Guava令牌桶RateLimiter限流原理

小平果118 Lv3
2022年01月22日 20:07 · 阅读 644

关注

在分布式系统中，应对高并发访问时，**缓存、限流、降级**是保护系统正常运行的常用方法。当请求量突发暴涨时，如果不加以限制访问，则可能导致整个系统崩溃，服务不可用。

同时有一些业务场景，比如短信验证码，或者其它第三方API调用，也需要提供必要的访问限制支持。还有一些资源消耗过大的请求，比如数据导出等，也有限制访问频率的需求。

常见的限流算法有

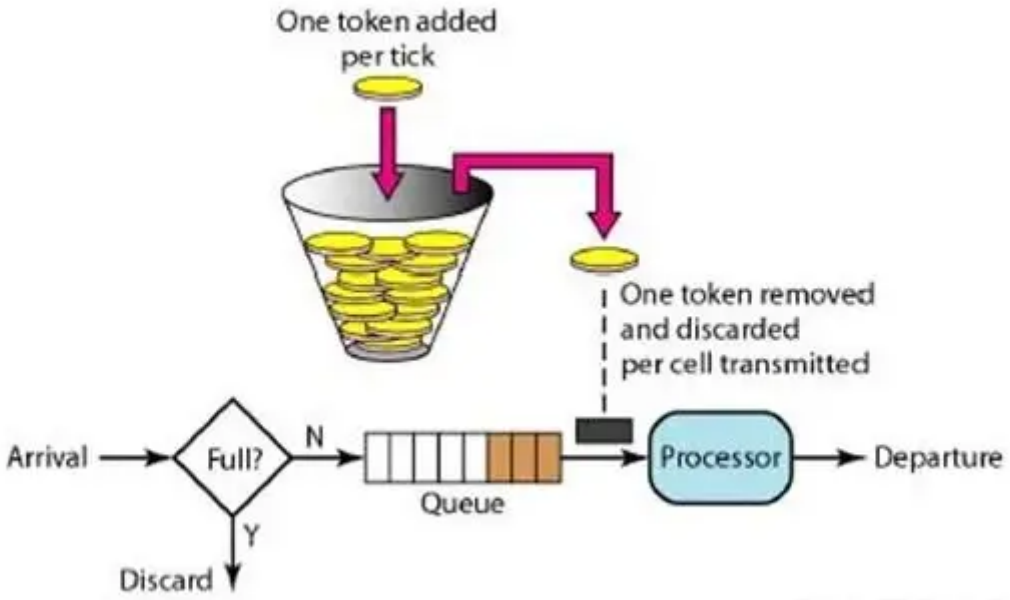
- 令牌桶算法，
- 漏桶算法
- 计数器算法。

本文主要对三个算法的基本原理及Google Guava包中令牌桶算法的实现RateLimiter进行介绍，下一篇文章介绍最近写的一个以RateLimiter为参考的分布式限流实现及计数器限流实现。

令牌桶算法

令牌桶算法的原理就是以一个恒定的速度往桶里放入令牌，每一个请求的处理都需要从桶里先获取一个令牌，当桶里没有令牌时，则请求不会被处理，要么排队等待，要么降级处理，要么直接拒绝服务。当桶里令牌满时，新添加的令牌会被丢弃或拒绝。

