

不得了解系列之限流



梦朝思夕

[+关注](#)

发布于: 2020 年 08 月 28 日



限流简介

现在说到高可用系统，都会说到高可用的保护手段：缓存、降级和限流，本博文就主要说说限流。限流是流量限速（Rate Limit）的简称，是指只允许指定的事件进入系统，超过的部分将被拒绝服务、排队或等待、降级等处理。对于server服务而言，限流为了保证一部分的请求流量可以得到正常的响应，总好过全部请求都不能得到响应，甚至导致系统雪崩。限流与熔断经常被人弄混，博主认为它们最大的区别在于限流主要在server实现，而熔断主要在client实现，当然了，一个服务既可以充当server也可以充当client，这也是让限流与熔断同时存在一个服务中，这两个概念才容易被混淆。

那为什么需要限流呢？很多人第一反应就是服务扛不住了所以需要限流。这是不全面的说法，博主认为限流是因为资源的稀缺或出于安全防范的目的，采取的自我保护的措施。限流可以保证使用有限的资源提供最大化的服务能力，按照预期流量提供服务，超过的部分将会拒绝服务、排队或等待、降级等处理。

现在的系统对限流的支持各有不同，但是存在一些标准。在HTTP RFC 6585标准中规定了『[429 Too Many Requests](#)』，429状态码表示用户在给定时间内发送了太多的请求，需要进行限流（“速率限制”），同时包含一个Retry-After 响应头用于告诉客户端多长时间后可以再次请求服务。

[复制代码](#)

```
1 HTTP/1.1 429 Too Many Requests
2 Content-Type: text/html
3 Retry-After: 3600
4
5
6
```

```

7      <title>Too Many Requests</title>
8
9
10     <h1>Too Many Requests</h1>
11     <p>I only allow 50 requests per hour to this Web site per
12         logged in user. Try again soon.</p>
13
14

```

很多应用框架同样集成了，限流功能并且在返回的Header中给出明确的限流标识。

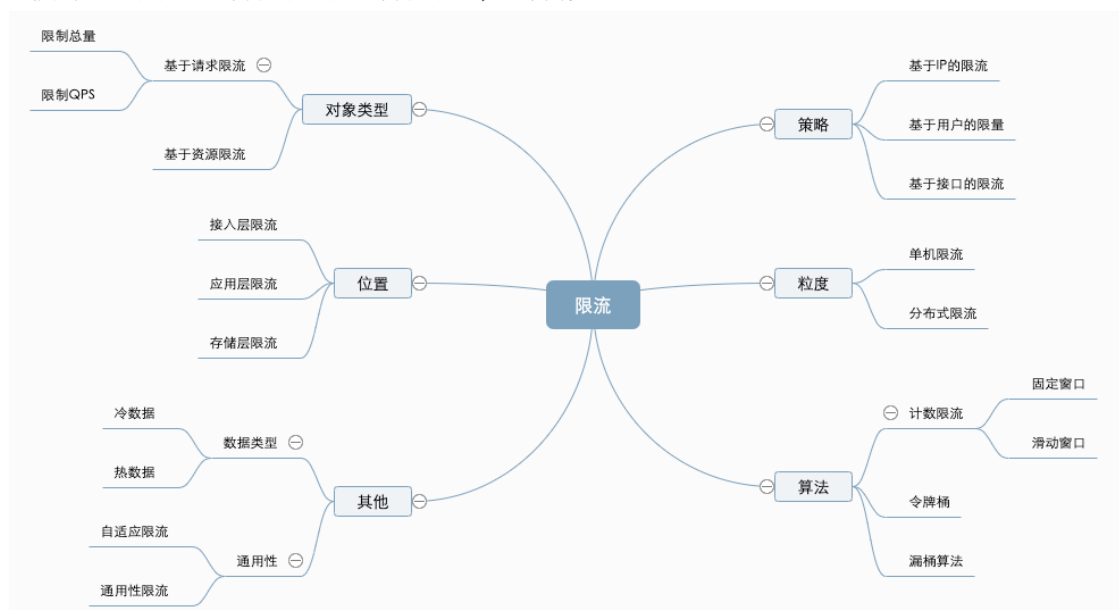
- X-Rate-Limit-Limit：同一个时间段所允许的请求的最大数目；
- X-Rate-Limit-Remaining：在当前时间段内剩余的请求的数量；
- X-Rate-Limit-Reset：为了得到最大请求数所等待的秒数。

这是通过响应头告诉调用方服务端的限流频次是怎样的，保证后端的接口访问上限，客户端也可以根据响应的Header调整请求。

限流分类

限流，拆分来看，就两个字限和流，限就是动词限制，很好理解。但是流在不同的场景之下就是不同资源或指标，多样性就在流中体现。在网络流量中可以是字节流，在数据库中可以是TPS，在API中可以是QPS亦可以是并发请求数，在商品中还可以是库存数... ...但是不管是哪一种『流』，这个流必须可以被量化，可以被度量，可以被观察到、可以统计出来。

