限流

限流是面试中的常见的面试题（尤其是大厂面试、高P面试）

为什么要限流

> 简单来说：

> 限流在很多场景中用来限制并发和请求量，比如说秒杀抢购，保护自身系统和下游系统不被巨型流量冲垮等。

以微博为例，例如某某明星公布了恋情,访问从平时的50万增加到了500万，系统的规划能力，最多可以支撑**200万访问**,那

行限流规则,保证是一个可用的状态,不至于**服务器崩溃**，所有请求不可用。

参考链接

系统架构知识图谱（一张价值10w的系统架构知识图谱）

https://www.processon.com/view/link/60fb9421637689719d246739

秒杀系统的架构

https://www.processon.com/view/link/61148c2b1e08536191d8f92f

限流的思想

> 在保证可用的情况下尽可能多增加进入的人数,其余的人在排队等待,或者返回友好提示,保证里面的进行系统的用户可以正常使用，防止系统雪崩。

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,但是一到五一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,
为什么要限流呢?

假如景区能容纳一万人,现在进去了三万人,势必摩肩接踵,整不好还会有事故发生,这样的结果就是所有人的体验都不好，如果发生了事故景区可能还要关闭,导致对外不可用，这样的后果就是所有人都觉得体验糟糕透了。

限流的算法

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。

限流的手段通常有计数器、漏桶、令牌桶。注意限流和限速（所有请求都会处理）的差别，视
业务场景而定。

（1）计数器:

在一段时间间隔内（时间窗/时间区间），处理请求的最大数量固定，超过部分不做处理。

（2）漏桶:

漏桶大小固定，处理速度固定，但请求进入速度不固定 青
况请求过多时，会丢弃过多的请求）。

（3）令牌桶：

令牌桶的大小固定，令牌的产生速度固定，但是消耗令牌（即请求）速度不固定（可以应对一些某些时间请求过多的情况）；每个请求都会从令牌桶中取出令牌，如果没有令牌则丢弃该次请求。

计数器算法

计数器限流定义：

在一段时间间隔内（时间窗/时间区间），处理请求的最大数量固定，超过部分不做处理。

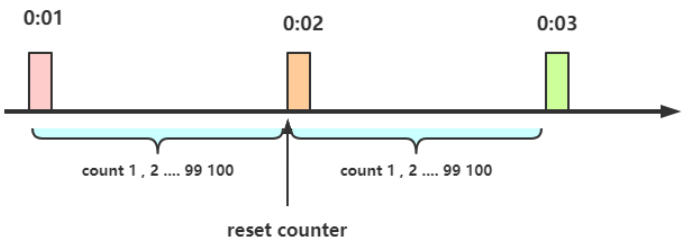
简单粗暴,比如指定线程池大小，指定数据库连接池大小、nginx连接数等,这都属于计数器算法。

计数器算法是限流算法里最简单也是最容易实现的一种算法。

举个例子,比如我们规定对于A接口，我们1分钟的访问次数不能超过100个。

那么我们可以这么做：

- 在一开 始的时候，我们可以设置一个计数器counter，每当一个请求过来的时候，counter就加1，如果counter的值大于100并且该请求与第一个请求的间隔时间还在1分钟之内，那么说明请求数过多,拒绝访问；
- 如果该请求与第一个请求的间隔时间大于1分钟，且counter的值还在限流范围内，那么就重置 counter,就是这么简单粗暴。



计算器限流的实现

```
package com.crazymaker.springcloud.ratelimit;

import lombok.extern.slf4j.Slf4j;
import org.junit.Test;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import
java.util.concurrent.atomic.AtomicIntege
import
```

```
java.util.concurrent.atomic.AtomicLong;

// 计速器 限速
@Slf4j
public class CounterLimiter
{

    // 起始时间
    private static long startTime =
System.currentTimeMillis();
    // 时间区间的时间间隔 ms
    private static long interval = 1000;
    // 每秒限制数量
    private static long maxCount = 2;
    //累加器
    private static AtomicLong accumulator =
new AtomicLong();

    // 计数判断, 是否超出限制
    private static long tryAcquire(long
taskId, int turn)
    {
        long nowTime =
System.currentTimeMillis();
        //在时间区间之内
        if (nowTime < startTime + interval)
        {
            long count =
accumulator.incrementAndGet();

            if (count <= maxCount)
            {
                return count;
            } else
            {
                return -count;
            }
        } else
        {
            //在时间区间之外
            synchronized
(CounterLimiter.class)
            {
                log.info("新时间区到
了,taskId{}, turn {}..", taskId, turn);
                // 再一次判断, 防止重复初始化
                if (nowTime > startTime +
interval)
                {
                    accumulator.set(0);
                    startTime = nowTime;
                }
            }
            return 0;
        }
    }
```

» 这是从请求参数里边，提前参数做限流

限流

» 限流是面试中的常见的面试题（尤其是大厂面试、高P面试）

» 为什么要限流

» 参考链接

› 限流的思想

› 日常生活中,有哪些需要限流的地方?

