Sentinel分布式系统的流量组件

0 关于Java1234_锋哥

作者：java1234_小锋

关于锋哥：http://www.java1234.vip/article/14

java学习路线图：http://www.java1234.vip/article/1

QQ：554605804

锋哥微信：java3459

官网站点：http://www.java1234.vip

java1234公众号：

1 Sentinel快速了解

1.1 Sentinel介绍

Sentinel是阿里开源的项目，提供了流量控制、熔断降级、系统负载保护等多个维度来保障服务之间的稳定性。
官网：https://github.com/alibaba/Sentinel/wiki

1.2 Sentinel 的历史

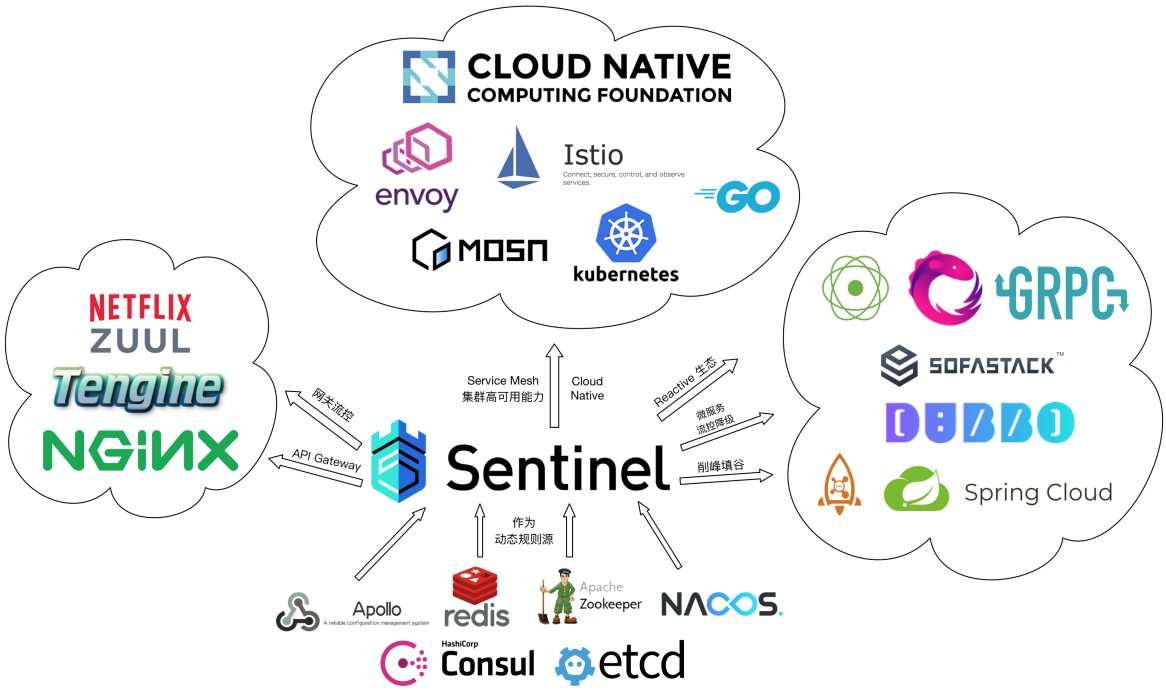
- 2012 年，Sentinel 诞生，主要功能为入口流量控制。
- 2013-2017 年，Sentinel 在阿里巴巴集团内部迅速发展，成为基础技术模块，覆盖了所有的核心场景。Sentinel 也因此积累了大量的流量归整场景以及生产实践。
- 2018 年，Sentinel 开源，并持续演进。
- 2019 年，Sentinel 朝着多语言扩展的方向不断探索，推出 C++ 原生版本，同时针对 Service Mesh 场景也推出了 Envoy 集群流量控制支持，以解决 Service Mesh 架构下多语言限流的问题。
- 2020 年，推出 Sentinel Go 版本，继续朝着云原生方向演进。

1.3 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、集群流量控制、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 扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

1.4 Sentinel 的主要特性

1.5 Sentinel 的开源生态



1.6 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。
- 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。

1.7 Sentinel 基本概念

资源

资源是 Sentinel 的关键概念。它可以是 Java 应用程序中的任何内容，例如，由应用程序提供的服务，或由应用程序调用的其它应用提供的服务，甚至可以是一段代码。在接下来的文档中，我们都会用资源来描述代码块。

只要通过 Sentinel API 定义的代码，就是资源，能够被 Sentinel 保护起来。大部分情况下，可以使用方法签名，URL，甚至服务名称作为资源名来标示资源。

规则

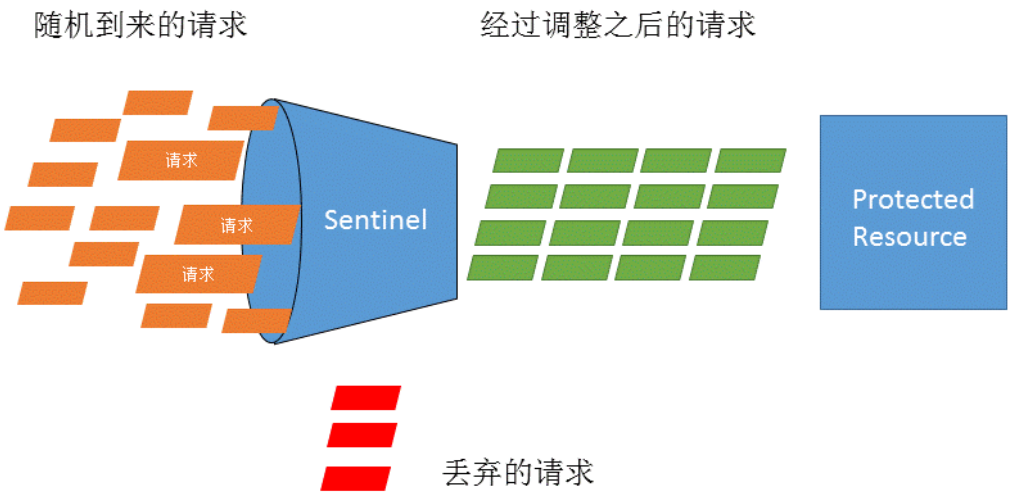
围绕资源的实时状态设定的规则，可以包括流量控制规则、熔断降级规则以及系统保护规则。所有规则可以动态实时调整。

1.8 Sentinel 功能和设计理念

流量控制

什么是流量控制

流量控制在网络传输中是一个常用的概念，它用于调整网络包的发送数据。然而，从系统稳定性角度考虑，在处理请求的速度上，也有非常多的讲究。任意时间到来的请求往往是随机不可控的，而系统的处理能力是有限的。我们需要根据系统的处理能力对流量进行控制。Sentinel 作为一个调配器，可以根据需要把随机的请求调整成合适的形状，如下图所示：



流量控制设计理念

流量控制有以下几个角度:

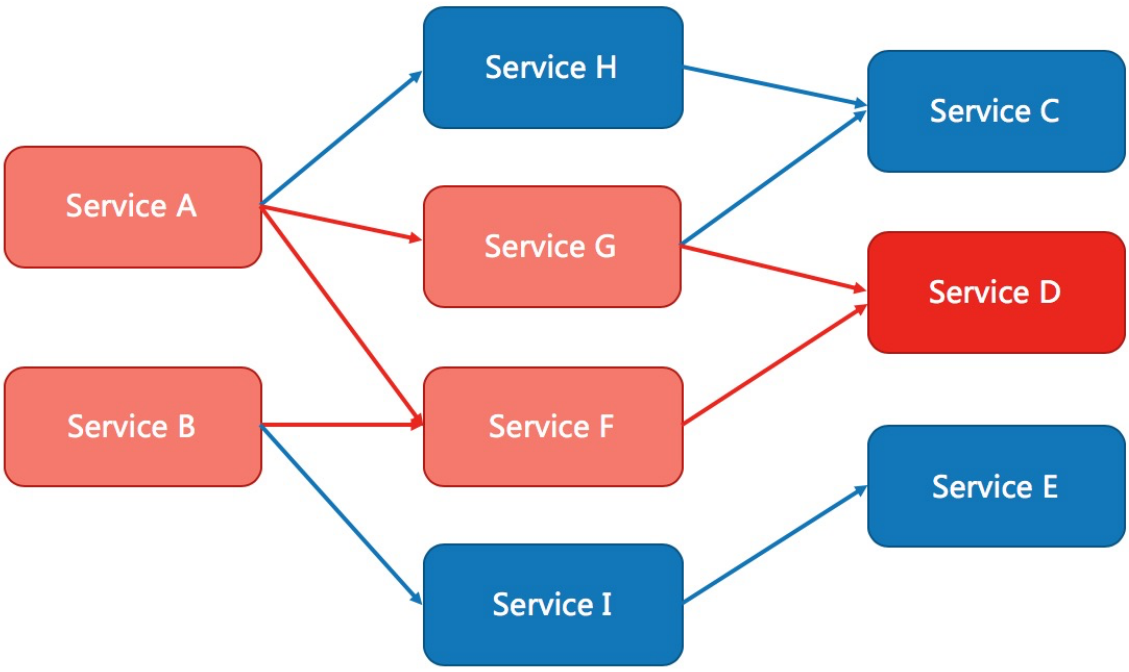
- 资源的调用关系，例如资源的调用链路，资源和资源之间的关系；
- 运行指标，例如 QPS、线程池、系统负载等；
- 控制的效果，例如直接限流、冷启动、排队等。

Sentinel 的设计理念是让您自由选择控制的角度，并进行灵活组合，从而达到想要的效果。

熔断降级

什么是熔断降级

除了流量控制以外，及时对调用链路中的不稳定因素进行熔断也是 Sentinel 的使命之一。由于调用关系的复杂性，如果调用链路中的某个资源出现了不稳定，可能会导致请求发生堆积，进而导致级联错误。



Sentinel 和 Hystrix 的原则是一致的: 当检测到调用链路中某个资源出现不稳定的表现，例如请求响应时间长或异常比例升高的时候，则对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联故障。

熔断降级设计理念

在限制的手段上，Sentinel 和 Hystrix 采取了完全不一样的方法。

Hystrix 通过 线程池隔离 的方式，来对依赖（在 Sentinel 的概念中对应 *资源*）进行了隔离。这样做的好处是资源和资源之间做到了最彻底的隔离。缺点是除了增加了线程切换的成本（过多的线程池导致线程数目过多），还需要预先给各个资源做线程池大小的分配。

