# 京淘订单业务

## 用户信息动态获取

### 业务需求

当用户登录成功后,动态获取用户信息.之后将userId值在业务层中灵活使用.

@RequestMapping("/show")

**public** String findCartList(Model model) {

**Long userId = 7L; //暂时写死**

List<Cart> cartList =

cartService.findCartListByUserId(userId);

model.addAttribute("cartList", cartList);

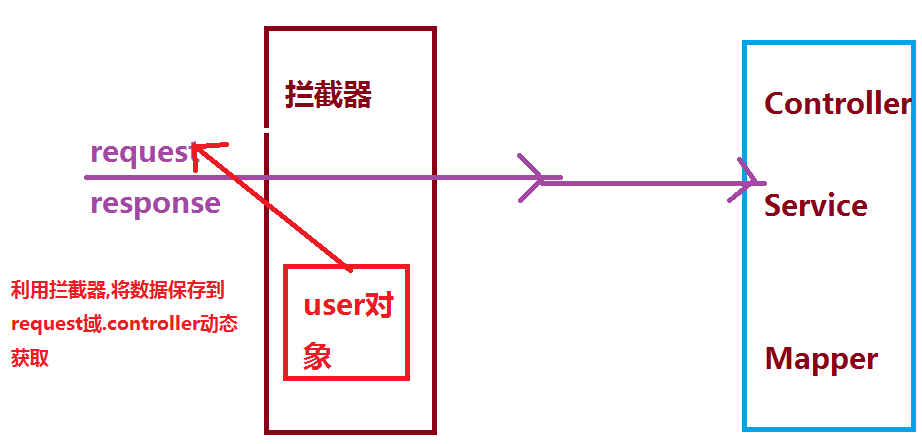
**return** "cart";//返回页面逻辑名称

}

难点问题:

1. 如何保证每次都能获取request对象 答案:拦截器可以实现
2. 拦截器和业务处理是2个类 拦截器中的数据怎么传递给下一个执行过程. 在拦截器中将数据保存到request域中.之后业务动态获取!!

### 利用request对象获取数据



#### 编辑拦截器

**if**(!StringUtils.*isEmpty*(userJSON)) {

//redis中有用户数据.拦截放行

//将userJSON转化为user对象.

User user = ObjectMapperUtil.*toObject*(userJSON, User.**class**);

//将user数据保存到request域中.

request.setAttribute("JT\_USER",user);

**return** **true**;

}

#### Controller动态获取数据

/\*\*

\* 1.实现商品列表信息展现

\* 2.页面取值: ${cartList}

\*/

@RequestMapping("/show")

**public** String findCartList(Model model,HttpServletRequest request) {

**User user = (User) request.getAttribute("JT\_USER");**

**Long userId = user.getId();**

List<Cart> cartList =

cartService.findCartListByUserId(userId);

model.addAttribute("cartList", cartList);

**return** "cart";//返回页面逻辑名称

}

### 利用Request对象说明

说明:一般企业开发时,都会利用request对象将数据进行保存,之后再获取.

问题:

当用户在业务层中获取用户信息时,必须传递Request对象或者是从request对象中获取的数据.

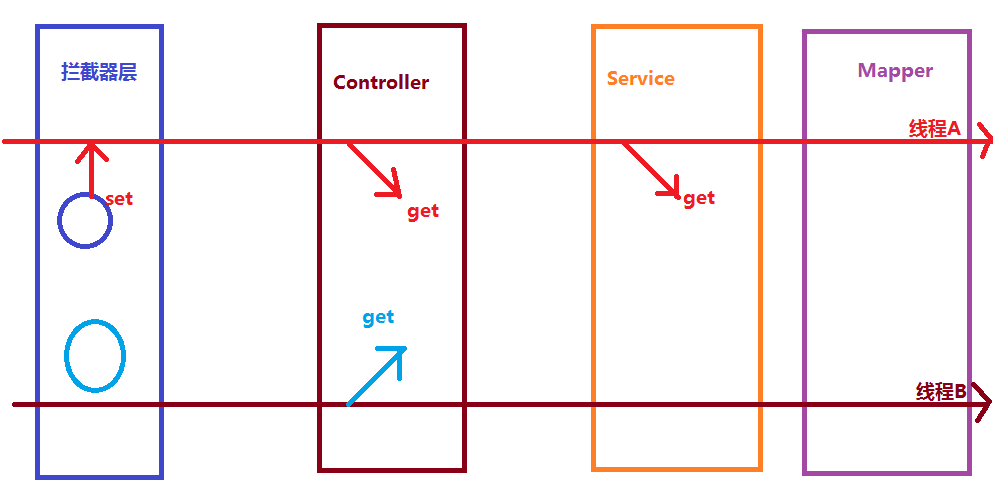
**public** List<Cart> findCartListByUserId(Long userId,HttpServletRequest request)

