

36 | 全栈开发中的算法（上）

2019-12-02 四火

全栈工程师修炼指南

[进入课程 >](#)

123456789

全栈开发中的算法（上）

讲述：四火

时长 18:20 大小 12.61M



你好，我是四火。

在本专栏中，我们已经接触到了全栈开发中的一些算法了。在这一讲和下一讲中，我又精心挑选了几个比较重要的。**和单纯地从数学角度去介绍算法不同，我想结合几个全栈开发中实际、典型的问题场景，向你介绍这几个相关的重要算法。**毕竟，我们关心的算法，其实就是可以解决实际问题的方法的一种数学抽象。

希望通过这两讲的学习，你能理解这些算法。除了理解算法原理本身，我们更要抓住它们的用途和算法自身的巧妙之处。今天我们来讲其中的第一个典型的问题场景——流量控制。

流量控制系统中的算法

对于全栈工程师来说，无论是网站，还是其它 Web 应用，一旦对外商用，就要考虑流量控制的问题。因此我们往往需要设计使用单独的流量控制模块，我们来看下面这样的一个问题。

假如说，我们现在需要给一组 Web API 设计一个流量控制系统，以避免请求对系统的过度冲击，对于任意一个用户账户 ID，每一个 API 都要满足下面所有的要求：

每分钟调用不能超过一万次；

每小时调用不能超过十万次；

每天调用不能超过一百万次；

每周调用不能超过一千万次；

.....

在继续往下阅读之前，请你先从算法和数据结构的角​​度思考，你觉得该怎么设计这个流量控制系统呢？

简化问题

在解决实际问题的時候，我们面临的问题往往是复杂的、多样的，因此，我们可以**考虑能不能先简化问题，再来尝试映射到某一个数学模型上。那些先不考虑的复杂条件，有的可能就是可以忽略掉的，而有的则是为了思路的清晰。一开始我们可以先忽略，有了解题的方法原型以后，再逐步加回来考虑。**

那就这个问题而言，我可以做如下的简化：


有大量的用户账户 ID，但是我们现在只考虑某一个特定的账户 ID，反正其它账户 ID 的做法也是一样的；

这里面有多个 Web API，但是我们可以只考虑其中特定的一个 API，反正其它 API 也是类似的；

这里面有多条规则，但是我们可以只考虑其中的一个规则，即“每分钟调用不能超过一万次”，至于其它的规则，原理上也是一样的。

为了简化问题，在这里我们也暂不考虑并发、分布式、线程安全等问题。

好，现在问题就简单多了，当我们把这个简化了的问题解决了之后，我们再引入多个用户 ID、多个 API 和多条规则这样的维度：

 复制代码

```
1 public class RateLimiter {
2     public boolean isAllowed() {
3         ... // 当每分钟调用不超过 10000 次，就返回 true，否则返回 false
4     }
5 }
```

简单计数

好，最先进入脑海的是采用简单计数的办法，我们给 RateLimiter 一个起始时间的时间戳。如果当前时间在距离起始时间一分钟以内，我们就看当前已经放进来了多少个请求，如果是 10000 个以内，就允许访问，否则就拒绝访问；如果当前时间已经超过了起始时间一分钟，就更新时间戳，并清零计数器。参考代码如下：

 复制代码

```
1 public class RateLimiter {
2     private long start = System.currentTimeMillis();
3     private int count = 0;
4
5     public boolean isAllowed() {
6         long now = System.currentTimeMillis();
7         if (now-start > 60*1000) {
8             start = now - (now-start)%(60*1000); // 所在时间窗口的起始位置
9             count = 0;
10        }
11
12        if (count < 10000) {
13            count++;
14            return true;
15        }
16        return false;
17    }
18 }
```

