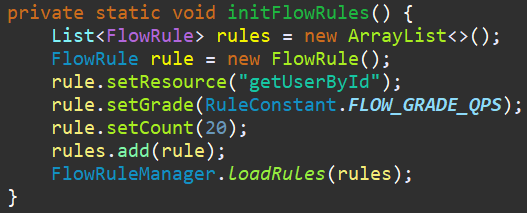


1. 定义一个普通的业务方法，充当需要保护的资源



在doSomething方法中，通过使用Sentinel中的SphU.entry(“doSomething”)定义一个资源来实现流控的逻辑，它表示当请求进入doSomething方法时，需要进行限流判断。如果抛出BlockException异常，则表示触发了限流。

1. 接着，针对要保护的资源即doSomething方法定义限流规则



针对doSomething，通过initFlowRules设置限流规则，其中参数含义如下：

·resource：设置需要保护的资源。这个资源的名称必须和Sph.entry种使用的

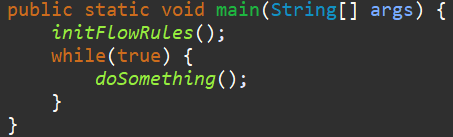
名称保持一致

·grade：限流阈值，QPS模式（1）或并发线程数模式（0）

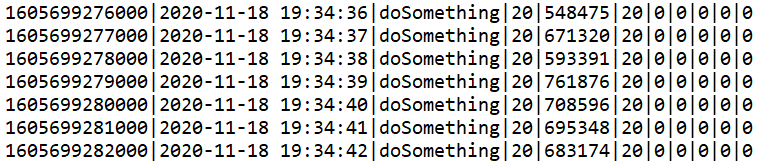
·count：限流阈值

上述代码的意思是，针对doSomething方法，每秒最多允许通过20个请求，也就是QPS为20。

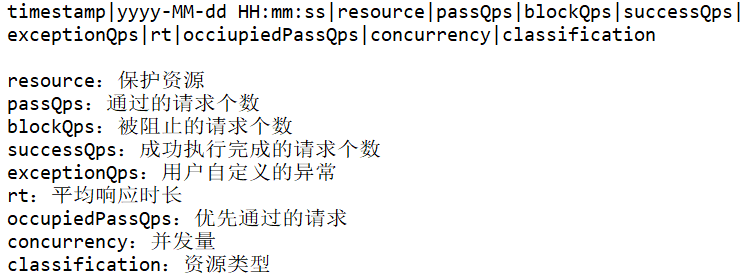
1. 最后，通过main方法进行测试



运行main方法之后，可以在${USER\_HOME}\logs\csp\${包名-类名}-metrics.log.date文件中看到如下日志：



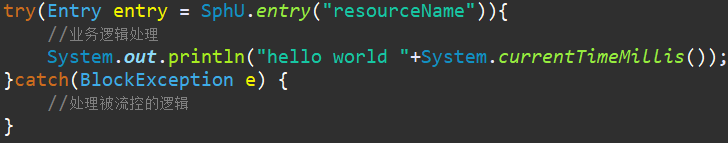
上述日志中对应字段的具体含义是：



从日志中可以看到，这个程序每秒稳定输出（doSomething）20次，和规则中预先设定的阈值时一样的，而被拒绝的请求每秒高达70多万次。

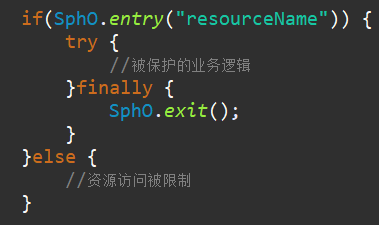
## 资源的定义方式

1. 在上一节中，我们通过抛出异常的方式来定义一个资源，也就是当资源被限流之后，会抛出一个BlockException异常，这时我们需要捕获该异常进行限流后的逻辑处理：



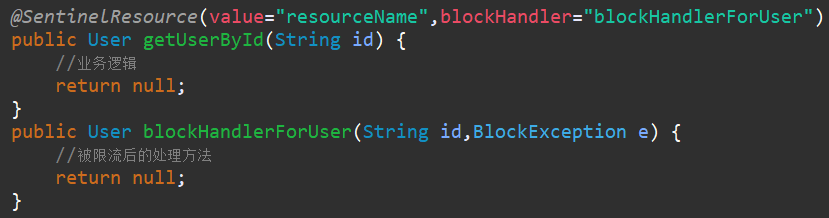
其中，resourceName可以定义方法名称、接口名称或者其他唯一标识。

1. 除此之外，还可以通过返回布尔值的方式来定义资源，代码如下：



在这种方式中，需要注意资源使用完之后要调用SphO.exit( )，否则会导致调用链记录异常，抛出ErrorEntryFreeException异常。

1. Sentinel还可以使用@SentinelResource注解的方式来定义资源，如下：



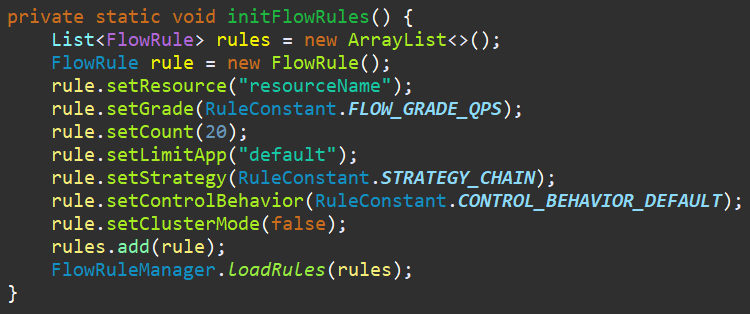
需要注意的是，blockHandler属性所配置的值blockHandlerForUser会在触发限流之后调用，这个方法的定义必须和原始方法getUserById的返回值、参数保持一致，而且需要增加BlockException参数。

Sentinel资源的定义还有更多的方式，感兴趣的读者可以去官网看看。

## Sentinel资源保护规则

### 限流规则

Sentinel支持多种保护规则：流量控制规则、熔断降级规则、系统保护规则、来源访问控制规则、热点参数规则。限流规则在前面的案例中简单使用国，先通过FlowRule来定义限流规则，然后通过FlowRuleManagement.loadRules来加载规则列表。完整的限流规则设置代码如下：



·resource：设置需要保护的资源。这个资源的名称必须和Sph.entry种使用的

名称保持一致

·grade：限流阈值，QPS模式（1）或并发线程数模式（0）

·count：限流阈值

·limitApp：是否需要针对调用来源进行限流，默认是default，即不区分调用来源

·strategy：调用关系限流策略-----直接、链路、关联

·constrolBehavior：流控行为，包括直接拒绝、排队等候、慢启动模式、默认是直

接拒绝。

·clusterMode：是否是集群限流，默认为否。

下面基于这几个参数的含义做一个详细分析。

### 基于并发数和QPS的流量控制

Sentinel流量控制统计有两种类型，通过grade属性来控制：

·并发线程数（FLOW\_GRADE\_THREAD）

·QPS（FLOW\_GRADE\_QPS）

**并发线程数**

并发线程数限流用来保护业务线程不被耗尽。比如，A服务调用B服务，而B服务因为某种原因导致服务不稳定或者响应延迟，那么对于A服务来说，它的吞吐量会下降，也意味着占用更多的线程（线程阻塞之后一直未释放），极端情况下会造成线程池耗尽。针对这个问题，一个常见的解决方案是通过不同业务逻辑使用不同的线程池来隔离业务自身的资源争夺问题，但是这个方案同样会造成线程数量过多带来的上下文切换问题。Sentinel并发线程数限流就是统计当前请求的上下文线程数量，如果超出阈值，新的请求就会被拒绝。