令牌桶算法的处理示意图如下（图片来自网络）

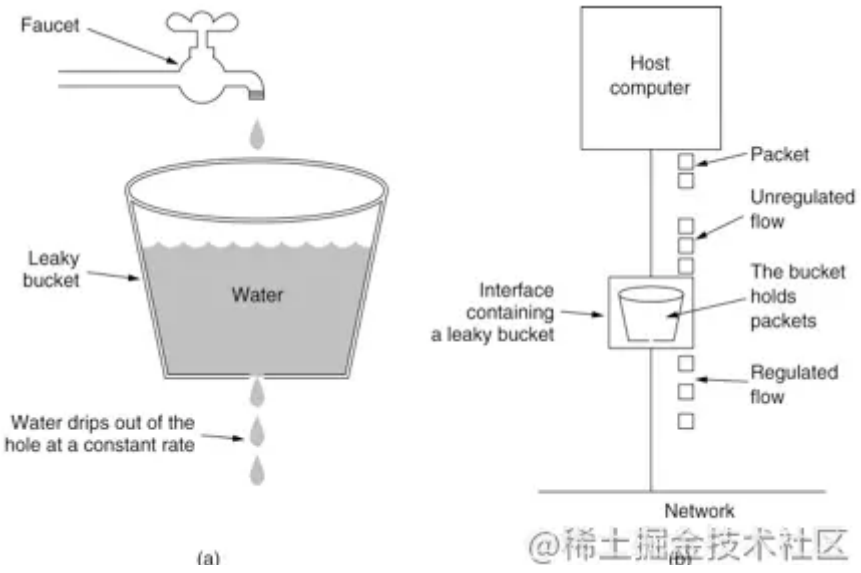


令牌桶算法主要是可以控制请求的平均处理速率，它允许预消费，即可以提前消费令牌，以应对突发请求，但是后面的请求需要为预消费买单（等待更长的时间），以满足请求处理的平均速率是一定的。

漏桶算法

漏桶算法的原理是水（请求）先进入漏桶中，漏桶以一定的速度出水（处理请求），当水流入速度大于流出速度导致水在桶内逐渐堆积直到桶满时，水会溢出（请求被拒绝）。

漏桶算法的处理示意图如下（图片来自网络）



漏桶算法主要是控制请求的处理速率，平滑网络上的突发流量，请求可以以任意速度进入漏桶中，但请求的处理则以恒定的速度进行。

计数器算法

计数器算法是限流算法中最简单的一种算法，限制在一个时间窗口内，至多处理多少个请求。比如每分钟最多处理10个请求，则从第一个请求进来的时间为起点，60s的时间窗口内只允许最多处理10个请求。下一个时间窗口又以前一时间窗口过后第一个请求进来的时间为起点。常见的比如一分钟内只能获取一次短信验证码的功能可以通过计数器算法来实现。

Guava RateLimiter解析

Guava是Google开源的一个工具包，其中的RateLimiter是实现了令牌桶算法的一个限流工具类。在pom.xml中添加guava依赖，即可使用RateLimiter

xml 复制代码

```xml
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>29.0-jre</version>
</dependency>
```

如下测试代码示例了RateLimiter的用法,

xml 复制代码

```
public static void main(String[] args) {
    RateLimiter rateLimiter = RateLimiter.create(1); //创建一个每秒产生一个令牌的令牌桶
    for(int i=1;i<=5;i++) {
        double waitTime = rateLimiter.acquire(i); //一次获取i个令牌
        System.out.println("acquire:" + i + " waitTime:" + waitTime);
```

```
}
```

运行后，输出如下，

xml 复制代码

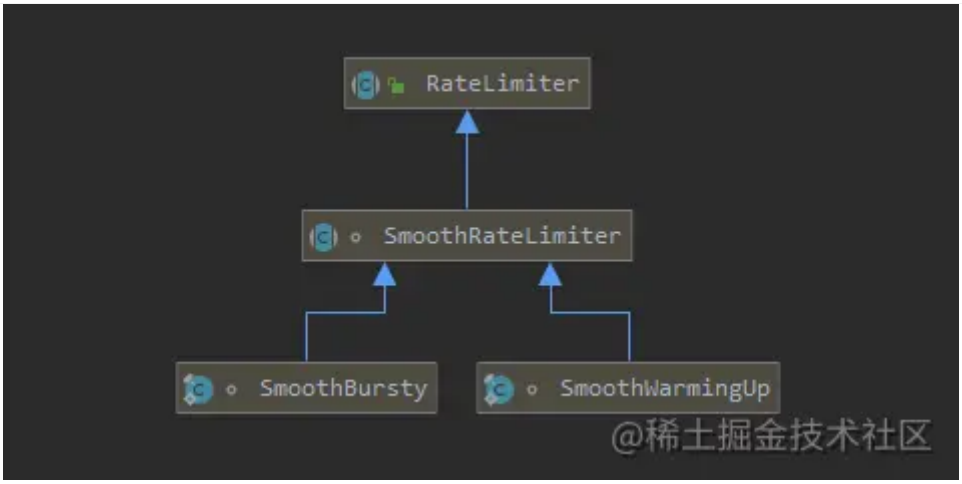
```
acquire:1 waitTime:0.0

acquire:2 waitTime:0.997729

acquire:3 waitTime:1.998076

acquire:4 waitTime:3.000303

acquire:5 waitTime:4.000223
```

