Sentinel

1. 功能描述

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 以流量为切入点，从流量 控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。拉勾教育系统引入主要是应用于秒杀时，下单接口的限流操作。防止流量过大，导致系统出问题。

2. 项目引入 Sentinel

a.添加 pom

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-alibaba-sentinel</artifactId>
  <version>0.2.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>sentinel-core</artifactId>
  <version>1.4.1</version>
</dependency>
<dependency>
   <groupId>com.alibaba.csp</groupId>
   <artifactId>sentinel-transport-simple-http</artifactId>
   <version>1.4.1</version>
   <exclusions>
      <exclusion>
        <artifactId>fastjson</artifactId>
        <groupId>com.alibaba</groupId>
      </exclusion>
   </exclusions>
</dependency>
<dependency>
   <groupId>com.alibaba.csp</groupId>
   <artifactId>sentinel-annotation-aspectj</artifactId>
   <version>1.4.1</version>
</dependency>
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-zookeeper</artifactId>
    <version>1.4.1</version>
    <exclusions>
      <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
      </exclusion>
```

```
        <exclusion>
          <groupId>log4j</groupId>
          <artifactId>log4j</artifactId>
        </exclusion>
      </exclusions>
</dependency>
```

b.@SentinelResource 注解切面类配置

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

c.异常拦截工具类

