<https://mp.weixin.qq.com/s/cA_lG9uEdACUhD6PuZguxA>

# 6 种限流实现方案

## **限流分类**

限流的实现方案有很多种，磊哥这里稍微理了一下，**限流的分类**如下所示：

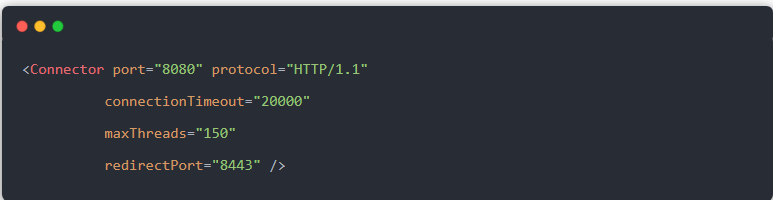
* **合法性验证限流**：比如验证码、IP 黑名单等，这些手段可以有效的防止恶意攻击和爬虫采集；
* **容器限流**：比如 Tomcat、Nginx 等限流手段，其中 Tomcat 可以设置最大线程数（maxThreads），当并发超过最大线程数会排队等待执行；而 Nginx 提供了两种限流手段：一是控制速率，二是控制并发连接数；
* **服务端限流**：比如我们在服务器端通过限流算法实现限流，此项也是我们本文介绍的重点。

合法性验证限流为最常规的业务代码，就是普通的验证码和 IP 黑名单系统，本文就不做过多的叙述了，我们重点来看下后两种限流的实现方案：容器限流和服务端限流。

## **容器限流**

### **Tomcat 限流**

Tomcat 8.5 版本的最大线程数在 conf/server.xml 配置中，如下所示：



其中 maxThreads 就是 Tomcat 的最大线程数，当请求的并发大于此值（maxThreads）时，请求就会排队执行，这样就完成了限流的目的。

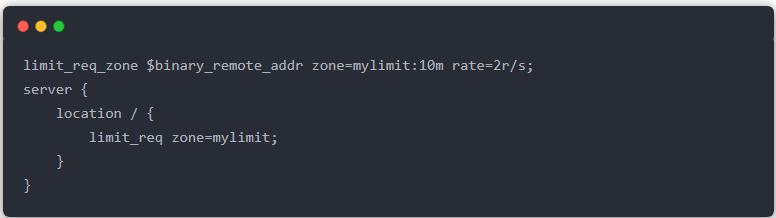
小贴士：maxThreads 的值可以适当的调大一些，此值默认为 150（Tomcat 版本 8.5.42），但这个值也不是越大越好，要看具体的硬件配置，需要注意的是每开启一个线程需要耗用 1MB 的 JVM 内存空间用于作为线程栈之用，并且线程越多 GC 的负担也越重。最后需要注意一下，操作系统对于进程中的线程数有一定的限制，Windows 每个进程中的线程数不允许超过 2000，Linux 每个进程中的线程数不允许超过 1000。

### **Nginx 限流**

Nginx 提供了两种限流手段：一是控制速率，二是控制并发连接数。

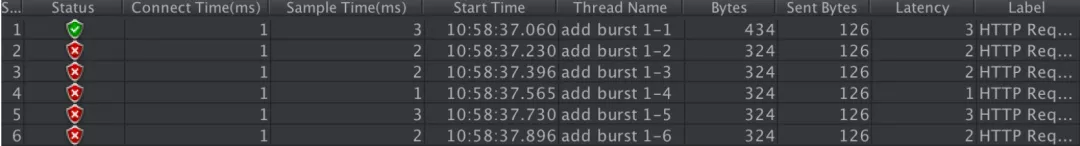
#### **控制速率**

我们需要使用 limit\_req\_zone 用来限制单位时间内的请求数，即速率限制，示例配置如下：



以上配置表示，限制每个 IP 访问的速度为 2r/s，因为 Nginx 的限流统计是基于毫秒的，我们设置的速度是 2r/s，转换一下就是 500ms 内单个 IP 只允许通过 1 个请求，从 501ms 开始才允许通过第 2 个请求。