限流的算法

计数器算法

» 计数器限流定义：

» 计算器限流的实现

» 计数器限流的严重问题

漏桶算法

» 漏桶算法原理

» 漏桶算法实现

» 漏桶的问题

令牌桶限流

» 令牌桶算法

» 令牌桶算法实现

» 令牌桶的好处

» Guava RateLimiter

Nginx漏桶限流

» Nginx限流的简单演示

```
    //线程池，用于多线程模拟测试
    private ExecutorService pool =
Executors.newFixedThreadPool(10);

    @Test
    public void testLimit()
    {

        // 被限制的次数
        AtomicInteger limited = new
AtomicInteger(0);
        // 线程数
        final int threads = 2;
        // 每条线程的执行轮数
        final int turns = 20;
        // 同步器
        CountDownLatch countDownLatch = new
CountDownLatch(threads);
        long start =
System.currentTimeMillis();
        for (int i = 0; i < threads; i++)
        {
            pool.submit(() ->
            {
                try
                {

                    for (int j = 0; j <
turns; j++)
                    {

                        long taskId =
Thread.currentThread().getId();
                        long index =
tryAcquire(taskId, j);
                        if (index <= 0)
                        {
                            // 被限制的次数累积

limited.getAndIncrement();
                        }
                        Thread.sleep(200);
                    }


                } catch (Exception e)
                {
                    e.printStackTrace();
                }
                //等待所有线程结束
                countDownLatch.countDown();

            });
        }
        try
```

» 这是从请求参数里边，提前参数做限流

限流

» 限流是面试中的常见的面试题（尤其是大厂面试、高P面试）

» 为什么要限流

» 参考链接

> 限流的思想

> 日常生活中,有哪些需要限流的地方?

限流的算法

计数器算法

» 计数器限流定义：

» 计算器限流的实现

» 计数器限流的严重问题

漏桶算法

» 漏桶算法原理

» 漏桶算法实现

» 漏桶的问题

令牌桶限流

» 令牌桶算法

» 令牌桶算法实现

» 令牌桶的好处

» Guava RateLimiter

Nginx漏桶限流

» Nginx限流的简单演示

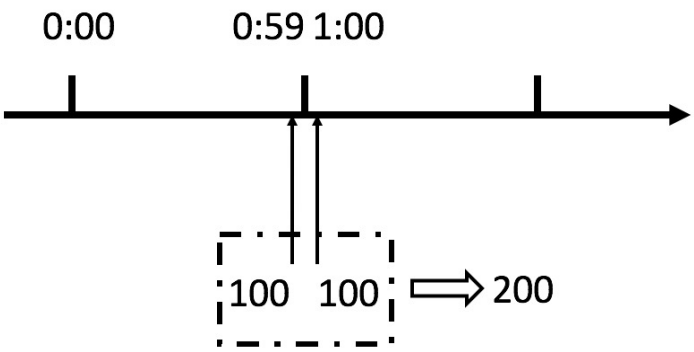
```
            {
                countDownLatch.await();
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            float time =
    (System.currentTimeMillis() - start) / 1000F;
            //输出统计结果

            log.info("限制的次数为: " +
    limited.get() +
                    ",通过的次数为: " + (threads *
    turns - limited.get()));
            log.info("限制的比例为: " + (float)
    limited.get() / (float) (threads * turns));
            log.info("运行的时长为: " + time);
        }


    }
```

计数器限流的严重问题

这个算法虽然简单，但是有一个十分致命的问题，那就是临界问题，我们看下图：



从上图中我们可以看到，假设有一个恶意用户，他在0:59时，瞬间发送了100个请求，并且1:00又瞬间发送了100个请求，那么其实这个用户在 1秒里面，瞬间发送了200个请求。

我们刚才规定的是1分钟最多100个请求（规划的吞吐量），也就是每秒钟最多1.7个请求，用户通过在时间窗口的重置节点处突发请求， 可以瞬间超过我们的速率限制。

用户有可能通过算法的这个漏洞，瞬间压垮我们的应用。

漏桶算法

漏桶算法限流的基本原理为：水（对应请求）从进水口进 桶里，漏桶以一定的速度出水（请求放行），当水流入速度 桶

» 这是从请求参数里边，提前参数做限流

内的总水量大于桶容量会直接溢出，请求被拒绝，如图所示。

大致的漏桶限流规则如下：

（1）进水口（对应客户端请求）以任意速率流入进入漏桶。

（2）漏桶的容量是固定的，出水（放行）速率也是固定的。

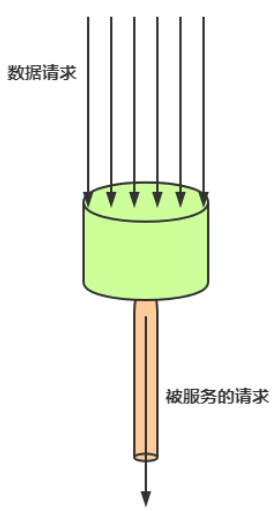
（3）漏桶容量是不变的，如果处理速度太慢，桶内水量会超出了桶的容量，则后面流入的水滴会溢出，表示请求拒绝。

漏桶算法原理

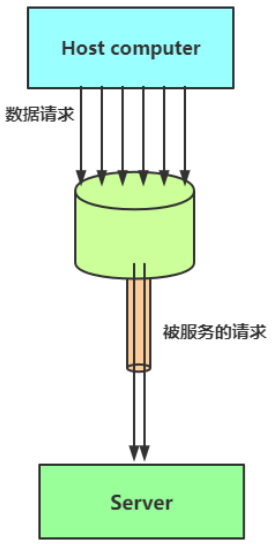
漏桶算法思路很简单：

> 水（请求）先进入到漏桶里，漏桶以一定的速度出水，当水流入速度过大会超过桶可接纳的容量时直接溢出。

可以看出漏桶算法能强行限制数据的传输速率。



漏桶算法



漏桶算法应用图示

漏桶算法其实很简单，可以粗略的认为就是注水漏水过程，往桶中以任意速率流入水，以一定速率流出水，当水超过**桶容量（capacity）**则丢弃，因为桶容量是不变的，保证了整体的速率。