```
@Slf4j
public final class ExceptionUtil {
  public static ResponseDTO<String> testHandleException(BlockException ex) {
            log.error("testHandleException :{}", ex.getClass().getCanonicalName(),ex);
        return ResponseDTO.ofError(ResultCode.FLOW_SENTINEL_ERROR.getState(),
ResultCode.FLOW_SENTINEL_ERROR.getMessage());
    }
}
```

d.方法限流实例

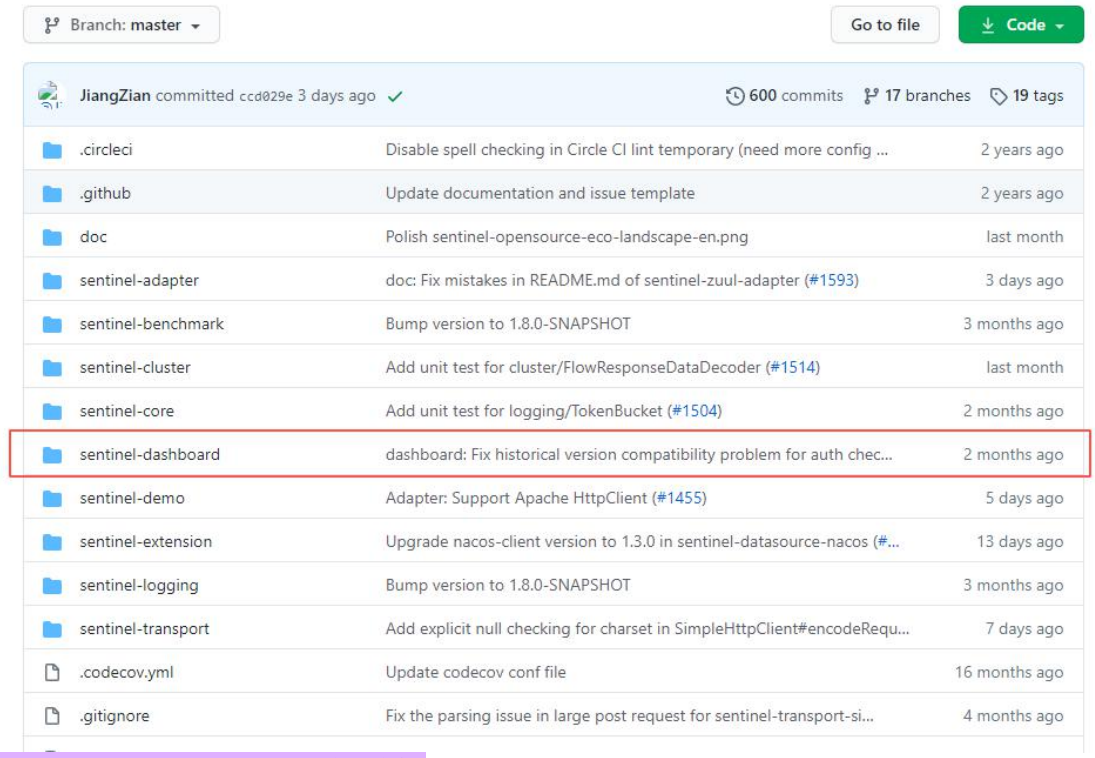
```
    @GetMapping("/test")
        @SentinelResource(value = "test", blockHandler = "testHandleException",
blockHandlerClass = {ExceptionUtil.class})
    public ResponseDTO<String> test() {
            log.info("SentinelController - test");
        return ResponseDTO.success();
    }
```

注：@SentinelResource 注解的 blockHandler 属性的值，必须跟 ExceptionUtil 类中的方法名一直，并且 ExceptionUtil 的方法必须是 static 的，方法的返回结果必须要跟 @SentinelResource 注解的方法返回结果一直。

3. Sentinel 控制台部署

a.下载控制台代码

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



注：sentinel-dashboard 就是控制台

b.sentinel-dashboard 是个 spring boot 项目，打成 jar

c.启动 sentinel-dashboard

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

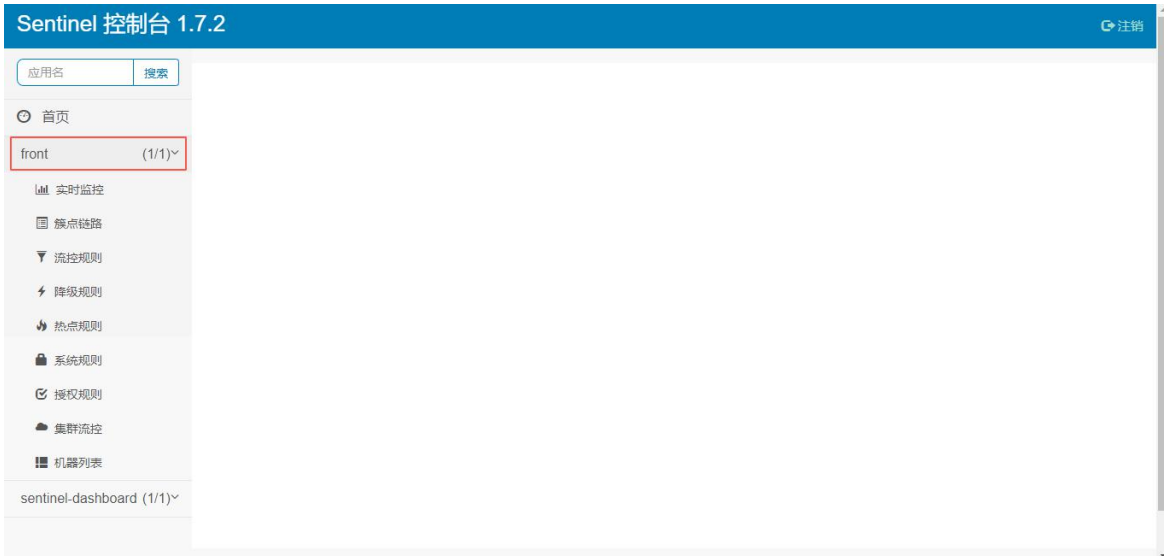
d.登录控制台

注：用户名&密码 默认都是 sentinel

注：进入首页可以看到有个 sentinel_dashboard 的项目，这个是我们自己的 sentinel_dashboard 服 务

4. 项目连接控制台

1. java -Dcsp.sentinel.dashboard.server=127.0.0.1:8080 -Dproject.name=front -jar XXXX
注：其中 XXX 为自己的项目名称，其中 127.0.0.1:8080 是控制台部署的机器 ip+端口



刷新控制台，可以看到多了一个 front 项目， 这个就是我们的服务。
注：当我们的服务启动完以后，必须访问一次限流接口，才能通过控制台看到我们的服务。

5. 规则数据持久化到 ZK

为什么要用 ZK 持久化？
上面的这种方式原理简单，一般用于入门测试使用，生产环境不能用。规则数据基于内存存储，在客户端重启后，所有规则都会丢失，需要重新配置。而且不适用于客户端多个实例，因为彼此之间不共享规则，倘若启动多个实例，需要多次重复配置。很明显，这不是我们想要的那种结果。

官方提供了三种模式，上面的“原始模式”、“pull 模式”、“push 模式”。pull 模式就是搞个文件存着，隔一会去请求一下，看看有没有变化，如果变了，就更新到内存，很明显这种模式存在延迟，也不建议上生产。官方给了三种示例推荐，Apollo、Nacos、Zookeeper，它们的使用类似，我们以 zookeeper 为例来看看怎么使用。

a.客户端项目（我们自己的服务）改造

