

秒杀系统

1. 秒杀系统

1.1 秒杀场景

- 电商抢购限量商品
- 卖周董演唱会的门票
- 火车票抢座 12306
-

1.2 为什么要做个系统

如果你的项目流量非常小，完全不用担心有并发的购买请求，那么做这样一个系统意义不大。但如果你的系统要像12306那样，接受高并发访问和下单的考验，那么你就需要一套完整的流程保护措施，来保证你系统在用户流量高峰期不会被搞挂了。

- 严格防止超卖：库存100件你卖了120件，等着辞职吧
- 防止黑产：防止不怀好意的人群通过各种技术手段把你本该下发给群众的利益全收入了囊中。
- 保证用户体验：高并发下，别网页打不开了，支付不成功了，购物车进不去了，地址改不了了。这个问题非常之大，涉及到各种技术，也不是一下子就能讲完的，甚至根本就没法讲完。

1.3 保护措施有哪些

- 乐观锁防止超卖 ---核心基础
- 令牌桶限流
- Redis 缓存
- 消息队列异步处理订单
-

2. 防止超卖

毕竟，你网页可以卡住最多是大家没参与到活动，上网口吐芬芳，骂你一波。但是你要是卖多了，本该拿到商品的用户可就不乐意了，轻则投诉你，重则找漏洞起诉赔偿。让你吃不了兜着走。


2.1 数据库表

```
-- -----  
-- Table structure for stock  
-- -----  
  
DROP TABLE IF EXISTS `stock`;  
CREATE TABLE `stock` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `name` varchar(50) NOT NULL DEFAULT '' COMMENT '名称',  
  `count` int(11) NOT NULL COMMENT '库存',
```

```
`sale` int(11) NOT NULL COMMENT '已售',
`version` int(11) NOT NULL COMMENT '乐观锁, 版本号',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-----
-- Table structure for stock_order
-----
DROP TABLE IF EXISTS `stock_order`;
CREATE TABLE `stock_order` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `sid` int(11) NOT NULL COMMENT '库存ID',
  `name` varchar(30) NOT NULL DEFAULT '' COMMENT '商品名称',
  `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT '创建时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2.2 分析业务

image-20200326184815171

2.3 开发代码

1. DAO代码

```
public interface StockDAO {
    Stock checkStock(Integer id); //校验库存
    void updateSale(Stock stock); //扣除库存
}

public interface OrderDAO {
    void createOrder(Order order); //创建订单
}
```

2. Service 代码

```
@Service
@Transactional
public class OrderServiceImpl implements OrderService {
    @Autowired
    private OrderDAO orderDAO;
    @Autowired
    private StockDAO stockDAO;
    @Override
    public Integer createOrder(Integer id) {
        //校验库存
```

```
        Stock stock = checkStock(id);
        //扣库存
        updateSale(stock);
        //下订单
        return createOrder(stock);
    }
    //校验库存
    private Stock checkStock(Integer id) {
        Stock stock = stockDAO.checkStock(id);
        if (stock.getSale().equals(stock.getCount())) {
            throw new RuntimeException("库存不足");
        }
        return stock;
    }
    //扣库存
    private void updateSale(Stock stock){
        stock.setSale(stock.getSale() + 1);
        stockDAO.updateSale(stock);
    }
    //下订单
    private Integer createOrder(Stock stock){
        Order order = new Order();
        order.setSid(stock.getId());
        order.setCreateDate(new Date());
        order.setName(stock.getName());
        orderDAO.createOrder(order);
        return order.getId();
    }
}
```

5. Controller代码

```
@RestController
@RequestMapping("stock")
public class StockController {
    @Autowired
    private OrderService orderService;
    //秒杀方法
    @GetMapping("sale")
    public String sale(Integer id){
        int orderId = 0;
        try{
            //根据商品id创建订单,返回创建订单的id
            orderId = orderService.createOrder(id);
            System.out.println("orderId = " + orderId);
            return String.valueOf(orderId);
        }catch (Exception e){
            e.printStackTrace();
            return e.getMessage();
        }
    }
}
```

```
}  
}
```

2.4 正常测试

在正常测试下发现没有任何问题

2.5 使用Jmeter进行压力测试

官网: <https://jmeter.apache.org/>

1. 介绍

Apache JMeter是Apache组织开发的基于Java的压力测试工具。用于对软件做压力测试，它最初被设计用于Web应用测试，但后来扩展到其他测试领域。它可以用于测试静态和动态资源，例如静态文件、Java 小程序、CGI 脚本、Java 对象、数据库、FTP 服务器，等等。JMeter 可以用于对服务器、网络或对象模拟巨大的负载，来自不同压力类别下测试它们的强度和分析整体性能。另外，JMeter能够对应用程序做功能/回归测试，通过创建带有断言的脚本来验证你的程序返回了你期望的结果

2. 安装Jmeter