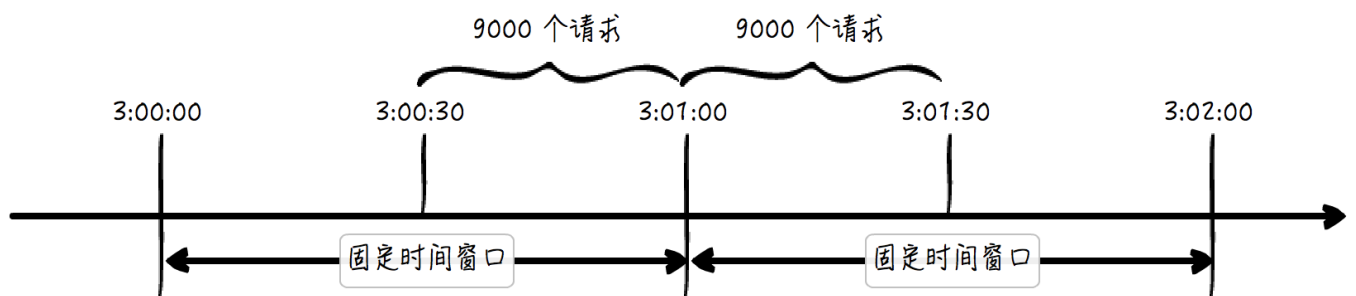
这样这个问题似乎就得到了解决。可是，刚才我们在解决问题的时候，似乎“擅自”强化了一个条件.....

从固定窗口到滑动窗口

这个条件就是“固定时间窗口”。

举个例子，从 3:00 到 3:01 这一分钟时间内，假如系统收到了 9000 个请求，而在 3:01 到 3:02 这接着的一分钟内，系统也收到了 9000 个请求，二者都满足要求。但是，这是我们给了一个假定的增强条件——固定时间窗口，而得到的结论。

假如说前面这 9000 个请求都分布在 3:00:30 到 3:01:00 之间，后面这 9000 个请求都分布在 3:01:00 到 3:01:30 之间，即从 3 点 00 分 30 秒到 3 点 01 分 30 秒这一分钟内，系统居然接纳了 $9000 + 9000 = 18000$ 个请求。因此，如果我们考虑的是“滑动时间窗口”，这显然违背了我们的每分钟一万次最大请求量的规则。请看下图：

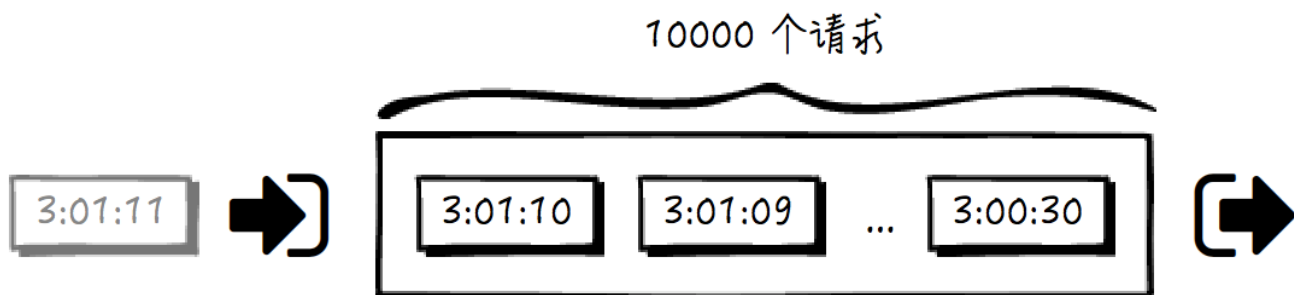


因此，相较来说，更实际的情况下，我们是要支持滑动时间窗口，也就是任意一分钟的时间窗口内，都要满足小于 10000 请求量的规则。看来，简单计数法需要改进。

时间队列

对于滑动窗口的问题，我们经常要引入队列来解决问题。因为队列的特点是先进先出，一头进，另一头出，而很多滑动窗口的问题，恰恰需要这个特性，这个问题也不例外。

假如我们维护一个最近时间戳的队列，这个队列长度不能超过 10000，那么，当新的请求到来的时候，我们只需要找到从“当前时间减 1 分钟”到“当前时间”这样一个滑动窗口区间。如果队列的尾部有任何存储的时间戳在这个区间之外（一分钟以前），那我们就把它从队列中拿掉。如果队列长度小于 10000，那么这个新的请求的时间戳就可以入队列，允许请求访问；反之，则不允许请求访问。请看下图：



