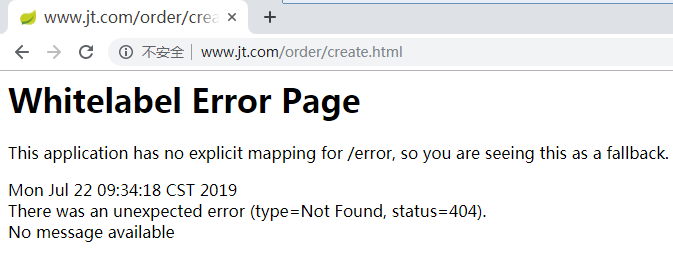
# 订单业务实现

## 跳转订单确认页面

### 页面分析



### 编辑Controller

@Controller

@RequestMapping("order")

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

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

**private** DubboCartService cartService;

@RequestMapping("/create")

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

//获取用户的购物车信息

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

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

model.addAttribute("carts", cartList);

**return** "order-cart"; //跳转页面

}

}

### 页面效果



## 关于SpringMVC传参方式

### 页面传参介绍

1.页面参数 当该属性提交时 userId=123

<input id="userId" name="userId" type="text" value="123"/>

### 简单参数接收

springMVC取值实质就是利用request的gagParameter方法获取数据.

**规则:页面的提交的name属性名必须和接收参数的名称一致**.

**public** User mvcTest(Integer userId,HttpServletRequest request) {

request.getParameter("userId");

}

### 利用对象的形式接收参数.

/\*\*

\*

\* <input name=userId value=123/>

\* 接收流程是什么???

\* 怎么取值 怎么赋值

\* 提示:

\* 1.无参构造

\* 2.get/set方法??

\*

\* 取值过程:

\* 1.利用反射机制实例化对象(必须有无参构造)

\* 2.利用对象的getXXXX()方法获取对象的属性名称

\* getUserId()~~~~~~将get去掉之后首字母小写 userId;

\* 3.利用request的getParamiter(userId).获取对象的属性的值.

\* 如果获取的数据为null.则属性的值为null.

\* 4.调用对象的setUserId(userId)完成赋值操作.

\* 5.最终形成一个已经实例化成功的user对象.

\*

\* **@param** user

\* **@return**

\*/

**public** User mvcTest(User user) {

**return** **new** User();

}

### SpringMVC为对象的引用赋值

//<input name="小河马.河马名称" type="text" value="沙皮" />

//<input name="小河马.河马品种" type="text" value="壮的很">

**class** User {

**int** userId;

String username;

河马 小河马;

}

**class** 河马 {

String 河马名称;

String 河马品种;

}

### SpringMVC为对象的引入集合赋值

<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}*"*/>

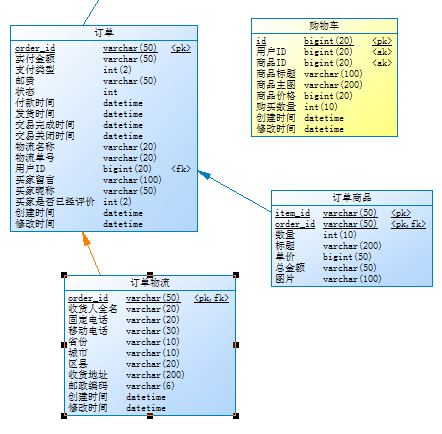
</c:forEach>

## 订单新增业务实现

### 订单表设计

订单与订单物流一对一

订单与订单商品一对多

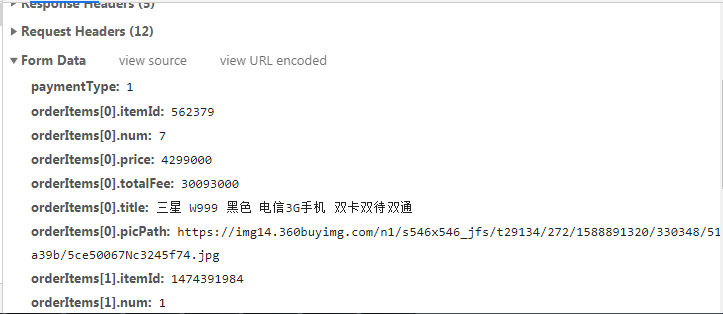


### 页面分析

1.url分析



2.页面提交参数



3.页面JS分析

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();

}

});

}

业务说明:当用户点击提交订单时.会将订单信息入库.同时返回orderId.用于查询成功的订单.

### 编辑Controller

/\*\*

\* 实现订单入库操作

\* url地址:http://www.jt.com/order/submit

\*/

@RequestMapping("/submit")

@ResponseBody

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

//1.业务要求返回 orderId号

String orderId =

orderService.saveOrder(order);

//2.校验orderId是否有值

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

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

}

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

}

### 编辑Service

@Service

**public** **class** OrderServiceImpl **implements** DubboOrderService {

@Autowired

**private** OrderMapper orderMapper;

@Autowired

**private** OrderShippingMapper orderShippingMapper;

@Autowired

**private** OrderItemMapper orderItemMapper;

/\*\*

\* 1.要求返回orderId号

\* 2.要求同时入库

\* 3.三张表分别入库

\*/

@Override

@Transactional //进行事务控制

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

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

Date now = **new** Date();

//1.入库订单信息

order.setOrderId(orderId);

order.setStatus(1); //表示未付款

order.setCreated(now);

order.setUpdated(now);

orderMapper.insert(order);

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

//2.订单物流入库

OrderShipping shipping = order.getOrderShipping();

shipping.setOrderId(orderId);

shipping.setCreated(now);

shipping.setUpdated(now);

orderShippingMapper.insert(shipping);

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

//3.订单商品入库

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

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

orderItem.setOrderId(orderId);

orderItem.setCreated(now);

orderItem.setUpdated(now);

orderItemMapper.insert(orderItem);

}

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

**return** orderId;

}

}

## 订单查询

### 业务需求

当用户点击生成订单时,如果订单入库成功则跳转到订单查询页面.展现订单信息.

