1. 单例模式

第一种：饿汉模式

饿汉模式在类被初始化时就已经在内存中创建了对象，以空间换时间，故不存在线程安全问题

第二种：双重锁懒汉模式(Double Check Lock)+volatile

优点就是，只有在对象需要被使用时才创建，第一次判断 INSTANCE == null为了避免非必要加锁，当第一次加载时才对实例进行加锁再实例化。这样既可以节约内存空间，又可以保证线程安全

**第三种：静态内部类模式：**

静态内部类的优点是：外部类加载时并不需要立即加载内部类，内部类不被加载则不去初始化INSTANCE，故而不占内存。即当SingleTon第一次被加载时，并不需要去加载SingleTonHoler，只有当getInstance()方法第一次被调用时，才会去初始化INSTANCE,第一次调用getInstance()方法会导致虚拟机加载SingleTonHoler类，这种方法不仅能确保线程安全，也能保证单例的唯一性，同时也延迟了单例的实例化。

饿汉式单例：

优点

1.实例的初始化由JVM装载类的时候进行，保证了线程的安全性  
2.实现简单方便  
3.实例的访问效率高  
缺点

1.不能实现懒加载，如果不调用getInstance(),那么这个类就白白的占据内存，资源的利用率不高

懒汉式单例：

优点

1.只有使用这个类的时候才初始化实例，优化了资源利用率  
缺点

1.为了实现线程安全，使用了同步方法获取，增加了访问的开销

双重检测机制+Volatile（第一重检测提高访问效率；第二重检测提高了线程的安全）

优点

1.实现懒加载  
2.通过缩小同步区域和第一次检查提高访问效率  
缺点

1.为了实现线程安全，使用了同步方法获取，增加了访问的开销

静态内部类实现单例

优点

1.即实现了线程安全，又实现了懒加载

1、一致性hash算法

场景：比如你有 *N* 个 *cache* 服务器（后面简称 *cache* ），那么如何将一个对象 *object* 映射到 *N* 个 *cache* 上呢，你很可能会采用类似下面的通用方法计算 *object* 的 *hash* 值，然后均匀的映射到到 *N* 个 *cache* ；

*hash(object)%N*

一切都运行正常，再考虑如下的两种情况；

*1*一个 *cache* 服务器 *m down* 掉了（在实际应用中必须要考虑这种情况），这样所有映射到 *cache m* 的对象都会失效，怎么办，需要把 *cache m* 从 *cache* 中移除，这时候 *cache* 是 *N-1* 台，映射公式变成了 *hash(object)%(N-1)* ；

*2*由于访问加重，需要添加 *cache* ，这时候 *cache* 是 *N+1* 台，映射公式变成了 *hash(object)%(N+1)* ；

*1* 和 *2* 意味着什么？这意味着突然之间几乎所有的 *cache* 都失效了。

*consistent hashing* 是一种 *hash* 算法，简单的说，在移除 */* 添加一个 *cache* 时，它能够尽可能小的改变已存在 *key* 映射关系，尽可能的满足单调性的要求。说白了，可以解决因为动态增加节点或者删除节点，而造成很多数据失效

1. 页面静态化（1是实现前后端分离，2是将页面静态化，可以减少）

第一：现实中的动态网页往往只是其中一部分会发生变化，例如电商网站的菜单、页面头部、页面尾部这些其实都不会经常发生变化，如果我们只是因为网页一小部分经常变化让用户每次请求都要重复访问这些重复的资源，这其实是非常消耗计算资源了，我们来做个计算吧，假如一个动态页面这些不变的内容有10k，该网页一天有1000万次的访问量，那么每天将消耗掉1亿kb的网络资源，这个其实很不划算的，而且这些**重复消耗的宽带资源并没有为网站的用户体验带来好处，相反还拖慢了网页加载的效率**。那么我们就得考虑拆分网页了，把网页做一个动静分离，让静态的部分当做不变的静态资源进行处理，动态的内容还是动态处理，然后在合适的地方将动静内容合并在一起。

第二：java的web开发里就会考虑使用模板语言Theamleaf替代jsp页面技术，例如模板语言velocity，这些模板语言都包含一个布局的功能，例如velocity就有这样的功能，我们看看velocity的布局模板实例，如下所示：

第三：使用Ajax实现“动静合并”；不过一般而言，我们使用ajax做动静分离都是都是从服务端请求一个html片段，到了浏览器后，使用dom技术将这个片段整合到页面里.