**QPS**

QPS（Queries Per Second）表示每秒的查询数，也就是一台服务器每秒能够响应的查询次数。当QPS达到限流的阈值时，就会触发限流策略。

### QPS流量控制行为

当QPS超过阈值时，就会触发流量控制行为，这种行为是通过controlBehavior来设置的，它包含：

·直接拒绝（RuleConstant.CONTROL\_BEHAVIOR\_DEFAULT）；

·Warm Up（RuleConstant.CONTROL\_BEHAVIOR\_WARM\_UP），冷启动（预热）；

·匀速排队（RuleConstant.CONTROL\_BEHAVIOR\_RATE\_LIMITER）；

·冷启动+匀速排队（RuleConstant.CONTROL\_BEHAVIOR\_WARM\_UP\_RATE\_LIMI

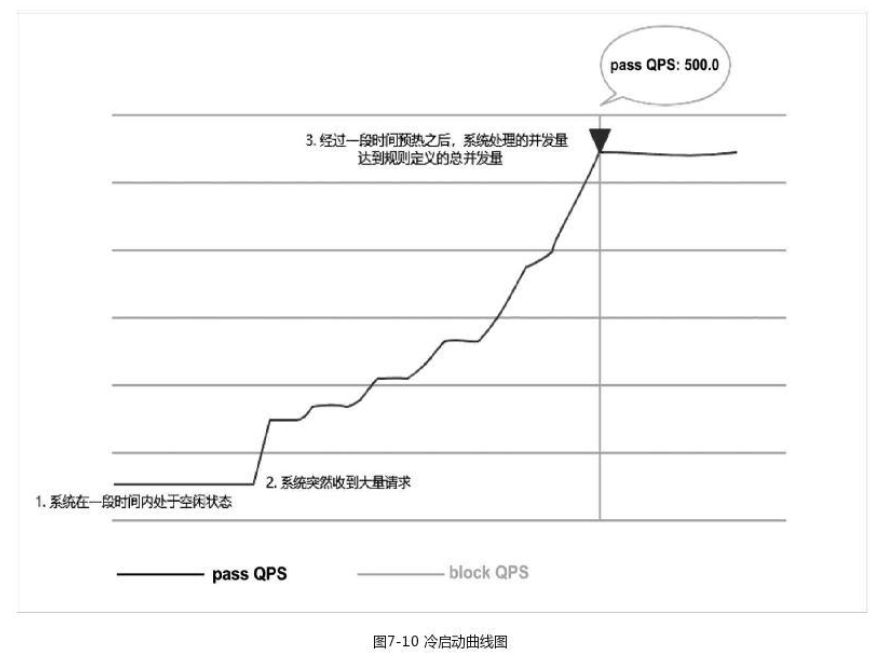
TER）

**直接拒绝**

直接拒绝是默认的流量控制方式，也就是当请求超出阈值时，直接抛出一个FlowException。

**Warm Up**

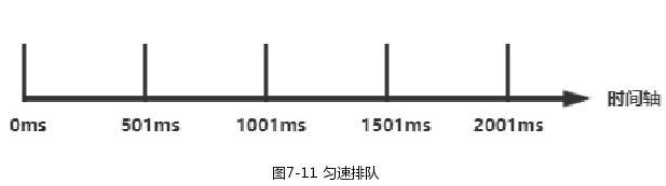
Warm Up是一种冷启动（预热）方式。当流量突然增大时，也就意味着系统从空闲状态突然切换到繁忙状态，有可能会瞬间把系统压垮。当我们希望请求处理的数量逐步递增，并在一个预期时间之后达到允许处理请求的最大值时，Warm Up就可以达到这个目的。如下图所示：



当前系统所能够处理的最大并发数是480，首先，在最下面标记的位置，系统一直处于空闲状态，接着请求量突然直线升高。这个时候系统并不是直接将QPS拉到最大值，而是在一定时间内逐步增加阈值，而中间这段时间就是一个系统逐步预热的过程。

**匀速排队**

匀速排队的方式会严格控制请求通过的间隔时间，也就是让请求以均匀的速度通过，其实相当于前面讲的漏桶限流算法。如下图所示：



当QPS=2时，意味着每个500ms才允许通过下一个请求。这种方式的好处是可以处理间隔性突发流量。

### 调用关系流量策略

调用关系包括调用方和被调用方法，一个方法又可能会调用其他方法，形成一个调用链。所谓的调用关系流量策略，就是根据不同的调用维度来触发流量控制。

·根据调用方限流

·根据调用链路入口限流

·具有关系的资源流量控制（关联流量控制）

1、**调用方限流**

所谓调用方限流，就是根据请求来源进行流量控制，我们需要设置limitApp属性来设置源信息，它有三个选项。

·default：表示不区分调用者，也就是任何访问调用者的请求都会进行限流统

计

·{some\_origin\_name}：设置特定的调用者，只有来自这个调用者的请求才会

进行流量统计和控制。

·other：表示针对除{some\_origin\_name}外的其他调用者进行流量控制。

由于同一个资源可以配置多条规则，如果多个规则设置的limitApp不一样，那么规则的生效顺序为：{some\_origin\_name} --> other --> default。

2、**根据调用链路入口限流**

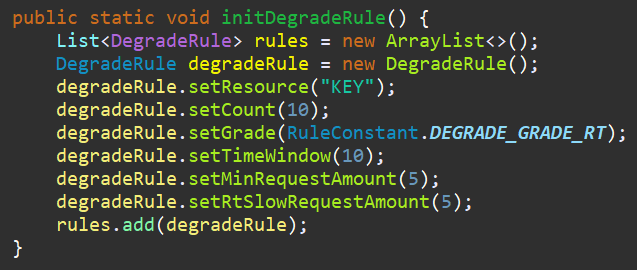
一个被限流保护的方法，可能来自不同的调用链路。比如针对资源nodeA，入口Entrance1和入口而Entrance2都调用了资源nodeA，那么Sentinel允许只根据某个入口来进行流量统计。比如我们针对nodeA资源，设置针对Entrance1入口的调用才会统计请求次数。它在一定程度上有点类似于调用方限流。

3、**关联流量控制**

当两个资源之间存在依赖关系或者资源争夺时，我们就说这两个资源存在关联。这两个存在依赖关系的资源在执行时可能会因为某个资源执行操作过于频繁而影响另一个资源的执行效率，所以关联流量控制（流控）就是限制其中一个资源的执行流量。

## Sentinel实现服务熔断

Sentinel实现服务熔断操作的配置和限流类似，不同之处在于限流采用的是FlowRule，而熔断中采用的是DegradeRule，配置代码如下：



其中，有几个属性说明如下：

·grade：熔断策略，支持秒级RT、秒级异常比例、分钟级异常数。默认是秒级RT

·timeWindow：熔断降级的时间窗口，单位为s。也就是触发熔断降级之后多长时

间内自动熔断

·rtSlowRequestAmount：在RT模式下，1s内持续多少个请求的平均RT超出阈值

后触发熔断，默认值是5

·minRequestAmount：触发的异常熔断最小数，请求数小于该值时即使异常比例

超出阈值也不会触发熔断，默认值是5

Sentinel提供了三种熔断策略，对于不同的策略，参数的含义也不同：

·平均响应时间（RuleConstant.DEGRADE\_GRADE\_RT）：如果1s内持续进入5个请

求，对应的平均响应时间都超过了阈值（count，单位为ms），那么在接下来

的时间窗口（timeWindow，单位为s）内，对这个方法的调用都会自动熔断，抛

出DegradeException。

·异常比例：（RuleConstant.DEGRADE\_GRADE\_EXCEPTION\_RATIO）：如果每秒资源数

≥minRequestAcmount（默认值为5），并且每秒的异常总数占总通过量的比例超

过阈值count（count的取值范围时[0.0，1.0]，代表0%~100%），则资源将进入降

级状态。同样，在接下来的timeWindow之内，对这个方法的调用都会自动触发

熔断。

·异常数（RuleConstant.DEGRADE\_GRADE\_EXCEPTION\_COUNT）：当资源最近一分钟

的异常数目超过阈值之后，会触发熔断。需要注意的是，如果timeWindow小于

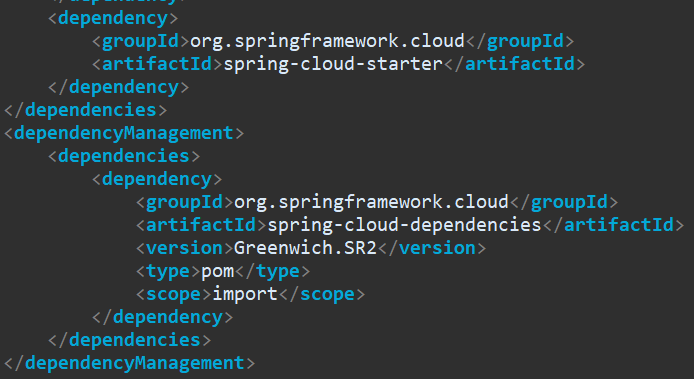
60s，则结束熔断状态后仍然可能再进入熔断的状态。

# Spring Cloud集成Sentinel实践

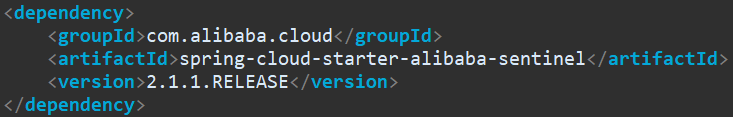
Spring Cloud Alibaba默认为Sentinel整合了Servlet、RestTemplate、FeignClient和Spring WebFlux。它不仅补全了Hystrix在Servlet和RestTemplate这一块的空白，而且还完全兼容了Hystrix在FeignClient中限流降级的用法，并支持灵活配置和调整流控规则。