我们把限流的分类基于不同的方式分为不同的类别，如下图。



因为篇幅有限，本文只会挑选几个常见的类型分类进行说明。

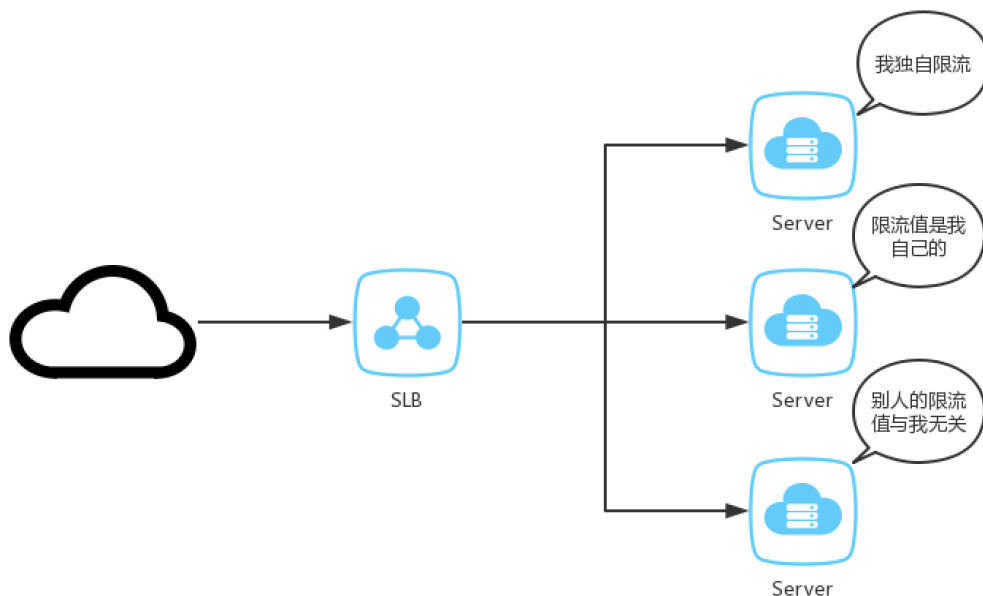
限流粒度分类

根据限流的粒度分类：

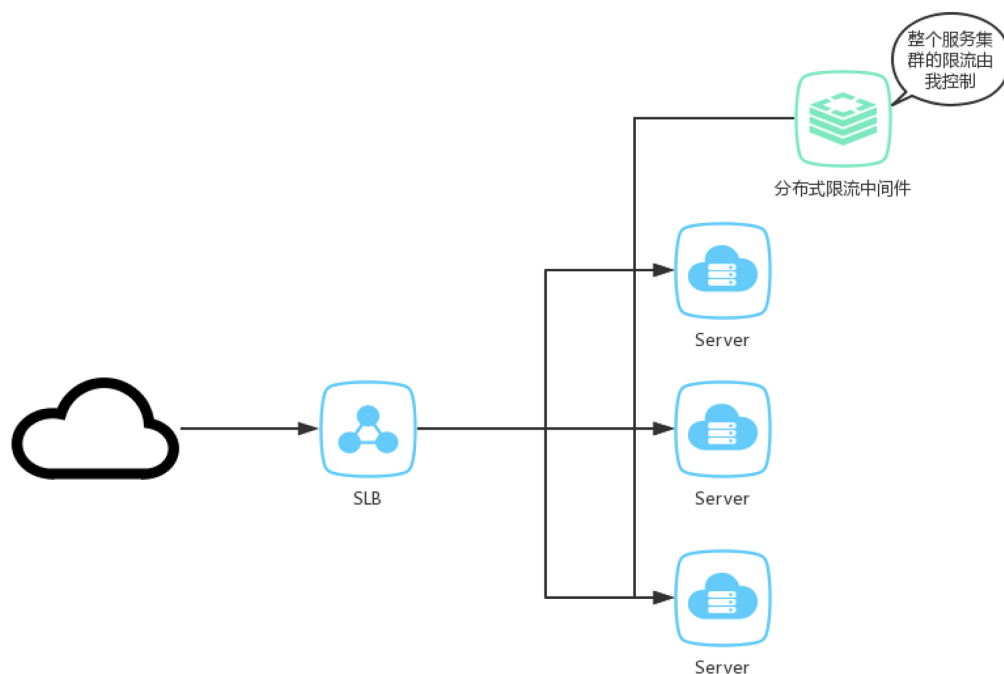
单机限流

分布式限流

现状的系统基本上都是分布式架构，单机的模式已经很少了，这里说的单机限流更加准确一点的说法是单服务节点限流。单机限流是指请求进入到某一个服务节点后超过了限流阈值，服务节点采取了一种限流保护措施。



分布式限流狭义的说法是在接入层实现多节点合并限流，比如NGINX+redis，分布式网关等，广义的分布式限流是多个节点（可以为不同服务节点）有机整合，形成整体的限流服务。



单机限流防止流量压垮服务节点，缺乏对整体流量的感知。分布式限流适合做细粒度不同的限流控制，可以根据场景不同匹配不同的限流规则。与单机限流最大的区别，分布式限流需要中心化存储，常见的使用redis实现。引入了中心化存储，就需要解决以下问题：

数据一致性

在限流模式中理想的模式为时间点一致性。时间点一致性的定义中要求所有数据组件的数据在任意时刻都是完全一致的，但是一般来说信息传播的速度最大是光速，其实并不能达到任意时刻一致，总有一定的时间不一致，对于我们CAP中的一致性来说只要达到读取到最新数据即可，达到这种情况并不需要严格的任意时间一致。这只能是理论当中的一致性模型，可以在限流中达到线性一致性即可。