```
# 1.下载jmeter  
https://jmeter.apache.org/download_jmeter.cgi  
下载地址:https://mirror.bit.edu.cn/apache//jmeter/binaries/apache-jmeter-  
5.2.1.tgz  
# 2.解压缩  
backups      ---用来对压力测试进行备份目录  
bin          ---Jmeter核心执行脚本文件  
docs         ---官方文档和案例  
extras       ---额外的扩展  
lib          ---第三方依赖库  
licenses     ---说明  
printable_docs ---格式化文档  
  
# 3.安装Jmeter  
0.要求: 必须事先安装jdk环境  
1.配置jmeter环境变量  
export JMETER_HOME=/Users/chenyannan/dev/apache-jmeter-5.2  
export  
PATH=$SCALA_HOME/bin:$JAVA_HOME/bin:$GRADLE_HOME/bin:$PATH:$JMETER_HOME/bin  
2.是配置生效  
source ~/.bash_profile  
3.测试jemeter
```

3. Jmeter使用

Don't use GUI mode for load testing !, only for Test creation and Test debugging.

For load testing, use CLI Mode (was NON GUI):

```
jmeter -n -t [jmx file] -l [results file] -e -o [Path to web report folder]
```

& increase Java Heap to meet your test requirements:

Modify current env variable HEAP="-Xms1g -Xmx1g -XX:MaxMetaspaceSize=256m" in the jmeter batch file

Check : <https://jmeter.apache.org/usermanual/best-practices.html>

4. Jmeter压力测试

```
jmeter -n -t [jmx file](jmx压力测试文件) -l [results file](结果输出的文件) -e -o  
[Path to web report folder](生成html版压力测试报告)
```

2.6 乐观锁解决商品超卖问题

说明: 使用乐观锁解决商品的超卖问题,实际上是把主要防止超卖问题交给数据库解决,利用数据库中定义的 **version** 字段以及数据库中的 **事务** 实现在并发情况下商品的超卖问题。

0. 校验库存的方法(不变)

```
//校验库存  
private Stock checkStock(Integer id){  
    Stock stock = stockDAO.checkStock(id);  
    if(stock.getSale().equals(stock.getCount())){  
        throw new RuntimeException("库存不足!!!");  
    }  
    return stock;  
}
```

```
<!--根据秒杀商品id查询库存-->  
    <select id="checkStock" parameterType="int" resultType="Stock">  
        select id,name,count,sale,version from stock  
        where id = #{id}  
    </select>
```

1. 更新库存方法改造

```
//扣除库存  
private void updateSale(Stock stock){  
    //在sql层面完成销量的+1 和 版本号+ 并且根据商品id和版本号同时查询更新的商品  
    stockDAO.updateSale(stock);  
}
```

```
<update id="updateSale" parameterType="Stock">
    update stock set
        sale=sale+1,
        version=version+1
    where
        id =#{id}
        and
        version = #{version}
</update>
```


2. 创建订单

```
//创建订单
private Integer createOrder(Stock stock){
    Order order = new Order();
    order.setSid(stock.getId()).setName(stock.getName()).setCreateDate(new Date());
    orderDAO.createOrder(order);
    return order.getId();
}
```


```
<!--创建订单-->
<insert id="createOrder" parameterType="Order" useGeneratedKeys="true"
keyProperty="id" >
    insert into stock_order values(#{id},#{sid},#{name},#{createDate})
</insert>
```

3. 完整的业务方法与Mapper.xml


- Service方法

image-20200328162237475

- StockDAOMapper.xml

image-20200328162319138

- OrderDAOMapper.xml

- image-20200328162404085

3. 接口限流

限流:是对某一时间窗口内的请求数进行限制,保持系统的可用性和稳定性,防止因流量暴增而导致的系统运行缓慢或宕机

3.1 接口限流

在面临高并发的抢购请求时，我们如果不对接口进行限流，可能会对后台系统造成极大的压力。大量的请求抢购成功时需要调用下单的接口，过多的请求打到数据库会对系统的稳定性造成影响。

3.2 如何解决接口限流

常用的限流算法有**令牌桶**和**漏桶(漏斗算法)**，而Google开源项目Guava中的RateLimiter使用的就是令牌桶控制算法。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：缓存的目的是提升系统访问速度和增大系统处理容量
- 降级：降级是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行
- 限流：限流的目的是通过对并发访问/请求进行限速，或者对一个时间窗口内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、排队或等待、降级等处理。

3.3 令牌桶和漏斗算法

image-20200401091230233

- **漏斗算法**: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直接溢出，可以看出漏桶算法能强行限制数据的传输速率。
- **令牌桶算法**:最初来源于计算机网络。在网络传输数据时，为了防止网络拥塞，需限制流出网络的流量，使流量以比较均匀的速度向外发送。令牌桶算法就实现了这个功能，可控制发送到网络上数据的数目，并允许突发数据的发送。大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。这意味，面对瞬时大流量，该算法可以在短时间内请求拿到大量令牌，而且拿令牌的过程并不是消耗很大的事情。

3.4 令牌桶简单使用

1. 项目中引入依赖

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>28.2-jre</version>
</dependency>
```

