弹力设计篇之"限流设计"

2018-03-20 陈皓

弹力设计篇之"限流设计"
陈皓
- 00:00 / 14:16

保护系统不会在过载的情况下导致问题，那么，我们就需要限流。

我们在一些系统中都可以看到这样的设计，比如，我们的数据库访问的连接池，还有我们的线程池，还有nginx下的用于限制瞬时并发连接数的limit_conn模块，限制每秒平均速率的limit_req模块，还有限制MQ的生产速，等等。

限流的策略

限流的目的是通过对并发访问进行限速，相关的策略一般是，一旦达到限制的速率，那么就会触发相应的限流行为。一般来说，触发的限流行为如下。

- 拒绝服务。把多出来的请求拒绝掉。一般来说，好的限流系统在受到流量暴增时，会统计当前哪个客户端来的请求最多，直接拒掉这个客户端，这种行为可以把一些不正常的或者是带有恶意的高并发访问抵挡掉。
- 服务降级。关闭或是把后端服务做降级处理。这样可以让服务有足够的资源来处理更多的请求。降级有很多方式，一种是把一些不重要的服务给停掉，把CPU、内存或是数据的资源让给更重要的功能；一种是不再返回全量数据，只返回部分数据。

因为全量数据需要做SQL Join操作，部分的数据则不需要，所以可以让SQL执行更快，还有最快的一种是直接返回预设的缓存，以牺牲一致性的方式来获得更大的性能吞吐。

- 特权请求。所谓特权请求的意思是，资源不够了，我只能把有限的资源分给重要的用户，比如：分给权利更高的VIP用户。在多租户系统下，限流的时候应该保大客户的，所以大客户有特权可以优先处理，而其它的非特权用户就得让路了。
- 延时处理。在这种情况下，一般会有一个队列来缓冲大量的请求，这个队列如果满了，那么就只能拒绝用户了，如果这个队列中的任务超时了，也要返回系统繁忙的错误了。使用缓冲队列只是为了减缓压力，一般用于应对短暂的峰刺请求。
- 弹性伸缩。动用自动化运维的方式对相应的服务做自动化的伸缩。这个需要一个应用性能的监控系统，能够感知到目前最繁忙的TOP 5的服务是哪几个。

然后去伸缩它们，还需要一个自动化的发布、部署和服务注册的运维系统，而且还要快，越快越好。否则，系统会被压死掉了。当然，如果是数据库的压力过大，弹性伸缩应用是没什么用的，这个时候还是应该限流。

限流的实现方式

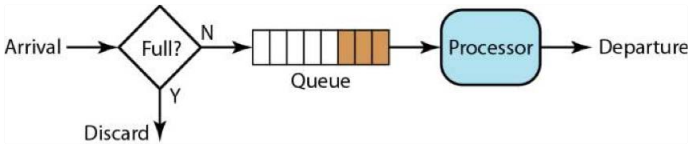
计数器方式

最简单的限流算法就是维护一个计数器Counter，当一个请求来时，就做加一操作，当一个请求处理完后就做减一操作。如果这个Counter大于某个数了（我们设定的限流阈值），那么就开始拒绝请求以保护系统的负载了。

