## 为什么要互联网项目要限流

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。缓存的目的是提升系统访问速度和增大系统能处理的容量，可谓是抗高并发流量的银弹；而降级是当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开；而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页），因此需有一种手段来限制这些场景的并发/请求量，即限流。

## 互联网雪崩效应解决方案

服务降级: 在高并发的情况， 防止用户一直等待，直接返回一个友好的错误提示给客户端。

服务熔断：在高并发的情况，一旦达到服务最大的承受极限，直接拒绝访问，使用服务降级。

服务隔离: 使用服务隔离方式解决服务雪崩效应

服务限流: 在高并发的情况，一旦服务承受不了使用服务限流机制（计时器（滑动窗口计数）、漏桶算法、令牌桶（Restlimite））

## 高并发限流解决方案

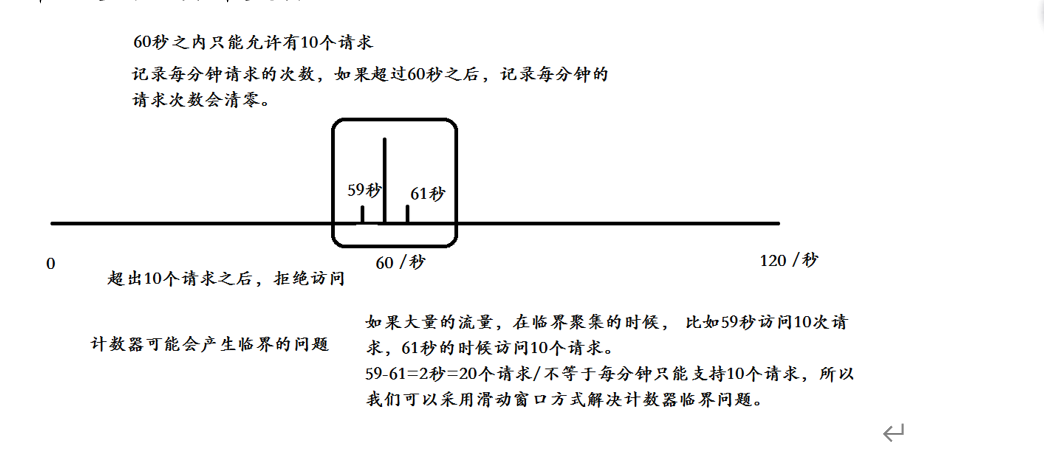
高并发限流解决方案限流算法(令牌桶、漏桶、计数器)、应用层解决限流（Nginx）