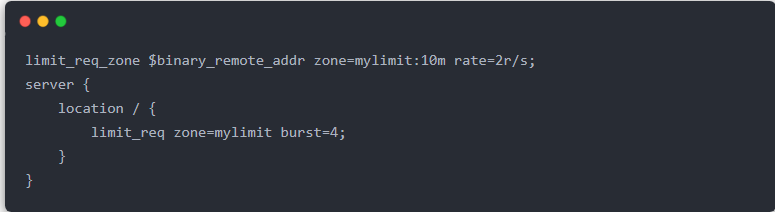
我们使用单 IP 在 10ms 内发并发送了 6 个请求的执行结果如下：



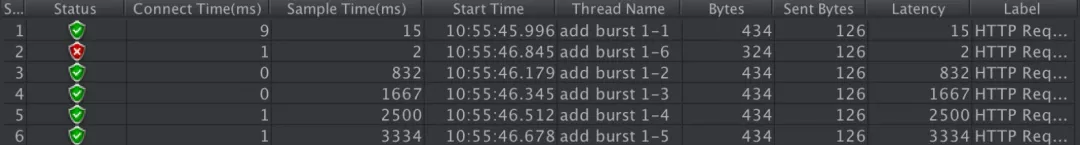
从以上结果可以看出他的执行符合我们的预期，只有 1 个执行成功了，其他的 5 个被拒绝了（第 2 个在 501ms 才会被正常执行）。

**速率限制升级版**

上面的速率控制虽然很精准但是应用于真实环境未免太苛刻了，真实情况下我们应该控制一个 IP 单位总时间内的总访问次数，而不是像上面那么精确但毫秒，我们可以使用 burst 关键字开启此设置，示例配置如下：



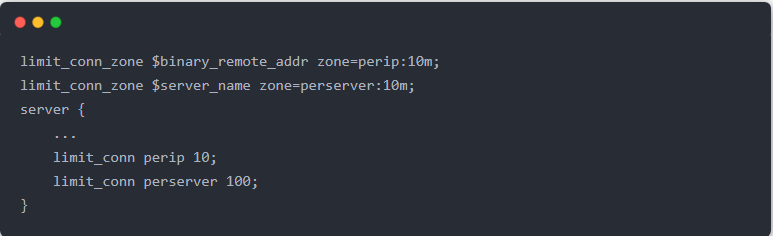
burst=4 表示每个 IP 最多允许4个突发请求，如果单个 IP 在 10ms 内发送 6 次请求的结果如下：



从以上结果可以看出，有 1 个请求被立即处理了，4 个请求被放到 burst 队列里排队执行了，另外 1 个请求被拒绝了。

#### **控制并发数**

利用 limit\_conn\_zone 和 limit\_conn 两个指令即可控制并发数，示例配置如下：



其中 limit\_conn perip 10 表示限制单个 IP 同时最多能持有 10 个连接；limit\_conn perserver 100 表示 server 同时能处理并发连接的总数为 100 个。