Sentinel 对这个问题采取了两种手段:

- 通过并发线程数进行限制

和资源池隔离的方法不同，Sentinel 通过限制资源并发线程的数量，来减少不稳定资源对其它资源的影响。这样不但没有线程切换的损耗，也不需要您预先分配线程池的大小。当某个资源出现不稳定的情况下，例如响应时间变长，对资源的直接影响就是会造成线程数的逐步堆积。当线程数在特定资源上堆积到一定的数量之后，对该资源的新请求就会被拒绝。堆积的线程完成任务后才开始继续接收请求。

- 通过响应时间对资源进行降级

除了对并发线程数进行控制以外，Sentinel 还可以通过响应时间来快速降级不稳定的资源。当依赖的资源出现响应时间过长后，所有对该资源的访问都会被直接拒绝，直到过了指定的时间窗口之后才重新恢复。

系统自适应保护

Sentinel 同时提供系统维度的自适应保护能力。防止雪崩，是系统防护中重要的一环。当系统负载较高的时候，如果还持续让请求进入，可能会导致系统崩溃，无法响应。在集群环境下，网络负载均衡会把本应这台机器承载的流量转发到其它的机器上去。如果这个时候其它的机器也处在一个边缘状态的时候，这个增加的流量就会导致这台机器也崩溃，最后导致整个集群不可用。

针对这个情况，Sentinel 提供了对应的保护机制，让系统的入口流量和系统的负载达到一个平衡，保证系统在能力范围之内处理最多的请求。

1.9 Sentinel 是如何工作的

Sentinel 的主要工作机制如下：

- 对主流框架提供适配或者显示的 API，来定义需要保护的资源，并提供设施对资源进行实时统计和调用链路分析。
- 根据预设的规则，结合对资源的实时统计信息，对流量进行控制。同时，Sentinel 提供开放的接口，方便您定义及改变规则。
- Sentinel 提供实时的监控系统，方便您快速了解目前系统的状态。

2 Sentinel快速入门

2.1 Sentinel HelloWorld实现

第一步：创建项目，引入依赖

为了方便后期sentinel和springcloud以及springcloud alibaba其他组件整合开发讲解，我们这边规范好springboot，springcloud，springcloud alibaba的版本，选用稳定的组合版本；

Spring Cloud Version	Spring Cloud Alibaba Version	Spring Boot Version
Spring Cloud 2020.0	2021.1	2.4.2.RELEASE
Spring Cloud Hoxton.SR8	2.2.5.RELEASE	2.3.2.RELEASE
Spring Cloud Greenwich.SR6	2.1.4.RELEASE	2.1.13.RELEASE
Spring Cloud Hoxton.SR3	2.2.1.RELEASE	2.2.5.RELEASE
Spring Cloud Hoxton.RELEASE	2.2.0.RELEASE	2.2.X.RELEASE
Spring Cloud Greenwich	2.1.2.RELEASE	2.1.X.RELEASE
Spring Cloud Finchley	2.0.4.RELEASE(停止维护，建议升级)	2.0.X.RELEASE
Spring Cloud Edgware	1.5.1.RELEASE(停止维护，建议升级)	1.5.X.RELEASE

具体查看：https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E

先创建父pom项目 **sentinel-test**

属性，依赖管理以及插件定义；

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <spring-cloud.version>Hoxton.SR8</spring-cloud.version>
    <springboot.version>2.3.2.RELEASE</springboot.version>
    <springcloudalibaba.version>2.2.4.RELEASE</springcloudalibaba.version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${springboot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
```

```
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${springcloudalibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>


<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

新建module项目 **sentinel-helloworld**

引入web和sentinel依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

application.yml配置

```
server:
  port: 80
  servlet:
      context-path: "/"
```

新建启动类：**SentinelHelloWorldApplication**

```
package com.java1234;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author java1234_小锋
 * @site www.java1234.com
```

```
 * @company Java知识分享网
 * @create 2021-05-04 15:18
 */
@SpringBootApplication
public class SentinelHelloWorldApplication {

    public static void main(String[] args) {
        SpringApplication.run(SentinelHelloWorldApplication.class,args);
    }
}
```

第二步：定义规则和使用限流规则

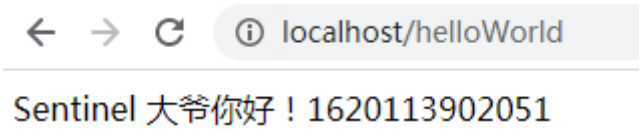
```
package com.java1234.controller;

import com.alibaba.csp.sentinel.Entry;
import com.alibaba.csp.sentinel.SphU;
import com.alibaba.csp.sentinel.slots.block.RuleConstant;
import com.alibaba.csp.sentinel.slots.block.flow.FlowRule;
import com.alibaba.csp.sentinel.slots.block.flow.FlowRuleManager;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.List;

/**
 * @author java1234_小锋
 * @site www.java1234.com
 * @company Java知识分享网
 * @create 2021-05-04 14:35
 */
@RestController
public class SentinelHelloWorld {


    @RequestMapping("helloWorld")
    public String helloWorld(){
        try(Entry entry=SphU.entry("HelloWorld")){ // 使用限流规则 HelloWorld
            return "Sentinel 大爷你好！"+System.currentTimeMillis();
        }catch (Exception e){
            e.printStackTrace();
            return "系统繁忙，请稍后！";  // 降级处理
        }
    }

    /**
     * 定义限流规则
     * PostConstruct 构造方法执行完后执行方法 定义和加载限流规则
     */
    @PostConstruct
    public void initFlowRules(){
```

```
        List<FlowRule> rules=new ArrayList<>(); // 定义限流规则集合
        FlowRule rule=new FlowRule();  // 定义限流规则
        rule.setResource("HelloWorld");  // 定义限流资源
        rule.setGrade(RuleConstant.FLOW_GRADE_QPS); // 定义限流规则类型
        rule.setCount(2); // 定义QPS阈值 每秒最多通过的请求个数
        rules.add(rule);  // 添加规则到集合
        FlowRuleManager.loadRules(rules); // 加载规则集合
    }
}
```

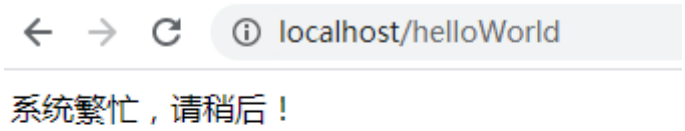
第三步：测试

浏览器请求：http://localhost/helloWorld

正常请求：



当访问频率超过QPS阈值2，则sentinel降级返回异常信息；



测试通过；

2.2 Sentinel dashboard控制台搭建

Sentinel 控制台是流量控制、熔断降级规则统一配置和管理的入口，它为用户提供了机器自发现、簇点链路自发现、监控、规则配置等功能。在 Sentinel 控制台上，我们可以配置规则并实时查看流量控制效果。

下载地址

https://github.com/alibaba/Sentinel/releases/

最新版本1.8.1

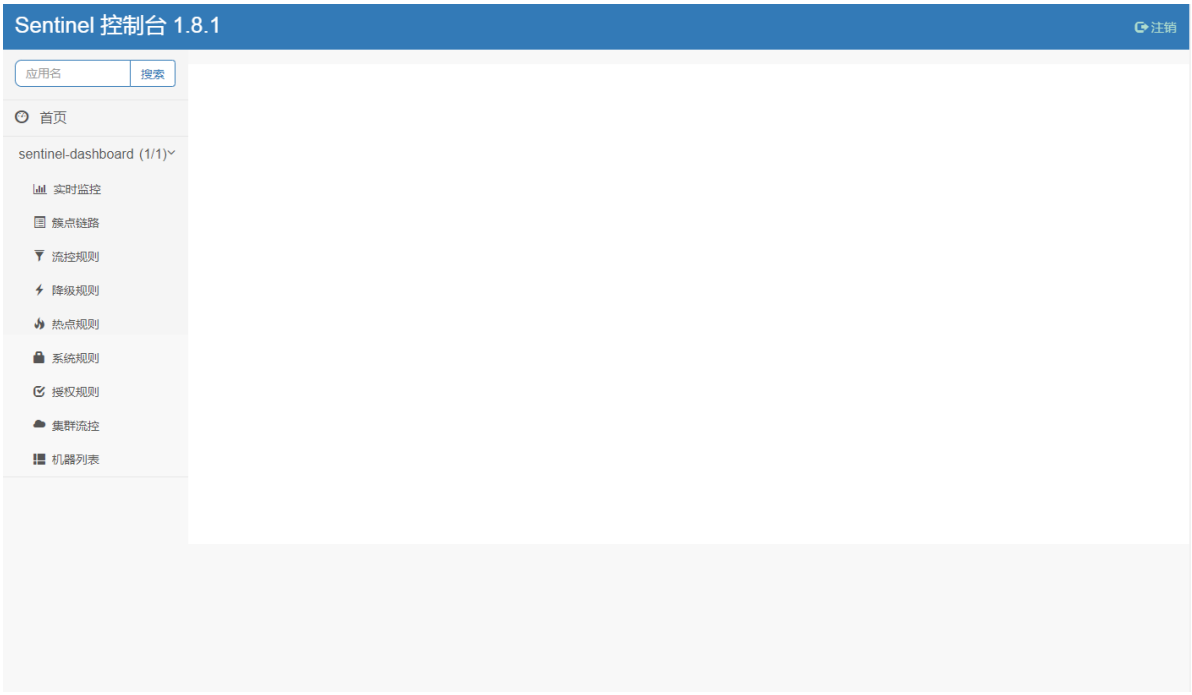
https://github.com/alibaba/Sentinel/releases/download/1.8.1/sentinel-dashboard-1.8.1.jar

注意：启动 Sentinel 控制台需要 JDK 版本为 1.8 及以上版本。

```
java -Dserver.port=8080 -Dcsp.sentinel.dashboard.server=localhost:8080 -
Dproject.name=sentinel-dashboard -jar sentinel-dashboard.jar
```