这个算法足够的简单粗暴了。

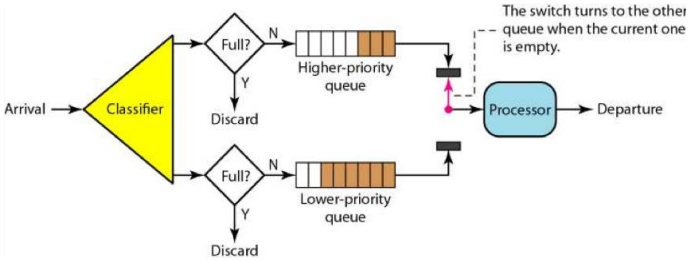
队列算法

在这个算法下，请求的速度可以是波动的，而处理的速度则是非常均速的。这个算法其实有点像一个FIFO的算法。



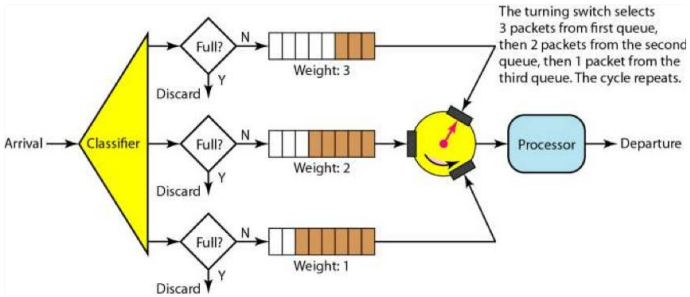
在上面这个FIFO的队列上，我们可以扩展出一些别的玩法。

一个是有优先级的队列，处理时先处理高优先级的队列，然后再处理低优先级的队列。 如下图所示，只有高优先级的队列被处理完成后，才会处理低优先级的队列。



有优先级的队列可能会导致低优先级队列长时间得不到处理。为了避免低优先级的队列被饿死，一般来说是分配不同的比例的处理时间到不同的队列上，于是我们有了带权重的队列。

如下图所示。有三个队列的权重分布是3:2:1，这意味着我们需要在权重为3的这个队列上处理3个请求后，再去权重为2的队列上处理2个请求，最后再去权重为1的队列上处理1个请求，如此返复。



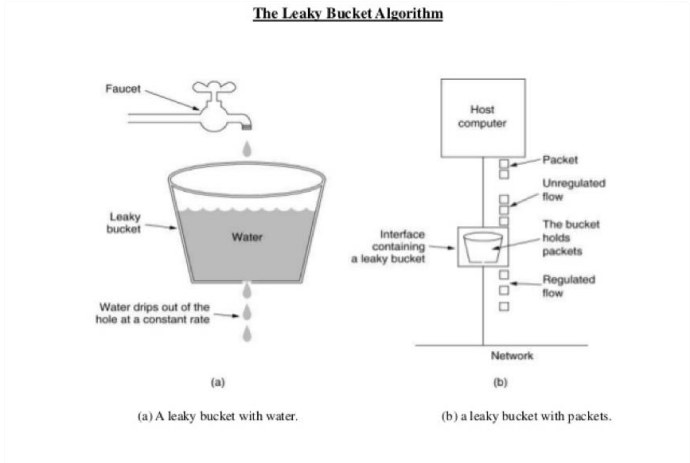
队列流控是以队列的的方式来处理请求。如果处理过慢，那么就会导致队列满，而开始触发限流。

但是，这样的算法需要用队列长度来控制流量，在配置上比较难以操作。如果队列过长，导致后端服务在队列没有满时就挂掉了。一般来说，这样的模型不能做push的，而是要做pull方式的会好一些。

漏斗算法 Leaky Bucket

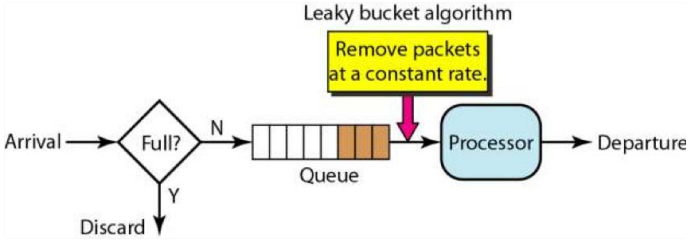
漏斗算法可以参看Wikipedia的相关词条 Leaky Bucket。

下图是一个漏斗算法的示意图 。



我们可以看到，就像一个漏斗一样，进来的水量就好像访问流量一样，而出去的水量就像是我们的系统处理请求一样。当访问流量过大时这个漏斗中就会积水，如果水太多了就会溢出。

一般来说，这个“漏斗”是用一个队列来实现的，当请求过多时，队列就会开始积压请求，如果队列满了，就会开拒绝请求。很多系统都有这样的设计，比如TCP。当请求的数量过多时，就会有一个sync backlog的队列来缓冲请求，或是TCP的滑动窗口也是用于流控的队列。

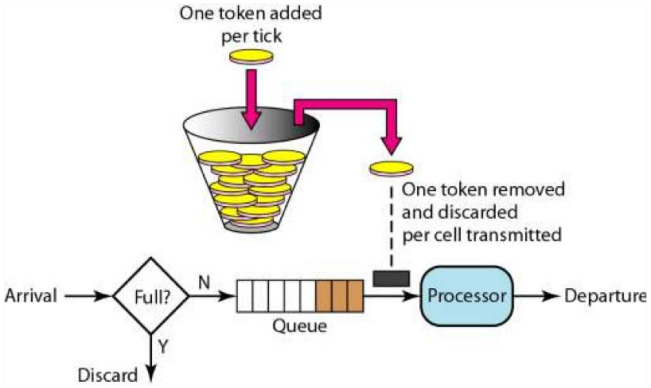


我们可以看到，漏斗算法其实就是在队列请求中加上一个限流器，来让Processor以一个均匀的速度处理请求。

令牌桶算法Token Bucket

关于令牌桶算法，主要是有一个中间人。在一个桶内按照一定的速率放入一些token，然后，处理程序要处理请求时，需要拿到token，才能处理；如果拿不到，则不处理。

下面这个图很清楚地说明了这个算法。



从理论上来说，令牌桶的算法和漏斗算法不一样的是，漏斗算法中，处理请求是以一个常量和恒定的速度处理的，而令牌桶算法则是在流量小的时候“攒钱”，流量大的时候，可以快速处理。

