# 购物功能实现

## 购物车数量修改

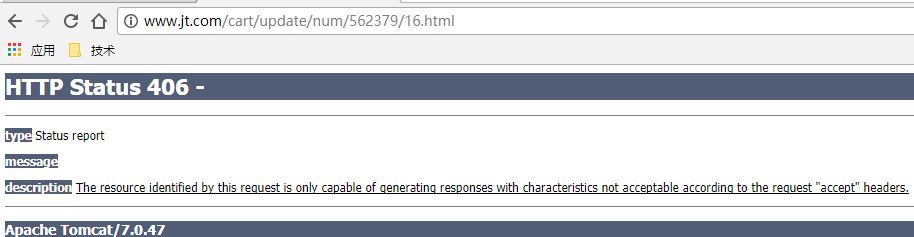
### 406报错

过程描述:

1. 请求格式

<http://www.jt.com/cart/update/num/562379/16.html>

页面报错:



406报错表示浏览器解析数据时,格式不匹配导致的.因为请求是以.html结尾的.那么浏览器在编译时,会动态的加载**静态页面**.要求服务器的返回值必须返回html页面.

如果服务器没有正确返回html页面信息则回报406错误.

总结:

如果页面要求跳转一般采用.html请求结尾.

如果请求是AJAX那么请求不能以.html结尾.所以有如下的配置

<!-- 防止springMVC框架返回json时和html冲突报 406 错误 -->

<servlet-mapping>

<servlet-name>springmvc-web</servlet-name>

<url-pattern>/service/\*</url-pattern>

</servlet-mapping>

### 页面分析

知识回顾:

前提:如果在js中对数据进行操作(加减等运算,或者对象的操作),必须将数据转化为**js对象**.否则不能操作.

\_thisInput.val(eval(\_thisInput.val()) + 1);

1.url定义



1. js定义

itemNumChange : **function**(){

$(".increment").click(**function**(){//＋

**var** \_thisInput = $(**this**).siblings("input");

\_thisInput.val(eval(\_thisInput.val()) + 1);

$.post("/service/cart/update/num/"+\_thisInput.attr("itemId")+"/"+\_thisInput.val(),**function**(data){

//$.post("/cart/update/num/"+\_thisInput.attr("itemId")+"/"+\_thisInput.val()+".html",function(data){

TTCart.refreshTotalPrice(); //将商品的数量和价格乘积完成后回显.

});

});

### 编辑前台Controller

@RequestMapping("/update/num/{itemId}/{num}")

@ResponseBody

**public** SysResult updateCartNum(@PathVariable Long itemId,@PathVariable Integer num){

**try** {

//1.获取用户Id

Long userId = 7L;

Cart cart = **new** Cart();

cart.setUserId(userId);

cart.setItemId(itemId);

cart.setNum(num);

cartService.updateCartNum(cart);

**return** SysResult.*oK*();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201, "购物车数量修改失败");

}

### 修改前台Service

@Override

**public** **void** updateCartNum(Cart cart) {

String url = "http://cart.jt.com/cart/update/num/"

+cart.getUserId()+"/"+cart.getItemId()+"/"+cart.getNum();

httpClient.doGet(url);

}

### 编辑后台Controller

//实现cart数量的修改

@RequestMapping("/update/num/{userId}/{itemId}/{num}")

@ResponseBody

**public** SysResult updateCartNum(

@PathVariable Long userId,

@PathVariable Long itemId,

@PathVariable Integer num){

**try** {

Cart cart = **new** Cart();

cart.setUserId(userId);

cart.setItemId(itemId);

cart.setNum(num);

cartService.updateCartNum(cart);