其中 `-Dserver.port=8080` 用于指定 Sentinel 控制台端口为 `8080`。

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



2.3 客户端接入控制台

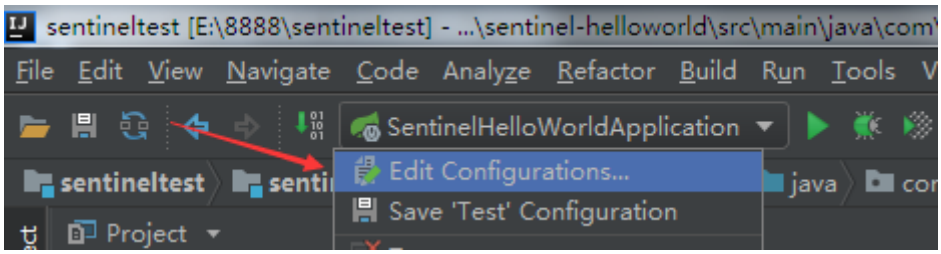
1，引入依赖JAR

客户端需要引入 Transport 模块来与 Sentinel 控制台进行通信。您可以通过 `pom.xml` 引入 JAR 包:

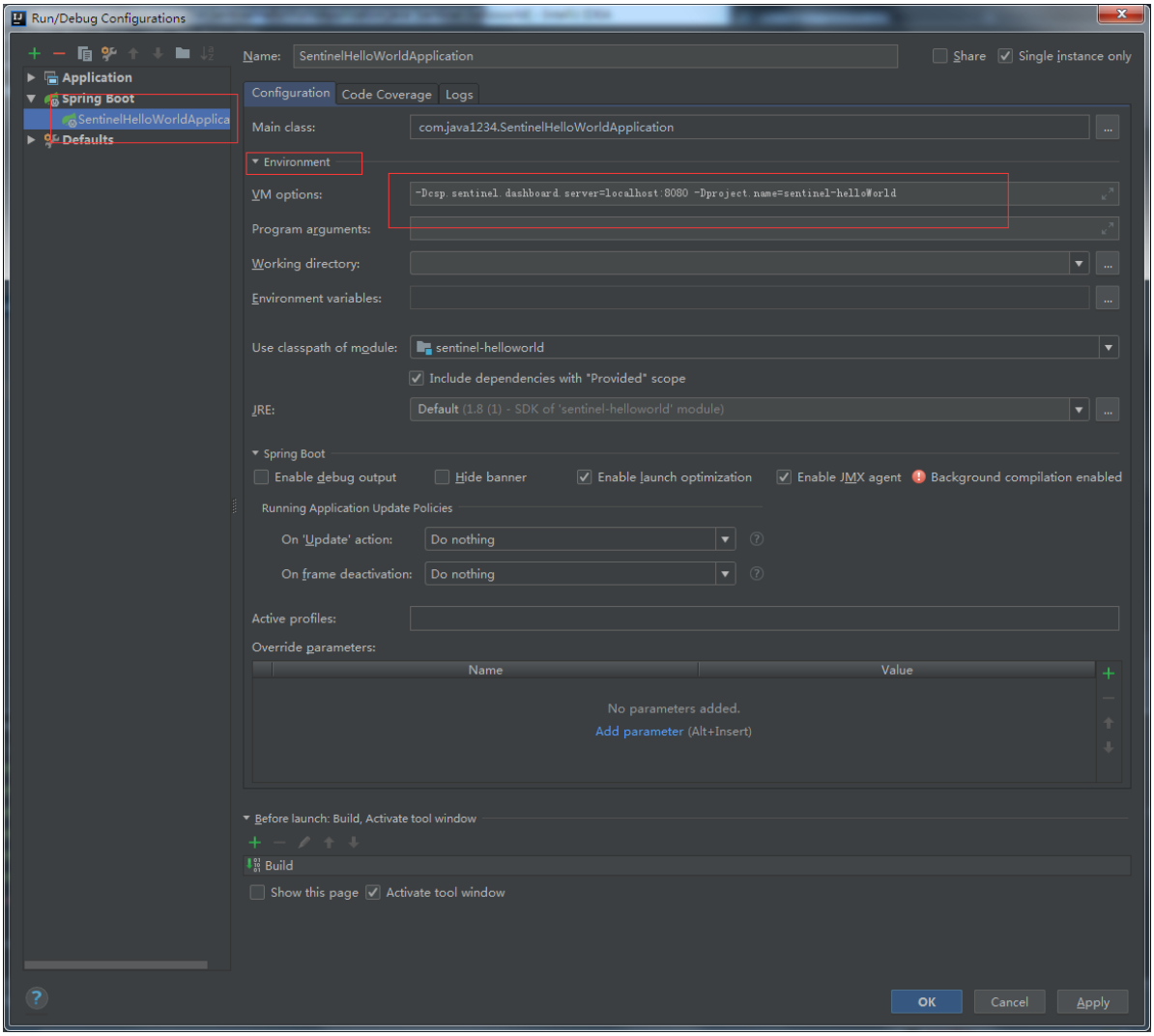
```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-transport-simple-http</artifactId>
    <version>1.8.0</version>
</dependency>
```

2，IDEA配置JVM启动参数

选择 `Edit Configurations` 编辑启动配置：



配置里面，找到环境变量，VM options配置



加下两个配置：

```
-Dcsp.sentinel.dashboard.server=localhost:8080 -Dproject.name=sentinel-helloWorld
```

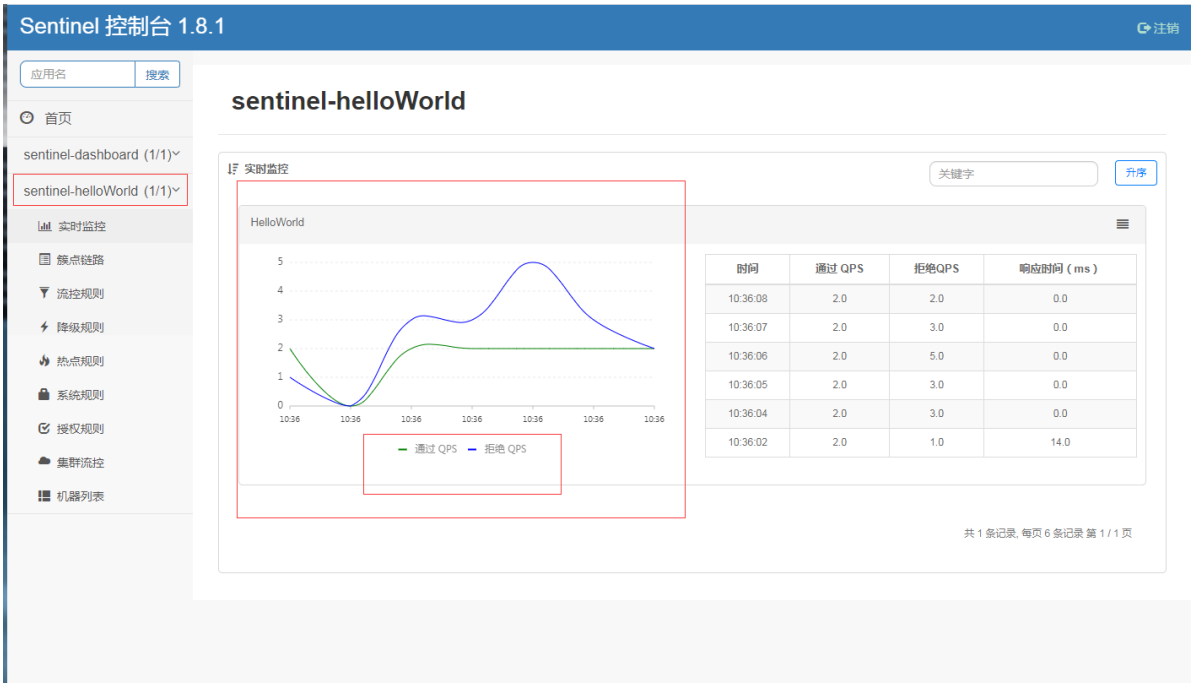
-Dcsp.sentinel.dashboard.server=localhost:8080： 指定控制台地址和端口

