# 项目说明

1. 项目描述： 一个纯后端的购物网站项目，个人独立完成，主要包括用户模块、分类模块、
2. 商品模块、购物车模块、收获地址模块和订单模块。
3. 技术栈： Spring、 Spring MVC、 Mybatis、 Mysql、 Redis、 Maven、 Nginx、 Tomcat、 Git
4. 项目内容：
   1. 实现了注册登录-搜索-购物车-提交订单-支付的电商购物基本流程。
   2. 实现项目从单服务器到 Tomcat 集群的架构演进及迭代开发。
   3. 实现从 Guava Cache 到 Redis 单服务再到 Redis 分布式的迁移。
   4. 使用 Redis+Cookie+Jackson+Filter 实现单点登录。
   5. 使用 Spring Schedule + Redis 实现分布式锁及定时关单。
   6. 使用 SpringMVC 进行全局异常处理和权限统一校验。

# 版本一说明

## 2.1 版本一主要内容

1. 项目描述： 一个纯后端的购物网站项目，个人独立完成，主要包括用户模块、
2. 分类模块、 商品模块、购物车模块、收获地址模块和订单模块。
3. 技术栈： Spring+Spring MVC+MyBatis+Tomcat+Maven+MySql+PageHelper+Nginx.
4. 项目内容：
5. 完成项目环境搭建与初始化(git 版本控制、 各种软件的环境搭建)。
6. 完成数据库表设计，主要包括(购物车表、商品类别表、订单表、支付信息表、
7. 用户表等)。
8. 完成表现层、业务层和持久层项目的分层构建；前台、后台以及订单系统等分
9. 开设计，方便后续扩展和升级。
10. 完成代码编写，并使用 Restlet 完成功能测试。

## 2.2 版本一使用到的技术和使用细节

### 2.2.1 SSM

### 2.2.2 Mysql

#### 2.2.2.1 数据库表设计

1. 用户表mmall\_user

主键：PRIMARY KEY (`id`)

密码：password 存储的是md5加密的密码

UNIQUE KEY：`user\_name\_unique` (`username`) USING BTREE

2. 商品表mmall\_product

主键：PRIMARY KEY (`id`)

2.1商品分类表mmall\_category

3. 订单表mmall\_order，订单商品item表mmall\_order\_item

4. 购物车表mmall\_cart

5. 支付信息表mmall\_pay\_info

6. 订单信息表mmall\_shipping

### 2.2.3 Maven

### 2.2.4 Nginx

### 2.2.5 Tomcat

### 2.2.6 Git

# 单点登陆

# 全局异常

## 4.1 Spring异常处理

<https://www.cnblogs.com/junzi2099/p/7840294.html>

Spring 统一异常处理有 3 种方式，分别为：

1. 使用 @ ExceptionHandler 注解
2. 实现 HandlerExceptionResolver 接口
3. 使用 @controlleradvice 注解

### 4.1.1 使用@ ExceptionHandler 注解

使用该注解有一个不好的地方就是：进行异常处理的方法必须与出错的方法在同一个Controller里面。使用如下：

|  |
| --- |
| @Controller  public class GlobalController {  /\*\*  \* 用于处理异常的  \* @return  \*/  @ExceptionHandler({MyException.class})  public String exception(MyException e) {  System.out.println(e.getMessage());  e.printStackTrace();  return "exception";  }  @RequestMapping("test")  public void test() {  throw new MyException("出错了！");  }  } |

### 4.1.2 实现 HandlerExceptionResolver 接口

这种方式可以进行全局的异常控制。例如：

|  |
| --- |
| @Slf4j  @Component  public class ExceptionResolver implements HandlerExceptionResolver{  @Override  public ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) {  log.error("{} Exception",httpServletRequest.getRequestURI(),e);  ModelAndView modelAndView = new ModelAndView(new MappingJacksonJsonView());  //当使用是jackson2.x的时候使用MappingJackson2JsonView，课程中使用的是1.9。  modelAndView.addObject("status", ResponseCode.ERROR.getCode());  modelAndView.addObject("msg","接口异常,详情请查看服务端日志的异常信息");  modelAndView.addObject("data",e.toString());  return modelAndView;  }  } |