**return** SysResult.*oK*();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201, "购物车数量修改失败");

}

### 编辑后台Service

@Override

**public** **void** updateCartNum(Cart cart) {

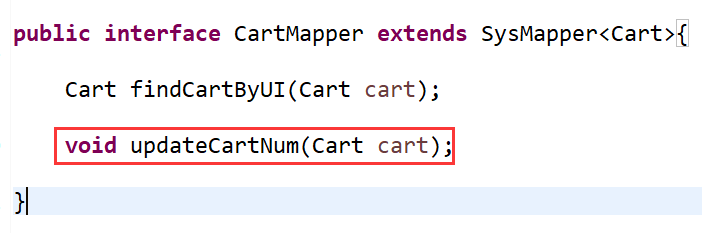
cart.setUpdated(**new** Date());

cartMapper.updateCartNum(cart);

}

### 编辑后台Mapper

1.编辑接口文件



1. 编辑映射文件

<!--修改购物车数量 -->

<update id=*"updateCartNum"*>

update tb\_cart set num = #{num}, updated = #{updated} where

item\_id = #{itemId} and user\_id = #{userId}

</update>

# 实现京淘权限控制

## 业务需求

### 需求说明

规定:如果用户没有登陆.则不能直接访问购物车列表数据/订单数据/物流数据等.

预案:如果用户没有登陆则先让用户跳转到登陆页面.

实现:使用SpringMVC提供的拦截器

1. 定义拦截路径(拦截的业务)
2. 拦截器中具体的操作.

### 使用拦截器步骤

1. 首先判断用户是否登录

1.1判读用户是否有cookie

1.2根据token数据检测redis缓存中是有用户信息

2.如果用户没有登陆则转化到用户登陆页面.如果用户已经登陆则拦截器放行

### 配置拦截器

<!--springMVC拦截器配置

Content Model : (mapping+, exclude-mapping\*, (bean | ref))

/\* 表示只拦截一级目录 /cart/show

/\*\* 表示拦截多级目录 /cart/show/aaa/bbb

<mvc:exclude-mapping path=""/>

-->

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=*"/cart/\*\*"*/>

<bean class=*"com.jt.web.intercept.UserInterceptor"*/>

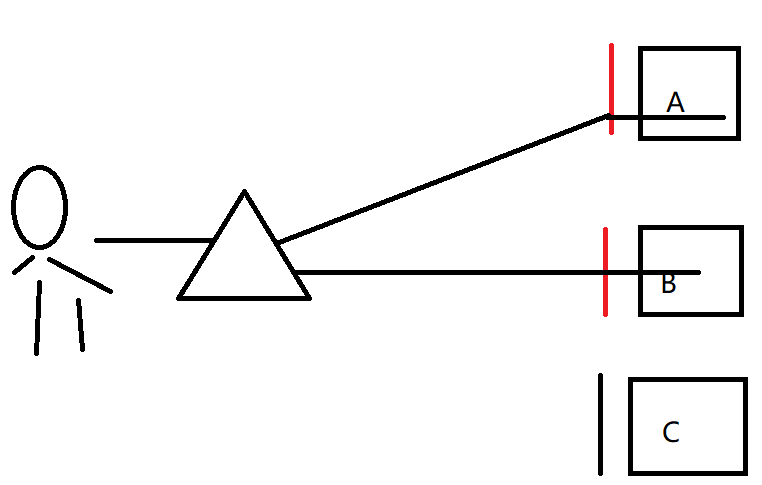
</mvc:interceptor>

</mvc:interceptors>

### 如何实现动态获取用户

说明:一般公司中可以通过域的方式动态的获取用户信息.

例如:session域.和session不能共享.有没有关系



采用拦截器可以在特定的服务器中实现用户信息通过Session获取.因为操作的是同一个服务器.同一个线程.