## Sentinel接入Spring Cloud

1. 创建一个名为spring-cloud-sentinel-samplp的Spring Boot项目，并集成Geenwich.SR2版本的Spring Cloud依赖。



1. 添加Sentinel依赖



1. 创建一个REST接口，通过@SentinelResource注解配置限流保护资源



上述代码中，配置限流资源有几种情况：

·Sentinel start在默认情况下会为所有的HTTP服务提供限流埋点，所以如果

只想对HTTP服务进行限流，那么只需要添加依赖即可，不需要修改任何代

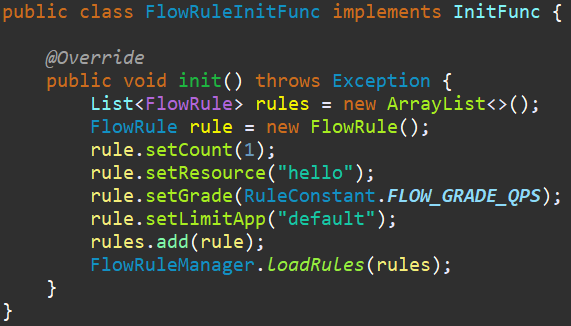
码。

·如果想要对特定的方法进行限流或降级，则需要通过@SentinelResource注

解来实现限流资源的定义。

·可以通过SphU.entry( )方法来配置资源。

1. 手动配置流控规则可以借助Sentinel的InitFunc SPI扩展接口来实现，只需要实现InitFunc接口，并在覆写的init方法中编写规则加载的逻辑即可。（除了手动配置流控规则，还可以通过Sentinel Dashboard来进行配置），



SPI是扩展点，如果需要被Sentinel加载，那么还要在resources目录下创建META-

INF/services/com.alibaba.csp.sentinel.init.InitFunc文件，文件内容就是自定义扩展点的全路径。



上述配置好之后，在初次访问任意资源的时候，Sentinel就会自动加载hello资源的流控规则。

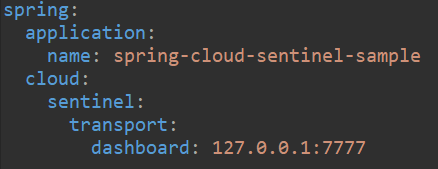
1. 启动服务，访问http://localhost:8080/say方法，当访问频率超过设定阈值时，就会触发限流。

上述配置过程中是基于手动配置来加载流控规则的，还有一种方式就是通过Sentinel Dashboard来进行配置。

## 基于Sentinel Dashboard实现流控配置

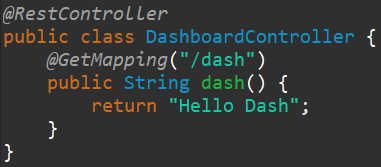
在上一节案例中，是通过代码手动配置流控规则的，除了手动配置流控规则，还可以通过Sentinel Dashboard配置流控规则，并且可以实现流控规则的动态配置。

1. 启动Sentinel Dashboard
2. 在application.yml中增加如下配置



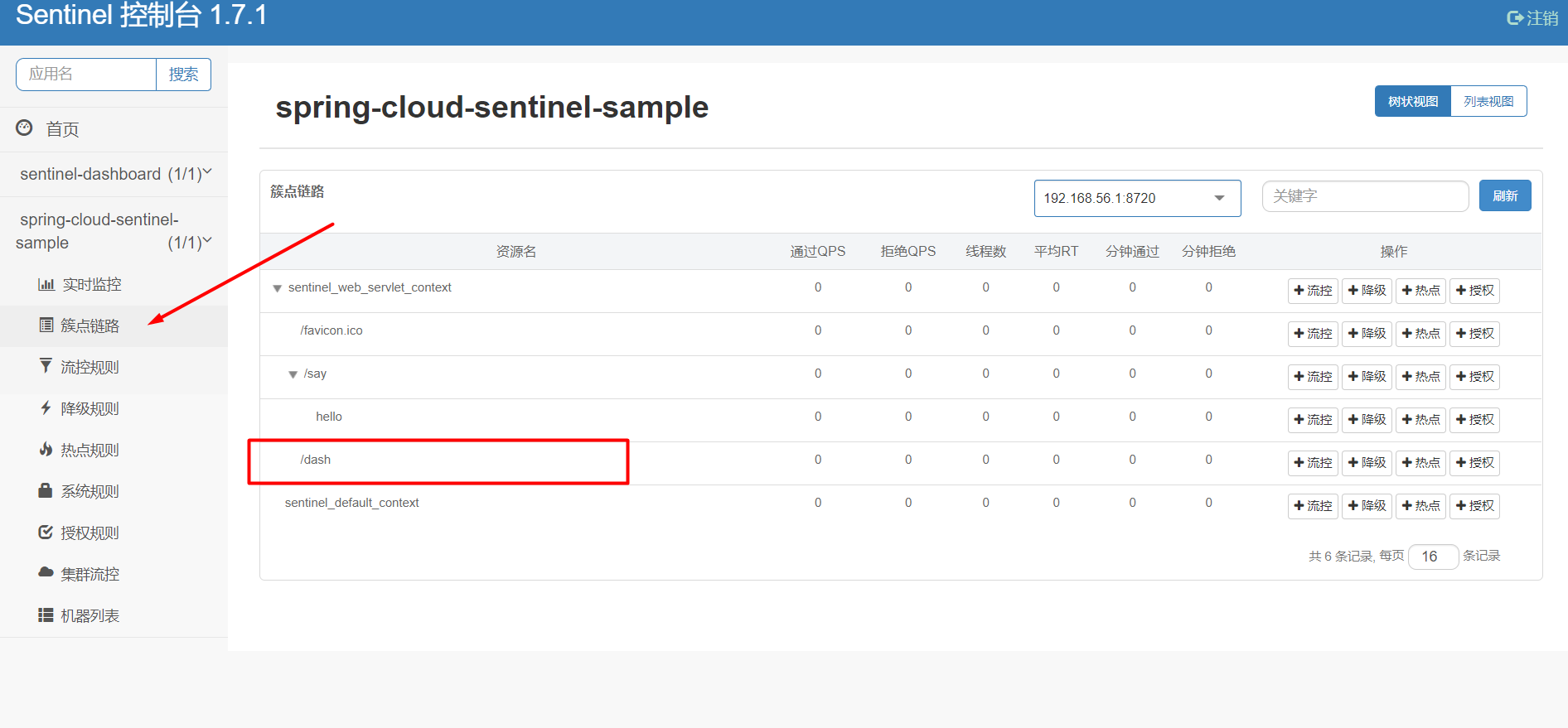
spring.cloud.sentinel.transport.dashboard指定的是Sentinel Dashboard的服务器地址。

1. 提供一个REST接口



此处不需要添加任何埋点，在默认情况下Sentinel Starter会对所有HTTP请求进行限流。

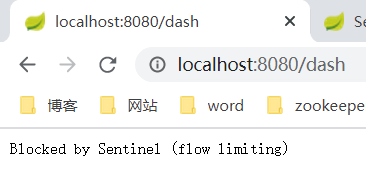
1. 启动服务访问http://localhost:8080/dash，由于没有配置流控规则，所以不存在限流行为。
2. 进入Sentinel Dashboard，进入spring.application.name对应的菜单，找到【簇点链路】，在列表中找到/dash这个REST接口的资源名称。



针对/dash这个资源名称，可以在操作栏中单击【流控】按钮设置流控规则。如下：



新增流控规则中的所有配置信息，实际就是FlowRule中对应的属性配置，为了演示效果，把单击阈值设置为1。新增完成之后，再次访问，当QPS超过1时，就可以看到限流效果。如下：

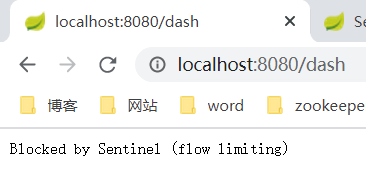


## 自定义URL限流异常

在默认情况下，URL触发限流后会直接返回如下信息：

Blocked by Sentinel ( flow limiting )

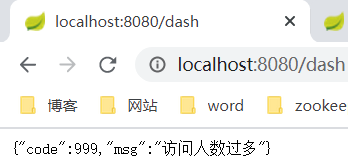
比如：



在实际应用中，大都采用JSON格式的数据，所以如果希望修改触发限流之后的返回结果形式，则可以通过自定义限流异常来处理，实现UrlBlockHandler接口并重写blocked方法：



