Guava接口限流

什么是限流

限流，当我们的系统被频繁的请求的时候，就有可能将系统压垮，所以为了解决这个问题，需要在每一个微服务中做限流操作，

生产场景下，还有可能要对并发访问较高的接口进行访问流量控制，防止过多的请求进入到后端服务器。对于限流的实现方式，我们之前已经接触过通过nginx限流，网关限流。但是他们都是对一个大的服务进行访问限流，如果现在只是要对某一个服务中的接口方法进行限流呢？这里推荐使用google提供的guava工具包中的RateLimiter进行实现，其内部是基于令牌桶算法进行限流计算

1.限流算法

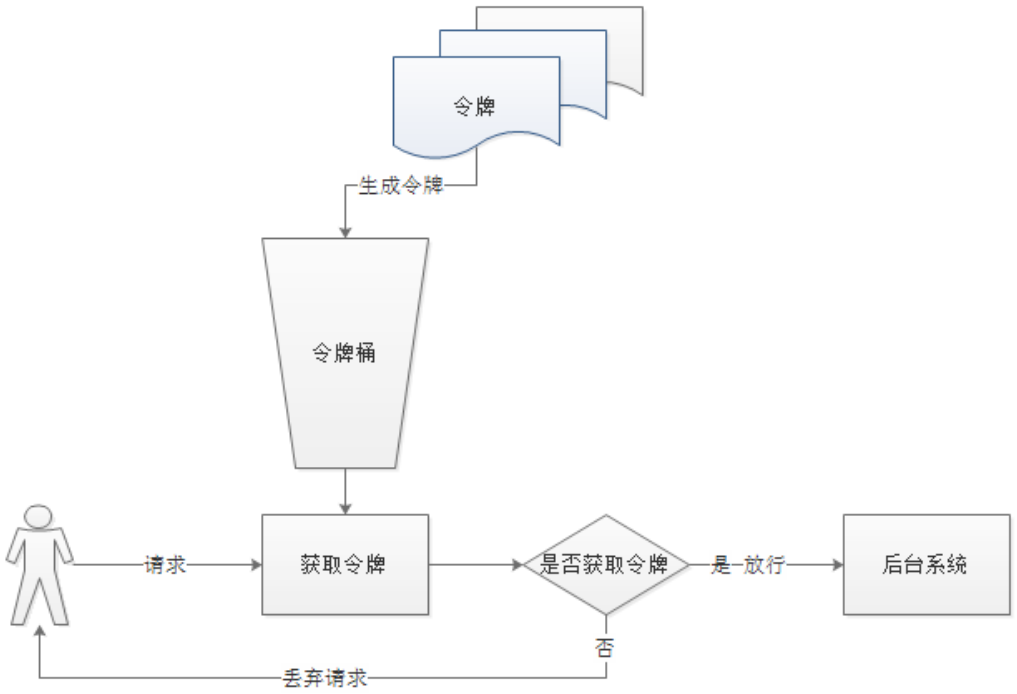
1.令牌桶限流

2.漏桶限流

3.计数器算法

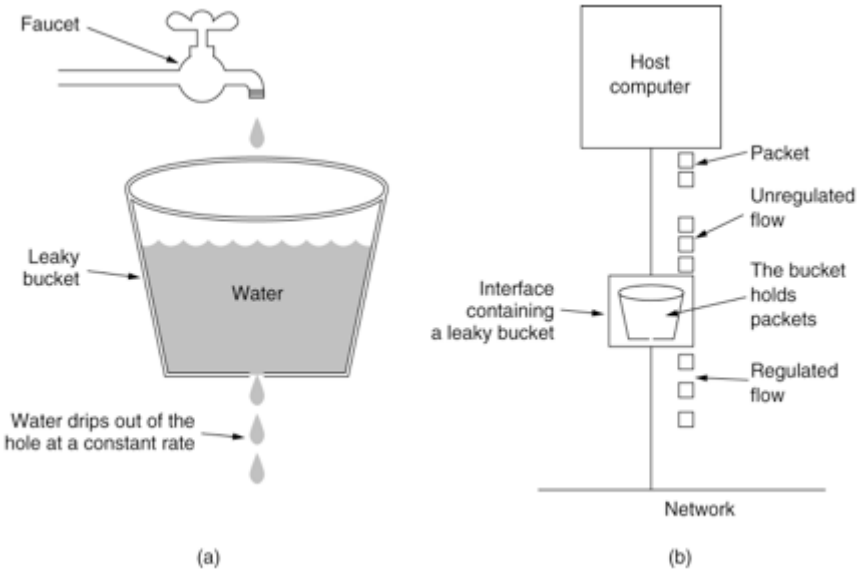
2 令牌桶算法

令牌桶算法是比较常见的限流算法之一，大概描述如下： 1）所有的请求在处理之前都需要拿到一个可用的令牌才会被处理； 2）根据限流大小，设置按照一定的速率往桶里添加令牌； 3）桶设置最大的放置令牌限制，当桶满时、新添加的令牌就被丢弃或者拒绝； 4）请求达到后首先要获取令牌桶中的令牌，拿着令牌才可以进行其他的业务逻辑，处理完业务逻辑之后，将令牌直接删除； 5）令牌桶有最低限额，当桶中的令牌达到最低限额的时候，请求处理完之后将不会删除令牌，以此保证足够的限流



这个算法的实现，有很多技术，Guava(读音: 瓜哇)是其中之一，redis客户端也有其实现。

3.漏桶算法



漏桶算法思路很简单，水（请求）先进入到漏桶里，漏桶以一定的速度出水，当水流入速度过大会直接溢出，可以看出漏桶算法能强行限制数据的传输速率。

1）添加依赖

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>28.0-jre</version>
</dependency>
```

2）自定义限流注解