时间一致性

这里的时间一致性与上述的时间点一致性不一样，这里就是指各个服务节点的时间一致性。一个集群有3台机器，但是在某一个A/B机器的时间为Tue Dec 3 16:29:28 CST 2019，C为Tue Dec 3 16:29:28 CST 2019，那么它们的时间就不一致。那么使用ntpd进行同步也会存在一定的误差，对于时间窗口敏感的算法就是误差点。

超时

在分布式系统中就需要网络进行通信，会存在网络抖动问题，或者分布式限流中间件压力过大导致响应变慢，甚至是超时时间阈值设置不合理，导致应用服务节点超时了，此时是放行流量还是拒绝流量？

性能与可靠性

分布式限流中间件的资源总是有限的，甚至可能是单点的（写入单点），性能存在上限。如果分布式限流中间件不可用时候如何退化为单机限流模式也是一个很好的降级方案。

限流对象类型分类

按照对象类型分类：

基于请求限流

基于资源限流

基于请求限流，一般的实现方式有**限制总量**和**限制QPS**。限制总量就是限制某个指标的上限，比如抢购某一个商品，放量是10w，那么最多只能卖出10w件。微信的抢红包，群里发一个红包拆分为10个，那么最多只能有10人可以抢到，第十一个人打开就会显示『手慢了，红包派完了』。



限制QPS，也是我们常说的限流方式，只要在接口层级进行，某一个接口只允许1秒只能访问100次，那么它的峰值QPS

只能为100。限制QPS的方式最难的点就是如何预估阈值，如何定位阈值，下文中说会到。

基于资源限流是基于服务资源的使用情况进行限制，需要定位到服务的关键资源有哪些，并对其进行限制，如限制TCP连接数、线程数、内存使用量等。限制资源更能有效地反映出服务当前地清理，但与限制QPS类似，面临着如何确认资源的阈值为多少。这个阈值需要不断地调优，不停地实践才可以得到一个较为满意地值。

限流算法分类

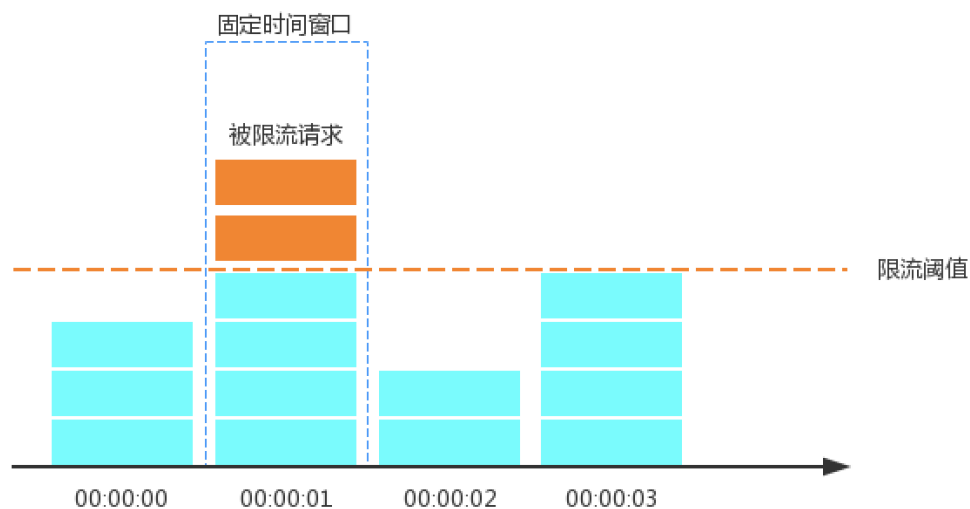
不论是按照什么维度，基于什么方式的分类，其限流的底层均是需要算法来实现。下面介绍实现常见的限流算法：

计数器
令牌桶算法
漏桶算法

计数器

固定窗口计数器


计数限流是最为简单的限流算法，日常开发中，我们说的限流很多都是说固定窗口计数限流算法，比如某一个接口或服务1s最多只能接收1000个请求，那么我们会设置其限流为1000QPS。该算法的实现思路非常简单，维护一个固定单位时间内的计数器，如果检测到单位时间已经过去就重置计数器为零。



其操作步骤：

1. 时间线划分为多个独立且固定大小窗口；
2. 落在每一个时间窗口内的请求就将计数器加1；
3. 如果计数器超过了限流阈值，则后续落在该窗口的请求都会被拒绝。但时间达到下一个时间窗口时，计数器会被重置为0。

下面实现一个简单的代码。

 复制代码

```
1 package limit
2
3 import (
4     "sync/atomic"
5     "time"
6 )
7
8 type Counter struct {
9     Count      uint64 // 初始计数器
10    Limit       uint64 // 单位时间窗口最大请求频次
11    Interval    int64  // 单位ms
12    RefreshTime int64  // 时间窗口
13 }
14
15 func NewCounter(count, limit uint64, interval, rt int64) *Counter {
16     return &Counter{
17         Count:      count,
18         Limit:       limit,
19         Interval:    interval,
20         RefreshTime: rt,
21     }
22 }
23
24 func (c *Counter) RateLimit() bool {
25     now := time.Now().UnixNano() / 1e6
26     if now < (c.RefreshTime + c.Interval) {
27         atomic.AddUint64(&c.Count, 1)
28         return c.Count <= c.Limit
29     } else {
30         c.RefreshTime = now
31         atomic.AddUint64(&c.Count, -c.Count)
32         return true
33     }
34 }
```

