分布式环境下限流方案的实现redis RateLimiter Guava,Token Bucket, Leaky Bucket

- 业务背景介绍
  对于web应用的限流，光看标题，似乎过于抽象，难以理解，那我们还是以具体的某一个应用场景来引入这个话题吧。
  在日常生活中，我们肯定收到过不少不少这样的短信，“双11约吗？，千款….”，“您有幸获得唱读卡，赶快戳链接…”。这种类型的短信是属于推广性质的短信。为什么我要说这个呢？听我慢慢道来。
  一般而言，对于推广营销类短信，它们针对某一群体(譬如注册会员)进行定点推送，有时这个群体的成员量比较大，譬如京东的会员，可以达到千万级别。因此相应的，发送推广短信的量也会增大。然而，要完成这些短信发送，我们是需要调用服务商的接口来完成的。倘若一次发送的量在200万条，而我们的服务商接口每秒能处理的短信发送量有限，只能达到200条每秒。那么这个时候就会产生问题了，我们如何能控制好程序发送短信时的速度呢？于是限流这个功能就得加上了
- 生产环境背景
  1、服务商接口所能提供的服务上限是400条/s
  2、业务方调用短信发送接口的速度未知，QPS可能达到800/s，1200/s，或者更高
  3、当服务商接口访问频率超过400/s时，超过的量将拒绝服务，多出的信息将会丢失
  4、线上为多节点布置，但调用的是同一个服务商接口
- 需求分析
  1、鉴于业务方对短信发送接口的调用频率未知，而服务商的接口服务有上限，为保证服务的可用性，业务层需要对接口调用方的流量进行限制——接口限流
- 需求设计
  方案一、在提供给业务方的Controller层进行控制。
  1、使用guava提供工具库里的RateLimiter类(内部采用令牌捅算法实现)进行限流

```
<!--核心代码片段-->
private RateLimiter rateLimiter = RateLimiter.create(400);//400表示每秒允许处理的量是400
 if(rateLimiter.tryAcquire()) {
   //短信发送逻辑可以在此处

 }
```

2、使用Java自带delayqueue的延迟队列实现(编码过程相对麻烦，此处省略代码)

3、使用Redis实现，存储两个key，一个用于计时，一个用于计数。请求每调用一次，计数器增加1，若在计时器时间内计数器未超过阈值，则可以处理任务

```
 if(!cacheDao.hasKey(API_WEB_TIME_KEY)) {
cacheDao.putToValue(API_WEB_TIME_KEY,0,(long)1, TimeUnit.SECONDS);
}
if(cacheDao.hasKey(API_WEB_TIME_KEY)&&cacheDao.incrBy(API_WEB_COUNTER_KEY,(long)1) > (long)400) {
    LOGGER.info("调用频率过快");
}
//短信发送逻辑
```

方案二、在短信发送至服务商时做限流处理
方案三、同时使用方案一和方案二

- 可行性分析
  最快捷且有效的方式是使用RateLimiter实现，但是这很容易踩到一个坑，单节点模式下，使用RateLimiter进行限流一点问题都没有。但是…线上是分布式系统，布署了多个节点，而且多个节点最终调用的是同一个短信服务商接口。虽然我们对单个节点能做到将QPS限制在400/s，但是多节点条件下，如果每个节点均是400/s，那么到服务商那边的总请求就是*节点数*x400/s，于是限流效果失效。使用该方案对单节点的阈值控制是难以适应分布式环境的，至少目前我还没想到更为合适的方式。
  对于第二种，使用delayqueue方式。其实主要存在两个问题，1：短信系统本身就用了一层消息队列，有用kafka，或者rabitmq，如果再加一层延迟队列，从设计上来说是不太合适的。2：实现delayqueue的过程相对

较麻烦，耗时可能比较长，而且达不到精准限流的效果
对于第三种，使用redis进行限流，其很好地解决了分布式环境下多实例所导致的并发问题。因为使用redis设置的计时器和计数器均是全局唯一的，不管多少个节点，它们使用的都是同样的计时器和计数器，因此可以做到非常精准的流控。同时，这种方案编码并不复杂，可能需要的代码不超过10行。

- 实施方案
  根据可行性分析可知，整个系统采取redis限流处理是成本最低且最高效的。
  具体实现

  1、在Controller层设置两个全局key，一个用于计数，另一个用于计时

```
private static final String API_WEB_TIME_KEY = "time_key";

    private static final String API_WEB_COUNTER_KEY = "counter_key";
```