用户通过Session获取数据的案例:

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

User user = ***objectMapper***.readValue(userJSON, User.**class**);

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

**return** **true**;//表示用户已经登陆 可以放行

}

服务器端代码

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

Long userId = user.getId(); //暂时写死

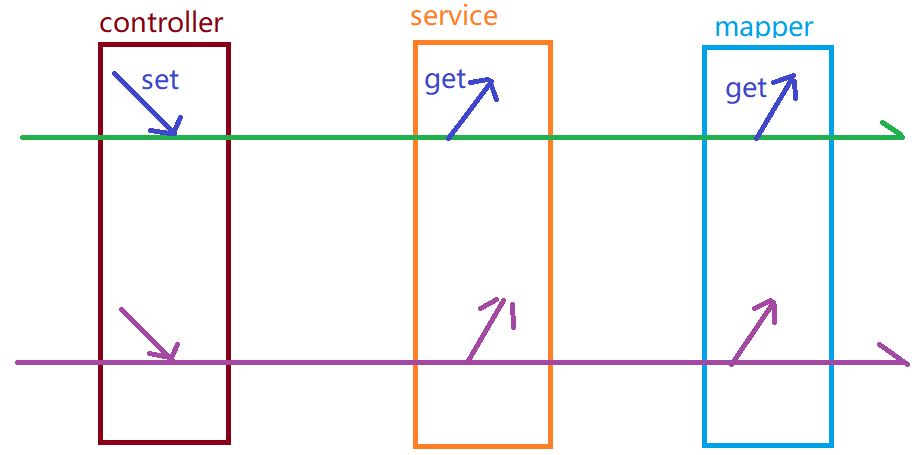
该方式问题:

1. 如果需要获取用户信息,那么在Controller的方法中必须添加Request对象.
2. 如果在业务层想要获取用户信息.那么必须通过传参的方式从controller将参数传递到Service层中.

## ThreadLocal

### 概念

说明:使用ThreadLocal**可以实现在一个线程内实现数据共享**.它是JDK元素提供的API.并且内部实现时线程是安全的.



说明:一般使用ThreadLocal传递数据都是公用的数据.ThreadLocal底层实现时.利用JVM底层直接创建的线程.所以gc(垃圾回收器)不能回收该线程.所以一般使用完之后需要手动关闭.否则必然内存泄漏.

### ThreadLocal工具类实现

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

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

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

*thread*.set(user);

}

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

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

}

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

*thread*.remove();

}

}

## 实现用户信息动态获取

### 编辑拦截器类

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

@Autowired

**private** JedisCluster jedisCluster;

**private** **static** **final** ObjectMapper ***objectMapper*** = **new** ObjectMapper();

/\*\*

\* 1.在调用controller方法之前拦截

\* boolean 代表

\* true代表放行 false表示拦截

\* 拦截器使用用户登陆校验

\* 1.获取客户端端的Cookie

\* 2.判断cookie是否有token数据

\* 3.判断redis中是否有用户json数据

\* 如果用户都满足要求则放行.否则跳转登录页面

\*/

@Override

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

**throws** Exception {

//1.获取客户端Cookie

Cookie[] cookies = request.getCookies();

//2.获取token数据

String token = **null**;

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

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

token = cookie.getValue();

**break**;

}

}

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

//表示用户已经含有token,判断redis中是否有数据

String userJSON = jedisCluster.get(token);

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

User user = ***objectMapper***.readValue(userJSON, User.**class**);

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

//将User数据保存到ThreadLocal中

UserThreadLocal.*set*(user);

**return** **true**;//表示用户已经登陆 可以放行

}

}

//如果程序执行到这里表示用户登陆有误,重定向到登陆页面

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

**return** **false**;

}

//在业务逻辑执行完成后拦截

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

// **TODO** Auto-generated method stub

}

//在业务逻辑执行完之后返回给客户端之前拦截

@Override

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

**throws** Exception {

//为了防止内存泄漏关闭ThreadLocal

UserThreadLocal.*remove*();