以一定速率流出水，

» 这是从请求参数里边，提前参数做限流

» 这是从请求参数里边，提前参数做限流

削峰:有大量流量进入时,会发生溢出,从而限流保护服务可用

缓冲:不至于直接请求到服务器, 缓冲压力

消费速度固定 因为计算性能固定

漏桶算法实现

```
package com.crazymaker.springcloud.ratelimit;

import lombok.extern.slf4j.Slf4j;
import org.junit.Test;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import
java.util.concurrent.atomic.AtomicInteger;

// 漏桶 限流
@Slf4j
public class LeakBucketLimiter {

    // 计算的起始时间
    private static long lastOutTime =
System.currentTimeMillis();
    // 流出速率 每秒 2 次
    private static int leakRate = 2;

    // 桶的容量
    private static int capacity = 2;

    //剩余的水量
```

```
    private static AtomicInteger water = new
AtomicInteger(0);

    //返回值说明:
    // false 没有被限制到
    // true 被限流
    public static synchronized boolean
isLimit(long taskId, int turn) {
        // 如果是空桶，就当前时间作为漏出的时间
        if (water.get() == 0) {
            lastOutTime =
System.currentTimeMillis();
            water.addAndGet(1);
            return false;
        }
        // 执行漏水
        int waterLeaked = ((int)
((System.currentTimeMillis() - lastOutTime) /
1000)) * leakRate;
        // 计算剩余水量
        int waterLeft = water.get() -
waterLeaked;
        water.set(Math.max(0, waterLeft));
        // 重新更新leakTimeStamp
        lastOutTime =
System.currentTimeMillis();
        // 尝试加水,并且水还未满 ，放行
        if ((water.get()) < capacity) {
            water.addAndGet(1);
            return false;
        } else {
            // 水满，拒绝加水， 限流
            return true;
        }

    }


    //线程池，用于多线程模拟测试
    private ExecutorService pool =
Executors.newFixedThreadPool(10);

    @Test
    public void testLimit() {

        // 被限制的次数
        AtomicInteger limited = new
AtomicInteger(0);
        // 线程数
        final int threads = 2;
        // 每条线程的执行轮数
        final int turns = 20;
        // 线程同步器
        CountDownLatch countDownLatch =
CountDownLatch(threads);
        long start =
```

» 这是从请求参数里边，提前参数做限流

限流

» 限流是面试中的常见的面试题（尤其是大厂面试、高P面试）

» 为什么要限流

» 参考链接

› 限流的思想

› 日常生活中,有哪些需要限流的地方?

限流的算法

计数器算法

» 计数器限流定义:

» 计算器限流的实现

» 计数器限流的严重问题

漏桶算法

» 漏桶算法原理

» 漏桶算法实现

» 漏桶的问题

令牌桶限流

» 令牌桶算法

» 令牌桶算法实现

» 令牌桶的好处

» Guava RateLimiter

Nginx漏桶限流

» Nginx限流的简单演示

```
System.currentTimeMillis();
        for (int i = 0; i < threads; i++) {
            pool.submit(() ->
            {
                try {

                    for (int j = 0; j <
turns; j++) {

                        long taskId =
Thread.currentThread().getId();
                        boolean intercepted =
isLimit(taskId, j);
                        if (intercepted) {
                            // 被限制的次数累积

limited.getAndIncrement();
                        }
                        Thread.sleep(200);
                    }


                } catch (Exception e) {
                    e.printStackTrace();
                }
                //等待所有线程结束
                countDownLatch.countDown();

            });
        }
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        float time =
(System.currentTimeMillis() - start) / 1000F;
        //输出统计结果

        log.info("限制的次数为: " +
limited.get() +
                ",通过的次数为: " + (threads *
turns - limited.get()));
        log.info("限制的比例为: " + (float)
limited.get() / (float) (threads * turns));
        log.info("运行的时长为: " + time);
    }
}
```

漏桶的问题

漏桶的出水速度固定，也就是请求放行速度是固定的。

网上抄来抄去的说法：

» 这是从请求参数里边，提前参数做限流

> 漏桶不能有效应对突发流量，但是能起到平滑突发流量（整流）的作用。

实际上的问题：

> 漏桶出口的速度固定，不能灵活的应对后端能力提升。比如，通过动态扩容，后端流量从1000QPS提升到1WQPS，漏桶没有办法。

令牌桶限流

令牌桶算法以一个设定的速率产生令牌并放入令牌桶，每次用户请求都得申请令牌，如果令牌不足，则拒绝请求。
令牌桶算法中新请求到来时会从桶里拿走一个令牌，如果桶内没有令牌可拿，就拒绝服务。当然，令牌的数量也是有上限的。令牌的数量与时间和发放速率强相关，时间流逝的时间越长，会不断往桶里加入越多的令牌，如果令牌发放的速度比申请速度快，令牌桶会放满令牌，直到令牌占满整个令牌桶，如图所示。