## ThreadLocal介绍

### 说明

ThreadLocal:**本地线程变量**

作用:可以在同一个线程内实现数据共享



### ThreadLocal工具类

**public** **class** UserThreadLocal {

/\*\*

\* 如何存取多个数据?? Map集合

\* ThreadLocal<Map<k,v>>

\*/

**private** **static** ThreadLocal<User> *thread* = **new** ThreadLocal<>();

//新增数据

**public** **static** **void** set(User user) {

*thread*.set(user);

}

//获取数据

**public** **static** User get() {

**return** *thread*.get();

}

//使用threadlocal切记关闭 防止内存泄漏

**public** **static** **void** remove() {

*thread*.remove();

}

}

### 编辑拦截器

@Component //将拦截器交给spring容器管理

**public** **class** UserInterceptor **implements** HandlerInterceptor{

@Autowired

**private** JedisCluster jedisCluster;

/\*\*

\* 在spring4版本中要求必须重写3个方法,不管是否需要

\* 在spring5版本中在接口中添加default属性,则省略不写.

\*/

/\*\*

\* 返回值结果:

\* true: 拦截放行.

\* false: 请求拦截. 重定向到登录页面

\*

\* 业务逻辑:

\* 1.获取Cookie数据

\* 2.从cookie中获取token(TICKET)

\* 3.判断redis缓存服务器中是否有数据.

\*/

@Override

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

**throws** Exception {

System.***out***.println("拦截器开始工作!!!!!");

String token = **null**;

//1.获取Cookie信息

Cookie[] cookies = request.getCookies();

**for** (Cookie cookie : cookies) {

**if**("JT\_TICKET".equals(cookie.getName())) {

token = cookie.getValue();

**break**;

}

}

//2.判断token是否有效

**if**(!StringUtils.*isEmpty*(token)) {

//4.判断redis中是否有数据

String userJSON = jedisCluster.get(token);

**if**(!StringUtils.*isEmpty*(userJSON)) {

//redis中有用户数据.拦截放行

//将userJSON转化为user对象.

//将user数据保存到request域中.

//request.setAttribute("JT\_USER",user);

//用户每次请求都将数据保存到session中,切记用完关闭

//request.getSession().setAttribute("JT\_USER",user);

User user = ObjectMapperUtil.*toObject*(userJSON, User.**class**);

UserThreadLocal.*set*(user);

**return** **true**;

}

}

//3.重定向到用户登录页面

response.sendRedirect("/user/login.html");

**return** **false**;//表示拦截

}

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

//request.getSession().removeAttribute("JT\_USER");

//调用移除操作

UserThreadLocal.*remove*();

}

}

### 动态获取userID

/\*\*

\* 1.实现商品列表信息展现

\* 2.页面取值: ${cartList}

\*/

@RequestMapping("/show")

**public** String findCartList(Model model) {

//User user = (User) request.getAttribute("JT\_USER");

//Long userId = user.getId();

Long userId = **UserThreadLocal.*get*().getId();**

List<Cart> cartList =

cartService.findCartListByUserId(userId);

model.addAttribute("cartList", cartList);

