**1. Spring 拦截器**

<http://blog.csdn.net/tonytfjing/article/details/39207551>

方法：

第1步：继承HandlerInterceptorAdapter类

第2步：重写preHandler方法和postHandler方法和afterCompletion方法。

**项目中使用拦截器，配合Java注解，实现用户行为跟踪**

**postHandle方法里面关键代码：**

1. **if** (handler **instanceof** HandlerMethod) {
2. HandlerMethod method = (HandlerMethod) handler;
3. Desc desc = method.getMethodAnnotation(Desc.**class**);   //获取方法上的注解
4. **if**(desc!= **null** && StringUtils.isNotEmpty(desc.value()) && StringUtil.isNotEmpty(desc.descType())) {
5. uriDesc = desc.value();
6. descType = desc.descType();
7. }
8. }
9. record(request, uri, uriDesc, descType, source);          //用户行为记录到数据库（访问过哪些url，行为描述等）

**2. Spring 过滤器**

## 2.1 Spring Web过滤器

<http://qq-22530757.iteye.com/blog/2177513>

<http://blog.csdn.net/liangxw1/article/details/51095484>

## 2.2 自定义过滤器

<https://www.cnblogs.com/lr393993507/p/5543025.html>

**创建一个Filter 只需两个步骤:**

(1)创建Filter 处理类

(2)在web.xml 文件中配置Filter

创建Filter 必须实现javax.servlet.Filter 接口，在该接口中定义了三个方法。   
• void init(FilterConfig config): 用于完成Filter 的初始化。   
• void destroy(): 用于Filter 销毁前，完成某些资源的回收。   
• void doFilter(ServletRequest request, ServletResponse response,FilterChain chain): 实现过滤功能，该方法就是对每个请求及响应增加的额外处理。

过滤器Filter也具有生命周期：init()->doFilter()->destroy()，由部署文件中的filter元素驱动。

下面是一个自定义过滤器的demo

1、创建Filter处理类

1. **public** **class** XssFilter **implements** Filter {
3. @Override
4. **public** **void** init(FilterConfig config) **throws** ServletException {
5. }
7. **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
8. **throws** IOException, ServletException {
9. XssHttpServletRequestWrapper xssRequest = **new** XssHttpServletRequestWrapper(
10. (HttpServletRequest) request);
11. chain.doFilter(xssRequest, response);
12. }
14. @Override
15. **public** **void** destroy() {
16. }
18. }

2、配置web.xml

1. <filter>
2. <filter-name>xssFilter</filter-name>
3. <filter-**class**>io.renren.common.xss.XssFilter</filter-**class**>
4. </filter>
6. <filter-mapping>
7. <filter-name>xssFilter</filter-name>
8. <url-pattern>/\*</url-pattern>
9. </filter-mapping>

**3. Spring Aop**

Apo各个角色介绍：

* 连接点(join point)  
  个人理解，可以简单的认为是方法的执行（项目中的方法都可以认为是连接点）
* 切点(point cut)  
  使用aop的目的，是减少重复代码，将一些重复调用的代码，借用aop，封装到一个切面来实现（比如日志记录）。那么，不是任何地方，我们都需要进行日志记录。切点的作用，就是确定范围，将需要使用aop的地方（连接点）通过切点筛选出来。
* 增强(advice)  
  你需要做的事情，比如日志记录、权限控制，可以放到这里
* 切面(Aspect)  
  aspect 由 pointcount 和 advice 组成, 它既包含了横切逻辑的定义, 也包括了连接点的定义，其两个重要的工作是：  
  1、如何通过切点（pointcut）定位到特定的连接点（joinpoint）上  
  2、如何在增强（advice）中编写切面代码（比如日志记录）

看到一篇文章，感觉写的很好，他举了一个例子，通俗解释了以上概念，内容如下：

首先我们知道, 在 Spring AOP 中 join point 指代的是所有方法的执行点, 而 point cut 是一个描述信息, 它修饰的是 join point, 通过 point cut, 我们就可以确定哪些 join point 可以被织入 Advice. 对应到我们在上面举的例子, 我们可以做一个简单的类比, join point 就相当于爪哇的小县城里的百姓, point cut 就相当于老王所做的指控, 即凶手是个男性, 身高约七尺五寸, 而 advice 则是施加在符合老王所描述的嫌疑人的动作: 抓过来审问.  
为什么可以这样类比呢?