小贴士：只有当 request header 被后端处理后，这个连接才进行计数。

## **服务端限流**

服务端限流需要配合限流的算法来执行，而算法相当于执行限流的“大脑”，用于指导限制方案的实现。

有人看到「算法」两个字可能就晕了，觉得很深奥，其实并不是。算法就相当于操作某个事务的具体实现步骤汇总，其实并不难懂，不要被它的表象给吓到哦~

**限流的常见算法**有以下三种：

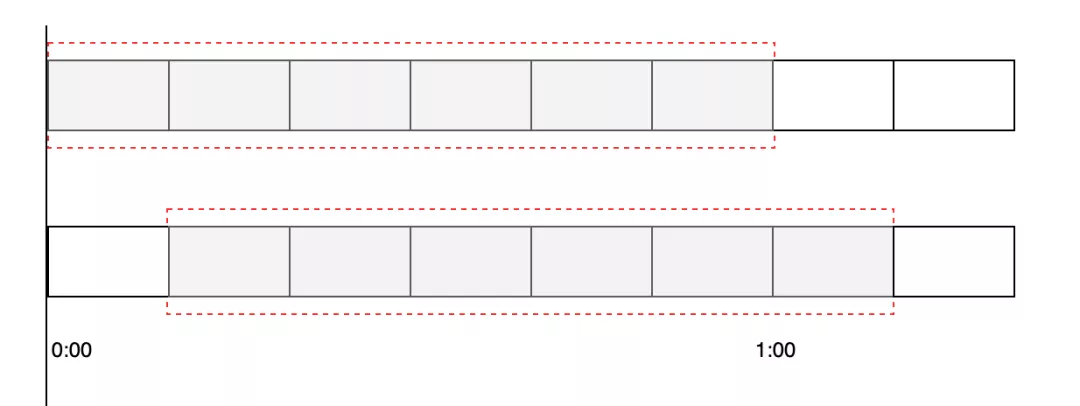
* **时间窗口算法**
* **漏桶算法**
* **令牌算法**

接下来我们分别看来。

### **1.时间窗口算法**

所谓的滑动时间算法指的是以当前时间为截止时间，往前取一定的时间，比如往前取 60s 的时间，在这 60s 之内运行最大的访问数为 100，此时算法的执行逻辑为，先清除 60s 之前的所有请求记录，再计算当前集合内请求数量是否大于设定的最大请求数 100，如果大于则执行限流拒绝策略，否则插入本次请求记录并返回可以正常执行的标识给客户端。

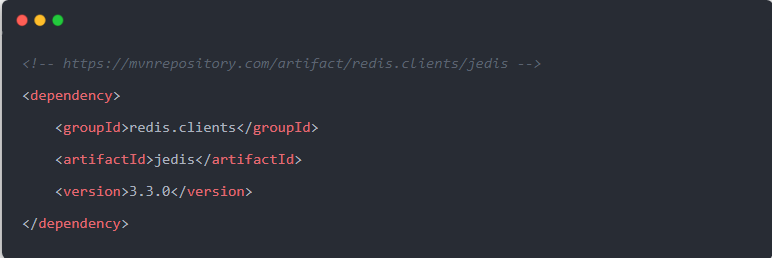
滑动时间窗口如下图所示：



其中每一小个表示 10s，被红色虚线包围的时间段则为需要判断的时间间隔，比如 60s 秒允许 100 次请求，那么红色虚线部分则为 60s。

我们可以借助 Redis 的有序集合 ZSet 来实现时间窗口算法限流，实现的过程是先使用 ZSet 的 key 存储限流的 ID，score 用来存储请求的时间，每次有请求访问来了之后，先清空之前时间窗口的访问量，统计现在时间窗口的个数和最大允许访问量对比，如果大于等于最大访问量则返回 false 执行限流操作，否则允许执行业务逻辑，并且在 ZSet 中添加一条有效的访问记录，具体实现代码如下。