2、对时间key的存在与否进行判断，并对计数器是否超过阈值进行判断

```
if(!cacheDao.hasKey(API_WEB_TIME_KEY)) {

            cacheDao.putToValue(API_WEB_TIME_KEY,0,(long)1,
TimeUnit.SECONDS);
            cacheDao.putToValue(API_WEB_COUNTER_KEY,0,(long)2,
TimeUnit.SECONDS);//时间到就重新初始化为

        }


if(cacheDao.hasKey(API_WEB_TIME_KEY)&&cacheDao.incrBy(API_WEB_COUNTER_KEY,
(long)1) > (long)400) {


            LOGGER.info("调用频率过快");

        }
         //短信发送逻辑
```

实施结果

可以达到非常精准的流控，截图会在后续的过程中贴出来。欢迎有疑问的小伙伴们在评论区提出问题，我看到后尽量抽时间回答的

http://blog.csdn.net/Justnow_/article/details/53055299

一、场景描述

很多做服务接口的人或多或少的遇到这样的场景，由于业务应用系统的负载能力有限，为了防止非预期的请求对系统压力过大而拖垮业务应用系统。

也就是面对大流量时，如何进行流量控制？

服务接口的流量控制策略：分流、降级、限流等。本文讨论下限流策略，虽然降低了服务接口的访问频率和并发量，却换取服务接口和业务应用系统的高可用。

实际场景中常用的限流策略：

- Nginx前端限流

  按照一定的规则如帐号、IP、系统调用逻辑等在Nginx层面做限流

- 业务应用系统限流

  1、客户端限流

  2、服务端限流

- 数据库限流

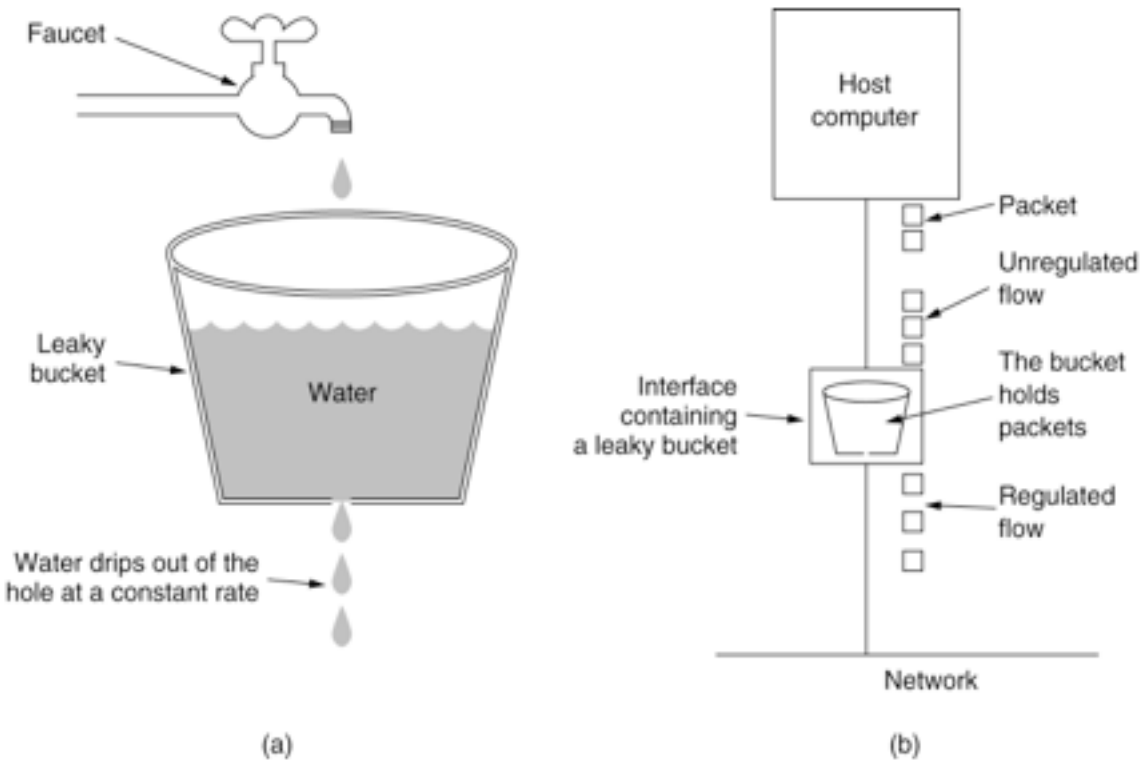
  红线区，力保数据库

二、常用的限流算法

常用的限流算法由：楼桶算法和令牌桶算法。本文不具体的详细说明两种算法的原理，原理会在接下来的文章中做说明。

1、漏桶算法

漏桶(Leaky Bucket)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(接口有响应速率),当水流入速度过大会直接溢出(访问频率超过接口响应速率),然后就拒绝请求,可以看出漏桶算法能强行限制数据的传输速率.示意图如下:

