**商城秒杀**

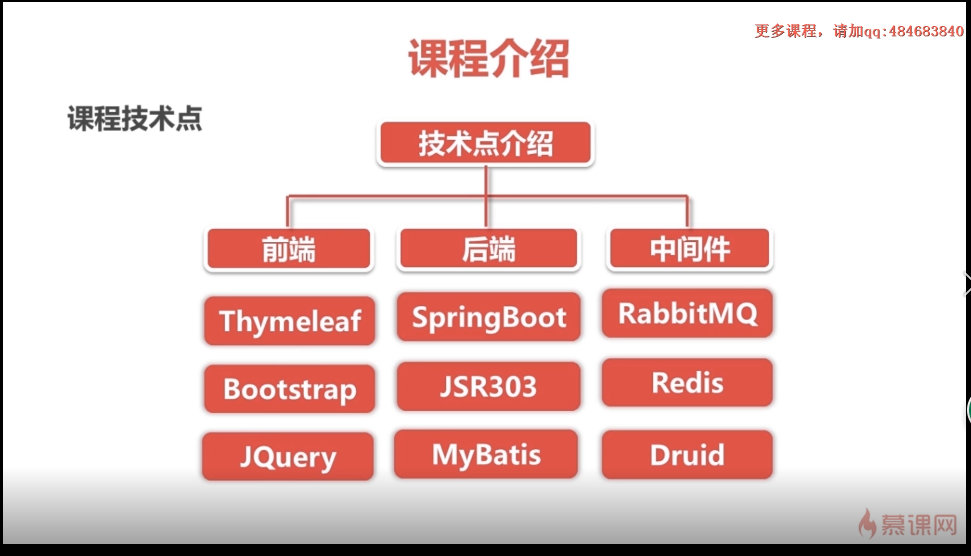
**一、项目铺垫**

**1.项目目标**

高并发网站下的应对

利用缓存和异步，分布式负载均衡(Nginx)