```
@Inherited
@Documented
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface AccessLimit {}
```

3）自定义切面类

```
@Component
@Scope
@Aspect
public class AccessLimitAop {

    @Autowired
    private HttpServletResponse httpServletResponse;

    private RateLimiter rateLimiter = RateLimiter.create(20.0);

    @Pointcut("@annotation(com.changgou.webSecKill.aspect.AccessLimit)")
    public void limit(){}

    @Around("limit()")
```

```
    public Object around(ProceedingJoinPoint proceedingJoinPoint){

        boolean flag = rateLimiter.tryAcquire();
        Object obj = null;

        try{
            if (flag){
                obj=proceedingJoinPoint.proceed();
            }else{
                String errorMessage = JSON.toJSONString(new
Result(false,StatusCode.ERROR,"fail"));
                outMessage(httpServletResponse,errorMessage);
            }
        }catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        return obj;

    }

    private void outMessage(HttpServletResponse response, String errorMessage) {

        ServletOutputStream outputStream = null;
        try {
            response.setContentType("application/json;charset=UTF-8");
            outputStream = response.getOutputStream();
            outputStream.write(errorMessage.getBytes("UTF-8"));
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                outputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
}
```

4）使用自定义限流注解

```
/**
 * 秒杀下单
 * @param time 当前时间段
 * @param id 秒杀商品id
 * @param code
 * @return
 */
@GetMapping("/add")
@ResponseBody
@AccessLimit //添加自定义限流注解
public Result addOrder(String time ,Long id,String code){

    String cookieValue = this.readCookie();
    String redisCode = (String) redisTemplate.boundValueOps( key: "randomcode_"+cookieValue).get();
    if (!redisCode.equals(code)){
        return new Result( flag: false, StatusCode.ERROR, message: "下单失败");
    }
    return secKillOrderFeign.add(time, id);
}
```

网关限流

网关可以做很多的事情，比如，限流，当我们的系统被频繁的请求的时候，就有可能将系统压垮，所以为了解决这个问题，需要在每一个微服务中做限流操作，但是如果有了网关，那么就可以在网关系统做限流，因为所有的请求都需要先通过网关系统才能路由到微服务中。

3 网关限流代码实现

需求：每个ip地址1秒内只能发送1次请求，多出来的请求返回429错误。

代码实现：

（1）spring cloud gateway 默认使用redis的RateLimter限流算法来实现。所以我们要使用首先需要引入redis的依赖

```
<!--redis-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
    <version>2.1.3.RELEASE</version>
</dependency>
```

（2）定义KeyResolver

在GatewayApplicatioin引导类中添加如下代码，KeyResolver用于计算某一个类型的限流的KEY也就是说，可以通过KeyResolver来指定限流的Key。

```
    //定义一个KeyResolver
    @Bean
    public KeyResolver ipKeyResolver() {
        return new KeyResolver() {
            @Override
            public Mono<String> resolve(ServerWebExchange exchange) {
                return
Mono.just(exchange.getRequest().getRemoteAddress().getHostName());
            }
        };
    }
```

（3）修改application.yml中配置项，指定限制流量的配置以及REDIS的配置，修改后最终配置如下：

```
spring:
  application:
    name: sysgateway
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '[/**]': # 匹配所有请求
            allowedOrigins: "*" #跨域处理 允许所有的域
            allowedMethods: # 支持的方法
            - GET
            - POST
            - PUT
            - DELETE
      routes:
```

```
      - id: goods
        uri: lb://goods
        predicates:
        - Path=/goods/**
        filters:
        - StripPrefix= 1
        - name: RequestRateLimiter #请求数限流 名字不能随便写
          args:
            key-resolver: "#{@ipKeyResolver}"
            redis-rate-limiter.replenishRate: 1 #令牌桶每秒填充平均速率
            redis-rate-limiter.burstCapacity: 1 #令牌桶总容量
      - id: system
        uri: lb://system
        predicates:
        - Path=/system/**
        filters:
        - StripPrefix= 1
  # 配置Redis 127.0.0.1可以省略配置
  redis:
    host: 192.168.200.128
    port: 6379
server:
  port: 9101
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:6868/eureka
  instance:
    prefer-ip-address: true
```

解释：

- burstCapacity：令牌桶总容量。
- replenishRate：令牌桶每秒填充平均速率。
- key-resolver：用于限流的键的解析器的 Bean 对象的名字。它使用 SpEL 表达式根据#{@beanName}从 Spring 容器中获取 Bean 对象。

通过在`replenishRate`和中设置相同的值来实现稳定的速率`burstCapacity`。设置`burstCapacity`高于时，可以允许临时突发`replenishRate`。在这种情况下，需要在突发之间允许速率限制器一段时间（根据`replenishRate`），因为2次连续突发将导致请求被丢弃（`HTTP 429 - Too Many Requests`）

key-resolver: "#{@userKeyResolver}" 用于通过SPEL表达式来指定使用哪一个KeyResolver.

如上配置：

表示 一秒内，允许 一个请求通过，令牌桶的填充速率也是一秒钟添加一个令牌。

最大突发状况 也只允许 一秒内有一次请求，可以根据业务来调整 。

（4）测试

启动redis

启动注册中心

启动商品微服务

启动gateway网关

打开浏览器 http://localhost:9101/goods/brand

快速刷新，当1秒内发送多次请求，就会返回429错误。