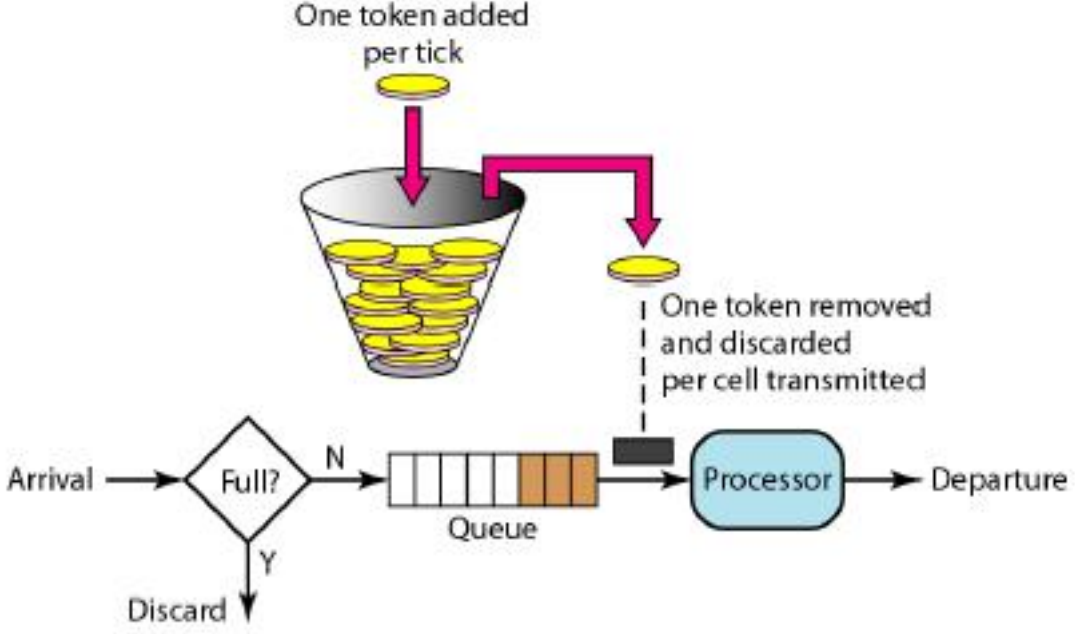


可见这里有两个变量,一个是桶的大小,支持流量突发增多时可以存多少的水(burst),另一个是水桶漏洞的大小(rate)。

因为漏桶的漏出速率是固定的参数,所以,即使网络中不存在资源冲突(没有发生拥塞),漏桶算法也不能使流突发(burst)到端口速率.因此,漏桶算法对于存在突发特性的流量来说缺乏效率.

2、令牌桶算法

令牌桶算法(Token Bucket)和 Leaky Bucket 效果一样但方向相反的算法,更加容易理解.随着时间流逝,系统会按恒定1/QPS时间间隔(如果QPS=100,则间隔是10ms)往桶里加入Token(想象和漏洞漏水相反,有个水龙头在不断的加水),如果桶已经满了就不再加了.新请求来临时,会各自拿走一个Token,如果没有Token可拿了就阻塞或者拒绝服务.



令牌桶的另外一个好处是可以方便的改变速度. 一旦需要提高速率,则按需提高放入桶中的令牌的速率. 一般会定时(比如100毫秒)往桶中增加一定数量的令牌, 有些变种算法则实时的计算应该增加的令牌的数量.

三、基于Redis功能的实现

简陋的设计思路：假设一个用户（用IP判断）每分钟访问某一个服务接口的次数不能超过10次，那么我们可以在Redis中创建一个键，并此时我们就设置键的过期时间为60秒，每一个用户对此服务接口的访问就把键值加1，在60秒内当键值增加到10的时候，就禁止访问服务接口。在某种场景中添加访问时间间隔还是很有必要的。

1）使用Redis的incr命令，将计数器作为Lua脚本

```
1 local current
2 current = redis.call("incr",KEYS[1])
3 if tonumber(current) == 1 then
4     redis.call("expire",KEYS[1],1)
5 end
```

Lua脚本在Redis中运行，保证了incr和expire两个操作的原子性。

2）使用Reids的列表结构代替incr命令

```
1 FUNCTION LIMIT_API_CALL(ip)
2 current = LLEN(ip)
3 IF current > 10 THEN
4     ERROR "too many requests per second"
5 ELSE
```

```
 6      IF EXISTS(ip) == FALSE
 7          MULTI
 8              RPUSH(ip,ip)
 9              EXPIRE(ip,1)
10          EXEC
11      ELSE
12          RPUSHX(ip,ip)
13      END
14      PERFORM_API_CALL()
15  END
```

