# SpringBoot 如何进行限流

大家好，我是飘渺。SpringBoot老鸟系列的文章已经写了四篇，每篇的阅读反响都还不错，那今天继续给大家带来老鸟系列的第五篇，来聊聊在SpringBoot项目中如何对接口进行限流，有哪些常见的限流算法，如何优雅的进行限流。

首先就让我们来看看为什么需要对接口进行限流？

### **为什么要进行限流？**

因为互联网系统通常都要面对大并发大流量的请求，在突发情况下（最常见的场景就是秒杀、抢购），瞬时大流量会直接将系统打垮，无法对外提供服务。那为了防止出现这种情况最常见的解决方案之一就是限流，当请求达到一定的并发数或速率，就进行等待、排队、降级、拒绝服务等。

例如，12306购票系统，在面对高并发的情况下，就是采用了限流。在流量高峰期间经常会出现提示语；"当前排队人数较多，请稍后再试！"

### **什么是限流？有哪些限流算法？**

限流是对某一时间窗口内的请求数进行限制，保持系统的可用性和稳定性，防止因流量暴增而导致的系统运行缓慢或宕机。

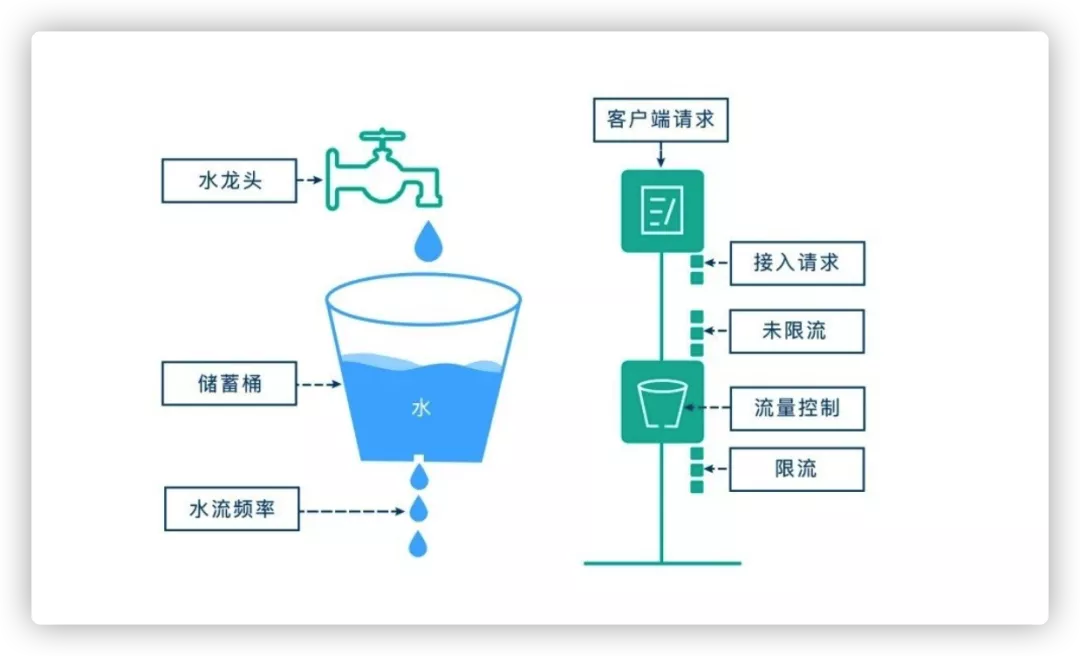
常见的限流算法有三种：

**1. 计数器限流**

计数器限流算法是最为简单粗暴的解决方案，主要用来限制总并发数，比如数据库连接池大小、线程池大小、接口访问并发数等都是使用计数器算法。

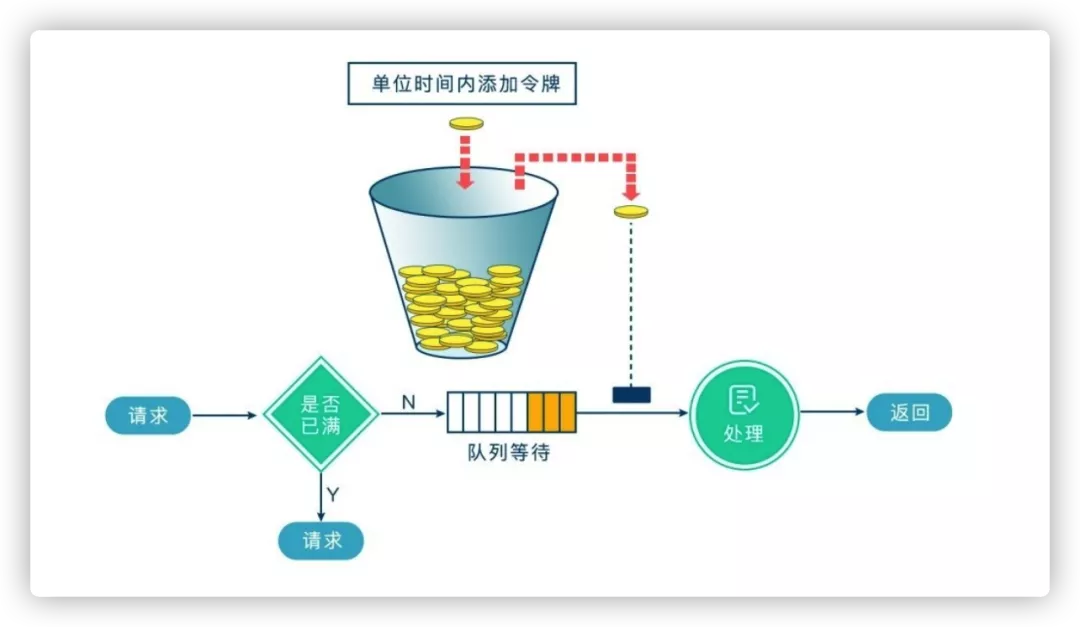