```
1.      @Slf4j
2. @Component
3. public class DataSourceZookeeperInit {
4.
5.     @Value("${spring.application.name}")
6.     private String groupId;
7.     @Value("${zookeeper.address}")
8.     private String zookeeperAddress;
9.
10.     @PostConstruct
11.     public void init() throws Exception {
12.             final String path = "/sentinel_rule_config/" + groupId + "/front";
13.         log.info("sentinel 数据源初始化 DataSourceZookeeperInit
zookeeperAddress=={}",zookeeperAddress);
14.         ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new
ZookeeperDataSource<>(zookeeperAddress, path,
15.                 source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));
16.         FlowRuleManager.register2Property(flowRuleDataSource.getProperty());
17.     }
18. }
```

注：客户端修改获取规则的地方为从 zookeeper 获取规则。重新启动客户端后，就会变成从 zookeeper 的固定 path 里获取 rule 规则。

```
1. @Slf4j
2. @RestController
3. @RequestMapping("/sentinel")
4. @Api(tags = "sentinel 接口")
5. public class SentinelController {
6.
7.       @Value("${spring.application.name}")
8.     private String groupId;
9.     @Value("${zookeeper.address}")
10.     private String zookeeperAddress;
11.
12.     private static final int RETRY_TIMES = 3;
13.     private static final int SLEEP_TIME = 1000;
14.
15.
16.     @GetMapping("/test")
17.         @SentinelResource(value = "test", blockHandler = "testHandleException",
blockHandlerClass = {ExceptionUtil.class})
```

```
18.     public ResponseDTO<String> test() {
19.         log.info("SentinelController - test");
20.         return ResponseDTO.success();
21.     }
22.
23.
24.     @GetMapping("/addRule")
25.     public ResponseDTO<String> addRule() throws Exception{
26.         final String rule = "[\n"
27.                 + "  {\n"
28.                 + "    \"resource\": \"test\",\n"
29.                 + "    \"controlBehavior\": 0,\n"
30.                 + "    \"count\": 1.0,\n"
31.                 + "    \"grade\": 1,\n"
32.                 + "    \"limitApp\": \"default\",\n"
33.                 + "    \"strategy\": 0\n"
34.                 + "  }\n"
35.                 + "]";
36.         CuratorFramework zkClient = CuratorFrameworkFactory.newClient(zookeeperAddress,
new ExponentialBackoffRetry
37.                 (SLEEP_TIME, RETRY_TIMES));
38.         zkClient.start();
39.         String path = "/sentinel_rule_config/" + groupId + "front";
40.         Stat stat = zkClient.checkExists().forPath(path);
41.         if (stat == null) {
42.
zkClient.create().creatingParentContainersIfNeeded().withMode(CreateMode.PERSISTENT).for
Path(path, null);
43.         }
44.         zkClient.setData().forPath(path, rule.getBytes());
45.         try {
46.             Thread.sleep(3000);
47.         } catch (InterruptedException e) {
48.             e.printStackTrace();
49.         } finally {
50.                 zkClient.close();
51.                 }
52.
53.         return ResponseDTO.success();
54.     }
55.
56.
57.     @GetMapping("/delRule")
58.     public ResponseDTO<String> delRule() throws Exception{
59.         CuratorFramework zkClient = CuratorFrameworkFactory.newClient(zookeeperAddress,
new ExponentialBackoffRetry
60.                 (SLEEP_TIME, RETRY_TIMES));
61.         zkClient.start();
62.         String path = "/sentinel_rule_config/" + groupId + "front";
63.         Stat stat = zkClient.checkExists().forPath(path);
64.         if (stat != null) {
```