我们借助 Jedis 包来操作 Redis，实现在 pom.xml 添加 Jedis 框架的引用，配置如下：

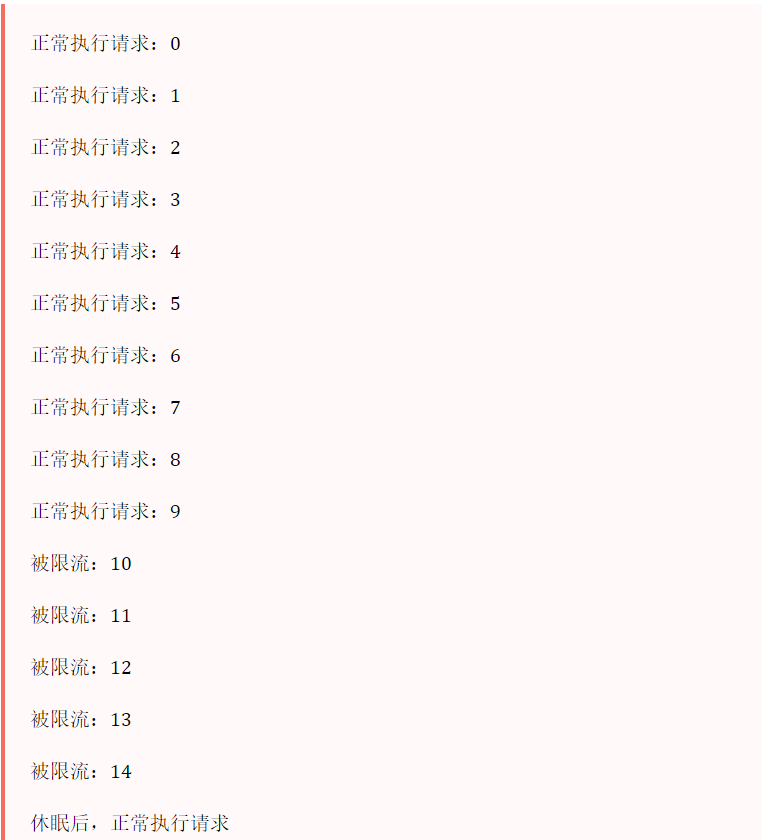


具体的 Java 实现代码如下：





以上程序的执行结果为：

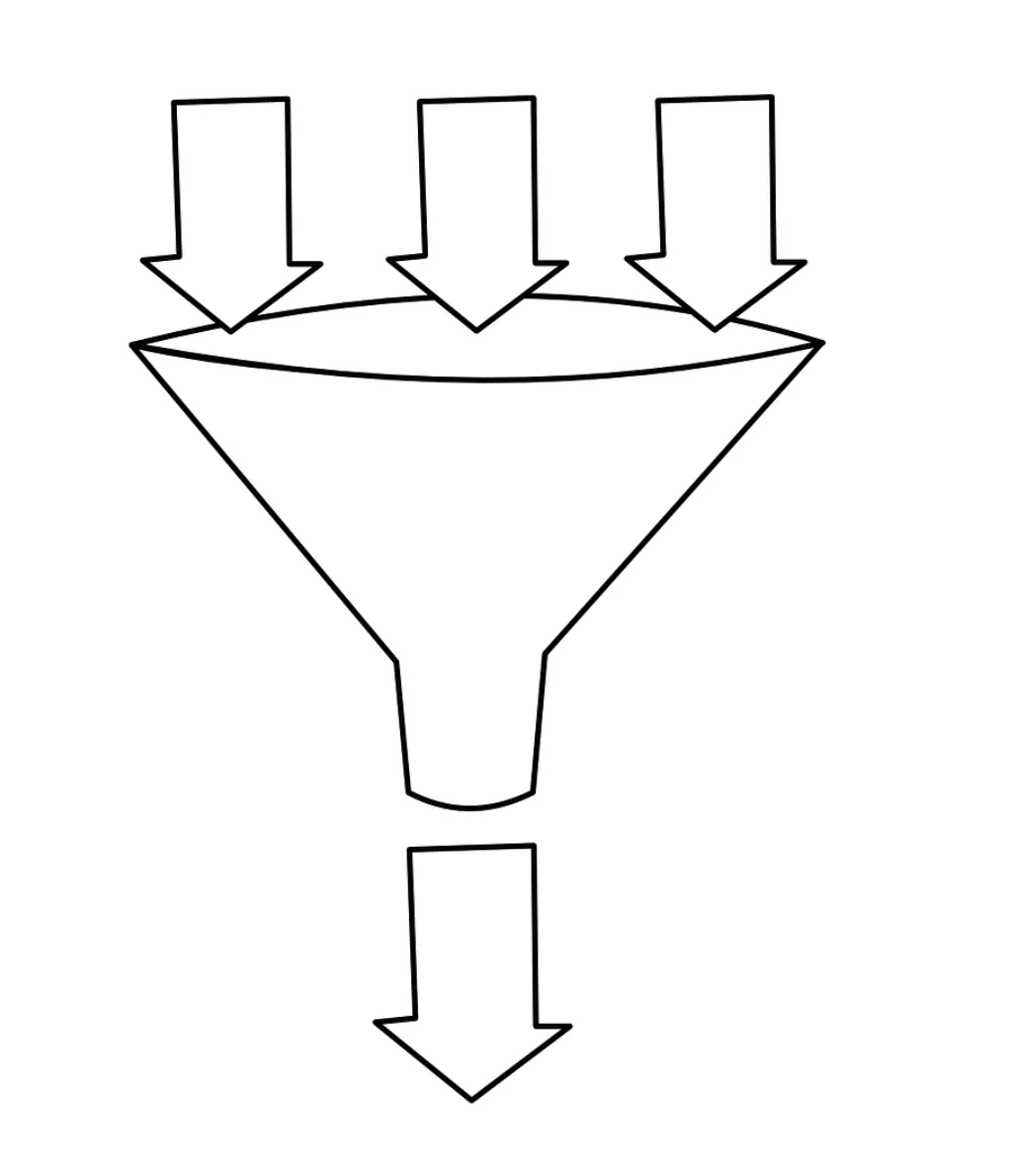


此实现方式存在的缺点有两个：

* 使用 ZSet 存储有每次的访问记录，如果数据量比较大时会占用大量的空间，比如 60s 允许 100W 访问时；
* 此代码的执行非原子操作，先判断后增加，中间空隙可穿插其他业务逻辑的执行，最终导致结果不准确。