效果如下：

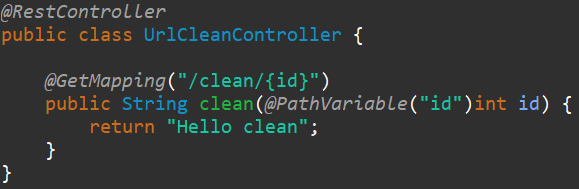


在覆写blocked方法中，包含了request对象和response对象，拥有了这两个对象，还可以执行服务端或客户端跳转。还有一种场景是，当触发限流之后，我们希望直接跳转到一个降级页面，可以通过以下配置来实现：



## URL资源清洗

Sentinel中HTTP服务的限流默认由Sentinel-Web-Servlet包中的CommonFiler来实现实现，这个Filter会把每个不同的URL都作为不同的资源来处理。但如果定义了包含路径变量的控制器接口，如下：



它提供一个携带{id}参数的REST风格API，对于每一个不同的{id}，URL也都不一样，所以在默认情况下Sentinel会把所有的URL当作资源来进行流控，这会导致两个问题：

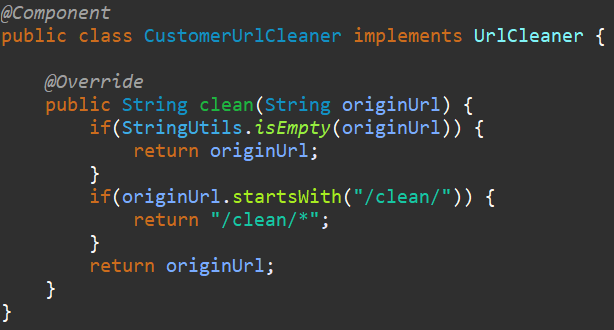
·限流统计不准确，实际需求时控制clean方法总的QPS，结果统计的是每个URL

的QPS

·导致Sentinel中资源数量过多，默认资源数量的阈值时6000，对于多出的资源

规则将不会生效

针对这两个问题可以通过UrlClean接口来实现资源清洗，也就是对于/clean/{id}这个URL，我们可以统一归集到/clean/\*资源下，具体配置代码如下，实现UrlCleaner接口，并重写clean方法即可。



# Sentinel集成Nacos实现动态流控规则

## 为什么要集成Nacos实现动态流控

通过前面对Sentinel的学习可以发现，Sentinel的理念是只需要开发者关注资源的定义，它默认会对资源进行流控。当然，我们还需要对定义的资源设置流控规则，前面演示了两种设置流控规则的方式：

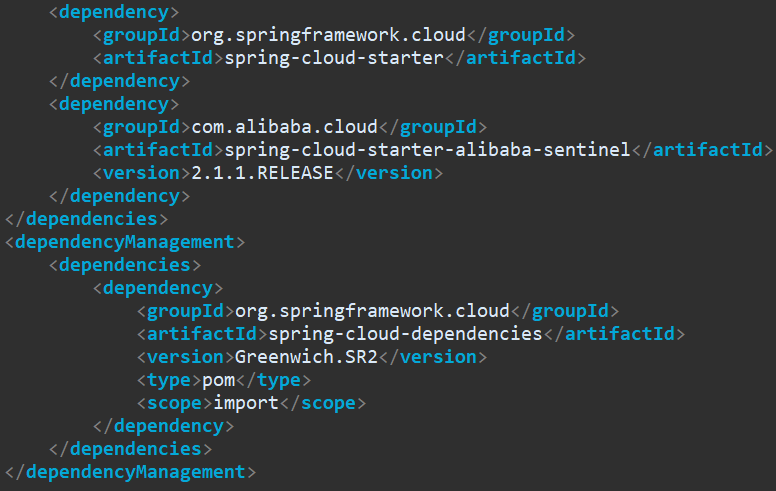
·通过FlowRuleManager.loadRules( List rules )手动配置流控规则

·在Sentinel Dashboard上针对资源动态创建流控规则

针对第一种设置方式，如果接入Sentinel Dashboard，那么同样支持动态修改流控规则。但是，这里会存在一个问题，基于Sentinel Dashboard所配置的流控规则，都是保护在内存中的，一旦应用重启了，这些规则都被被清除。为了解决这个问题，Sentinel提供了动态数据源支持。目前，Sentinel支持Consul、ZooKeeper、Redis、Nacos、Apollo、etcd等数据源的扩展。接下来通过一个案例演示Spring Cloud Sentinel集成Nacos实现动态流控规则。

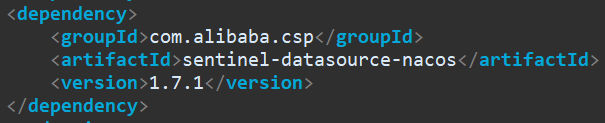
## 集成Nacos实现动态流控规则

1. 创建一个名为spring-cloud-sentinel-dynamic-rule的SpringBoot工程（此处版本为2.1.1.RELEASE），导入依赖

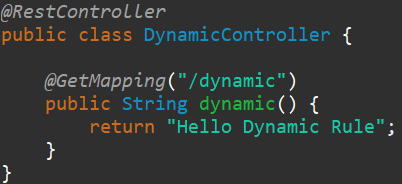


这里集成了Greenwich.SR2版本的Spring Cloud依赖以及添加了sentinel的依赖包。

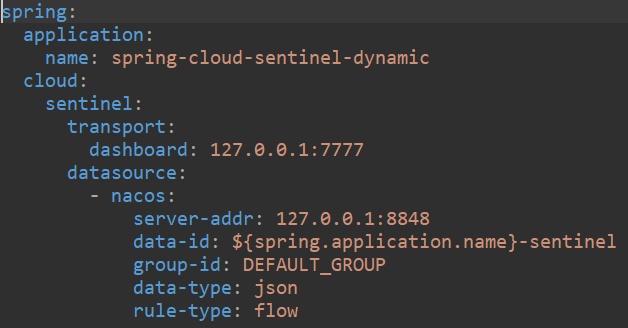
1. 导入sentinel集成nacos所需的数据源依赖



1. 创建一个REST接口，用于测试



1. 在application.yml中添加数据源配置



部分说明配置如下：

·datasource：目前支持redis、apollo、zk、file、nacos，选择什么类型的数据

源就配置相应的key即可

·data-id：Nacos中DataID的名称，可以设置成包含${spring.application.name}

的值，方便区分不同应用的配置

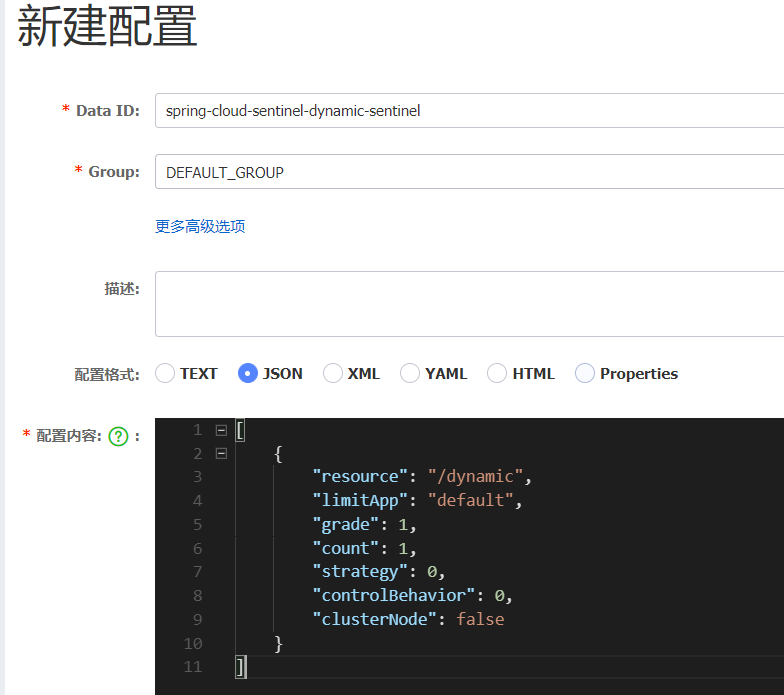
·group-id：DataID所属的组

·data-type：配置项的内容格式，Spring Cloud Alibaba Sentinel提供了JSON和

XML两种格式。如果需要自定义，则可以将值配置为custom，

并配置converter-class指向converter类。

1. 启动Sentinel控制台和Nacos控制台后，运行该工程。然后登入控制台，创建流控配置规则，配置信息如下：



该配置项的DataID和Group以及配置格式需要与application.yml中配置的一致。而配置内容其实就是FlowRule的属性，包含了要保护的资源、阈值等。

