限流，永远都不是一件简单的事！

作者： 咖啡拿铁 2020-11-26 06:48:51

网络 通信技术

随着微服务的流行，服务之间的稳定性变得越发重要，往往我们会花很多经历在维护服务的稳定性上，限流和熔断降级是我们最常用的两个手段。前段时间在群里有些小伙伴对限流的使用些疑问，再加上最近公司大促也做了限流相关的事，所以在这里总结一下写写自己对限流的一些看法。

背景

随着微服务的流行，服务之间的稳定性变得越发重要，往往我们会花很多经历在维护服务的稳定性上，限流和熔断降级是我们最常用的两个手段。前段时间在群里有些小伙伴对限流的使用些疑问，再加上最近公司大促也做了限流相关的事，所以在这里总结一下写写自己对限流的一些看法。

刚才说了限流是我们保证服务稳定性的手段之一，但是他并不是所有场景的稳定性都能保证，和他名字一样他只能在大流量或者突发流量的场景下才能发挥出自己的作用。比如我们的系统最高支持100QPS，但是突然有1000QPS请求打了进来，可能这个时候系统就会直接挂掉，导致后面一个请求都处理不了，但是如果我们有限流的手段，无论他有多大的QPS，我们都只处理100QPS的请求，其他请求都直接拒绝掉，虽然有900的QPS的请求我们拒绝掉了，但是我们的系统没有挂掉，我们系统仍然可以不断的处理后续的请求，这个是我们所期望的。有同学可能会说，现在都上的云了，服务的动态伸缩应该是特别简单的吧，如果我们发现流量特别大的时候，自动扩容机器到可以支撑目标QPS那不就不需要限流了吗?其实有这个想法的同学应该还挺多的，有些同学可能被一些吹牛的文章给唬到了，所以才会这么想，这个想法在特别理想化的时候是可以实现的，但是在现实中其实有下面几个问题：

- 扩容是需要时间。扩容简单来说就是搞一个新的机器，然后重新发布代码，做java的同学应该是知道发布成功一个代码的时间一般不是以秒级计算，而是以分钟级别计算，有时候你扩容完成，说不定流量尖峰都过去了。
- 扩容到多少是个特别复杂的问题。扩容几台机器这个是比较复杂的，需要大量的压测计算，以及整条链路上的一个扩容，如果扩容了你这边的机器之后，其他团队的机器没有扩容可能最后还是有瓶颈这个也是一个问题。