}

}

### 使用数据动态获取

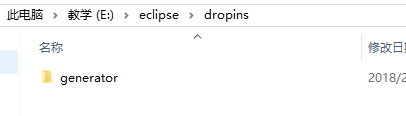
说明:使用ThreadLocal可以在任何地方获取数据



# 订单模块业务实现

## 代码自动生成工具

### 添加插件



### 配置文件介绍

#表示是否不生成注释

suppressAllComments=true

#表示数据库驱动包

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/jtdb?characterEncoding=utf-8

username=root

password=root

#代码生成到哪个项目中

targetProject=jt-order

#pojo生成的包路径

modeltargetPackage=com.jt.order.pojo

#映射文件生成的包路径

sqltargetPackage=com.jt.order.mapper

#mapper接口文件所在包路径

clienttargetPackage=com.jt.order.mapper

### 编辑插件配置文件

<generatorConfiguration>

<!--加载配置文件 -->

<properties resource=*"generatorConfig.properties"*/>

<!--数据库驱动包路径 -->

<classPathEntry location=*"E:\WorkJarSource\connDriver\mysql-connector-java-5.1.10-bin.jar"* />

<context id=*"tarena"*>

<commentGenerator>

<property name=*"suppressAllComments"* value=*"${suppressAllComments}"*/>

</commentGenerator>

<jdbcConnection driverClass=*"${driverClass}"* connectionURL=*"${url}"* userId=*"${username}"* password=*"${password}"* />

<javaModelGenerator targetPackage=*"${modeltargetPackage}"* targetProject=*"${targetProject}"* />

<sqlMapGenerator targetPackage=*"${sqltargetPackage}"* targetProject=*"${targetProject}"* />

<javaClientGenerator targetPackage=*"${clienttargetPackage}"* targetProject=*"${targetProject}"* type=*"XMLMAPPER"* />

<!--table标签表示使用的表有哪些 -->

<table tableName=*"tb\_order"* domainObjectName=*"Order"* enableCountByExample=*"false"* enableUpdateByExample=*"false"*

enableDeleteByExample=*"false"* enableSelectByExample=*"false"* selectByExampleQueryId=*"false"*/>

<table tableName=*"tb\_order\_shipping"* domainObjectName=*"OrderShipping"* enableCountByExample=*"false"* enableUpdateByExample=*"false"*

enableDeleteByExample=*"false"* enableSelectByExample=*"false"* selectByExampleQueryId=*"false"*/>

<table tableName=*"tb\_order\_item"* domainObjectName=*"OrderItem"* enableCountByExample=*"false"* enableUpdateByExample=*"false"*

enableDeleteByExample=*"false"* enableSelectByExample=*"false"* selectByExampleQueryId=*"false"*/>

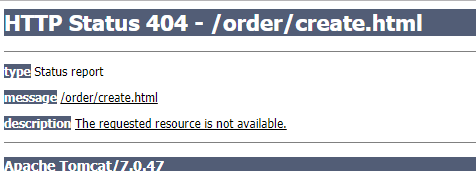
</context>

</generatorConfiguration>

## 跳转到订单确认页面

### 页面分析

说明:当点击去结算按钮时,要跳转到订单确认页面



### 编辑Controller

@Controller

@RequestMapping("/order")

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

@Autowired

**private** CartService cartService;

//http://www.jt.com/order/create.html

@RequestMapping("/create")

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

//展现购物车商品数据

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

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

model.addAttribute("carts", carts);

**return** "order-cart";

}

}

### 编辑mvc配置文件

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=*"/cart/\*\*"*/>

<mvc:mapping path=*"/order/\*\*"*/>

<bean class=*"com.jt.web.intercept.UserInterceptor"*/>

</mvc:interceptor>

</mvc:interceptors>

### 页面效果



## 订单入库操作

### SpringMVC为参数赋值高级应用

案例1:参数接收必须保证name属性的值与接收参数名称一致

<input name=”username” type=”text” />

Public String add(String username){…}

案例2:利用SpringMVC为对象赋值

<input name=”username” type=”text” />

Public String add(User user){…}

案例3:**springMVC可以为对象的引用赋值**

Public class dog{dogName…..}

Public class user{

private Dog dog;