1. 登录Sentinel Dashboard，找到对应项目名称菜单下【流控规则】，就可以看到在Nacos上所配置的流控规则已经被加载了。如下：



当我们在Nacos的控制台上修改流控规则后，可以同步在Sentinel Dashboard上看到流控规则的变化了。

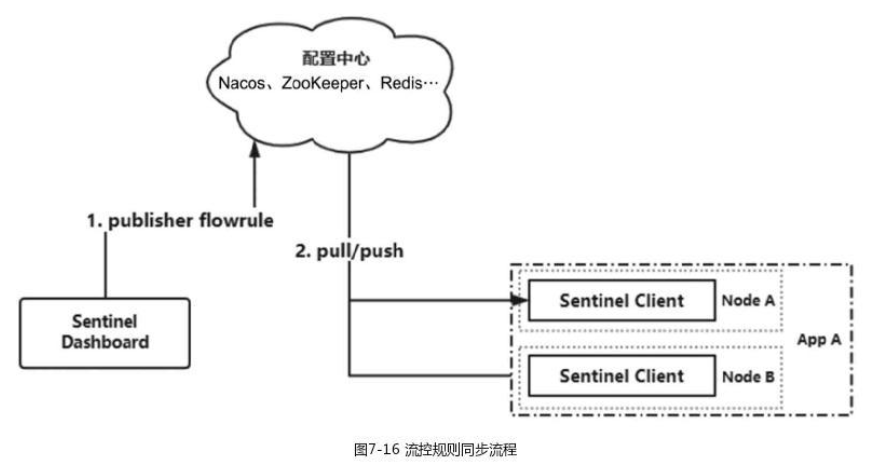
## 存在的问题

Sentinel通过集成Nacos以数据源的方式实现动态流控规则后，接口流控规则的动态修改就存在于以下两个地方：

·Sentinel Dashboard

·Nacos控制台

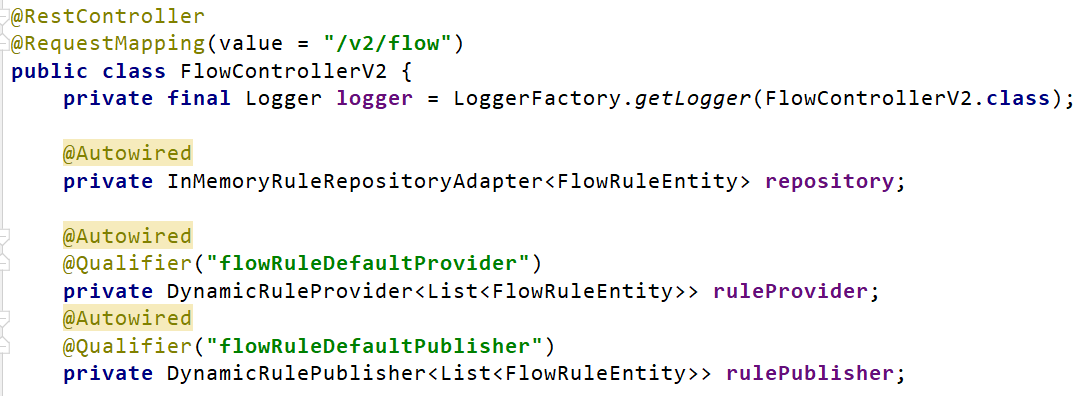
那么问题来了，将Nacos作为一个流控规则的持久化平台，在Nacos控制台中修改流控规则，同步到Sentinel Dashboard，但是正常的操作过程应该是开发者在Sentinel Dashboard上修改流控规则后同步到Nacos，可惜遗憾的是，目前Sentinel Dashboard并不支持该功能。所以，Nacos名义上是“Datasource”，但实际上充当的仍然是配置中心的角色，开发者可以在Nacos控制台动态修改流控规则并实现规则同步。在实际开发中，很难避免在不清楚情况的前提下，部分开发者通过Sentinel Dashboard来管理流控规则，部分开发者通过Nacos来管理流控规则，这将会导致非常严重的问题。如下图所示：



Nacos是在此处扮演的角色应该是一个“Datasource”，所以笔者强烈建议大家不要在Nacos上修改流控规则，因为这种修改危险系数很好，毕竟Nacos的UI并不是专门负责流控规则维护的。这也就意味着流控规则的管理应该集中在Sentinel Dashboard上，我们需要实现Sentinel Dashboard来动态维护流控规则并同步到Nacos上，虽然目前官方还没有提供支持，但是我们大家可以自己来实现。

## 实现规则同步

针对Sentinel集成Nacos实现动态流控规则后在Nacos控制台修改流控规则可以同步到Sentinel Dashboard，但Sentinel Dashboard修改流控规则后不能同步到Nacos控制台的这个问题，由于官方没有提供解决方案，我们可以自行进行解决。在Sentinel Dashboard的“流控规则”下的所有操作，都会调用Sentinel-Dashboard源码中的FlowControllerV1类，这个类中包含流控规则的本地化CRUD操作。另外在com.alibaba.csp.sentinel.dashboard.controller.v2包下存在一个FlowControllerV2类，这个类同样提供流控规则的CRUD，和V1版本不同的是，它可以实现指定数据源的规则拉取和发布。部分代码如下：



FlowControllerV2依赖以下两个非常重要的类：

·DynamicRuleProvider：动态规则的拉取，从指定数据源中获取流控规则后在

Sentinel Dashboard中展示

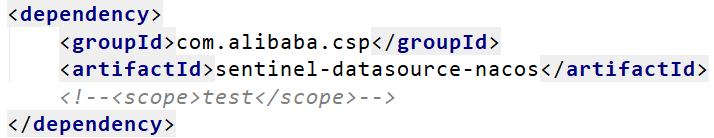
·DynamicRulePublisher：动态规则的发布，将在Sentinel Dashboard中修改的

规则同步到指定数据源中

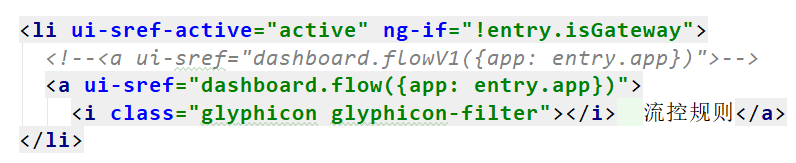
我们可以扩展这两个类，然后集成Nacos来实现Sentinel Dashboard规则的同步

通过修改Sentinel Dashboard的源码实现，具体步骤如下：

1. 在GitHub中下载Sentinel Dashboard的源码，下载完成后使用IDEA工具打开${Sentinel\_home}/sentinel-dashboard工程。
2. 在pom.xml中把sentinel-datasource-nacos依赖的<scope>注释掉。

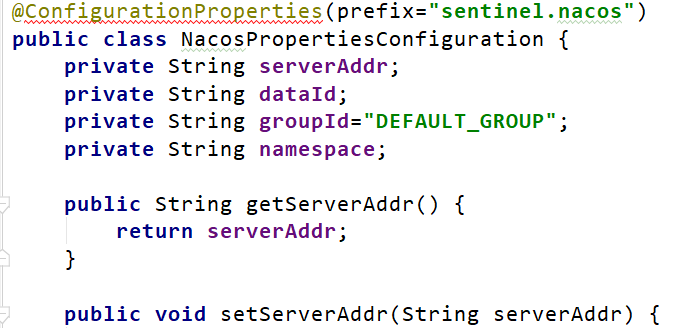


1. 修改resources/app/scripts/directives/sidebar/sidebar.html文件中的下面这段代码，将dashboard.flowV1改为dashboard.flow，就是去掉V1。