所以单纯的扩容是解决不了这个问题的，限流仍然是我们必须掌握的技能！

基本原理

计数器算法

相似话题

网络设备
623内容

网络优化
560内容

4G/5G
1100内容

网络管理

AISummit人工智能大会

全部话题 >>

同话题下的热门内容

用了TCP协议，就一定不会丢包

如何设计一个分布式 ID 发号器

曾经5亿用户用过的社交App即

面试突击：GET 和 POST 有什

我有七种实现Web实时消息推送

一文带你弄懂 CDN 的技术原理

Session 和 Cookies 有什么区别

异频SMTC专题研究分析

编辑推荐

FTP与SFTP两者有什么区别

你的手机支持5Gwifi吗？5G上

VXLAN与EVPN的结合使用

VXLAN技术介绍：三层的网络网络

什么是通信原理？原来这么简单

相关专题

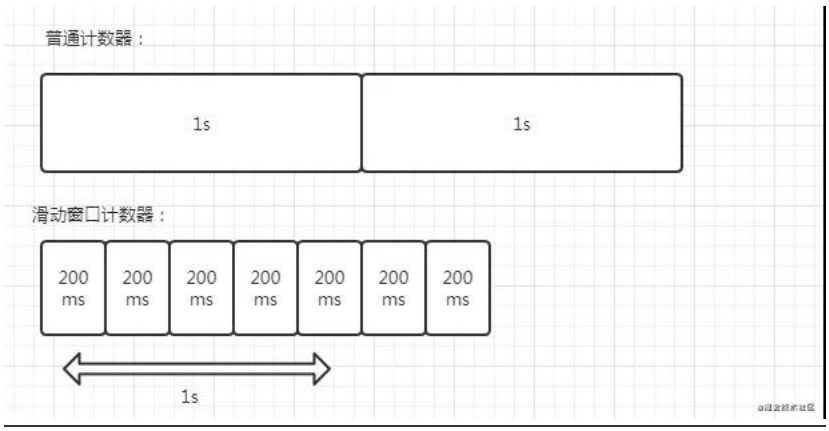
```
2.      long lastTime = System.currentTimeMillis();
3.      int maxQps = 100;
4.      Object lock = new Object();
5.      boolean check(){
6.          synchronized (lock){
7.              long now = System.currentTimeMillis();
8.              if (now - lastTime > 1000){
9.                  lastTime = now;
10.                 curQps = 0;
11.             }
12.             curQps++;
13.             if (curQps > maxQps){
14.                 return false;
15.             }
16.         }
17.         return true;
18.     }
```

这个代码比较简单，我们定义了当前的qps,以及上一次刷新累加变量的时间，还有我们的最大qps和我们的lock锁，我们每次检查的时候，都需要判断是否需要刷新，如果需要刷新那么需要把时间和qps都进行重置，然后再进行qps的累加判断。

这个算法因为太简单了所以带来的问题也是特别明显，如果我们最大的qps是100，在0.99秒的时候来了100个请求，然后在1.01秒的时候又来了100个请求，这个是可以通过我们的程序的，但是我们其实在0.03秒之内通过了200个请求，这个肯定不符合我们的预期，因为很有可能这200个请求直接就会将我们机器给打挂。

滑动窗口计数器

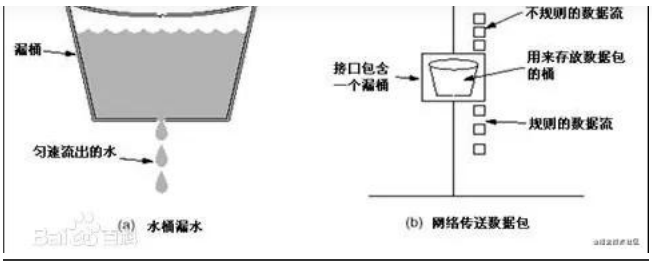
为了解决上面的临界的问题，我们这里可以使用滑动窗口来解决这个问题：



如上图所示，我们将1s的普通计数器，分成了5个200ms，我们统计的当前qps都需要统计最近的5个窗口的所有qps，再回到刚才的问题，0.99秒和1.01秒其实都在我们的最近5个窗口之内，所以这里不会出现刚才的临界的突刺问题。

其实换个角度想，我们普通的计数器其实就是窗口数量为1的滑动窗口计数器，只要我们分的窗口越多，我们使用计数器方案的时候统计就会越精确，但是相对来说维护的窗口的成本就会增加，等会我们介绍sentinel的时候会详细介绍他是怎么实现滑动窗口计数的。

漏斗算法

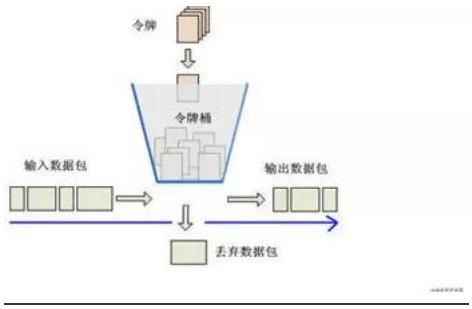


在漏斗算法中我们需要关注漏桶和匀速流出，不论流量有多大都会先到漏桶中，然后以均匀的速度流出。如何在代码中实现这个匀速呢?比如我们想让匀速为100q/s，那么我们可以得到每流出一个流量需要消耗10ms，类似一个队列，每隔10ms从队列头部取出流量进行放行，而我们的队列也就是漏桶，当流量大于队列的长度的时候，我们就可以拒绝超出的部分。