**2.技术栈**



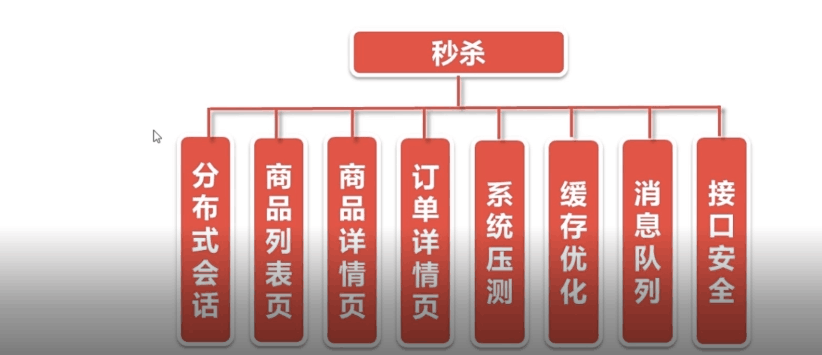
**Thymeleaf:页面模板**

**RabbitMQ:消息队列**

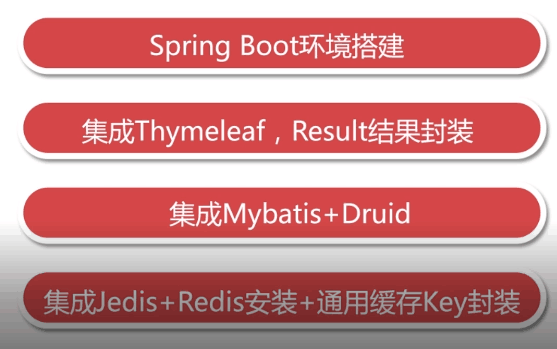
**Druid:数据库连接池**

**JSR303:服务端验证框架，做参数校验**

**3.概述**



**二、项目框架搭建**



Springboot取代SpringMvc, Springboot零配置

Result:封装返回结果

Thymeleaf:做页面模板

Redis相关通用缓存Key封装：实现类—》抽象类—》接口（模板模式）

**三、实现登录功能**



1. **数据库设计**

**create table miaosha\_user**

**(**

**id BIGINT(20) primary key not null auto\_increment,**

**nickname VARCHAR(255) NOT NULL,**

**pwd VARCHAR(32) DEFAULT NULL, //MD5（MD5（pass明文+固定salt） + 随机salt）,两次MD5，第一次防止用户密码在网络端被盗，第二次防止数据库被反推**

**salt VARCHAR(10) DEFAULT NULL,**

**head VARCHAR(128) DEFAULT NULL, //头像**

**register\_date DATETIME DEFAULT NULL,**

**last\_login\_time DATETIME DEFAULT NULL,**

**login\_count INT(11) DEFAULT 0 //登录次数**

**)** **ENGINE=INNODB DEFAULT CHARSET=utf8;**

a为啥两次MD5，http是明文协议，现将用户端的明文密码MD5，再把MD5之后的给服务端再次MD5，再把MD5和salt同时给服务器

b.用户登录

bootstrap 画页面

jquery-va.. 做form表单验证

layer 弹框

**c.JSR303参数校验**

不用每次都要参数校验

1. 在参数前面打上标签

**public** Result<Boolean> doLogin(HttpServletResponse response, @Valid LoginVo loginVo)

**2.在需要校验的地方打上标签**

@NotNull  
@IsMobile  
**private** String **mobile**;  
  
@NotNull  
@Length(min=32)  
**private** String **password**;

1. 看ismobeil类

d.异常拦截

拦截相应异常输出相对友好的提示

GlobalException

E．分布式session

实际过程中有多台服务器，用户一开始的session在第一台服务器上，但是第二次请求被分配到第二台服务器，此时session就会全部丢失：可以session同步，但是性能不高，实现复杂。通常用分布式session

*//登录成功后，给用户生成一个token标识此用户写入cookie传递给客户端，客户端在随后的访问中在  
// cookie中上传此token，服务端根据此token取用户的session信息*

基于cookie共享的session分布式,这里的session并没有存到容器，而是存入了缓存，用redis来管理session，因为每个用户的token是唯一的，用其做key，管理器session，客户端在随后的访问中通过每次用cookie携带token发送，服务端根据token找到对应的用户，这样即使换了服务器，原来的session也不会丢失，这就是分布式缓存。

cookie是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器之后向同一服务器再次发起请求时被携带上，用于告知服务端两个请求是否来自同一浏览器。不同的浏览器储存cookie的位置不同，解析方式也不同

像电商这种场景，一般一个浏览器只允许一个用户登录，切换用户时，就是登录新的用户，就又给这个新用户生成一个sessionId(即随机生成的串token来标识这个用户)，写到cookie中，要替换调，因为是不同用户了

**四、实现秒杀功能**



1. 数据库设计：商品表，订单表，秒杀商品表，秒杀订单表

商品表

CREATE TABLE goods

(

id BIGINT(20) PRIMARY KEY NOT NULL AUTO\_INCREMENT,

goods\_name VARCHAR(16) DEFAULT NULL,

goods\_title VARCHAR(64) DEFAULT NULL,

goods\_img VARCHAR(64) DEFAULT NULL,

goods\_detail LONGTEXT,

goods\_price DECIMAL(10,2) DEFAULT '0.00',

goods\_stock INT(11) DEFAULT '0' COMMENT'库存'

)ENGINE=INNODB DEFAULT CHARSET=utf8;

秒杀商品表

CREATE TABLE miaosha\_goods

(

id BIGINT(20) PRIMARY KEY NOT NULL AUTO\_INCREMENT COMMENT'秒杀的商品表',

goods\_id BIGINT(20) DEFAULT NULL COMMENT'商品id',

miaosha\_price DECIMAL(10,2) DEFAULT '0.00',

stock\_count INT(11) DEFAULT NULL,

start\_date DATETIME DEFAULT NULL,

end\_date DATETIME DEFAULT NULL，

CONSTRAINT `fk\_id` FOREIGN KEY (`goods\_id`) REFERENCES `goods`(`id`),

)ENGINE=INNODB DEFAULT CHARSET=utf8;

订单表

CREATE TABLE order\_info

(

id BIGINT(20) PRIMARY KEY NOT NULL AUTO\_INCREMENT,

user\_id BIGINT(20) DEFAULT NULL,

goods\_id BIGINT(20) DEFAULT NULL,

/\*收获地址\*/

delivery\_addr\_id BIGINT(20) DEFAULT NULL,

goods\_name VARCHAR(16) DEFAULT NULL,

goods\_count INT(11) DEFAULT '0',

goods\_price DECIMAL(10,2) DEFAULT '0.00',

order\_channel TINYINT(4) DEFAULT '0' COMMENT'1pc,2Android,3ios',

STATUS TINYINT(4) DEFAULT '0' COMMENT'订单状态，0新建未支付，1已支付，2已发货，3已收货，4已退款，5已完成',

create\_date DATETIME DEFAULT NULL,

pay\_date DATETIME DEFAULT NULL

)ENGINE=INNODB DEFAULT CHARSET=utf8;

