谈谈高并发系统的限流

1. **缓存**

缓存比较好理解，在大型高并发系统中，如果没有缓存数据库将分分钟被爆，系统也会瞬间瘫痪。使用缓存不单单能够提升系统访问速度、提高并发访问量，也是保护数据库、保护系统的有效方式。大型网站一般主要是“读”，缓存的使用很容易被想到。在大型“写”系统中，缓存也常常扮演者非常重要的角色。比如累积一些数据批量写入，内存里面的缓存队列（生产消费），以及HBase写数据的机制等等也都是通过缓存提升系统的吞吐量或者实现系统的保护措施。甚至消息中间件，你也可以认为是一种分布式的数据缓存。

1. **降级**

服务降级是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。降级往往会指定不同的级别，面临不同的异常等级执行不同的处理。根据服务方式：可以拒接服务，可以延迟服务，也有时候可以随机服务。根据服务范围：可以砍掉某个功能，也可以砍掉某些模块。总之服务降级需要根据不同的业务需求采用不同的降级策略。主要的目的就是服务虽然有损但是总比没有好。

1. **限流**

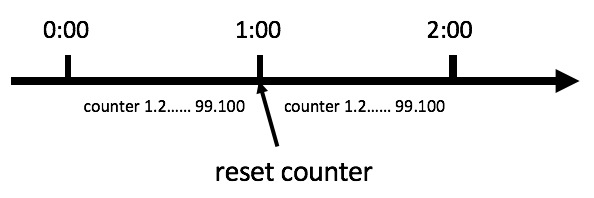
限流可以认为服务降级的一种，限流就是限制系统的输入和输出流量已达到保护系统的目的。一般来说系统的吞吐量是可以被测算的，为了保证系统的稳定运行，一旦达到的需要限制的阈值，就需要限制流量并采取一些措施以完成限制流量的目的。比如：延迟处理，拒绝处理，或者部分拒绝处理等等。

1. **限流的算法**

常见的限流算法有：计数器、漏桶和令牌桶算法。

* 1. **计数器**

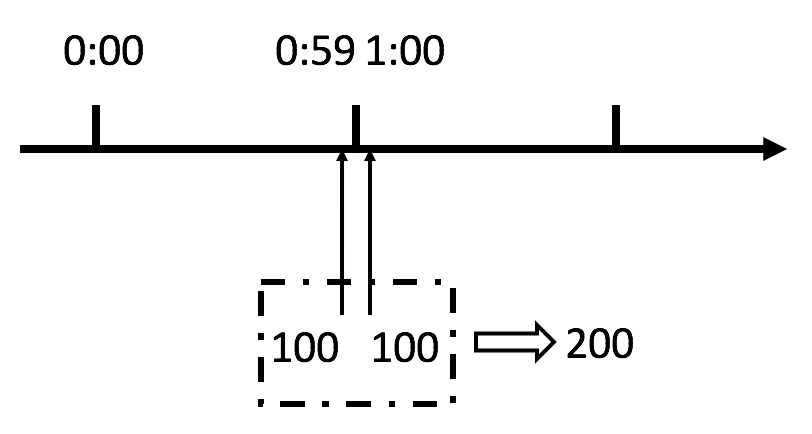
计数器法是限流算法里最简单也是最容易实现的一种算法。比如我们规定，对于A接口来说，我们1分钟的访问次数不能超过100个。那么我们可以这么做：在一开 始的时候，我们可以设置一个计数器counter，每当一个请求过来的时候，counter就加1，如果counter的值大于100并且该请求与第一个 请求的间隔时间还在1分钟之内，那么说明请求数过多；如果该请求与第一个请求的间隔时间大于1分钟，且counter的值还在限流范围内，那么就重置 counter，具体算法的示意图如下：



简单的伪代码如下：

|  |
| --- |
| public class CounterDemo {  public long timeStamp = getNowTime();  public int reqCount = 0;  public final int limit = 100; // 时间窗口内最大请求数  public final long interval = 1000; // 时间窗口ms  public boolean grant() {  long now = getNowTime();  if (now < timeStamp + interval) {  // 在时间窗口内  reqCount++;  // 判断当前时间窗口内是否超过最大请求控制数  return reqCount <= limit;  }else {  timeStamp = now;  // 超时后重置  reqCount = 1;  return true;  }  }  } |