漏斗算法同样的也有一定的缺点：无法应对突发流量(和上面的临界突刺不一样，不要混淆)。比如一瞬间来了100个请求，在漏桶算法中只能一个一个的过去，当最后一个请求流出的时候时间已经过了一秒了，所以漏斗算法比较适合请求到达比较均匀，需要严格控制请求速率的场景。

令牌桶算法

为了解决突发流量情况，我们可以使用令牌桶算法，如下图所示：



这个图上需要关注三个阶段：

- 生产令牌：我们在这里同样的还是假设最大qps是100，那么我们从漏斗的每10ms过一个流量转化成每10ms生产一个令牌，直到达到最大令牌。
- 消耗令牌：我们每一个流量都会消耗令牌桶，这里的消耗的规则可以多变，既可以是简单的每个流量消耗一个令牌，又可以是根据不同的流量数据包大小或者流量类型来进行不同的消耗规则，比如查询的流量消耗1个令牌，写入的流量消耗2个令牌。
- 判断是否通过：如果令牌桶足够那么我们就允许流量通过，如果不足够可以等待或者直接拒绝，这个就可以采用漏斗那种用队列来控制。

单机限流

上面我们已经介绍了限流的一些基本算法，我们把这些算法应用到我们的分布式服务中又可以分为两种，一个是单机限流，一个是集群限流。单机限流指的是每台机器各自做自己的限流，互不影响。我们接下来看看单机限流怎么去实现呢?

guava

guava是谷歌开源的java核心工具库，里面包括集合，缓存，并发等好用的工具，当然也提供了我们这里所需要的的限流的工具，核心类就是RateLimiter。

```
1.  //  RateLimiter rateLimiter = RateLimiter.create(100, 500, TimeUnit.MILLISECONDS); 预热的rateL...
2.      RateLimiter rateLimiter = RateLimiter.create(100); // 简单的rateLimit
3.      boolean limitResult = rateLimiter.tryAcquire();
```

复制

```
@VisibleForTesting
static RateLimiter create(double permitsPerSecond, SleepingStopwatch stopwatch) {
  RateLimiter rateLimiter = new SmoothBursty(stopwatch, maxBurstSeconds: 1.0 /* maxBurstSeconds */);
  rateLimiter.setRate(permitsPerSecond);
  return rateLimiter;
}
```

普通的令牌桶创建了一个SmoothBursty的类,这个类也就是我们实现限流的关键，具体怎么做限流的在我们的tryAcquire中：

```
public boolean tryAcquire(int permits, long timeout, TimeUnit unit) {
  long timeoutMicros = max(unit.toMicros(timeout), 0);
  checkPermits(permits);
  long microsToWait;
  synchronized (mutex()) {                                    1
    long nowMicros = stopwatch.readMicros();
    if (!canAcquire(nowMicros, timeoutMicros)) {              2
      return false;
    } else {
      microsToWait = reserveAndGetWaitLength(permits, nowMicros);   3
    }
  }
  stopwatch.sleepMicrosUninterruptibly(microsToWait);         4
  return true;
}
```

这里分为四步：

- Step1: 加上一个同步锁，需要注意一下这里在sentinel中并没有加锁这个环节，在guava中是有这个的，后续也会将sentinel的一些问题。
- Step2: 判断是否能申请令牌桶，如果桶内没有足够的令牌并且等待时间超过我们的timeout，这里我们就不进行申请了。
- Step3: 申请令牌并获取等待时间，在我们tryAcquire中的timeout参数就是就是我们的最大等待时间，如果我们只是调用tryAcquire(),不会出现等待，第二步的时候已经快速失败了。
- Step4: sleep等待的时间。

扣除令牌的方法具体在reserverEarliestAvailable方法中：

```
@Override
final long reserveEarliestAvailable(int requiredPermits, long nowMicros) {
  resync(nowMicros);
  long returnValue = nextFreeTicketMicros;
  double storedPermitsToSpend = min(requiredPermits, this.storedPermits);
  double freshPermits = requiredPermits - storedPermitsToSpend;
  long waitMicros =
      storedPermitsToWaitTime(this.storedPermits, storedPermitsToSpend)
          + (long) (freshPermits * stableIntervalMicros);

  this.nextFreeTicketMicros = LongMath.saturatedAdd(nextFreeTicketMicros, waitMicros);
  this.storedPermits -= storedPermitsToSpend;
  return returnValue;
}
```