### **2.漏桶算法**

漏桶算法的灵感源于漏斗，如下图所示：



滑动时间算法有一个问题就是在一定范围内，比如 60s 内只能有 10 个请求，当第一秒时就到达了 10 个请求，那么剩下的 59s 只能把所有的请求都给拒绝掉，而漏桶算法可以解决这个问题。

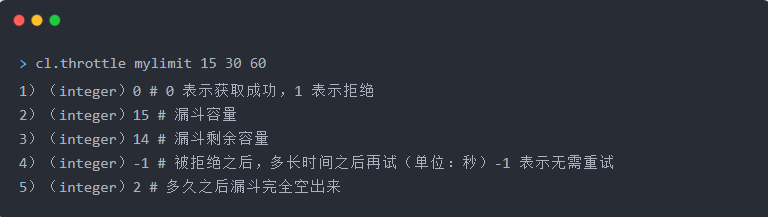
漏桶算法类似于生活中的漏斗，无论上面的水流倒入漏斗有多大，也就是无论请求有多少，它都是以均匀的速度慢慢流出的。当上面的水流速度大于下面的流出速度时，漏斗会慢慢变满，当漏斗满了之后就会丢弃新来的请求;当上面的水流速度小于下面流出的速度的话，漏斗永远不会被装满，并且可以一直流出。

漏桶算法的实现步骤是，先声明一个队列用来保存请求，这个队列相当于漏斗，当队列容量满了之后就放弃新来的请求，然后重新声明一个线程定期从任务队列中获取一个或多个任务进行执行，这样就实现了漏桶算法。

上面我们演示 Nginx 的控制速率其实使用的就是漏桶算法，当然我们也可以借助 Redis 很方便的实现漏桶算法。

我们可以使用 Redis 4.0 版本中提供的 Redis-Cell 模块，该模块使用的是漏斗算法，并且提供了原子的限流指令，而且依靠 Redis 这个天生的分布式程序就可以实现比较完美的限流了。

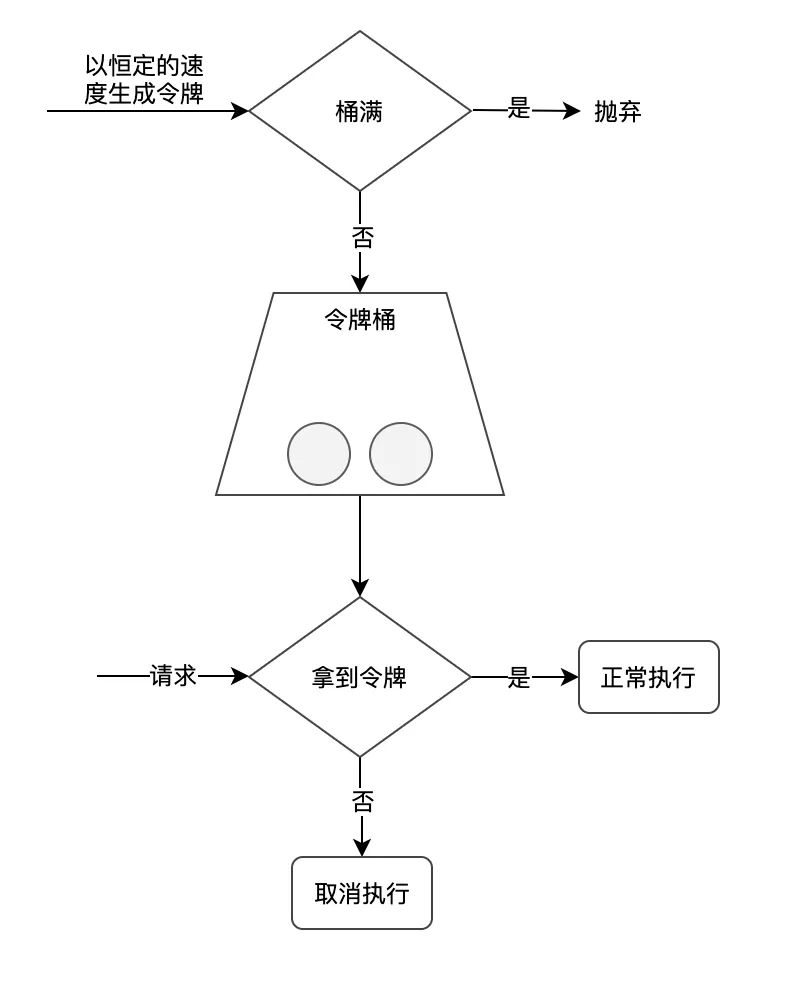
Redis-Cell 实现限流的方法也很简单，只需要使用一条指令 cl.throttle 即可，使用示例如下：



