**云笔记**

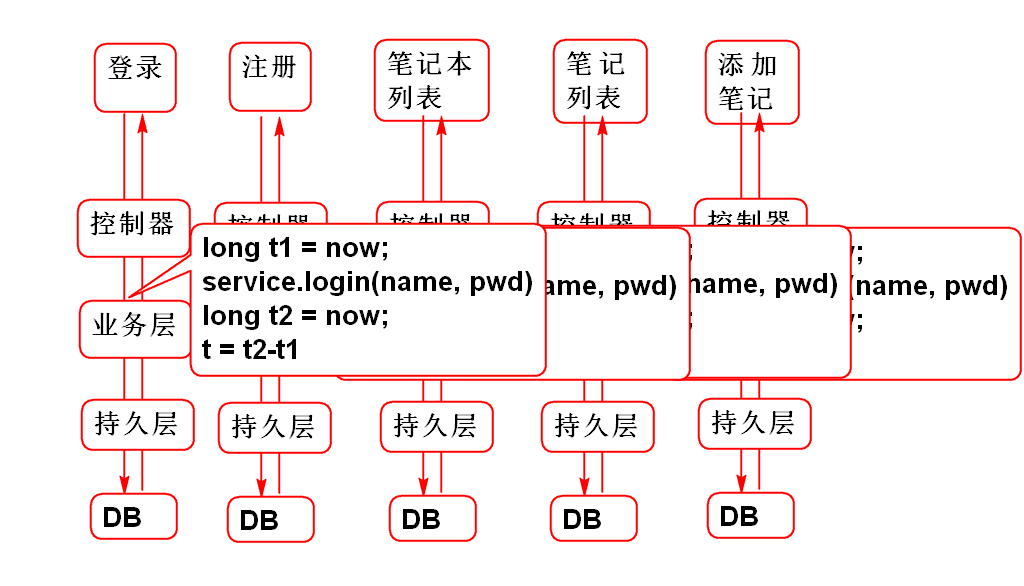
**AOP 面向切面编程**

切面(儿): 事务的横截面

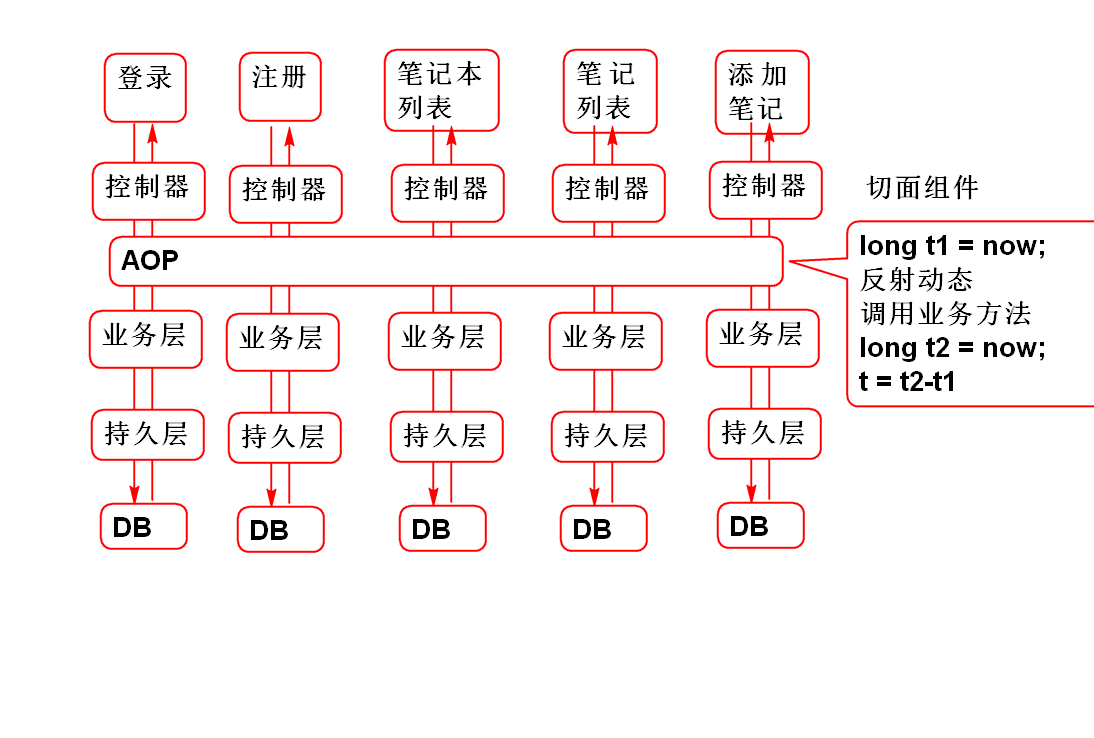
特点: 在不改变软件原有功能情况下为软件插入(扩展)横切面功能.

对于横向功能利用AOP可以大大简化软件的开发:

没有使用AOP:



使用AOP后:



**开发一个AOP案例:**

1. 导入Aspect J 包

Spring AOP 底层利用了AspectJ实现的!

<dependency> <groupId>aspectj</groupId> <artifactId>aspectjweaver</artifactId> <version>1.5.3</version> </dependency>

1. 创建切面组件

@Component @Aspect public class DemoAspect { //声明test方法将在 userService的全部方法之前运行 @Before("bean(userService)") public void test(){ System.out.println("Hello World!"); } }

1. 配置Spring AOP: spring-aop.xml

<!-- 配置组件扫描 --> <context:component-scan base-package="cn.tedu.note.aop"/> <!-- 使 @Aspect 注解生效 --> <aop:aspectj-autoproxy/>

1. 测试

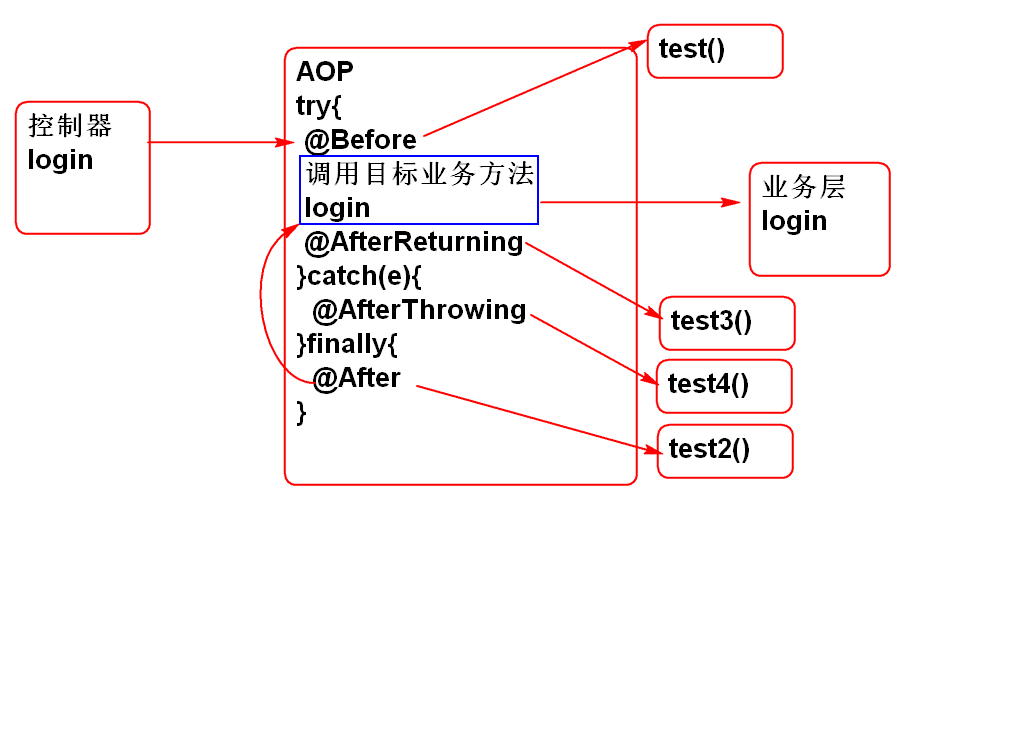
Hello World! 将在userService的业务方法之前执行

**通知**

目标方法: 被AOP拦截的业务方法, 称为目标方法

切面方法在执行时机：就在目标方法之前, 之后执行.