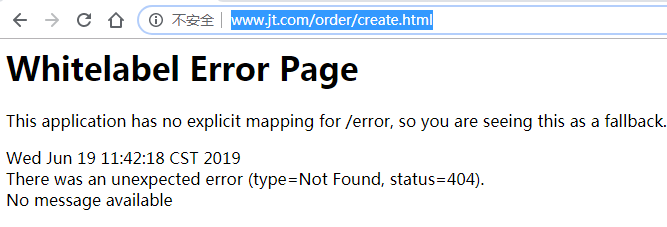
**return** "cart";//返回页面逻辑名称

}

## 跳转订单确认页

### 页面分析

1.当用户点击按钮时跳转到订单的确认页面



2.页面取值分析

说明:当用户跳转订单确认页时,需要显示用户购买的商品信息.

<c:forEach items=*"${carts}"* var=*"cart"* varStatus=*"status"*>

<c:set var=*"totalPrice"* value=*"${ totalPrice + (cart.itemPrice \* cart.num)}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].itemId"* value=*"${cart.itemId}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].num"* value=*"${cart.num }"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].price"* value=*"${cart.itemPrice}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].totalFee"* value=*"${cart.itemPrice \* cart.num}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].title"* value=*"${cart.itemTitle}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].picPath"* value=*"${cart.itemImage}"*/>

</c:forEach>

### 编辑消费者Controller

说明:编辑前台OrderController.通过cartService获取购物车信息.之后跳转页面展现.

@Controller

@RequestMapping("/order")

**public** **class** OrderController {

@Reference(timeout = 3000,check = **false**)

**private** DubboOrderService orderService;

@Reference(timeout = 3000,check = **false**)

**private** DubboCartService cartService;

/\*\*

\* 跳转订单确认页面

\* url:http://www.jt.com/order/create.html

\* 页面名称: order-cart.jsp

\* 页面取值: items="${carts}"

\*/

@RequestMapping("/create")

**public** String orderCreate(Model model) {

Long userId = UserThreadLocal.*get*().getId();

List<Cart> carts = cartService.findCartListByUserId(userId);

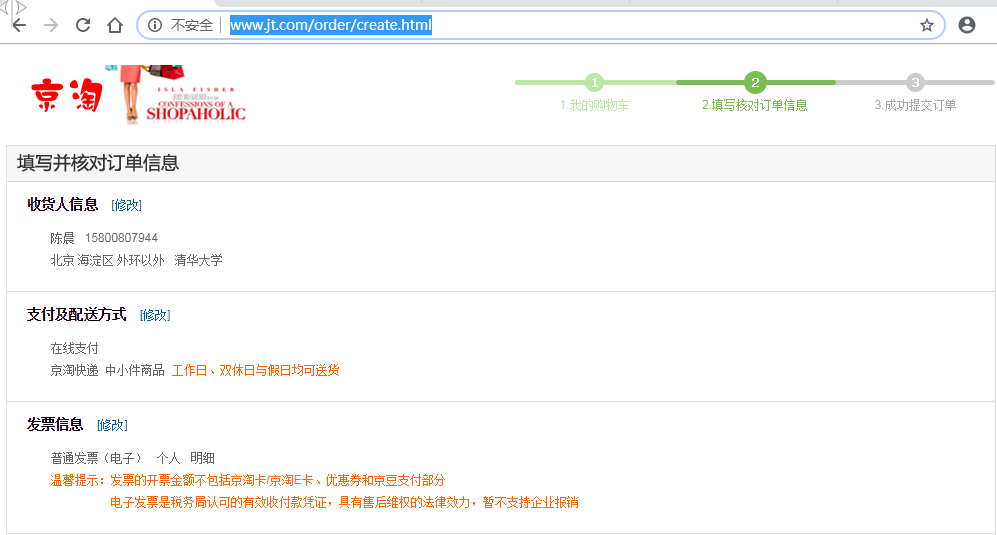
model.addAttribute("carts", carts);