Rate Limit使用Redis的列表作为容器，LLEN用于对访问次数的检查，一个事物中包含了RPUSH和EXPIRE两个命令，用于在第一次执行计数是创建列表并设置过期时间，

RPUSHX在后续的计数操作中进行增加操作。

四、基于令牌桶算法的实现

令牌桶算法可以很好的支撑突然额流量的变化即满令牌桶数的峰值。

```
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

import com.google.common.base.Preconditions;
import com.netease.datastream.util.framework.LifeCycle;

 20 public class TokenBucket implements LifeCycle {

// 默认桶大小个数 即最大瞬间流量是64M
 private static final int DEFAULT_BUCKET_SIZE = 1024 * 1024 * 64;

// 一个桶的单位是1字节
 private int everyTokenSize = 1;

// 瞬间最大流量
 private int maxFlowRate;
```

```
// 平均流量
 private int avgFlowRate;

// 队列来缓存桶数量：最大的流量峰值就是 = everyTokenSize*DEFAULT_BUCKET_SIZE 64M
= 1 * 1024 * 1024 * 64
 private ArrayBlockingQueue<Byte> tokenQueue = new
ArrayBlockingQueue<Byte>(DEFAULT_BUCKET_SIZE);

private ScheduledExecutorService scheduledExecutorService =
Executors.newSingleThreadScheduledExecutor();

private volatile boolean isStart = false;

private ReentrantLock lock = new ReentrantLock(true);

private static final byte A_CHAR = 'a';

public TokenBucket() {
 }

public TokenBucket(int maxFlowRate, int avgFlowRate) {
 this.maxFlowRate = maxFlowRate;
 this.avgFlowRate = avgFlowRate;
 }

public TokenBucket(int everyTokenSize, int maxFlowRate, int avgFlowRate) {
 this.everyTokenSize = everyTokenSize;
 this.maxFlowRate = maxFlowRate;
 this.avgFlowRate = avgFlowRate;
 }

public void addTokens(Integer tokenNum) {

// 若是桶已经满了，就不再家如新的令牌
 for (int i = 0; i < tokenNum; i++) {
 tokenQueue.offer(Byte.valueOf(A_CHAR));
 }
 }

public TokenBucket build() {

start();
 return this;
 }

/**
 * 获取足够的令牌个数
```

```
 *
 * @return
 */
 public boolean getTokens(byte[] dataSize) {

Preconditions.checkNotNull(dataSize);
 Preconditions.checkArgument(isStart, "please invoke start method first
!");

int needTokenNum = dataSize.length / everyTokenSize + 1;// 传输内容大小对应的
桶个数

final ReentrantLock lock = this.lock;
 lock.lock();
 try {
 boolean result = needTokenNum <= tokenQueue.size(); // 是否存在足够的桶数量
 if (!result) {
 return false;
 }

int tokenCount = 0;
 for (int i = 0; i < needTokenNum; i++) {
 Byte poll = tokenQueue.poll();
 if (poll != null) {
 tokenCount++;
 }
 }

return tokenCount == needTokenNum;
 } finally {
 lock.unlock();
 }
 }

@Override
 public void start() {

// 初始化桶队列大小
 if (maxFlowRate != 0) {
 tokenQueue = new ArrayBlockingQueue<Byte>(maxFlowRate);
 }

// 初始化令牌生产者
 TokenProducer tokenProducer = new TokenProducer(avgFlowRate, this);
 scheduledExecutorService.scheduleAtFixedRate(tokenProducer, 0, 1,
TimeUnit.SECONDS);
 isStart = true;
```

```
}

@Override
 public void stop() {
 isStart = false;
 scheduledExecutorService.shutdown();
 }

@Override
 public boolean isStarted() {
 return isStart;
 }

class TokenProducer implements Runnable {

private int avgFlowRate;
 private TokenBucket tokenBucket;

public TokenProducer(int avgFlowRate, TokenBucket tokenBucket) {
 this.avgFlowRate = avgFlowRate;
 this.tokenBucket = tokenBucket;
 }

@Override
 public void run() {
 tokenBucket.addTokens(avgFlowRate);
 }
 }

public static TokenBucket newBuilder() {
 return new TokenBucket();
 }

public TokenBucket everyTokenSize(int everyTokenSize) {
 this.everyTokenSize = everyTokenSize;
 return this;
 }

public TokenBucket maxFlowRate(int maxFlowRate) {
 this.maxFlowRate = maxFlowRate;
 return this;
 }

public TokenBucket avgFlowRate(int avgFlowRate) {
 this.avgFlowRate = avgFlowRate;
 return this;
 }
```

```
private String stringCopy(String data, int copyNum) {

StringBuilder sbuilder = new StringBuilder(data.length() * copyNum);

for (int i = 0; i < copyNum; i++) {
 sbuilder.append(data);
 }

return sbuilder.toString();

}

public static void main(String[] args) throws IOException,
InterruptedException {

tokenTest();
 }

private static void arrayTest() {
 ArrayBlockingQueue<Integer> tokenQueue = new ArrayBlockingQueue<Integer>
(10);
 tokenQueue.offer(1);
 tokenQueue.offer(1);
 tokenQueue.offer(1);
 System.out.println(tokenQueue.size());
 System.out.println(tokenQueue.remainingCapacity());
 }

private static void tokenTest() throws InterruptedException, IOException {
 TokenBucket tokenBucket =
TokenBucket.newBuilder().avgFlowRate(512).maxFlowRate(1024).build();

BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(new FileOutputStream("/tmp/ds_test")));
 String data = "xxxx";// 四个字节
 for (int i = 1; i <= 1000; i++) {

Random random = new Random();
 int i1 = random.nextInt(100);
 boolean tokens = tokenBucket.getTokens(tokenBucket.stringCopy(data,
i1).getBytes());
 TimeUnit.MILLISECONDS.sleep(100);
 if (tokens) {
 bufferedWriter.write("token pass --- index:" + i1);
 System.out.println("token pass --- index:" + i1);
 } else {
 bufferedWriter.write("token rejuect --- index" + i1);
 System.out.println("token rejuect --- index" + i1);
```

```
 }

bufferedWriter.newLine();
 bufferedWriter.flush();
 }

bufferedWriter.close();
 }

}
```