第一次获取一个令牌时，等待0s立即可获取到（这里之所以不需要等待是因为令牌桶的预消费特性），第二次获取两个令牌，等待时间1s，这个1s就是前面获取一个令牌时因为预消费没有等待延到这次来等待的时间，这次获取两个又是预消费，所以下一次获取（取3个时）就要等待这次预消费需要的2s了，依此类推。可见预消费不需要等待的时间都由下一次来买单，以保障一定的平均处理速率（上例为1s一次）

RateLimiter有两种实现：

1. SmoothBursty： 令牌的生成速度恒定。使用 RateLimiter.create(double permitsPerSecond) 创建的是 SmoothBursty 实例。
2. SmoothWarmingUp：令牌的生成速度持续提升，直到达到一个稳定的值。WarmingUp，顾名思义就是有一个热身的过程。使用 RateLimiter.create(double permitsPerSecond, long warmupPeriod, TimeUnit unit) 时创建就是 SmoothWarmingUp 实例，其中 warmupPeriod 就是热身达到稳定速度的时间。

类结构如下



关键属性及方法解析（以 SmoothBursty 为例）

源码解析：

使用方法

java 复制代码

```
RateLimiter rateLimiter = RateLimiter.create(QPS);

if(rateLimiter.tryAcquire()){
System.out.println("Handle request");
}else{
System.out.println("Reject request");
}
```

核心字段：

- permitsPerSecond：每秒产生的令牌数；
- maxBurstSeconds： /** 桶中最多可以保存多少秒存入的令牌数 */
- maxPermits：最大令牌数 等于permitsPerSecond*maxBurstSeconds

- storedPermits：当前已存储的令牌数；
- **nextFreeTicketMicros：核心参数，表示下一个可以分配令牌的时间点；**可以是过去也可以是将来的时间点

核心思想 - 预分配

1. sleep到nextFreeTicketMicros这个时间点
2. 不管当前storedPermits是多少，此时tryAcquire都能分配到令牌
3. 将nextFreeTicketMicros推迟到nextFreeTicketMicros+(permits - storedPermits)*stableIntervalMicros; 这样即使我现在storedPermits的不够分配的，但是将下一次分配的时间点向后推，在这个时间点前不会再分配令牌，这样也能达到限流效果。

这里说的是一个思路，主要以理解为准，一些边界或者WarmUp实现的细节就忽略了。

说白了，不是我们传统思路的，攒够了N个令牌后，再分配。而是先分配，然后将下次分配的时间移后。

举例说明：

假设每秒可分配10个令牌，当前时间为2s，所以已存储了20个令牌，现在我要分配30个令牌，一种做法是等待1秒，在第3秒的时候攒够30个了，然后分配返回分配令牌；而Guava的做法是，直接返回，然后将下一个分配令牌的时间点定为第3秒。

这两种情况都实现了3秒钟分配30个令牌的QPS需求

核心方法

create(double permitsPerSecond)

create(double permitsPerSecond) 方法时，创建的是 SmoothBursty 实例，默认设置 maxBurstSeconds 为1s。SleepingStopwatch 是guava中的一个时钟类实现。

并通过调用 SmoothBursty.doSetRate(double, long) 方法进行初始化

java 复制代码

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

SmoothBursty(SleepingStopwatch stopwatch, double maxBurstSeconds) {
    super(stopwatch);
    this.maxBurstSeconds = maxBurstSeconds;
}
```

- 在node中不需要实现对应的create实例，直接new一个class即可。

java 复制代码