这里虽然看起来过程比较多，但是如果我们只是调用tryAcquire()，就只需要关注两个红框：

- Step1: 根据当前最新时间发放token,在guava中没有采用使用其他线程异步发放token的方式，把token的更新放在了我们每次调用限流方法中，这个设计可以值得学习一下，很多时候不一定需要异步线程去执行也可以达到我们想要的目的，并且也没有异步线程的复杂。
- Step2: 扣除令牌，这里我们已经在canAcquire中校验过了，令牌一定能扣除成功。

使用sentinel的限流稍微比guava复杂很多，下面写了一个最简单的代码：

复制

```
String KEY = "test";
        // ============== 初始化规则 =========
        List<FlowRule> rules = new ArrayList<FlowRule>();
        FlowRule rule1 = new FlowRule();
        rule1.setResource(KEY);
        // set limit qps to 20
        rule1.setCount(20);
        rule1.setGrade(RuleConstant.FLOW_GRADE_QPS);
        rule1.setLimitApp("default");
        rules.add(rule1);
        rule1.setControlBehavior(CONTROL_BEHAVIOR_DEFAULT);
        FlowRuleManager.loadRules(rules);
        // ================ 限流判定 ===========
        Entry entry = null;

        try {
            entry = SphU.entry(KEY);
            // do something

        } catch (BlockException e1) {
            // 限流会抛出BlockException 异常
        }finally {
            if (entry != null) {
                entry.exit();
            }
        }
```


- Step1：在sentinel中比较强调Resource这个概念，我们所保护的或者说所作用于都是基于Resource来说，所以我们首先需要确定我们的Resource的key，这里我们简单的设置为test了。
- Step2：然后我们初始化我们这个Resource的一个限流规则，我们这里选择的是针对QPS限流并且策略选择的是默认，这里默认的话就是使用的滑动窗口版的计数器，然后加载到全局的规则管理器里面,整个规则的设置和guava的差别比较大。
- Step3: 在sentinel第二个比较重要的概念就是Entry,Entry表示一次资源操作，内部会保存当前invocation信息,在finally的时候需要对entry进行退出。我们执行限流判定的时候实际上也就是获取Entry，SphU.entry也就是我们执行我们上面限流规则的关键，这里和guava不一样如果被限流了，就会抛出BlockException，我们在进行限流的处理。

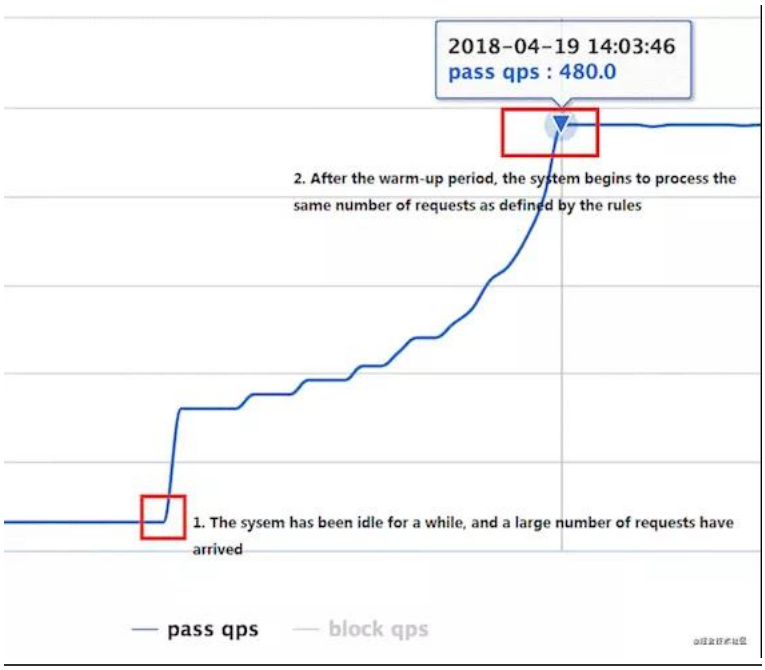
虽然sentinel的使用整体比guava复杂很多，但是算法的可选比guava的限流也多一点。