### url分析

1.页面分析

<http://www.jt.com/order/success.html?id=15637768434759>

### 编辑Controller

/\*\*

\* ${order.orderId}

\* **@param** id

\* **@return**

\*/

@RequestMapping("success")

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

Order order = orderService.findOrderById(id);

model.addAttribute("order", order);

**return** "success";

}

### 编辑Service

/\*\*

\* order对象中应该包含三部分数据

\* order对象 orderShipping对象 orderItem对象

\* 思考:

\* 1.3张表关联查询 where,left join

\* 2. 单表查询

\*核心业务思路:

\* 应该尽可能的将多表关联查询,转化为单表查询.

\*/

@Override

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

Order order = orderMapper.selectById(id);

OrderShipping shipping = orderShippingMapper.selectById(id);

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

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

List<OrderItem> items =

orderItemMapper.selectList(queryWrapper);

order.setOrderItems(items)

.setOrderShipping(shipping);

**return** order;

}

### 页面效果展现



## 超时订单处理

### 业务需求

当订单30分钟没有完成支付,应该将订单的状态由1改为6.

sql:

update tb\_order set status=6,updated=#{date}

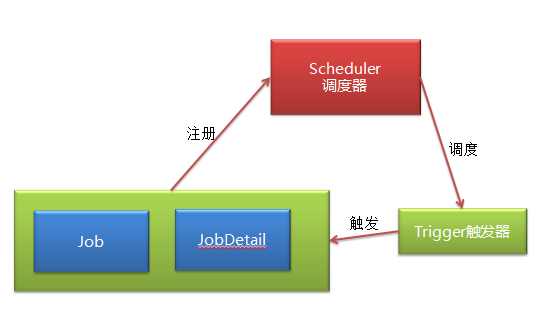
where status = 1 and 当前时间-创建时间 > 30分钟 ??

开启一个线程 要求这个线程每隔1分钟执行一次.

### Quartz定时任务

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。

### Quartz工作流程



1. 调度器:统一管理任务的管理中心
2. 触发器:规定任务多久执行一次.
3. JOB : 自己具体的业务需求

JobDetail: 负责job与quartz通信.

工作流程:

1. 首先调度器启动
2. 任务定义之后封装到jobDetail交给调度器管理.
3. 调度器负责任务的执行.当任务需要处理时通过触发器开启新的线程去执行任务.当job的任务执行完成之后.将线程关闭.

### 导入jar包

程序启动时一定会加载Quartz的配置信息.

<!--添加Quartz的支持 -->

<dependency>

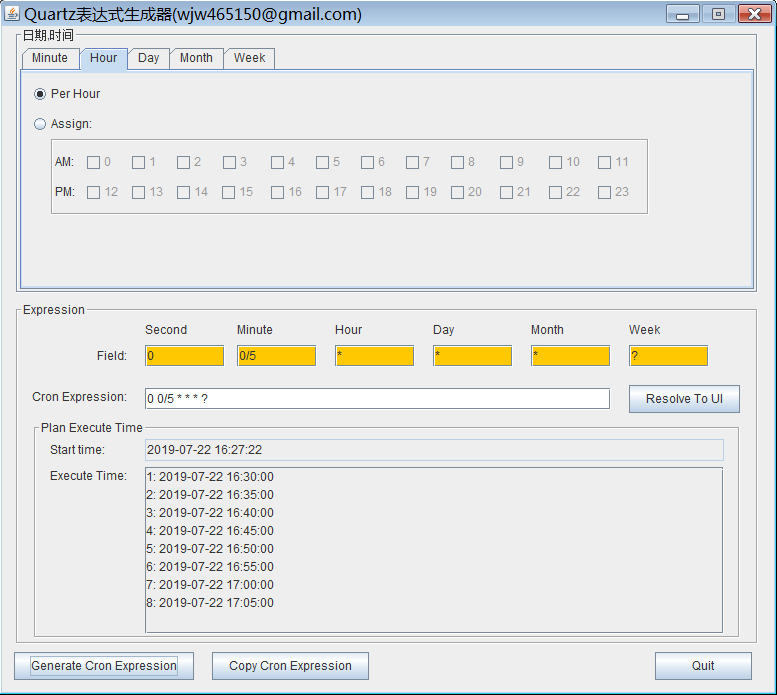
<groupId>org.springframework.boot</groupId>

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

</dependency>

### 导入表达式生成器

说明:将课前资料中表达式生成器导入项目.了解时间表达式的用法.



### 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**)

//创建一个job任务

.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()) //触发器绑定任务

.withIdentity("orderQuartz")

.withSchedule(scheduleBuilder).build();

}

}

### 编辑定时任务

//准备订单定时任务

@Component //id=orderQuartz

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

@Autowired

**private** OrderMapper orderMapper;

/\*\*

\* 标记超时订单.

\*修改数据: 1.status=6 2.updated=now

\* 条件: status=1 and created < 当前时间-30分钟

\*

\*/

@Override

@Transactional

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

//格林威治时间 毫秒 秒 分 时 天 周 月 年

//代表当前的时间

Calendar calendar = Calendar.*getInstance*();

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

Date timeOut = calendar.getTime();

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

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

UpdateWrapper<Order> updateWrapper =

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

updateWrapper.eq("status", 1)

.lt("created",timeOut);

orderMapper.update(order, updateWrapper);

System.***out***.println("定时任务完成 更新数据库!!!!!!");

}

}

### 效果展现