}

<input name=”dog.dogName” type=”text” />

Public String show(User user){

}

案例4:利用SpringMVC可以为对象的属性集合赋值

<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}*"*/>

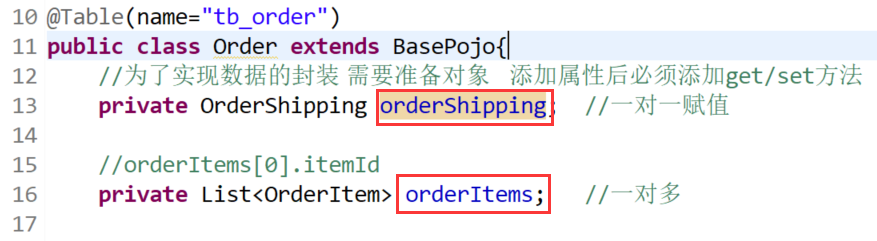
<input type=*"hidden"* name=*"orderItems[*${status.index}*].totalFee"* value=*"*${cart.itemPrice \* cart.num}*"*/>

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

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

</c:forEach>

对象的封装:



### 页面分析

1. url请求



1. 页面JS



### 编辑前台Controller

//实现订单入库操作 /service/order/submit

@RequestMapping("/submit")

@ResponseBody

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

**try** {

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

order.setUserId(userId);

String orderId = orderService.saveOrder(order);

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

**throw** **new** RuntimeException();

}

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

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201, "订单入库失败");

}

### 编辑前台Service

@Override

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

String url = "http://order.jt.com/order/create";

String orderId = **null**;

**try** {

String orderJSON = ***objectMapper***.writeValueAsString(order);

Map<String, String> params = **new** HashMap<>();

params.put("orderJSON", orderJSON);

//从后台获取的返回值数据 采用sysResult返回

String resultJSON = httpClient.doPost(url, params);

SysResult sysResult = ***objectMapper***.readValue(resultJSON, SysResult.**class**);

**if**(sysResult.getStatus() == 200){

orderId = (String) sysResult.getData();

}

} **catch** (Exception e) {

e.printStackTrace();

**throw** **new** RuntimeException();

}

**return** orderId;

}

### 编辑后台Controller

@Controller

@RequestMapping("/order")

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

@Autowired

**private** OrderService orderService;

**private** **static** **final** ObjectMapper ***objectMapper*** = **new** ObjectMapper();

//实现后台订单入库

@RequestMapping("/create")

@ResponseBody

**public** SysResult saveOrder(String orderJSON){

**try** {

Order order = ***objectMapper***.readValue(orderJSON, Order.**class**);

String orderId = orderService.saveOrder(order);

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

**throw** **new** RuntimeException();

}

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

} **catch** (Exception e) {

e.printStackTrace();

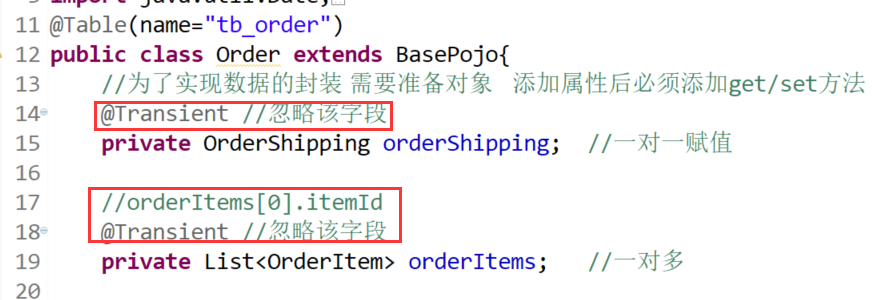
}

**return** SysResult.*build*(201,"新增订单失败");

}

}

### 编辑后台POJO



### 编辑后台Service

@Service

**public** **class** OrderServiceImpl **implements** OrderService {

@Autowired

**private** OrderMapper orderMapper;

@Autowired

**private** OrderItemMapper orderItemMapper;

@Autowired

**private** OrderShippingMapper orderShippingMapper;

//登录用户id+当前时间戳

@Override

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

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

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

//实现订单入库操作

order.setOrderId(orderId);

order.setStatus(1); // 表示未付款

order.setCreated(date);

order.setUpdated(date);

orderMapper.insert(order);