这个算法虽然简单，但是有一个十分致命的问题，那就是临界问题，我们看下图：

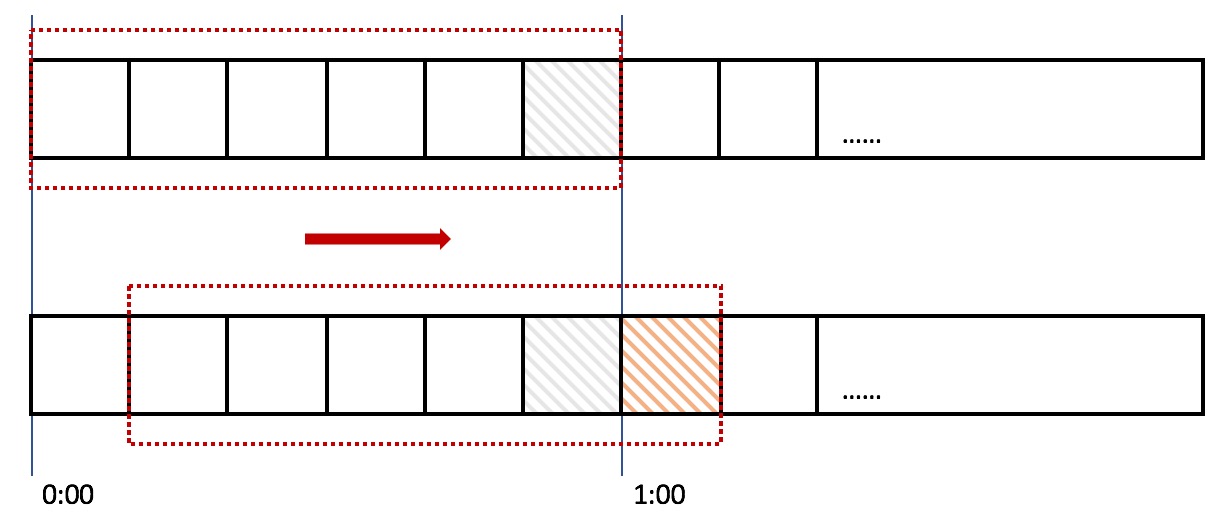


从上图中我们可以看到，假设有一个恶意用户，他在0:59时，瞬间发送了100个请求，并且1:00又瞬间发送了100个请求，那么其实这个用户在 1秒里面，瞬间发送了200个请求。我们刚才规定的是1分钟最多100个请求，也就是每秒钟最多1.7个请求，用户通过在时间窗口的重置节点处突发请求， 可以瞬间超过我们的速率限制。用户有可能通过算法的这个漏洞，瞬间压垮我们的应用。

聪明的朋友可能已经看出来了，刚才的问题其实是因为我们统计的精度太低。那么如何很好地处理这个问题呢？或者说，如何将临界问题的影响降低呢？我们可以看下面的滑动窗口算法。

* 1. **滑动窗口**

滑动窗口，又称rolling window。为了解决这个问题，我们引入了滑动窗口算法。如果学过TCP网络协议的话，那么一定对滑动窗口这个名词不会陌生。下面这张图，很好地解释了滑动窗口算法。



在上图中，整个红色的矩形框表示一个时间窗口，在我们的例子中，一个时间窗口就是一分钟。然后我们将时间窗口进行划分，比如图中，我们就将滑动窗口 划成了6格，所以每格代表的是10秒钟。每过10秒钟，我们的时间窗口就会往右滑动一格。每一个格子都有自己独立的计数器counter，比如当一个请求 在0:35秒的时候到达，那么0:30~0:39对应的counter就会加1。

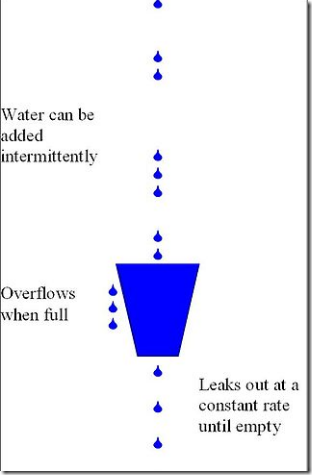
那么滑动窗口怎么解决刚才的临界问题的呢？我们可以看上图，0:59到达的100个请求会落在灰色的格子中，而1:00到达的请求会落在橘黄色的格 子中。当时间到达1:00时，我们的窗口会往右移动一格，那么此时时间窗口内的总请求数量一共是200个，超过了限定的100个，所以此时能够检测出来触 发了限流。

我再来回顾一下刚才的计数器算法，我们可以发现，计数器算法其实就是滑动窗口算法。只是它没有对时间窗口做进一步地划分，所以只有1格。

由此可见，当滑动窗口的格子划分的越多，那么滑动窗口的滚动就越平滑，限流的统计就会越精确。

* 1. **漏桶算法**

漏桶算法即leaky bucket是一种非常常用的限流算法，可以用来实现流量整形（Traffic Shaping）和流量控制（Traffic Policing）。贴了一张维基百科上示意图帮助大家理解：