测试代码:

 复制代码

```
1 package limit
2
3 import (
4     "fmt"
5     "testing"
6     "time"
7 )
8
9 func Test_Counter(t *testing.T) {
10     counter := NewCounter(0, 5, 100, time.Now().Unix())
11     for i := 0; i < 10; i++ {
12         go func(i int) {
13             for k := 0; k <= 10; k++ {
14                 fmt.Println(counter.RateLimit())
15                 if k%3 == 0 {
16                     time.Sleep(102 * time.Millisecond)
17                 }
18             }
19         }(i)
20     }
21     time.Sleep(10 * time.Second)
22 }
```

看了上面的逻辑, 有没有觉得固定窗口计数器很简单, 对, 就是这么简单, 这就是它的一个优点实现简单。同时也存在两个比较严重缺陷。试想一下, 固定时间窗口1s限流阈值为100, 但是前100ms, 已经请求来了99个, 那么后续的

900ms只能通过一个了，就是一个缺陷，基本上没有应对突发流量的能力。第二个缺陷，在00:00:00这个时间窗口的后500ms，请求通过了100个，在00:00:01这个时间窗口的前500ms还有100个请求通过，对于服务来说相当于1秒内请求量达到了限流阈值的2倍。

滑动窗口计数器

滑动时间窗口算法是对固定时间窗口算法的一种改进，这个词被大众所知实在TCP的流量控制中。固定窗口计数器可以说是滑动窗口计数器的一种特例，滑动窗口的操作步骤：

1. 将单位时间划分为多个区间，一般都是均分为多个小的时间段；
2. 每一个区间内都有一个计数器，有一个请求落在该区间内，则该区间内的计数器就会加一；
3. 每过一个时间段，时间窗口就会往右滑动一格，抛弃最老的一个区间，并纳入新的一个区间；
4. 计算整个时间窗口内的请求总数时会累加所有的时间片段内的计数器，计数总和超过了限制数量，则本窗口内所有的请求都被丢弃。

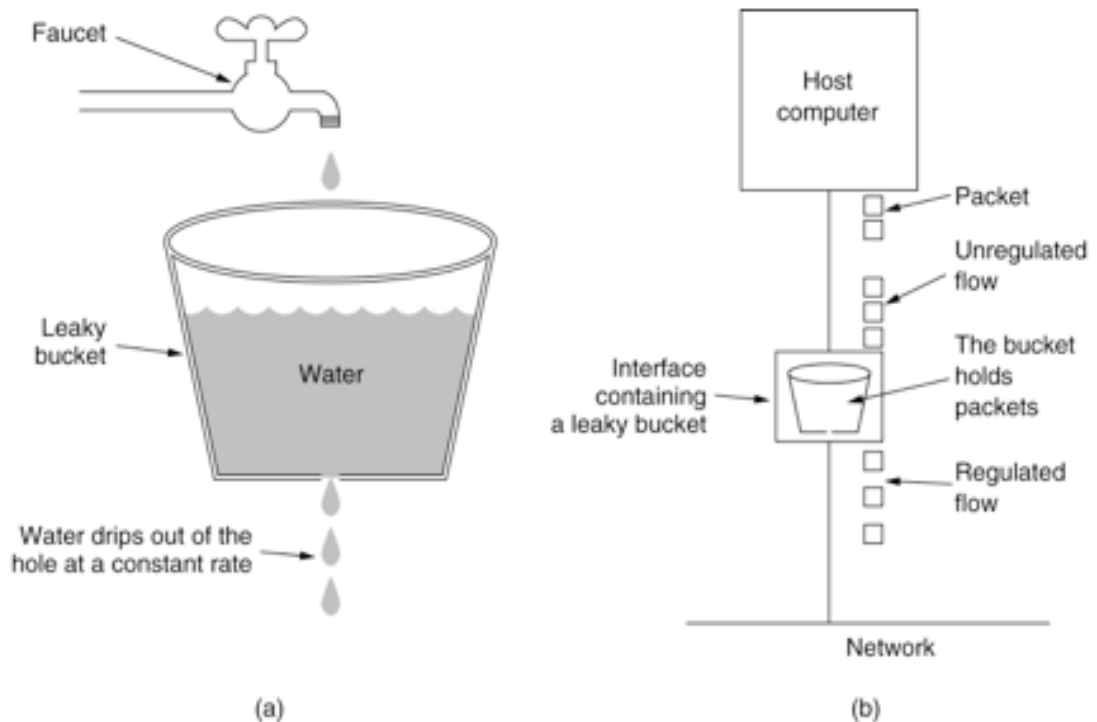
时间窗口划分的越细，并且按照时间"滑动"，这种算法避免了固定窗口计数器出现的上述两个问题。缺点是时间区间的精度越高，算法所需的内存容量就越大。

常见的实现方式主要有基于redis zset的方式和循环队列实现。基于redis zset可将Key为限流标识ID，Value保持唯一，可以用UUID生成，Score 也记为同一时间戳，最好是纳秒级的。使用redis提供的 ZADD、EXPIRE、ZCOUNT 和 zremrangebyscore 来实现，并同时注意开启 Pipeline 来尽可能提升性能。实现很简单，但是缺点就是zset的数据结构会越来越大。