**return** "order-cart";

}

}

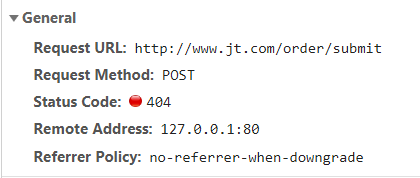
### 页面效果



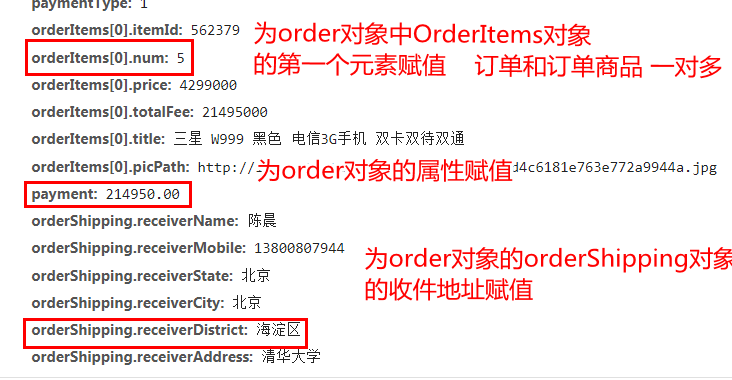
## 订单入库

### 页面分析

#### url请求地址



#### 参数定义



#### 页面代码分析

<form id=*"orderForm"* class=*"hide"*>

<input type=*"hidden"* name=*"paymentType"* value=*"1"*/>

<c:forEach items=*"${carts}"* var=*"cart"* varStatus=*"status"*>

<c:set var=*"totalPrice"* value=*"${ totalPrice + (cart.itemPrice \* cart.num)}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].itemId"* value=*"${cart.itemId}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].num"* value=*"${cart.num }"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].price"* value=*"${cart.itemPrice}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].totalFee"* value=*"${cart.itemPrice \* cart.num}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].title"* value=*"${cart.itemTitle}"*/>

<input type=*"hidden"* name=*"orderItems[${status.index}].picPath"* value=*"${cart.itemImage}"*/>

</c:forEach>

<input type=*"hidden"* name=*"payment"* value=*"<fmt:formatNumber groupingUsed="*false" maxFractionDigits=*"2"* minFractionDigits=*"2"* value=*"${totalPrice/100 }"*/>"/>

<input type=*"hidden"* name=*"orderShipping.receiverName"* value=*"陈晨"*/>

<input type=*"hidden"* name=*"orderShipping.receiverMobile"* value=*"13800807944"*/>

<input type=*"hidden"* name=*"orderShipping.receiverState"* value=*"北京"*/>

<input type=*"hidden"* name=*"orderShipping.receiverCity"* value=*"北京"*/>

<input type=*"hidden"* name=*"orderShipping.receiverDistrict"* value=*"海淀区"*/>

<input type=*"hidden"* name=*"orderShipping.receiverAddress"* value=*"清华大学"*/>

</form>

#### 返回值数据分析

说明:表单提交之后进行业务处理.返回值使用**SysResult**对象.并且需要传递入库的orderId值.

jQuery.ajax( {

type : "POST",

dataType : "json",

url : "/order/submit",

data : $("#orderForm").serialize(),

cache : **false**,

success : **function**(result) {

**if**(result.status == 200){

location.href = "/order/success.html?id="+result.data;

}**else**{

$("#submit\_message").html("订单提交失败，请稍后重试...").show();

}

},

error : **function**(error) {

$("#submit\_message").html("亲爱的用户请不要频繁点击, 请稍后重试...").show();

}

});

### 编辑消费者Controller

/\*\*

\* 实现订单入库

\*

\*/

@RequestMapping("/submit")

@ResponseBody

**public** SysResult insertOrder(Order order) {

**try** {

Long userId = UserThreadLocal.*get*().getId();

order.setUserId(userId);

String orderId =

orderService.insertOrder(order);

**if**(!StringUtils.*isEmpty*(orderId)) {

**return** SysResult.*ok*(orderId);

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*fail*();