秒杀订单表

CREATE TABLE miaosha\_order

(

id BIGINT(20) PRIMARY KEY NOT NULL AUTO\_INCREMENT,

user\_id BIGINT(20) DEFAULT NULL,

order\_id BIGINT(20) DEFAULT NULL,

goods\_id BIGINT(20) DEFAULT NULL

)ENGINE=INNODB DEFAULT CHARSET=utf8;

1. 页面设计：商品列表页，商品详情页，订单详情页

商品列表页：goods\_list

商品详情页：goods\_detail

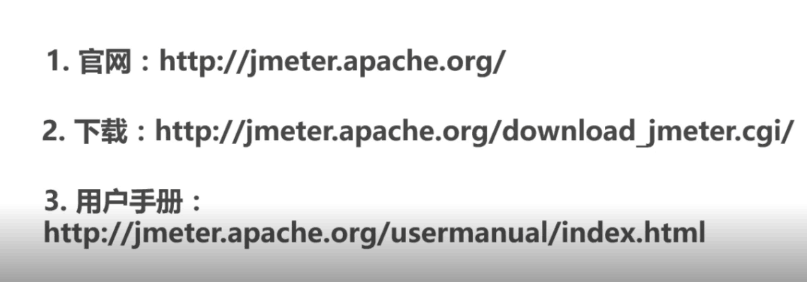
c.秒杀操作

**五、JMeter压测**



检查系统能承受多大的QPS(并发，“[每秒查询率](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=144099820&ss_c=ssc.citiao.link)”，是一台服务器每秒能够响应的查询次数，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准)

a.JMeter入门

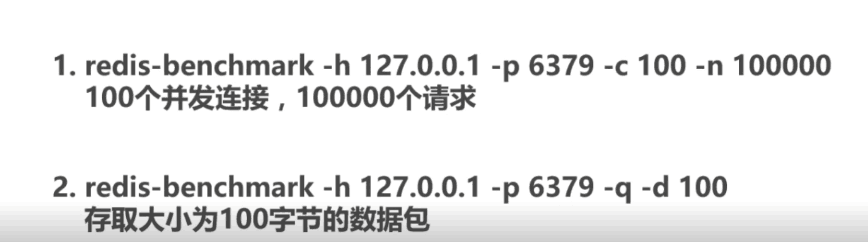


按照步骤来，看聚合报告中的吞吐量，通过查看cpu的负载，发现是mysql占用最高，可以优化，对use/info压测发现读缓存比读数据库qps高很多

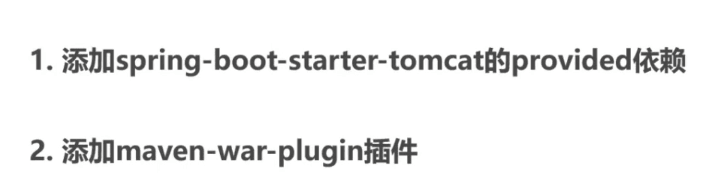
b.自定义变量

config.txt配置文件模拟用户

c.redis压测（Linux下）



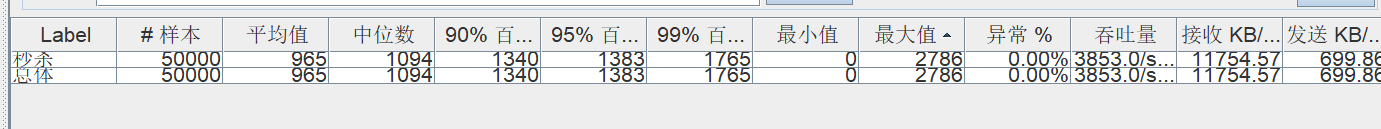
d.springboot打包war包



打包jar包：cd到项目路径执行mvn clean package

优化前：miaosha/do\_miaosha 5000\*10,第一次执行，改变数据库信息2400QPS

第二次执行不改变数据库信息，3342QPS



不安全：多用户访问，秒杀可能会把库存减成负数，原因是例如只剩一个库存，A线程判断满足条件可以秒杀，此时B线程进入也可以秒杀，秒杀了， A线程继续秒杀(判断库存小于0之前已经判断了),造成库存为-1

商品超卖问题是秒杀系统中除了性能优化，另一个最为关注的问题之一。由于我们的代码在生产环境中面临着超高并发的挑战，考虑业务需求我们也几乎不会对控制层代码进行加锁保证线程安全。所以，不做其他操作的情况下，极有可能发生商品超卖的问题。