### 4.1.3 使用 @ControllerAdvice+ @ ExceptionHandler 注解

上文说到 @ ExceptionHandler 需要进行异常处理的方法必须与出错的方法在同一个Controller里面。那么当代码加入了 @ControllerAdvice，则不需要必须在同一个 controller 中了。这也是 Spring 3.2 带来的新特性。从名字上可以看出大体意思是控制器增强。 也就是说，@controlleradvice + @ ExceptionHandler 也可以实现全局的异常捕捉。

**请确保此WebExceptionHandle 类能被扫描到并装载进 Spring 容器中。**

|  |
| --- |
| @ControllerAdvice  @ResponseBody  public class WebExceptionHandle {  private static Logger logger = LoggerFactory.getLogger(WebExceptionHandle.class);  /\*\*  \* 400 - Bad Request  \*/  @ResponseStatus(HttpStatus.BAD\_REQUEST)  @ExceptionHandler(HttpMessageNotReadableException.class)  public ServiceResponse handleHttpMessageNotReadableException(HttpMessageNotReadableException e) {  logger.error("参数解析失败", e);  return ServiceResponseHandle.failed("could\_not\_read\_json");  }    /\*\*  \* 405 - Method Not Allowed  \*/  @ResponseStatus(HttpStatus.METHOD\_NOT\_ALLOWED)  @ExceptionHandler(HttpRequestMethodNotSupportedException.class)  public ServiceResponse handleHttpRequestMethodNotSupportedException(HttpRequestMethodNotSupportedException e) {  logger.error("不支持当前请求方法", e);  return ServiceResponseHandle.failed("request\_method\_not\_supported");  }  /\*\*  \* 415 - Unsupported Media Type  \*/  @ResponseStatus(HttpStatus.UNSUPPORTED\_MEDIA\_TYPE)  @ExceptionHandler(HttpMediaTypeNotSupportedException.class)  public ServiceResponse handleHttpMediaTypeNotSupportedException(Exception e) {  logger.error("不支持当前媒体类型", e);  return ServiceResponseHandle.failed("content\_type\_not\_supported");  }  /\*\*  \* 500 - Internal Server Error  \*/  @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)  @ExceptionHandler(Exception.class)  public ServiceResponse handleException(Exception e) {  if (e instanceof BusinessException){  return ServiceResponseHandle.failed("BUSINESS\_ERROR", e.getMessage());  }    logger.error("服务运行异常", e);  e.printStackTrace();  return ServiceResponseHandle.failed("server\_error");  }  } |

如果 @ExceptionHandler 注解中未声明要处理的异常类型，则默认为参数列表中的异常类型。所以还可以写成这样：

|  |
| --- |
| @ControllerAdvice  public class GlobalExceptionHandler {  @ExceptionHandler()  @ResponseBody  String handleException(Exception e){  return "Exception Deal! " + e.getMessage();  }  } |

参数对象就是 Controller 层抛出的异常对象！

继承 ResponseEntityExceptionHandler 类来实现针对 Rest 接口 的全局异常捕获，并且可以返回自定义格式：

|  |
| --- |
| @Slf4j  @ControllerAdvice  public class ExceptionHandlerBean extends ResponseEntityExceptionHandler {  /\*\*  \* 数据找不到异常  \* @param ex  \* @param request  \* @return  \* @throws IOException  \*/  @ExceptionHandler({DataNotFoundException.class})  public ResponseEntity<Object> handleDataNotFoundException(RuntimeException ex, WebRequest request) throws IOException {  return getResponseEntity(ex,request,ReturnStatusCode.DataNotFoundException);  }  /\*\*  \* 根据各种异常构建 ResponseEntity 实体. 服务于以上各种异常  \* @param ex  \* @param request  \* @param specificException  \* @return  \*/  private ResponseEntity<Object> getResponseEntity(RuntimeException ex, WebRequest request, ReturnStatusCode specificException) {  ReturnTemplate returnTemplate = new ReturnTemplate();  returnTemplate.setStatusCode(specificException);  returnTemplate.setErrorMsg(ex.getMessage());  return handleExceptionInternal(ex, returnTemplate,  new HttpHeaders(), HttpStatus.OK, request);  }  } |