然而，我们可能会问，Processor的处理速度因为有队列的存在，所以其总是能以最大处理能力来处理请求的，这也是我们所希望的方式。因此，令牌桶和漏斗都是受制于Processor的最大处理能力的。无论令牌桶里有多少令牌，也无论队列中还有多少请求。总之，Processor在大流量来临时总是按照自己最大的处理能力来处理的。

但是，试想一下，如果我们的Processor只是一个非常简单的任务分配器，比如像Nginx这样的基本没有什么业务逻辑的网关，那么它的处理速度一定很快，不会有什么瓶颈，而其用来把请求转发给后端服务，那么在这种情况下，这两个算法就有不一样的情况了。

漏斗算法会以一个稳定的速度转发，而令牌桶算法平时流量不大时在“攒钱”，流量大时，可以一次发出队列里有的请求，而后就受到令牌桶的流控限制。

另外，令牌桶还可能做成第三方的一个服务，这样可以在分布式的系统中对全局进行流控，这也是一个很好的方式。

基于响应时间的动态限流

上面的算法有个不好的地方，就是需要设置一个确定的限流值。这就要求我们每次发布服务时都做相应的性能测试，找到系统最大的性能值。

当然，性能测试并不是很容易做的。有关性能测试的方法请参看我在CoolShell上的这篇文章《性能测试应该怎么做》。虽然性能测试比较不容易，但是还是应该要做的。

然而，在很多时候，我们却并不知道这个限流值，或是很难给出一个合适的值。其基本会有如下的一些因素：

- 实际情况下，很多服务会依赖于数据库。所以，不同的用户请求，会对不同的数据集进行操作。就算是相同的请求，可能数据集也不一样（比如，现在很多应用都会有一个时间线Feed流，不同的用户关心的主题人人不一样，数据也不一样）。

而且数据库的数据是在不断地变化的，可能前两天性能还行，因为数据量增加导致性能变差。在这种情况下，我们很难给出一个确定的一成不变的值，因为关系型数据库对于同一条SQL语句的执行时间其实是不可预测的（NoSQL的就比RDBMS的可预测性要好）。

- 不同的API有不同的性能。我们要在线上为每一个API配置不同的限流值，这点太难配置，也很难管理。
- 而且，现在的服务都是能自动化伸缩的，不同大小的集群的性能也不一样，所以，在自动化伸缩的情况下，我们要动态地调整限流的阈值，这点太难做到了。

基于上述这些原因，我们的限流的值是很难被静态地设置成恒定的一个值。

我们想使用一种动态限流的方式。这种方式，不再设定一个特定的流控值，而是能够动态地感知系统的压力来自动化地限流。

这方面设计的典范是TCP协议的拥塞控制的算法。TCP使用RTT - Round Trip Time 来探测网络的延时和性能，从而设定相应的“滑动窗口”的大小，以让发送的速率和网络的性能相匹配。这个算法是非常精妙的，我们完全可以借鉴在我们的流控技术中。

我们记录下每次调用后端请求的响应时间，然后在一个时间区间内（比如，过去10秒）的请求计算一个响应时间的P90或P99值，也就是把过去10秒内的请求的响应时间排个序，然后看90%或99%的位置是多少。

这样，我们就知道有多少请求大于某个响应时间。如果这个P90或P99超过我们设定的阈值，那么我们就自动限流。

这个设计中有几个要点。

- 你需要计算的一定时间内的P90或P99。在有大量请求的情况下，这个非常地耗内存也非常地耗CPU，因为需要对大量的数据进行排序。

解决方案有两种，一种是不记录所有的请求，采样就好了，另一种是使用一个叫蓄水池的近似算法。关于这个算法这里我不就多说了，《编程珠玑》里讲过这个算法，你也可以自行Google，英文叫 Reservoir Sampling。

- 这种动态流控需要像TCP那样，你需要记录一个当前的QPS.如果发现后端的P90/P99响应太慢，那么就可以把这个QPS减半，然后像TCP一样走慢启动的方式，直接到又开始变慢，然后减去1/4的QPS，再慢启动，然后再减去1/8的QPS.....