* @Before: 切面方法在目标方法之前执行
* @After: 切面方法在目标方法之后执行
* @AfterReturning: 切面方法在目标方法正常结束之后执行
* @AfterThrowing: 切面方法在目标方法异常之后执行

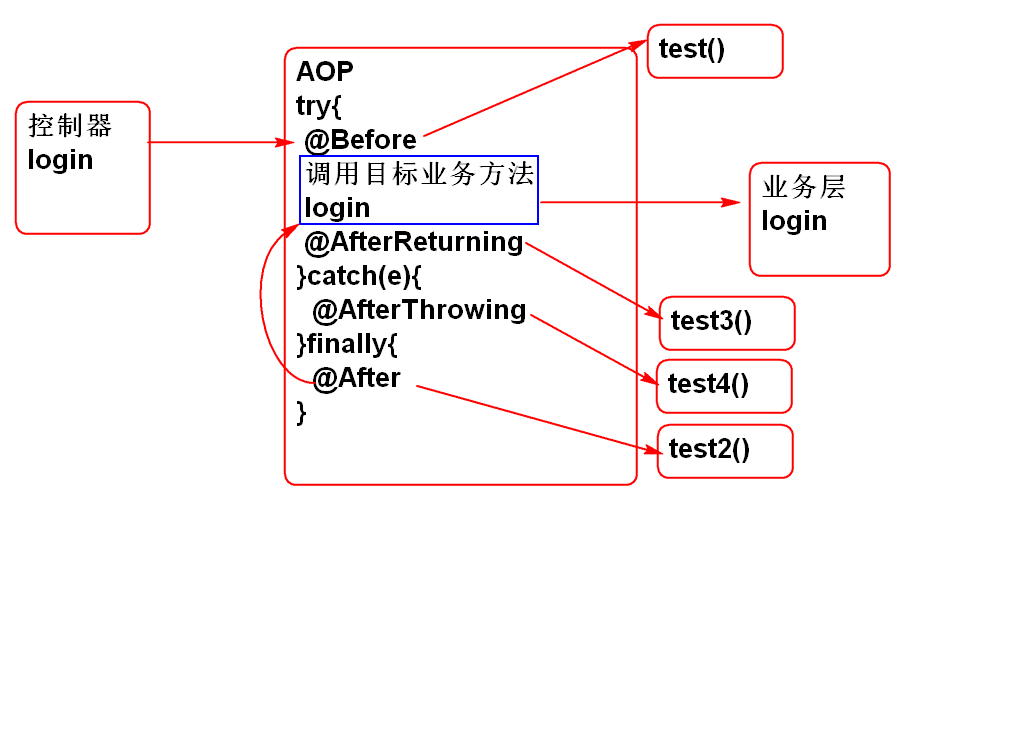


案例:

/\*\* \* 创建一个切面组件, 就是一个普通的JavaBean \*/ @Component @Aspect public class DemoAspect { //声明test方法将在 userService的全部方法之前运行 @Before("bean(userService)") public void test(){ System.out.println("Hello World!"); } @After("bean(userService)") public void test2(){ System.out.println("After"); } @AfterReturning("bean(userService)") public void test3(){ System.out.println("AfterReturning"); } @AfterThrowing("bean(userService)") public void test4(){ System.out.println("AfterThrowing"); } }

**Around 通知**

环绕通知, 可以在业务方法前后调用



案例:

@Component @Aspect public class Demo1Aspect { /\*\* \* 环绕通知 方法: \* 1. 必须有返回值值Object \* 2. 必须有参数 ProceedingJoinPoint \* 3. 必须抛出异常 \* 4. 需要在方法中调用 jp.proceed() \* 5. 返回业务方法的返回值 \* @param jp \* @return \* @throws Throwable \*/ @Around("bean(userService)") public Object test5(ProceedingJoinPoint jp) throws Throwable{ Object val = jp.proceed(); System.out.println("业务结果:"+val); throw new UserNotFoundException( "就是不让登录"); } }

**切入点**

用于定位APO的切入位置: 用于指定切入到具体的方法类

* bean组件切入点
  + bean(userService)
  + bean(noteService)
  + bean(userService) || bean(noteService) || bean(notebookService)
  + bean(\*Service)
* 类切入点
  + within(类名)
  + within(类名) || within(类名)
  + within(cn.tedu.note.service.impl.UserServiceImpl)
  + within(cn.tedu.note.*.impl.*ServiceImpl)
* 方法切入点 (execution: 执行)
  + execution(修饰词 类名.方法名(参数类型))
  + execution(\* cn.tedu.note.service.UserService.login(..))
  + execution(\* cn.tedu.note.*.*Service.list\*(..))

注意: 一致统一的类和方法的命名规则将有助于编写有效的 切入点表达式!

**AOP 底层原理**

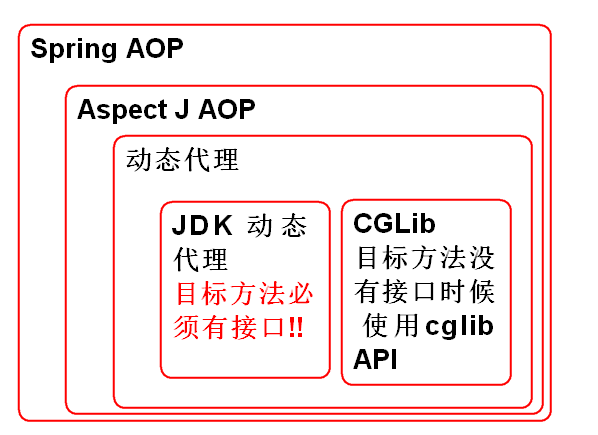
代理模式: 不改变原有功能为软件扩展新功能.

AOP封装了动态代理功能, 提供了更加简便的使用方式!

经典面试问题:

AOP的底层技术是什么? 答案: 使用了 动态代理 技术.

关键点:



1. Spring AOP 利用了 AspectJ AOP实现的!
2. AspectJ AOP 的底层用了动态代理
3. 动态代理有两种
   * 目标方法有接口时候自动选用 JDK 动态代理
   * 目标方法没有接口时候选择 CGLib 动态代理

利用异常可以分析AOP调用原理:

java.lang.NullPointerException at cn.tedu.note.service.impl.UserServiceImpl.login(UserServiceImpl.java:34) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source) at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source) at java.lang.reflect.Method.invoke(Unknown Source) at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:317) at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:183) at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:150) at org.springframework.aop.framework.adapter.MethodBeforeAdviceInterceptor.invoke(MethodBeforeAdviceInterceptor.java:51) at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:172) at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:91) at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:172) at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:204) at com.sun.proxy.$Proxy21.login(Unknown Source) at cn.tedu.note.controller.UserController.login(UserController.java:34) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source) at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source) at java.lang.reflect.Method.invoke(Unknown Source) at org.springframework.web.method.support.InvocableHandlerMethod.invoke(InvocableHandlerMethod.java:215) at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:132) at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:104) at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandleMethod(RequestMappingHandlerAdapter.java:745) at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:686) at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:80) at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:925) at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:856) at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:953) at org.springframework.web.servlet.FrameworkServlet.doPost(FrameworkServlet.java:855) at javax.servlet.http.HttpServlet.service(HttpServlet.java:650) at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:829) at javax.servlet.http.HttpServlet.service(HttpServlet.java:731) at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:303) at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208) at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52) at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:241) at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208) at cn.tedu.note.web.DemoFilter.doFilter(DemoFilter.java:28) at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:241) at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208) at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:220) at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:122) at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:505) at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:169) at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:103) at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:956) at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:116) at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:423) at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1079) at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:625) at org.apache.tomcat.util.net.JIoEndpoint$SocketProcessor.run(JIoEndpoint.java:318) at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source) at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source) at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61) at java.lang.Thread.run(Unknown Source)

**AOP 拦截器 过滤器**

1. 过滤器: 拦截处理WEB请求!
2. Spring MVC 拦截器: 拦截处理Spring MVC的请求流程
3. AOP: 拦截Spring中各个组件之间方法请求.