### 4.1.4 区别

@ControllerAdvice只是标注当前bean是对Controller增强，具体是增强什么，则是配合@ExceptionHandler对异常进行精确处理。HandlerExceptionResolver则可以进行全局的异常控制，因此，所有的controller发生的异常都会进入其实现类，我们需要做通用处理。查看源码，可以发现HandlerExceptionResolver接口的抽象类AbstractHandlerMethodExceptionResolver和其子类ExceptionHandlerExceptionResolver一起组成使用@ExceptionHandler注释的方法进行异常解析的功能。现在基本都使用@ControllerAdvice+@ExceptionHandler注解来实现全局的异常捕获。

# 拦截器

## 5.1 使用Spring MVC拦截器实现权限统一校验

### 5.1.1 配置

1. dispatcher-servlet.xml

|  |
| --- |
| <mvc:interceptors>  <!-- 定义在这里的，所有的都会拦截-->  <mvc:interceptor>  <!--manage/a.do /manage/\*-->  <!--manage/b.do /manage/\*-->  <!--manage/product/save.do /manage/\*\*-->  <!--manage/order/detail.do /manage/\*\*-->  <mvc:mapping path="/manage/\*\*"/>  <mvc:exclude-mapping path="/manage/user/login.do"/>  <bean class="com.mmall.controller.common.interceptor.AuthorityInterceptor" />  </mvc:interceptor>  </mvc:interceptors> |

分析配置中的节点

* + interceptors节点：这个节点是SpringMVC的拦截器集配置节点，在这个节点里面我们可以声明多个interceptor
  + interceptor节点：这个节点是配置拦截路径以及你拦截器实现类的节点
    - mapping节点：符合mapping路径匹配的请求都会经过拦截器
    - exclude-mapping节点：符合exclude-mapping路径匹配的请求都不会经过拦截器
    - bean节点：配置我们自己的实现类，实现类要实现HandlerInterceptor接口

path情况分析

* + 如果要拦截同一个controller下的所有的请求，如/manage/\*\*\*.do，path应该为 /manage/\*
  + 如果要拦截请求路径为manage包下不同controller的所有请求，如 /manage/product/b.do，/manage/order/list.do，path应该为/manage/\*\*
  + 如果想要某个请求不走拦截器，那么可以配置exclude-mapping，path应该为具体的url，准确到方法，如/manage/login.do

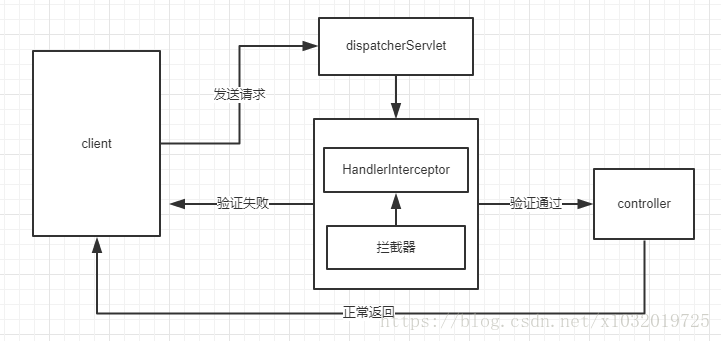
### 5.1.2 使用，coding

|  |
| --- |
| /\*\*  \* Created by dnhua  \*/  @Slf4j  public class AuthorityInterceptor implements HandlerInterceptor{  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  return true;  }  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  log.info("postHandle");  }  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  log.info("afterCompletion");  }  } |

spring MVC拦截器请求流程

spring请求流程具体如下：

* 1. 浏览器/客户端发送请求给我们的服务器
  2. 请求来到Spring的dispatchServlet进行请求分发
  3. dispatchServlet检查我们在dispatch-servlet.xml定义好的规则，按照规则判断请求是否走拦截器
  4. 如果判断请求不走拦截器，请求直接分发到对应的controller(这里默认有对应的controller)
  5. 如果判断请求是走拦截器的，就走我们自己实现的拦截器，验证请求，通过就放行，走到controller，反之则提前返回给客户端



### 5.1.3 原理分析-关键源码分析

#### 5.1.3.1 简单回答

#### 5.1.3.2 源码分析

* HandlerInterceptor接口
* HandlerExecutionChain类
* HandlerMapping接口
* DispatcherServlet类