漏桶算法

漏桶算法是水先进入到漏桶里，漏桶再以一定的速率出水，当流入水的数量大于流出水时，多余的水直接溢出。把水换成请求来看，漏桶相当于服务器队列，但请求量大于限流阈值时，多出来的请求就会被拒绝服务。漏桶算法使用队列实现，可以以固定的速率控制流量的访问速度，可以做到流量的“平整化”处理。

大家可以通过网上最流行的一张图来理解。



漏桶算法实现步骤：

1. 将每个请求放入固定大小的队列进行存储；
2. 队列以固定速率向外流出请求，如果队列为空则停止流出；
3. 如队列满了则多余的请求会被直接拒绝。

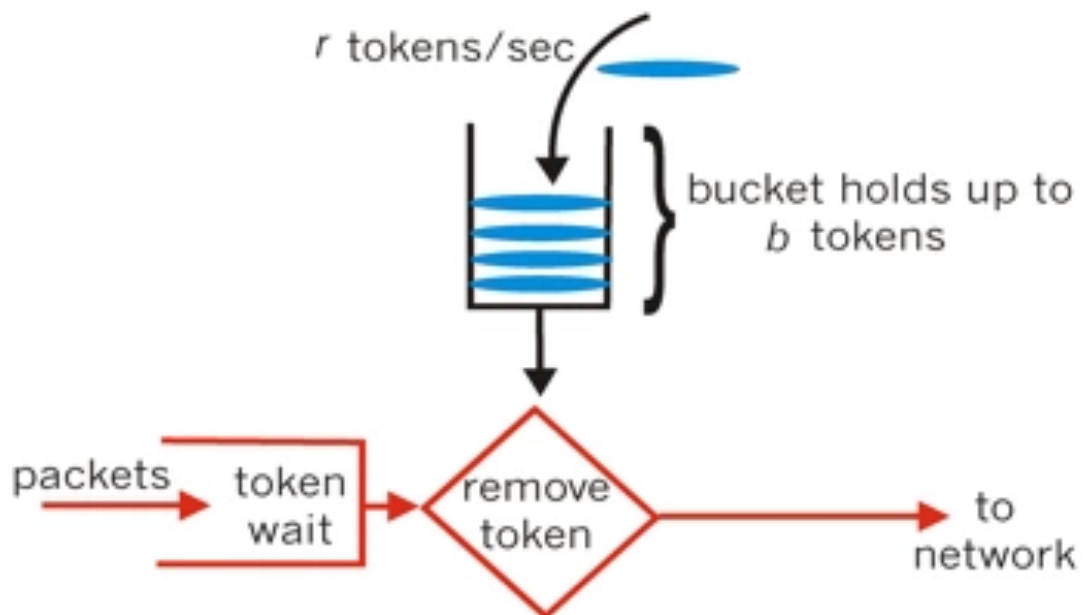
漏桶算法有一个明显的缺陷：当短时间内有大量的突发请求时，即使服务器负载不高，每个请求也都得在队列中等待一段时间才能被响应。

令牌桶算法

令牌桶算法的原理是系统会以一个恒定的速率往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。从原理上看，令牌桶算法和漏桶算法是相反的，前者为“进”，后者为“出”。漏桶算法与令牌桶算法除了“方向”上的不同还有一个更加主要的区别：令牌桶算法限制的是平均流入速率（允许突发请求，只要有足够的令牌，支持一次拿多个令牌），并允许一定程度突发流量；

令牌桶算法的实现步骤：

1. 令牌以固定速率生成并放入到令牌桶中；
2. 如果令牌桶满了则多余的令牌会直接丢弃，当请求到达时，会尝试从令牌桶中取令牌，取到了令牌请求可以执行；
3. 如果桶空了，则拒绝该请求。



四种策略该如何选择？

固定窗口：实现简单，但是过于粗暴，除非情况紧急，为了能快速止损眼前的问题可以作为临时应急的方案。

滑动窗口：限流算法简单易实现，可以应对有少量突增流量场景。

漏桶：对于流量绝对均匀有很强要求，资源的利用率上不是极致，但其宽进严出模式，保护系统的同时还留有部

分余量，是一个通用性方案。

令牌桶：系统经常有突增流量，并尽可能的压榨服务的性能。

怎么做限流？

不论使用上述的哪一种分类或者实现方式，系统都会面临一个共同的问题：如何确认限流阈值。

有人团队根据经验先设定一个小的阈值，后续慢慢进行调整；有的团队是通过进行压力测试后总结出来。这种方式的问题在于压测模型与线上环境不一定一致，接口的单压不能反馈整个系统的状态，全链路压测又难以真实反应实际流量场景流量比例。再换一个思路是通过压测+各应用监控数据。根据系统峰值的QPS与系统资源使用情况，进行等水位放大预估限流阈值，问题在于系统性能拐点未知，单纯的预测不一定准确甚至极大偏离真实场景。

正如《Overload Control for Scaling WeChat Microservices》所说，在具有复杂依赖关系的系统中，对特定服务的进行过载控制可能对整个系统有害或者服务的实现有缺陷。

希望后续可以出现一个更加AI的运行反馈自动设置限流阈值的系统，可以根据当前QPS、资源状态、RT情况等多种关联数据动态地进行过载保护。

不论是哪一种方式给出的限流阈值，系统都应该关注以下几点：

1. 运行指标状态，比如当前服务的QPS、机器资源使用情况、数据库的连接数、线程的并发数等；



go限流类库使用

限流的类库有很多，不同语言的有不同的类库，如大Java的有concurrency-limits、Sentinel、Guava 等，这些类库都有很多的分析和使用方式了，本文主要介绍Golang的限流类库就是Golang的扩展库：