**声明式事务处理**

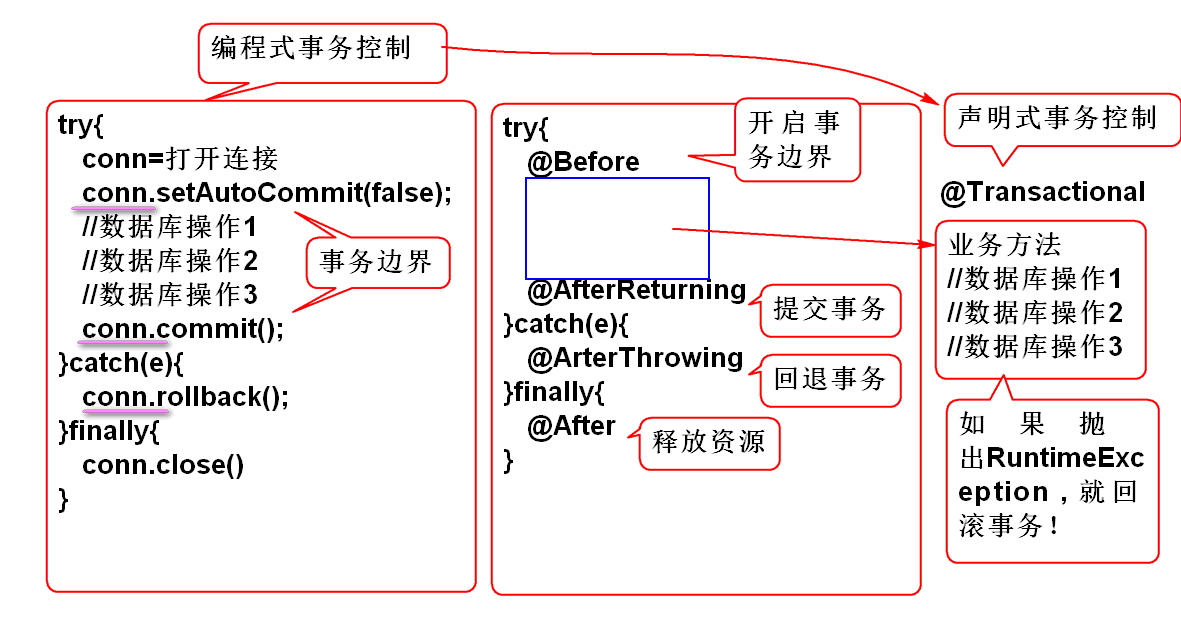
**编程式事务处理**

传统编程事务处理非常繁琐:

try{ conn=打开连接 conn.setAutoCommit(false); //数据库操作1 //数据库操作2 //数据库操作3 conn.commit(); }catch(e){ conn.rollback(); }finally{ conn.close() }

**声明式事务处理**

声明式事务处理, 底层是利用AOP实现的, 只需要简单配置即可使用.



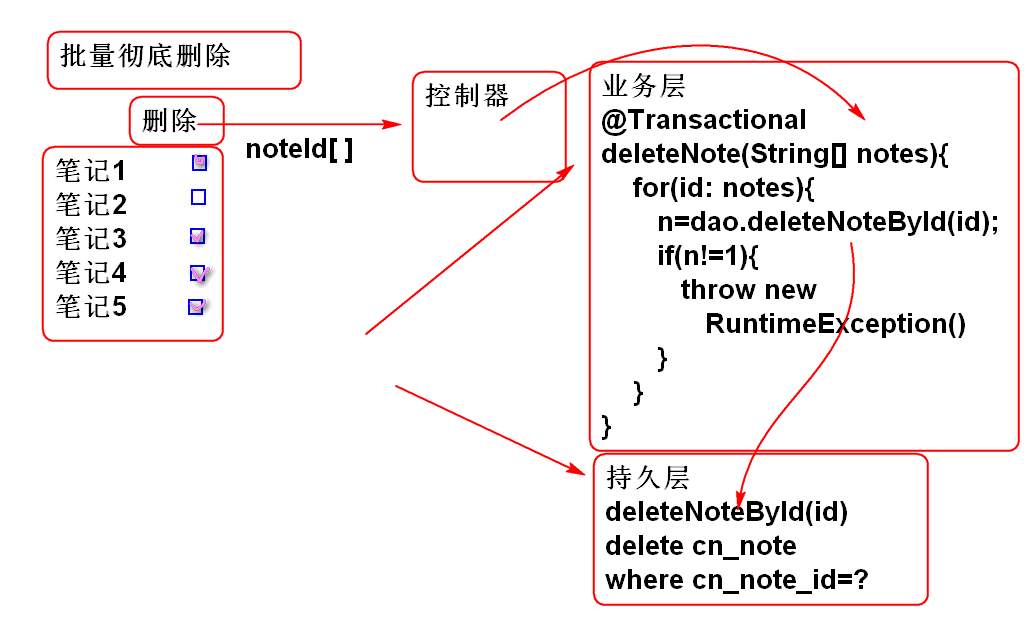
**使用声明式事务处理**

1. 配置事务管理器

<!-- spring-mybatis.xml --> <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"> <property name="dataSource" ref="dataSource"/> </bean> <!-- 设置 注解驱动的事务管理 --> <tx:annotation-driven transaction-manager="txManager"/>

在业务方法上使用 事务注解即可

案例: 批量删除



1. 开发持久层

持久层 NoteDao

int deleteNoteById(String noteId);

SQL NoteMapper.xml

<delete id="deleteNoteById" parameterType="string"> delete from cn\_note where cn\_note\_id=#{noteId} </delete>

1. 业务层

业务层接口方法 NoteService

int deleteNotes(String... noteIds) throws NoteNotFoundException;

实现业务方法 NoteServiceImpl

@Transactional public int deleteNotes(String... noteIds) throws NoteNotFoundException { for(String id: noteIds){ int n=noteDao.deleteNoteById(id); if(n!=1){ throw new NoteNotFoundException("ID错误"); } } return noteIds.length; }

抛出异常 NoteNotFoundException 时候会触发Spring事务回滚操作.

1. 测试 NoteServerTest

@Test public void testDeleteNotes(){ String id1 = "3febebb3-a1b7-45ac-83ba-50cdb41e5fc1"; String id2 = "9187ffd3-4c1e-4768-9f2f-c600e835b823"; String id3 = "ebd65da6-3f90-45f9-b045-782928a5e2c0"; String id4 = "A";//"fed920a0-573c-46c8-ae4e-368397846efd"; int n = service.deleteNotes( id1, id2, id3, id4); //int n = service.deleteNotes( // new String[]{id1, id2, id3, id4}); System.out.println(n); }

在提交的ID有错误时候会出现异常, 并且回滚数据库事务.

**作业**

1. 实现性能测试功能
2. 为业务层增加声明式事务管理

# 云笔记

## Spring 事务管理

### 只读属性

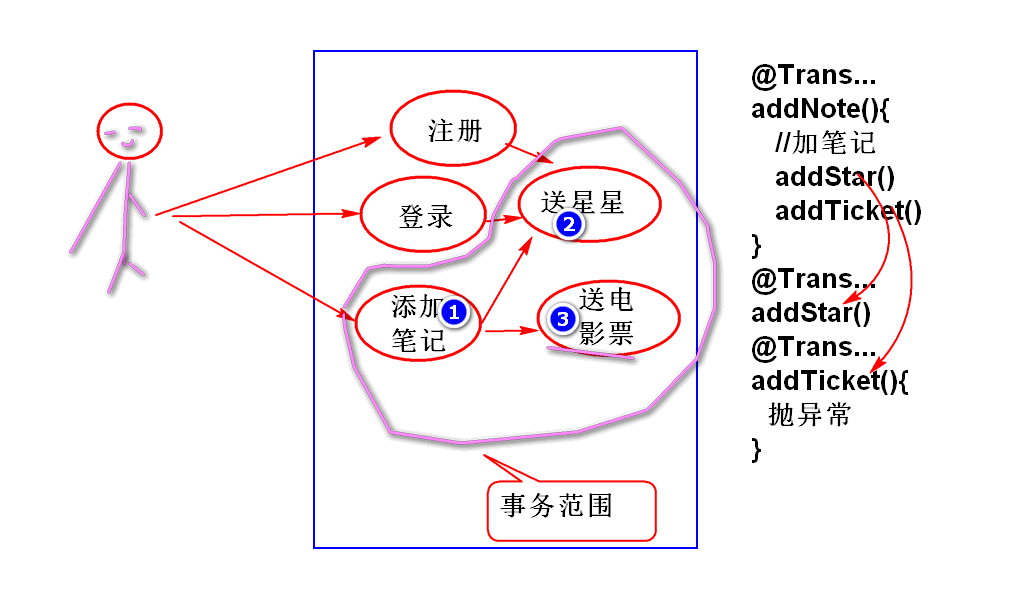
对于单纯读取数据库操作, 可以设置readOnly=true, 可以提高数据的放效率.