总结：1、使用Thymeleaf模板生成静态页面模板 2、使用Ajax来实现“动态页面和静态页面的合并”，即使用Ajax向服务端发出请求，并获取响应数据，然后采用dom技术，将响应数据整合到静态页面模板中去

1. 防止秒杀按钮重复提交（disabled） **防止用户提交多次**

4、前端js+html实现简单验证码（分散用户的请求，缓解同一时间很多用户同时点击秒杀按钮，产生高并发）

<div class="new-web-row">

<span class="web-form-span " >验证码</span>

<input type="text" class=" web-form-input " required="required" id="Yzm">

<input type = "button" id="code" value="点我验证" onclick="createCode()"/>

</div>

生成验证码并进行验证：

var code ; //在全局定义验证码

//产生验证码

function createCode(){

code = "";

var codeLength = 4;//验证码的长度

**var checkCode = document.getElementById("code");**

var random = new Array(0,1,2,3,4,5,6,7,8,9,'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R', 'S','T','U','V','W','X','Y','Z');//随机数

for(var i = 0; i < codeLength; i++) {//循环操作

var index = Math.floor(Math.random()\*36);//取得随机数的索引（0~35）

code += random[index];//根据索引取得随机数加到code上

}

checkCode.value = code;//把code值赋给验证码

}

//校验验证码

document.getElementById("Yzm").addEventListener("change",validate);

function validate(){

var inputCode = document.getElementById("Yzm").value.toUpperCase(); //取得输入的验证码并转化为大写

if(inputCode.length <= 0) { //若输入的验证码长度为0

alert("请输入验证码！"); //则弹出请输入验证码

$("#Yzm").focus();

YZM = false;

}

else if(inputCode != code ) { //若输入的验证码与产生的验证码不一致时

alert("验证码输入错误！@\_@"); //则弹出验证码输入错误

createCode();//刷新验证码

$("#Yzm").val("");//清空文本框

$("#Yzm").focus();//重新聚焦验证码框

YZM = false;

}

else { //输入正确时

$("#Yzm").blur();//绑定验证码输入正确时要做的事

YZM = true;

}

};

总结：前端优化（**页面静态化、秒杀按钮禁止多次提交、秒杀之前输入验证码**）

**后端优化（controller层接口限流优化+三级缓冲优化）**

**1、controller层接口优化（redis+自定义注解+拦截器实现）**

@AccessLimit(seconds=5, maxCount=5) **在controller层接口方法上面加上注解，限流**

@Retention(***RUNTIME***)

@Target(***METHOD***)

**public** **@interface** AccessLimit {

**int** seconds();

**int** maxCount();

}

当执行接口方法之前，会去执行拦截器中的prehander方法！

@Service

**public** **class** AccessInterceptor **extends** **HandlerInterceptorAdapter**{

@Override

**public** **boolean** **preHandle**(HttpServletRequest request, HttpServletResponse response, Object handler)**throws** Exception {

HandlerMethod hm = (HandlerMethod)handler;

AccessLimit accessLimit = hm.getMethodAnnotation(AccessLimit.**class**);

**if**(accessLimit == **null**) {

**return** **true**;

}

**int** seconds = accessLimit.seconds();

**int** maxCount = accessLimit.maxCount();

String key = request.getRequestURI();

key += "\_" + user.getId();

AccessKey ak = AccessKey.*withExpire*(seconds);

Integer count = redisService.get(ak, key, Integer.**class**);

**if**(count == **null**) {

redisService.set(key, 1);

redisService.expire(key, ak); //一定要设置其缓存时间

}**else** **if**(count < maxCount) {

redisService.incr(ak, key);

}**else** {

render(response, CodeMsg.*ACCESS\_LIMIT\_REACHED*);

**return** **false**;

}

}

**return** **true**;

}

秒杀controller层，即MiaoshaController，有那几个方法：

1：实现接口InitializingBean并重写该方法afterPropertiesSet（初始化这个bean后，执行该方法）

/\* \*

\* 系统初始化

\* \*/

**public** **void** afterPropertiesSet() **throws** Exception {

List<GoodsVo> goodsList = goodsService.listGoodsVo();

**if**(goodsList == **null**) {

**return**;

}

**for**(GoodsVo goods : goodsList) {

redisService.set(GoodsKey.*getMiaoshaGoodsStock*, ""+goods.getId(), goods.getStockCount());

localOverMap.put(goods.getId(), **false**);

}

}

第一步：从数据库中加载所有秒杀商品的id和对应库存，存放到redis中去，key为id\_stock , value为商品stock 第二步：初始化本地标记，concurrentHashMap，key为id，value为true，表示有库存