如：使用 AomicInteger 来进行统计当前正在并发执行的次数，如果超过域值就直接拒绝请求，提示系统繁忙。

**2. 漏桶算法**

漏桶算法

漏桶算法思路很简单，我们把水比作是请求，漏桶比作是系统处理能力极限，水先进入到漏桶里，漏桶里的水按一定速率流出，当流出的速率小于流入的速率时，由于漏桶容量有限，后续进入的水直接溢出（拒绝请求），以此实现限流。

**3. 令牌桶算法**

令牌桶算法

令牌桶算法的原理也比较简单，我们可以理解成医院的挂号看病，只有拿到号以后才可以进行诊病。

系统会维护一个令牌（token）桶，以一个恒定的速度往桶里放入令牌（token），这时如果有请求进来想要被处理，则需要先从桶里获取一个令牌（token），当桶里没有令牌（token）可取时，则该请求将被拒绝服务。令牌桶算法通过控制桶的容量、发放令牌的速率，来达到对请求的限制。

### **基于Guava工具类实现限流**

Google开源工具包Guava提供了限流工具类RateLimiter，该类基于令牌桶算法实现流量限制，使用十分方便，而且十分高效，实现步骤如下：

#### **第一步：引入guava依赖包**

<dependency>  
    <groupId>com.google.guava</groupId>  
    <artifactId>guava</artifactId>  
    <version>30.1-jre</version>  
</dependency>