@Transactional(readOnly=true) public List<Map<String, Object>> listNotesInTrashBin( String userId) throws UserNotFoundException {

### 事务的传播

业务过程重构时候需要业务方法调用业务方法, 这样就需要一个业务方法的事务传播到另外一个业务方法中, 整合为一个事务.

案例:



步骤:

1. 创建数据表:

create table cn\_star ( cn\_star\_id varchar(50), cn\_user\_id varchar(50), cn\_stars int, primary key(cn\_star\_id) );

1. 声明 StarsDao

public interface StarsDao { Stars findStarsByUserId(String userId); int insertStars(Stars stars); int updateStars(Stars stars); }

1. 声明SQL StarsMapper.xml

<mapper namespace="cn.tedu.note.dao.StarsDao"> <select id="findStarsByUserId" parameterType="string" resultType="cn.tedu.note.entity.Stars"> select cn\_star\_id as id, cn\_user\_id as userId, cn\_stars as stars from cn\_star where cn\_user\_id = #{userId} </select> <insert id="insertStars" parameterType="cn.tedu.note.entity.Stars"> insert into cn\_star( cn\_star\_id, cn\_user\_id, cn\_stars )values( #{id}, #{userId}, #{stars} ) </insert> <update id="updateStars" parameterType="cn.tedu.note.entity.Stars"> update cn\_star set cn\_stars = #{stars} where cn\_user\_id=#{userId} or cn\_star\_id=#{id} </update> </mapper>

1. 声明业务层接口方法 NoteService

boolean addStars(String userId, int stars) throws UserNotFoundException;

1. 实现方法 NoteServiceImpl

@Transactional public boolean addStars(String userId, int stars) throws UserNotFoundException { if(userId==null||userId.trim().isEmpty()){ throw new UserNotFoundException("ID空"); } User user=userDao.findUserById(userId); if(user==null){ throw new UserNotFoundException("木有人"); } //检查是否已经有星了 Stars st=starsDao.findStarsByUserId(userId); if(st==null){//如果没有星星 String id = UUID.randomUUID().toString(); st = new Stars(id, userId, stars); int n = starsDao.insertStars(st); if(n!=1){ throw new RuntimeException("失败"); } }else{//如果有星星,就在现有星星数量上增加 int n = st.getStars()+stars; if(n<0){ // n = 0; throw new RuntimeException("扣分太多!"); } st.setStars(n); n = starsDao.updateStars(st); if(n!=1){ throw new RuntimeException("失败"); } } return true; }

1. 测试: NoteServiceTest

@Test public void testAddStars(){ String userId="03590914-a934-4da9-ba4d-b41799f917d1"; boolean b = service.addStars(userId, 5); System.out.println(b); b = service.addStars(userId, 6); System.out.println(b); }

1. 重构 addNote方法, 实现添加笔记时候送星星功能, 进而实现事务的传播:

@Transactional public Note addNote(String userId, String notebookId, String title) throws UserNotFoundException, NotebookNotFoundException { if(userId==null||userId.trim().isEmpty()){ throw new UserNotFoundException("ID空"); } User user=userDao.findUserById(userId); if(user==null){ throw new UserNotFoundException("木有人"); } if(notebookId==null||notebookId.trim().isEmpty()){ throw new NotebookNotFoundException("ID空"); } int n=notebookDao.countNotebookById(notebookId); if(n!=1){ throw new NotebookNotFoundException("没有笔记本"); } if(title==null || title.trim().isEmpty()){ title="葵花宝典"; } String id = UUID.randomUUID().toString(); String statusId = "1"; String typeId = "1"; String body = ""; long time=System.currentTimeMillis(); Note note = new Note(id, notebookId, userId, statusId, typeId, title, body, time, time); n = noteDao.addNote(note); if(n!=1){ throw new NoteNotFoundException("保存失败"); } //当前的事务, 会传播到 addStart方法中 //整合为一个事务! addStars(userId, 5); return note; }

1. 测试...

### 事务传播(propagation)属性

重点掌握 Propagation.REQUIRED

1. @Transactional(propagation=Propagation.REQUIRED)
   * 需要事务, 如果没有事务创建新事务, 如果当前有事务参与当前事务
2. @Transactional(propagation=Propagation.MANDATORY)
   * 必须有事务, 如果当前没有事务就抛异常
3. @Transactional(propagation=Propagation.NEVER)
   * 绝不, 绝对不能有事务, 如果在事务中调用则抛出异常
4. @Transactional(propagation=Propagation.NESTED)
   * 嵌套, 必须被嵌套到其他事务中
5. @Transactional(propagation=Propagation.NOT\_SUPPORTED)
   * 不支持事务
6. @Transactional(propagation=Propagation.SUPPORTS)
   * 支持事务, 如果没有事务也不会创建新事务
7. @Transactional(propagation=Propagation.REQUIRES\_NEW)
   * 必须是新事务, 如果有当前事务, 挂起当前事务并且开启新事务.

### 事务隔离属性

一共有4种, 一般采用 @Transactional(isolation=Isolation.READ\_COMMITTED) 级别, 是并发性能和安全性折中的选择. 是大多数软件项目采用的隔离级别.

### 声明式事务

就是所有业务方法使用@Transactional, 如果是只读方法, 建议增加readOnly=true

面试题目:

Spring 声明式事务是如何实现的? 答案: 利用AOP技术实现.

## 文件的上载与下载

### HTTP协议中文件是如何下载的

<http://doc.tedu.cn/rfc/rfc2616.txt>

根据HTTP1.1协议, 知道服务器向客户端传输数据如下:

HTTP/1.1 200 OK Content-Type: image/png Content-Length: 130 数据内容

安装这个规则, 可以利用Servlet向客户端发送自定义图片:

1. 声明Servlet

public class ImageServlet extends HttpServlet { private static final long serialVersionUID = 1L; protected void doGet( HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { //发送照片 byte[] png = createPng(); response.setContentType("image/png"); response.setContentLength(png.length); //在消息body中发送消息数据 response.getOutputStream().write(png); } /\*\* \* 创建一个图片, 并且编码为 png 格式, 返回 \* 编码以后的数据 \*/ private byte[] createPng() throws IOException{ BufferedImage img = new BufferedImage(200, 80, BufferedImage.TYPE\_3BYTE\_BGR); //在图片上绘制内容 img.setRGB(100, 40, 0xffffff); //将图片编码为PNG ByteArrayOutputStream out = new ByteArrayOutputStream(); ImageIO.write(img, "png", out); out.close(); byte[] png = out.toByteArray(); return png; } }

1. 配置Servlet

<servlet> <description></description> <display-name>ImageServlet</display-name> <servlet-name>ImageServlet</servlet-name> <servlet-class>cn.tedu.note.web.ImageServlet</servlet-class> </servlet> <servlet-mapping> <servlet-name>ImageServlet</servlet-name> <url-pattern>/demoimg</url-pattern> </servlet-mapping>

1. 测试

http://localhost:8080/note/demoimg

显示生产的图片

1. 在网页中显示图片:

<h1>自定义图片</h1> <p> Servlet 处理自定义图片 <img alt="" src="demoimg"> </p>

利用Spring MVC也可以动态生产图片:

1. 添加控制器方法 UserController

/\* \* @ResponseBody 注解会自动处理控制返回值 \* 1. 如果是JavaBean(数组,集合) 返回JSON \* 2. 如果是byte数字, 则将byte数组直接装入 \* 响应消息的body \*/ //produces="image/png"用于设置 content-type @RequestMapping(value="/image.do", produces="image/png") @ResponseBody public byte[] image() throws Exception{ return createPng(); } /\*\* \* 创建一个图片, 并且编码为 png 格式, 返回 \* 编码以后的数据 \*/ private byte[] createPng() throws IOException{ BufferedImage img = new BufferedImage(200, 80, BufferedImage.TYPE\_3BYTE\_BGR); //在图片上绘制内容 img.setRGB(100, 40, 0xffffff); //将图片编码为PNG ByteArrayOutputStream out = new ByteArrayOutputStream(); ImageIO.write(img, "png", out); out.close(); byte[] png = out.toByteArray(); return png; }

1. 在页面中显示图片:

<p> Spring MVC 处理自定义图片 <img alt="" src="user/image.do"> </p>

根据RFC 2616 的 19.5.1 章节内容, 设置HTTP协议头, 就可以实现图片下载保存功能:

Content-Type: application/octet-stream Content-Disposition: attachment; filename="fname.ext"

1. 声明控制器方法 UserController

@RequestMapping(value="/downloadimg.do", produces="application/octet-stream") @ResponseBody public byte[] downloadimg( HttpServletResponse res) throws IOException{ //参考: 19.5.1 Content-Disposition //Content-Disposition: attachment; filename="fname.ext" res.setHeader("Content-Disposition", "attachment; filename=\"demo.png\""); return createPng(); }

1. 网页上设置下载链接

<h1>下载图片</h1> <a href="user/downloadimg.do">图片</a>

Excel 文件的下载: 与下载图片一样, 只需要更换API生产文件数据就可以实现下载功能了.