参考：

http://xiaobaoqiu.github.io/blog/2015/07/02/ratelimiter/

http://redisdoc.com/string/incr.html

http://www.cnblogs.com/zhengyun_ustc/archive/2012/11/17/topic1.html

http://www.cnblogs.com/exceptioneye/p/4783904.html

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。缓存的目的是提升系统访问速度和增大系统能处理的容量，可谓是抗高并发流量的银弹；而降级是当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开；而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页），因此需有一种手段来限制这些场景的并发/请求量，即限流。

限流的目的是通过对并发访问/请求进行限速或者一个时间窗口内的的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或告知资源没有了）、排队或等待（比如秒杀、评论、下单）、降级（返回兜底数据或默认数据，如商品详情页库存默认有货）。

一般开发高并发系统常见的限流有：限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如nginx的limit_conn模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit_req模块，限制每秒的平均速率）；其他还有如限制远程接口调用速率、限制MQ的消费速率。另外还可以根据网络连接数、网络流量、CPU或

内存负载等来限流。

先有缓存这个银弹，后有限流来应对618、双十一高并发流量，在处理高并发问题上可以说是如虎添翼，不用担心瞬间流量导致系统挂掉或雪崩，最终做到有损服务而不是不服务；限流需要评估好，不可乱用，否则会正常流量出现一些奇怪的问题而导致用户抱怨。

在实际应用时也不要太纠结算法问题，因为一些限流算法实现是一样的只是描述不一样；具体使用哪种限流技术还是要根据实际场景来选择，不要一味去找最佳模式，白猫黑猫能解决问题的就是好猫。

因在实际工作中遇到过许多人来问如何进行限流，因此本文会详细介绍各种限流手段。那么接下来我们从限流算法、应用级限流、分布式限流、接入层限流来详细学习下限流技术手段。

限流算法

常见的限流算法有：令牌桶、漏桶。计数器也可以进行粗暴限流实现。

令牌桶算法

令牌桶算法是一个存放固定容量令牌的桶，按照固定速率往桶里添加令牌。令牌桶算法的描述如下：

- 假设限制2r/s，则按照500毫秒的固定速率往桶中添加令牌；
- 桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃或拒绝；
- 当一个n个字节大小的数据包到达，将从桶中删除n个令牌，接着数据包被发送到网络上；
- 如果桶中的令牌不足n个，则不会删除令牌，且该数据包将被限流（要么丢弃，要么缓冲区等待）。

漏桶算法

漏桶作为计量工具（The Leaky Bucket Algorithm as a Meter）时，可以用于流量整形（Traffic Shaping）和流量控制（TrafficPolicing），漏桶算法的描