秒杀商品和正常下单的逻辑是一致的，一旦确定可以下单了，业务逻辑主要包含两件事：

扣减库存 (前提库存程序判断肯定是够的，注意是程序，不代表真的够);

创建秒杀订单，或者正常订单，反正是要落地的。

上面两步必须要么全部成功，要么全部失败。很容易想到，我们可以通过事务机制来保证这一原子特性**。**

**六、页面优化技术(为了减少对数据库的访问，提升高并发性能**



a.加各种缓存，减少与数据库交互，实现高并发

页面缓存：由于并发瓶颈在数据库，想办法如何减少对数据库的访问，所以加若干缓存来提高，通过各种粒度的缓存，最大粒度页面缓存到最小粒度的对象级缓存。

1.取缓存 （缓存里面存的是html）  
2.手动渲染模板  
3.结果输出（直接输出html代码）

**GoodsController里面的toList方法改造一下，**当访问goods\_list页面的时候，如果从缓存中取到就返回这个html,(这里方法的返回格式已经设置为text/html，这样就是返回html的源代码)，如果取不到，利用ThymeleafViewResolver的getTemplateEngine().process和我们获取到的数据，渲染模板，并且在返回到前端之前保存至缓存里面，然后之后再来获取的时候，只要缓存里面存的goods\_list页面的html还没有过期，那么直接返回给前端即可。缓存后就不需要重复从数据库获取html页面模板，在大用户同时访问比较有用。注意缓存时间不能过长，否则页面时效性弱，60s即可(一分钟内做修改，页面看不到改变)

一般这个页面缓存时间，也不会很长，防止数据的时效性很低。但是可以防止短时间大并发访问

**URL缓存**

这里的url缓存相当于页面缓存，针对不同的详情页显示不同缓存页面，对不同的url进行缓存（redisService.set(GoodsKey.getGoodsDetail, “”+goodsId, html)，与页面缓存实质一样。

**对象缓存**

相比页面缓存是更细粒度缓存。在实际项目中， 不会大规模使用页面缓存，对象缓存就是当用到用户数据的时候，可以从缓存中取出。比如：更新用户密码，根据token来获取用户缓存对象。

MiaoshaUserService里面增加getById方法，先去取缓存，如果缓存中拿不到，那么就去取数据库，然后再设置到缓存中去

*对象缓存，更新数据库与缓存，数据库更新缓存一定也要更新，一定保证数据一致性，*

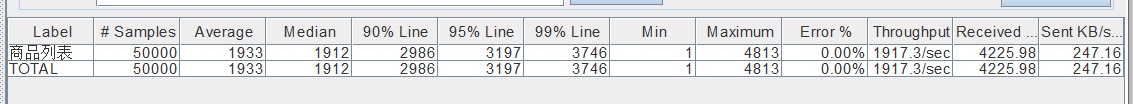
**更新用户密码：更新数据库与缓存，一定保证数据一致性，修改token关联的对象以及id关联的对象，先更新数据库后删除缓存，不能直接删除token，删除之后就不能登录了，再将token以及对应的用户信息一起再写回缓存里面去,这里我们修改哪个字段就更新哪个字段，没必要更新整个USER,提升sql性能**

**问题：更新**是先把数据存到数据库中，成功后，再让缓存失效，可以反过来吗

答：不可，如果先让缓存失效，此时有一个读操作，未更新的数据库信息又加载进了缓存，再更改数据库信息，导致数据不一致

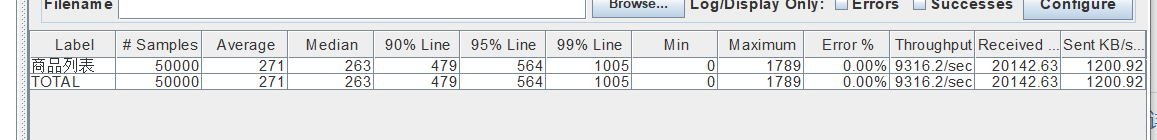
**优化前压测goods/to\_list**

**第三次结果**



**Mysql负载很大**

**缓存优化后：**



**稳定8000左右，负载都在redis，mysql负载很少**

**性能与当前计算机状态有关，总体是提升了很多**

**B．前后端分离（实际是html+ajax，静态的东西都用html写好做缓存，动态的才和服务端交互）**

**页面静态化+前后端分离**

1. 常用技术AngularJS、Vue.js

2. 优点：利用浏览器的缓存

本文只是简单的实现页面静态化：

将页面直接缓存到用户的浏览器上面，好处：用户访问数据的时候，不用去请求服务器，直接在本地缓存中取得需要的页面缓存。

未作页面静态化：请求某一个页面，访问缓存，查看缓存中是否有，缓存中有直接返回，缓存中没有的话，将数据渲染到html页面再存到缓存，再将整个html页面返回给客户端显示。