1. HandlerInterceptor接口

|  |
| --- |
| default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  return true;  }  default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView) throws Exception {  }  default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,  @Nullable Exception ex) throws Exception {  } |

preHandle是在找到处理handler对象的HandlerMapping之后，HandlerAdapter调度handler之前执行。

postHandle是在HandlerAdapter调度handler之后，DispatcherServlet渲染视图之前执行，可以通过ModelAndView来向视图中添加一些信息等。

afterCompletion是在渲染视图结束后执行，主要可以用来进行事后的资源清理。

2. HandlerExecutionChain类

从函数名中我们可以看出这些方法分别是做什么的，分别是执行interceptorList中所有interceptor的preHandle、postHandle和afterCompletion方法。

先从applyPreHandle()看起，我们发现这个方法就是做的这样一个工作，按照列表中interceptor的顺序来执行它们的preHandle方法，直到有一个返回false。再看一下返回false后这个方法所做的工作，这时会调用triggerAfterCompletion方法，此时this.interceptorIndex指向上一个返回true的interceptor的位置，所以它会按逆序执行所有返回true的interceptor的afterCompletion方法。换言之，也就是对于任意的返回false的interceptor都不会执行afterCompletion方法。而且是中断之前所有的preHandle执行完成之后才会执行afterCompletion方法。

接下来是applyPostHandle()，这个方法较为简单，就是按照逆序执行所有interceptor的postHandle方法。

最后的triggerAfterCompletion()也是一样，就是从最后一次preHandle成功的interceptor处逆序执行afterCompletion。

3. HandlerMapping接口

HandlerExecutionChain是通过HandlerMapping的getHandler方法返回的。

继承该接口的类是来实现请求和handler对象的映射关系的。

|  |
| --- |
| HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception; |

4. DispatcherServlet类

DispatcherServlet类中调用HandlerMapping的getHandler的方法为getHandler（同名）

* getHandler
* doDispatch

# RESTful改造

# 七、定时关单

## 7.1 Spring Schedule实现定时关单

### 7.1.1 Spring Schedule的介绍

在日常项目运行中，我们总会有需求在某一时间段周期性的执行某个动作。比如每天在某个时间段导出报表，或者每隔多久统计一次现在在线的用户量。在springboot中可以有很多方案去帮我们完成定时器的工作，有Java自带的java.util.Timer类，也有强大的调度器Quartz，还有SpringBoot自带的Scheduled，今天主要说说Scheduled。

| **框架名称** | **Cron表达式** | **固定间隔执行** | **固定频率执行** | **任务持久化** | **难易度** |
| --- | --- | --- | --- | --- | --- |
| [TimerTask](https://docs.oracle.com/javase/8/docs/api/java/util/TimerTask.html) | 不支持 | 支持 | 支持 | 不支持 | 一般 |
| [schedule](https://spring.io/guides/gs/scheduling-tasks/) | 支持 | 支持 | 支持 | 不支持 | 简单 |
| [Quartz](https://baike.baidu.com/item/quartz/3643055) | 支持 | 支持 | 支持 | 支持 | 难 |

在实际应用中，如果没有分布式场景(quartz 支持分布式, schedule 不支持(需要自己实现,用分布式锁)，schedule跟spring结合的更好，还是很适用的。

### 7.1.2 Cron表达式

cron一共有7位，但是最后一位是年，可以留空，所以我们可以写6位：

* 第一位，表示秒，取值0-59
* 第二位，表示分，取值0-59
* 第三位，表示小时，取值0-23
* 第四位，日期天/日，取值1-31
* 第五位，日期月份，取值1-12
* 第六位，星期，取值1-7，星期一，星期二...，注：不是第1周，第二周的意思 另外：1表示星期天，2表示星期一。
* 第七位，年份，可以留空，取值1970-2099