<https://github.com/golang/time/rate>。可以进去语言类库的代码都值得去研读一番，学习过Java的同学是否对AQS的设计之精妙而感叹呢！

`time/rate` 也有其精妙的部分，下面开始进入类库学习阶段。

github.com/golang/time/rate

进行源码分析前的，最应该做的是了解类库的使用方式、使用场景和API。对业务有了初步的了解，阅读代码就可以事半功倍。因为篇幅有限后续的博文在对多个限流类库源码做分析。

类库的API文档：<https://godoc.org/golang.org/x/time/rate>。

`time/rate`类库是基于令牌桶算法实现的限流功能。前面说令牌桶算法的原理是系统会以一个恒定的速率往桶里放入令牌，那么桶就有一个固定的大小，往桶中放入令牌的速率也是恒定的，并且允许突发流量。查看文档发现一个函数：

```
1 func NewLimiter(r Limit, b int) *Limiter
```

[复制代码](#)

`newLimiter`返回一个新的限制器，它允许事件的速率达到`r`，并允许最多突发`b`个令牌。也就是说`Limiter`限制时间的发生频率，但这个桶一开始容量就为`b`，并且装满`b`个令牌（令牌池中最多有`b`个令牌，所以一次最多只能允许`b`个事件发生，一个事件花费掉一个令牌），然后每一个单位时间间隔（默认1s）往桶里放入`r`个令牌。

```
1 limiter := rate.NewLimiter(10, 5)
```

[复制代码](#)


上面的例子表示，令牌桶的容量为5，并且每一秒中就往桶里放入10个令牌。细心的读者都会发现函数`NewLimiter`第一个参数是`Limit`类型，可以看源码就会发现`Limit`实际上就是`float64`的别名。

[复制代码](#)

```
1 // Limit defines the maximum frequency of some events.
2 // Limit is represented as number of events per second.
3 // A zero Limit allows no events.
4 +vne limit float64
```

限流器还可以指定往桶里放入令牌的时间间隔，实现方式如下：

```
1 limiter := rate.NewLimiter(rate.Every(100*time.Millisecond), 5)
```

 复制代码


这两个例子的效果是一样的，使用第一种方式不会出现在每一秒间隔一下子放入10个令牌，也是均匀分散在100ms的间隔放入令牌。

rate.Limiter提供了三类方法用来限速：

Allow/AllowN
Wait/WaitN
Reserve/ReserveN


下面对比这三类限流方式的使用方式和适用场景。先看第一类方法：

```
1 func (lim *Limiter) Allow() bool
2 func (lim *Limiter) AllowN(now time.Time, n int) bool
```

 复制代码

Allow 是AllowN(time.Now(), 1)的简化方法。那么重点就在方法 AllowN上了，API的解释有点抽象，说得云里雾里的，可以看看下面的API文档解释：


```
1 AllowN reports whether n events may happen at time now.
2 Use this method if you intend to drop / skip events that exceed the rate limit.
3 Otherwise use Reserve or Wait.
```

 复制代码


实际上就是为了说，方法 AllowN在指定的时间时是否可以从令牌桶中取出N个令牌。也就意味着可以限定N个事件是否可以在指定的时间同时发生。这个两个方法是无阻塞，也就是说一旦不满足，就会跳过，不会等待令牌数量足够才执行。也就是文档中的第二行解释，如果打算丢失或跳过超出速率限制的时间，那么久请使用该方法。比如使用之前实例化好的限流器，在某一个时刻，服务器同时收到超过了8个请求，如果令牌桶内令牌小于8个，那么这8个请求就会被丢弃。

一个小示例：

```
1 func AllowDemo() {
2     limiter := rate.NewLimiter(rate.Every(200*time.Millisecond), 5)
3     i := 0
4     for {
5         i++
6         if limiter.Allow() {
7             fmt.Println(i, "====Allow====", time.Now())
8         } else {
9             fmt.Println(i, "====Disallow====", time.Now())
10        }
11        time.Sleep(80 * time.Millisecond)
12        if i == 15 {
13            return
14        }
15    }
16 }
```

 复制代码

执行结果：

 复制代码

```

1 1 ====Allow===== 2019-12-14 15:54:09.9852178 +0800 CST m=+0.005998001
2 2 ====Allow===== 2019-12-14 15:54:10.1012231 +0800 CST m=+0.122003301
3 3 ====Allow===== 2019-12-14 15:54:10.1823056 +0800 CST m=+0.203085801
4 4 ====Allow===== 2019-12-14 15:54:10.263238 +0800 CST m=+0.284018201
5 5 ====Allow===== 2019-12-14 15:54:10.344224 +0800 CST m=+0.365004201
6 6 ====Allow===== 2019-12-14 15:54:10.4242458 +0800 CST m=+0.445026001
7 7 ====Allow===== 2019-12-14 15:54:10.5043101 +0800 CST m=+0.525090301
8 8 ====Allow===== 2019-12-14 15:54:10.5852232 +0800 CST m=+0.606003401
9 9 ====Disallow===== 2019-12-14 15:54:10.6662181 +0800 CST m=+0.686998301
10 10 ====Disallow===== 2019-12-14 15:54:10.7462189 +0800 CST m=+0.766999101
11 11 ====Allow===== 2019-12-14 15:54:10.8272182 +0800 CST m=+0.847998401
12 12 ====Disallow===== 2019-12-14 15:54:10.9072192 +0800 CST m=+0.927999401
13 13 ====Allow===== 2019-12-14 15:54:10.9872224 +0800 CST m=+1.008002601
14 14 ====Disallow===== 2019-12-14 15:54:11.0672253 +0800 CST m=+1.088005501
15 15 ====Disallow===== 2019-12-14 15:54:11.1472946 +0800 CST m=+1.168074801