基于并发数(线程数)

我们之前介绍的都是基于QPS的，在sentinel中提供了基于并发数的策略，效果类似于信号量隔离，当我们需要让业务线程池不被慢调用耗尽，我们就可以使用这种模式。

通常来说我们同一个服务提供的http接口都是使用的一个线程池，比如我们使用的tomcat-web服务器那么我们就会有个tomcat的业务线程池，如果在http中有两个方法A和B,B的速度相对来说比较快，A的速度相对来说比较慢，如果大量的调用A这个方法，由于A的速度太慢，线程得不到释放，有可能导致线程池被耗尽，另一个方法B就得不到线程。这个场景我们之前有遇到过直接导致整个服务所接收的请求全部被拒绝。有的同学说限制A的QPS不是就可以了吗，要注意的是QPS是每秒的，如果我们这个A接口的耗时大于1s，那么下一波A来了之后QPS是要重新计算的。

- Warm Up：设置为Behavior为CONTROL_BEHAVIOR_WARM_UP,类似之前guava中介绍的warmup。预热启动方式。当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。这个模式下QPS的曲线图如下：



- 匀速排队：设置Behavior为CONTROL_BEHAVIOR_RATE_LIMITER,这个模式其实就是漏斗算法，优缺点之前也讲解过了
- Warm Up + 匀速排队：设置Behavior为CONTROL_BEHAVIOR_WARM_UP_RATE_LIMITER，之前warm up到高水位之后使用的是滑动窗口的算法限流，这个模式下继续使用匀速排队的算法。

基于调用关系

sentinel提供了更为复杂的一种限流，可以基于调用关系去做更为灵活的限流：

- 根据调用方限流：调用方的限流使用比较复杂，需要调用ContextUtil.enter(resourceName, origin),origin就是我们的调用方标识，然后在我们的rule设置参数的时候，对limitApp进行设置就可以进行对调用方的限流：
- 设置为default,默认对所有调用方都限流。
- 设置为{some_origin_name},代表对特定的调用者才限流。
- 设置为other，会对配置的一个referResource参数代表的调用者除外的进行限流。

关联流量控制：在sentinel中也支持，两个有关联的资源可以互相影响流量控制，比如有两个接口都使用的是同一个资源，一个接口比较重要，另外一个接口不是那么重要，我们可以设置一个规则当重要的接口大量访问的时候，就可以对另外一个不重要接口进行限流，防止这个接口突然出现流量影响重要的接口。

sentinel的一些问题

sentinel虽然提供了这么多算法，但是也有一些问题：

- 首先来说sentinel上手比较难，对比guava的两行代码来说，使用sentinel需要了解一些名词，然后针对这些名词再来使用，虽然sentinel提供了一些注解来帮助我们简化使用，但是整体来说还是比guava要复杂。
- sentinel有一定的运维成本，sentinel的使用往往需要搭建sentinel的server后台，对比guava的开箱即用来说，有一定的运维成本。
- sentinel的限流统计有一定的并发问题，在sentinel的源码中是没有加锁的地方的，极端情况下如果qps限制的是10，如果有100个同时过限流的逻辑，这个时候都会通过，而guava不会发生这样的情况。

们整个集群的QPS可能有没有500，甚至在400的时候就被限流了，这个是我们真实场景中所遇到过的。既然单机限流有问题，那么我们应该设计一个更加完善的集群限流的方案