做了页面静态化：第一次是去请求后台要渲染好的html页面，之后的请求都是直接访问用户本地浏览器的缓存的html页面 ，静态资源，然后前端通过Ajax来访问后端，只去获取页面需要显示的数据返回即可。

没做的：每次都要向后台请求渲染的html页面

**a.商品详情页静态化**

在先前的操作中我们将商品详情页面进行了redis缓存，因为这个接口只是展示相应产品详情和秒杀倒计时等信息，只要显示几个关键信息即可，所以其他的都可以实现静态化。这种技术，实际就是前端分为两部分，一部分是不改变的html块，还有一块就是数据，它只要后端传数据到前端即可，用到ajax技术来请求数据。

为了在浏览器端进行缓存，以及控制缓存时间，在application.properties里面添加配置静态资源配置项

b.秒杀静态化

Get和Post的区别

Get是幂等的，代表从服务端获取数据，无论调用多少次都不会对服务端数据造成影响

Post非幂等，会令服务端数据变化

如果你的请求会对服务端数据影响，绝对不能用GET

C．解决库存秒杀成负

@Update(**"update miaosha\_goods set stock\_count = stock\_count - 1 where goods\_id = #{goodsId} and stock\_count > 0"**)  
**public int** reduceStock(MiaoshaGoods g);

减库存时候加**stock\_count > 0，防止库存减成负数，有这个条件数据库会对线程上锁，不会出现两个线程同时进行**

这里同时要对miaosha\_order表的uid和goodsid建立唯一索引确保一个用户只能秒杀一个，不能两次请求秒杀了两个

注意：视频上是错的这里：这两步都完成了进行压测，发现库存是为0了，但是订单数远大于库存数，就是库存5却生成了11个秒杀订单，此时库存数量还是0

原因：看这里

@Transactional  
 **public** OrderInfo miaosha(MiaoshaUser user, GoodsVo goods) {  
*// //减库存 下订单  
// goodsService.reduceStock(goods);  
// //写订单，order\_info maiosha\_order  
// return orderService.createOrder(user, goods);*

这是秒杀的逻辑，先减库存再下订单，这里有个问题，用户执行减库存的操作失败了，但是依然会去执行下订单的操作，这样就形成了生成的订单数大于卖出的商品数。

这样修改一下就可以了

*//减库存 下订单 写入秒杀订单***if**(**goodsService**.reduceStock(goods)== 0 ){  
 **return null**;*//如果这条减库存的SQL执行失败返回0，那么就直接返回，不要再执行下面的下订单了。*}  
*//order\_info maiosha\_order***return orderService**.createOrder(user, goods);

**public int** reduceStock(GoodsVo goods) {  
*// MiaoshaGoods g = new MiaoshaGoods();  
// g.setGoodsId(goods.getId());  
// goodsDao.reduceStock(g);* MiaoshaGoods g = **new** MiaoshaGoods();  
 g.setGoodsId(goods.getId());  
 *//flag标记这条SQL是否执行成功。0不成功，1成功。* **int** flag=**goodsDao**.reduceStock(g);  
 **return** flag;  
 }

修改后当**goodsDao**.reduceStock(g);减库存失败后，**if**(**goodsService**.reduceStock(goods)== 0 ){  
 **return null**;会直接返回，不做下订单项目

D．静态资源优化

1.js/css压缩

2.多个js/css组合在一起，以一个连接请求，减少连接数

3.CDN就近访问

**七、接口优化**





**拓展：数据库分库分表中间件，mycat**

1. **概述**

**高并发秒杀接口优化**

秒杀业务场景，并发量很大，瓶颈在数据库，怎么解决，加缓存。用户发起请求时，从浏览器开始,在浏览器上做页面静态化直接将页面缓存到用户的浏览器端，然后请求到达网站之前可以部署CDN节点，让请求先访问CDN，到达网站时候使用页面缓存。页面缓存再进一步的话，粒度再细一点的话就是对象缓存。缓存层依次请求完之后，才是数据库。通过一层一层的访问缓存逐步的削减到达数据库的请求数量，这样才能保证网站在大并发之下抗住压力。

