0.1 秒杀常见问题

```
有问题或者宝贵意见联系我的QQ,非常希望你的加入!
```

> 秒杀注意事项以及整体简略设计
>
> ### 0.0.1 1.如何解决卖超问题
>
> --在sql加上判断防止数据边为负数
> --数据库加唯一索引防止用户重复购买
> --redis预减库存减少数据库访问　内存标记减少redis访问　请求先入队列缓冲，异步下单，增强用户体验

0.0.2 注册功能 -- 如果有前端的牛人加入修改几个页面那是再好不过了哈哈哈

0.0.3 全局异常处理拦截

```
1.定义全局的异常拦截器
2.定义了全局异常类型
3.只返回和业务有关的
4.详情请看GlobleException
```

0.0.4 页面级缓存thymeleafViewResolver

```
1.详细请看basecontroller 缓存渲染页面
```

0.0.5 对象级缓存redis

```
redis永久缓存对象减少压力
redis预减库存减少数据库访
内存标记方法减少redis访问
```

0.0.6 订单处理队列rabbitmq

```
请求先入队缓冲，异步下单，增强用户体验
请求出队，生成订单，减少库存
客户端定时轮询检查是否秒杀成功
```

0.0.7 解决分布式session

```
--生成随机的uuid作为cookie返回并redis内存写入
--拦截器每次拦截方法，来重新获根据cookie获取对象
--下一个页面拿到key重新获取对象
--HandlerMethodArgumentResolver 方法 supportsParameter 如果为true 执行 resolveArgument 方法获取miaoshauser对象
--如果有缓存的话 这个功能实现起来就和简单，在一个用户访问接口的时候我们把访问次数写到缓存中，在加上一个有效期。
    通过拦截器. 做一个注解 @AccessLimit 然后封装这个注解，可以有效的设置每次访问多少次，有效时间是否需要登录!
```

0.0.8 秒杀安全 -- 安全性设计

```
秒杀接口隐藏
数字公式验证码
接口防刷限流(通用 注解，拦截器方式)
```

0.0.9 通用缓存key的封装采用什么设计模式

```
模板模式的优点
-具体细节步骤实现定义在子类中，子类定义详细处理算法是不会改变算法整体结构
-代码复用的基本技术，在数据库设计中尤为重要
-存在一种反向的控制结构，通过一个父类调用其子类的操作，通过子类对父类进行扩展增加新的行为，符合“开闭原则”
-缺点：   每个不同的实现都需要定义一个子类，会导致类的个数增加，系统更加庞大
```

0.0.10 redis的库存如何与数据库的库存保持一致

```
redis的数量不是库存,他的作用仅仅只是为了阻挡多余的请求透穿到DB，起到一个保护的作用
因为秒杀的商品有限，比如10个，让1万个请求区访问DB是没有意义的，因为最多也就只能10个
请求下单成功，所有这个是一个伪命题，我们是不需要保持一致的
```

0.0.11 redis 预减成功，DB扣减库存失败怎么办

```
-其实我们可以不用太在意，对用户而言，秒杀不中是正常现象，秒杀中才是意外，单个用户秒杀中
-1.本来就是小概率事件，出现这种情况对于用户而言没有任何影响
-2.对于商户而言，本来就是为了活动拉流量人气的，卖不完还可以省一部分费用，但是活动还参与了，也就没有了任何影
响
-3.对网站而言，最重要的是体验，只要网站不崩溃，对用户而言没有任何影响
```

0.0.12 为什么redis数量会减少为负数

```
    //预见库存
    long stock = redisService.decr(GoodsKey.getMiaoshaGoodsStock,""+goodsId) ;
    if(stock <0){
    localOverMap.put(goodsId, true);
    return Result.error(CodeMsg.MIAO_SHA_OVER);
    }
    假如redis的数量为1,这个时候同时过来100个请求，大家一起执行decr数量就会减少成-99这个是正常的
    进行优化后改变了sql写法和内存写法则不会出现上述问题
```

0.0.13 为什么要单独维护一个秒杀结束标志

```
-1.前提所有的秒杀相关的接口都要加上活动是否结束的标志，如果结束就直接返回，包括轮寻的接口防止一直轮寻
-2.管理后台也可以手动的更改这个标志，防止出现活动开始以后就没办法结束这种意外的事件
```

0.0.14 rabbitmq如何做到消息不重复不丢失即使服务器重启

```
-1.exchange持久化
-2.queue持久化
-3.发送消息设置MessageDeliveryMode.persisent这个也是默认的行为
-4.手动确认
```