述如下：

- 一个固定容量的漏桶，按照常量固定速率流出水滴；
- 如果桶是空的，则不需流出水滴；
- 可以以任意速率流入水滴到漏桶；
- 如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。

令牌桶和漏桶对比：

- 令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；
- 漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；
- 令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；
- 漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；
- 令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；
- 两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

另外有时候我们还使用计数器来进行限流，主要用来限制总并发数，比如数据库连接池、线程池、秒杀的并发数；只要全局总请求数或者一定时间段的总请求数设定的阀值则进行限流，是简单粗暴的总数量限流，而不是平均速率限流。

到此基本的算法就介绍完了，接下来我们首先看看应用级限流。

应用级限流

限流总并发/连接/请求数

对于一个应用系统来说一定会有极限并发/请求数，即总有一个TPS/QPS阀值，如果超了阀值则系统就会不响应用户请求或响应的非常慢，因此我们最好进行过载保护，防止大量请求涌入击垮系统。

如果你使用过Tomcat，其Connector 其中一种配置有如下几个参数：

acceptCount：如果Tomcat的线程都忙于响应，新来的连接会进入队列排队，如果超出排队大小，则拒绝连接；

maxConnections： 瞬时最大连接数，超出的会排队等待；

maxThreads：Tomcat能启动用来处理请求的最大线程数，如果请求处理量一直远远大于最大线程数则可能会僵死。

详细的配置请参考官方文档。另外如Mysql（如max_connections）、Redis（如tcp-backlog）都会有类似的限制连接数的配置。

限流总资源数

如果有的资源是稀缺资源（如数据库连接、线程），而且可能有多个系统都会去使用它，那么需要限制应用；可以使用池化技术来限制总资源数：连接池、线程池。比如分配给每个应用的数据库连接是100，那么本应用最多可以使用100个资源，超出了可以等待或者抛异常。

限流某个接口的总并发/请求数

如果接口可能会有突发访问情况，但又担心访问量太大造成崩溃，如抢购业务；这个时候就需要限制这个接口的总并发/请求数总请求数了；因为粒度比较细，可以为每个接口都设置相应的阀值。可以使用Java中的AtomicLong进行限流：

==========

```
try {
    if(atomic.incrementAndGet() > 限流数) {
        //拒绝请求
    }
    //处理请求
} finally {
    atomic.decrementAndGet();
}
```

==========

适合对业务无损的服务或者需要过载保护的服务进行限流，如抢购业务，超出了大小要么让用户排队，要么告诉用户没货了，对用户来说是可以接受的。而一些开放平台也会限制用户调用某个接口的试用请求量，也可以用这种计数器方式实现。这种方式也是简单粗暴的限流，没有平滑处理，需要根据实际情况选择使用；

限流某个接口的时间窗请求数

即一个时间窗口内的请求数，如想限制某个接口/服务每秒/每分钟/每天的请求数/调用量。如一些基础服务会被很多其他系统调用，比如商品详情页服务会调用基础商品服务调用，但是怕因为更新量比较大将基础服务打挂，这时我们要对每秒/每分钟的调用量进行限速；一种实现方式如下所示：

==========

```
LoadingCache<Long, AtomicLong> counter =
        CacheBuilder.newBuilder()
                .expireAfterWrite(2, TimeUnit.SECONDS)
                .build(new CacheLoader<Long, AtomicLong>() {
                    @Override
                    public AtomicLong load(Long seconds) throws Exception
{
                        return new AtomicLong(0);
                    }
                });
long limit = 1000;
while(true) {
    //得到当前秒
    long currentSeconds = System.currentTimeMillis() / 1000;
    if(counter.get(currentSeconds).incrementAndGet() > limit) {
        System.out.println("限流了:" + currentSeconds);
```

```
            continue;
        }
        //业务处理
    }
```

==========

我们使用Guava的Cache来存储计数器，过期时间设置为2秒（保证1秒内的计数器是有的），然后我们获取当前时间戳然后取秒数来作为KEY进行计数统计和限流，这种方式也是简单粗暴，刚才说的场景够用了。

平滑限流某个接口的请求数

之前的限流方式都不能很好地应对突发请求，即瞬间请求可能都被允许从而导致一些问题；因此在一些场景中需要对突发请求进行整形，整形为平均速率请求处理（比如5r/s，则每隔200毫秒处理一个请求，平滑了速率）。这个时候有两种算法满足我们的场景：令牌桶和漏桶算法。Guava框架提供了令牌桶算法实现，可直接拿来使用。