System.***out***.println("订单数据入库成功!!!!!");

//实现订单物流入库

OrderShipping orderShipping = order.getOrderShipping();

orderShipping.setOrderId(orderId);

orderShipping.setCreated(date);

orderShipping.setUpdated(date);

orderShippingMapper.insert(orderShipping);

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

//订单商品入库

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

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

orderItem.setOrderId(orderId);

orderItem.setCreated(date);

orderItem.setUpdated(date);

orderItemMapper.insert(orderItem);

}

System.***out***.println("订单操作入库成功!!!!!");

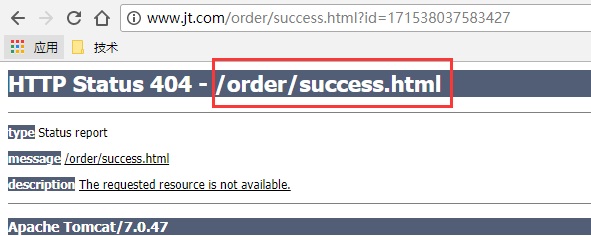
**return** orderId;

}

}

## 订单数据回显

### 页面分析



### 编辑前台Controller

//根据OrderId查询订单信息

@RequestMapping("/success")

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

Order order = orderService.findOrderById(id);

model.addAttribute("order", order);

**return** "success";

}

### 编辑前台Service

@Override

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

String url = "http://order.jt.com/order/query/"+id;

String orderJSON = httpClient.doGet(url);

Order order = **null**;

**try** {

order = ***objectMapper***.readValue(orderJSON, Order.**class**);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** order;

}

### 编辑后台Controller

//根据orderId查询订单信息

@RequestMapping("/query/{orderId}")

@ResponseBody

**public** Order findOrderById(@PathVariable String orderId){

**return** orderService.findOrderById(orderId);

}

### 编辑后台Service

//做关联查询 查询3张表

@Override

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

Order order = orderMapper.selectByPrimaryKey(orderId);

OrderShipping orserShipping = orderShippingMapper.selectByPrimaryKey(orderId);

OrderItem orderItem = **new** OrderItem();

orderItem.setOrderId(orderId);

List<OrderItem> orderItemList = orderItemMapper.select(orderItem) ;

//将数据封装

order.setOrderShipping(orserShipping);

order.setOrderItems(orderItemList);

**return** order;

}

### 页面效果



# 定时任务

## 业务需求

### 业务场景介绍

1. 验证码.

策略:1.将验证码存redis中 2.定时任务将验证码删除

1. QQ定期祝福生日快乐
2. 定时发送短信
3. 订单超时处理

### 工具介绍

1. timer

Timer,[定时器](https://baike.baidu.com/item/%E5%AE%9A%E6%97%B6%E5%99%A8/5109454)，功能是在指定的时间间隔内反复触发指定窗口的定时器事件[1]  。

功能单一:只能指定间隔执行定时任务.

发短信

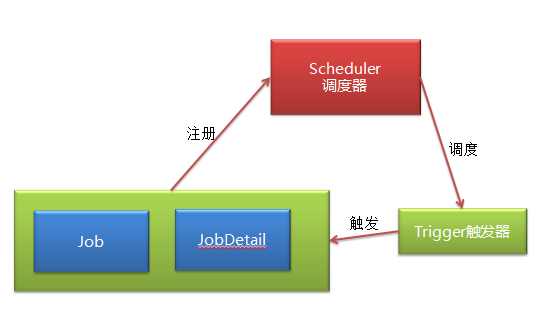
1. Quartz 石英钟定时任务

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。

### 业务需求

说明:如果用户新增订单后,.常时间没有支付,则应该将订单关闭.将状态信息由1改为6.

### 定时任务调用原理



调度器:负载时间监控,如果任务到了指定时间则调用器开始工作.

触发器:当调度器执行任务时,会调用触发器开启线程完成任务.

JOB:定时任务统称.