这个过程，参考代码如下：

 复制代码

```
1 public class RateLimiter {
2     private Queue<Long> queue = new LinkedList<>();
3
4     public boolean isAllowed() {
5         long now = System.currentTimeMillis();
6         while (!queue.isEmpty() && queue.peek() < now-60*1000) {
7             // 如果请求已经是在一分钟以前了，忽略
8             queue.remove();
9         }
10        if (queue.size() < 10000) {
11            queue.add(now);
12            return true;
13        }
14        return false;
15    }
16 }
```

你可以看到，这个算法从时间消耗上看，颇为高效，但是在支持滑动窗口的同时，我们也能看到，付出的代价是一个数量级上相当于窗口宽度的空间复杂度，其实它就是这个队列的空间消耗，在这里队列最大长度就是 10000。

如果我们允许队列的长度较大，队列造成的空间消耗和单个处理请求的最大时间消耗就可能会成为问题，我们能优化一下吗？

能。那么这种情况下，一种“妥协”的办法就是，队列中的每个元素，不再是实际请求精确到毫秒的时间戳，而是特定某一秒中包含的请求数目，比如队列的其中一个元素表示 3:00:01 到 3:00:02 之间对应应有 150 个请求。用这种方法，对于上述这个一分钟内流量限制的问题，我们可以把队列长度严格控制在 60（因为是 60 秒），每个元素都表示特定某一秒中的请求数目。当然，这个方法损失的是时间窗口毫秒级的精度。而这，就是我们控制

时间窗口队列的长度所采用的一种较为常见的优化方式，它虽**损失了精度，但却降低了空间复杂度**。

好，规则已经做到严格匹配了，可是在实际应用中，在很多情况下，这还是有问题。为什么呢？

细化控制粒度

这要从流量控制的动机说起，**我们建立流量控制这个系统的目的，是为了避免对于系统的冲击，而无论使用固定窗口，还是滑动窗口，根据当前的规则，我们都只能限定这个一分钟窗口内的流量符合要求，却不能做到更细粒度的控制。**

举个极端的例子，一分钟内这一万个请求，如果均匀地分布在这一分钟的窗口中，系统很可能就不会出问题；但如果这一万个请求，全部集中在最开始的一秒钟内，系统就压垮了，这样的流量控制就没有起到有效的防御作用了。

那好，如果我们要做到系统可以接受的更细的粒度。举例来说，如果我们可以做到按秒控制，那么继续按照 10000 个 / 分钟来计算的话，这个限制就可以换算成不要超过 $10000/60 \approx 167$ 个 / 秒。

漏桶算法

漏桶 (Leaky Bucket) 算法就是可以带来更细粒度控制的限流算法，它的粒度取决于系统所支持的准确最小时间间隔，比如毫秒。

你可以想象一个有缺漏的桶，无论我们怎样往里面放水（发送请求），水都有可能以两种方式从桶中排出来：

从漏口往外流，如果桶中有水，这个流速是一定的（这就是**系统满载时，限流的流速**）；

注水太快，水从桶中溢出（这就是**请求被拒绝了，限流效果产生**）。

另外，由于请求的最小单位是一个，因此桶的大小不得小于 1。我们要求请求发送的速度不得小于漏水的速度，但我们更多时候会设置一定的桶容量，这就意味着系统允许一定程度的富余以应对突发量。这个桶大小，也就是突发量，被称为 burst。

于是，我们每次都可以根据流速以及上一次的流量检测时间，获知在考虑漏水的情况下，如果接纳当前请求，那么桶中将达到怎样的水位，是否会超过 burst。如果不超过，就允许此次访问，反之拒绝。参考代码如下：