```
class GuavaRateLimiter {

  constructor(permitsPerSecond) {

    this.permitsPerSecond = permitsPerSecond // 每秒产生的令牌数

    this.maxBurstSeconds = 1 // 桶中最多可以保存1秒的令牌数

    this.rateLimiter = new SmoothBursty(new SleepingStopwatch(), this.maxBurstSeconds)


    this.rateLimiter.setRate(permitsPerSecond)

  }


  setRate(permitsPerSecond) {

    this.rateLimiter.setRate(permitsPerSecond)

    Logger.info('update rate : ' + permitsPerSecond)

  }

}
```

doSetRate(double, long)

doSetRate方法中:

1. 调用 resync(nowMicros) 对 storedPermits 与 nextFreeTicketMicros 进行了调整——如果当前时间晚于 nextFreeTicketMicros，则计算这段时间内产生的令牌数，累加到 storedPermits 上，并更新下次可获取令牌时间 nextFreeTicketMicros 为当前时间。

2. 计算 stableIntervalMicros 的值，1/permitsPerSecond。

3. 调用 doSetRate(double, double) 方法计算 maxPermits 值（maxBurstSeconds*permitsPerSecond），并根据旧的 maxPermits 值对 storedPermits 进行调整。

java 复制代码

```java
@Override
final void doSetRate(double permitsPerSecond, long nowMicros) {
        resync(nowMicros);
        double stableIntervalMicros = SECONDS.toMicros(1L) / permitsPerSecond;
        this.stableIntervalMicros = stableIntervalMicros;
        doSetRate(permitsPerSecond, stableIntervalMicros);
}

/** Updates {@code storedPermits} and {@code nextFreeTicketMicros} based on the current time. */
void resync(long nowMicros) {
        // if nextFreeTicket is in the past, resync to now
        if (nowMicros > nextFreeTicketMicros) {
        double newPermits = (nowMicros - nextFreeTicketMicros) / coolDownIntervalMicros();
        storedPermits = min(maxPermits, storedPermits + newPermits);
        nextFreeTicketMicros = nowMicros;
        }
}

@Override
void doSetRate(double permitsPerSecond, double stableIntervalMicros) {
        double oldMaxPermits = this.maxPermits;
        maxPermits = maxBurstSeconds * permitsPerSecond;
        if (oldMaxPermits == Double.POSITIVE_INFINITY) {
                // 什么场景下会这么大？普通的应用遇不到
                storedPermits = maxPermits;
        } else {
                storedPermits =
                        (oldMaxPermits == 0.0)
                                ? 0.0 // initial state
                                : storedPermits * maxPermits / oldMaxPermits;
        }
```

在node中实现如下：

java 复制代码

```
/**
 * 恒定速率令牌限流器
 */
class SmoothRateLimiter {
    constructor(stopwatch) {
        this.stopwatch = stopwatch
        this.storedPermits = 0 // double
        this.maxPermits = 0 // double
        this.stableIntervalMicros = 0 // double
        this.nextFreeTicketMicros = 0 // long
    }

    setRate(permitsPerSecond) {
        let nowMicros = this.stopwatch.readMicros()
        this.resync(nowMicros)
        this.stableIntervalMicros = 1000000 / permitsPerSecond // double
        this.doSetRate(permitsPerSecond, this.stableIntervalMicros)
    }

    queryEarliestAvailable(nowMicros) {
        debug(this.nextFreeTicketMicros + ':' + nowMicros)
```

```
}

reserveEarliestAvailable(requiredPermits, nowMicros) {
    this.resync(nowMicros)
    let returnValue = this.nextFreeTicketMicros
    let storedPermitsToSpend = min(requiredPermits, this.storedPermits)
    let freshPermits = requiredPermits - storedPermitsToSpend
    debug('freshPermits : ' + freshPermits)
    let waitMicros = freshPermits * this.stableIntervalMicros
    debug('waitMicros : ' + waitMicros)
    debug('nextFreeTicketMicros : ' + this.nextFreeTicketMicros)
    this.nextFreeTicketMicros += waitMicros
    this.storedPermits -= storedPermitsToSpend
    debug('storedPermits : ' + this.storedPermits)
    debug('maxPermits : ' + this.maxPermits)
    return returnValue
}

