在分布式高可用设计中，限流应该是应用最广泛的技术手段之一，今天一起来讨论一下，为什么需要限流，以及常见的限流算法都有哪些。

常见限流算法

限流是服务降级的一种手段，顾名思义，通过限制系统的流量，从而实现保护系统的目的。

合理的限流配置，需要了解系统的吞吐量，所以，限流一般需要结合容量规划和压测来进行。当外部请求接近或者达到系统的最大阈值时，触发限流，采取其他的手段进行降级，保护系统不被压垮。常见的降级策略包括延迟处理、拒绝服务、随机拒绝等。

限流后的策略，其实和 Java 并发编程中的线程池非常类似，我们都知道，线程池在任务满的情况下，可以配置不同的拒绝策略，比如：

- AbortPolicy，会丢弃任务并抛出异常；
- DiscardPolicy，丢弃任务，不抛出异常；
- DiscardOldestPolicy 等，当然也可以自己实现拒绝策略。

Java 的线程池是开发中一个小的功能点，但是见微知著，也可以引申到系统的设计和架构上，将知识进行合理地迁移复用。

限流方案中有一点非常关键，那就是如何判断当前的流量已经达到我们设置的最大值，具体有不同的实现策略，下面进行简单分析。

计数器法

一般来说，我们进行限流时使用的是单位时间内的请求数，也就是平常说的 QPS，统计 QPS 最直接的想法就是实现一个计数器。

计数器法是限流算法里最简单的一种算法，我们假设一个接口限制 100 秒内的访问次数不能超过 10000 次，维护一个计数器，每次有新的请求过来，计数器加 1。这时候判断，如果计数器的值小于限流值，并且与上一次请求的时间间隔还在 100 秒内，允许请求通过，否则拒绝请求；如果超出了时间间隔，要将计数器清零。

下面的代码里使用 AtomicInteger 作为计数器，可以作为参考：

```
public class CounterLimiter {
    //初始时间
    private static long startTime = System.currentTimeMillis();
    //初始计数值
    private static final AtomicInteger ZERO = new AtomicInteger(0);
    //时间窗口限制
    private static final int interval = 10000;
    //限制通过请求
    private static int limit = 100;
    //请求计数
    private AtomicInteger requestCount = ZERO;
    //获取限流
    public boolean tryAcquire() {
        long now = System.currentTimeMillis();
        //在时间窗口内
        if (now < startTime + interval) {
            //判断是否超过最大请求
            if (requestCount.get() < limit) {
                requestCount.incrementAndGet();
                return true;
            }
            return false;
        } else {
            //超时重置
            requestCount = ZERO;
            startTime = now;
            return true;
        }
    }
}
```

计数器策略进行限流，可以从单点扩展到集群，适合应用在分布式环境中。单点限流使用内存即可，如果扩展到集群限流，可以用一个单独的存储节点，比如 Redis 或者 Memcached 来进行存储，在固定的时间间隔内设置过期时间，就可以统计集群流量，进行整体限流。

计数器策略有一个很大的缺点，是对临界流量不友好，限流不够平滑。假设这样一个场景，我们限制用户一分钟下单不超过 10 万次，现在在两个时间窗口的交汇点，前后一秒钟内，分别发送 10 万次请求。也就是说，窗口切换的这两秒钟内，系统接收了 20 万下单请求，这个峰值可能会超过系统阈值，影响服务稳定性。

对计数器算法的优化，就是避免出现两倍窗口限制的请求，可以使用滑动窗口算法实现，感兴趣的同学可以去了解一下。

漏桶和令牌桶算法

漏桶算法和令牌桶算法，在实际应用中更加广泛，也经常被拿来对比，所以我们放在一起进行分析。

漏桶算法可以用漏桶来对比，假设现在有一个固定容量的桶，底部钻一个小孔可以漏水，我们通过控制漏水的速度，来控制请求的处理，实现限流功能。

漏桶算法的拒绝策略很简单，如果外部请求超出当前阈值，则会在水桶里积蓄，一直到溢出，系统并不关心溢出的流量。漏桶算法是从出口处限制请求速率，并不存在上面计数器法的临界问题，请求曲线始终是平滑的。

漏桶算法的一个核心问题是，对请求的过滤太精准了，我们常说，水至清则无鱼，其实在限流里也是一样的， 我们限制每秒下单 10 万次，那 10 万零 1 次请求呢？是不是必须拒绝掉呢？

大部分业务场景下这个答案是否定的，虽然限流了，但还是希望系统允许一定的突发流量，这时候就需要令牌桶算法。

再来看一下令牌桶算法，在令牌桶算法中，假设我们有一个大小恒定的桶，这个桶的容量和设定的阈值有关，桶里放着很多令牌，通过一个固定的速率，往里边放入令牌，如果桶满了，就把令牌丢掉，最后桶中可以保存的最大令牌数永远不会超过桶的大小。

当有请求进入时，就尝试从桶里取走一个令牌，如果桶里是空的，那么这个请求就会被拒绝。

不知道你有没有应用过 Google 的 Guava 开源工具包，在 Guava 中，就有限流策略的工具类 RateLimiter，RateLimiter 基于令牌桶算法实现流量限制，使用非常方便。

RateLimiter 会按照一定的频率往桶里扔令牌，线程拿到令牌才能执行，RateLimter 的 API 可以直接应用，主要方法是 acquire 和 tryAcquire，acquire 会阻塞，tryAcquire 方法则是非阻塞的。下面是一个简单的示例：

```
public class LimiterTest {
    public static void main(String[] args) throws InterruptedException {
        //允许10个，permitsPerSecond
        RateLimiter limiter = RateLimiter.create(100);
        for(int i=1;i<200;i++){
            if (limiter.tryAcquire(1)){
                System.out.println("第"+i+"次请求成功");
            }else{
                System.out.println("第"+i+"次请求拒绝");
            }
        }
    }
}
```

不同限流算法的比较

计数器算法实现比较简单，特别适合集群情况下使用，但是要考虑临界情况，可以应用滑动窗口策略进行优化，当然也是要看具体的限流场景。

漏桶算法和令牌桶算法，漏桶算法提供了比较严格的限流，令牌桶算法在限流之外，允许一定程度的突发流量。在实际开发中，我们并不需要这么精准地对流量进行控制，所以令牌桶算法的应用更多一些。

如果我们设置的流量峰值是 permitsPerSecond=N，也就是每秒钟的请求量，计数器算法会出现 2N 的流量，漏桶算法会始终限制N的流量，而令牌桶算法允许大于 N，但不会达到 2N 这么高的峰值。

关于这几种限流算法的扩展讨论，我之前在博客中也分析过，可以点击： 96秒破百亿，双11如何抗住高并发流量，作为补充阅读。

总结

这一课时总结了系统限流的常用策略，包括计数器法、漏桶算法、令牌桶算法。

在你的工作中，在对系统进行高可用设计时，都做了哪些工作呢？比如如何进行容量评估，超出系统水位如何进行限流，欢迎留言分享你的经验。

精选评论

****7162：**

> 这些限流都是单机jvm 层的吧

讲师回复：

> 单机还是分布式要看具体实现，比如计数器方法可以使用外部的Redis存储实现分布式限流