cron中，还有一些特殊的符号，含义如下：

|  |
| --- |
| (\*)星号：可以理解为每的意思，每秒，每分，每天，每月，每年...  (?)问号：问号只能出现在日期和星期这两个位置，表示这个位置的值不确定，每天3点执行，所以第六位星期的位置，我们是不需要关注的，就是不确定的值。同时：日期和星期是两个相互排斥的元素，通过问号来表明不指定值。比如，1月10日，比如是星期1，如果在星期的位置是另指定星期二，就前后冲突矛盾了。  (-)减号：表达一个范围，如在小时字段中使用“10-12”，则表示从10到12点，即10,11,12  (,)逗号：表达一个列表值，如在星期字段中使用“1,2,4”，则表示星期一，星期二，星期四  (/)斜杠：如：x/y，x是开始值，y是步长，比如在第一位（秒） 0/15就是，从0秒开始，每15秒，最后就是0，15，30，45，60 另：\*/y，等同于0/y |

一些例子：

|  |
| --- |
| "0 0 12 \* \* ?" 每天中午12点触发  "0 15 10 ? \* \*" 每天上午10:15触发  "0 15 10 \* \* ?" 每天上午10:15触发  "0 15 10 \* \* ? \*" 每天上午10:15触发  "0 15 10 \* \* ? 2005" 2005年的每天上午10:15触发  "0 \* 14 \* \* ?" 在每天下午2点到下午2:59期间的每1分钟触发  "0 0/5 14 \* \* ?" 在每天下午2点到下午2:55期间的每5分钟触发  "0 0/5 14,18 \* \* ?" 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发  "0 0-5 14 \* \* ?" 在每天下午2点到下午2:05期间的每1分钟触发  "0 10,44 14 ? 3 WED" 每年三月的星期三的下午2:10和2:44触发  "0 15 10 ? \* MON-FRI" 周一至周五的上午10:15触发  "0 15 10 15 \* ?" 每月15日上午10:15触发  "0 15 10 L \* ?" 每月最后一日的上午10:15触发  "0 15 10 ? \* 6L" 每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6L 2002-2005" 2002年至2005年的每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6#3" 每月的第三个星期五上午10:15触发  每隔5秒执行一次：\*/5 \* \* \* \* ?  每隔1分钟执行一次：0 \*/1 \* \* \* ?  每天23点执行一次：0 0 23 \* \* ?  每天凌晨1点执行一次：0 0 1 \* \* ?  每月1号凌晨1点执行一次：0 0 1 1 \* ?  每月最后一天23点执行一次：0 0 23 L \* ?  每周星期天凌晨1点实行一次：0 0 1 ? \* L |

这里有个在线生成器：<https://tool.lu/crontab/>

### 7.1.3 配置和实战

#### 7.1.3.1 配置

applicationContext.xml

|  |
| --- |
| <task:annotation-driven/> |

#### 7.1.3.2 定时关单版本一

|  |
| --- |
| @Component  @Slf4j  public class CloseOrderTask {  @Autowired  private IOrderService iOrderService;    @Scheduled(cron="0 \*/1 \* \* \* ?")//每1分钟(每个1分钟的整数倍)  public void closeOrderTaskV1(){  log.info("关闭订单定时任务启动");  int hour = Integer.parseInt(PropertiesUtil.getProperty("close.order.task.time.hour","2"));  iOrderService.closeOrder(hour);  log.info("关闭订单定时任务结束");  }  } |