令牌桶限流大致的规则如下：
（1）进水口按照某个速度，向桶中放入令牌。
（2）令牌的容量是固定的，但是放行的速度不是固定的，只要桶中还有剩余令牌，一旦请求过来就能申请成功，然后放行。
（3）如果令牌的发放速度，慢于请求到来速度，桶内就无牌可领，请求就会被拒绝。

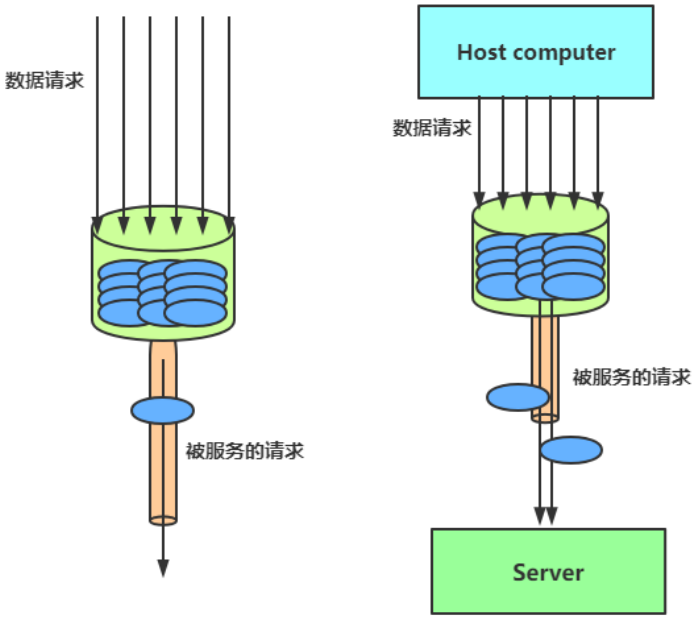
总之，令牌的发送速率可以设置，从而可以对突发的出口流量进行有效的应对。

令牌桶算法

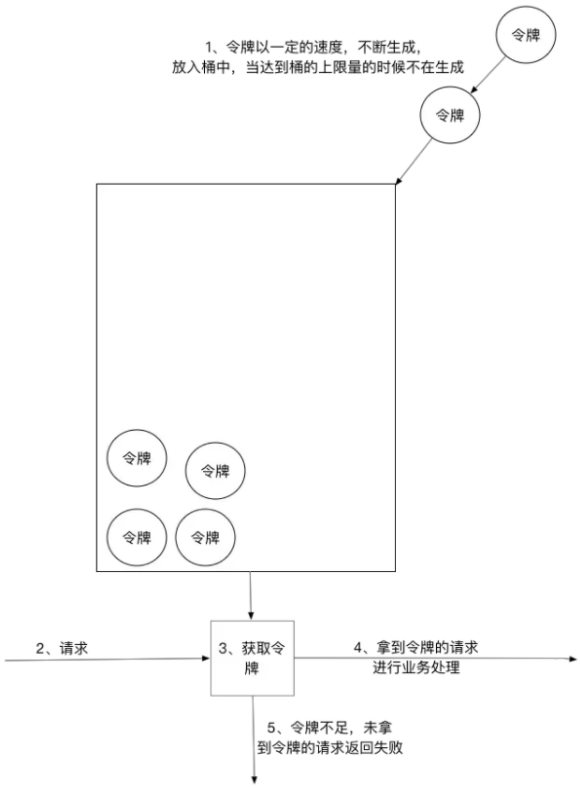
令牌桶与漏桶相似,不同的是令牌桶桶中放了一些令牌,服务请求到达后,要获取令牌之后才会得到服务,举个例子,我们平时去食堂吃饭,都是在食堂内窗口前排队的,这就好比是漏桶算法,大量的人员聚集在食堂内窗口外,以一定的速度享受服务,如果涌进来的人太多,食堂装不下了,可能就有一部分人站到食堂外了,这就没有享受到食堂的服务,称之为溢出,溢出可以继续请求,也就是继续排队,那么这样有什么问题呢?

如果这时候有特殊情况,比如有些赶时间的志愿者啦、或者高三要高考啦,这种情况就是突发情况,如果也用漏桶算法那也得慢慢排队,这也就没有解决我们的需求,对于很多应用场景来说，除了要求能够限制数据的平均传输速率外，还要求允许某种程度的突[illegible]
这时候漏桶算法可能就不合适了，令牌桶算法更为适合。

示，令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，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。



令牌桶算法　　令牌桶算法应用图示



令牌桶算法实现

```
package com.crazymaker.springcloud.ratelimit;

import lombok.extern.slf4j.Slf4j;
import org.junit.Test;

import java.util.concurrent.CountDownLa
import java.util.concurrent.ExecutorService;
```

» 这是从请求参数里边，提前参数做限流

```
import java.util.concurrent.Executors;
import
java.util.concurrent.atomic.AtomicInteger;

// 令牌桶 限速
@Slf4j
public class TokenBucketLimiter {
    // 上一次令牌发放时间
    public long lastTime =
System.currentTimeMillis();
    // 桶的容量
    public int capacity = 2;
    // 令牌生成速度 /s
    public int rate = 2;
    // 当前令牌数量
    public AtomicInteger tokens = new
AtomicInteger(0);
    ;

    //返回值说明:
    // false 没有被限制到
    // true 被限流
    public synchronized boolean
isLimited(long taskId, int applyCount) {
        long now =
System.currentTimeMillis();
        //时间间隔,单位为 ms
        long gap = now - lastTime;

        //计算时间段内的令牌数
        int reverse_permits = (int) (gap *
rate / 1000);
        int all_permits = tokens.get() +
reverse_permits;
        // 当前令牌数
        tokens.set(Math.min(capacity,
all_permits));
        log.info("tokens {} capacity {} gap
{} ", tokens, capacity, gap);

        if (tokens.get() < applyCount) {
            // 若拿不到令牌,则拒绝
            // log.info("被限流了.." + taskId
+ ", applyCount: " + applyCount);
            return true;
        } else {
            // 还有令牌，领取令牌
            tokens.getAndAdd( - applyCount);
            lastTime = now;

            // log.info("剩余令牌.." +
tokens);
            return false;
        }

    }
```