```

第二类方法：因为ReserveN比较复杂，第二类先说WaitN。


 复制代码

```

1 func (lim *Limiter) Wait(ctx context.Context) (err error)
2 func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)

```

类似Wait 是WaitN(ctx, 1)的简化方法。与AllowN不同的是WaitN会阻塞，如果令牌桶内的令牌数不足N个，WaitN会阻塞一段时间，阻塞时间的时长可以用第一个参数ctx进行设置，把 context 实例为context.WithDeadline或context.WithTimeout进行制定阻塞的时长。

 复制代码

```

1 func WaitNDemo() {
2     limiter := rate.NewLimiter(10, 5)
3     i := 0
4     for {
5         i++
6         ctx, canle := context.WithTimeout(context.Background(), 400*time.Millisecond)
7         if i == 6 {
8             // 取消执行
9             canle()
10        }
11        err := limiter.WaitN(ctx, 4)
12
13        if err != nil {
14            fmt.Println(err)
15            continue
16        }
17        fmt.Println(i, ",执行: ", time.Now())
18        if i == 10 {
19            return
20        }
21    }
22 }

```

执行结果：

 复制代码

```

1 1 ,执行: 2019-12-14 15:45:15.538539 +0800 CST m=+0.011023401
2 2 ,执行: 2019-12-14 15:45:15.8395195 +0800 CST m=+0.312003901
3 3 ,执行: 2019-12-14 15:45:16.2396051 +0800 CST m=+0.712089501
4 4 ,执行: 2019-12-14 15:45:16.6395169 +0800 CST m=+1.112001301
5 5 ,执行: 2019-12-14 15:45:17.0385893 +0800 CST m=+1.511073701
6 context canceled
7 7 ,执行: 2019-12-14 15:45:17.440514 +0800 CST m=+1.912998401
8 8 ,执行: 2019-12-14 15:45:17.8405152 +0800 CST m=+2.312999601
9 9 ,执行: 2019-12-14 15:45:18.2405402 +0800 CST m=+2.713024601
10 10 ,执行: 2019-12-14 15:45:18.6405179 +0800 CST m=+3.113002301

```

适用于允许阻塞等待的场景，比如消费消息队列的消息，可以限定最大的消费速率，过大了就会被限流避免消费者负载过高。

第三类方法：

```
1 func (lim *Limiter) Reserve() *Reservation
2 func (lim *Limiter) ReserveN(now time.Time, n int) *Reservation
```

[复制代码](#)

与之前的两类方法不同的是Reserve/ReserveN返回了Reservation实例。Reservation在API文档中有5个方法：

```
1 func (r *Reservation) Cancel() // 相当于CancelAt(time.Now())
2 func (r *Reservation) CancelAt(now time.Time)
3 func (r *Reservation) Delay() time.Duration // 相当于DelayFrom(time.Now())
4 func (r *Reservation) DelayFrom(now time.Time) time.Duration
5 func (r *Reservation) OK() bool
```

[复制代码](#)

通过这5个方法可以让开发者根据业务场景进行操作，相比前两类的自动化，这样的操作显得复杂多了。通过一个小示例来学习Reserve/ReserveN：

```
1 func ReserveNDemo() {
2     limiter := rate.NewLimiter(10, 5)
3     i := 0
4     for {
5         i++
6         reserve := limiter.ReserveN(time.Now(), 4)
7         // 如果为false说明拿不到指定数量的令牌，比如需要的令牌数大于令牌桶容量的场景
8         if !reserve.OK() {
9             return
10        }
11        ts := reserve.Delay()
12        time.Sleep(ts)
13        fmt.Println("执行: ", time.Now(), ts)
14        if i == 10 {
15            return
16        }
17    }
18 }
```

[复制代码](#)

执行结果：

```
1 执行: 2019-12-14 16:22:26.6446468 +0800 CST m=+0.008000201 0s
2 执行: 2019-12-14 16:22:26.9466454 +0800 CST m=+0.309998801 247.999299ms
3 执行: 2019-12-14 16:22:27.3446473 +0800 CST m=+0.708000701 398.001399ms
4 执行: 2019-12-14 16:22:27.7456488 +0800 CST m=+1.109002201 399.999499ms
5 执行: 2019-12-14 16:22:28.1456465 +0800 CST m=+1.508999901 398.997999ms
6 执行: 2019-12-14 16:22:28.5456457 +0800 CST m=+1.908999101 399.0003ms
7 执行: 2019-12-14 16:22:28.9446482 +0800 CST m=+2.308001601 399.001099ms
8 执行: 2019-12-14 16:22:29.3446524 +0800 CST m=+2.708005801 399.998599ms
9 执行: 2019-12-14 16:22:29.7446514 +0800 CST m=+3.108004801 399.9944ms
10 执行: 2019-12-14 16:22:30.1446475 +0800 CST m=+3.508000901 399.9954ms
```