#### **第二步：给接口加上限流逻辑**

@Slf4j  
@RestController  
@RequestMapping("/limit")  
public class LimitController {  
    /\*\*  
     \* 限流策略 ：1秒钟2个请求  
     \*/  
    private final RateLimiter limiter = RateLimiter.create(2.0);  
  
    private DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  
  
    @GetMapping("/test1")  
    public String testLimiter() {  
        //500毫秒内，没拿到令牌，就直接进入服务降级  
        boolean tryAcquire = limiter.tryAcquire(500, TimeUnit.MILLISECONDS);  
  
        if (!tryAcquire) {  
            log.warn("进入服务降级，时间{}", LocalDateTime.now().format(dtf));  
            return "当前排队人数较多，请稍后再试！";  
        }  
  
        log.info("获取令牌成功，时间{}", LocalDateTime.now().format(dtf));  
        return "请求成功";  
    }  
}

以上用到了RateLimiter的2个核心方法：create()、tryAcquire()，以下为详细说明

* acquire() 获取一个令牌, 改方法会阻塞直到获取到这一个令牌, 返回值为获取到这个令牌花费的时间
* acquire(int permits) 获取指定数量的令牌, 该方法也会阻塞, 返回值为获取到这 N 个令牌花费的时间
* tryAcquire() 判断时候能获取到令牌, 如果不能获取立即返回 false
* tryAcquire(int permits) 获取指定数量的令牌, 如果不能获取立即返回 false
* tryAcquire(long timeout, TimeUnit unit) 判断能否在指定时间内获取到令牌, 如果不能获取立即返回 false
* tryAcquire(int permits, long timeout, TimeUnit unit) 同上

#### **第三步：体验效果**

通过访问测试地址：http://127.0.0.1:8080/limit/test1，反复刷新并观察后端日志

WARN  LimitController:35 - 进入服务降级，时间2021-09-25 21:39:37  
WARN  LimitController:35 - 进入服务降级，时间2021-09-25 21:39:37  
INFO  LimitController:39 - 获取令牌成功，时间2021-09-25 21:39:37  
WARN  LimitController:35 - 进入服务降级，时间2021-09-25 21:39:37  
WARN  LimitController:35 - 进入服务降级，时间2021-09-25 21:39:37  
INFO  LimitController:39 - 获取令牌成功，时间2021-09-25 21:39:37  
  
WARN  LimitController:35 - 进入服务降级，时间2021-09-25 21:39:38  
INFO  LimitController:39 - 获取令牌成功，时间2021-09-25 21:39:38  
WARN  LimitController:35 - 进入服务降级，时间2021-09-25 21:39:38  
INFO  LimitController:39 - 获取令牌成功，时间2021-09-25 21:39:38

从以上日志可以看出，1秒钟内只有2次成功，其他都失败降级了，说明我们已经成功给接口加上了限流功能。

当然了，我们在实际开发中并不能直接这样用。至于原因嘛，你想呀，你每个接口都需要手动给其加上tryAcquire()，业务代码和限流代码混在一起，而且明显违背了DRY原则，代码冗余，重复劳动。代码评审时肯定会被老鸟们给嘲笑一番，啥破玩意儿！

所以，我们这里需要想办法将其优化 - 借助自定义注解+AOP实现接口限流。

### **基于AOP实现接口限流**

基于AOP的实现方式也非常简单，实现过程如下：

#### **第一步：加入AOP依赖**

<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-aop</artifactId>  
</dependency>

#### **第二步：自定义限流注解**

@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.METHOD})  
@Documented  
public @interface Limit {  
    */\*\*  
     \* 资源的key,唯一  
     \* 作用：不同的接口，不同的流量控制  
     \*/*  
    String key() default "";  
  
    */\*\*  
     \* 最多的访问限制次数  
     \*/*  
    double permitsPerSecond () ;  
  
    */\*\*  
     \* 获取令牌最大等待时间  
     \*/*  
    long timeout();  
  
    */\*\*  
     \* 获取令牌最大等待时间,单位(例:分钟/秒/毫秒) 默认:毫秒  
     \*/*  
    TimeUnit timeunit() default TimeUnit.MILLISECONDS;  
  