» 这是从请求参数里边，提前参数做限流

限流

限流的算法

计数器算法

漏桶算法

令牌桶限流

Nginx漏桶限流

```
    //线程池，用于多线程模拟测试
    private ExecutorService pool =
Executors.newFixedThreadPool(10);

    @Test
    public void testLimit() {

        // 被限制的次数
        AtomicInteger limited = new
AtomicInteger(0);
        // 线程数
        final int threads = 2;
        // 每条线程的执行轮数
        final int turns = 20;


        // 同步器
        CountDownLatch countDownLatch = new
CountDownLatch(threads);
        long start =
System.currentTimeMillis();
        for (int i = 0; i < threads; i++) {
            pool.submit(() ->
            {
                try {

                    for (int j = 0; j <
turns; j++) {

                        long taskId =
Thread.currentThread().getId();
                        boolean intercepted =
isLimited(taskId, 1);
                        if (intercepted) {
                            // 被限制的次数累积

limited.getAndIncrement();
                        }

                        Thread.sleep(200);
                    }


                } catch (Exception e) {
                    e.printStackTrace();
                }
                //等待所有线程结束
                countDownLatch.countDown();

            });
        }
        try {
            countDownLatch.await();
        } catch (InterruptedException e)
            e.printStackTrace();
```

» 这是从请求参数里边，提前参数做限流

限流

限流的算法

计数器算法

漏桶算法

令牌桶限流

Nginx漏桶限流

```
        }
        float time =
(System.currentTimeMillis() - start) / 1000F;
        //输出统计结果

        log.info("限制的次数为: " +
limited.get() +
                ",通过的次数为: " + (threads *
turns - limited.get()));
        log.info("限制的比例为: " + (float)
limited.get() / (float) (threads * turns));
        log.info("运行的时长为: " + time);
    }


}
```

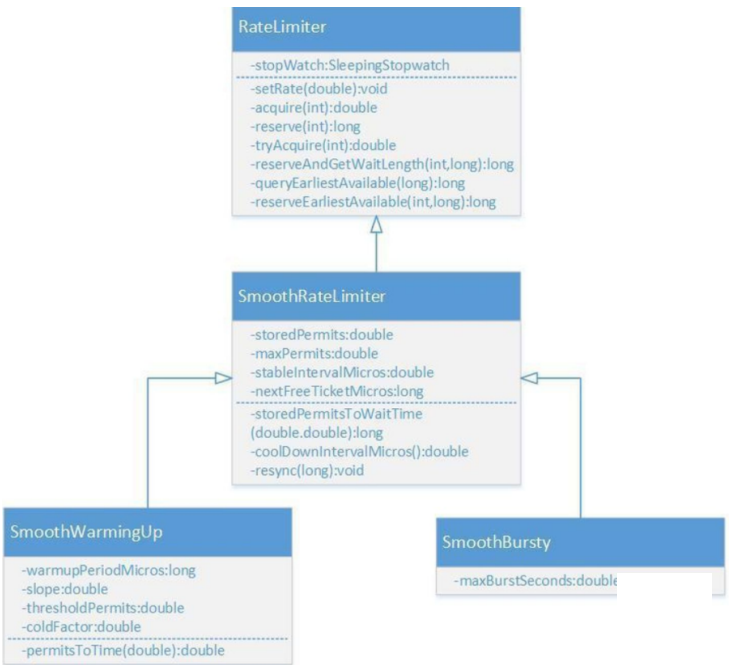
令牌桶的好处

令牌桶的好处之一就是可以方便地应对 突发出口流量（后端能力的提升）。

比如，可以改变令牌的发放速度，算法能按照新的发送速率调大令牌的发放数量，使得出口突发流量能被处理。

Guava RateLimiter

`Guava` 是Java领域优秀的开源项目，它包含了Google在Java项目中使用一些核心库，包含集合(Collections)，缓存(Caching)，并发编程库(Concurrency)，常用注解(Common annotations)，String操作，I/O操作方面的众多非常实用的函数。 Guava的 `RateLimiter` 提供了令牌桶算法实现：平滑突发限流(SmoothBursty)和平滑预热限流(SmoothWarmingUp)实现。



RateLimiter 的类图如上所示，

Nginx漏桶限流

Nginx限流的简单演示

每六秒才处理一次请求,如下

```
  limit_req_zone  $arg_sku_id
zone=skuzone:10m      rate=6r/m;
  limit_req_zone  $http_user_id
zone=userzone:10m      rate=6r/m;
  limit_req_zone  $binary_remote_addr
zone=perip:10m      rate=6r/m;
  limit_req_zone  $server_name
zone=perserver:1m   rate=6r/m;
```