 复制代码

```
1 public class RateLimiter {
2     private float leakingRate = 10000f/60/1000; // 每一毫秒能够漏掉的水
3     private float remaining = 0; // 桶中余下的水
4     private float burst = 0; // 桶容量
5     private long lastTime = System.currentTimeMillis(); // 最近一次流量检测时间
6
7     public boolean isAllowed() {
8         long now = System.currentTimeMillis();
9         remaining = Math.max(0, remaining - (now-lastTime)*leakingRate); // 如果
10        lastTime = now;
11
12        if (remaining+1 <= burst) {
13            remaining++;
14            return true;
15        }
16        return false;
17    }
18 }
```

从复杂度上你也可以看到，我们通过变量 remaining 记录每一个请求到达的时刻，桶中水的余量，整个空间复杂度是常量级的。当然了，我们的控制已经不是针对“一分钟规则”了，控制粒度上更加细化，更符合我们对系统保护的实际情况，因此这个方法的应用更广。

令牌桶算法

还有一种和漏桶算法本质上一致，但是实现上有所不同的方法，叫做令牌桶（Token Bucket）算法。说它们实现上不同是因为，漏桶是不断往外漏水，看能不能把陆续到来的请求给消耗掉；而令牌桶呢，则是在令牌桶内会定期放入令牌，每一个请求到来，都要去令牌桶内取令牌，取得了才可以继续访问系统，否则就会被流量控制系统拒绝掉。

就像我们的问题，每 $60 \times 1000 / 10000 = 6$ 毫秒就要向令牌桶内放置一个令牌。和前面的漏桶算法一样，我们并不一定要真的建立一个放入令牌的线程来做这个放入令牌的工作，而是使用和上面类似的算法，在请求到来的时候，根据上次剩余的令牌数和上次之后流逝的时

间，计算当前桶内是否还有完整的一张令牌，如果没有令牌，就拒绝请求，否则允许请求。因此，从这个角度说，漏桶和令牌桶这二者在思想本质上是一致的。

总结思考

今天我通过一个常见的流量控制系统，向你介绍了全栈开发中几个典型的算法，包括基于固定时间窗口的简单计数法，滑动时间窗口的队列法，还有实际应用中更为常见的漏桶算法和令牌桶算法。希望通过今天的学习，你已经理解了它们的工作原理。

现在我来提两个问题吧：

漏桶算法我给出了示例代码，而具有一定相似性的令牌桶算法我没有给出示例代码，如果你理解了这两者，能否写出令牌桶算法的代码呢？

为了简化问题，我在一开始的时候讲了，我们不考虑并发的问题。现在，如果我们把上面无论哪一种算法的代码，改成支持多个线程并发访问的情形，即要求保证线程安全，你觉得需要对代码做怎样的修改呢？

选修课堂：Diffie–Hellman 密钥交换

我们在 [🔗\[第 02 讲\]](#) 中介绍 HTTPS 加密的时候，提到了 Pre-master Secret 生成的方式，其中一种就是 Diffie–Hellman 密钥交换这一算法的变种（如有遗忘，请回看），但是，我们并没有讲其中加密具体的算法原理。那么，下面我就来看一下 Diffie–Hellman 密钥交换，这个常见的 HTTPS 加密算法，是怎样做到**正向计算简单、逆向求解困难**，来保证安全性的。

密钥计算过程

Diffie–Hellman 密钥交换是一种在非保护信道中安全地创建共享密钥方法，它的出现在如今众所周知的 RSA 算法发明之前。现在让我们来玩一个角色扮演游戏，假设你要和我进行通信，我们就来使用这种办法安全地创建共享密钥：