但是仅仅依靠缓存还不够，所以还需要进行接口优化。  
**接口优化核心思路：减少数据库的访问。（数据库抗并发的能力有限）**

* **使用Redis预减库存减少对数据库的访问**
* **使用内存标记减少Redis的访问**
* **使用RabbitMQ队列缓冲，异步下单，增强用户体验**

具体实现步骤：

1. **系统初始化，把商品库存数量加载到Redis上面来**
2. **收到请求，Redis预减库存（先减少Redis里面的库存数量，库存不足，直接返回），如果库存已经到达临界值的时候，即=0，就不需要继续往下走，直接返回失败**
3. **库存有，请求入队，立即返回排队中**
4. **请求出队，生成订单，减少库存**
5. **客户端轮询，是否秒杀成功**

**B．安装RabbitMQ**

## RabbitMQ在Windows上面的安装与集成

安装rabbitmq首先要首先安装erlang。

**1.在Win环境下安装Erlang**

**步骤一：下载erlang**

下载地址：

64位：<http://erlang.org/download/otp_win64_20.3.exe>

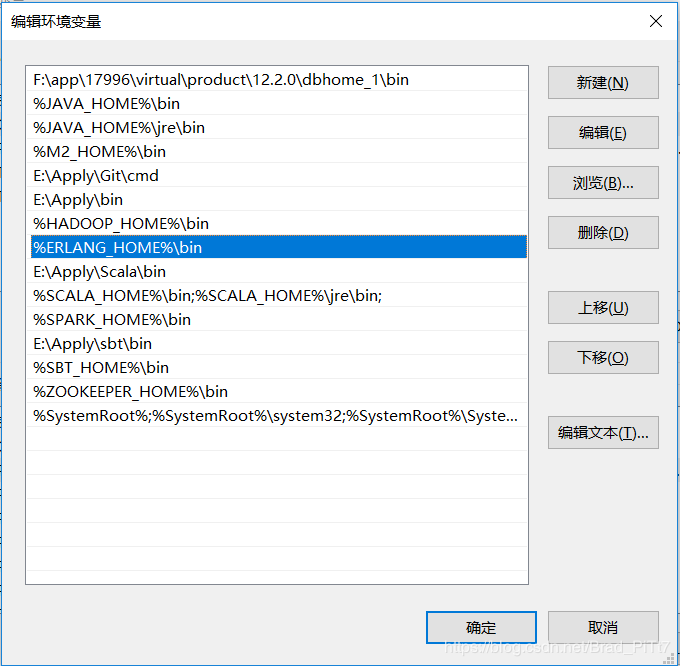
32位：<http://erlang.org/download/otp_win32_20.3.exe>

自己选择适合自己系统的，进行下载安装。

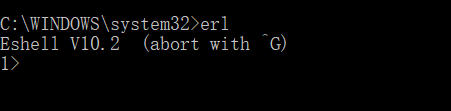
**步骤二：安装erlang**

直接点击exe安装，安装路径自己配的要记住在哪里，最好自己指定一个安装目录，等等会用到。

**步骤三：最重要的一步了，配置环境变量**

配置ERLANG\_HOME环境变量，其值指向erlang的安装目录(就是步骤二的路径)。另外将 ;%ERLANG\_HOME%\bin 加入到Path中。  
  
新建添加即可：  
  
**步骤四：检查安装是否成功**

打开cmd，输入erl

  
**2.在Win环境下安装RabbitMQ**  
下载资源：  
RabbitMQ，下载地址 <http://www.rabbitmq.com/install-windows.html>

对应版本（必须是与mq版本适应）的erlang，下载地址 <http://www.erlang.org/downloads/20.2>  
首先安装erlang，然后安装rabbitmq。

**安装完RabbitMQ以后，服务会自动运行**，这时环境变量里的ERLANG\_HOME会自动生成，在”环境变量”中检查是否存在，如果不存在，请在”环境变量”中手动添加，配置Erlang环境变量ERLANG\_HOME＝E:\Apply\Erlang213\erl10.2。如果没有，点击"新建"。

详细RabbitMQ安装请参考：<https://blog.csdn.net/h363659487/article/details/80913354>

登录<http://127.0.0.1:15672/>

默认用户和密码均为guest，但是这个只能用来连接本机，远程连接服务器需要自己在config配置文件写入[{rabbit,[{loopback\_users,[]}]}].