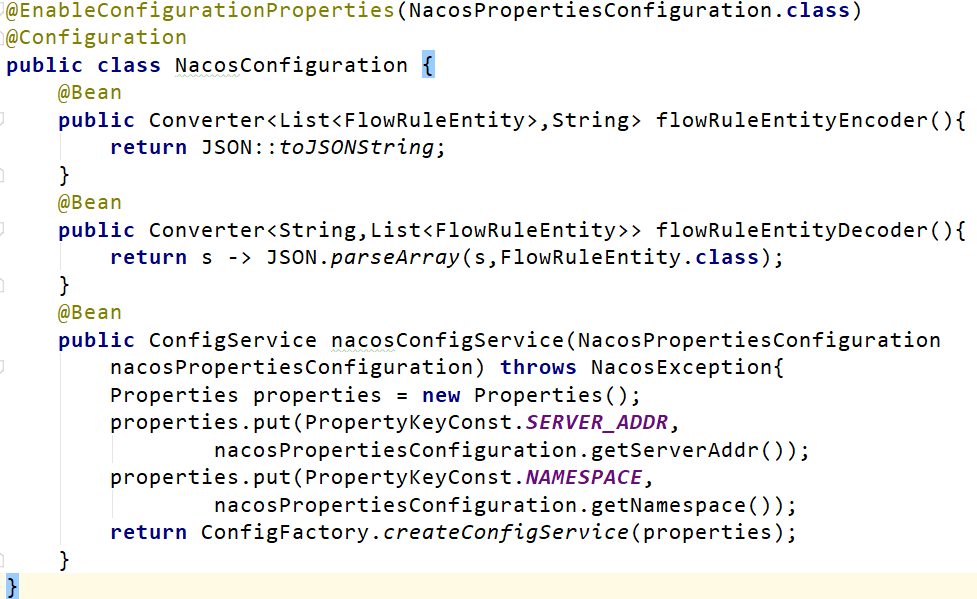
修改之后，会调用FlowControllerV2中的接口。

1. 在com.alibaba.csp.sentinel.dashboard.rule包中创建一个nacos包，并创建一个类用来加载外部化配置。



分别对应着Nacos的服务器地址，配置项的dataID、groupId、namespance

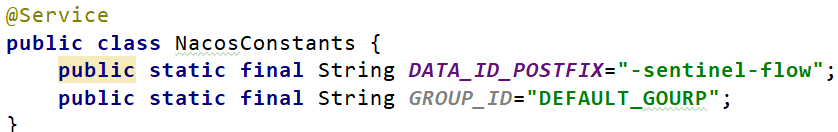
1. 创建一个Nacos配置类NacosConfiguration



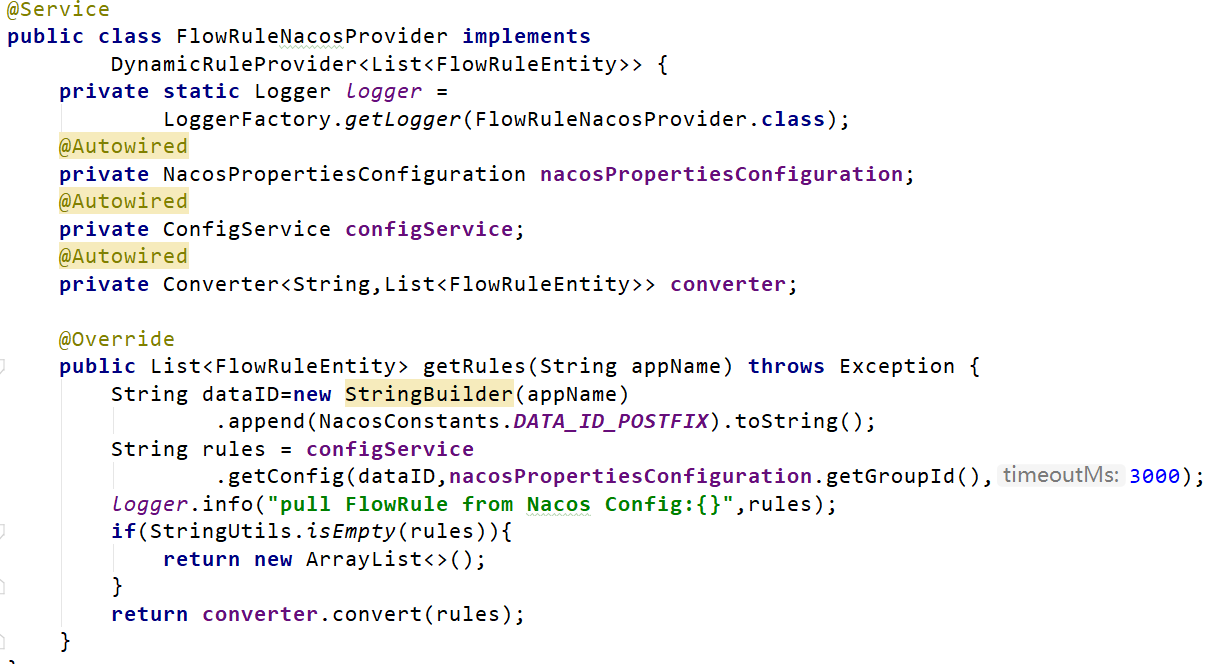
注入Converter转换器，将FlowRuleEntity转化成FlowRule，实现反向转换。

注入Nacos配置服务ConfigService。

1. 创建一个常量类NacosConstants，分别表示默认的GROUP\_ID和DATA\_ID的后缀



1. 实现动态从Nacos配置中心获取流控规则

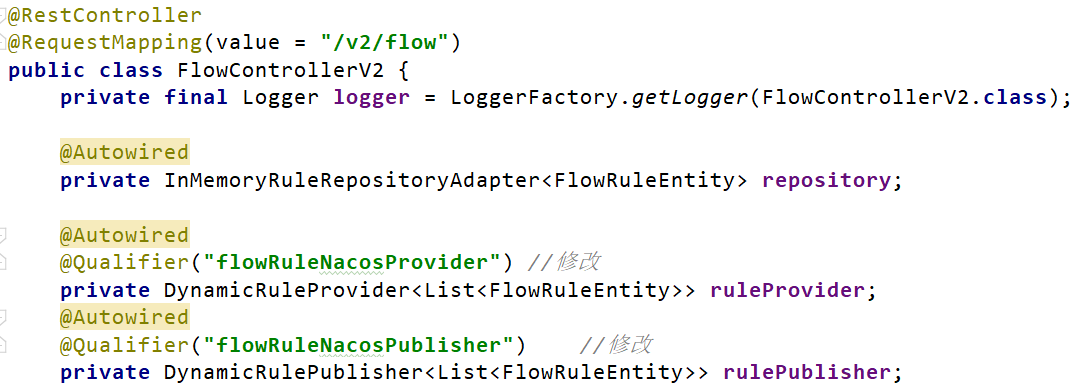


在第5章中讲过Nacos配置中心，所以这段代码不能理解。主要是通过ConfigService.getConfig方法从Nacos Config Server中读取指定配置信息，并通过converter转化为FlowRule配置。

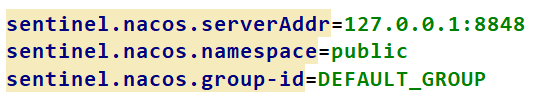
1. 创建一个流控规则发布类，在Sentinel Dashboard上修改完配置之后，需要调用该发布方法将数据持久化到Nacos中。



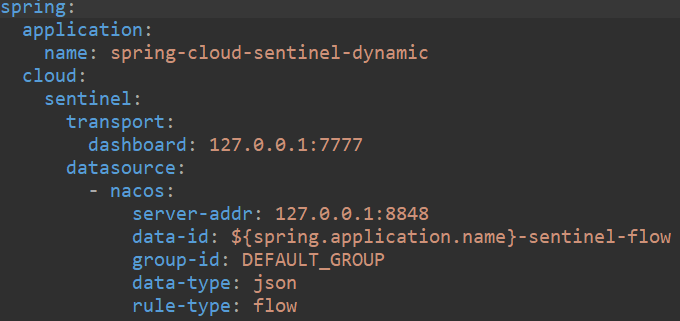
1. 修改FlowControllerV2类，将上面配置的两个类注入进来，表示规则的拉取和规则和发布统一使用我们前面自定义的两个示例。



1. 在application.properties文件中添加Nacos服务器端的配置信息



1. 使用mvn clean package打成一个jar包，然后按照前面介绍的方法启动该jar包即可。
2. 对于应用程序来说，需要改动的地方比较少，只要注意配置文件中的data-id的命名要以-sentinel-flow结尾即可，因为在Sentinel Dashboard中我们写了一个固定的后缀。

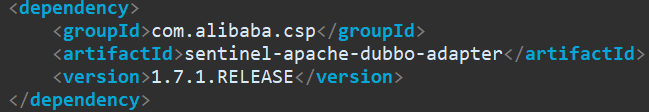


后续的测试过程就比较简单了。

# Dubbo集成Sentinel实现限流

## 介绍

1. Sentinel提供了与Dubbo整合的模块Sentinel Apache Dubbo Adapter，可以对服务提供方和服务消费者进行流控，在使用的时候，只需要添加以下依赖：



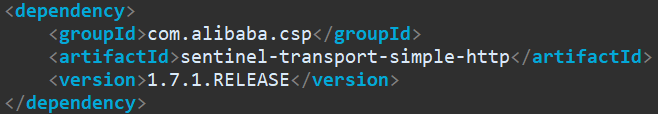
添加依赖后，Dubbo服务中的接口和方法（包括服务端和消费端）就会成为Sentinel中的资源，只需对指定的资源配置流控规则就可以实现Sentinel流控功能。