Guava 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。

SmoothBursty

==========

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
```

将得到类似如下的输出：

0.0

0.198239

0.196083

0.200609

0.199599

0.19961

==========

1、RateLimiter.create(5) 表示桶容量为5且每秒新增5个令牌，即每隔200毫秒新增一个令牌；

2、limiter.acquire()表示消费一个令牌，如果当前桶中有足够令牌则成功（返回值为0），如果桶中没有令牌则暂停一段时间，比如发令牌间隔是200毫秒，则等待200毫秒后再去消费令牌（如上测试用例返回的为0.198239，差不多等待了200毫秒桶中才有令牌可用），这种实现将突发请求速率平均为了固定请求速率。

再看一个突发示例：

==========

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(5));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1))
```

将得到类似如下的输出：

0.0

0.98745

0.183553

0.199909

==========

limiter.acquire(5)表示桶的容量为5且每秒新增5个令牌，令牌桶算法允许一定程度的突发，所以可以一次性消费5个令牌，但接下来的limiter.acquire(1)将等待差不多1秒桶中才能有令牌，且接下来的请求也整形为固定速率了。

==========

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(10));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1));
```

将得到类似如下的输出：

```
0.0

1.997428

0.192273

0.200616
```

==========

同上边的例子类似，第一秒突发了10个请求，令牌桶算法也允许了这种突发（允许消费未来的令牌），但接下来的limiter.acquire(1)将等待差不多2秒桶中才能有令牌，且接下来的请求也整形为固定速率了。

接下来再看一个突发的例子：

==========

```
RateLimiter limiter = RateLimiter.create(2);

System.out.println(limiter.acquire());
Thread.sleep(2000L);
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
```

System.*out*.println(limiter.acquire());
System.*out*.println(limiter.acquire());

将得到类似如下的输出：

0.0

0.0

0.0

0.0

0.499876

0.495799

==========

1、创建了一个桶容量为2且每秒新增2个令牌；

2、首先调用limiter.acquire()消费一个令牌，此时令牌桶可以满足（返回值为0）；

3、然后线程暂停2秒，接下来的两个limiter.acquire()都能消费到令牌，第三个limiter.acquire()也同样消费到了令牌，到第四个时就需要等待500毫秒了。

此处可以看到我们设置的桶容量为2（即允许的突发量），这是因为SmoothBursty中有一个参数：最大突发秒数（maxBurstSeconds）默认值是1s，突发量/桶容量=速率*maxBurstSeconds，所以本示例桶容量/突发量为2，例子中前两个是消费了之前积攒的突发量，而第三个开始就是正常计算的了。令牌桶算法允许将一段时间内没有消费的令牌暂存到令牌桶中，留待未来使用，并允许未来请求的这种突发。

SmoothBursty通过平均速率和最后一次新增令牌的时间计算出下次新增令牌的时间的，另外需要一个桶暂存一段时间内没有使用的令牌（即可以突发的令牌数）。另外RateLimiter还提供了tryAcquire方法来进行无阻塞或可超时的令牌消费。

因为SmoothBursty允许一定程度的突发，会有人担心如果允许这种突发，假设突然间来了很大的流量，那么系统很可能扛不住这种突发。因此需要一种平滑速率的限流工具，从而系统冷启动后慢慢的趋于平均固定速率（即刚开始速率小一些，然后慢慢趋于我们设置的固定速率）。Guava也提供了SmoothWarmingUp来实现这种需求，其可以认为是漏桶算法，但是在某些特殊场景又不太一样。

SmoothWarmingUp创建方式：RateLimiter.create(doublepermitsPerSecond, long warmupPeriod, TimeUnit unit)

permitsPerSecond表示每秒新增的令牌数，warmupPeriod表示在从冷启动速率过渡到平均速率的时间间隔。

示例如下：

==========

```
RateLimiter limiter = RateLimiter.create(5, 1000, TimeUnit.MILLISECONDS);
for(int i = 1; i < 5;i++) {
    System.out.println(limiter.acquire());
}
Thread.sleep(1000L);
for(int i = 1; i < 5;i++) {
    System.out.println(limiter.acquire());
}
```

将得到类似如下的输出：

```
0.0

0.51767

0.357814

0.219992

0.199984
```

0.0

0.360826

0.220166

0.199723

0.199555

==========

速率是梯形上升速率的，也就是说冷启动时会以一个比较大的速率慢慢到平均速率；然后趋于平均速率（梯形下降到平均速率）。可以通过调节warmupPeriod参数实现一开始就是平滑固定速率。

到此应用级限流的一些方法就介绍完了。假设将应用部署到多台机器，应用级限流方式只是单应用内的请求限流，不能进行全局限流。因此我们需要分布式限流和接入层限流来解决这个问题。

分布式限流

分布式限流最关键的是要将限流服务做成原子化，而解决方案可以使使用redis+lua或者nginx+lua技术进行实现，通过这两种技术可以实现的高并发和高性能。

首先我们来使用redis+lua实现时间窗内某个接口的请求数限流，实现了该功能后可以改造为限流总并发/请求数和限制总资源数。Lua本身就是一种编程语言，也可以使用它实现复杂的令牌桶或漏桶算法。

redis+lua实现中的lua脚本：

==========