0.0.15 为什么threadlocal存储user对象，原理

```
1.并发编程中重要的问题就是数据共享，当你在一个线程中改变任意属性时，所有的线程都会因此受到影响，同时会看到第
一个线程修改后的值<br>
有时我们希望如此，比如：多个线程增大或减小同一个计数器变量<br>
但是，有时我们希望确保每个线程，只能工作在它自己 的线程实例的拷贝上，同时不会影响其他线程的数据<br>

举例：  举个例子，想象你在开发一个电子商务应用，你需要为每一个控制器处理的顾客请求，生成一个唯一的事务ID，同
时将其传到管理器或DAO的业务方法中，
以便记录日志。一种方案是将事务ID作为一个参数，传到所有的业务方法中。但这并不是一个好的方案，它会使代码变得冗
余。
你可以使用ThreadLocal类型的变量解决这个问题。首先在控制器或者任意一个预处理器拦截器中生成一个事务ID
然后在ThreadLocal中 设置事务ID，最后，不论这个控制器调用什么方法，都能从threadlocal中获取事务ID
而且这个应用的控制器可以同时处理多个请求，
同时在框架 层面，因为每一个请求都是在一个单独的线程中处理的，所以事务ID对于每一个线程都是唯一的，而且可以从
所有线程的执行路径获取
运行结果可以看出每个线程都在维护自己的变量：
```

```
 Starting Thread: 0 : Fri Sep 21 23:05:34 CST 2018<br>
 Starting Thread: 2 : Fri Sep 21 23:05:34 CST 2018<br>
 Starting Thread: 1 : Fri Jan 02 05:36:17 CST 1970<br>
 Thread Finished: 1 : Fri Jan 02 05:36:17 CST 1970<br>
 Thread Finished: 0 : Fri Sep 21 23:05:34 CST 2018<br>
 Thread Finished: 2 : Fri Sep 21 23:05:34 CST 2018<br>

 局部线程通常使用在这样的情况下，当你有一些对象并不满足线程安全，但是你想避免在使用synchronized关键字<br>
 块时产生的同步访问，那么，让每个线程拥有它自己的对象实例<br>
 注意：局部变量是同步或局部线程的一个好的替代，它总是能够保证线程安全。唯一可能限制你这样做的是你的应用设计约
束<br>
 所以设计threadlocal存储user不会对对象产生影响，每次进来一个请求都会产生自身的线程变量来存储
```

0.0.16 maven 隔离

```
maven隔离就是在开发中，把各个环境的隔离开来，一般分为
 本地（local）
 开发(dev)
 测试(test)
 线上(prod)
 在环境部署中为了防止人工修改的弊端！ spring.profiles.active=@activatedProperties@
```

0.0.17 redis 分布式锁实现方法

```
我用了四种方法 ， 分别指出了不同版本的缺陷以及演进的过程 orderclosetask
V1---->>版本没有操作，在分布式系统中会造成同一时间，资源浪费而且很容易出现并发问题
V2--->>版本加了分布式redis锁，在访问核心方法前，加入redis锁可以阻塞其他线程访问,可以
很好的处理并发问题,但是缺陷就是如果机器突然宕机，或者线路波动等，就会造成死锁，一直
不释放等问题
V3版本-->>很好的解决了这个问题v2的问题，就是加入时间对比如果当前时间已经大与释放锁的时间
说明已经可以释放这个锁重新在获取锁，setget方法可以把之前的锁去掉在重新获取,旧值在于之前的
值比较，如果无变化说明这个期间没有人获取或者操作这个redis锁，则可以重新获取
V4---->>采用成熟的框架redisson,封装好的方法则可以直接处理，但是waittime记住要这只为0
```

0.0.18 服务降级--服务熔断(过载保护)）

```
自动降级： 超时.失败次数,故障,限流
人工降级：秒杀，双11

9.所有秒杀相关的接口比如：秒杀，获取秒杀地址，获取秒杀结果，获取秒杀验证码都需要加上
秒杀是否开始结束的判断
```

0.0.19 RPC事务补偿

```
当集中式进行服务化RPC演进成分布式的时候，事务则成为了进行分布式的一个痛点，本项目的做法为：
1.进行流程初始化，当分别调用不用服务化接口的时候，成功则进行流程，失败则返回并进行状态更新
将订单状态变为回滚
2.使用定时任务不断的进行处理rollback的订单进行回滚
```

0.0.20 秒杀类似场景sql的写法注意事项

```
1.在秒杀一类的场景里面，因为数据量亿万级所有即使有的有缓存有的时候也是扛不住的，不可避免的透穿到DB
 所有在写一些sql的时候就要注意：
 1.一定要避免全表扫描，如果扫一张大表的数据就会造成慢查询，导致数据的连接池直接塞满,导致事故
 首先考虑在where和order by 设计的列上建立索引
 例如： 1. where 子句中对字段进行 null 值判断 .
        2. 应尽量避免在 where 子句中使用=或<>操作符
        3. 应尽量避免在 where 子句中使用 or 来连接条件
        4. in 和 not in 也要慎用，否则会导致全表扫描( 如果索引 会优先走索引 不会导致全表扫描
         字段上建了索引后，使用in不会全表扫描，而用not in 会全表扫描 低版本的mysql是两种情况都会全表扫
描。
         5.5版本后以修。而且在优化大表连接查询的时候，有一个方法就是将join操作拆分为in查询）
        5. select id from t where name like '%abc%' 或者
        6.select id from t where name like '%abc' 或者
        7. 若要提高效率，可以考虑全文检索。
```

```
        8.而select id from t where name like 'abc%' 才用到索引 慢查询一般在测试环境不容易复现
        9.应尽量避免在 where 子句中对字段进行表达式操作 where num/2  num=100*2
2.合理的使用索引  索引并不是越多越好，使用不当会造成性能开销
3.尽量避免大事务操作，提高系统并发能力
4.尽量避免象客户端返回大量数据，如果返回则要考虑是否需求合理，实在不得已则需要在设计一波了！！！！！
```

0.0.21 网站访问统计实现

利用lua脚本进行对redis操作,登陆时，每次登陆成功则记录访问（具体你想在什么时段进行统计自己说了算）

0.0.22 项目进行dubbo+ZK改造

```
├── miaosha-admin  登录模块
│   ├── pom.xml
│   └── miaosha-admin-api
│   └── miaosha-admin-service
│   └── miaosha-admin-web
│   └── miaosha-common
│
│
├── miaosha-order   订单秒杀模块
│   ├── pom.xml
│   └── miaosha-order-api
│   └── miaosha-order-service
│   └── miaosha-order-web
│   └── miaosha-order-common
│
│
├── miaosha-message   消息模块
│   ├── pom.xml
│   └── miaosha-message-api
│   └── miaosha-message-service
│   └── miaosha-message-web
│   └── miaosha-message-common
│
│
```