1. Sentinel Apache Dubbo Adapter实现限流的核心原理是基于Dubbo的SPI机制实现Filter扩展，Dubbo的Filter机制是专门为服务提供方和服务消费方调用过程进行拦截设计的，每次执行远程方法，该拦截都会被执行。同时，Sentinel Apache Dubbo Adapter还可以自定义开启或者关闭某个过滤器（Filter）的功能，下面这段代码表示关闭消费端的过滤器。

## Dubbo服务接入Sentinel Dashboard

Spring-cloud-starter-alibaba-sentinel目前无法支持Dubbo服务的限流，所以针对Dubbo服务的限流只能使用sentinel-apache-dubbo-adapter，但这个适配组件并没有自动接入Sentinel Dashboard，需要通过以下步骤来进行接入：

1. 引入sentinel-transport-simple-http依赖，这个依赖可以上报应用相关信息到控制台：



1. 启动Sentinel Dashboard，将Dubbo服务打包成jar包，启动该jar时添加启动参数，如下：

-Djava.net.preferIPv4Stack=true -Dcsp.sentinel.api.port=8720

-Dcsp.sentinel.dashboard.server=127.0.0.1:7777

-Dproject.name=spring-cloud.sentinel-dubbo.provider

参数说明如下：

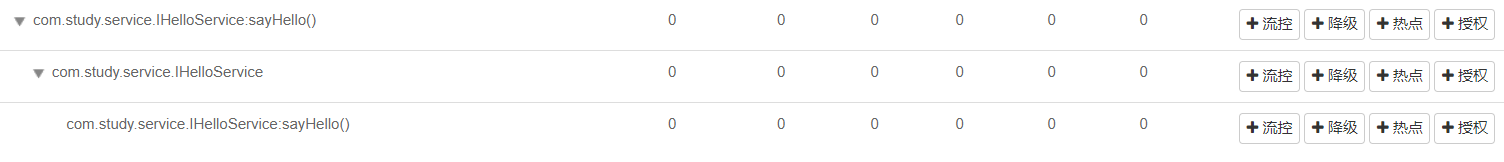
·-Djava.net.preferIPv4Stack：表示只支持IPv4

·-Dcsp.sentinel.api.port：客户端的port，用于上报应用的信息

·-Dcsp.sentinel.dashboard.server：Sentinel Dashboard地址

·-Dproject.name：该应用在Sentinel Dashboard左栏中显示的名称

1. 启动服务消费者，调用一次服务提供者的接口。因为只有这样才能在Sentinel Dashboar上看到服务提供的相关信息。
2. 登录Sentinel Dashboard之后，进入【簇点链路】，就可以看到如下所示的资源信息：



需要注意的是，限流可以通过服务接口或者服务方法设置

·服务接口：resourceName为接口的全限定名（包类路径），如上图中的

com.study.IHelloService

·服务方法：resourceName为接口全限定名:方法名，如上图中的

com.study.IHelloService.sayHello( )

## Dubbo服务限流规则配置与持久化

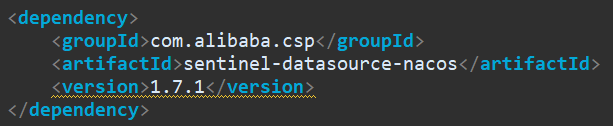
Dubbo限流同样可以通过以下集中方式来实现。

·Sentinel Dashboard

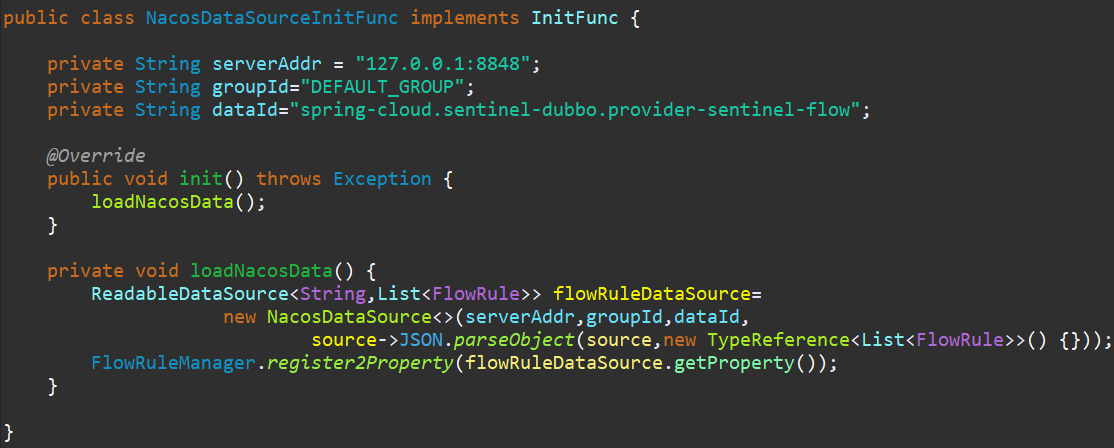
·FlowRuleManager.loadRules( List rules )

在前面讲过基于Sentinel Dashboard来实现流控规则配置，集成Nacos后可以将配置规则持久化到Nacos中。这种规则的持久化机制在Spring Cloud Sentinel中是自动实现的，在Sentinel Apache Dubbo Adapter组件中并没有实现该功能。下面演示一下在Dubbo服务中如何实现规则的持久化。

1. 添加sentinel-datasource-nacos的依赖



1. 通过Sentinel提供的InitFunc扩展点，实现Nacos数据源的配置



1. NacosDataSourceInitFunc要实现加载，需要在resources目录下的META-INF/services中创建一个名称为com.alibaba.csp.sentinel.init.InitFunc的文件，文件内容为NacosDataSourceInitFunc的全路径。



1. 访问Sentinel Dashboard，在针对某个资源创建流控规则时，这个规则会同步到Nacos配置中心（前提是对Sentinel Dashboard修改了实现双向同步，否则只能从Nacos同步到Sentinel Dashboard，详见Sentinel集成Nacos小节）。而当Nacos配置中心发生变化时，会触发事件机制通知Dubbo应用重新加载流控规则。

# Sentinel热点限流

## 认识热点限流

1. **认识热点限流**

热点数据表示经常访问的数据，在有些场景中我们希望针对这些访问频次非常高的数据进行限流，就比如针对一段时间内频繁访问的用户地址进行限流，或者针对频繁访问的某个用户ID进行限流。为此Sentinel提供了热点参数限流的策略，它是一种特殊的限流，在普通限流的基础上对同一个受保护的资源区根据请求中的参数分别处理，该策略只对包含热点参数的资源调用生效。

1. **使用场景**

热点限流在以下场景中使用较多：

·服务网关层：例如防止网络爬虫和恶意攻击，一种常用方法就是限制爬虫的

IP地址，客户端IP地址就是一种热点参数。

·写数据的服务：例如业务系统提供写数据的服务，数据会写入数据库之类的

存储系统，存储系统的底层会加锁写磁盘上的文件，部分存储系统会将某一

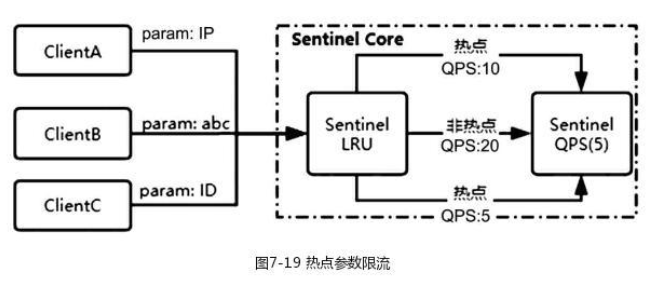
类数据写入同一个文件。如果底层写同一文件，会出现抢占锁的情况，导致

出现大量超时和失败。出现这种情况时一般有两种解决办法：修改存储设计、

对热点参数限流。

1. **热点参数统计策略**

Sentinel通过LRU策略结合滑动窗口机制来实现热点参数的统计，其中，LRU策略可以统计单位时间内最常访问的热点数据，滑动窗口机制可以协助统计每个参数的QPS。如下图所示：