**2.集成RabbitMQ**

* 添加依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

* 1
* 2
* 3
* 4
* 在application.properties配置文件里面添加RabbitMQ配置信息

#RabbitMQ配置

spring.rabbitmq.host=127.0.0.1

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

spring.rabbitmq.virtual-host=/

#消费者数量

spring.rabbitmq.listener.simple.concurrency=10

#消费者最大数量

spring.rabbitmq.listener.simple.max-concurrency=10

#消费，每次从队列中取多少个,取多了，可能处理不过来

spring.rabbitmq.listener.simple.prefetch=1

spring.rabbitmq.listener.auto-startup=true

#消费失败的数据重新压入队列

spring.rabbitmq.listener.simple.default-requeue-rejected=true

#发送,队列满的时候，发送不进去，启动重置

spring.rabbitmq.template.retry.enabled=true

#一秒钟之后重试

spring.rabbitmq.template.retry.initial-interval=1000

#

spring.rabbitmq.template.retry.max-attempts=3

#最大间隔 10s

spring.rabbitmq.template.retry.max-interval=10000

spring.rabbitmq.template.retry.multiplier=1.0

**创建消息接收者和消息发送者**

**Config创建一个**

**建交换机，再绑定相应队列**

**建立发送者send**

**建立接受者reciever**

*发送者，将消息往外面发送的时候，并不是直接投递到队列里面去，而是先发送到交换机上面，然后由交换机发送数据到queue上面去，  
\* 做了依次路由。*

**为啥不直接放到queue?**

**有交换机，可以支持topic模式等，可以更加灵活的发送**

**Mq支持四种交换机模式**

**1.** Direct模式，直接发送接收

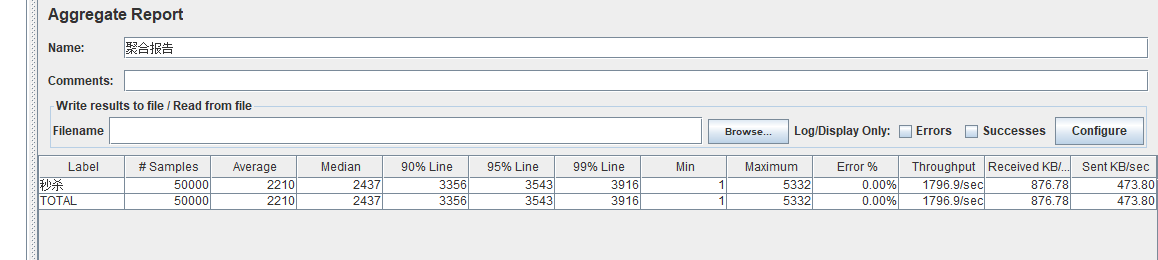
**2.** Topic模式,指定将什么消息发给哪个queue

3. Fanout模式，广播模式，发给多个queue，没有指定的key,交换机将收到的消息发给所有绑定的queue

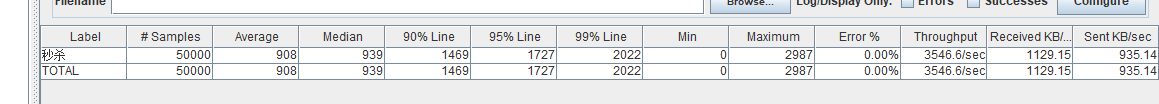
4. Header模式,后面跟条件where，可以一个key满足，可以所有key满足，key和value都满足等条件可供选择，用字节数组接收