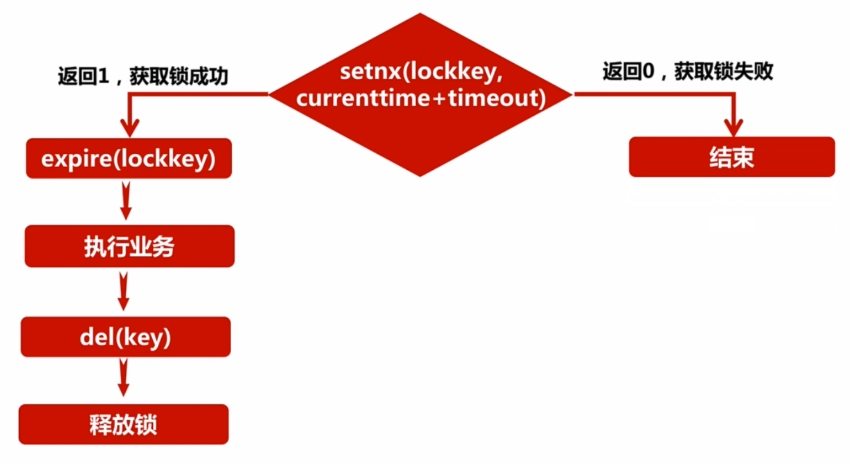
存在的问题：在tomcat集群环境中会被重复执行。

## 7.2 Redis分布式锁-定时关单的升级1

### 7.2.1 Redis分布式锁介绍

相关命令：

* setnx
* getset
* expire
* del



### 7.2.2 实战

|  |
| --- |
| @Scheduled(cron="0 \*/1 \* \* \* ?")  public void closeOrderTaskV2(){  log.info("关闭订单定时任务启动");  long lockTimeout = Long.parseLong(PropertiesUtil.getProperty("lock.timeout","5000"));  Long setnxResult = RedisShardedPoolUtil.setnx(  Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String.valueOf  (System.currentTimeMillis()+lockTimeout));  if(setnxResult != null && setnxResult.intValue() == 1){  //如果返回值是1，代表设置成功，获取锁  closeOrder(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }else{  log.info("没有获得分布式锁:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }  log.info("关闭订单定时任务结束");  }  private void closeOrder(String lockName){  RedisShardedPoolUtil.expire(lockName,5);//有效期5秒，防止死锁  log.info("获取{},ThreadName:{}", Const.REDIS\_LOCK.  CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  int hour = Integer.parseInt(PropertiesUtil.getProperty("close.order.task.time.hour","2"));  iOrderService.closeOrder(hour);  RedisShardedPoolUtil.del(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  log.info("释放{},ThreadName:{}", Const.REDIS\_LOCK.  CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  log.info("===============================");  } |

问题：如果因为某种原因关闭了服务，未释放锁，会发生死锁。

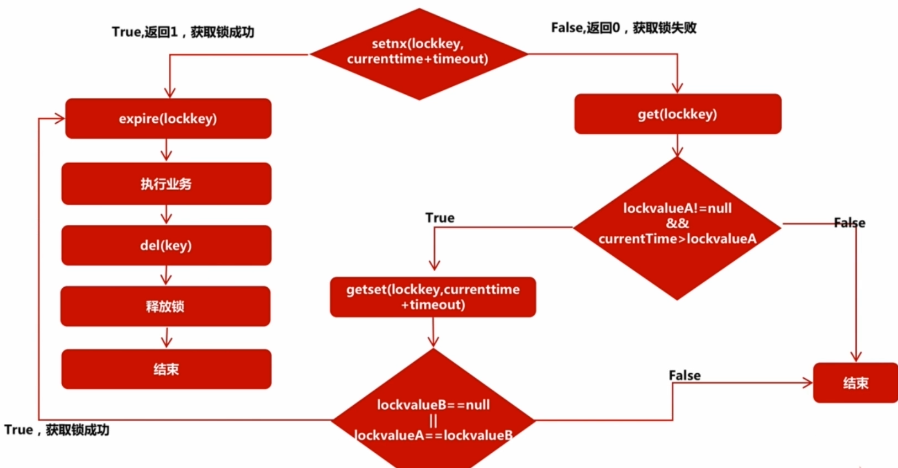
尝试解决，温柔的关闭tomcat，然后加上以下这段代码：

|  |
| --- |
| @PreDestroy  public void delLock(){  RedisShardedPoolUtil.del(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  } |