2. 令牌桶算法的基本使用

```
public class StockController {
    @Autowired
    private OrderService orderService;

    //创建令牌桶实例
    private RateLimiter rateLimiter = RateLimiter.create(40);
```

```

    @GetMapping("sale")
    public String sale(Integer id){
        //1.没有获取到token请求一直知道获取到token 令牌
        //log.info("等待的时间: "+ rateLimiter.acquire());

        //2.设置一个等待时间,如果在等待的时间内获取到了token 令牌,则处理业务,如果在等待
        时间内没有获取到响应token则抛弃
        if(!rateLimiter.tryAcquire(2, TimeUnit.SECONDS)){
            System.out.println("当前请求被限流,直接抛弃,无法调用后续秒杀逻辑....");
            return "抢购失败!";
        }
        System.out.println("处理业务.....");
        return "抢购成功";
    }
}

```

3.5使用令牌桶算法实现乐观锁+限流

1. 使用令牌桶改造controller实现乐观锁+限流

```

//开发一个秒杀方法 乐观锁防止超卖+ 令牌桶算法限流
@GetMapping("killtoken")
public String killtoken(Integer id){
    System.out.println("秒杀商品的id = " + id);
    //加入令牌桶的限流措施
    if(!rateLimiter.tryAcquire(3, TimeUnit.SECONDS)){
        log.info("抛弃请求: 抢购失败,当前秒杀活动过于火爆,请重试");
        return "抢购失败,当前秒杀活动过于火爆,请重试!";
    }
    try {
        //根据秒杀商品id 去调用秒杀业务
        int orderId = orderService.kill(id);
        return "秒杀成功,订单id为: " + String.valueOf(orderId);
    }catch (Exception e){
        e.printStackTrace();
        return e.getMessage();
    }
}
}

```

4. 隐藏秒杀接口

在前几次课程中,我们完成了防止超卖商品和抢购接口的限流,已经能够防止大流量把我们的服务器直接搞炸,这篇文章中,我们要开始关心一些细节问题。我们现在设计的系统还有一些问题:

1. 我们应该在一定的时间内执行秒杀处理,不能再任意时间都接受秒杀请求。如何加入时间验证?

2. 对于稍微懂点电脑的，又会动歪脑筋的人来说开始通过抓包方式获取我们的接口地址。然后通过脚本进行抢购怎么办？
3. 秒杀开始之后如何限制单个用户的请求频率，即单位时间内限制访问次数？


这个章节主要讲解秒杀系统中，关于抢购（下单）接口相关的单用户防刷措施，主要说几块内容：

- 限时抢购
- 抢购接口隐藏
- 单用户限制频率（单位时间内限制访问次数）

4.1 限时抢购的实现

使用Redis来记录秒杀商品的时间,对秒杀过期的请求进行拒绝处理!!

1. 启动redis服务

image-20200424205958039

2. 将秒杀商品放入Redis并设置超时

这里我们使用String类型 以kill + 商品id作为key 以商品id作为value,设置180秒超时(可随意设置时间)

```
127.0.0.1:6379> set kill1 1 EX 180
OK
```

image-20200424210057315

3. 抢购中加入时间控制

- 整合当前项目操作redis服务,这里使用spring-boot-starter-data-redis操作Redis,引入依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- 修改配置连接redis

```
spring.redis.port=6379
spring.redis.host=localhost
spring.redis.database=0
```

- 通过redis控制抢购超时的请求

```
@Service
@Transactional
public class OrderServiceImpl implements OrderService {
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public Integer createOrder(Integer id) {
        //redis校验抢购时间
        if(!stringRedisTemplate.hasKey("kill" + id)){
            throw new RuntimeException("秒杀超时,活动已经结束啦!!!");
        }
        //校验库存
        Stock stock = checkStock(id);
        //扣库存
        updateSale(stock);
        //下订单
        return createOrder(stock);
    }
}
```