这个过程有点像个阻尼运行的过程，然后整个限流的流量会在一个值上下做小幅振动。这么做的目的是，如果后端扩容伸缩后性能变好，系统会自动适应后端的最大性能。

- 这种动态限流的方式实现起来并不容易。大家可以看一下TCP的算法。TCP相关的一些算法，我写在了CoolShell上的《TCP的那些事（下）》这篇文章中。你可以用来做参考来实现。

我在现在创业中的Ease Gateway的产品中实现了这个算法。

限流的设计要点

限流主要是有四个目的。

1. 为了向用户承诺SLA。我们保证我们的系统在某个速度下的响应时间以及可用性。
2. 同时，也可以用来阻止在多租户的情况下，某一用户把资源耗尽而让所有的用户都无法访问的问题。
3. 为了应对突发的流量。
4. 节约成本。我们不会为了一个不常见的尖峰来把我们的系统扩容到最大的尺寸。而是在有限的资源下能够承受比较高的流量。

在设计上，我们还要有以下的考量。

- 限流应该是在架构的早期考虑。当架构形成后，限流不是很容易加入。
- 限流模块必需是非常好的性能，而且对流量的变化也是非常灵敏的，否则太过迟钝的限流，系统早因为过载而挂掉了。
- 限流应该有个手动的开关，这样在应急的时候，可以手动操作。
- 当限流发生时，应该有个监控事件通知。让我们知道有限流事件发生，这样，运维人员可以及时跟进。而且还可以自动化触发扩容或降级，以缓解系统压力。
- 当限流发生时，对于拒掉的请求，我们应该返回一个特定的限流错误码。这样，可以和其它错误区分开来。而客户端看到限流，可以调整发送速度，或是走重试机制。
- 限流应该让后端的服务感知到。限流发生时，我们应该在协议头中塞进一个标识，比如HTTP Header中，放入一个限流的级别，告诉后端服务目前正在限流中。这样，后端服务可以根据这个标识决定是否做降级。

小结

好了，我们来总结一下今天分享的主要内容。首先，限流的目的是为了保护系统不在过载的情况下导致问题。接着讲了几种限流的策略。然后讲了，限流的算法，包括计数器、队列、漏斗和令牌桶。然后讨论了如何基于响应时间来限流。最后，我总结了限流设计的要点。下篇文章中，我们讲述降级设计。希望对你有帮助。

也欢迎你分享一下你实现过怎样的限流机制？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
  - 认识故障和弹力设计
  - 隔离设计Bulkheads
  - 异步通讯设计Asynchronous
  - 幂等性设计Idempotency
  - 服务的状态State
  - 补偿事务Compensating Transaction
  - 重试设计Retry
  - 熔断设计Circuit Breaker
  - 限流设计Throttle
  - 降级设计degradation
  - 弹力设计总结
- 管理设计篇
  - 分布式锁Distributed Lock
  - 配置中心Configuration Management
  - 边车模式Sidecar
  - 服务网格Service Mesh
  - 网关模式Gateway
  - 部署升级策略
- 性能设计篇
  - 缓存Cache
  - 异步处理Asynchronous
  - 数据库扩展
  - 秒杀Flash Sales
  - 边缘计算Edge Computing

松花皮蛋me

2018-03-20

老师好，请问下微信红包退还是怎么设计的，如果使用redis过期通知，订阅者下线再连接期间过期的信息不过再通知

LI

2018-05-25

文章都很好，就是缺少代码落地，看起来很理论

张乐乐

2018-04-12

我们动态限流一般是根据资源来进行的，CPU/内存/带宽，存储对于出流部件可以转为带宽。根据响应时间来限流这个想起来比较难实施，后面再研究下。
此外，限流也可以考虑做多级，不同阶段不同的阈值限制，分层去限制，比如操作体统OS层，链接处理，业务处理。
也需要，考虑针对异常用户的识别限制，很多时候，一个异常用户带来的影响会特别大。

华烬

2018-03-21

期待秒杀的文章，不过好像要等挺久的

作者回复

2018-03-22

是的，中间还会有区块链加塞

shufang

2018-03-20

限流看着怎么有点像熔断?

作者回复

2018-03-24

我反而觉得熔断是限流的一种��

jackwoo

2018-06-16

计数器公司有用过，实现起来比较简单

A圆规

2018-05-31

最好有代码落地，有点书本。

kuzan

2018-05-30

读两遍，感觉自己之前还是肤浅了

李志博

2018-04-12

期待加防刷设计

kingeasternsun

2018-04-09

文章中一开始提到的limit-req指?

权乐观

2018-04-01

感觉漏斗是最弱鸡的队列啊