/**
 * 延迟计算
 * @param {*} nowMicros
 */
resync(nowMicros) {
    if (nowMicros > this.nextFreeTicketMicros) {
```

```
            this.nextFreeTicketMicros = nowMicros

            debug('resync : ' + this.nextFreeTicketMicros + ' : ' + nowMicros + ' : ' + this.storedPermits)

        }

    }

}

/** 恒速令牌桶 */

class SmoothBursty extends SmoothRateLimiter {

    constructor(stopwatch, maxBurstSeconds) {

        super(stopwatch)

        this.maxBurstSeconds = maxBurstSeconds

    }


    doSetRate(permitsPerSecond, stableIntervalMicros) {

        let oldMaxPermits = this.maxPermits

        this.maxPermits = this.maxBurstSeconds * permitsPerSecond

        this.storedPermits = (oldMaxPermits === 0.0) ? 0.0 : this.storedPermits * this.maxPermits / oldMaxPermits

    }

}
```

acquire(int)核心的方法

调用 acquire(int) 方法获取指定数量的令牌，

1. 调用 reserve(int) 方法，该方法最终调用 reserveEarliestAvailable(int, long) 。来更新下次可取令牌时间点与当前存储的令牌数，并返回本次可取令牌的时间点，根据该时间点计

2. 阻塞等待（1）中返回的等待时间

3. 返回等待的时间（秒）

java 复制代码

```java
/** 获取指定数量（permits）的令牌，阻塞直到获取到令牌，返回等待的时间*/
@CanIgnoreReturnValue
public double acquire(int permits) {
        long microsToWait = reserve(permits);
        stopwatch.sleepMicrosUninterruptibly(microsToWait);
        return 1.0 * microsToWait / SECONDS.toMicros(1L);
}
// 预先申请额度，但这个额度需要时间到了才可以使用。具体是否需要等待，取决于返回的时间值
final long reserve(int permits) {
        checkPermits(permits);
        synchronized (mutex()) {
                return reserveAndGetWaitLength(permits, stopwatch.readMicros());
        }
}

/** 本次获取令牌仍返回需要等待的时间*/
final long reserveAndGetWaitLength(int permits, long nowMicros) {
        long momentAvailable = reserveEarliestAvailable(permits, nowMicros);
        return max(momentAvailable - nowMicros, 0);
}

/** 针对此次需要获取的令牌数更新下次可取令牌时间点与存储的令牌数，返回本次可取令牌的时间点*/
@Override
final long reserveEarliestAvailable(int requiredPermits, long nowMicros) {
        resync(nowMicros); // 更新当前数据
        long returnValue = nextFreeTicketMicros;
        double storedPermitsToSpend = min(requiredPermits, this.storedPermits); // 本次可消费的令牌数
        double freshPermits = requiredPermits - storedPermitsToSpend; // 需要新增的令牌数
        long waitMicros =
                storedPermitsToWaitTime(this.storedPermits, storedPermitsToSpend)
                        + (long) (freshPermits * stableIntervalMicros); // 需要等待的时间

        this.nextFreeTicketMicros = LongMath.saturatedAdd(nextFreeTicketMicros, waitMicros); // 更新下次可取令
        this.storedPermits -= storedPermitsToSpend; // 更新当前存储的令牌数
        return returnValue;
}
```

tryAcquire(int,long,TimeUnit)

tryAcquire(int,long,TimeUnit) 方法则是在指定超时时间内尝试获取令牌，如果获取到或超时时间到则返回是否获取成功

1. 先判断是否能在指定超时时间内获取到令牌，通过 nextFreeTicketMicros <= nowMicros + timeoutMicros 是否为true来判断，即当前时间+超时时间 在可取令牌时间 之后，当可取（预消费的特性），否则不可获取。
2. 如果不可获取，立即返回false。
3. 如果可获取，则调用 reserveAndGetWaitLength(permits, nowMicros) 来更新下次可取令牌时间点与当前存储的令牌数，返回等待时间（逻辑与前面相同），并阻塞等待相应的时间，返回true。

- node实现

java 复制代码