**2、点击秒杀按钮，请求秒杀，调用miaosha方法**

@RequestMapping(value="/{path}/do\_miaosha", method=RequestMethod.POST)

@ResponseBody

**public** Result<Integer> miaosha();

**if**(user == **null**) {

**return** Result.*error*(CodeMsg.*SESSION\_ERROR*);

}

//内存标记，减少redis访问

**boolean** over = localOverMap.get(goodsId);

**if**(over) {

**return** Result.*error*(CodeMsg.*MIAO\_SHA\_OVER*);

}

//预减库存

**long** stock = redisService.decr(GoodsKey.*getMiaoshaGoodsStock*, ""+goodsId);//10

**if**(stock < 0) {

localOverMap.put(goodsId, **true**);

**return** Result.*error*(CodeMsg.*MIAO\_SHA\_OVER*);

}

//判断是否已经秒杀到了

MiaoshaOrder order = orderService.getMiaoshaOrderByUserIdGoodsId(user.getId(), goodsId);

**if**(order != **null**) {

redisService.incr(GoodsKey.*getMiaoshaGoodsStock*, ""+goodsId); **return** Result.*error*(CodeMsg.*REPEATE\_MIAOSHA*);

}

//入队，将用户信息和秒杀商品信息封装到message中去

MiaoshaMessage mm = **new** MiaoshaMessage();

mm.setUser(user);

mm.setGoodsId(goodsId);

sender.sendMiaoshaMessage(mm);

//给用户返回一个“正在排队中...”的消息

**return** Result.*success*(0);//排队中

}

1. 获取秒杀结果的接口

@RequestMapping(value="/GetResult", method=RequestMethod.GET)

@ResponseBody

**public** Result<Long> getMiaoShaResult(Model model,MiaoshaUser user,

@RequestParam("goodsId")**long** goodsId) {

model.addAttribute("user", user);

**if**(user == **null**) {

**return** Result.*error*(CodeMsg.*SESSION\_ERROR*);

}

**long** result =miaoshaService.getMiaoshaResult(user.getId(), goodsId);

**return** Result.*success*(result);

}

RabbitMQ中：

发送消息：

**public** **void** sendMiaoshaMessage(MiaoshaMessage mm) {

String msg = RedisService.*beanToString*(mm);

*log*.info("send message:"+msg);

amqpTemplate.convertAndSend(MQConfig.***MIAOSHA\_QUEUE***, msg);

}

接受消息：（**指定队列，轮询方式监听队列，并取出消息来消费**）

@RabbitListener(queues=MQConfig.***MIAOSHA\_QUEUE***)

**public** **void** receive(String message) {

*log*.info("receive message:"+message);

MiaoshaMessage mm = RedisService.*stringToBean*(message, MiaoshaMessage.**class**);

MiaoshaUser user = mm.getUser();

**long** goodsId = mm.getGoodsId();

GoodsVo goods = goodsService.getGoodsVoByGoodsId(goodsId);

**int** stock = goods.getStockCount();

**if**(stock <= 0) {

**return**;

}

//判断是否已经秒杀到了

MiaoshaOrder order = orderService.getMiaoshaOrderByUserIdGoodsId(user.getId(), goodsId);

**if**(order != **null**) {

**return**;

}

//减库存 下订单 写入秒杀订单

miaoshaService.miaosha(user, goods);

}

miaoshaService

@Transactional**（rollbackFor = Exception.class）**

**public** OrderInfo miaosha(MiaoshaUser user, GoodsVo goods) {

Long lastTime = 5000l;//5秒后过期

While(**redisService.setnx(“lock”,System.currentTimeMillis())** == 0){

Long lockTime = redisService.get(“lock”);

If(system.currentTimeMillis()>=lastTime+lockTime){

redisService.del(“lock”);

}

Thread.sleep(1000);

}

//减库存 下订单 写入秒杀订单

**boolean** success = goodsService.reduceStock(goods);

**if**(success) {

//order\_info maiosha\_order

orderService.createOrder(user, goods);

}**else** {

setGoodsOver(goods.getId());

}

redisService.del(“lock”);

Return “chenggong”;

}

注意：将“扣减库存”和“生成订单”放到一个事务中

**public** **long** getMiaoshaResult(Long userId, **long** goodsId) {

MiaoshaOrder order = orderService.getMiaoshaOrderByUserIdGoodsId(userId, goodsId);