Redis

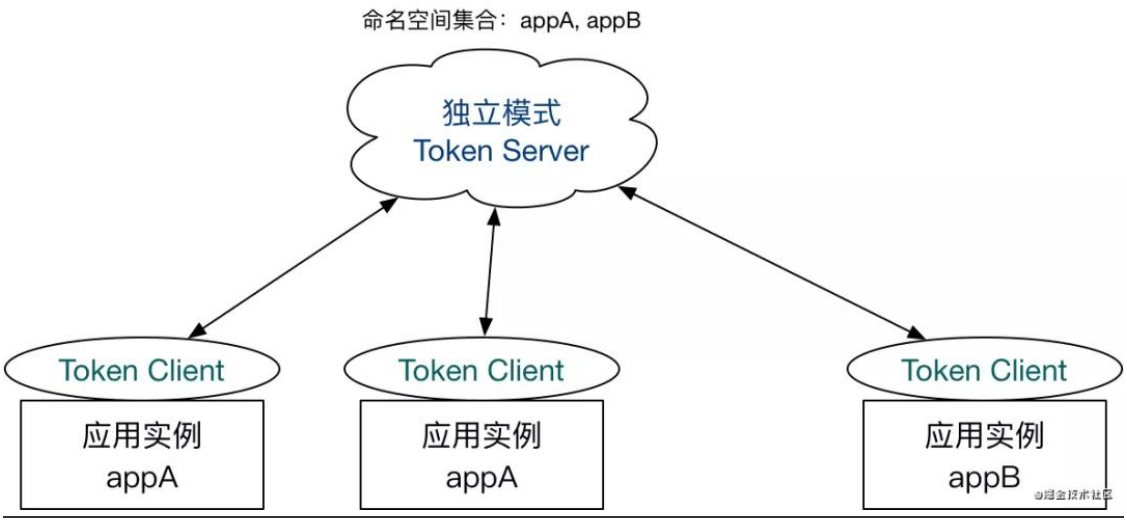
这个方案不依赖限流的框架，我们整个集群使用同一个redis即可，需要自己封装一下限流的逻辑，这里我们使用最简单的计数器去设计，我们将我们的系统时间以秒为单位作为key，设置到redis里面(可以设置一定的过期时间用于空间清理)，利用redis的int原子加法，每来一个请求都进行+1，然后再判断当前值是否超过我们限流的最大值。

redis的方案实现起来整体来说比较简单，但是强依赖我们的系统时间，如果不同机器之间的系统时间有偏差限流就有可能不准确。

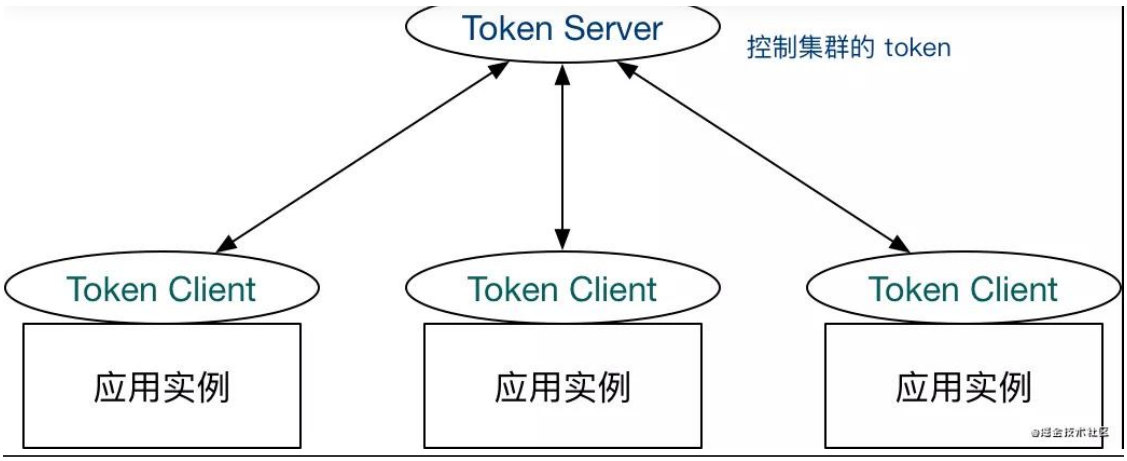
sentinel

在sentinel中提供了集群的解决方案，这个对比其他的一些限流框架是比较有特色的。在sentinel中提供了两种模式：

- 独立模式：限流服务作为单独的server进行部署，如下图所示，所有的应用都向单独部署的token-server进行获取token,这种模式适用于跨服务之间的全局限流，比如下面图中，A和B都会去token-server去拿，这个场景一般来说比较少，更多的还是服务内集群的限流比较多。