通信的你和我都协议商定了质数 p 和另一个底数 g ；

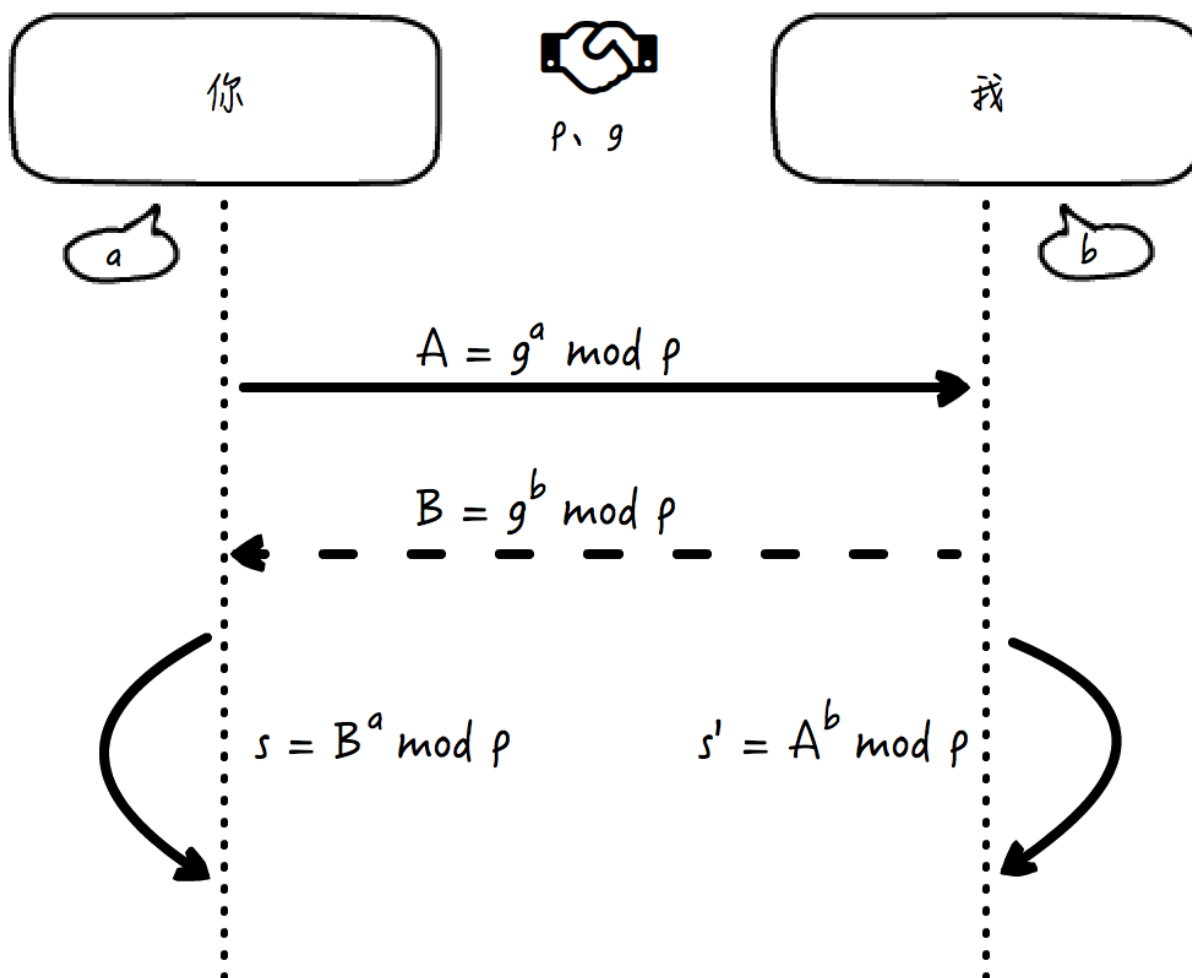
你呢，先生成一个只有你自己知道的随机整数 a ，并将结果 $A = g^a \bmod p$ 发给我；

我呢，也生成一个只有我自己知道的随机整数 b ，并将结果 $B = g^b \bmod p$ 发给你；

你根据我发过来的 B ，计算得到 $s = B^a \bmod p$;