```
65.             zkClient.delete().deletingChildrenIfNeeded().forPath(path);
66.         }
67.         try {
68.             Thread.sleep(3000);
69.         } catch (InterruptedException e) {
70.             e.printStackTrace();
71.         } finally {
72.             zkClient.close();
73.             }
74.         return ResponseDTO.success();
75.     }
76. }
```

注：可以写个测试类，本地启动先测试下效果，其中 test 为需要限流的方法，addRule 为添加限流规则的方法，delRule 为删除规则的方法。

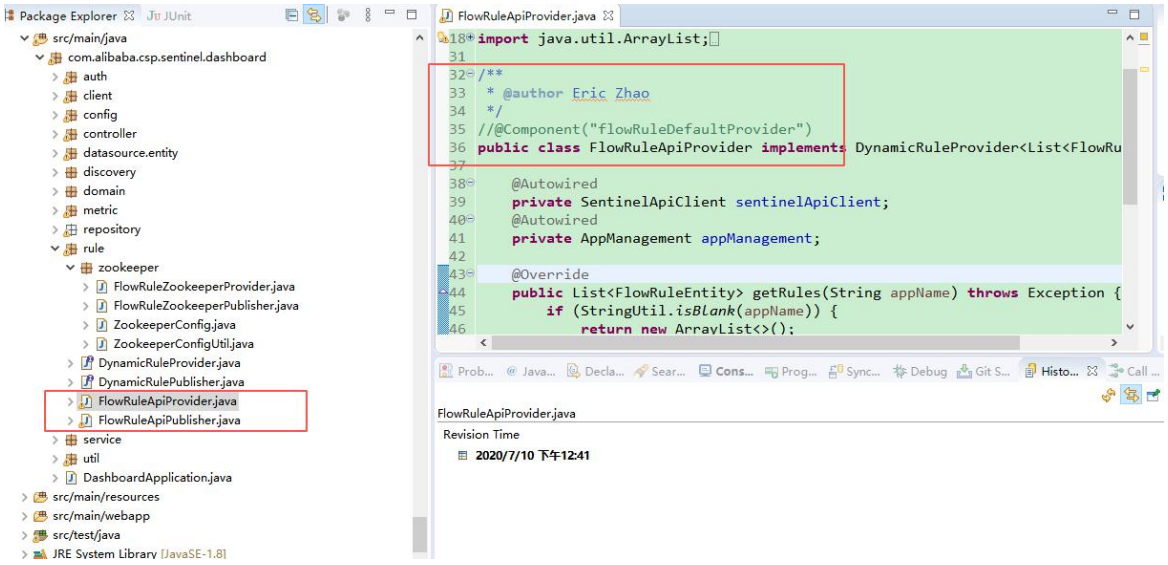
b.改造 sentinel_dashboard 控制台项目源码

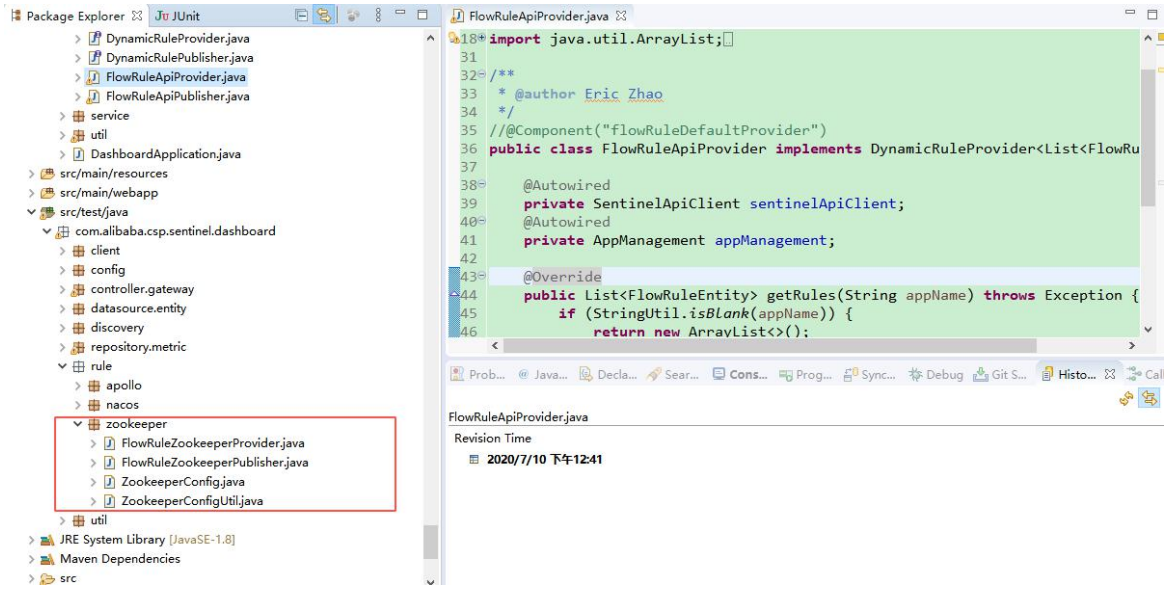
先修改一下 pom 文件，把 scope 注释掉。

sentinel-dashboard/pom.xml

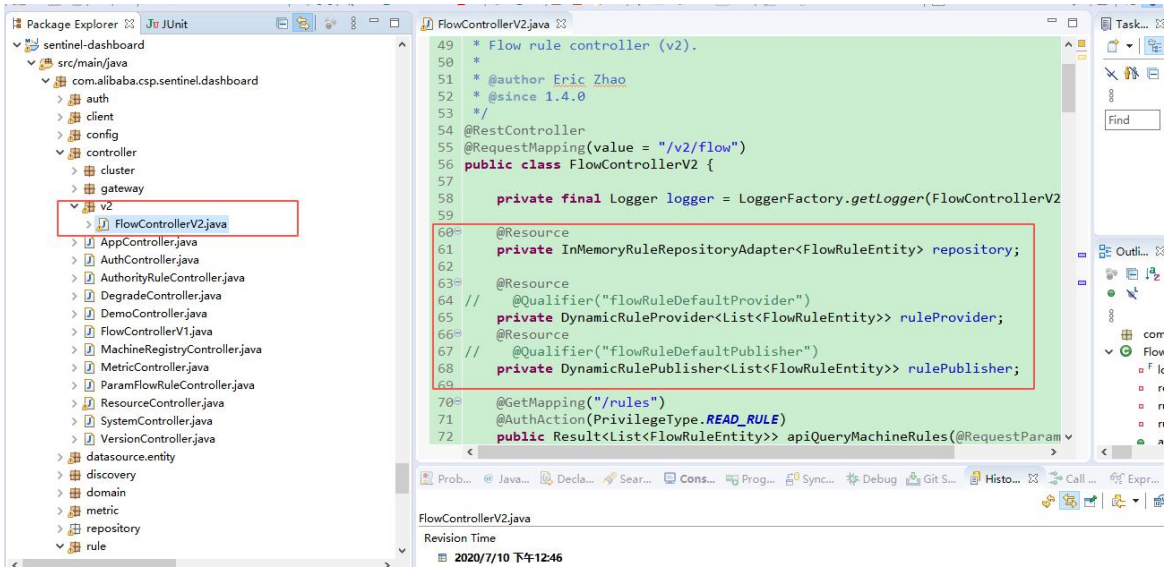
```
110         <dependency>
111             <groupId>com.ctrip.framework.apollo</groupId>
112             <artifactId>apollo-openapi</artifactId>