-Dproject.name=sentinel-helloWorld： 设置控制台上显示的项目名称

3，测试

重新启动项目，http://localhost/helloWorld 多运行刷新几次；

然后我们查看Sentineil控制台中实时监控信息；

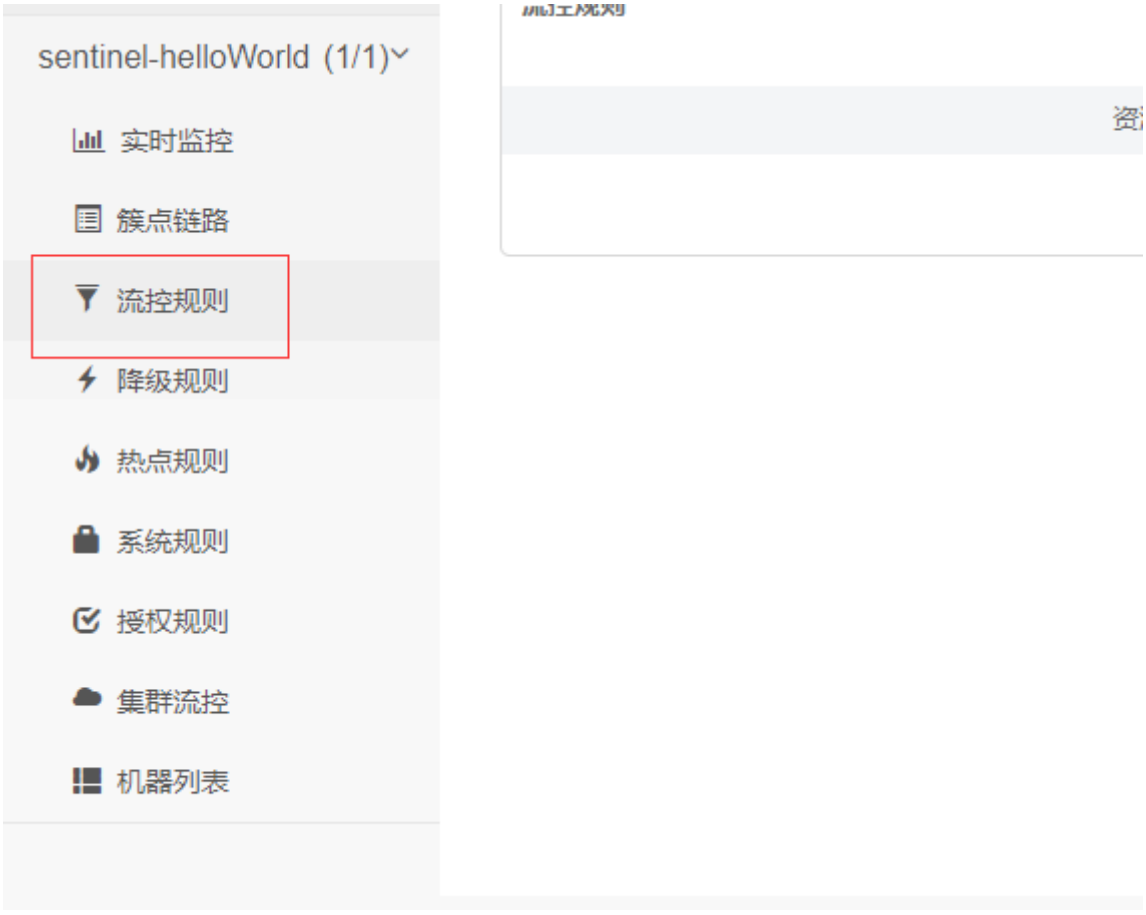


2.4 Sentinel控制台设置限流规则

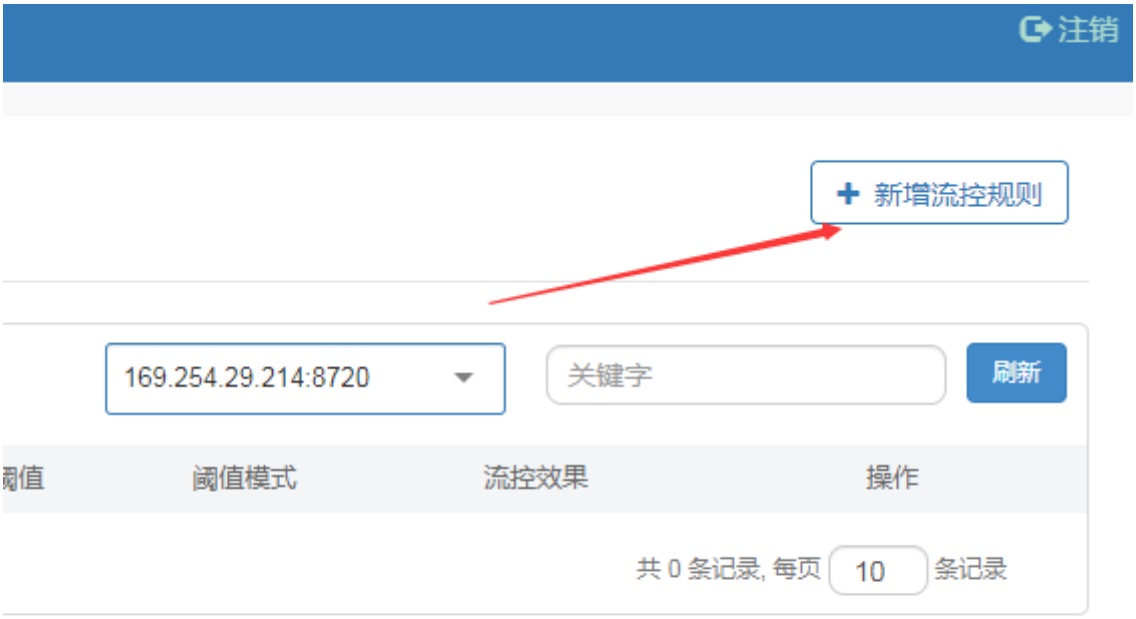
之前我们通过硬编码方式，设置限流规则，这种方式缺点是不方便修改维护规则，不建议使用；

我们以后用Sentinel控制台里设置规则的方式来操作；

点击 流控规则 菜单



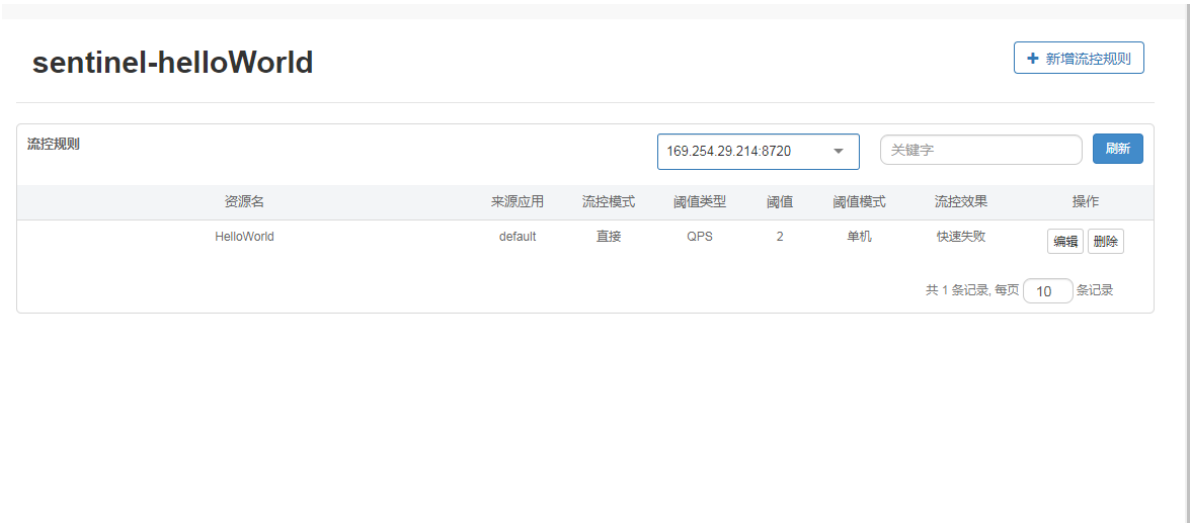
点击 新增流控规则 按钮



填写流控规则，然后点击 新增 按钮

（注意：流控规则的“资源名”要和代码里使用的规则名称一致）

我们可以很方便编辑，删除规则名称；



测试

先把硬编码的流控规则注释掉

```
    @RequestMapping("helloWorld")
    public String helloWorld(){
        try(Entry entry=SphU.entry( name: "HelloWorld")){ // 使用限流规则 HelloWorld
            return "Sentinel 大爷你好！"+System.currentTimeMillis();
        }catch (Exception e){
            e.printStackTrace();
            return "系统繁忙，请稍后！";  // 降级处理
        }
    }

    /**
     * 定义限流规则
     * PostConstruct 构造方法执行完后执行方法 定义和加载限流规则
     */
    /*@PostConstruct
    public void initFlowRules(){
        List<FlowRule> rules=new ArrayList<>(); // 定义限流规则集合
        FlowRule rule=new FlowRule();  // 定义限流规则
        rule.setResource("HelloWorld");  // 定义限流资源
        rule.setGrade(RuleConstant.FLOW_GRADE_QPS); // 定义限流规则类型
        rule.setCount(2); // 定义QPS阈值 每秒最多通过的请求个数
        rules.add(rule);  // 添加规则到集合
        FlowRuleManager.loadRules(rules); // 加载规则集合
    }*/
}
```

然后，http://localhost/helloWorld 浏览器地址，多刷新几次，Sentinel实时监控就有数据了。

3 Sentinel基本使用

3.1 定义资源

资源是 Sentinel 的关键概念。它可以是 Java 应用程序中的任何内容，例如，由应用程序提供的服务，或由应用程序调用的其它应用提供的服务，甚至可以是一段代码。在接下来的文档中，我们都会用资源来描述代码块。

只要通过 Sentinel API 定义的代码，就是资源，能够被 Sentinel 保护起来。大部分情况下，可以使用方法签名，URL，甚至服务名称作为资源名来标示资源。

Sentinel 可以简单的分为 Sentinel 核心库和 Dashboard。核心库不依赖 Dashboard，但是结合 Dashboard 可以取得最好的效果。

我们说的资源，可以是任何东西，服务，服务里的方法，甚至是一段代码。使用 Sentinel 来进行资源保护，主要分为几个步骤:

1. 定义资源
2. 定义规则
3. 检验规则是否生效

先把可能需要保护的资源定义好（埋点），之后再配置规则。也可以理解为，只要有了资源，我们就可以在任何时候灵活地定义各种流量控制规则。在编码的时候，只需要考虑这个代码是否需要保护，如果需要保护，就将之定义为一个资源。

对于主流的框架，我们提供适配，只需要按照适配中的说明配置，Sentinel 就会默认定义提供的服务，方法等为资源。

3.1.1 方式一：主流框架的默认适配

为了减少开发的复杂程度，我们对大部分的主流框架，例如 Web Servlet、Dubbo、Spring Cloud、gRPC、Spring WebFlux、Reactor 等都做了适配。您只需要引入对应的依赖即可方便地整合 Sentinel。可以参见: 主流框架的适配。

3.1.2 方式二：抛出异常的方式定义资源

`SphU` 包含了 try-catch 风格的 API。用这种方式，当资源发生了限流之后会抛出 `BlockException`。这个时候可以捕捉异常，进行限流之后的逻辑处理。示例代码如下:

```
// 1.5.0 版本开始可以利用 try-with-resources 特性（使用有限制）
// 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。
try (Entry entry = SphU.entry("resourceName")) {
  // 被保护的业务逻辑
  // do something here...
} catch (BlockException ex) {
  // 资源访问阻止，被限流或被降级
  // 在此处进行相应的处理操作
}
```

特别地，若 entry 的时候传入了热点参数，那么 exit 的时候也一定要带上对应的参数（`exit(count, args)`），否则可能会有统计错误。这个时候不能使用 try-with-resources 的方式。另外通过 `Tracer.trace(ex)` 来统计异常信息时，由于 try-with-resources 语法中 catch 调用顺序的问题，会导致无法正确统计异常数，因此统计异常信息时也不能在 try-with-resources 的 catch 块中调用 `Tracer.trace(ex)`。

手动 exit 示例：

```
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，注意数目不能太多（超过 1K），超出几千请作为参数传入而不要直接作为资源名
  // EntryType 代表流量类型（inbound/outbound），其中系统规则只对 IN 类型的埋点生效
  entry = SphU.entry("自定义资源名");
  // 被保护的业务逻辑
  // do something...
} catch (BlockException ex) {
  // 资源访问阻止，被限流或被降级
  // 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
  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();
  }
}
```

热点参数埋点示例：

```
Entry entry = null;
try {
    // 若需要配置例外项，则传入的参数只支持基本类型。
    // EntryType 代表流量类型，其中系统规则只对 IN 类型的埋点生效
    // count 大多数情况都填 1，代表统计为一次调用。
    entry = SphU.entry(resourceName, EntryType.IN, 1, paramA, paramB);
    // Your logic here.
} catch (BlockException ex) {
    // Handle request rejection.
} finally {
    // 注意：exit 的时候也一定要带上对应的参数，否则可能会有统计错误。
    if (entry != null) {
```

```
        entry.exit(1, paramA, paramB);
    }
}
```