### 限流算法

常见的限流算法有：令牌桶、漏桶。计数器也可以进行粗暴限流实现。

#### 计数器

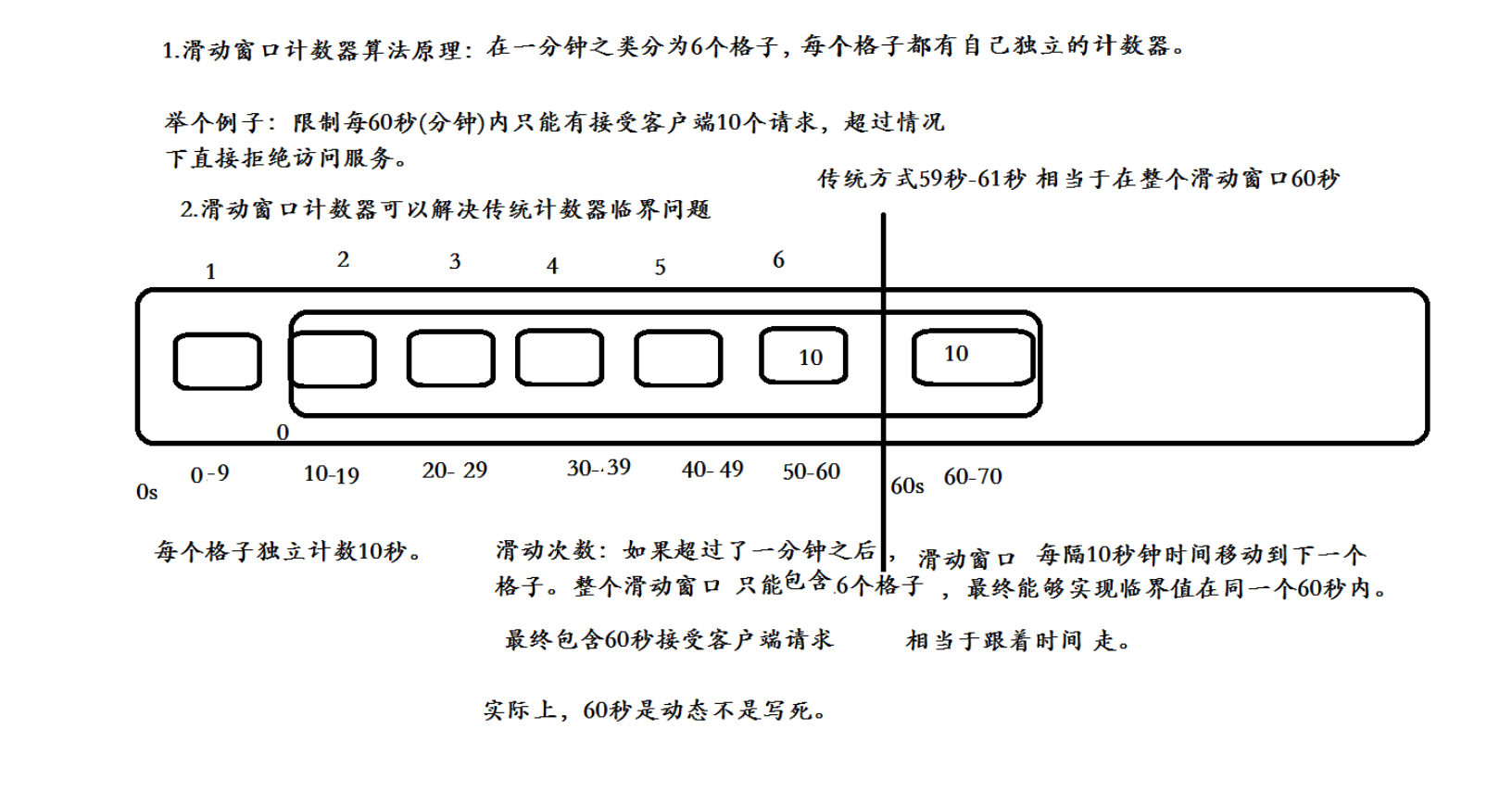
  它是限流算法中最简单最容易的一种算法，比如我们要求某一个接口，1分钟内的请求不能超过10次，我们可以在开始时设置一个计数器，每次请求，该计数器+1；如果该计数器的值大于10并且与第一次请求的时间间隔在1分钟内，那么说明请求过多，如果该请求与第一次请求的时间间隔大于1分钟，并且该计数器的值还在限流范围内，那么重置该计数器



**这种方式有个弊端，可能会产生临界的问题，见上图，假如第59秒进行了10个请求，61秒的时候又进行了10个请求，这20个请求都会执行成功，如果在临界的时候突然有大量请求，可能会违背定义固定速率。解决方法是使用滑动窗口计数器。**

|  |
| --- |
| /\*\*   \* 功能说明: 纯手写计数器方式<br>   \*/  **public** **class** LimitService {    **private** **int** limtCount = 60;// 限制最大访问的容量        AtomicInteger atomicInteger = **new** AtomicInteger(0); // 每秒钟 实际请求的数量  **private** **long** start = System.*currentTimeMillis*();// 获取当前系统时间  **private** **int** interval = 60;// 间隔时间60秒    **public** **boolean** acquire() {  **long** newTime = System.*currentTimeMillis*();  **if** (newTime > (start + interval)) {                    // 判断是否是一个周期                    start = newTime;                    atomicInteger.set(0); // 清理为0  **return** **true**;              }              atomicInteger.incrementAndGet();// i++;  **return** atomicInteger.get() <= limtCount;        }    **static** LimitService *limitService* = **new** LimitService();    **public** **static** **void** main(String[] args) {                ExecutorService newCachedThreadPool = Executors.*newCachedThreadPool*();  **for** (**int** i = 1; i < 100; i++) {  **final** **int** tempI = i;                    newCachedThreadPool.execute(**new** Runnable() {    **public** **void** run() {  **if** (*limitService*.acquire()) {                                     System.***out***.println("你没有被限流,可以正常访问逻辑 i:" + tempI);                               } **else** {                                     System.***out***.println("你已经被限流呢  i:" + tempI);                               }                         }                    });              }        }    } |