漏桶算法的主要概念如下：

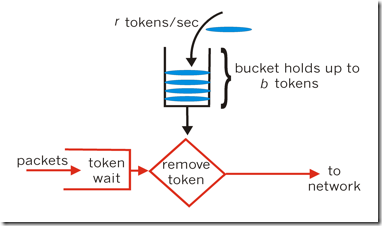
* 一个固定容量的漏桶，按照常量固定速率流出水滴；
* 如果桶是空的，则不需流出水滴；
* 可以以任意速率流入水滴到漏桶；
* 如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。

漏桶算法比较好实现，在单机系统中可以使用队列来实现（.Net中TPL DataFlow可以较好的处理类似的问题，你可以在这里找到相关的介绍），在分布式环境中消息中间件或者Redis都是可选的方案。

* 1. **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：

* 令牌将按照固定的速率被放入令牌桶中。比如每秒放10个。
* 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）。



令牌算法是根据放令牌的速率去控制输出的速率，也就是上图的to network的速率。to network我们可以理解为消息的处理程序，执行某段业务或者调用某个RPC。

* 1. **漏桶和令牌桶的比较**

令牌桶可以在运行时控制和调整数据处理的速率，处理某时的突发流量。放令牌的频率增加可以提升整体数据处理的速度，而通过每次获取令牌的个数增加或者放慢令牌的发放速度和降低整体数据处理速度。而漏桶不行，因为它的流出速率是固定的，程序处理速度也是固定的。

整体而言，令牌桶算法更优，但是实现更为复杂一些。

1. **限流算法实现**
   1. Guava

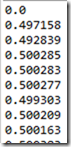
Guava是一个Google开源项目，包含了若干被Google的Java项目广泛依赖的核心库，其中的RateLimiter提供了令牌桶算法实现：平滑突发限流(SmoothBursty)和平滑预热限流(SmoothWarmingUp)实现。

* + 1. **常规速率**

创建一个限流器，设置每秒放置的令牌数：2个。返回的RateLimiter对象可以保证1秒内不会给超过2个令牌，并且是固定速率的放置。达到平滑输出的效果。

|  |
| --- |
| public void test(){  /\*\*  \* 创建一个限流器，设置每秒放置的令牌数：2个。速率是每秒可以2个的消息。  \* 返回的RateLimiter对象可以保证1秒内不会给超过2个令牌，并且是固定速率的放置。达到平滑输出的效果  \*/  RateLimiter r = RateLimiter.create(2);  while (true){  /\*\*  \* acquire()获取一个令牌，并且返回这个获取这个令牌所需要的时间。如果桶里没有令牌则等待，直到有令牌。  \* acquire(N)可以获取多个令牌。  \*/  System.out.println(r.acquire());  }  } |

上面代码执行的结果如下图，基本是0.5秒一个数据。拿到令牌后才能处理数据，达到输出数据或者调用接口的平滑效果。acquire()的返回值是等待令牌的时间，如果需要对某些突发的流量进行处理的话，可以对这个返回值设置一个阈值，根据不同的情况进行处理，比如过期丢弃。

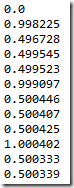


* + 1. **突发流量**

突发流量可以是突发的多，也可以是突发的少。首先来看个突发多的例子。还是上面例子的流量，每秒2个数据令牌。如下代码使用acquire方法，指定参数。

|  |
| --- |
| System.out.println(r.acquire(2));  System.out.println(r.acquire(1));  System.out.println(r.acquire(1));  System.out.println(r.acquire(1)); |

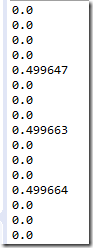
得到如下类似的输出。



如果要一次新处理更多的数据，则需要更多的令牌。代码首先获取2个令牌，那么下一个令牌就不是0.5秒之后获得了，还是1秒以后，之后又恢复常规速度。这是一个突发多的例子，如果是突发没有流量，如下代码：

|  |
| --- |
| System.out.println(r.acquire(1));  Thread.sleep(2000);  System.out.println(r.acquire(1));  System.out.println(r.acquire(1));  System.out.println(r.acquire(1)); |