* join point --> 爪哇的小县城里的百姓: 因为根据定义, join point 是所有可能被织入advice的候选的点, 在Spring AOP中, 则可以认为所有方法执行点都是join point. 而在我们上面的例子中, 命案发生在小县城中, 按理说在此县城中的所有人都有可能是嫌疑人.
* point cut --> 男性, 身高约七尺五寸: 我们知道, 所有的方法(joint point) 都可以织入 advice, 但是我们并不希望在所有方法上都织入 advice, 而 pointcut 的作用就是提供一组规则来匹配joinpoint, 给满足规则的 joinpoint 添加 advice. 同理, 对于县令来说, 他再昏庸, 也知道不能把县城中的所有百姓都抓起来审问, 而是根据凶手是个男性, 身高约七尺五寸, 把符合条件的人抓起来. 在这里 凶手是个男性, 身高约七尺五寸 就是一个修饰谓语, 它限定了凶手的范围, 满足此修饰规则的百姓都是嫌疑人, 都需要抓起来审问.
* advice --> 抓过来审问, advice 是一个动作, 即一段 Java 代码, 这段 Java 代码是作用于 point cut 所限定的那些 join point 上的. 同理, 对比到我们的例子中, 抓过来审问 这个动作就是对作用于那些满足 男性, 身高约七尺五寸 的爪哇的小县城里的百姓.
* aspect: aspect 是 point cut 与 advice 的组合, 因此在这里我们就可以类比: "根据老王的线索, 凡是发现有身高七尺五寸的男性, 都要抓过来审问" 这一整个动作可以被认为是一个 aspect.

**实现Aop的两种方式：**

xml配置方式 ：<http://blog.csdn.net/xiaoxian8023/article/details/17258933>

AspectJ注解方式：<http://blog.csdn.net/xiaoxian8023/article/details/17285809>

**4. 缓存（Redis）**

**5. 全局错误捕获**

<http://blog.csdn.net/eson_15/article/details/51731567>

下面是一个全局捕获异常的demo：继承HandlerExceptionResolver接口，重写resolveException方法。

1. **public** **class** RRExceptionHandler **implements** HandlerExceptionResolver {
2. **private** Logger logger = LoggerFactory.getLogger(getClass());
4. @Autowired
5. **private** EmailCacheHandler emailCacheHandler;
7. @Override
8. **public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object o, Exception e) {
9. String uri = request.getRequestURI();
10. logger.error("ajax request error, uri : " + uri + ", message : " + e.getMessage());
11. //打印异常栈信息
12. e.printStackTrace();
13. /\*\*\*\*\*\*\*\*写入redis队列\*\*\*\*\*\*\*\*\*\*/
14. emailCacheHandler.save(e);   //加入到redis，以便后期处理
15. **if** (StringUtils.isEmpty(request.getHeader("X-Requested-With"))) {
16. **try**{
17. response.getWriter().print(e.getMessage());
18. **return** **new** ModelAndView();
19. }
20. **catch** (Exception ex) {
21. logger.error("RRExceptionHandler====异常处理失败", e);
22. }
23. }
25. R r = **new** R();
26. response.setContentType("application/json; charset=utf-8");
27. response.setCharacterEncoding("UTF-8");
28. **if** (e **instanceof** RRException) {
29. r.put("code", ((RRException) e).getCode());
30. r.put("message", ((RRException) e).getMessage());
31. } **else** **if** (e **instanceof** AuthenticationException) {
32. r = R.error("没有权限，请联系管理员授权");
33. }  **else** {
34. r = R.error();
35. }
37. logger.error(e.getMessage(), e);
38. **try** {
39. String json = JacksonUtil.toJson(r);
40. response.getWriter().print(json);
41. } **catch** (Exception ex) {
42. logger.error("RRExceptionHandler====异常处理失败", e);
43. }
45. **return** **new** ModelAndView();
46. }
47. }

**6. 发邮件**

<https://www.cnblogs.com/hehaiyang/p/4279201.html>

Spring的邮件发送的核心是MailSender接口，在Spring3.0中提供了一个实现类JavaMailSenderImpl，这个类是发送邮件的核心类。可以通过在配置文件中配置使用，也可以自己硬编码到代码中

**7. 发短信**

**8. 定时任务**

**9. Spring 实现多数据库及读写分离方法**

**10. Druid连接池实现web监控**

第1步：pom文件

1. **<dependency>**
2. **<groupId>**com.alibaba**</groupId>**
3. **<artifactId>**druid**</artifactId>**
4. **<version>**${druid-version}**</version>**
5. **</dependency>**