```
local key = KEYS[1] --限流KEY（一秒一个）
local limit = tonumber(ARGV[1])       --限流大小
local current = tonumber(redis.call("INCRBY", key, "1")) --请求数+1
if current > limit then --如果超出限流大小
```

```
  return 0
elseif current == 1 then  --只有第一次访问需要设置2秒的过期时间
  redis.call("expire", key,"2")
end
return 1
```

==========

如上操作因是在一个lua脚本中，又因Redis是单线程模型，因此是线程安全的。如上方式有一个缺点就是当达到限流大小后还是会递增的，可以改造成如下方式实现：

==========

```
local key = KEYS[1] --限流KEY（一秒一个）
local limit = tonumber(ARGV[1])       --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
  return 0
else  --请求数+1，并设置2秒过期
  redis.call("INCRBY", key,"1")
  redis.call("expire", key,"2")
  return 1
end
```

==========

如下是Java中判断是否需要限流的代码：

==========

```
public static boolean acquire() throws Exception {
    String luaScript = Files.toString(new File("limit.lua"),
Charset.defaultCharset());
    Jedis jedis = new Jedis("192.168.147.52", 6379);
    String key = "ip:" + System.currentTimeMillis()/ 1000; //此处将当前时间戳取秒数
```

```
    Stringlimit = "3"; //限流大小
    return (Long)jedis.eval(luaScript,Lists.newArrayList(key),
Lists.newArrayList(limit)) == 1;
}
```

==========

因为Redis的限制（Lua中有写操作不能使用带随机性质的读操作，如TIME）不能在Redis Lua中使用TIME获取时间戳，因此只好从应用获取然后传入，在某些极端情况下（机器时钟不准的情况下），限流会存在一些小问题。

使用Nginx+Lua实现的Lua脚本：

==========

```
local locks = require "resty.lock"

local function acquire()
    local lock =locks:new("locks")
    local elapsed, err =lock:lock("limit_key") --互斥锁
    local limit_counter =ngx.shared.limit_counter --计数器

    local key = "ip:" ..os.time()
    local limit = 5 --限流大小
    local current =limit_counter:get(key)

    if current ~= nil and current + 1> limit then --如果超出限流大小
        lock:unlock()
        return 0
    end
    if current == nil then
        limit_counter:set(key, 1, 1) --第一次需要设置过期时间，设置key的值为1，过期时间为1秒
    else
         limit_counter:incr(key, 1) --第二次开始加1即可
```

```
    end
    lock:unlock()
    return 1
end
ngx.print(acquire())
```

==========

实现中我们需要使用lua-resty-lock互斥锁模块来解决原子性问题(在实际工程中使用时请考虑获取锁的超时问题)，并使用ngx.shared.DICT共享字典来实现计数器。如果需要限流则返回0，否则返回1。使用时需要先定义两个共享字典（分别用来存放锁和计数器数据）：

==========

```
http {
    ……
    lua_shared_dict locks 10m;
    lua_shared_dict limit_counter 10m;
}
```

==========

有人会纠结如果应用并发量非常大那么redis或者nginx是不是能抗得住；不过这个问题要从多方面考虑：你的流量是不是真的有这么大，是不是可以通过一致性哈希将分布式限流进行分片，是不是可以当并发量太大降级为应用级限流；对策非常多，可以根据实际情况调节；像在京东使用Redis+Lua来限流抢购流量，一般流量是没有问题的。

对于分布式限流目前遇到的场景是业务上的限流，而不是流量入口的限流；流量入口限流应该在接入层完成，而接入层笔者一般使用Nginx。

参考资料

https://en.wikipedia.org/wiki/Token_bucket

https://en.wikipedia.org/wiki/Leaky_bucket

http://redis.io/commands/incr

http://nginx.org/en/docs/http/ngx_http_limit_req_module.html

http://nginx.org/en/docs/http/ngx_http_limit_conn_module.html

https://github.com/openresty/lua-resty-limit-traffic

http://nginx.org/en/docs/http/ngx_http_core_module.html#limit_rate

http://www.blogjava.net/stevenjohn/archive/2016/06/14/430882.html

http://www.mincoder.com/article/2943.shtml