4.2 抢购接口隐藏

对于稍微懂点电脑的，又会动歪脑筋的人来说，点击F12打开浏览器的控制台，就能在点击抢购按钮后，获取我们抢购接口的链接。（手机APP等其他客户端可以抓包来拿到）一旦坏蛋拿到了抢购的链接，只要稍微写点爬虫代码，模拟一个抢购请求，就可以不通过点击下单按钮，直接在代码中请求我们的接口，完成下单。所以就有了成千上万的薅羊毛军团，写一些脚本抢购各种秒杀商品。

他们只需要在抢购时刻的000毫秒，开始不间断发起大量请求，觉得比大家在APP上点抢购按钮要快，毕竟人的速度又极限，更别说APP说不定还要经过几层前端验证才会真正发出请求。

所以我们需要将抢购接口进行隐藏，**抢购接口隐藏（接口加盐）的具体做法：**

- 每次点击秒杀按钮，先从服务器获取一个秒杀验证值（接口内判断是否到秒杀时间）。
- Redis以缓存用户ID和商品ID为Key，秒杀地址为Value缓存验证值
- 用户请求秒杀商品的时候，要带上秒杀验证值进行校验。
- 具体流程:


image-20200615202456610

1.库表结构

image-20200615204044522

```
SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;
--
-- Table structure for user
--
```

```
DROP TABLE IF EXISTS `user`;  
CREATE TABLE `user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',  
  `name` varchar(80) DEFAULT NULL COMMENT '用户名',  
  `password` varchar(40) DEFAULT NULL COMMENT '用户密码',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;  
  
SET FOREIGN_KEY_CHECKS = 1;
```

image-20200615205655259

2.控制器代码

```
//生成md5值的方法  
@RequestMapping("md5")  
public String getMd5(Integer id, Integer userid) {  
  String md5;  
  try {  
    md5 = orderService.getMd5(id, userid);  
  } catch (Exception e) {  
    e.printStackTrace();  
    return "获取md5失败: "+e.getMessage();  
  }  
  return "获取md5信息为: "+md5;  
}
```

image-20200615205443846

3.业务层代码


```
@Override  
public String getMd5(Integer id, Integer userid) {  
  //检验用户的合法性  
  User user = userDAO.findById(userid);  
  if(user==null) throw new RuntimeException("用户信息不存在!");  
  log.info("用户信息: [{}]", user.toString());  
  //检验商品的合法性  
  Stock stock = stockDAO.checkStock(id);  
  if(stock==null) throw new RuntimeException("商品信息不合法!");  
  log.info("商品信息: [{}]", stock.toString());  
  //生成hashkey  
  String hashKey = "KEY_"+userid+"_"+id;  
  //生成md5//这里!Q*jS#是一个盐 随机生成  
  String key = DigestUtils.md5DigestAsHex((userid+id+"!Q*jS#").getBytes());  
  stringRedisTemplate.opsForValue().set(hashKey, key, 3600, TimeUnit.SECONDS);  
  log.info("Redis写入: [{}] [{}]", hashKey, key);  
  return key;  
}
```

 image-20200615205501408


4.DAO代码和Entity

```
@Data
public class User {
    private Integer id;
    private String name;
    private String password;
}
```

```
@Mapper
public interface UserDAO {
    User findById(Integer id);
}
```

 image-20200615205542311

```
<select id="findById" parameterType="Integer" resultType="User">
    select id,name,password from user where id=#{id}
</select>
```

 image-20200615205618586

5.数据库添加用户记录

 image-20200615205708746

6.查看商品信息

 image-20200615205728734

7.启动项目访问生成md5接口

 image-20200615205810462

8.携带验证值验证下单即可

1.controller代码

 image-20200615214741078

```
//开发一个秒杀方法 乐观锁防止超卖+ 令牌桶算法限流
@GetMapping("killtokenmd5")
public String killtoken(Integer id,Integer userid,String md5) {
    System.out.println("秒杀商品的id = " + id);
    //加入令牌桶的限流措施
    if (!rateLimiter.tryAcquire(3, TimeUnit.SECONDS)) {
        log.info("抛弃请求: 抢购失败,当前秒杀活动过于火爆,请重试");
        return "抢购失败,当前秒杀活动过于火爆,请重试!";
    }
    try {
        //根据秒杀商品id 去调用秒杀业务
        int orderId = orderService.kill(id,userid,md5);
        return "秒杀成功,订单id为: " + String.valueOf(orderId);
    } catch (Exception e) {
        e.printStackTrace();
        return e.getMessage();
    }
}
```