1. 导入POI API, 用于处理Excel文件

<dependency> <groupId>org.apache.poi</groupId> <artifactId>poi</artifactId> <version>3.13</version> </dependency>

1. 利用POI生成Excel文件数据:

private byte[] createExcel() throws IOException{ //创建工作簿 HSSFWorkbook workbook=new HSSFWorkbook(); //创建工作表 HSSFSheet sheet= workbook.createSheet("Demo"); //在工作表中创建数据行 HSSFRow row = sheet.createRow(0); //创建行中的格子 HSSFCell cell = row.createCell(0); cell.setCellValue("Hello World!"); //将Excel文件保存为 byte 数组 ByteArrayOutputStream out = new ByteArrayOutputStream(); workbook.write(out); out.close(); return out.toByteArray(); }

1. 利用Spring MVC控制器下载Excel:

@RequestMapping(value="/excel.do", produces="application/octet-stream") @ResponseBody public byte[] excel( HttpServletResponse res) throws IOException{ //参考: 19.5.1 Content-Disposition res.setHeader("Content-Disposition", "attachment; filename=\"demo.xls\""); return createExcel(); }

1. 在网页上连接下载控制器:

<h1>下载Excel</h1> <a href="user/excel.do">Excel 文件</a>

## 上载文件

上载文件协议 <http://doc.tedu.cn/rfc-cn/RFC1867.txt>

文件上载:

### 1. 客户端:

<h1>文件上载</h1> <FORM ENCTYPE="multipart/form-data" ACTION="user/upload.do" METHOD="POST"> File to process:<br> <INPUT NAME="userfile1" TYPE="file"><br> <INPUT NAME="userfile2" TYPE="file"><br> <INPUT TYPE="submit" VALUE="Send File"> </FORM>

### 2. 服务器端:

1. 导入 上载组件:

<dependency> <groupId>commons-fileupload</groupId> <artifactId>commons-fileupload</artifactId> <version>1.3.2</version> </dependency>

1. 配置上载解析器 spring-mvc.xml

<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"> <!-- one of the properties available; the maximum file size in bytes --> <property name="maxUploadSize" value="10000000"/> <property name="defaultEncoding" value="UTF-8"/> </bean>

1. 开发控制器:

@RequestMapping("/upload.do") @ResponseBody public JsonResult upload( MultipartFile userfile1, MultipartFile userfile2) throws Exception{ //Spring MVC 中可以利用 MultipartFile //接收 上载的文件! 文件中的一切数据 //都可以从 MultipartFile 对象中找到 //获取上再是原始文件名 String file1 = userfile1.getOriginalFilename(); String file2 = userfile2.getOriginalFilename(); System.out.println(file1); System.out.println(file2); //保存文件的3种方法: //1. transferTo(目标文件) // 将文件直接保存到目标文件, 可以处理大文件 //2. userfile1.getBytes() 获取文件的全部数据 // 将文件全部读取到内存, 适合处理小文件!! //3. userfile1.getInputStream() // 获取上载文件的流, 适合处理大文件 //保存的目标文件夹: /home/soft01/demo File dir = new File("D:/demo"); dir.mkdir(); File f1 = new File(dir, file1); File f2 = new File(dir, file2); //第一种保存文件 //userfile1.transferTo(f1); //userfile2.transferTo(f2); //第三种 利用流复制数据 InputStream in1 = userfile1.getInputStream(); FileOutputStream out1 = new FileOutputStream(f1); int b; while((b=in1.read())!=-1){ out1.write(b); } in1.close(); out1.close(); InputStream in2 = userfile2.getInputStream(); FileOutputStream out2= new FileOutputStream(f2); byte[] buf= new byte[8\*1024]; int n; while((n=in2.read(buf))!=-1){ out2.write(buf, 0, n); } in2.close(); out2.close(); return new JsonResult(true); }

1. 测试
2. 实现笔记列表导出到Excel功能
3. 实现文件上载功能

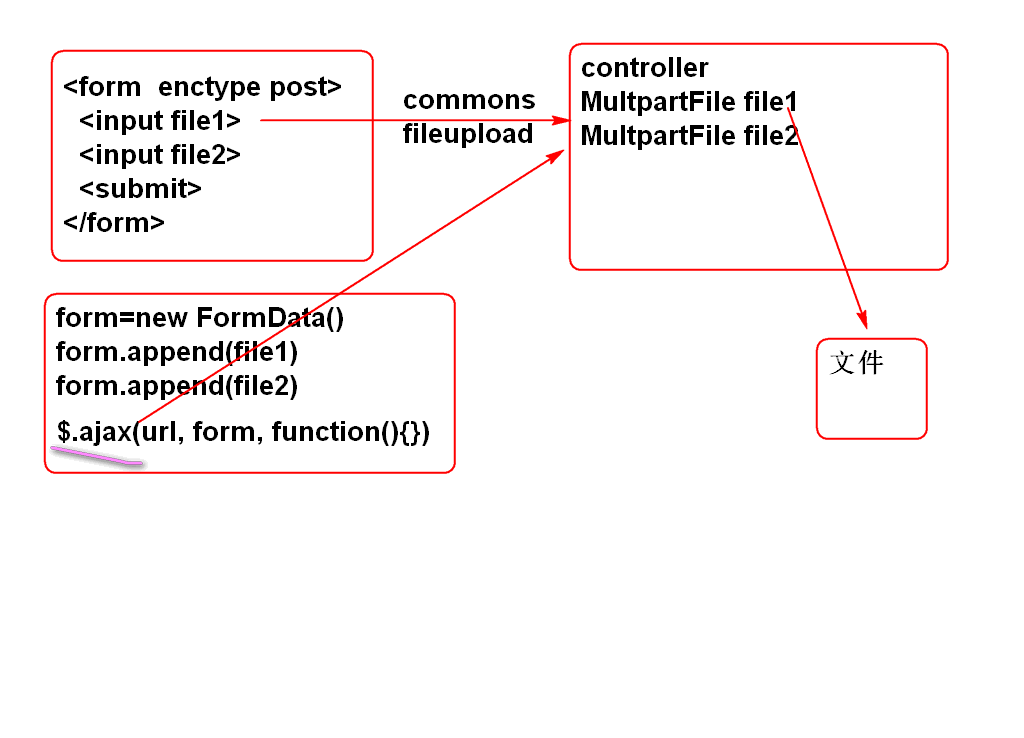
# 云笔记

## 文件上载

### Ajax 文件上载

利用 FormData 对象和 Spring MVC 配合可以实现Ajax文件上载功能:

原理:



案例:

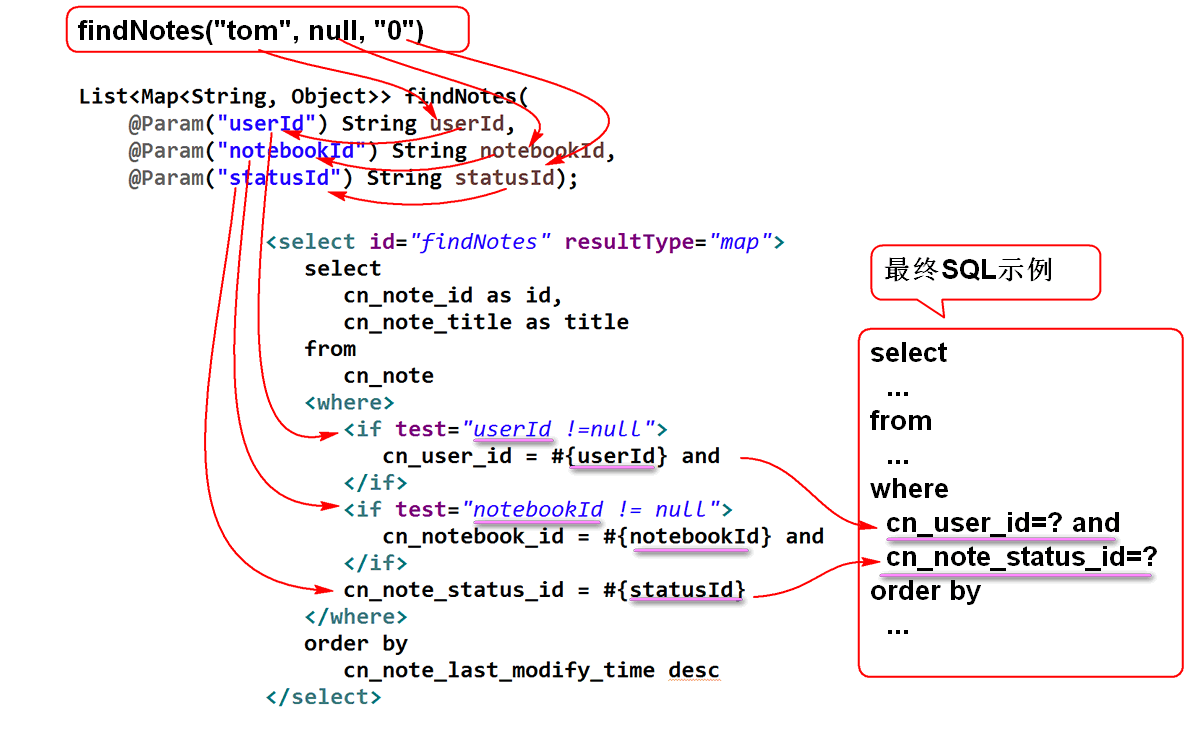
<!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <title>Insert title here</title> <script type="text/javascript" src="scripts/jquery.min.js"></script> <script type="text/javascript"> $(function(){ $('#upload').click(ajaxUpload); }); function ajaxUpload(){ var file1 = $('#file1')[0].files[0]; var file2 = $('#file2')[0].files[0]; //创建内存中的表单对象 var form = new FormData(); //添加向服务器传输的数据 form.append('userfile1', file1); form.append('userfile2', file2); $.ajax({ url:'user/upload.do', data: form, type: 'POST', dataType: 'json', contentType: false, processData: false, success: function(obj){ if(obj.state==0){ $('#result').html("成功!"); } } }); } </script> </head> <body> <h1>Ajax 文件上载</h1> <input type="file" id="file1"> <br> <input type="file" id="file2"> <br> <input type="button" id="upload" value="上载" > <div id="result"></div> </body> </html>

提示: 服务端重用上节课的案例.

## 多参数查询

在MyBatis中, 利用 @Param 注解, 可以实现多参数查询

原理:



案例:

1. NoteDao接口

List<Map<String, Object>> findNotes( @Param("userId") String userId, @Param("notebookId") String notebookId, @Param("statusId") String statusId);

1. SQL 语句: NoteMapper.xml

<select id="findNotes" resultType="map"> select cn\_note\_id as id, cn\_note\_title as title from cn\_note <where> <if test="userId !=null"> cn\_user\_id = #{userId} and </if> <if test="notebookId != null"> cn\_notebook\_id = #{notebookId} and </if> cn\_note\_status\_id = #{statusId} </where> order by cn\_note\_last\_modify\_time desc </select>

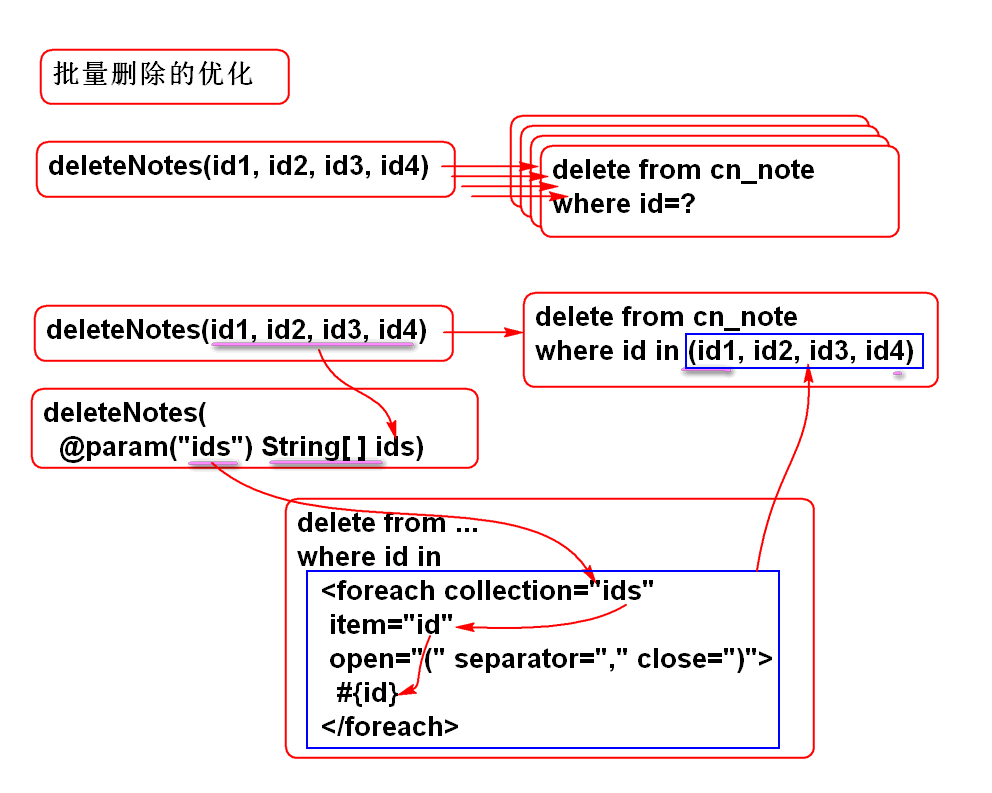
## 动态SQL

MyBatis 提供了灵活的动态SQL功能, 只需要使用映射文件的标签就可以到达灵活的拼接SQL语句的功能:

### <foreach> 标签的使用

批量删除笔记功能:

原理:



1. 持久层方法 NoteDao

int deleteNotes( @Param("ids") String... ids);

1. 定义SQL NoteMapper.xml

<delete id="deleteNotes"> delete from cn\_note where cn\_note\_id in <foreach collection="ids" item="id" open="(" separator="," close=")"> #{id} </foreach> </delete>

1. 测试 NoteDaoTest

@Test public void testDeleteNotes(){ String id1="07305c91-d9fa-420d-af09-c3ff209608ff"; String id2="5565bda4-ddee-4f87-844e-2ba83aa4925f"; String id3="1ec185d6-554a-481b-b322-b562485bb8e8"; int n = dao.deleteNotes(id1, id2, id3); System.out.println(n); }

### trim 和 set 的使用

set标签用于生成 set 子句

trim 标签用于去除多余的 逗号, and 或 or

<update id="updateNote" parameterType="cn.tedu.note.entity.Note"> update cn\_note <set> <trim suffixOverrides=","> <if test="notebookId!=null"> cn\_notebook\_id=#{notebookId}, </if> <if test="userId!=null"> cn\_user\_id=#{userId}, </if> <if test="statusId!=null"> cn\_note\_status\_id=#{statusId}, </if> <if test="typeId!=null"> cn\_note\_type\_id=#{typeId}, </if> <if test="title!=null"> cn\_note\_title=#{title}, </if> <if test="body!=null"> cn\_note\_body=#{body}, </if> <if test="lastModifyTime != null"> cn\_note\_last\_modify\_time=#{lastModifyTime} </if> </trim> </set> where cn\_note\_id=#{id} </update>