```
class GuavaRateLimiter {
  /**
   * 在指定超时时间timeout内尝试获取令牌permits，如果获取到或超时时间到则返回是否获取成功
   * @param {int} permits
   * @param {int} timeout // MILLISECONDS 微秒
   */
  tryAcquire(permits, timeout) {
    let timeoutMicros = timeout * 1000
    let microsToWait = 0
    let nowMicros = this.rateLimiter.stopwatch.readMicros()
    if (!this.canAcquire(nowMicros, timeoutMicros)) {  //判断是否能在超时时间内获取指定数量的令牌
      return false
    } else {
```

```
    }

    this.rateLimiter.stopwatch.sleepMicrosUninterruptibly(microsToWait)

    return true

}


/**

 * 只要可取时间小于当前时间nowMicros + 超时时间 timeoutMicros，则可获取（可预消费的特性！）

 * @param {*} nowMicros

 * @param {*} timeoutMicros

 * @returns

 */

canAcquire(nowMicros, timeoutMicros) {

    return this.rateLimiter.queryEarliestAvailable(nowMicros) <= nowMicros + timeoutMicros

}


/**

 * 获取可发放令牌的时间

 * @param {*} permits

 * @param {*} nowMicros

 * @returns

 */

reserveAndGetWaitLength(permits, nowMicros) {

    let momentAvailable = this.rateLimiter.reserveEarliestAvailable(permits, nowMicros)

    return max(momentAvailable - nowMicros, 0) //保证不为负数
```

```
}

queryEarliestAvailable(nowMicros) {
   return this.nextFreeTicketMicros
}

reserveEarliestAvailable(requiredPermits, nowMicros) {
    this.resync(nowMicros)
        let returnValue = this.nextFreeTicketMicros
        let storedPermitsToSpend = min(requiredPermits, this.storedPermits)
        let freshPermits = requiredPermits - storedPermitsToSpend

        let waitMicros = freshPermits * this.stableIntervalMicros

        this.nextFreeTicketMicros += waitMicros
        this.storedPermits -= storedPermitsToSpend

        return returnValue
    }
```

java 复制代码

```
public boolean tryAcquire(int permits, long timeout, TimeUnit unit) {
    // 将timeout换算成微秒
    long timeoutMicros = max(unit.toMicros(timeout), 0);
    checkPermits(permits);
```

```
    synchronized (mutex()) {
      long nowMicros = stopwatch.readMicros();
      // 判断是否可以在timeoutMicros时间范围内获取令牌
      if (!canAcquire(nowMicros, timeoutMicros)) {
        return false;
      } else {
        // 获取令牌，并返回需要等待的毫秒数
        microsToWait = reserveAndGetWaitLength(permits, nowMicros);
      }
    }
    // 等待microsToWait时间
    stopwatch.sleepMicrosUninterruptibly(microsToWait);
    return true;
  }
```

上述是尝试获取令牌的代码，其中关键函数是reserveAndGetWaitLength

java 复制代码

```
  final long reserveAndGetWaitLength(int permits, long nowMicros) {
    long momentAvailable = reserveEarliestAvailable(permits, nowMicros);
    return max(momentAvailable - nowMicros, 0);
  }
  @Override
  final long reserveEarliestAvailable(int requiredPermits, long nowMicros) {
    resync(nowMicros);
```