- 内嵌模式：在内嵌模式下，我们会把server部署到我们应用实例中，我们也可以通过接口转换我们的server-client身份，当然我们可以自己引入一些zk的一些逻辑设置让我们的leader去当server,机器挂了也可以自动切换。这种比较适合同一个服务集群之间的限流，灵活性比较好，但是要注意的是大量的token-server的访问也有可能影响我们自己的机器。



当然sentinel也有一些兜底的策略，如果token-server挂了我们可以退化成我们单机限流的模式，不会影响我们正常的服务。

实战

我们上面已经介绍了很多限流的工具，但是很多同学对怎么去限流仍然比较迷惑。我们如果对一个场景或者一个资源做限流的话有下面几个点需要确认一下：

- 什么地方去做限流
- 限多少流
- 怎么去选择工具

什么地方去做限流

这个问题比较复杂，很多公司以及很多团队的做法都不相同，在美团的时候搞了一波SOA,那个时候我记得所有的服务所有的接口都需要做限流，叫每个团队去给接口评估一个合理的QPS上限，这样做理论上来说是对的，我们每个接口都应该给与一个上限，防止把整体系统拖垮，但是这样做的成本是非常之高的，所以大部分公司还是选择性的去做限流。

首先我们需要确定一些核心的接口，比如电商系统中的下单，支付，如果流量过大那么电商系统中的路径就有问题，比如像对账这种边缘的接口(不影响核心路径)，我们可以不设置限流。

其次我们不一定只在接口层才做限流，很多时候我们直接在网关层把限流做了，防止流量进一步渗透到核心系统中。当然前端也能做限流，当前端捕获到限流的错误码之后，前端可以提示等待信息，这个其实也算是限流的一部分。其实当限流越在下游触发我们的资源的浪费就越大，因为在下游限流之前上游已经做了很多工作了，如果这时候触发限流那么之前的工作就会白费，如果涉及到一些回滚的工作还会加大我们的负担，所以对于限流来说应该是越上层触发越好。

限多少流

限多少流这个问题大部分的时候可能就是一个历史经验值，我们可以通过日常的qps监控图，然后再在这个接触上加一点冗余的QPS可能这个就是我们的限流了。但是有一个场景需要注意，那就是大促(这里指的是电商系统里面的场景，其他系统类比流量较高的场景)的时候，我们系统的流量就会突增，再也不是我们日常的QPS了，这种情况下，往往需要我们在大促之前给我们系统进行全链路压测，压测出一个合理的上限，然后限流就基于这个上限去设置。

怎么去选择工具

一般来说大一点的互联网公司都有自己的统一限流的工具这里直接采用就好。对于其他情况的话，如果没有集群限流或者熔断这些需求，我个人觉得选择RateLimter是一个比较不错的选择，应该其使用比较简单，基本没有学习成本，如果有其他的一些需求我个人觉得选择sentinel，至于hytrix的话我个人不推荐使用，因为这个已经不再维护了。

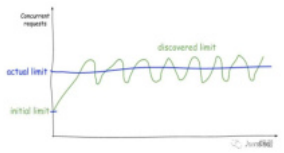
总结

相关推荐

五种限流算法，七种限流方式，挡住突发流量?

这篇文章介绍实现限流的几种方式，主要是窗口算法和桶算法，两者各有优势。

2022-03-18 14:33:22 限流 算法 微服务



常用的限流框架，你都会用吗?

作为应对高并发的手段之一，限流并不是一个新鲜的话题了。从Guava的Ratelimiter到Hystrix，以及Sentinel都可作为限流的工具。

2021-05-21 12:36:16 限流 代码 Java

一文搞懂Nginx限流，原来这么简单

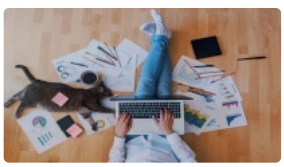
Nginx现在已经是最火的负载均衡之一，在流量陡增的互联网面前，接口限流也是很有必要的，尤其是针对高并发的场景。

2019-05-27 14:03:48 开发 技能 代码

我司“双11”限流方案，进来抄作业!