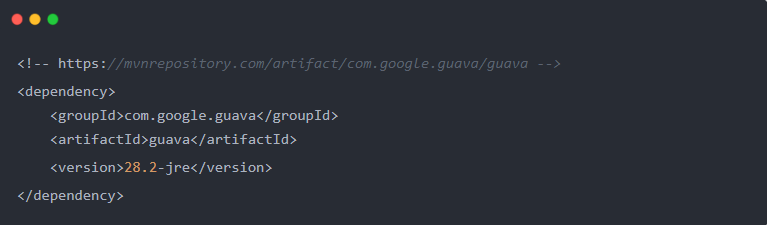
其中 15 为漏斗的容量，30 / 60s 为漏斗的速率。

### **3.令牌算法**

在令牌桶算法中有一个程序以某种恒定的速度生成令牌，并存入令牌桶中，而每个请求需要先获取令牌才能执行，如果没有获取到令牌的请求可以选择等待或者放弃执行，如下图所示：



我们可以使用 Google 开源的 guava 包，很方便的实现令牌桶算法，首先在 pom.xml 添加 guava 引用，配置如下：



具体实现代码如下：



以上程序的执行结果为：

正常执行方法，ts:2020-05-15T14:46:37.175Z

正常执行方法，ts:2020-05-15T14:46:37.237Z

正常执行方法，ts:2020-05-15T14:46:37.339Z

正常执行方法，ts:2020-05-15T14:46:37.442Z

正常执行方法，ts:2020-05-15T14:46:37.542Z

正常执行方法，ts:2020-05-15T14:46:37.640Z

正常执行方法，ts:2020-05-15T14:46:37.741Z

正常执行方法，ts:2020-05-15T14:46:37.840Z

正常执行方法，ts:2020-05-15T14:46:37.942Z

正常执行方法，ts:2020-05-15T14:46:38.042Z

正常执行方法，ts:2020-05-15T14:46:38.142Z

从以上结果可以看出令牌确实是每 100ms 产生一个，而 acquire() 方法为阻塞等待获取令牌，它可以传递一个 int 类型的参数，用于指定获取令牌的个数。它的替代方法还有 tryAcquire()，此方法在没有可用令牌时就会返回 false 这样就不会阻塞等待了。当然 tryAcquire() 方法也可以设置超时时间，未超过最大等待时间会阻塞等待获取令牌，如果超过了最大等待时间，还没有可用的令牌就会返回 false。

注意：使用 guava 实现的令牌算法属于程序级别的单机限流方案，而上面使用 Redis-Cell 的是分布式的限流方案。

## **总结**

本文提供了 6 种具体的实现限流的手段，他们分别是：Tomcat 使用 maxThreads 来实现限流；Nginx 提供了两种限流方式，一是通过 limit\_req\_zone 和 burst 来实现速率限流，二是通过 limit\_conn\_zone 和 limit\_conn 两个指令控制并发连接的总数。最后我们讲了时间窗口算法借助 Redis 的有序集合可以实现，还有漏桶算法可以使用 Redis-Cell 来实现，以及令牌算法可以解决 Google 的 guava 包来实现。

需要注意的是借助 Redis 实现的限流方案可用于分布式系统，而 guava 实现的限流只能应用于单机环境。如果你嫌弃服务器端限流麻烦，甚至可以在不改代码的情况下直接使用容器限流（Nginx 或 Tomcat），但前提是能满足你的业务需求。