第二章 Spring 框架（二）

课程目标

1、 上章回顾

2、 Spring 与 Junit 整合测试

3、 Spring 的注解使用

4、 Spring 中的 Aop 深入详解

课程内容

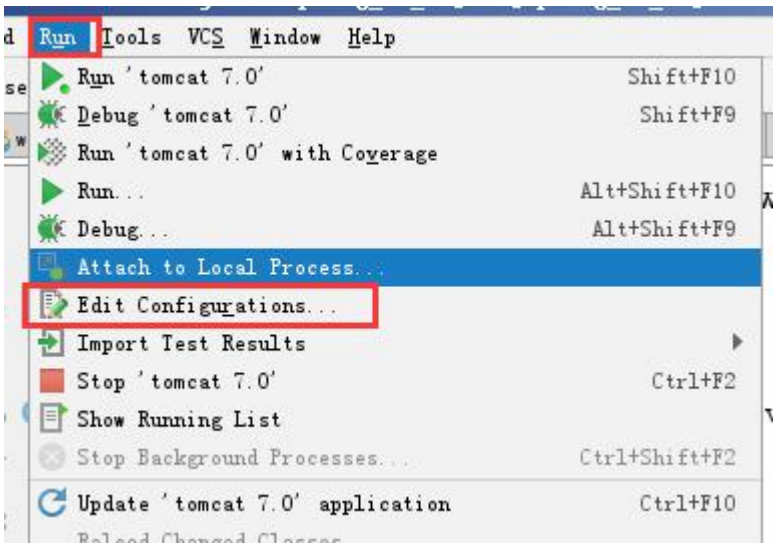
1、 上章回顾