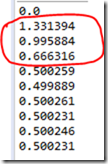
得到如下类似的结果：



等了两秒钟之后，令牌桶里面就积累了3个令牌，可以连续不花时间的获取出来。处理突发其实也就是在单位时间内输出恒定。这两种方式都是使用的RateLimiter的子类SmoothBursty。另一个子类是SmoothWarmingUp，它提供的有一定缓冲的流量输出方案。

|  |
| --- |
| /\*\*  \* 创建一个限流器，设置每秒放置的令牌数：2个。速率是每秒可以210的消息。  \* 返回的RateLimiter对象可以保证1秒内不会给超过2个令牌，并且是固定速率的放置。达到平滑输出的效果  \* 设置缓冲时间为3秒  \*/  RateLimiter r = RateLimiter.create(2,3,TimeUnit.SECONDS);  while (true) {  /\*\*  \* acquire()获取一个令牌，并且返回这个获取这个令牌所需要的时间。如果桶里没有令牌则等待，直到有令牌。  \* acquire(N)可以获取多个令牌。  \*/  System.out.println(r.acquire(1));  System.out.println(r.acquire(1));  System.out.println(r.acquire(1));  System.out.println(r.acquire(1));  } |

输出结果如下图，由于设置了缓冲的时间是3秒，令牌桶一开始并不会0.5秒给一个消息，而是形成一个平滑线性下降的坡度，频率越来越高，在3秒钟之内达到原本设置的频率，以后就以固定的频率输出。图中红线圈出来的3次累加起来正好是3秒左右。这种功能适合系统刚启动需要一点时间来“热身”的场景。



* 1. **Nginx**

对于Nginx接入层限流可以使用Nginx自带了两个模块：连接数限流模块ngx\_http\_limit\_conn\_module和漏桶算法实现的请求限流模块ngx\_http\_limit\_req\_module。

* + 1. **ngx\_http\_limit\_conn\_module**

我们经常会遇到这种情况，服务器流量异常，负载过大等等。对于大流量恶意的攻击访问，会带来带宽的浪费，服务器压力，影响业务，往往考虑对同一个ip的连接数，并发数进行限制。ngx\_http\_limit\_conn\_module 模块来实现该需求。该模块可以根据定义的键来限制每个键值的连接数，如同一个IP来源的连接数。并不是所有的连接都会被该模块计数，只有那些正在被处理的请求（这些请求的头信息已被完全读入）所在的连接才会被计数。

我们可以在nginx\_conf的http{}中加上如下配置实现限制：

|  |
| --- |
| #限制每个用户的并发连接数，取名one  limit\_conn\_zone $binary\_remote\_addr zone=one:10m;  #配置记录被限流后的日志级别，默认error级别  limit\_conn\_log\_level error;  #配置被限流后返回的状态码，默认返回503  limit\_conn\_status 503; |

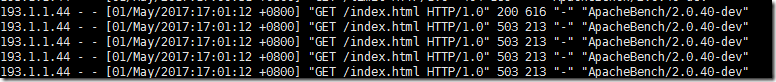
然后在server{}里加上如下代码：

|  |
| --- |
| #限制用户并发连接数为1  limit\_conn one 1; |

然后我们是使用ab测试来模拟并发请求：

ab -n 5 -c 5 http://10.23.22.239/index.html

得到下面的结果，很明显并发被限制住了，超过阈值的都显示503：



另外刚才是配置针对单个IP的并发限制，还是可以针对域名进行并发限制，配置和客户端IP类似。

|  |
| --- |
| #http{}段配置  limit\_conn\_zone $ server\_name zone=perserver:10m;  #server{}段配置  limit\_conn perserver 1; |

* + 1. **ngx\_http\_limit\_req\_module**

上面我们使用到了ngx\_http\_limit\_conn\_module 模块，来限制连接数。那么请求数的限制该怎么做呢？这就需要通过ngx\_http\_limit\_req\_module 模块来实现，该模块可以通过定义的键值来限制请求处理的频率。特别的，可以限制来自单个IP地址的请求处理频率。 限制的方法是使用了漏斗算法，每秒固定处理请求数，推迟过多请求。如果请求的频率超过了限制域配置的值，请求处理会被延迟或被丢弃，所以所有的请求都是以定义的频率被处理的。

|  |
| --- |
| 在http{}中配置  #区域名称为one，大小为10m，平均处理的请求频率不能超过每秒一次。  limit\_req\_zone $binary\_remote\_addr zone=one:10m rate=1r/s; |

|  |
| --- |
| 在server{}中配置  #设置每个IP桶的数量为5  limit\_req zone=one burst=5; |

上面设置定义了每个IP的请求处理只能限制在每秒1个。并且服务端可以为每个IP缓存5个请求，如果操作了5个请求，请求就会被丢弃。

使用ab测试模拟客户端连续访问10次：ab -n 10 -c 10 http://10.23.22.239/index.html

如下图，设置了通的个数为5个。一共10个请求，第一个请求马上被处理。第2-6个被存放在桶中。由于桶满了，没有设置nodelay因此，余下的4个请求被丢弃。



1. **来源**

<https://www.cnblogs.com/haoxinyue/p/6792309.html>

https://www.cnblogs.com/clds/p/5850070.html