```
        long returnValue = nextFreeTicketMicros;
        // 计算需要用掉多少令牌
        double storedPermitsToSpend = min(requiredPermits, this.storedPermits);
        // 如果需要获取的大于暂存的令牌，计算还欠多少令牌
        double freshPermits = requiredPermits - storedPermitsToSpend;
        // 计算需要多久才能攒够欠下的令牌
        long waitMicros =
            storedPermitsToWaitTime(this.storedPermits, storedPermitsToSpend)
                + (long) (freshPermits * stableIntervalMicros);
        // 最妙的一点，赊账！！不像我们想的那样sleep(waitMicros)，而是将下一个可分配令牌的时间点向后挪
        this.nextFreeTicketMicros = LongMath.saturatedAdd(nextFreeTicketMicros, waitMicros);
        this.storedPermits -= storedPermitsToSpend;
        return returnValue;
    }
```

以上就是 SmoothBursty 实现的基本处理流程。注意两点：

1. RateLimiter 通过限制后面请求的等待时间，来支持一定程度的突发请求——预消费的特性。

2. RateLimiter 令牌桶的实现并不是起一个线程不断往桶里放令牌，而是以一种延迟计算的方式（参考resync函数），在每次获取令牌之前计算该段时间内可以产生多少令牌，将产生的令牌加入令牌桶中并更新数据来实现，比起一个线程来不断往桶里放令牌高效得多。（想想如果需要针对每个用户限制某个接口的访问，则针对每个用户都得创建一个RateLimiter，并起一个线程来控制令牌存放的话，如果在线用户数有几十上百万，起线程来控制是一件多么恐怖的事情）

本文介绍了限流的三种基本算法，其中令牌桶算法与漏桶算法主要用来限制请求处理的速度，可将其归为限速，计数器算法则是用来限制一个时间窗口内请求处理的数量，可将其归为限量（对速度不限制）。Guava 的 RateLimiter 是令牌桶算法的一种实现，但 RateLimiter 只适用于单机应用，在分布式环境下就不适用了。虽然已有一些开源项目可用于分布式环境下的限流管理，如阿里的Sentinel，但对于小型项目来说，引入Sentinel可能显得有点过重，但限流的需求在小型项目中也是存在的.

参考：cidoliu.github.io/2021/02/24/...

分类： 前端 标签： Node.js

评论

输入评论（Enter换行，Ctrl + Enter发送）

相关推荐

李子捌 6月前 Redis 后端

架构师如何讲解Redis限流——令牌桶限流

902 13 3

超级爽朗的郑 2月前 分布式 Spring Cloud

Gateway+Redis实现令牌桶限流算法

1180 10 2

艾小仙 7月前 面试 后端

面试官：你说说限流的原理?

5184 80 2

蛮三刀酱 2年前 架构

【秒杀系统】零基础上手秒杀系统（二）：令牌桶限流 + 再谈超卖

狂奔滴小马　1月前　Node.js

我用 nodejs 爬了一万多张小姐姐壁纸

4.2w　429　108

顽疾　8月前　后端　算法

限流算法-令牌桶算法｜ 8月更文挑战

735　1　评论

已注销　2年前　Java

限流降级神器，带你解读阿里巴巴开源 Sentinel 实现原理

544　4　评论

逐九　1年前　Go

限流-令牌桶实现(go版本)

521　1　评论

TEAVAMC　1年前　分布式

基于 Redis 和 Lua 实现分布式令牌桶限流

1117　6　2

万俊峰Kevin　3月前　Go　微服务

Go 分布式令牌桶限流 + 兜底策略

1788　7　评论

天天天天君　2年前　Go

漏水桶和令牌桶的限流算法

275　1　2

哪能一直都快乐　8月前　后端

基于RateLimiter和Redis+Lua实现的限流组件

750　9　评论

SpringBoot中文社区　1年前　Spring Boot

在springboot中使用Guava基于令牌桶实现限流

PHP进阶架构师　1年前　PHP

基于PHP+Redis令牌桶限流

1047　点赞　评论

山鸡1　1年前　Redis

Redis——限流算法之滑动窗口、漏斗限流的原理及java实现

4325　7　4

yes的练级攻略　2年前　Java

面试官：说说降级、熔断、限流

1.5w　74　5

小圆规　1月前　面试　设计

面试官问：限流或RateLimit要怎么设计

745　7　3

低调的码农　2年前　架构

高并发下漏洞桶限流设计方案 - Redis

2666　26　6

Alone381　3年前　Node.js　前端

前端的焦虑，你想过30岁以后的前端路怎么走吗?

4.4w　648　354

小帅不太帅　3月前　GitHub　Node.js

😜微信每天自动给女[男]朋友发早安和土味情话

2.6w　296　93