`SphU.entry()` 的参数描述：

参数名	类型	解释	默认值
entryType	`EntryType`	资源调用的流量类型，是入口流量（`EntryType.IN`）还是出口流量（`EntryType.OUT`），注意系统规则只对 IN 生效	`EntryType.OUT`
count	`int`	本次资源调用请求的 token 数目	1
args	`Object[]`	传入的参数，用于热点参数限流	无

注意：`SphU.entry(xxx)` 需要与 `entry.exit()` 方法成对出现，匹配调用，否则会导致调用链记录异常，抛出 `ErrorEntryFreeException` 异常。常见的错误：

- 自定义埋点只调用 `SphU.entry()`，没有调用 `entry.exit()`
- 顺序错误，比如：`entry1 -> entry2 -> exit1 -> exit2`，应该为 `entry1 -> entry2 -> exit2 -> exit1`

3.1.3 方式三：返回布尔值方式定义资源

`SphO` 提供 if-else 风格的 API。用这种方式，当资源发生了限流之后会返回 `false`，这个时候可以根据返回值，进行限流之后的逻辑处理。示例代码如下：

```
// 资源名可使用任意有业务语义的字符串
if (SphO.entry("自定义资源名")) {
  // 务必保证finally会被执行
  try {
    /**
    * 被保护的业务逻辑
    */
  } finally {
    SphO.exit();
  }
} else {
  // 资源访问阻止，被限流或被降级
  // 进行相应的处理操作
}
```

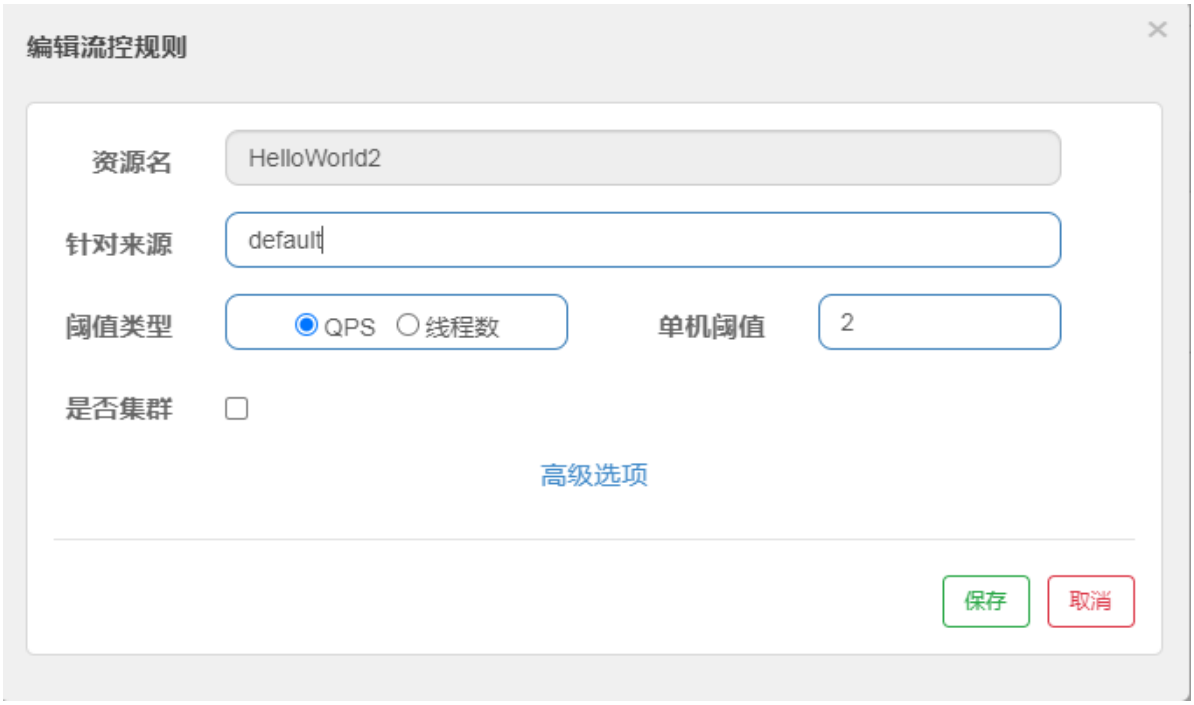
注意：`SphO.entry(xxx)` 需要与 SphO.exit() 方法成对出现，匹配调用，位置正确，否则会导致调用链记录异常，抛出 ErrorEntryFreeException ` 异常。

实例：

```
/**
 * 返回布尔值方式定义资源
 * @return
```

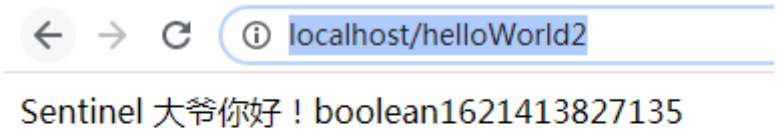
```
 */
@RequestMapping("helloWorld2")
public String helloWorld2(){
    // 资源名可使用任意有业务语义的字符串
    if (SphO.entry("HelloWorld2")) {
        // 务必保证finally会被执行
        try {
            /**
             * 被保护的业务逻辑
             */
            return "Sentinel 大爷你好! boolean"+System.currentTimeMillis();
        } finally {
            SphO.exit();
        }
    } else {
        // 资源访问阻止，被限流或被降级
        // 进行相应的处理操作
        return "系统繁忙，请稍后! ";  // 降级处理
    }
}
```

sentinel控制台添加流控规则：

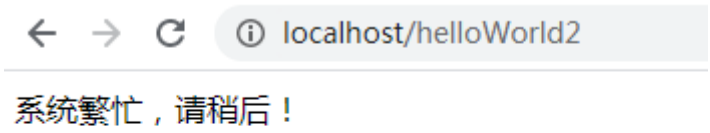


访问 http://localhost/helloWorld2

正常访问显示：



如果频繁访问会出现：



触碰降级规则，返回降级信息；

3.1.4 方式四：注解方式定义资源

Sentinel 支持通过 `@SentinelResource` 注解定义资源并配置 `blockHandler` 和 `fallback` 函数来进行限流之后的处理。

第一步：引入`@SentinelResource`注解依赖支持

Sentinel 提供了 `@SentinelResource` 注解用于定义资源，并提供了 AspectJ 的扩展用于自动定义资源、处理 `BlockException` 等。使用 Sentinel Annotation AspectJ Extension 的时候需要引入以下依赖：

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

第二步：创建AspectJ 配置类

Spring Cloud Alibaba

若您是通过 Spring Cloud Alibaba 接入的 Sentinel，则无需额外进行配置即可使用 `@SentinelResource` 注解。

Spring AOP

若您的应用使用了 Spring AOP（无论是 Spring Boot 还是传统 Spring 应用），您需要通过配置的方式将 `SentinelResourceAspect` 注册为一个 Spring Bean：

```
@Configuration
public class SentinelAspectConfiguration {

    @Bean
    public SentinelResourceAspect sentinelResourceAspect() {
        return new SentinelResourceAspect();
    }
}
```

第三步：新建Controller测试方法

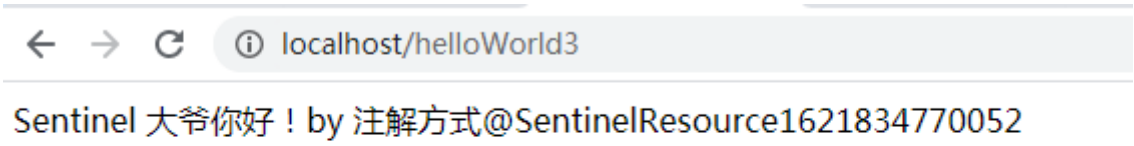
```
/**
 * 注解方式定义资源
 * @SentinelResource  value 资源名称
 * @SentinelResource blockHandler 调用被限流/降级/系统保护的时候调用的方法
 * @return
 */
@SentinelResource(value = "helloWorld3",blockHandler =
"blockHandlerForHelloWorld3")
@RequestMapping("helloWorld3")
public String helloWorld3(){
    return "Sentinel 大爷你好！by 注解方式
@SentinelResource"+System.currentTimeMillis();
}

/**
 * 原方法调用被限流/降级/系统保护的时候调用
 * @param ex
 * @return
 */
public String blockHandlerForHelloWorld3(BlockException ex) {
    ex.printStackTrace();
    return "系统繁忙，请稍后！";
}
```

第四步：Sentinel控制台新增流控规则

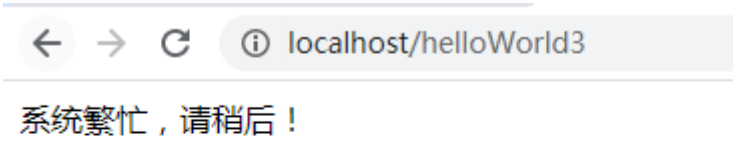
第五步：测试

浏览器请求：http://localhost/helloWorld3

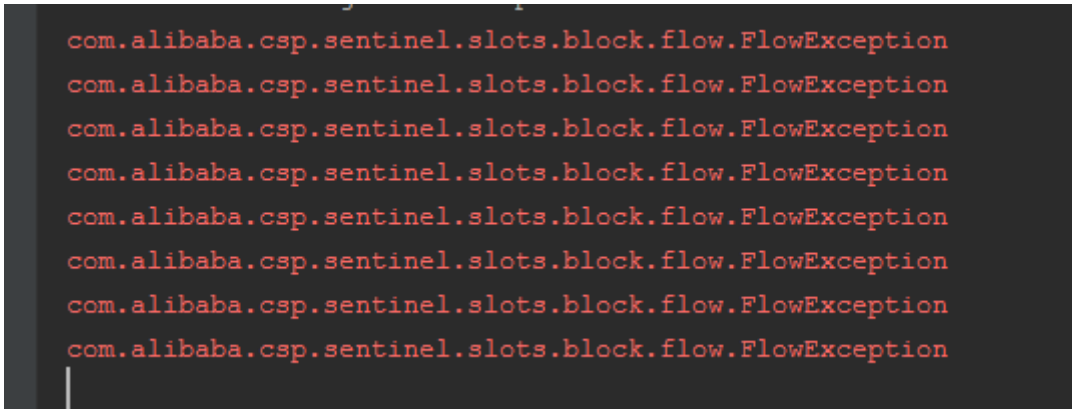
正常访问：



频繁访问：



控制台异常打印：



3.1.5 方式五：异步调用支持

Sentinel 支持异步调用链路的统计。在异步调用中，需要通过 `SphU.asyncEntry(xxx)` 方法定义资源，并通常需要在异步的回调函数中调用 `exit` 方法。