日常生活中，有哪些需要限流的地方?像我旁边有一个国家景区，平时可能根本没什么人前往，但是一到十一或者春节就人满为患，这时候景区管理人员就会实行一系列的政策来限制进入人流量，

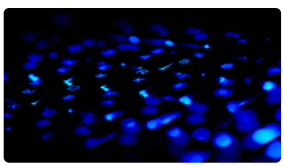
2021-10-21 06:39:41



亿级流量架构的服务限流思路与方法

限流算法很多，常见的有三类，分别是计数器算法、漏桶算法、令牌桶算法，下面逐一讲解。

2021-10-12 10:00:25



面试必备：4种经典限流算法讲解

最近，我们的业务系统引入了Guava的RateLimiter限流组件，它是基于令牌桶算法实现的,而令牌桶是非常经典的限流算法。本文将跟大家一起学习几种经典的限流算法。

2021-05-31 07:01:46 限流 算法 令牌

2019-12-13 08:52:48 高并发 系统 限流

断降级和服务限流。

2019-08-13 15:36:57

聊聊高并发系统之限流特技-2

上一篇《聊聊高并发系统限流特技-1》讲了限流算法、应用级限流、分布式限流;本篇将介绍接入层限流实现。

2016-11-28 08:58:43

详解Nginx限流方案的实现

一般对外暴露的系统,在促销或者黑客攻击时会涌来大量的请求,为了保护系统不被瞬间到来的高并发流量给打垮, 就需要限流。

2021-07-23 14:58:28

服务器设计方案之应用限流

在一个高并发系统中对流量的把控是非常重要的，当巨大的流量直接请求到我们的服务器上没多久就可能造成接口不可用，不处理的话甚至会造成整个应用不可用。

2019-01-23 16:44:37 服务器 应用 限流

分布式限流，你想知道的都在这里

在一个高并发系统中对流量的把控是非常重要的，当巨大的流量直接请求到我们的服务器上没多久就可能造成接口不可用，不处理的话甚至会造成整个应用不可用。

2019-04-24 08:31:43

不会高并发系统限流，肯定进不了大厂！

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。本文结合作者的一些经验介绍限流的相关概念、算法和常规的实现方式。

2020-02-20 08:00:37 缓存 降级 限流

保障系统高可用性之限流

我们经常可以看到各大云服务厂商号称自己的服务五个九，这里的五个九指的就是系统服务时间占整体时间的99.999%，按照一年来算的话，系统宕机时间必须小于5分16秒。

2020-05-21 14:10:36

分布式服务限流实战，已经为你排好坑了

由于API接口无法控制调用方的行为，因此当遇到瞬时请求量激增时，会导致接口占用过多服务器资源，使得其他请求响应速度降低或是超时，更有甚者可能导致服务器宕机。

2019-08-27 08:30:19

分布式服务限流实战，已经为你排好坑了

由于API接口无法控制调用方的行为，因此当遇到瞬时请求量激增时，会导致接口占用过多服务器资源，使得其他请求响应速度降低或是超时，更有甚者可能导致服务器宕机。

手机流量大战套路揭秘：“不限量”都是有条件的

随着手机流量取消漫游费大限临近，满大街“不限量套餐”的广告正在猛烈轰击着人们的眼球。目前的“不限量套餐”其实也是有“套路”的，引发了强烈的争议和吐槽。而工信部已经发话了，要求运营

2018-06-11 23:59:06 手机流量 运营商 不限量套餐

为什么我们都感觉最近流量被偷走了？

4G以来，人们不再抱怨网速慢，而是流量跑得快。实际上，这也只是历史的新一轮循环，从有手机上网以来，这样的循环就一直存在。与语音不同，流量的单位使用量不是固定的，单价也不是固

2015-11-06 16:11:43 流量 4G

《极限挑战》之后南京将再一次全城拥堵

2015-08-07 13:36:49