JOBDetail: 定时任务的具体的细节

调用过程:

1. 新创建定时任务.将任务交给调度器管理
2. 调用器负责监控时间,当到了指定的时间后.调用触发器开启新的线程去完成任务.
3. 当触发器接收到执行指令后.开启线程完成具体的任务.

## Spring整合Quartz

### 添加jar包

<!-- 定时任务 -->

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.2.1</version>

</dependency>

<!--定时任务需要依赖c3p0jar包 -->

<dependency>

<groupId>c3p0</groupId>

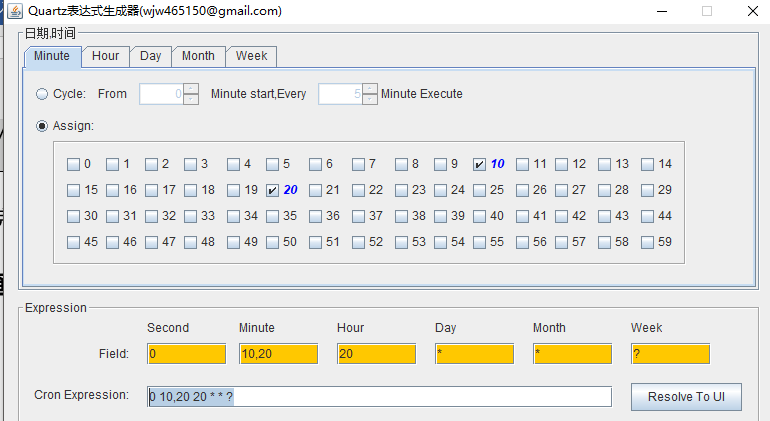
<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

添加完成后将项目打包

### 查看表达式生成器



### 编辑配置文件

<!-- 定义任务bean -->

<bean name=*"paymentOrderJobDetail"* class=*"org.springframework.scheduling.quartz.JobDetailFactoryBean"*>

<!-- 指定具体的job类 -->

<property name=*"jobClass"* value=*"com.jt.order.job.PaymentOrderJob"* />

<!-- 指定job的名称 -->

<property name=*"name"* value=*"paymentOrder"* />

<!-- 指定job的分组 -->

<property name=*"group"* value=*"Order"* />

<!-- 必须设置为true，如果为false，当没有活动的触发器与之关联时会在调度器中删除该任务 -->

<property name=*"durability"* value=*"true"*/>

<!-- 指定spring容器的key，如果不设定在job中的jobmap中是获取不到spring容器的 -->

<property name=*"applicationContextJobDataKey"* value=*"applicationContext"*/>

</bean>

<!-- 定义触发器 -->

<bean id=*"cronTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"paymentOrderJobDetail"* />

<!-- 每一分钟执行一次 -->

<property name=*"cronExpression"* value=*"0 0/1 \* \* \* ?"* />

</bean>

<!-- 定义调度器 -->

<bean class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"triggers"*>

<list>

<ref bean=*"cronTrigger"* />

</list>

</property>

</bean>

### 编辑定时任务

**public** **class** PaymentOrderJob **extends** QuartzJobBean{

/\*\*

\* 要求:如果用户2天没有支付.则将订单做超时处理

\* 业务逻辑:

\* update tb\_order set status = 6,updated = now()

\* where status = 1 and created < 现在时间-2天

\*/

@Override

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

//删除2天天的恶意订单

ApplicationContext applicationContext = (ApplicationContext) context.getJobDetail().getJobDataMap().get("applicationContext");

//计算2天前的时间

Date agoDate = **new** DateTime().minusDays(2).toDate();

//获取orderMapper接口

OrderMapper orderMapper = applicationContext.getBean(OrderMapper.**class**);

orderMapper.updateStatus(agoDate);

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

}

}

### 编辑Mapper映射文件

<!--执行定时任务 -->

<update id=*"updateStatus"*>

<![CDATA[

update tb\_order set status = 6,updated = now() where

status = 1 and created < #{agoDate}

]]>

</update>