这是从请求参数里边，提前参数做限流

这是从请求参数里边，提前参数，进行限流的次数统计key。

在http块里边定义限流的内存区域 zone。

```
  limit_req_zone  $arg_sku_id
zone=skuzone:10m      rate=6r/m;
  limit_req_zone  $http_user_id
zone=userzone:10m      rate=6r/m;
  limit_req_zone  $binary_remote_addr
zone=perip:10m      rate=6r/m;
  limit_req_zone  $server_name
zone=perserver:1m   rate=10r/s;
```

在location块中使用 限流zone，参考如下：

```
    #  ratelimit by sku id
    location  = /ratelimit/sku {
      limit_req  zone=skuzone;
      echo "正常的响应";
    }
```

测试

```
[root@cdh1 ~]#
/vagrant/LuaDemoProject/sh/linux/openresty-
restart.sh
shell dir is:
/vagrant/LuaDemoProject/sh/linux
Shutting down openrestry/nginx:  pid is
```

```
13485
Shutting down  succeeded!
OPENRESTRY_PATH:/usr/local/openresty
PROJECT_PATH:/vagrant/LuaDemoProject/src
nginx: [alert] lua_code_cache is off; this
will hurt performance in
/vagrant/LuaDemoProject/src/conf/nginx-
seckill.conf:90
openrestry/nginx starting succeeded!
pid is 14197
```

```
[root@cdh1 ~]# curl
http://cdh1/ratelimit/sku?sku_id=1
正常的响应
root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
正常的响应
[root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
限流后的降级内容
[root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
限流后的降级内容
[root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
限流后的降级内容
[root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
限流后的降级内容
[root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
限流后的降级内容
[root@cdh1 ~]#  curl
http://cdh1/ratelimit/sku?sku_id=1
正常的响应
```

从Header头部提前参数

1、nginx是支持读取非nginx标准的用户自定义header的，但是需要在http或者server下开启header的下划线支持:

underscores_in_headers on;

2、比如我们自定义header为X-Real-IP,通过第二个nginx获取该header时需要这样:

$http_x_real_ip; (一律采用小写，而且前面多了个http_)

```
 underscores_in_headers on;

  limit_req_zone  $http_user_id
zone=userzone:10m      rate=6r/m;
  server {
```

» 这是从请求参数里边，提前参数做限流

```
    listen       80 default;
    server_name  nginx.server *.nginx.server;
    default_type 'text/html';
    charset utf-8;


#  ratelimit by user id
    location  = /ratelimit/demo {
      limit_req  zone=userzone;
      echo "正常的响应";
    }



    location = /50x.html{
      echo "限流后的降级内容";
    }

    error_page 502 503 =200 /50x.html;

  }
```

测试

```
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
正常的响应
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]# curl -H "USER-ID:1"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]# curl -H "USER_ID:2"
http://cdh1/ratelimit/demo
正常的响应
[root@cdh1 ~]# curl -H "USER_ID:2"
http://cdh1/ratelimit/demo
限流后的降级内容
[root@cdh1 ~]#
[root@cdh1 ~]# curl -H "USER_ID:2"
http://cdh1/ratelimit/demo
限流后的降级内容
```

```
[root@cdh1 ~]# curl -H "USER-ID:3"
http://cdh1/ratelimit/demo
正常的响应
[root@cdh1 ~]# curl -H "USER-ID:3"
http://cdh1/ratelimit/demo
限流后的降级内容
```


Nginx漏桶限流的三个细分类型，即burst、nodelay参数详解

每六秒才处理一次请求,如下

```
limit_req_zone  $arg_user_id
zone=limti_req_zone:10m      rate=10r/m;
```

不带缓冲队列的漏桶限流

limit_req zone=limti_req_zone;

- 严格依照在limti_req_zone中配置的rate来处理请求
- 超过rate处理能力范围的，直接drop
- 表现为对收到的请求无延时

> 假设1秒内提交10个请求，可以看到一共10个请求，9个请求都失败了,直接返回503，

接着再查看 /var/log/nginx/access.log，印证了只有一个请求成功了，其它就是都直接返回了503，即服务器拒绝了请求。

```
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:11:08:01 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
```

带缓冲队列的漏桶限流

limit_req zone=limti_req_zone burst=5;

- 依照在limti_req_zone中配置的rate来处理请求
- 同时设置了一个大小为5的缓冲队列，在缓冲队列中的请求会等待慢慢处理
- **超过了burst缓冲队列长度和rate处理能力的请求被直接丢弃**
- 表现为对收到的请求有延时

> 假设1秒内提交10个请求，则可以发现在1s内，**在服务器接收到10个并发请求后，先处理1个请求，同时将5个请求放入burst缓冲队列中，等待处理。而超过（burst+1）数量的请求就被直接抛弃了，即直接抛弃了4个请求。**burst缓存的5个请求每隔6s处理一次。

接着查看 /var/log/nginx/access.log日志

```
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:35 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:35 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:35 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:35 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:35 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:41 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:47 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:53 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:39:59 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:40:05 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
```

带瞬时处理能力的漏桶限流

limit_req zone=req_zone burst=5 nodelay;

如果设置nodelay，会在瞬时提供处理(burst + rate)个请求的能力，请求数量超过（**burst + rate**）的时候就会直接返回503，峰值范围内的请求，**不存在请求需要等待的情况。**

