<https://mp.weixin.qq.com/s?__biz=MzI4NjI2OTYwNg==&mid=2247484703&idx=1&sn=43fa79d0a7df1d9f3414a536407ba19f&chksm=ebdeccd6dca945c00edaddde7db1e773269ba5814d9da76cea0794ccb80c5bf18a0730539b5c&mpshare=1&scene=1&srcid=&pass_ticket=O0U9gt%2Blc7Z5GJ9TuH2Jn9X5SM0%2FXP1nMRH6xpzNDjb%2FC6WpK%2FruKS9BLGt5pvfU#rd>

**为什么需要限流？**

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。限流可以认为服务降级的一种，限流通过限制请求的流量以达到保护系统的目的。

一般来说，系统的吞吐量是可以计算出一个阈值的，为了保证系统的稳定运行，一旦达到这个阈值，就需要限制流量并采取一些措施以完成限制流量的目的。比如：延迟处理，拒绝处理，或者部分拒绝处理等等。否则，很容易导致服务器的宕机。

**现有的方案**

Google的Guava工具包中就提供了一个限流工具类——RateLimiter，本文也是通过使用该工具类来实现限流功能。RateLimiter是基于“令牌通算法”来实现限流的。

**令牌桶算法**

令牌桶算法是一个存放固定容量令牌（token）的桶，按照固定速率往桶里添加令牌。令牌桶算法基本可以用下面的几个概念来描述：

假如用户配置的平均发送速率为r，则每隔1/r秒一个令牌被加入到桶中。

桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃或拒绝。

当一个n个字节大小的数据包到达，将从桶中删除n个令牌，接着数据包被发送到网络上。

如果桶中的令牌不足n个，则不会删除令牌，且该数据包将被限流（要么丢弃，要么缓冲区等待）。

**限流器实现**

**1.pom文件中引入Guava包**

*<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->*  
 <dependency>  
 <groupId>com.google.guava</groupId>  
 <artifactId>guava</artifactId>  
 <version>27.0.1-jre</version>  
 </dependency>

**2.自定义拦截器，并在拦截器中实现限流**

a）定义一个拦截器抽象类，用于多个拦截器复用，主要是继承HandlerInterceptorAdapter，重写preHandle方法；并提供preFilter抽象方法，供子类实现。

*/\*\*  
 \* @author linzhiqiang  
 \* @date 2019/4/17  
 \*/*  
public abstract **class** AbstractInterceptor extends HandlerInterceptorAdapter {  
 **private** Logger logger = LoggerFactory.getLogger(AbstractInterceptor.class);  
  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
 ResponseEnum result;  
 **try** {  
 result = preFilter(request);  
 } catch (Exception e) {  
 logger.error("preHandle catch a exception:" + e.getMessage());  
 result = ResponseEnum.FAIL;  
 }  
 **if** (ResponseEnum.SUCCESS.code.equals(result.code)) {  
 return true;  
 }  
 handlerResponse(result, response);  
 return false;  
 }  
  
  
 */\*\*  
 \* 自定义pre处理  
 \*  
 \* @param request  
 \* @return  
 \*/*  
 protected abstract ResponseEnum preFilter(HttpServletRequest request);  
  
 */\*\*  
 \* 错误处理事件  
 \*  
 \* @param result  
 \* @param response  
 \*/*  
 **private** void handlerResponse(ResponseEnum result, HttpServletResponse response) {  
 ResponseDto responseDto = **new** ResponseDto();  
 responseDto.setCode(result.code);  
 responseDto.setStatus(result.status);  
 responseDto.setMessage(result.message);  
 response.setStatus(HttpServletResponse.SC\_OK);  
 response.setContentType(MediaType.APPLICATION\_JSON\_UTF8\_VALUE);  
 PrintWriter printWriter = null;  
 **try** {  
 printWriter = response.getWriter();  
 printWriter.write(JsonUtils.**to**Json(responseDto));  
 } catch (Exception e) {  
 logger.error("handlerResponse catch a exception:" + e.getMessage());  
 } finally {  
 **if** (printWriter != null) {  
 printWriter.close();  
 }  
 }  
 }  
}