#### 滑动窗口计数



滑动窗口计数有很多使用场景，比如说限流防止系统雪崩。相比计数实现，滑动窗口实现会更加平滑，能自动消除毛刺。

滑动窗口原理是在每次有访问进来时，先判断前 N 个单位时间内的总访问量是否超过了设置的阈值，并对当前时间片上的请求数 +1。

#### 令牌桶算法

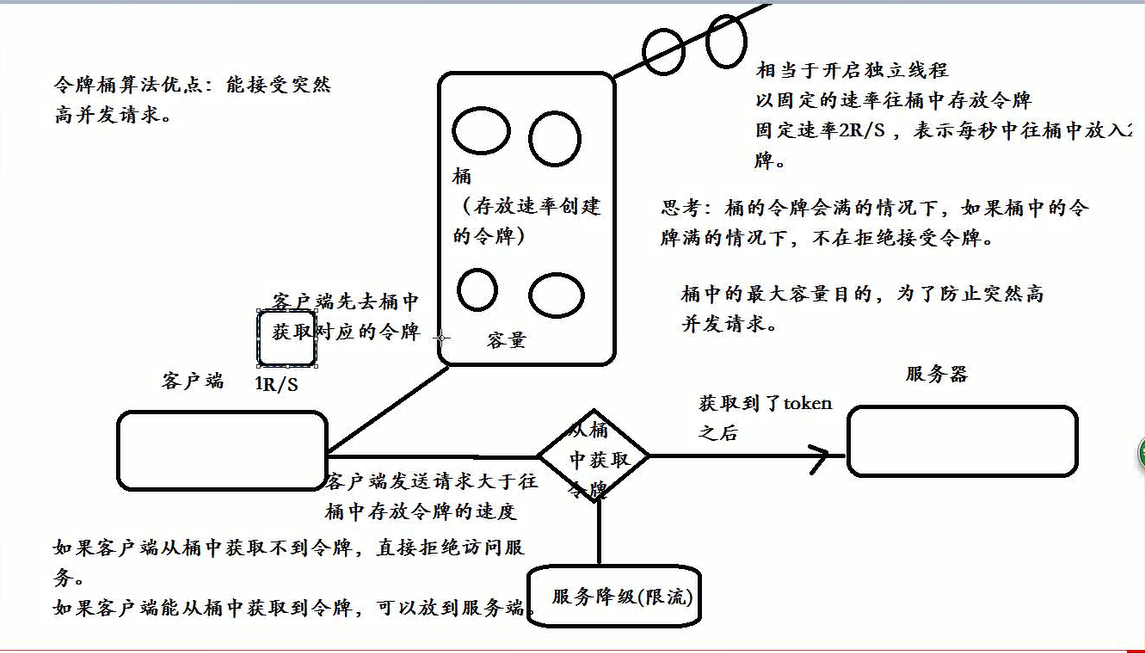
令牌桶算法是一个存放固定容量令牌的桶，按照固定速率往桶里添加令牌。令牌桶算法的描述如下：

假设限制2r/s(每秒钟放2个令牌)，则按照500毫秒的固定速率往桶中添加令牌；

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）。



**使用RateLimiter实现令牌桶限流**

RateLimiter是guava提供的基于令牌桶算法的实现类，可以非常简单的完成限流特技，并且根据系统的实际情况来调整生成token的速率。

通常可应用于抢购限流防止冲垮系统；限制某接口、服务单位时间内的访问量，譬如一些第三方服务会对用户访问量进行限制；限制网速，单位时间内只允许上传下载多少字节等。

下面来看一些简单的实践，需要先引入guava的maven依赖。

|  |
| --- |
| <parent>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-parent</artifactId>              <version>2.0.0.RELEASE</version>        </parent>        <dependencies>              <dependency>                    <groupId>org.springframework.boot</groupId>                    <artifactId>spring-boot-starter-web</artifactId>              </dependency>              <dependency>                    <groupId>com.google.guava</groupId>                    <artifactId>guava</artifactId>                    <version>25.1-jre</version>              </dependency>        </dependencies> |