第2步：打开Druid的监控统计功能：

Druid的监控统计功能是通过filter-chain扩展实现，如果你要打开监控统计功能，需要配置StatFilter

1. <!—自己定制的StatLogger、可以将druid记录到文件和日志打印等操作--**>**
2. **<bean** id="statLogger" class="com.lizhihui.druid.StatLogger"**></bean>**
3. **<bean** id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
4. init-method="init" destroy-method="close"**>**
5. **<property** name="url"**>**
6. **<value>**${jdbc.url}**</value>**
7. **</property>**
8. **<property** name="username"**>**
9. **<value>**${jdbc.username}**</value>**
10. **</property>**
11. **<property** name="password"**>**
12. **<value>**${jdbc.password}**</value>**
13. **</property>**
14. **<property** name="initialSize"**>**
15. **<value>**${jdbc.initialSize}**</value>**
16. **</property>**
17. **<property** name="maxActive"**>**
18. **<value>**${jdbc.maxActive}**</value>**
19. **</property>**
20. <!—配置多久打印日志一次，如果配置了statLogger ，则一定要配置此项，单位为毫秒--**>**
21. **<property** name="timeBetweenLogStatsMillis" value="30000" **/>**
22. <!—配置自定义Logger，打印数据库各种信息--**>**
23. **<property** name=" statLogger" ref=" statLogger" **>**
24. <!—关键：  监控统计拦截的filters --**>**
25. **<property** name="proxyFilters"**>**
26. **<list>**
27. **<ref** bean="stat-filter" **/>**
28. **<ref** bean="wall-filter" **/>**
29. **</list>**
30. **</property>**
31. **</bean>**
33. **<bean** id="stat-filter" class="com.alibaba.druid.filter.stat.StatFilter"**>**
34. **<property** name="slowSqlMillis" value="1000" **/>**
35. **<property** name="logSlowSql" value="true" **/>**  <!-- 慢SQL记录-->
36. **<property** name="mergeSql" value="true" **/>**
37. **</bean>**
39. **<bean** id="wall-filter" class="com.alibaba.druid.wall.WallFilter"**>**
40. **<property** name="dbType" value="mysql" **/>**
41. **<property** name="config" ref="wall-filter-config" **/>**
42. **</bean>**
44. **<bean** id="wall-filter-config" class="com.alibaba.druid.wall.WallConfig"**>**
45. **<property** name="multiStatementAllow" value="true" **/>**
46. **</bean>**
48. <!-- 配置druid监控spring jdbc -->
49. **<bean** id="druid-stat-interceptor" class="com.alibaba.druid.support.spring.stat.DruidStatInterceptor"**></bean>**
50. **<bean** id="druid-stat-pointcut" class="org.springframework.aop.support.JdkRegexpMethodPointcut" scope="prototype"**>**
51. **<property** name="patterns"**>**
52. **<list>**
53. **<value>**io.renren.modules.\*.controller.\***</value>**
54. **<value>**io.renren.modules.\*.service.\***</value>**
55. **</list>**
56. **</property>**
57. **</bean>**
58. **<aop:config>**
59. **<aop:advisor** advice-ref="druid-stat-interceptor" pointcut-ref="druid-stat-pointcut"**/>**
60. **</aop:config>**
62. **<bean** id="transactionManager"
63. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
64. **<property** name="dataSource"**>**
65. **<ref** bean="dataSource" **/>**
66. **</property>**
67. **</bean>**
69. **<bean** id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>**
70. **<property** name="configLocation" value="classpath:mybatis.xml" **/>**
71. **<property** name="mapperLocations" value="classpath:mapper/\*\*/\*.xml" **/>**
72. **<property** name="dataSource" ref="dataSource" **/>**
73. **</bean>**
75. **<bean** class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>**
76. **<property** name="basePackage" value="io.renren.modules.\*.dao" **/>**
77. **</bean>**

第3步：在web.xml文件后，添加

1. **<servlet>**
2. **<servlet-name>**DruidStatView**</servlet-name>**
3. **<servlet-class>**com.alibaba.druid.support.http.StatViewServlet**</servlet-class>**
4. **</servlet>**
6. **<servlet-mapping>**
7. **<servlet-name>**DruidStatView**</servlet-name>**
8. **<url-pattern>**/druid/\***</url-pattern>**
9. **</servlet-mapping>**