我根据你发过来的 A ，计算得到 $s' = A^b \bmod p$ 。

这个过程用简单的图示来表示就是：



你看，整个过程中，只有 a 、 b 这两个数分别是你和我各自知道并保密的，而其它交换的数据全部都是公开的。对于你来说，已经有了 a ，又得到我传过来的 B ，于是你算出了 s ；对于我来说，已经有了 b ，又得到了你传过来的 A ，于是我算出了 s' 。

有趣的是，经过计算，你得到的 s 和我得到的 s' ，这两个数总是相等的，既然相等，那这个值也就可以用作你我之间通信的对称密钥了。也就是说，**通信双方分别算得了相等的密钥，这也就避免了密钥传递的风险**。可是，为什么 s 和 s' 它们是相等的呢？

质数和模幂运算

因为， g 的 a 次方再 b 次方，等于 g 的 b 次方再 a 次方，即便每次幂运算后加上 p 来取模，也不影响最后结果的相等性，换言之：

$$g^{ab} \bmod p = (g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$

上面这样的，先求幂，再取模的运算，我们把它简单称为“模幂运算”。在实际应用中， g 可以取一个比较小的数，而 a 、 b 和 p ，都要取非常大的数，而且 p 往往会取一个“极大”的质数——因为质数在此会具备这样一个重要性质，模幂运算结果会在小于 p 的非负整数中均匀分布；而另外一个原因是，由于 g 的 a 次方或 b 次方会非常大，需要一个“上限”，一个使得生成的数无论是传输还是存储都能够可行的方法。**因此大质数 p 的取模运算被用来设定上限并将大数化小，且保持原有的逆向求解困难性。**

说到逆向求解的困难性，这是根据数学上 [离散对数](#) 求解的特性所决定的，具体说来，就是这样一个模幂等式：

$$g^a \bmod p = A$$

从难度上看，该式具有如下三个特性：

特性 ①：已知 g 、 a 和 p ，求 A 容易；

特性 ②：已知 g 、 p 和 A ，求 a 困难；

特性 ③：已知 a 、 p 和 A ，求 g 也困难。

正好，Diffie-Hellman 密钥交换利用了其中的特性 ① 和特性 ②。比如 a 是超过 100 位的正整数，而 p 则达到了 300 位，那么在这种情况下，如果有恶意的攻击者，得到了 g 、 p ，截获了 A ，但是他根据这些信息，考虑我们前面介绍的公式 $A = g^a \bmod p$ ，在现有科技能达到的算力下，几乎是无法求解出其中的 a 来的。无法知道 a ，无法进而求得对称密钥 s （因为 s 需要通过 $B^a \bmod p$ 求得），这就起到了加密的作用，这也是 Diffie-Hellman 密钥交换能够实现的原理。

扩展阅读

【基础】文中我提到了算法的时间复杂度和空间复杂度，这是属于算法的基础知识。如果不太熟悉的话可以阅读一下这个 [词条](#)，以及 [这篇文章](#)，而在 [这里](#) 则有常见算法

的时间复杂度列表。

选修课堂中介绍的 Diffie-Hellman 密钥交换利用了模幂公式的“正向计算简单，逆向求解困难”这一特点，这个特点非常重要，还有一个相关的技术 RSA 也利用了这一特点。本来我是把 RSA 加密技术的原理介绍和 Diffie-Hellman 密钥交换放在一起讲述的，但是经过仔细斟酌，我觉得 RSA 涉及到的数学知识稍多，整体理解起来明显偏难，因此为了专栏内容和难度的一致性，我忍痛把它拿出去了，并放在了我自己的博客上，感兴趣的话可以 [移步阅读](#)。



全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 35 | 网站性能优化（下）

精选留言 (2)

写留言



靠人品去赢

2019-12-02

这篇文章，让我联想到秒杀活动的设计，反正就是不让你大流量直接进入到服务器就是了，想了一下所谓的秒杀，真的就是耍猴。

展开 ∨





许童童

2019-12-02

老师的文章写得真好，读起来意犹未尽
展开 ∨