**C、秒杀接口优化**

**优化前do\_miaosha三次压测结果，并且存在数据库减成负的情况，平均大概2000左右**



**加了redis缓存，并且解决额数据库库存负数，此时订单是同步下单**



具体实现步骤：

1. **系统初始化，把商品库存数量加载到Redis上面来**

*\* 继承InitializingBean，系统初始化时会回调此函数，在这里我们将库存预存入redis*

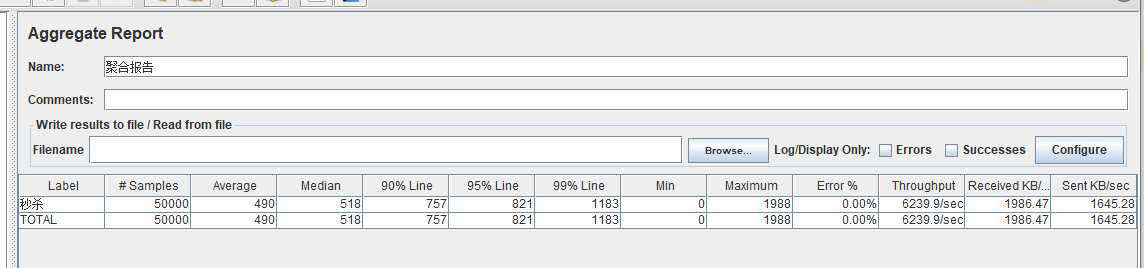
1. **收到请求，Redis预减库存（先减少Redis里面的库存数量，库存不足，直接返回），如果库存已经到达临界值的时候，即=0，就不需要继续往下走，直接返回失败，比如redis只有十个库存，这样10后面的请求对数据库都没有压力.如果库存还有进入3,当第11个用户请求，库存减成负数，判断直接返回，也就是说后面的操作没必要访问redis数据库**
2. **请求入队，立即返回排队中(异步下单，将请求放到消息队列中，直接返回排队中，这样秒杀后客户端可以立马得到相应，增强用户体验**

*//接收到mq的消息，这里我们访问数据库，因为mq队列里的数据不多，大部分在提升排队中，只有很少的用户进来*

*//先入队,这里入队信息不多，入队后直接返回排队中，reciever收到sender信息，再和数据库交互，形成异步处理*

1. **请求出队，生成订单，减少库存(**
2. **客户端轮询，是否秒杀成功**

**做了如上处理，起飞**

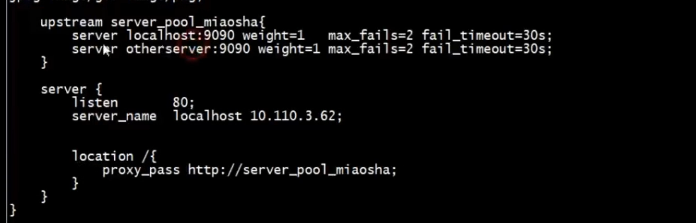


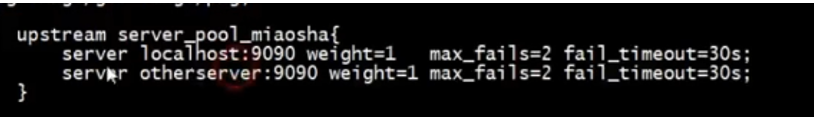
**D.nignx横向扩展**

**1.安装nignx**

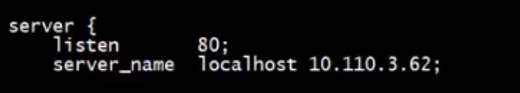
**2.打开nignx配置文件，config，vi ….config**

**3.在配置文件中的server**

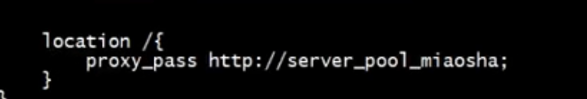




**这是一个服务器集群server….miaosha,Server后面是各个服务器，weight就是他们的负载1/(1+1),max\_fails最大失败两次，失败超过两次后将不再分配给此服务器**

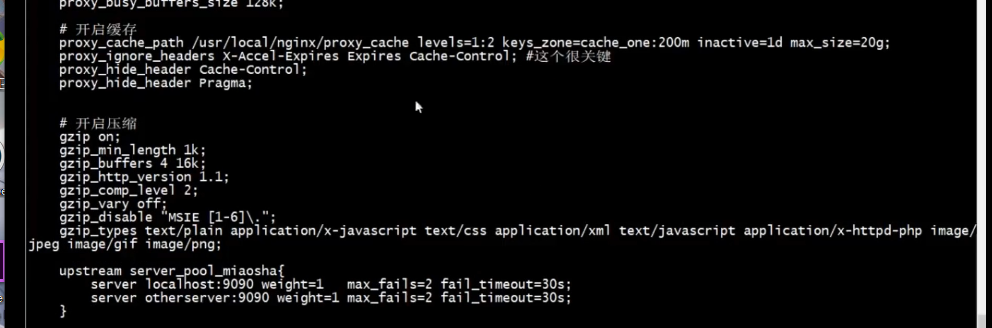


**监听10…服务器80端口的所有请求**



**将监听到的请求都反向代理给ser…miaosha服务器集群**

**NIGNX也可以配缓存**

****

**如果nignx都顶不住，还有LVS**

Tomcat

Nignx

Tomcat

**浏览器-----------LVS …………………..**

Nignx

**通常一个NIGNX挂几台服务器已经很快了**

**八、安全优化**