**if**(order != **null**) {//秒杀成功

**return** order.getOrderId();

}**else** {

**boolean** isOver = getGoodsOver(goodsId);

**if**(isOver) {

**return** -1;

}**else** {

**return** 0;

}

}

}

注意：可以根据“userID”和“商品ID”，**判断订单表中是否有该订单生成**，有则说明已经秒杀成功，返回订单信息！！

**整理思路：**

用户点击按钮秒杀-->调用controller层的do\_miaoshao接口，在这个秒杀接口里面，会先查看“本地标记”，对应商品在redis中是否还有库存，从而减少redis访问，然后在访问redis，进行预减库存，并判断该用户是否已经秒杀过，否则将用户信息和秒杀商品封装到message中，并向rabbitMQ的消息队列发送消息-->此时，消息队列取出消息，并解析消息，然后调用miaoshaService的miaosha方法“扣减库存和生成订单”-->miaosha方法中定义了事务，进行**扣减库存和生成订单**

前台，客户端通过“轮询”的方法，**不断请求“getMiaoShaResult的接口”**，进而调用miaoshaService.getMiaoshaResult(user.getId(), goodsId);若“订单表中生成了一个新的UserID和goodId”的order记录，则说明秒杀成功，返回秒杀成功的订单！！

**function doMiaosha(path){**

$.ajax({

url:"/miaosha/"+path+"/do\_miaosha",

type:"POST",

data:{

goodsId:$("#goodsId").val()

},

success:function(data){

if(data.code == 0){

//window.location.href="/order\_detail.htm?orderId="+data.data.id;

**getMiaoshaResult($("#goodsId").val());**

}else{

layer.msg(data.msg);

}

},

error:function(){

layer.msg("客户端请求有误");

}

});

**function getMiaoshaResult(goodsId){**

g\_showLoading();

$.ajax({

url:"/miaosha/result",

type:"GET",

data:{

goodsId:$("#goodsId").val(),

},

success:function(data){

if(data.code == 0){

var result = data.data;

if(result < 0){

layer.msg("对不起，秒杀失败");

}else if(result == 0){//继续轮询

**setTimeout(function(){**

getMiaoshaResult(goodsId);

}, 200);

}else{

layer.confirm("恭喜你，秒杀成功！查看订单？", {btn:["确定","取消"]},

function(){

window.location.href="/order\_detail.htm?orderId="+result;

},

function(){

layer.closeAll();

});

}

}else{

layer.msg(data.msg);

}

},

error:function(){

layer.msg("客户端请求有误");

}

});

}

创新性：

1. 使用了concurrentHashMap来记录redis中的秒杀商品库存情况，key对应的是商品ID，value为true表示还有商品，false表示没有商品了，从而减少了“redis的读取”
2. 接口限流那一块，一开始直接在每一个接口中，使用redis.get(uri+userID)去获取次数，**代码重复性比较高,并且不够灵活**，所以采用了通用的限流方法“redis+自定义注解+拦截器”实现

**两个项目介绍：**

1. 研究生管理系统的设计与开发

1）来由：这个项目是“数据库设计”的课程大作业的项目，是由我和另外一个同学共同完成的，其中我负责后台代码的编写工作

2）涉及的知识点：这个项目主要分为三大模块，即学生模块、教师模块、管理员模块，主要涉及到的知识点有：对象的增删查改、编码过滤器、登录拦截器、使用“单例模式+session监听器+原子类”实现了在线人数统计、使用“ThreadLocal+session监听器”来保存用户的状态信息、使用“单例模式+静态工厂模式+策略模式”代替if else结构实现“多条件查询”、使用redis防止用户恶意登录网站以及使用“redis+自定义注解+拦截器”实现controller层的接口限流

总结：1、编码过滤器（实现filter接口，重写doFilter方法，设置request.setCharacterEncoding("utf-8")，使用了责任链模式，然后在web.xml配置文件中配置filter）2、登录拦截器（继承handerInterceptAdapted这个父类，重写里面的preHander这个方法，对未登录的用户进行拦截，在applicationcontext.xml文件中，配置拦截器，对哪些类哪些方法进行拦截）3、在线人数统计，使用“单例模式+原子类+session监听器，**实现HTTPSessionAttributeListener接口**”实现，当用户登录时候，将用户信息保存在session中，同时触发“session监听器中的attributeAdded方法，调用单例模式的计数类，调用AtomicInteger的increamentAndGet方法，当登出用户的时候，删除session，触发attributeRemoved方法”，进而调用AtomicInteger原子类的decreamentAndGet方法 4、“多条件查询”实现，记录key，如果key=id，则按照id进行查询，如果key=name，则按照name进行查询等等，比如按照用户ID、用户Phore、用户的姓名、模糊查询啊等条件，策略接口（查询抽象方法）、具体策略（按照ID查询、按照Phore查询。。。），构建一个单例模式的静态工厂，提供一个静态变量hashMap，将具体策略的对象实例以“K-V”形式存放到hashMap中去，并提供通过key来获取实例的get方法 5、使用redis保存用户ip\_Username的登录错误次数，每次登录的时候，先从redis中，取出“ip+username”key对应的value次数，若大于10，则拒绝，并设置key的缓存时间为10分钟，否则查询数据库，若错误，则incr增加一次，否则成功，则删除对应缓存，获取IP，可以使用request.getRemoteAddr()