113             <version>1.2.0</version>
114             <scope>test</scope>
115         </dependency>
116 
117         <!--for Zookeeper rule publisher sample-->
118         <dependency>
119             <groupId>org.apache.curator</groupId>
120             <artifactId>curator-recipes</artifactId>
121             <version>${curator.version}</version>
122             <!--
123             <scope>test</scope> -->
124         </dependency>
125 
126         <dependency>
127             <groupId>junit</groupId>
128             <artifactId>junit</artifactId>
129             <scope>test</scope>
130         </dependency>
131         <dependency>
132             <groupId>org.mockito</groupId>
133             <artifactId>mockito-core</artifactId>
134             <scope>test</scope>
135         </dependency>
136         <dependency>
137             <groupId>com.github.stefanbirkner</groupId>
138             <artifactId>system-rules</artifactId>
```

找到 rule 包，添加个 zookeeper 文件夹，里面有 4 个类。
可以直接从工程的 test 测试代码里，直接把 zookeeper 包抄过去就行，并把 rule 下原来的 FlowRuleApiProvider 和 FlowRuleApiPublisher 给注释掉。

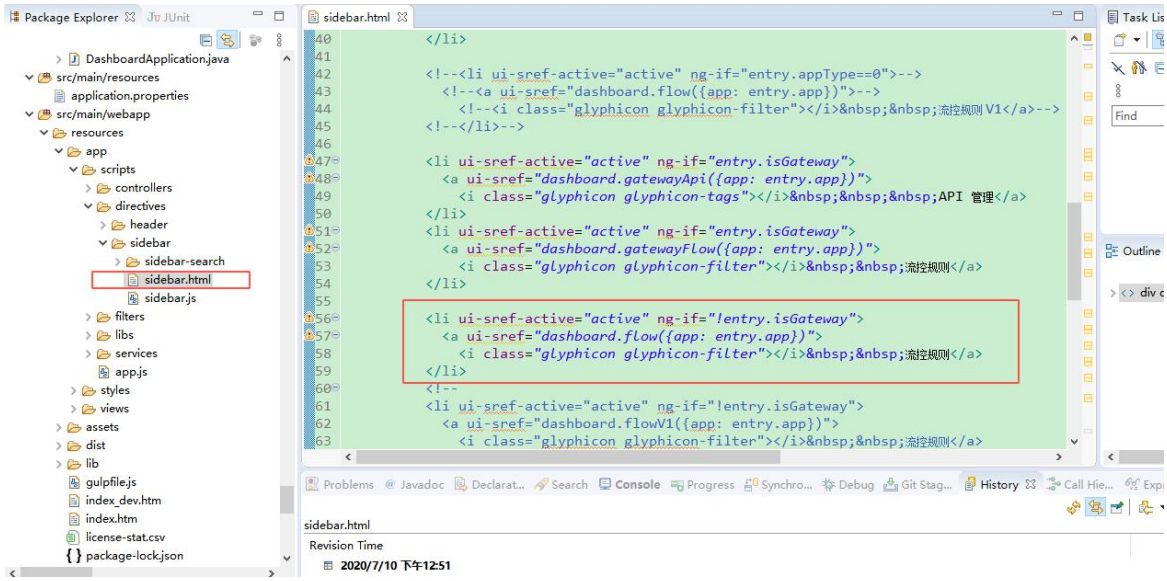




test 源码里已经提供了基于三种中间件的配置代码了，抄过去就行。
抄过去后，修改一下 RULE_ROOT_PATH，保持和客户端配置的是一致的。
之后找到 Controller 包下的 v2 包，如果你设置的 FlowRuleZookeeperProvider 和 publisher 两个 bean 有名字，可以在 autowired 时指定为你设置的名字，或者用@Resource。



最后修改一下 sidebar.html，将原来的 flowV1 改为如图



这样就 ok 了。

重新启动 dashboard 项目，重启客户端。这样 dashboard 就已经和 zookeeper 关联起来了，dashboard 的操作就由原来的操作客户端的 api，变成了操作 zookeeper。你所有在 dashboard 界面上做的配置，都会存储到 zookeeper 中，并实时推送到客户端。客户端重启后，dashboard 不受影响。这样就完成了多实例共享流控规则。