之后，就可以打开 <http://10.206.2.248:8082/druid/sql.html> 看到数据库各种信息了

此处还可以配置登录账户和密码，如果不配置，则无账户密码登录监控页面，如下：

1. **<servlet>**
2. **<servlet-name>**DruidStatView**</servlet-name>**
3. **<servlet-class>**com.alibaba.druid.support.http.StatViewServlet**</servlet-class>**
4. <!—
5. deny优先于allow，如果在deny列表中，就算在allow列表中，也会被拒绝。
6. 如果allow没有配置或者为空，则允许所有访问
7. --**>**
8. **<init-param>**
9. **<param-name>**allow**</param-name>**
10. **<param-value>**128.242.127.1/24,127.0.0.1**</param-value>**
11. **</init-param>**
12. **<init-param>**
13. **<param-name>**deny**</param-name>**
14. **<param-value>**192.168.1.118**</param-value>**
15. **</init-param>**
16. <!-- 在StatViewSerlvet输出的html页面中，有一个功能是Reset All，执行这个操作之后，会导致所有计数器清零，重 新计数 -->
17. **<init-param>**
18. **<param-name>**resetEnable**</param-name>**
19. **<param-value>**false**</param-value>**
20. **</init-param>**
22. **<init-param>**
23. **<param-name>**loginUsername**</param-name>**
24. **<param-value>**druid**</param-value>**
25. **</init-param>**
26. **<init-param>**
27. **<param-name>**loginPassword**</param-name>**
28. **<param-value>**druid**</param-value>**
29. **</init-param>**
30. **</servlet>**
32. **<servlet-mapping>**
33. **<servlet-name>**DruidStatView**</servlet-name>**
34. **<url-pattern>**/druid/\***</url-pattern>**
35. **</servlet-mapping>**

第4步：可以自定义类，实现数据库信息log记录，只要在第1步的配置：

1. **<bean** id="statLogger" class="com.lizhihui.druid.StatLogger"**></bean>**

然后编写这个类的代码

1. /\*\*
2. \* druid监控sql信息(activeCount)日志打印
3. \* Created by lizhihui on 2017/10/17.
4. \*/
5. **public** **class** StatLogger **extends** DruidDataSourceStatLoggerAdapter {
6. **private** **static** Log         LOG    = LogFactory.getLog(StatLogger.**class**);
7. **private** Log                logger = LOG;
9. @Setter
10. @Getter
11. **private** Map<String, Object> map   = **new** LinkedHashMap<>();
13. **public** StatLogger(){
14. **this**.configFromProperties(System.getProperties());
15. }
17. /\*\*
18. \* @since 0.2.21
19. \*/
20. @Override
21. **public** **void** configFromProperties(Properties properties) {
22. String property = properties.getProperty("druid.stat.loggerName");
23. **if** (property != **null** && property.length() > 0) {
24. setLoggerName(property);
25. }
26. }
28. **public** Log getLogger() {
29. **return** logger;
30. }
32. @Override
33. **public** **void** setLoggerName(String loggerName) {
34. logger = LogFactory.getLog(loggerName);
35. }
37. @Override
38. **public** **void** setLogger(Log logger) {
39. **if** (logger == **null**) {
40. **throw** **new** IllegalArgumentException("logger can not be null");
41. }
42. **this**.logger = logger;
43. }
45. **public** **boolean** isLogEnable() {
46. **return** logger.isInfoEnabled();
47. }
49. **public** **void** log(String value) {
50. logger.info(value);
51. }
53. @Override
54. **public** **void** log(DruidDataSourceStatValue statValue) {
55. **if** (!isLogEnable()) {
56. **return**;
57. }
58. **int** activeCount = statValue.getActiveCount();
59. map.put("activeCount", activeCount);
60. String maxActive = PropertyUtils.getProperty("hibernate.properties", "jdbc.max\_size");
61. **if** (StringUtil.isNotEmpty(maxActive) && (activeCount == Integer.valueOf(maxActive))) {
62. String text = JSONUtils.toJSONString(map);
63. log(text);
64. }
65. }
66. }

**11. MyBatis**

**12. Spirng多环境打包方法**

**13. 权限管理（shrio）**

**14. Http协议**

**15. Linux基本使用**

**16. 项目发布脚本**

**17. Mysql知识及Mysql压测方法**

**18. poi导出百万数据的方法**

<http://blog.csdn.net/happyljw/article/details/52809244>

**思路：**

建立一个list容器 -> 存储10000行的数据到list并写入excel -> 完毕则清空list -> 再次存储10000行的数据到list并写入excel -> 直到结束

**两个关键的地方**：

1、使用SXSSFWorkbook创建Excel文档对象；

2、数据分批写入list，并及时清空list

**19. spring mvc中常用的封装类**

**19.1 BeanUtils**

**20.如何区分ajax请求**

<http://blog.csdn.net/javajiawei/article/details/50563154>

1、以X打头的头域作为非HTTP标准协议，一般是某种技术的出现而产生或者某个组织指定的， X-Requested-With头域就是用来判断一个请求是传统的HTTP请求，还是Ajax请求。也就是说Ajax的请求一般都会带上X-Requested-With头域。