### where 和 trim 标签的使用

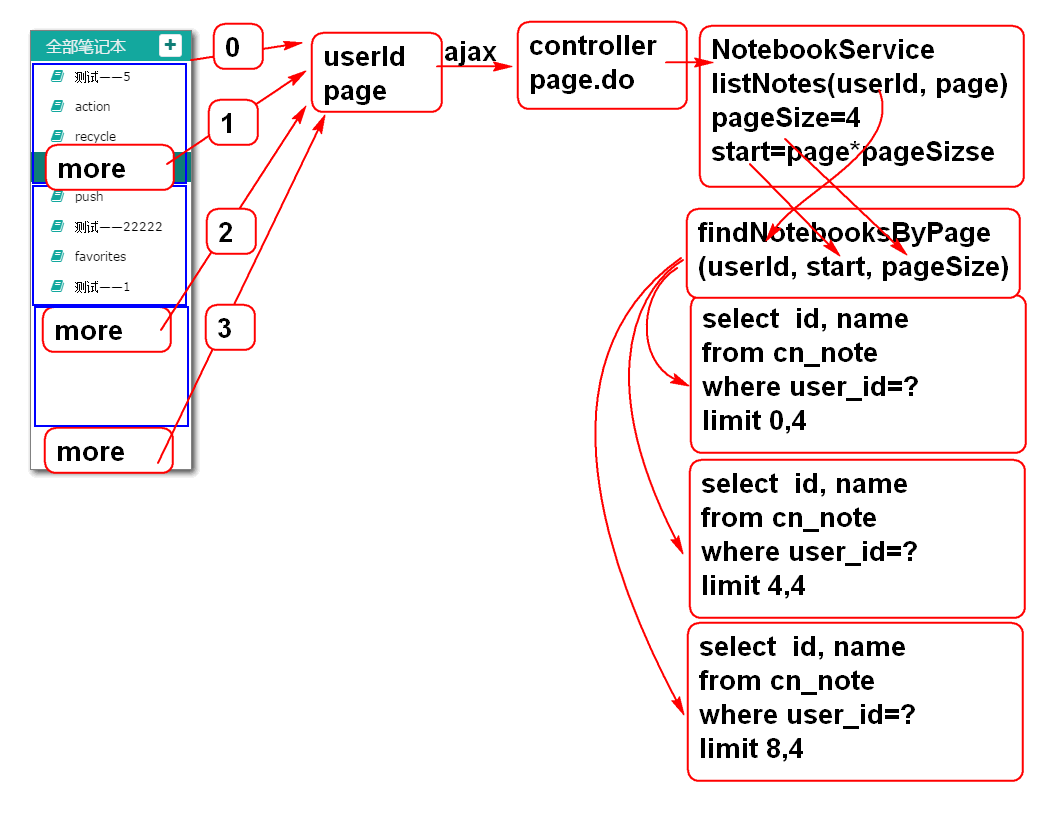
where标签用于生成 where 子句

trim 标签用于去除多余的 逗号, and 或 or

<select id="findNotes" resultType="map"> select cn\_note\_id as id, cn\_note\_title as title from cn\_note <where> <trim suffixOverrides="and"> <if test="userId !=null"> cn\_user\_id = #{userId} and </if> <if test="notebookId != null"> cn\_notebook\_id = #{notebookId} and </if> <if test="statusId!=null"> cn\_note\_status\_id = #{statusId} </if> </trim> </where> order by cn\_note\_last\_modify\_time desc </select>

## 翻页功能

翻页公用与其他业务功能的设计是一样的:



实现步骤

### 服务端

1. 持久层接口 NotebookDao

List<Map<String, Object>> findNotebooksByPage( @Param("userId") String userId, @Param("start") int start, @Param("pageSize") int pageSize, @Param("table") String table);

1. SQL 定义 NotebookMapper.xml

<select id="findNotebooksByPage" resultType="map"> select cn\_notebook\_id as id, cn\_notebook\_name as name from ${table} where cn\_user\_id = #{userId} order by cn\_notebook\_createtime desc limit #{start}, #{pageSize} </select>

${table} 可以替换SQL语句的表名, 实现动态SQL拼接查询, 尽量不要用于参数部分, 否则可能有SQL注入风险.

limit 子句是MySQL专用的, 其他数据库不支持.

1. 业务层接口 NotebookService

List<Map<String, Object>>listNotebooks( String userId, Integer page) throws UserNotFoundException;

1. 实现业务层方法 NotebookServiceImpl

@Value("#{jdbc.pageSize}") private int pageSize; public List<Map<String, Object>> listNotebooks( String userId, Integer page) throws UserNotFoundException { if(userId==null || userId.trim().isEmpty()){ throw new UserNotFoundException("ID不能空"); } User user = userDao.findUserById(userId); if(user==null){ throw new UserNotFoundException("用户不存在"); } if(page==null){ page = 0; } int start = page \* pageSize; String table = "cn\_notebook"; return notebookDao.findNotebooksByPage( userId, start, pageSize, table); }

变量table用于动态设置SQL查询的表名

1. 在配置文件中证据页面大小参数 jdbc.properties

pageSize=4

1. 测试 NotebookServiceTest

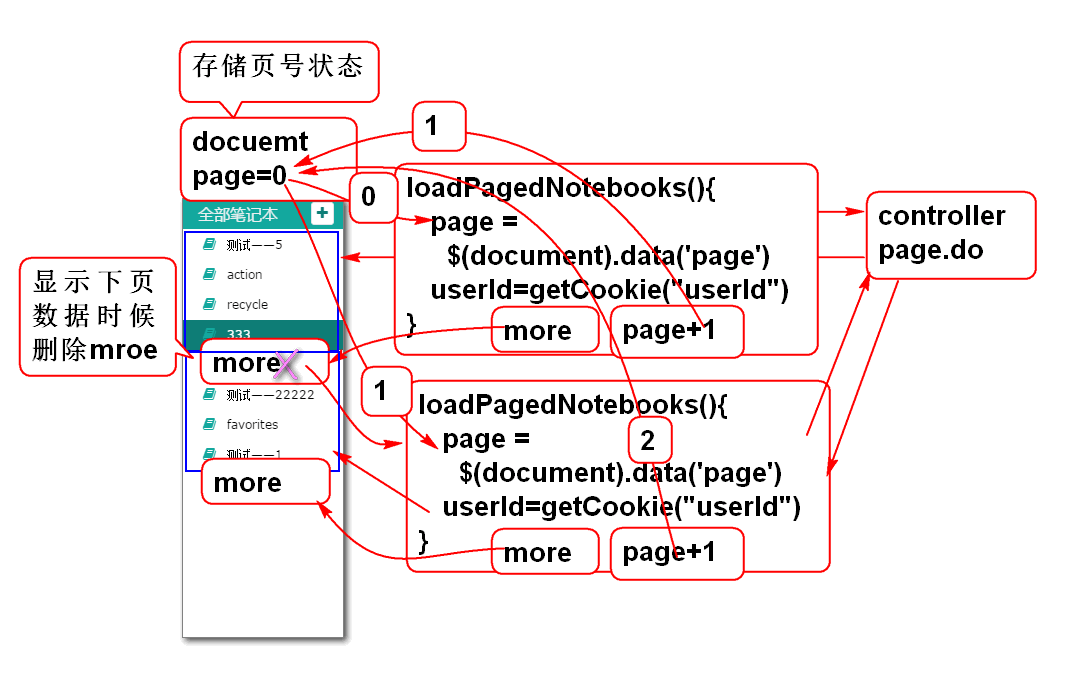
@Test public void testAddStars(){ String userId="03590914-a934-4da9-ba4d-b41799f917d1"; boolean b = service.addStars(userId, 5); System.out.println(b); b = service.addStars(userId, 6); System.out.println(b); }

1. 控制器 NotebookController

@RequestMapping("/page.do") @ResponseBody public JsonResult page(String userId, Integer page){ List<Map<String, Object>> list= notebookService.listNotebooks( userId, page); return new JsonResult(list); }

### 客户端

原理:



1. 重构ready方法:

//在document对象中存翻页页号状态 $(document).data('page',0); //加载第一页数据 loadPagedNotebooks(); //点击more时候加载下一页数据 $('#notebook-list').on( 'click','.more', loadPagedNotebooks);

1. 添加事件处理方法

function loadPagedNotebooks(){ var page = $(document).data('page'); var userId = getCookie('userId'); //从服务器拉去数据 var url = 'notebook/page.do'; var data = {userId: userId, page:page}; $.getJSON(url, data, function(result){ if(result.state==SUCCESS){ var notebooks = result.data; showPagedNotebooks(notebooks, page); $(document).data('page', page+1); }else{ alert(result.message); } }); }

1. 添加 分页显示笔记本列表方法:

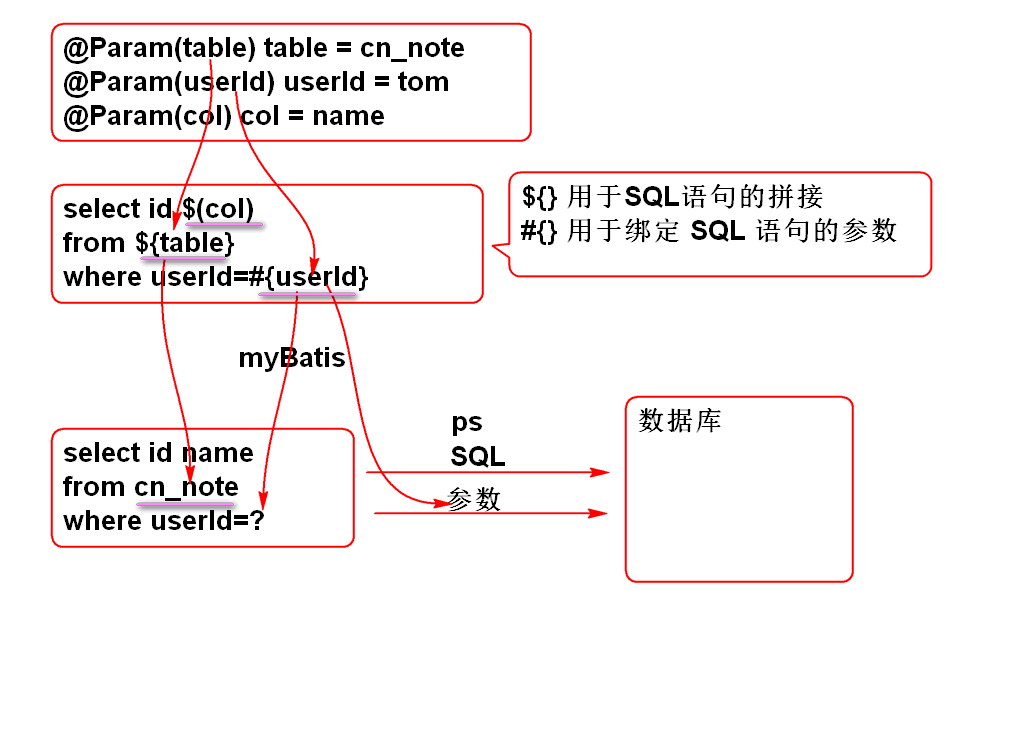
function showPagedNotebooks(notebooks, page){ var ul = $('#notebook-list ul'); if(page==0){//第一页时候清空 ul中的li ul.empty(); }else{//不是第一页, 只删除.more元素 ul.find('.more').remove(); } for(var i=0; i<notebooks.length; i++){ var notebook=notebooks[i]; var li = notebookTemplate.replace( '[name]', notebook.name); li = $(li); li.data('notebookId', notebook.id); ul.append(li); } if(notebooks.length!=0){ ul.append(moreTemplate); } } var moreTemplate = '<li class="online more">'+ '<a><i class="fa fa-plus" title="online" '+ 'rel="tooltip-bottom"></i> 加载更多...</a>'+ '</li>';

1. 测试 ...

## ${} 与 #{}

在MyBatis中

* ${} 用于进行动态SQL拼接, 仅仅是拼接SQL语句, 有SQL注入风险, 不要将用户输入的 参数 利用 ${} 进行拼接
* #{} 用于绑定SQL语句的参数, 也就是生成参数 ?, 然后利用PS执行带参数的SQL



## MyBatis返回自增类型的ID

MyBatis可以返回自动增加的ID:

1. SQL:

create table p\_person( id int not null AUTO\_INCREMENT, name varchar(100), primary key(id) ); insert into p\_person (name) values ('李老师'); insert into p\_person (id, name) values (null,'李老师');

id AUTO\_INCREMENT 就是MySQL的自增类型

1. 实体:

public class Person implements Serializable { private static final long serialVersionUID = -2365398342302306276L; private Integer id; private String name; public Person() { } public Person(Integer id, String name) { super(); this.id = id; this.name = name; } public Integer getId() { return id; } public void setId(Integer id) { this.id = id; } public String getName() { return name; } public void setName(String name) { this.name = name; } @Override public String toString() { return "Person [id=" + id + ", name=" + name + "]"; } @Override public int hashCode() { final int prime = 31; int result = 1; result = prime \* result + ((id == null) ? 0 : id.hashCode()); return result; } @Override public boolean equals(Object obj) { if (this == obj) return true; if (obj == null) return false; if (getClass() != obj.getClass()) return false; Person other = (Person) obj; if (id == null) { if (other.id != null) return false; } else if (!id.equals(other.id)) return false; return true; } }

1. PersonMapper.xml

<mapper namespace="cn.tedu.note.dao.PersonDao"> <!-- 在插入时候读取自动生成的ID 打开useGeneratedKeys="true" 可以读取自增ID 设置 keyProperty="id" 告诉MyBatis将读取结果 赋值给 参数 person对象的 id 属性 --> <insert id="addPerson" parameterType="cn.tedu.note.entity.Person" useGeneratedKeys="true" keyProperty="id"> insert into p\_person( id, name )values( null, #{name} ) </insert> </mapper>

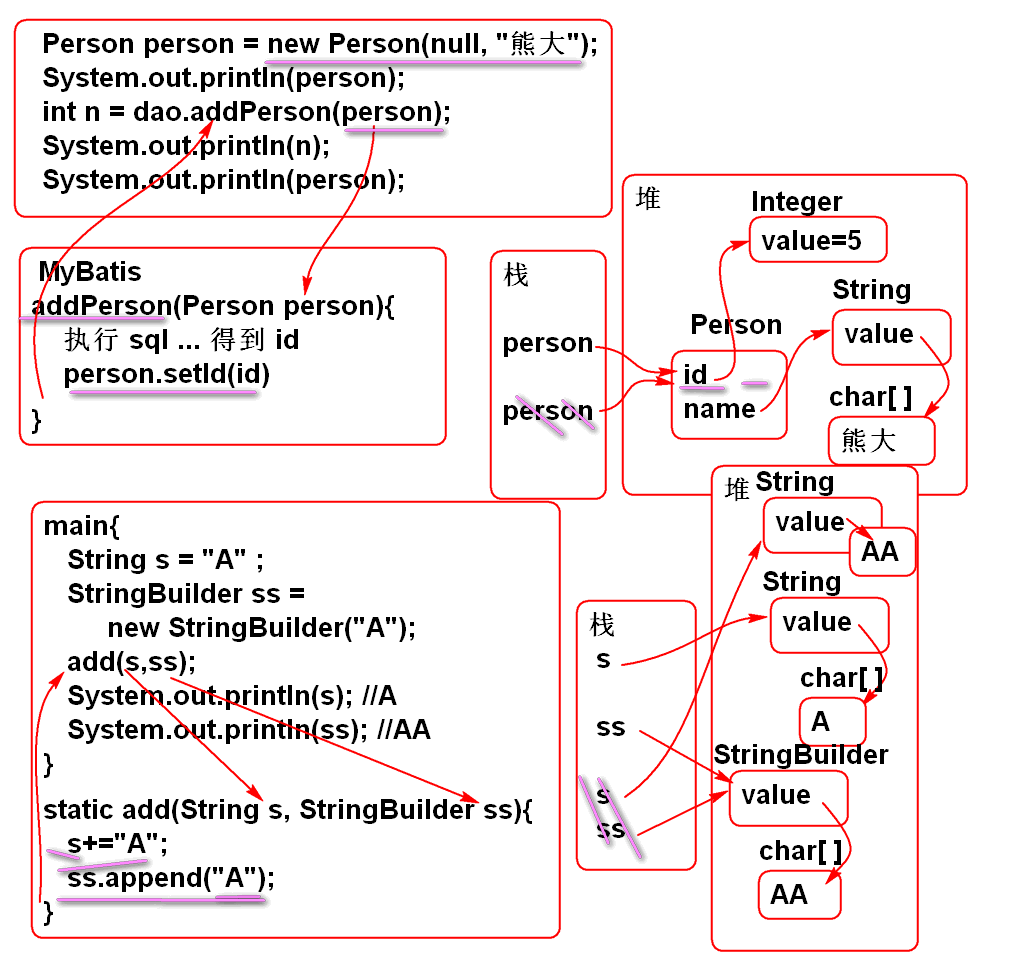
1. 测试:

public class PersonDaoTest extends BaseTest {

PersonDao dao; @Before public void initDao(){ dao = ctx.getBean( "personDao", PersonDao.class); } @Test public void testAddPerson(){ Person person = new Person(null, "熊大"); System.out.println(person); int n = dao.addPerson(person); System.out.println(n); System.out.println(person); }

}

ID赋值原理:



练习:

public class Demo01 { public static void main(String[] args) { int n = 2; final int[] ary = {2}; test(n, ary); System.out.println(n); //2 System.out.println(ary[0]); //3 } public static void test( Integer i, int[] ary){ i = i++; ary[0]++; } }

## 作业

1. 完成云笔记列表翻页功能
2. 实现自增ID功能
3. 实现Ajax文件上载功能