}

### 编辑服务提供者

@Service

**public** **class** DubboOrderServiceImpl **implements** DubboOrderService {

@Autowired

**private** OrderMapper orderMapper;

@Autowired

**private** OrderItemMapper orderItemMapper;

@Autowired

**private** OrderShippingMapper orderShippingMapper;

/\*\*

\* 一个业务逻辑需要操作3张表数据

\* 1.添加事务控制

\* 2.表数据分析 order orderItems orderShipping

\* 3.准备orderId '订单号：登录用户id+当前时间戳',

\* 4.操作3个mapper分别入库

\*/

@Transactional

@Override

**public** String insertOrder(Order order) {

//1.获取orderId

String orderId = ""+order.getUserId()+System.*currentTimeMillis*();

Date date = **new** Date();

//状态：1、未付款2、已付款3、未发货4、已发货5、交易成功6、交易关闭

//2.入库订单

order.setOrderId(orderId)

.setStatus(1)

.setCreated(date)

.setUpdated(date);

orderMapper.insert(order);

System.***out***.println("订单入库成功!!!!");

//3.入库订单物流

OrderShipping shipping = order.getOrderShipping();

shipping.setOrderId(orderId)

.setCreated(date)

.setUpdated(date);

orderShippingMapper.insert(shipping);

System.***out***.println("订单物流入库成功!!!!!");

//4.入库订单商品 1自己遍历集合分别入库 2.自己编辑sql语句批量入库

//insert into order\_items(xxxx,xxx,xxx) values(xxxx,xxx....),(xxx,xxx,xxx),(xxxxx)

List<OrderItem> orderLists = order.getOrderItems();

**for** (OrderItem orderItem : orderLists) {

orderItem.setOrderId(orderId)

.setCreated(date)

.setUpdated(date);

orderItemMapper.insert(orderItem);

}

System.***out***.println("订单商品入库成功!!!");

**return** orderId;

}

}

## 订单查询

### 业务说明

说明:当用户提交订单后.根据返回的orderId查询订单信息.

思路:3张表根据orderId都需要查询.之后封装数据返回.

跳转页面: success.jsp

页面数据要求: ${order.payment}元 服务器向页面传递order对象.

### 编辑Controller

/\*\*

\* 根据订单信息,查询数据

\*/

@RequestMapping("/success")

**public** String findOrderById(String id,Model model) {

Order order = orderService.findOrderById(id);

model.addAttribute("order", order);

**return** "success";

}

### 编辑提供者Service

@Override

**public** Order findOrderById(String id) {

Order order = orderMapper.selectById(id);

OrderShipping shipping = orderShippingMapper.selectById(id);

QueryWrapper<OrderItem> queryWrapper = **new** QueryWrapper<OrderItem>();

queryWrapper.eq("order\_id",id);

List<OrderItem> list = orderItemMapper.selectList(queryWrapper);

order.setOrderItems(list).setOrderShipping(shipping);