> 假设1秒内提交10个请求，则可以发现在1s内，服务器端处理了6个请求（峰值速度：burst+10s内一个请求）。对于剩下的4个请求，直接返回503，在下一秒如果继续向服务端发送10个请求，服务端会直接拒绝这10个请求并返回503。

接着查看 /var/log/nginx/access.log日志

```
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 200 625 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
ApacheBench/2.3218.94.83.136 - - [28/Nov/2017:17:58:54 +0800] "GET / HTTP/1.0" 503 537 "-" "ApacheBench/2.3" "-"
```

可以发现在1s内，服务器端处理了6个请求（峰值速度：burst+原来的处理速度）。对于剩下的4个请求，直接返回503。

> 但是，总数额度和速度*时间保持一致， 就是额度用完了，需要等到一个有额度的时间段，才开始接收新的请求。如果一次处理了5个请求，相当于占了30s的额度，6*5=30。因为设定了6s处理1个请求，所以直到30
> s 之后，才可以再处理一个请求，即如果此时向服务端发送10个请求，会返回9个503，一个200

分布式限流组件

why

> 但是Nginx的限流指令只能在同一块内存区域有效，而在生产场景中秒杀的外部网关往往是多节点部署，所以这就需要用到分布式限流组件。

高性能的分布式限流组件可以使用Redis+Lua来开发，京东的抢购就是使用Redis+Lua完成的限流。并且无论是Nginx外部网关还是Zuul内部网关，都可以使用Redis+Lua限流组件。

理论上，接入层的限流有多个维度：

（1）用户维度限流：在某一时间段内只允许用户提交一次请求，比如可以采取客户端IP或者用户ID作为限流的key。

（2）商品维度的限流：对于同一个抢购商品，在某个时间段内只允许一定数量的请求进入，可以采取秒杀商品ID作为限流的key。

什么时候用nginx限流：

用户维度的限流，可以在ngix 上进行，因为使用nginx限流内存来存储用户id，比用redis 的key，来存储用户id，效率高。

什么时候用redis+lua分布式限流：

商品维度的限流，可以在redis上进行，不需要大量的计算访问次数的key，另外，可以控制所有的接入层节点的访问秒杀请求的总量。

redis+lua分布式限流组件

```
--- 此脚本的环境： redis 内部，不是运行在 nginx 内部

---方法：申请令牌
--- -1 failed
--- 1 success
--- @param key key 限流关键字
--- @param apply  申请的令牌数量
local function acquire(key, apply)
    local times = redis.call('TIME');
    -- times[1] 秒数   -- times[2] 微秒数
    local curr_mill_second = times[1] * 1000000 + times[2];
    curr_mill_second = curr_mill_second / 1000;

    local cacheInfo = redis.pcall("HMGET", key, "last_mill_second", "curr_permits", "max_permits", "rate")
    --- 局部变量：上次申请的时间
    local last_mill_second = cacheInfo[1];
    --- 局部变量：之前的令牌数
    local curr_permits = tonumber(cacheInfo[2]);
    --- 局部变量：桶的容量
    local max_permits = tonumber(cacheInfo[3]);
    --- 局部变量：令牌的发放速率
    local rate = cacheInfo[4];
    --- 局部变量：本次的令牌数
    local local_curr_permits = 0;

    if (type(last_mill_second) ~= 'boole
```

» 这是从请求参数里边，提前参数做限流

```
and last_mill_second ~= nil) then
        -- 计算时间段内的令牌数
        local reverse_permits =
math.floor(((curr_mill_second -
last_mill_second) / 1000) * rate);
        -- 令牌总数
        local expect_curr_permits =
reverse_permits + curr_permits;
        -- 可以申请的令牌总数
        local_curr_permits =
math.min(expect_curr_permits, max_permits);
    else
        -- 第一次获取令牌
        redis.pcall("HSET", key,
"last_mill_second", curr_mill_second)
        local_curr_permits = max_permits;
    end

    local result = -1;
    -- 有足够的令牌可以申请
    if (local_curr_permits - apply >= 0) then
        -- 保存剩余的令牌
        redis.pcall("HSET", key,
"curr_permits", local_curr_permits - apply);
        -- 为下次的令牌获取，保存时间
        redis.pcall("HSET", key,
"last_mill_second", curr_mill_second)
        -- 返回令牌获取成功
        result = 1;
    else
        -- 返回令牌获取失败
        result = -1;
    end
    return result
end
--eg
-- /usr/local/redis/bin/redis-cli  -a 123456
--eval
/vagrant/LuaDemoProject/src/luaScript/redis/r
ate_limiter.lua key , acquire 1  1

-- 获取 sha编码的命令
-- /usr/local/redis/bin/redis-cli  -a 123456
script load "$(cat
/vagrant/LuaDemoProject/src/luaScript/redis/r
ate_limiter.lua)"
-- /usr/local/redis/bin/redis-cli  -a 123456
script exists
"cf43613f172388c34a1130a760fc699a5ee6f2a9"

-- /usr/local/redis/bin/redis-cli -a 123456
evalsha
"cf43613f172388c34a1130a760fc699a5ee6f2a9" 1
"rate_limiter:seckill:1"  init 1  1
-- /usr/local/redis/bin/redis-cli -a 12
evalsha
```