|  |
| --- |
| /\*\*   \* 功能说明:使用RateLimiter 实现令牌桶算法   \*   \*/  @RestController  **public** **class** IndexController {        @Autowired  **private** OrderService orderService;        // 解释：1.0 表示 每秒中生成1个令牌存放在桶中        RateLimiter rateLimiter = RateLimiter.*create*(1.0);          // 下单请求        @RequestMapping("/order")  **public** String order() {              // 1.限流判断              // 如果在500秒内 没有获取不到令牌的话，则会一直等待              System.***out***.println("生成令牌等待时间:" + rateLimiter.acquire());  **boolean** acquire = rateLimiter.tryAcquire(500, TimeUnit.***MILLISECONDS***);  **if** (!acquire) {                    System.***out***.println("你在怎么抢，也抢不到，因为会一直等待的，你先放弃吧！");  **return** "你在怎么抢，也抢不到，因为会一直等待的，你先放弃吧！";              }                // 2.如果没有达到限流的要求,直接调用订单接口  **boolean** isOrderAdd = orderService.addOrder();  **if** (isOrderAdd) {  **return** "恭喜您,抢购成功!";              }  **return** "抢购失败!";        }    } |

#### 漏桶算法

漏桶作为计量工具（The Leaky Bucket Algorithm as a Meter）时，可以用于流量整形（Traffic Shaping）和流量控制（TrafficPolicing），漏桶算法的描述如下：

一个固定容量的漏桶，按照常量固定速率流出水滴；

如果桶是空的，则不需流出水滴；

可以以任意速率流入水滴到漏桶；

如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。

令牌桶和漏桶对比：

令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；

漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；

令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；

漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；

令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；

两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

另外有时候我们还使用计数器来进行限流，主要用来限制总并发数，比如数据库连接池、线程池、秒杀的并发数；只要全局总请求数或者一定时间段的总请求数设定的阀值则进行限流，是简单粗暴的总数量限流，而不是平均速率限流。

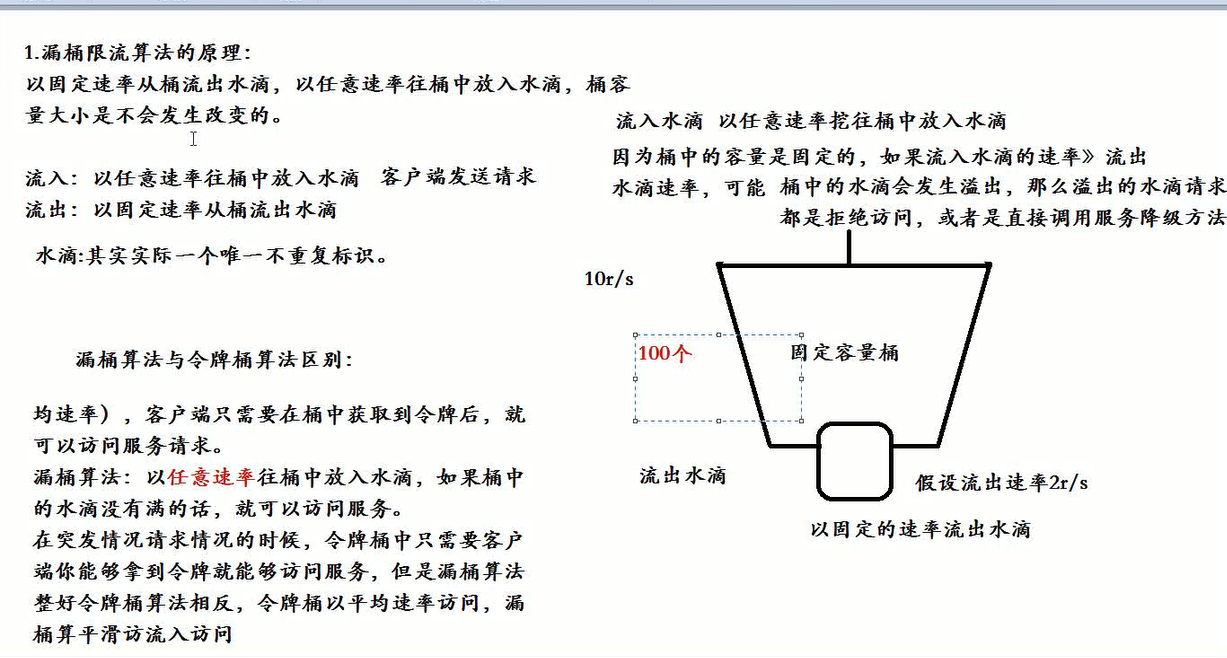
一个固定的漏桶，以常量固定的速率流出水滴。

如果桶中没有水滴的话，则不会流出水滴

如果流入的水滴超过桶中的流量，则流入的水滴可能会发生溢出，溢出的水滴请求是无法访问的，直接调用服务降级方法，桶中的容量是不会发生变化。

#### 漏桶算法与令牌桶算法区别

  ​    主要区别在于“漏桶算法”能够强行限制数据的传输速率，而“令牌桶算法”在能够限制数据的平均传输速率外，还允许某种程度的突发传输。在“令牌桶算法”中，只要令牌桶中存在令牌，那么就允许突发地传输数据直到达到用户配置的门限，因此它适合于具有突发特性的流量。