    */\*\*  
     \* 得不到令牌的提示语  
     \*/*  
    String msg() default "系统繁忙,请稍后再试.";  
}

#### **第三步：使用AOP切面拦截限流注解**

@Slf4j  
@Aspect  
@Component  
public class LimitAop {  
    */\*\*  
     \* 不同的接口，不同的流量控制  
     \* map的key为 Limiter.key  
     \*/*  
    private final Map<String, RateLimiter> limitMap = Maps.newConcurrentMap();  
  
    @Around("@annotation(com.jianzh5.blog.limit.Limit)")  
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable{  
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();  
        Method method = signature.getMethod();  
        *//拿limit的注解*  
        Limit limit = method.getAnnotation(Limit.class);  
        if (limit != null) {  
            *//key作用：不同的接口，不同的流量控制*  
            String key=limit.key();  
            RateLimiter rateLimiter = null;  
            *//验证缓存是否有命中key*  
            if (!limitMap.containsKey(key)) {  
                *// 创建令牌桶*  
                rateLimiter = RateLimiter.create(limit.permitsPerSecond());  
                limitMap.put(key, rateLimiter);  
                log.info("新建了令牌桶={}，容量={}",key,limit.permitsPerSecond());  
            }  
            rateLimiter = limitMap.get(key);  
            *// 拿令牌*  
            boolean acquire = rateLimiter.tryAcquire(limit.timeout(), limit.timeunit());  
            *// 拿不到命令，直接返回异常提示*  
            if (!acquire) {  
                log.debug("令牌桶={}，获取令牌失败",key);  
                this.responseFail(limit.msg());  
                return null;  
            }  
        }  
        return joinPoint.proceed();  
    }  
  
    */\*\*  
     \* 直接向前端抛出异常  
     \* @param msg 提示信息  
     \*/*  
    private void responseFail(String msg)  {  
        HttpServletResponse response=((ServletRequestAttributes) RequestContextHolder.getRequestAttributes()).getResponse();  
        ResultData<Object> resultData = ResultData.fail(ReturnCode.LIMIT\_ERROR.getCode(), msg);  
        WebUtils.writeJson(response,resultData);  
    }  
}

#### **第四步：给需要限流的接口加上注解**

@Slf4j  
@RestController  
@RequestMapping("/limit")  
public class LimitController {  
      
    @GetMapping("/test2")  
    @Limit(key = "limit2", permitsPerSecond = 1, timeout = 500, timeunit = TimeUnit.MILLISECONDS,msg = "当前排队人数较多，请稍后再试！")  
    public String limit2() {  
        log.info("令牌桶limit2获取令牌成功");  
        return "ok";  
    }  
  
  
    @GetMapping("/test3")  
    @Limit(key = "limit3", permitsPerSecond = 2, timeout = 500, timeunit = TimeUnit.MILLISECONDS,msg = "系统繁忙，请稍后再试！")  
    public String limit3() {  
        log.info("令牌桶limit3获取令牌成功");  
        return "ok";  
    }  
}

### **第五步：体验效果**

通过访问测试地址：http://127.0.0.1:8080/limit/test2，反复刷新并观察输出结果：

正常响应时：

{"status":100,"message":"操作成功","data":"ok","timestamp":1632579377104}

触发限流时：

{"status":2001,"message":"系统繁忙，请稍后再试！","data":null,"timestamp":1632579332177}

通过观察得之，基于自定义注解同样实现了接口限流的效果。

### **小结**

一般在系统上线时我们通过对系统压测可以评估出系统的性能阀值，然后给接口加上合理的限流参数，防止出现大流量请求时直接压垮系统。今天我们介绍了几种常见的限流算法（重点关注令牌桶算法），基于Guava工具类实现了接口限流并利用AOP完成了对限流代码的优化。

哇撒！这几个SpringBoot前后端分离项目（附源码），star过千，快去收藏夹吃灰吧。。。