» 这是从请求参数里边，提前参数做限流

```
"cf43613f172388c34a1130a760fc699a5ee6f2a9" 1
"rate_limiter:seckill:1"  acquire 1

--local rateLimiterSha =
"e4e49e4c7b23f0bf7a2bfee73e8a01629e33324b";

---方法：初始化限流 Key
--- 1 success
--- @param key key
--- @param max_permits  桶的容量
--- @param rate  令牌的发放速率
local function init(key, max_permits, rate)
    local rate_limit_info =
redis.pcall("HMGET", key, "last_mill_second",
"curr_permits", "max_permits", "rate")
    local org_max_permits =
tonumber(rate_limit_info[3])
    local org_rate = rate_limit_info[4]

    if (org_max_permits == nil) or (rate ~=
org_rate or max_permits ~= org_max_permits)
then
        redis.pcall("HMSET", key,
"max_permits", max_permits, "rate", rate,
"curr_permits", max_permits)
    end
    return 1;
end
--eg
-- /usr/local/redis/bin/redis-cli -a 123456 -
-eval
/vagrant/LuaDemoProject/src/luaScript/redis/r
ate_limiter.lua key , init 1  1
-- /usr/local/redis/bin/redis-cli -a 123456 -
-eval
/vagrant/LuaDemoProject/src/luaScript/redis/r
ate_limiter.lua  "rate_limiter:seckill:1"  ,
init 1  1


---方法：删除限流 Key
local function delete(key)
    redis.pcall("DEL", key)
    return 1;
end
--eg
-- /usr/local/redis/bin/redis-cli  --eval
/vagrant/LuaDemoProject/src/luaScript/redis/r
ate_limiter.lua key , delete


local key = KEYS[1]
local method = ARGV[1]
if method == 'acquire' then
    return acquire(key, ARGV[2], ARGV[3]
elseif method == 'init' then
```

» 这是从请求参数里边，提前参数做限流

```
    return init(key, ARGV[2], ARGV[3])
elseif method == 'delete' then
    return delete(key)
else
    --ignore
end
```

> 在redis中，为了避免重复发送脚本数据浪费网络资源，可以使用script load命令进行脚本数据缓存，并且返回一个哈希码作为脚本的调用句柄，
>
> 每次调用脚本只需要发送哈希码来调用即可。

分布式令牌限流实战

可以使用redis+lua，实战一票下边的简单案例：

令牌按照1个每秒的速率放入令牌桶，桶中最多存放2个令牌，那系统就只会允许持续的每秒处理2个请求，

或者每隔2 秒，等桶中2 个令牌攒满后，一次处理2个请求的突发情况，保证系统稳定性。

商品维度的限流

当秒杀商品维度的限流，当商品的流量，远远大于涉及的流量时，开始随机丢弃请求。

Nginx的令牌桶限流脚本getToken_access_limit.lua执行在请求的access阶段，但是，该脚本并没有实现限流的核心逻辑，仅仅调用缓存在Redis内部的rate_limiter.lua脚本进行限流。

getToken_access_limit.lua脚本和rate_limiter.lua脚本的关系，具体如图10-17所示。

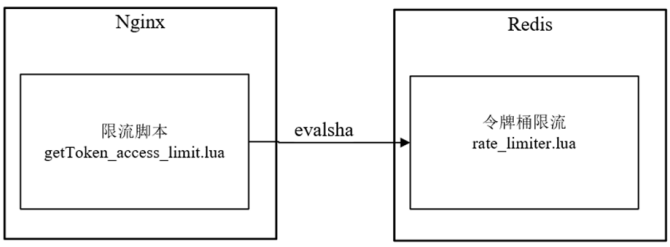


图10-17 getToken_access_limit.lua脚本和rate_limiter.lua脚本关系

什么时候在Redis中加载rate_limiter.lua脚本呢?

和秒杀脚本一样，该脚本是在Java程序启动商品秒杀时，　　生Redis的加载和缓存的。

» 这是从请求参数里边，提前参数做限流

还有一点非常重要，Java程序会将脚本加载完成之后的sha1编码，去通过自定义的key（具体为"lua:sha1:rate_limiter"）缓存在Redis中，以方便Nginx的getToken_access_limit.lua脚本去获取，并且在调用evalsha方法时使用。

注意：使用redis集群，因此每个节点都需要各自缓存一份脚本数据

```
/**
* 由于使用redis集群，因此每个节点都需要各自缓存一份脚本数据
* @param slotKey 用来定位对应的slot的slotKey
*/
public void storeScript(String slotKey){
if (StringUtils.isEmpty(unlockSha1) ||
!jedisCluster.scriptExists(unlockSha1,
slotKey)){
   //redis支持脚本缓存，返回哈希码，后续可以继续用来调用脚本
    unlockSha1 =
jedisCluster.scriptLoad(DISTRIBUTE_LOCK_SCRIPT_UNLOCK_VAL, slotKey);
   }
}
```

常见的限流组件

redission分布式限流采用令牌桶思想和固定时间窗口，trySetRate方法设置桶的大小，利用redis key过期机制达到时间窗口目的，控制固定时间窗口内允许通过的请求量。

spring cloud gateway集成redis限流,但属于网关层限流

疯狂创客圈
粉丝 - 1205 关注 - 2

+加关注

« 上一篇： 架构必看：12306抢票亿级流量架构演进（图解+秒懂+史上最全）
» 下一篇： seata AT模式实战（图解 秒懂 史上最全）

posted @ 2021-08-25 22:09 疯狂创客圈 阅读(6940) 评论(0) 编辑 收藏 举报