第一步：启动类加注解 `@EnableAsync`，让项目支持异步调用支持

```
@SpringBootApplication
@EnableAsync
public class SentinelHelloWorldApplication {

    public static void main(String[] args) {
        SpringApplication.run(SentinelHelloWorldApplication.class,args);
    }
}
```

第二步：创建AsyncService异步调用类以及方法

```
package com.java1234.service;

import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

/**
 * @author java1234_小锋
 * @site www.java1234.com
 * @company Java知识分享网
 * @create 2021-05-27 13:18
 */
@Service
public class AsyncService {

    @Async
    public void doSomethingAsync(){
        System.out.println("async start...");
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        System.out.println("async end...");
    }
}
```

第三步：创建Controller方法

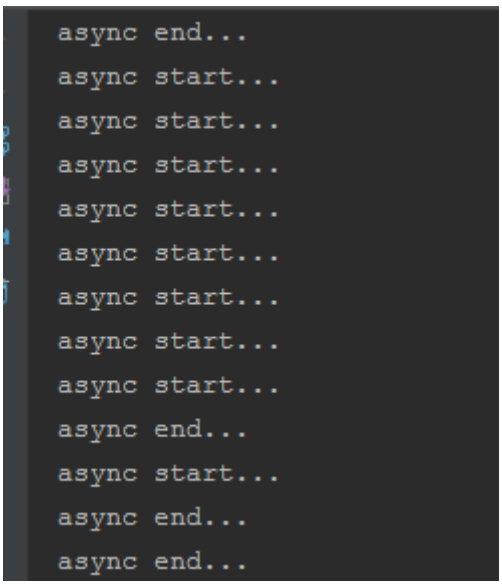
```
@RequestMapping("helloWorld4")
public void helloWorld4(){
    AsyncEntry asyncEntry =null;
    try {
        asyncEntry = SphU.asyncEntry("helloWorld4");
        asyncService.doSomethingAsync();
    } catch (BlockException e) {
        System.out.println("系统繁忙，请稍后！");
    }finally {
        if(asyncEntry!=null){
            asyncEntry.exit();
        }
    }
}
```

第四步：Sentinel控制台新增流控规则

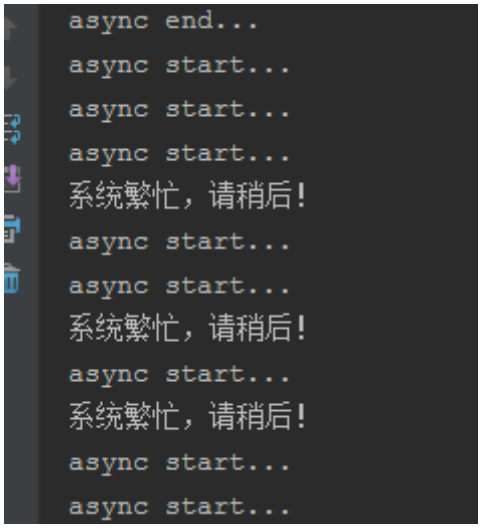
第五步：测试

浏览器请求：http://localhost/helloWorld4

正常访问控制台输出：



频繁访问控制台输出：



3.2 规则的种类

Sentinel 的所有规则都可以在内存态中动态地查询及修改，修改之后立即生效。同时 Sentinel 也提供相关 API，供您来定制自己的规则策略。

Sentinel 支持以下几种规则：

- **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**

3.2.1 流量控制规则

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。

重要属性：

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

限流类型分为:

- **QPS** 每秒请求数限制
- **线程数** 资源使用线程数限制

流控模式

- **直接** 资源直接限流，这个就是简单的限流。

- **关联** 关联模式需要填写关联资源的路径，意为如果关联资源的流量超额之后，限流自己（自己为资源名填写的路径）。
- **链路** 如果是链路模式需要填写入口资源，限制入口资源对自己的调用。

流控效果

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F1owException。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（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。

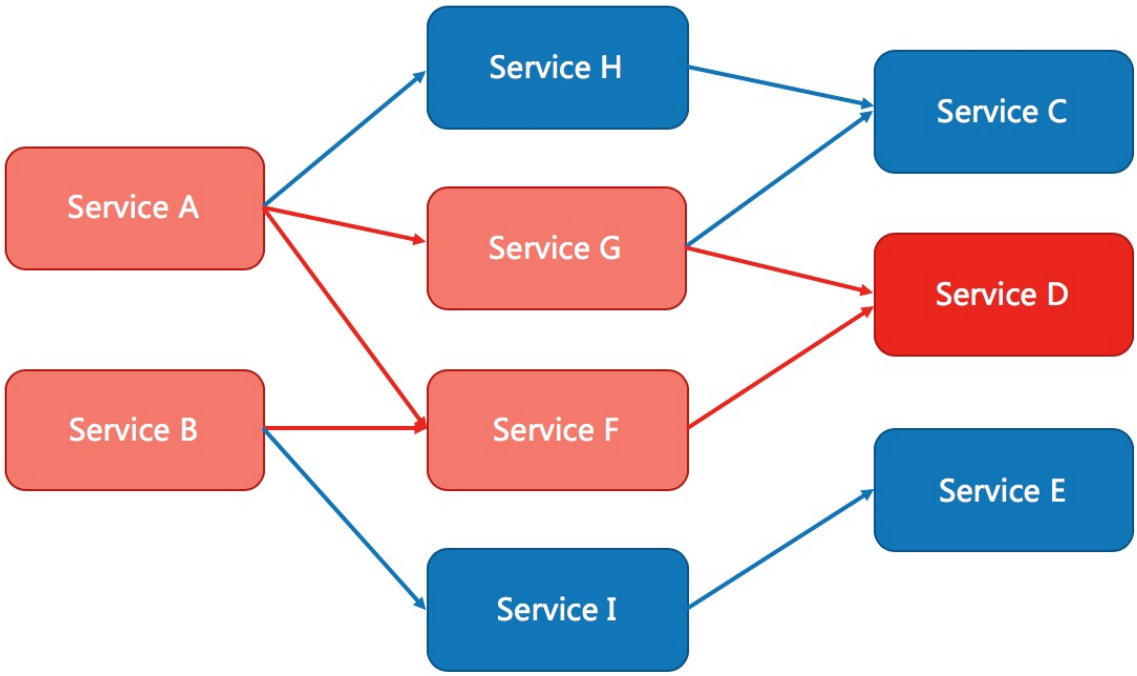
排队等待（RuleConstant.cONTROL_BEHAV工OR_RATE_LIMITER）方式会严格控制请求通过的间隔时间,也即是让请求以均匀的速度通过，对应的是漏桶算法。

同一个资源可以同时有多个限流规则，检查规则时会依次检查。

3.2.2 熔断降级规则

概述

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。一个服务常常会调用别的模块，可能是另外的一个远程服务、数据库，或者第三方 API 等。例如，支付的时候，可能需要远程调用银联提供的 API；查询某个商品的价格，可能需要进行数据库查询。然而，这个被依赖服务的稳定性是不能保证的。如果依赖的服务出现了不稳定的情况，请求的响应时间变长，那么调用服务的方法的响应时间也会变长，线程会产生堆积，最终可能耗尽业务自身的线程池，服务本身也变得不可用。



现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。因此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。

> **注意**：本文档针对 Sentinel 1.8.0 及以上版本。1.8.0 版本对熔断降级特性进行了全新的改进升级，请使用最新版本以更好地利用熔断降级的能力。

熔断策略

Sentinel 提供以下几种熔断策略：

- 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。
- 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%。
- 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。为了统计异常比例或异常数，需要通过`Tracer.trace(ex)`记录业务异常。示例：

```
Entry entry = null;
try {
  entry = SphU.entry(key, EntryType.IN, key);

  // Write your biz code here.
  // <<BIZ CODE>>
} catch (Throwable t) {
  if (!BlockException.isBlockException(t)) {
    Tracer.trace(t);
  }
} finally {
  if (entry != null) {
    entry.exit();
  }
}
```

开源整合模块，如 Sentinel Dubbo Adapter, Sentinel Web Servlet Filter 或`@SentinelResource`注解会自动统计业务异常，无需手动调用。

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	默认值
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	慢调用比例
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）	5
statIntervalMs	统计时长（单位为 ms），如 60*1000 代表分钟级（1.8.0 引入）	1000 ms
slowRatioThreshold	慢调用比例阈值，仅慢调用比例模式有效（1.8.0 引入）	

同一个资源可以同时有多个降级规则。

理解上面规则的定义之后，我们可以通过调用 `DegradeRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则。

```
private void initDegradeRule() {
    List<DegradeRule> rules = new ArrayList<>();
    DegradeRule rule = new DegradeRule();
    rule.setResource(KEY);
    // set threshold RT, 10 ms
    rule.setCount(10);
    rule.setGrade(RuleConstant.DEGRADE_GRADE_RT);
    rule.setTimeWindow(10);
    rules.add(rule);
    DegradeRuleManager.loadRules(rules);
}
```

熔断器事件监听

Sentinel 支持注册自定义的事件监听器监听熔断器状态变换事件（state change event）。示例：

```
EventObserverRegistry.getInstance().addStateChangeObserver("logging",
    (prevState, newState, rule, snapshotValue) -> {
        if (newState == State.OPEN) {
            // 变换至 OPEN state 时会携带触发时的值
            System.err.println(String.format("%s -> OPEN at %d,
snapshotValue=%.2f", prevState.name(),
                TimeUtil.currentTimeMillis(), snapshotValue));
        } else {
            System.err.println(String.format("%s -> %s at %d", prevState.name(),
newState.name(),
                TimeUtil.currentTimeMillis()));
        }
    });
```

3.2.3 系统保护规则

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。

系统规则包含下面几个重要的属性：

Field	说明	默认值
highestSystemLoad	`load1` 触发值，用于触发自适应控制阶段	-1 (不生效)
avgRt	所有入口流量的平均响应时间	-1 (不生效)
maxThread	入口流量的最大并发数	-1 (不生效)
qps	所有入口资源的 QPS	-1 (不生效)
highestCpuUsage	当前系统的 CPU 使用率（0.0-1.0）	-1 (不生效)