仍然存在问题，关闭耗时，而且如果是直接kill掉tomcat就无能为力了。

## 7.3 Redis分布式锁-定时关单的升级2

|  |
| --- |
| @Scheduled(cron="0 \*/1 \* \* \* ?")  public void closeOrderTaskV3(){  log.info("关闭订单定时任务启动");  long lockTimeout = Long.parseLong(PropertiesUtil.getProperty("lock.timeout","5000"));  Long setnxResult = RedisShardedPoolUtil.setnx  (Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String.valueOf  (System.currentTimeMillis()+lockTimeout));  if(setnxResult != null && setnxResult.intValue() == 1){  closeOrder(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }else{  //未获取到锁，继续判断，判断时间戳，看是否可以重置并获取到锁  String lockValueStr = RedisShardedPoolUtil  .get(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  if(lockValueStr != null && System.currentTimeMillis() > Long.parseLong(lockValueStr)){  String getSetResult = RedisShardedPoolUtil  .getSet(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String  .valueOf(System.currentTimeMillis()+lockTimeout));  //再次用当前时间戳getset。  //返回给定的key的旧值，->旧值判断，是否可以获取锁  //当key没有旧值时，即key不存在时，返回nil ->获取锁  //这里我们set了一个新的value值，获取旧的值。  if(getSetResult == null || (getSetResult != null && StringUtils.equals(lockValueStr,getSetResult))){  //真正获取到锁  closeOrder(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }else{  log.info("没有获取到分布式锁:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }  }else{  log.info("没有获取到分布式锁:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }  }  log.info("关闭订单定时任务结束");  } |



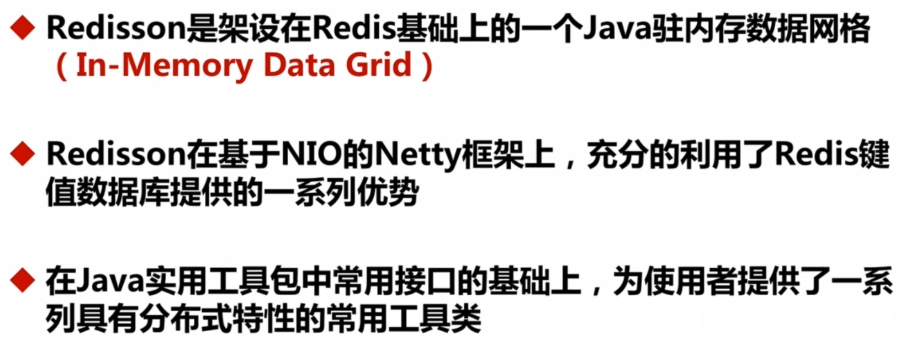
说明：

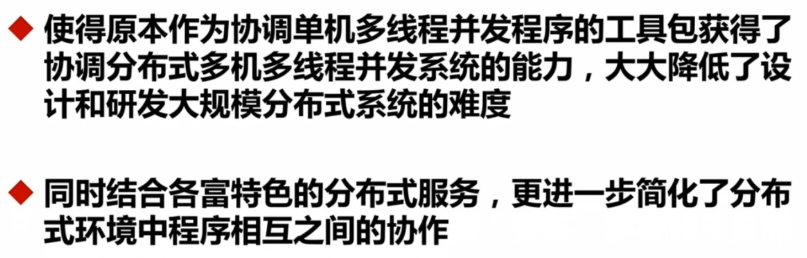
String getSetResult = RedisShardedPoolUtil

.getSet(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String.valueOf(System.currentTimeMillis()+lockTimeout));

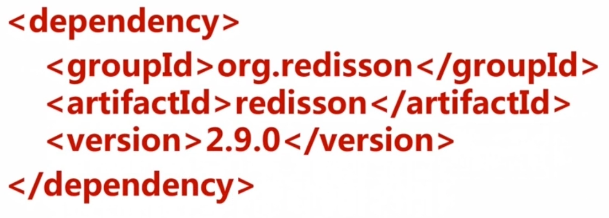
这很重要，把当前时间加上一个时间放进redis用来判断锁的状态。

## 7.4 Redisson框架解决定时关单的问题





* + 1. 添加依赖
    2. 需要添加fasterxml依赖





|  |
| --- |
| @Scheduled(cron="0 \*/1 \* \* \* ?")  public void closeOrderTaskV4(){  RLock lock = redissonManager.getRedisson().getLock(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  boolean getLock = false;  try {  if(getLock = lock.tryLock(0,50, TimeUnit.SECONDS)){  log.info("Redisson获取到分布式锁:{},ThreadName:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  int hour = Integer.parseInt(PropertiesUtil.getProperty("close.order.task.time.hour","2"));  iOrderService.closeOrder(hour);  }else{  log.info("Redisson没有获取到分布式锁:{},ThreadName:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  }  } catch (InterruptedException e) {  log.error("Redisson分布式锁获取异常",e);  } finally {  if(!getLock){  return;  }  lock.unlock();  log.info("Redisson分布式锁释放锁");  }  } |