b）定义流量控制拦截器，流量控制拦截器继承自上面的拦截器抽象类，在preFilter方法中进行流量控制。

*/\*\*  
 \*  
 \* @author linzhiqiang  
 \* @date 2019/4/17  
 \*/*  
**@Component**("rateLimitInterceptor")  
public **class** **RateLimitInterceptor** **extends** **AbstractInterceptor** {  
 **private** **Logger** logger = **LoggerFactory**.getLogger(**RateLimitInterceptor**.**class**);  
  
 */\*\*  
 \* 单机全局限流器(限制QPS为250)  
 \*/*  
 **private** static **final** **RateLimiter** rateLimiter = **RateLimiter**.create(300);  
  
 public static void setRate(double limiterQPS){  
 rateLimiter.setRate(limiterQPS);  
 }  
 **@Override**  
 **protected** **ResponseEnum** preFilter(**HttpServletRequest** request) {  
 **if** (!rateLimiter.tryAcquire()) {  
 logger.warn("限流中......");  
 **return** **ResponseEnum**.**RATE\_LIMIT**;  
 }  
 **return** **ResponseEnum**.**SUCCESS**;  
 }  
}

使用Guava提供的RateLimiter类来实现流量控制，过程很简单：定义了一个QPS为1的全局限流器（便于测试），使用tryAcquire()方法来尝试获取令牌，如果成功则返回ResponseEnum.OK，否则返回ResponseEnum.RATE\_LIMIT。

**3.继承WebMvcConfigurerAdapter来添加自定义拦截器**

**@Configuration**  
public **class** **MyWebAppConfigurer** **extends** **WebMvcConfigurationSupport** {  
 **@Override**  
 public void addInterceptors(**InterceptorRegistry** registry) {  
 *// 多个拦截器组成一个拦截器链*  
 *// addPathPatterns 用于添加拦截规则*  
 *// excludePathPatterns 用户排除拦截*  
 registry.addInterceptor(**new** **RateLimitInterceptor**()).addPathPatterns("/\*\*");  
 **super**.addInterceptors(registry);  
 }  
}

**4.写一个Controller来提供一个简单的访问接口**

*/\*\*  
 \* request测试专用  
 \* @return  
 \*/*  
 @RequestMapping(value = "getUserList", method = RequestMethod.GET)  
 public String getUserList() {  
 String result = null;  
 **try** {  
 result = "请求成功";  
 }catch (Exception e){  
 logger.error("请求失败", e);  
 return JsonUtils.**to**Json(ResponseUtils.failInServer(result));  
 }  
 return JsonUtils.**to**Json(ResponseUtils.success(result));  
 }

上文使用到的ResponseEnum是一个返回Code的枚举：

*/\*\*  
 \* @description:  
 \* @Date : 2019/3/19 19:04  
 \* @Author : 樊康康-(kangkang.fan@mljr.com)  
 \*/*  
**public** **enum** **ResponseEnum** {  
  
 SUCCESS("200","000000","请求成功"),  
 FAIL("200","100000","请求失败"),  
 FAIL\_BY\_PARAMS("200","200000","请求参数异常"),  
 FAIL\_IN\_SERVER("200","300000","服务器内部异常"),  
 RATE\_LIMIT("200","400000","限流中");  
  
 **public** **String** status;  
 **public** **String** code;  
 **public** **String** message;  
  
 ResponseEnum(**String** s, **String** s1, **String** s2) {  
 this.status = s;  
 this.code = s1;  
 this.message = s2;  
 }  
}

**5.使用Postman来测试接口**

快速并且反复的调用接口，可以很容易的看到两种结果。

反复调用时，Console输出如下：