理解上面规则的定义之后，我们可以通过调用 `SystemRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则。

```
private void initSystemRule() {
    List<SystemRule> rules = new ArrayList<>();
    SystemRule rule = new SystemRule();
    rule.setHighestSystemLoad(10);
    rules.add(rule);
    SystemRuleManager.loadRules(rules);
}
```

控制台对应操作：

在开始之前，我们先了解一下系统保护的目的：

- 保证系统不被拖垮
- 在系统稳定的前提下，保持系统的吞吐量

长期以来，系统保护的思路是根据硬指标，即系统的负载 (load1) 来做系统过载保护。当系统负载高于某个阈值，就禁止或者减少流量的进入；当 load 开始好转，则恢复流量的进入。**这个思路给我们带来了不可避免的两个问题**：

- load 是一个“结果”，如果根据 load 的情况来调节流量的通过率，那么就始终有**延迟性**。也就意味着通过率的任何调整，都会过一段时间才能看到效果。当前通过率是使 load 恶化的一个动作，那么也至少要过 1 秒之后才能观测到；同理，如果当前通过率调整是让 load 好转的一个动作，也需要 1 秒之后才能继续调整，这样就浪费了系统的处理能力。所以我们看到的曲线，总是会有抖动。
- **恢复慢**。想象一下这样的一个场景（真实），出现了这样一个问题，下游应用不可靠，导致应用 RT 很高，从而 load 到了一个很高的点。过了一段时间之后下游应用恢复了，应用 RT 也相应减少。这个时候，其实应该大幅度增大流量的通过率；但是由于这个时候 load 仍然很高，通过率的恢复仍然不高。

TCP BBR 的思想给了我们一个很大的启发。我们应该根据系统能够处理的请求，和允许进来的请求，来做平衡，而不是根据一个间接的指标（系统 load）来做限流。最终我们追求的目标是 **在系统不被拖垮的情况下，提高系统的吞吐率，而不是 load 一定要到低于某个阈值**。如果我们还是按照固有的思维，超过特定的 load 就禁止流量进入，系统 load 恢复就放开流量，这样做的结果是无论我们怎么调参数，调比例，都是按照果来调节因，都无法取得良好的效果。

Sentinel 在系统自适应保护的做法是，用 load1 作为启动自适应保护的因子，而允许通过的流量由处理请求的能力，即请求的响应时间以及当前系统正在处理的请求速率来决定。

3.2.4 访问控制规则

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。

> 调用方信息通过 `ContextUtil.enter(resourceName, origin)` 方法中的 `origin` 参数传入。

规则配置

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。

示例

比如我们希望控制对资源 `test` 的访问设置白名单，只有来源为 `appA` 和 `appB` 的请求才可通过，则可以配置如下白名单规则：

```
AuthorityRule rule = new AuthorityRule();
rule.setResource("test");
rule.setStrategy(RuleConstant.AUTHORITY_WHITE);
rule.setLimitApp("appA,appB");
AuthorityRuleManager.loadRules(Collections.singletonList(rule));
```

详细示例请参考 AuthorityDemo.

```
/*
 * Copyright 1999-2018 Alibaba Group Holding Ltd.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.alibaba.csp.sentinel.demo.authority;

import java.util.Collections;

import com.alibaba.csp.sentinel.Entry;
import com.alibaba.csp.sentinel.SphU;
import com.alibaba.csp.sentinel.context.ContextUtil;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import com.alibaba.csp.sentinel.slots.block.RuleConstant;
import com.alibaba.csp.sentinel.slots.block.authority.AuthorityRule;
```

```
import com.alibaba.csp.sentinel.slots.block.authority.AuthorityRuleManager;

/**
 * Authority rule is designed for limiting by request origins. In blacklist
mode,
 * requests will be blocked when blacklist contains current origin, otherwise
will pass.
 * In whitelist mode, only requests from whitelist origin can pass.
 *
 * @author Eric Zhao
 */
public class AuthorityDemo {

    private static final String RESOURCE_NAME = "testABC";

    public static void main(String[] args) {
        System.out.println("========Testing for black list========");
        initBlackRules();
        testFor(RESOURCE_NAME, "appA");
        testFor(RESOURCE_NAME, "appB");
        testFor(RESOURCE_NAME, "appC");
        testFor(RESOURCE_NAME, "appE");

        System.out.println("========Testing for white list========");
        initWhiteRules();
        testFor(RESOURCE_NAME, "appA");
        testFor(RESOURCE_NAME, "appB");
        testFor(RESOURCE_NAME, "appC");
        testFor(RESOURCE_NAME, "appE");
    }

    private static void testFor(/*@NonNull*/ String resource, /*@NonNull*/
String origin) {
        ContextUtil.enter(resource, origin);
        Entry entry = null;
        try {
            entry = SphU.entry(resource);
            System.out.println(String.format("Passed for resource %s, origin is
%s", resource, origin));
        } catch (BlockException ex) {
            System.err.println(String.format("Blocked for resource %s, origin is
%s", resource, origin));
        } finally {
            if (entry != null) {
                entry.exit();
            }
            ContextUtil.exit();
        }
    }

    private static void initWhiteRules() {
        AuthorityRule rule = new AuthorityRule();
        rule.setResource(RESOURCE_NAME);
        rule.setStrategy(RuleConstant.AUTHORITY_WHITE);
        rule.setLimitApp("appA,appE");
        AuthorityRuleManager.loadRules(Collections.singletonList(rule));
    }
```

```
    private static void initBlackRules() {
        AuthorityRule rule = new AuthorityRule();
        rule.setResource(RESOURCE_NAME);
        rule.setStrategy(RuleConstant.AUTHORITY_BLACK);
        rule.setLimitApp("appA,appB");
        AuthorityRuleManager.loadRules(Collections.singletonList(rule));
    }
}
```

控制台配置方式：

3.2.5 热点规则

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。

Sentinel 利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控。热点参数限流支持集群模式。

基本使用

要使用热点参数限流功能，需要引入以下依赖：

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-parameter-flow-control</artifactId>
    <version>x.y.z</version>
</dependency>
```

然后为对应的资源配置热点参数限流规则，并在 `entry` 的时候传入相应的参数，即可使热点参数限流生效。

> 注：若自行扩展并注册了自己实现的 `SlotChainBuilder`，并希望使用热点参数限流功能，则可以在 chain 里面合适的地方插入 `ParamFlowSlot`。

那么如何传入对应的参数以便 Sentinel 统计呢？我们可以通过 `SphU` 类里面几个 `entry` 重载方法来传入：

```
public static Entry entry(String name, EntryType type, int count, Object... args)
throws BlockException

public static Entry entry(Method method, EntryType type, int count, Object...
args) throws BlockException
```

其中最后的一串 `args` 就是要传入的参数，有多个就按照次序依次传入。比如要传入两个参数 `paramA` 和 `paramB`，则可以：

```
// paramA in index 0, paramB in index 1.
// 若需要配置例外项或者使用集群维度流控，则传入的参数只支持基本类型。
SphU.entry(resourceName, EntryType.IN, 1, paramA, paramB);
```

注意：若 entry 的时候传入了热点参数，那么 exit 的时候也一定要带上对应的参数（`exit(count, args)`），否则可能会有统计错误。正确的示例：

```
Entry entry = null;
try {
    entry = SphU.entry(resourceName, EntryType.IN, 1, paramA, paramB);
    // Your logic here.
} catch (BlockException ex) {
    // Handle request rejection.
} finally {
    if (entry != null) {
        entry.exit(1, paramA, paramB);
    }
}
```

对于 `@SentinelResource` 注解方式定义的资源，若注解作用的方法上有参数，Sentinel 会将它们作为参数传入 `SphU.entry(res, args)`。比如以下的方法里面 `uid` 和 `type` 会分别作为第一个和第二个参数传入 Sentinel API，从而可以用于热点规则判断：

```
@SentinelResource("myMethod")
public Result doSomething(String uid, int type) {
  // some logic here...
}
```

热点参数规则

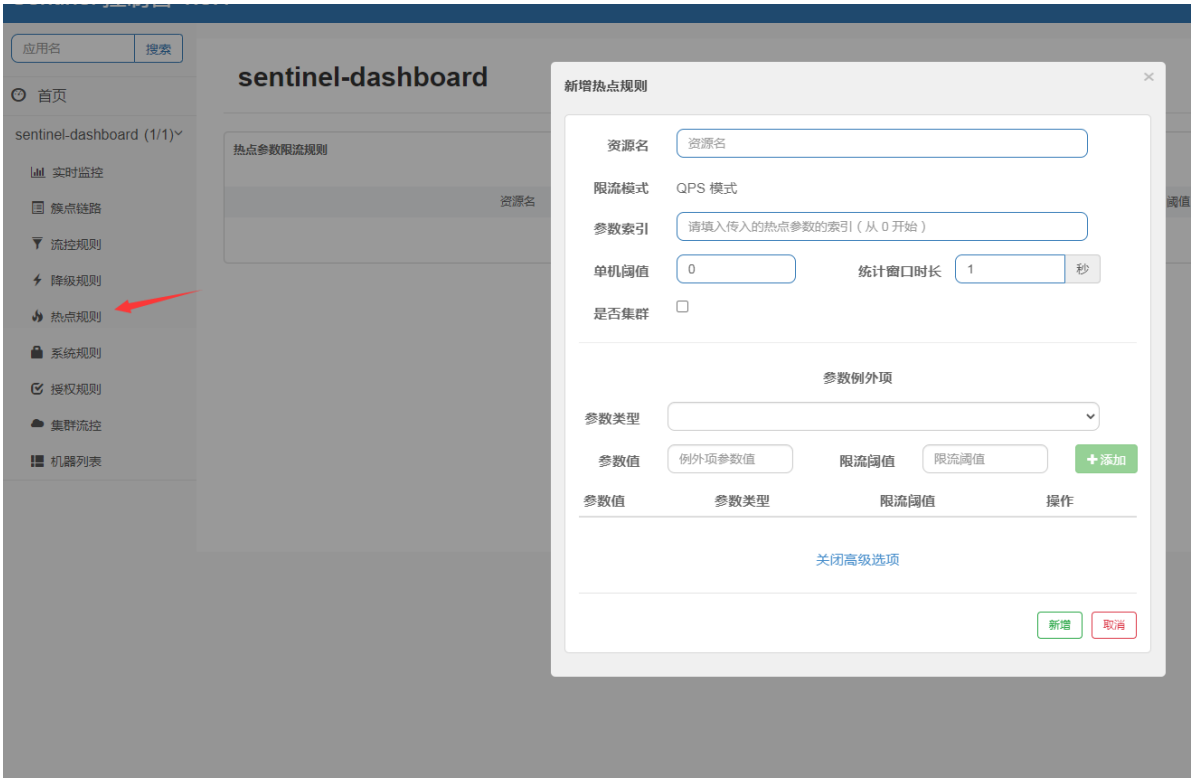
热点参数规则（`ParamFlowRule`）类似于流量控制规则（`FlowRule`）：

属性	说明	默认值
resource	资源名，必填	
count	限流阈值，必填	
grade	限流模式	QPS 模式
durationInSec	统计窗口时间长度（单位为秒），1.6.0 版本开始支持	1s
controlBehavior	流控效果（支持快速失败和匀速排队模式），1.6.0 版本开始支持	快速失败
maxQueueingTimeMs	最大排队等待时长（仅在匀速排队模式生效），1.6.0 版本开始支持	0ms
paramIdx	热点参数的索引，必填，对应 `SphU.entry(xxx, args)` 中的参数索引位置	
paramFlowItemList	参数例外项，可以针对指定的参数值单独设置限流阈值，不受前面 `count` 阈值的限制。**仅支持基本类型和字符串类型**	
clusterMode	是否是集群参数流控规则	`false`
clusterConfig	集群流控相关配置	

我们可以通过 `ParamFlowRuleManager` 的 `loadRules` 方法更新热点参数规则，下面是一个示例：