2.service代码

```
@Override
public int kill(Integer id, Integer userid, String md5) {

    //校验redis中秒杀商品是否超时
    //      if(!stringRedisTemplate.hasKey("kill"+id))
    //          throw new RuntimeException("当前商品的抢购活动已经结束啦~~");

    //先验证签名
    String hashKey = "KEY_"+userid+"_"+id;
    String s = stringRedisTemplate.opsForValue().get(hashKey);
    if (s==null) throw new RuntimeException("没有携带验证签名,请求不合法!");
    if (!s.equals(md5)) throw new RuntimeException("当前请求数据不合法,请稍后再试!");

    //校验库存
    Stock stock = checkStock(id);
    //更新库存
    updateSale(stock);
    //创建订单
    return createOrder(stock);
}
```

image-20200615215224288

4.3 单用户限制频率

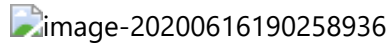
假设我们做好了接口隐藏,但是像我上面说的,总有无聊的人会写一个复杂的脚本,先请求hash(md5)值,再立刻请求购买,如果你的app下单按钮做的很差,大家都要开抢后0.5秒才能请求成功,那可能会让脚本依然能够在大家前面抢购成功。

我们需要在做一个额外的措施，来限制单个用户的抢购频率。

其实很简单的就能想到用redis给每个用户做访问统计，甚至是带上商品id，对单个商品做访问统计，这都是可行的。

我们先实现一个对用户的访问频率限制，我们在用户申请下单时，检查用户的访问次数，超过访问次数，则不让他下单！

- 具体流程



1.controller代码

```
//开发一个秒杀方法 乐观锁防止超卖+ 令牌桶算法限流
@GetMapping("killtokenmd5limit")
public String killtokenlimit(Integer id,Integer userid,String md5) {
    //加入令牌桶的限流措施
    if (!rateLimiter.tryAcquire(3, TimeUnit.SECONDS)) {
        log.info("抛弃请求: 抢购失败,当前秒杀活动过于火爆,请重试");
        return "抢购失败,当前秒杀活动过于火爆,请重试!";
    }
    try {
        //加入单用户限制调用频率
        int count = userService.saveUserCount(userid);
        log.info("用户截至该次的访问次数为: [{}]", count);
        boolean isBanned = userService.getUserCount(userid);
        if (isBanned) {
            log.info("购买失败,超过频率限制!");
            return "购买失败, 超过频率限制!";
        }
        //根据秒杀商品id 去调用秒杀业务
        int orderId = orderService.kill(id,userid,md5);
        return "秒杀成功,订单id为: " + String.valueOf(orderId);
    } catch (Exception e) {
        e.printStackTrace();
        return e.getMessage();
    }
}
```




2.Service接口及实现

接口

```
public interface UserService {
    //向redis中写入用户访问次数
    int saveUserCount(Integer userId);
    //判断单位时间调用次数
```

```
        boolean getUserCount(Integer userId);  
    }  
}
```

image-20200616191030293

实现

```
@Service  
@Transactional  
@Slf4j  
public class UserServiceImpl implements UserService{  
  
    @Autowired  
    private StringRedisTemplate stringRedisTemplate;  
  
    @Override  
    public int saveUserCount(Integer userId) {  
        //根据不同用户id生成调用次数的key  
        String limitKey = "LIMIT" + "_" + userId;  
        //获取redis中指定key的调用次数  
        String limitNum = stringRedisTemplate.opsForValue().get(limitKey);  
        int limit = -1;  
        if (limitNum == null) {  
            //第一次调用放入redis中设置为0  
            stringRedisTemplate.opsForValue().set(limitKey, "0", 3600,  
TimeUnit.SECONDS);  
        } else {  
            //不是第一次调用每次+1  
            limit = Integer.parseInt(limitNum) + 1;  
            stringRedisTemplate.opsForValue().set(limitKey, String.valueOf(limit),  
3600, TimeUnit.SECONDS);  
        }  
        return limit;//返回调用次数  
    }  
  
    @Override  
    public boolean getUserCount(Integer userId) {  
        String limitKey = "LIMIT" + "_" + userId;  
        //跟库用户调用次数的key获取redis中调用次数  
        String limitNum = stringRedisTemplate.opsForValue().get(limitKey);  
        if (limitNum == null) {  
            //为空直接抛弃说明key出现异常  
            log.error("该用户没有访问申请验证值记录, 疑似异常");  
            return true;  
        }  
        return Integer.parseInt(limitNum) > 10; //false代表没有超过 true代表超过  
    }  
}
```

image-20200616192013277

3.测试调用

image-20200616204841045