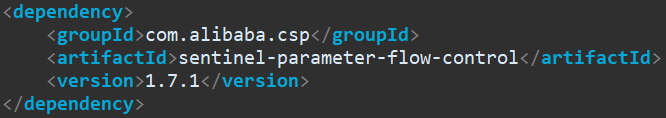


Sentinel会根据请求的参数来判断哪些是热点参数，然后通过热点参数限流规则，

将QPS超过设定阈值的请求进行阻塞。

## 使用热点参数限流

1. 引入热点参数限流包依赖sentinel-parameter-flow-control

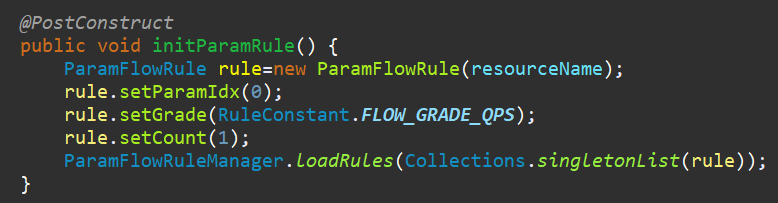


1. 创建一个接口，并定义限流埋点，此处针对参数id配置热点限流规则



针对热点参数，需要通过SphU.entry (resourceName , EntryType.IN , 1, id) 方法设置，其中该方法的最后一个参数是一个可变参数，有多个热点参数时就按照次序依次传入。以上配置表示将参数id作为热点参数进行限流。

1. 针对上述sayHello方法这个资源设置热点参数限流规则，通过ParamFlowRuleManager.loadRules方法加载热点参数规则。



@PostConstruct该注解被用来修饰一个非静态的void（）方法。被@PostConstruct修饰的方法会在服务器加载Servlet的时候运行，并且只会被服务器执行一次。PostConstruct在构造函数之后执行，init（）方法之前执行。

1. 通过测试工具或快速刷新浏览器来测试热点限流。测试完成后，访问Sentinel Dashboard，进入【实时监控】查看限流效果。



也可以在【热点规则】中查看到配置的热点参数限流规则。

## @SentinelResource热点参数限流

如果是通过@SentinelResource注解来定义资源的，当注解所配置的方法上有参数时，Sentinel会把这些参数传入Sphu.entry(res , args)。比如下面这段代码：



默认情况下，当用户访问这个接口时就会触发热点限流规则的验证（关于@SentinelResource注解的用法前面已经介绍过）。

## 热点参数规则说明

热点参数规则是通过ParamFlowRule来配置的，它的大部分属性和FlowRule类似，下面针对ParamFluwRule特定的属性进行简单说明：

·durationInSec：统计窗口时间的长度，单位为秒

·maxQueueingTimeMS：最长排队等待时长，只有当流行为controlBehavior设置为

均速排队模式时生效

·paramIdx：热点参数的索引，属于必填项，它对应的是SphU.entry(xxx，args)中

参数索引位置

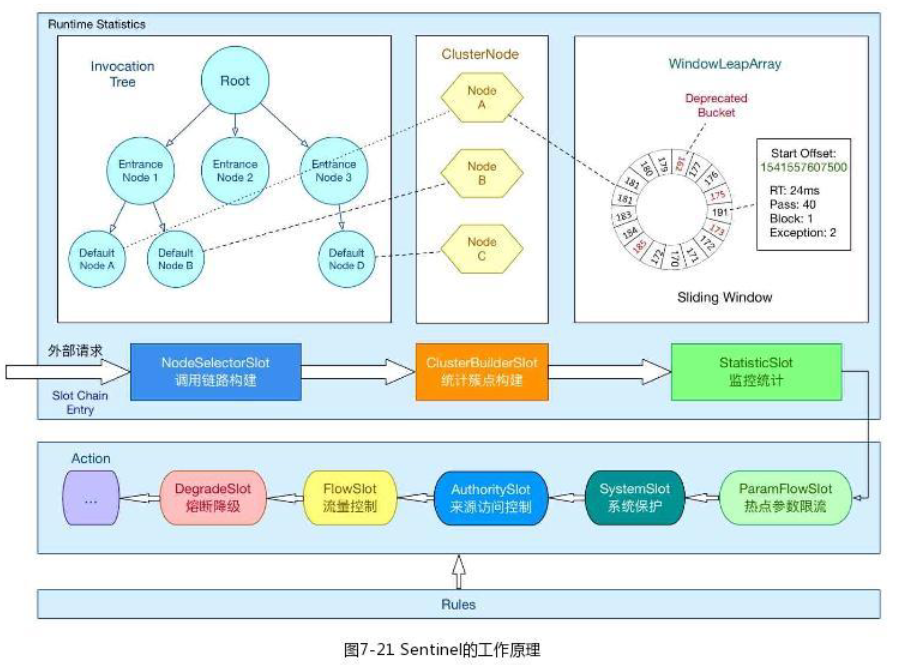
·paramFlowItemList：针对指定参数值单独设置限流阈值，不受count阈值的影响

对于grade、count等属性，在学习FlowRule时已经介绍过。如果想要知道ParamFlowRule有多少个属性，请查看源码。

# Sentinel的原理分析

## Sentinel工作原理

Sentinel的核心分为三部分：工作流程、数据结构和限流算法，工作原理（整体架构）如下图所示：



可以看出，调用链路是Sentinel的工作主流程，有各个Slot插槽组成，将不同的Slot按照顺序串在一起（职责链模式），从而将不同的功能（限流、降级、系统保护）组合在一起。Sentinel中各个Slot承担了不同的职责，例如LogSlot负责记录日志、StaticSlot负责统计指标数据、FlowSlot负责限流等。这是一种职责分离的设计，每个模块更聚焦于实现某个功能。

在Sentinel中，所有的资源都对应一个资源名称（resourceName），每次访问该资源都会创建一个Entry对象，在创建Entry的同时，会创建一系列功能槽（Slot Chain），这些槽组成一个职责链，每个槽负责不同的职责。

·NodeSelectorSlot：负责收集资源的调用路径，以树状结构存储调用栈，用于根

据调用路径来限流降级。

·ClusterBuilderSlost：负责创建以资源名维度统计的ClusterNode，以及创建每个

ClusterNode下按调用来源origin划分的StatisticNode。

·LogSlot：在出现限流、熔断、系统保护时负责记录日志。

·AuthoritySlot：权限控制，支持黑名单和白名单两种策略。

·SystemSlot：控制总的入口流量，限制条件依次是总QPS、总线程数、RT阈值、

操作系统当前load1、操作系统当前CPU利用率。

·FlowSlot：根据限流规则和各个Node中的统计数据进行限流判断。

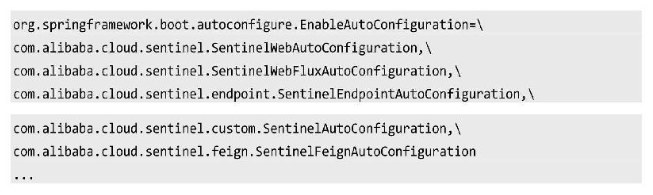
·DegradeSlot：根据熔断规则和各个Node中的统计数据进行服务降级。

·StatisticSlot：统计不同维度的请求数、通过数、限流数、线程数等runtime信息，

这些数据存储在DefaultNode、OriginNode和ClusterNode中。

## Spring Cloud Sentinel工作原理分析

在Spring Cloud中使用Sentinel实现限流的场景中，不需要任何配置，Sentinel会自动保护所有的HTTP服务。本节重点讲解该实现机制。在Spring-Cloud-Alibaba-Sentinel包中，我们知道Starter组件会用到自动装配，所以直接找到META-INF/spring.factories文件，如下：



这里EnableAutoConfiguration自动装配了5个配置类：

·SentinelWebAutoConfiguration是对WebServlet环境的支持

·SentinelWebFluxAutoConfiguration是对Spring WebFlux的支持

·SentinelEndpointAutoConfiguration暴露Endpoint信息

·SentinelFeignAutoConfiguration用于适配Feign组件

·SentinelAutoConfiguration支持对RestTemplate的服务调用使用Sentinel进行保护

更多分析请参考教材书。

## Sentinel核心源码模块

为了更好地理解Sentinel的实现原理，Sentinel源码的本本为1.7.1，可在GitHub上下载，源码模块如下：

·sentinel-adapter：负责针对主流开源框架进行限流适配，比如Dubbo、gRPC、Zuul

等

·sentinel-core：Sentinel核心库，提供限流、降级等实现

·sentinel-dashboard：Sentinel控制台模块，提供可视化监控和管理

·sentinel-demo：官方提供的案例

·sentinel-extension：实现不同组件的数据源扩展，比如Nacos、ZooKeeper、Appllo

等

·sentinel-transport：通信协议处理模块

限流及熔断的核心逻辑都是在sentinel-core中实现的，所以我们主要针对sentinel-core模块进行分析（更多源码分析请参考教材书）。