### 封装RateLimiter

自定义注解封装RateLimiter.实例:

|  |
| --- |
| @RequestMapping("/myOrder")  @ExtRateLimiter(value = 10.0, timeOut = 500)  **public** String myOrder() **throws** InterruptedException {              System.***out***.println("myOrder");  **return** "SUCCESS";  } |

**自定义注解**

|  |
| --- |
| @Target(value = ElementType.***METHOD***)  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** ExtRateLimiter {  **double** value();    **long** timeOut();  } |

**编写AOP**

|  |
| --- |
| @Aspect  @Component  **public** **class** RateLimiterAop {        // 存放接口是否已经存在  **private** **static** ConcurrentHashMap<String, RateLimiter> *rateLimiterMap* = **new** ConcurrentHashMap<String, RateLimiter>();          @Pointcut("execution(public \* com.itmayeidu.api.\*.\*(..))")  **public** **void** rlAop() {        }          @Around("rlAop()")  **public** Object doBefore(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {              MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();              // 使用Java反射技术获取方法上是否有@ExtRateLimiter注解类              ExtRateLimiter extRateLimiter = signature.getMethod().getDeclaredAnnotation(ExtRateLimiter.**class**);  **if** (extRateLimiter == **null**) {                    // 正常执行方法                    Object proceed = proceedingJoinPoint.proceed();  **return** proceed;              }              // ############获取注解上的参数 配置固定速率 ###############              // 获取配置的速率  **double** value = extRateLimiter.value();              // 获取等待令牌等待时间  **long** timeOut = extRateLimiter.timeOut();              RateLimiter rateLimiter = getRateLimiter(value, timeOut);              // 判断令牌桶获取token 是否超时  **boolean** tryAcquire = rateLimiter.tryAcquire(timeOut, TimeUnit.***MILLISECONDS***);  **if** (!tryAcquire) {                    serviceDowng();  **return** **null**;              }              // 获取到令牌,直接执行..              Object proceed = proceedingJoinPoint.proceed();  **return** proceed;          }          // 获取RateLimiter对象  **private** RateLimiter getRateLimiter(**double** value, **long** timeOut) {              // 获取当前URL              ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();              HttpServletRequest request = attributes.getRequest();              String requestURI = request.getRequestURI();              RateLimiter rateLimiter = **null**;  **if** (!*rateLimiterMap*.containsKey(requestURI)) {                    // 开启令牌通限流                    rateLimiter = RateLimiter.*create*(value); // 独立线程  *rateLimiterMap*.put(requestURI, rateLimiter);              } **else** {                    rateLimiter = *rateLimiterMap*.get(requestURI);              }  **return** rateLimiter;        }          // 服务降级  **private** **void** serviceDowng() **throws** IOException {              // 执行服务降级处理              System.***out***.println("执行降级方法,亲,服务器忙！请稍后重试!");              ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();              HttpServletResponse response = attributes.getResponse();              response.setHeader("Content-type", "text/html;charset=UTF-8");              PrintWriter writer = response.getWriter();  **try** {                    writer.println("执行降级方法,亲,服务器忙！请稍后重试!");              } **catch** (Exception e) {                } **finally** {                    writer.close();              }          }    **public** **static** **void** main(String[] args) {              // 使用Java反射技术获取方法上是否有@ExtRateLimiter注解类              ExtRateLimiter extRateLimiter = IndexController.**class**.getClass().getAnnotation(ExtRateLimiter.**class**);              System.***out***.println(extRateLimiter);        }    } |

**运行效果**

|  |
| --- |
| @RequestMapping("/myOrder")  @ExtRateLimiter(value = 10.0, timeOut = 500)  **public** String myOrder() **throws** InterruptedException {              System.***out***.println("myOrder");  **return** "SUCCESS";  } |