[复制代码](#)

如果在执行Delay()之前操作Cancel()那么返回的时间间隔就会为0，意味着可以立即执行操作，不进行限流。

```
1 func ReserveNDemo2() {
2     limiter := rate.NewLimiter(5, 5)
3     i := 0
4     for {
5         i++
```

[复制代码](#)

```
6         reserve := limiter.ReserveN(time.Now(), 4)
7         // 如果为flase说明拿不到指定数量的令牌, 比如需要的令牌数大于令牌桶容量的场景
8         if !reserve.OK() {
9             return
10        }
11
12        if i == 6 || i == 5 {
13            reserve.Cancel()
14        }
15        ts := reserve.Delay()
16        time.Sleep(ts)
17        fmt.Println(i, "执行: ", time.Now(), ts)
18        if i == 10 {
19            return
20        }
21    }
22 }
```


执行结果:

 复制代码

```
1 1 执行:  2019-12-14 16:25:45.7974857 +0800 CST m=+0.007005901 0s
2 2 执行:  2019-12-14 16:25:46.3985135 +0800 CST m=+0.608033701 552.0048ms
3 3 执行:  2019-12-14 16:25:47.1984796 +0800 CST m=+1.407999801 798.9722ms
4 4 执行:  2019-12-14 16:25:47.9975269 +0800 CST m=+2.207047101 799.0061ms
5 5 执行:  2019-12-14 16:25:48.7994803 +0800 CST m=+3.009000501 799.9588ms
6 6 执行:  2019-12-14 16:25:48.7994803 +0800 CST m=+3.009000501 0s
7 7 执行:  2019-12-14 16:25:48.7994803 +0800 CST m=+3.009000501 0s
8 8 执行:  2019-12-14 16:25:49.5984782 +0800 CST m=+3.807998401 798.0054ms
9 9 执行:  2019-12-14 16:25:50.3984779 +0800 CST m=+4.607998101 799.0075ms
10 10 执行:  2019-12-14 16:25:51.1995131 +0800 CST m=+5.409033301 799.0078ms
```

看到这里time/rate的限流方式已经完成, 除了上述的三类限流方式, time/rate还提供了动态调整限流器参数的功能。

相关API如下:

 复制代码

```
1 func (lim *Limiter) SetBurst(newBurst int) // 相当于SetBurstAt(time.Now(), newBurst).
2 func (lim *Limiter) SetBurstAt(now time.Time, newBurst int) // 重设令牌桶的容量
3 func (lim *Limiter) SetLimit(newLimit Limit) // 相当于SetLimitAt(time.Now(), newLimit)
4 func (lim *Limiter) SetLimitAt(now time.Time, newLimit Limit) // 重设放入令牌的速率
```

这四个方法可以让程序根据自身的状态动态的调整令牌桶速率和令牌桶容量。

结尾

通过上述一系列讲解, 相信大家对各有限流的应用场景和优缺点也有了大致的掌握, 希望在日常开发中有所帮助。限流仅仅是整个服务治理中的一个小环节, 需要与多种技术结合使用, 才可以更好的提升服务的稳定性的同时提高用户体验。

附录

<https://github.com/uber-go/ratelimit>
https://en.wikipedia.org/wiki/Token_bucket
<https://www.cs.columbia.edu/~ruigu/papers/socc18-final100.pdf>
<https://github.com/alibaba/Sentinel>
<https://tools.ietf.org/html/rfc6585>
<https://www.yiichina.com/doc/guide/2.0/rest-rate-limiting>
<https://github.com/RussellLuo/slidingwindow>

<http://zim.logdown.com/posts/300977-distributed-rate-limiter>

<https://www.yuque.com/clip/dev-wiki/axo1wb?language=en-us>

发布于: 2020 年 08 月 28 日 | 阅读数: 721

版权声明: 本文为 InfoQ 作者【梦朝思夕】的原创文章。

原文链接:【<https://xie.infoq.cn/article/92d04bbf51ec2b18d1371dcc3>】。文章转载请联系作者。

限流



梦朝思夕

还未添加个人签名 2013.11.15 加入

还未添加个人简介

+ 关注

👍 点赞 | ★ 收藏 | 微信 | 微博 | 部落 | 举报

评论

快抢沙发! 虚位以待

发布

暂无评论



促进软件开发及相关领域知识与创新的传播

商务专区

AWS 技术专区 Intel 精彩芯体验 Google Cloud
华为云5G+X联创营 华为云 DevRun 专区 Intel AI
腾讯Techo开发者大会 百度技术沙龙
T11数据智能峰会 迅雷链技术专区
OPPO技术开放日 云+社区开发者大会
百度AI 开放平台 华为云 MeetUp
华为昇腾技术沙龙 DevRun鲲鹏开发者沙龙2020

InfoQ

关于我们
我要投稿
合作伙伴
加入我们
关注我们

联系我们

内容投稿: editors@geekbang.com
业务合作: hezuo@geekbang.com
反馈投诉: feedback@geekbang.com
加入我们: zhaopin@geekbang.com
联系电话: 010-64738142
地址: 北京市朝阳区叶青大厦北园

InfoQ 近期会议

深圳 全球架构师峰会 09月11-12日
上海 全球人工智能与机器学习技术大会 09月24-25日
北京 全球软件开发大会 10月15-17日
北京 全球大前端技术大会 11月24-25日