```
ParamFlowRule rule = new ParamFlowRule(resourceName)
    .setParamIdx(0)
    .setCount(5);
// 针对 int 类型的参数 PARAM_B，单独设置限流 QPS 阈值为 10，而不是全局的阈值 5.
ParamFlowItem item = new ParamFlowItem().setObject(String.valueOf(PARAM_B))
    .setClassType(int.class.getName())
    .setCount(10);
rule.setParamFlowItemList(Collections.singletonList(item));

ParamFlowRuleManager.loadRules(Collections.singletonList(rule));
```

控制台配置方式：



4 Spring Cloud整合Sentinel

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。

第一步：新建sentinel-springcloud子模块项目，pom.xml添加`spring-cloud-starter-alibaba-sentinel`依赖

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

第二步：新建SentinelHelloWorldControllerc测试类

```
package com.java1234.controller;


import com.alibaba.csp.sentinel.annotation.SentinelResource;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


/**
 * @author java1234_小锋
 * @site www.java1234.com
 * @company Java知识分享网
 * @create 2021-05-04 14:35
 */
```

```
@RestController
public class SentinelHelloWorldController {


    /**
     * 注解方式定义资源
     * @SentinelResource  value 资源名称
     * @SentinelResource blockHandler 调用被限流/降级/系统保护的时候调用的方法
     * @return
     */
    @SentinelResource(value = "helloWorld_springcloud",blockHandler =
"blockHandlerForHelloWorld3")
    @RequestMapping("helloWorld3")
    public String helloWorld3(){
        return "Sentinel 大爷你好! by 注解方式
@SentinelResource"+System.currentTimeMillis();
    }

    /**
     * 原方法调用被限流/降级/系统保护的时候调用
     * @param ex
     * @return
     */
    public String blockHandlerForHelloWorld3(BlockException ex) {
        ex.printStackTrace();
        return "系统繁忙，请稍后！";
    }


}
```

第三步：`application.yml`配置本地项目接入Dashboard控制台

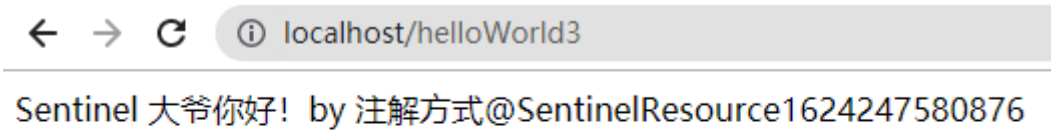
```
spring:
  application:
    name: sentinel_springcloud  # 设置应用名称
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080  # 设置Sentinel连接控制台的主机地址和端口
```

第四步：Sentinel Dashboard增加流控规则

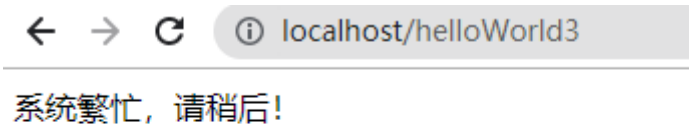
第五步：测试

浏览器地址栏输入：http://localhost/helloWorld3

正常显示：



如果频繁访问，则被流控



5 sentinel整合nacos实现配置持久化

细心的伙伴将会发现，当sentinel重新启动时，sentinel dashboard中原来的数据将会全部消失，这样就需要重新定义限流规则，无疑是不可取的。sentinel默认是把配置放内存里的。

我们可以将sentinel中定义的限流规则保存到Nacos配置中心里面，实现持久化。

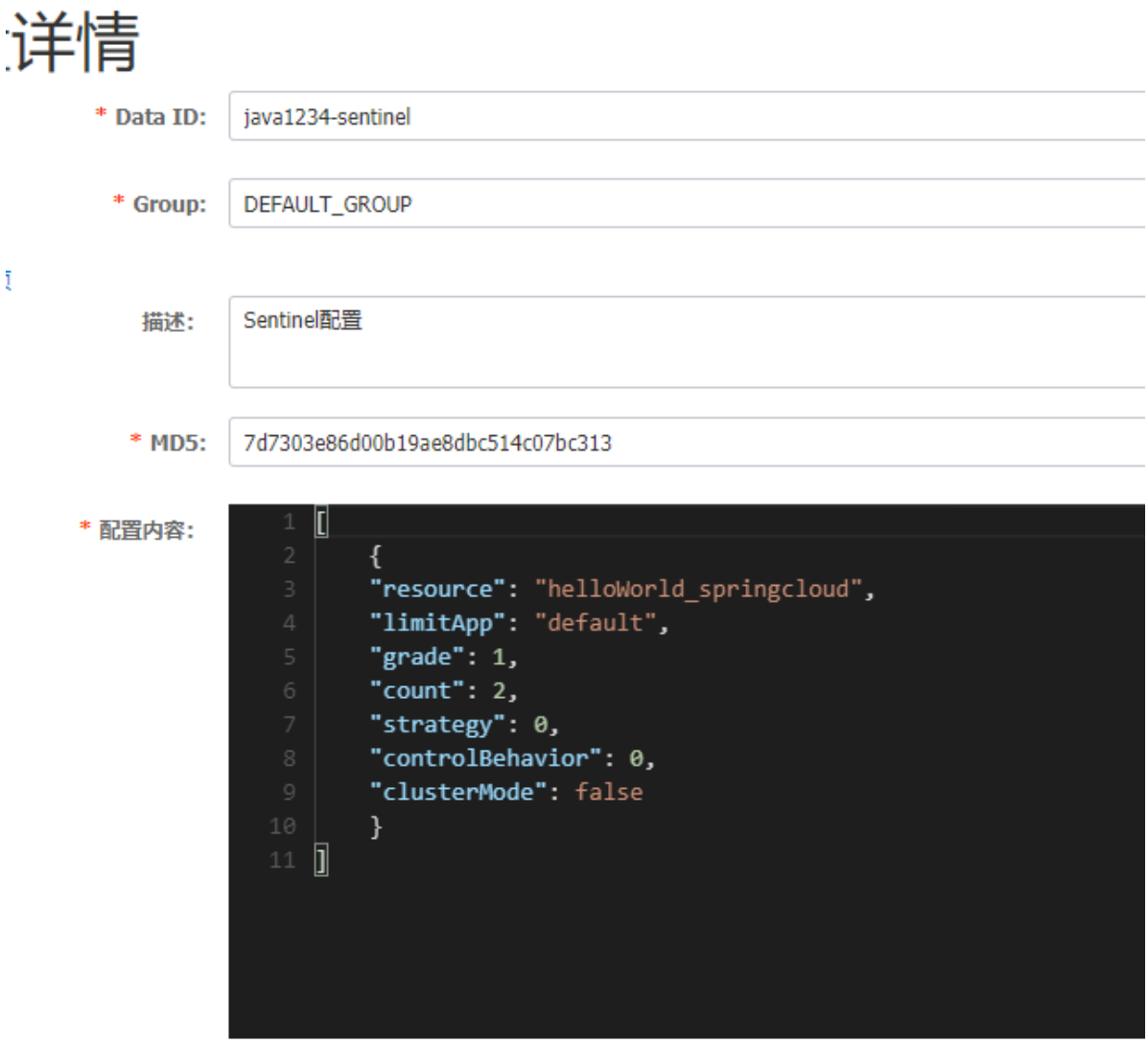
nacos视频教程地址：http://www.java1234.vip/course/151

具体实现步骤：

1，添加sentinel的nacos支持

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

2，在nacos中定义策略




```
[
    {
        "resource": "helloWorld_springcloud",
        "limitApp": "default",
        "grade": 1,
        "count": 2,
        "strategy": 0,
        "controlBehavior": 0,
        "clusterMode": false
    }
]
```

Data Id :和配置中的data_id相对应
Group:配置文件中的group-id
配置内容：
resource：接口名，即限流规则的作用对象
limitApp：流控针对的调用来源，若为 default 则不区分调用来源
grade：限流阈值类型（QPS 或并发线程数）；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
count：限流阈值
strategy：调用关系限流策略（ 直接、链路、关联 ）
controlBehavior：流量控制效果（直接拒绝、Warm Up、匀速排队）
clusterMode：是否为集群模式

3，项目配置，sentinel下面的dataSource中配置nacos

```yaml
spring:
  application:
    name: sentinel_springcloud  # 设置应用名称
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080  # 设置Sentinel连接控制台的主机地址和端口
      datasource:
        ds:
          nacos:
            server-addr: localhost:8848  # nacos连接地址
            group-id: DEFAULT_GROUP  # nacos连接的分组
            rule-type: flow  # 流控规则  rule-type 配置表示该数据源中的规则属于哪种类型的规则(flow，degrade，authority，system，param-flow，gw-flow，gw-api-group)
            data-id: java1234-sentinel  # 读取配置文件的 data-id
            data-type: json #  读取培训文件类型为json
```

4，测试

我们启动项目，发现sentinel控制台读取到了nacos里的配置，频繁刷新也能流控；