**解决了哪些问题：1、**在开发“在线人数统计”功能时候，一开始，单例类里面使用了“一个volatile修饰的全局变量来记录当前人数”，然后，后来查了文档，了解到volatile不能保证原子性操作，于是改成使用原子类 2、使用“redis限制恶意用户错误登录次数”，一开始，只是使用了“用户登录使用的IP”作为key，然后错误登录的次数，作为值，后来思考了一下，这样做，不合理，因为，可能存在同个IP下，不同用户登录的情况，于是使用“IP+UserName”作为key

“多条件查询”实现：

**策略接口**

public interface SearchInterface {

List<user> searchUsers(String value) throws DataAccessException;

}

**具体策略（实现类）**

public class FindOwnerById implements SearchInterface {

@Autowired

private UserService userService;

public Collection<Owner> searchUsers(String value) throws DataAccessException{

return userService.findOwnerById(value);

}

}

public class FindOwnerByName implements SearchInterface {

@Autowired

private UserService userService;

public Collection<Owner> searchUsers(String value) throws DataAccessException{

return userService.findOwnerByName(value);

}

}

public class FindOwnerByPhore implements SearchInterface {

@Autowired

private UserService userService;

public Collection<Owner> searchUsers(String value) throws DataAccessException{

return userService.findOwnerByPhore(value);

}

}

public class FindOwnerByLike implements SearchInterface {

@Autowired

private UserService userService;

public Collection<Owner> searchUsers(String value) throws DataAccessException{

return userService.findOwnerByLike(value);

}

}

**策略工厂：（使用单例模式获取策略工厂实例）**

public class StrategyFactory {

private static StrategyFactory factory = new StrategyFactory();

private StrategyFactory(){

}

private static Map strategyMap = new HashMap<>();

static{

strategyMap.put(“id”, new FindOwnerByIdStrategy());

strategyMap.put(“name”, new FindOwnerByNameStrategy());

strategyMap.put(“phore”, new FindOwnerByPhoreStrategy());

strategyMap.put(“like”, new FindOwnerByLikeStrategy());

}

public Strategy getWitchStrategy(Integer type){

return strategyMap.get(type);

}

public static StrategyFactory getInstance(){

return factory;

}

}

测试：

public Collection<Owner> findOwner (String key ， String value) throws DataAccessException{

return StrategyFactory.getInstance().getWitchStrategy(key).findOwner(value);

}

**总结：因为if else结构会导致比较凌乱，并且，不容易扩展，不够灵活，所以采用“策略模式+静态工厂模式+单例模式”代替if else的结构**

2 高并发的秒杀系统（对于do\_miaoshao接口，在jmeter压力测试，当同时模拟5万个请求时，QPS为QPS: 2114） QPS（Query Per Second）每秒处理多少请求

1）来由：该项目是自己想要去了解相关的“高并发解决方案”以及Redis缓存和rabbitMQ消息队列而自学的一个项目

2）涉及到的知识点有：该项目主要是基于springBoot+redis+rabbitMQ实现的一个电商秒杀系统，**该项目着重去解决高并发带来的两个问题**：第一个是高并发环境下，如何尽可能缓解数据库层面的压力，提高系统的高并发 ， 以及高并发环境下，如何防止秒杀商品少卖和超卖 ，其中本项目涉及到的知识点有：**前端优化**（使用了页面缓存、页面静态化、CDN缓存静态资源、使用一些限流技术（比如禁止秒杀按钮多次提交、点击秒杀之前，使用验证码分流） ； **后端优化**：在controller层，使用“redis+自定义注解+拦截器”实现通用的接口限流 ； 使用“内存标记+redis预减库存+RabbitMQ异步下单”三级缓冲，尽可能减小数据库层面的压力）