**return** order;

}

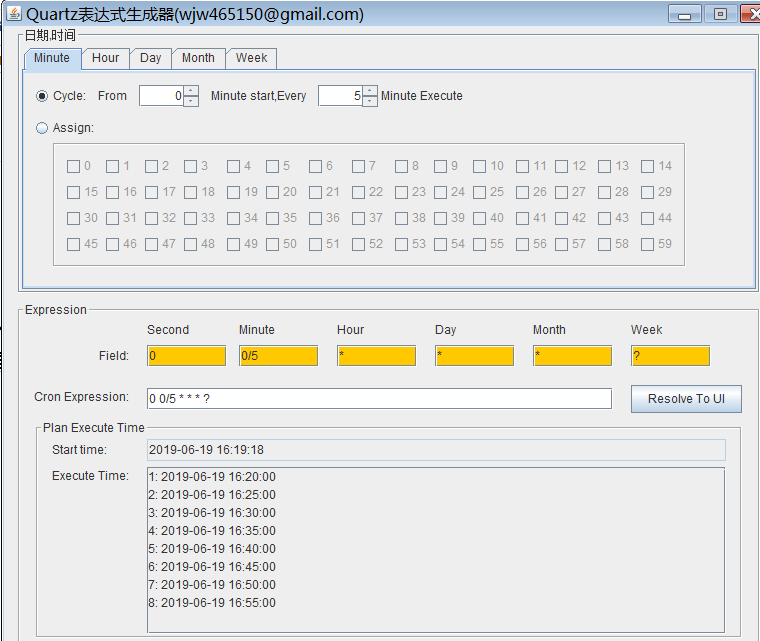
### 页面效果



## Quartz定时任务

### 导入时间表达式工具

说明:将课前资料中的Quartz项目导入项目,之后执行Main.java展现表达式工具



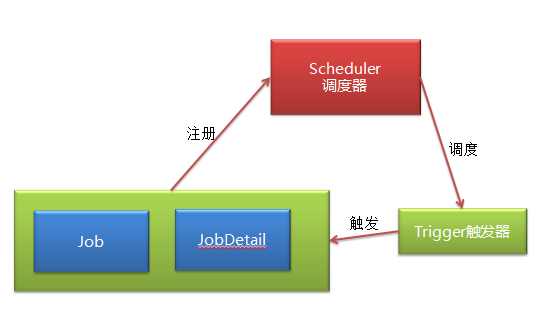
该工具是时间表达式生成器.用于定时任务.

### 业务需求

说明:如果用户提交了订单.在30分钟之内没有完成支付,则将订单的状态由1改为6.

### 定时任务Quartz

Quartz是OpenSymphony开源组织在Job scheduling领域又一个开源项目，它可以与J2EE与J2SE应用程序相结合也可以单独使用。Quartz可以用来创建简单或为运行十个，百个，甚至是好几万个Jobs这样复杂的程序。Jobs可以做成标准的Java组件或 EJBs。Quartz的最新版本为Quartz 2.3.0。



1. 调度器 负责任务管理.内部有时钟监控
2. 触发器 当调度器需要执行任务时,通过触发器启动新的线程去执行.
3. JOB/JobDetail 定义任务.

### 定时任务配置

#### 引入jar包

<!--添加Quartz的支持 -->

<dependency>

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

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

</dependency>

### 编辑配置类

@Configuration

**public** **class** OrderQuartzConfig {

//定义任务详情

@Bean

**public** JobDetail orderjobDetail() {

//指定job的名称和持久化保存任务

**return** JobBuilder

.*newJob*(OrderQuartz.**class**)

.withIdentity("orderQuartz")

.storeDurably()

.build();

}

//定义触发器

@Bean

**public** Trigger orderTrigger() {

/\*SimpleScheduleBuilder builder = SimpleScheduleBuilder.simpleSchedule()

.withIntervalInMinutes(1) //定义时间周期

.repeatForever();\*/

CronScheduleBuilder scheduleBuilder

= CronScheduleBuilder.*cronSchedule*("0 0/1 \* \* \* ?");

**return** TriggerBuilder

.*newTrigger*()

.forJob(orderjobDetail())

.withIdentity("orderQuartz")

.withSchedule(scheduleBuilder).build();

}

}

### 编辑定时任务类

//准备订单定时任务

@Component

**public** **class** OrderQuartz **extends** QuartzJobBean{

@Autowired

**private** OrderMapper orderMapper;

/\*\*

\* 业务思想:

\* 用户30分钟内,没有支付则将状态改为6交易关闭

\* sql:

\* update tb\_order set status = 6,

\* updated=#{date}

\* where status=1 and created<now-30

\*

\*/

//当程序执行时 执行该方法

@Override

@Transactional

**protected** **void** executeInternal(JobExecutionContext context) **throws** JobExecutionException {

Calendar calendar = Calendar.*getInstance*(); //获取当前时间

calendar.add(Calendar.***MINUTE***, -30);

Date timeOutDate = calendar.getTime();

Order order = **new** Order();

order.setStatus(6).setUpdated(**new** Date());

UpdateWrapper<Order> updateWrapper = **new** UpdateWrapper<Order>();

updateWrapper.eq("status", 1)

.lt("created", timeOutDate);

orderMapper.update(order, updateWrapper);

System.***out***.println("定时任务执行成功!!!!!");

}

}