2、可以利用下面的代码区分是否是ajax请求：

1. **if**(request.getHeader("X-Requested-With") != **null**
2. &&request.getHeader("X-Requested-With").equalsIgnoreCase("XMLHttpRequest")){
3. out.print("该请求是 AJAX 异步HTTP请求。");
4. } **else** {
5. out.print("该请求是传统的 同步HTTP请求。");
6. }

# 21. Spring MVC请求处理流程（待完善）

<https://www.cnblogs.com/dreamworlds/p/5396112.html>

<https://www.cnblogs.com/hujiapeng/p/5765636.html>

# 22. Spirng事务控制

<http://blog.csdn.net/bao19901210/article/details/41724355>

<http://blog.csdn.net/daijin888888/article/details/51822257>

**TransactionDefinition 事务定义信息（隔离、传播、超时、只读）**

**脏读：**一个事务读取了另一个事务改写但还未提交的数据，如果这些数据被回滚，则读到的数据是无效的。

**不可重复读：**在同一事务中，多次读取同一数据返回的结果有所不同。

**幻读：**一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。

**事务隔离级别：（五种）**

**DEFAULT**：使用后端数据库默认的隔离级别（Spring中的选择项）

**READ\_UNCOMMITED**：允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读

**READ\_COMMITTED**：允许在并发事务已经提交后读取。可防止脏读，但幻读和不可重复读仍可发生

**REPEATABLE\_READ**：对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生

**SERIALIZABLE**：完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的

其中，MySQL默认采用REPEATABLE\_READ隔离级别；Oracle默认采用READ\_COMMITTED隔离级别

**事务的作用：**将若干的数据库操作作为一个整体控制,一起成功或一起失败。

**原子性：**指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

**一致性：**指事务前后数据的完整性必须保持一致。

**隔离性：**指多个用户并发访问数据库时，一个用户的事务不能被其他用户的事务所干扰，多个并发事务之间数据要相互隔离。

**持久性：**指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，即时数据库发生故障也不应该对其有任何影响。

**务传播行为：（七种）**

**REQUIRED：**支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

**SUPPORTS：**支持当前事务，如果当前没有事务，就以非事务方式执行。

**MANDATORY：**支持当前事务，如果当前没有事务，就抛出异常。

**REQUIRES\_NEW：**新建事务，如果当前存在事务，把当前事务挂起。

**NOT\_SUPPORTED：**以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

**NEVER：**以非事务方式执行，如果当前存在事务，则抛出异常。

**NESTED：**如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与REQUIRED类似的操作。拥有多个可以回滚的保存点，内部回滚不会对外部事务产生影响。只对DataSourceTransactionManager有效。

* 虽然 @Transactional 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 建议不要在接口或者接口方法上使用该注解，因为这只有在使用基于接口的代理时它才会生效。另外， @Transactional 注解应该只被应用到 public 方法上，这是由 Spring AOP 的本质决定的。如果你在 protected、private 或者默认可见性的方法上使用 @Transactional 注解，这将被忽略，也不会抛出任何异常。
* 默认配置下，spring只有在抛出的异常为运行时unchecked异常时才回滚该事务，也就是抛出的异常为RuntimeException的子类(Errors也会导致事务回滚)，而抛出checked异常则不会导致事务回滚，可以用(rollbackFor=Exception.class)指定checked异常回滚
* 今天在开发中，遇到了如题这样的问题，我自己手动抛出异常，数据还是会写入的数据库中，并且不会发生回滚，导致产生了垃圾数据。后来上网查了一下，发现spring中，注解的事务回滚是默认RunTimeException才会回滚，如果抛出的不是RunTimeException，spring是默认不会回滚的。并且，你的程序中try catch了，哪怕是RunTimeException，也是不会回滚的。
* @Transactional 的事务开启 ，或者是基于接口的 或者是基于类的代理被创建。所以在同一个类中一个方法调用另一个方法有事务的方法，事务是不会起作用的。
* 通过元素的 “proxy-target-class” 属性值来控制是基于接口的还是基于类的代理被创建。如果 “proxy-target-class” 属值被设置为 “true”，那么基于类的代理将起作用（这时需要CGLIB库cglib.jar在CLASSPATH中）。如果 “proxy-target-class” 属值被设置为 “false” 或者这个属性被省略，那么标准的JDK基于接口的代理将起作用。
* 注意仅仅 @Transactional 注解的出现不足于开启事务行为，它仅仅是一种元数据。必须在配置文件中使用配置元素，才真正开启了事务行为。
* 数据库表引擎是Myisam不支持事务，InnoDB支持（切记）
* 确认有如下配置：<tx:annotation-driven transaction-manager="transactionManager"/>