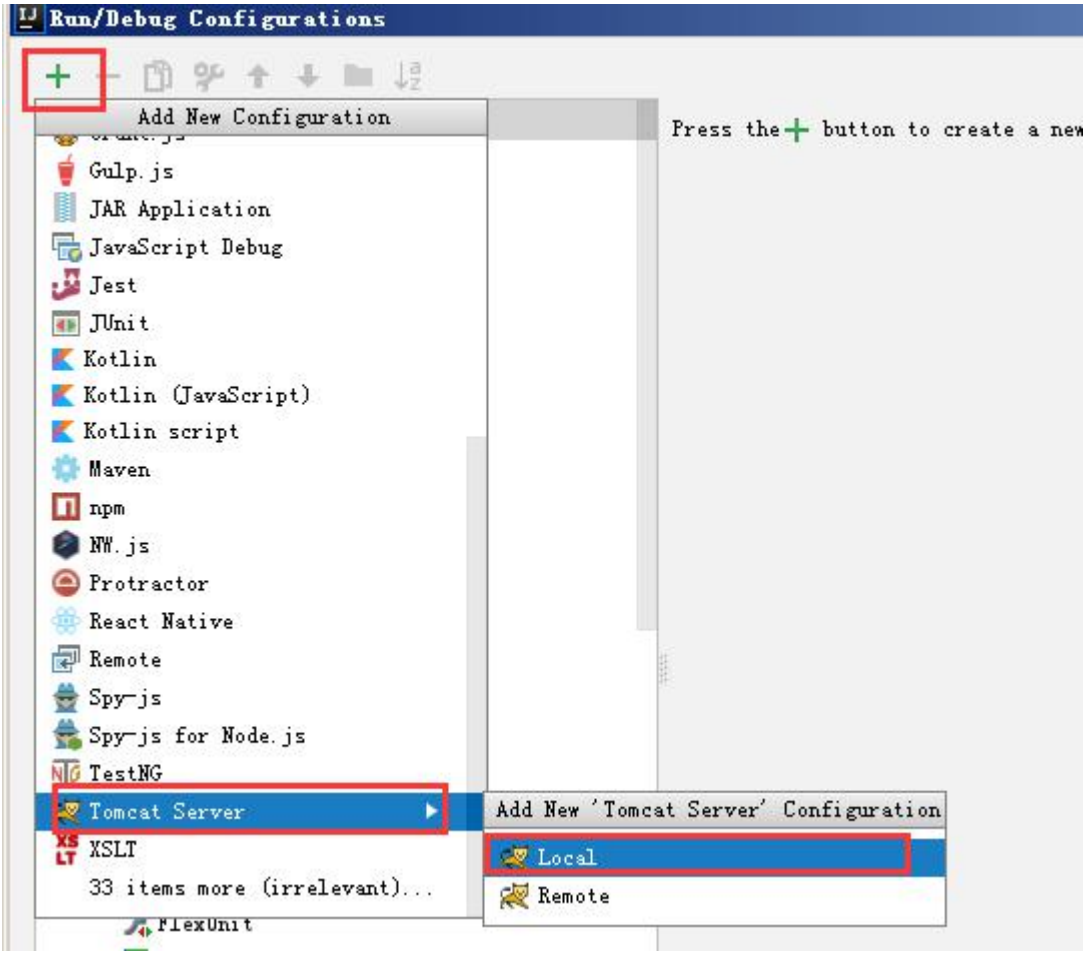
补充知识点：

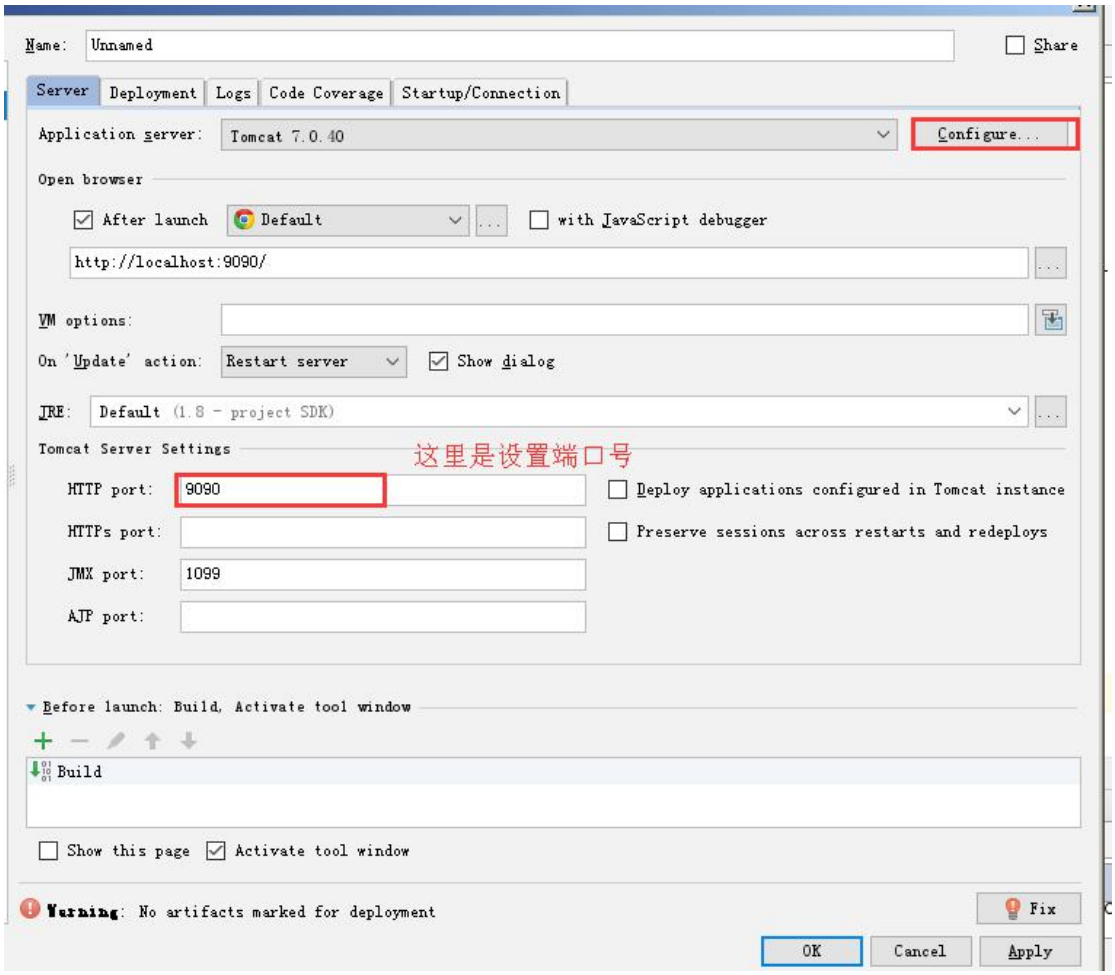
1.1）如何在 idea 中开发 web 项目？（开发步骤）

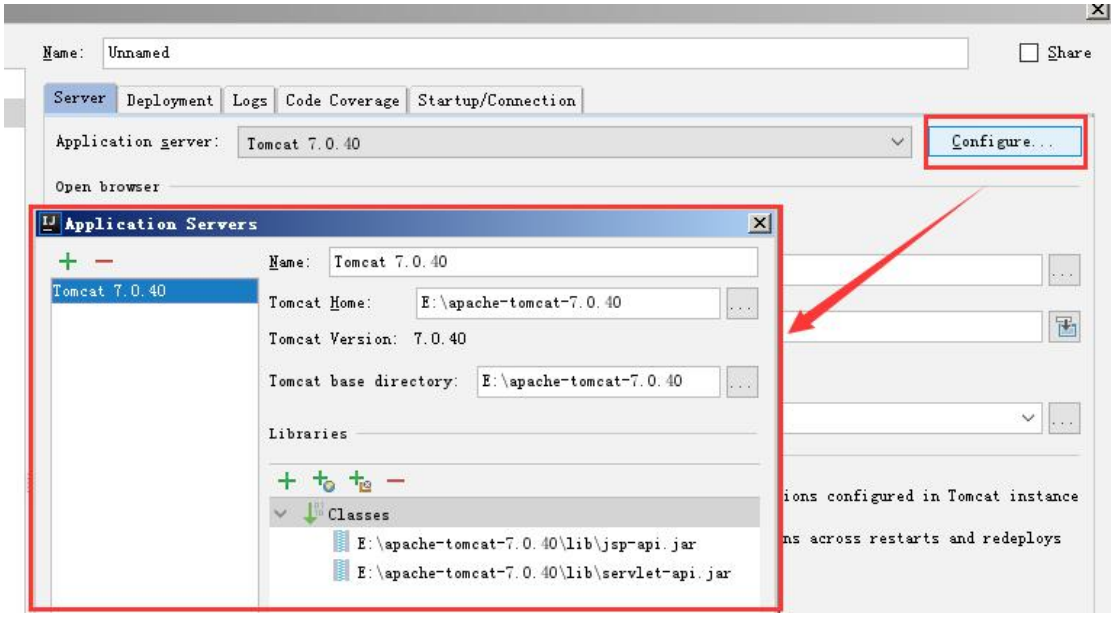
第一步：在 idea 中新建一个 web 工程。

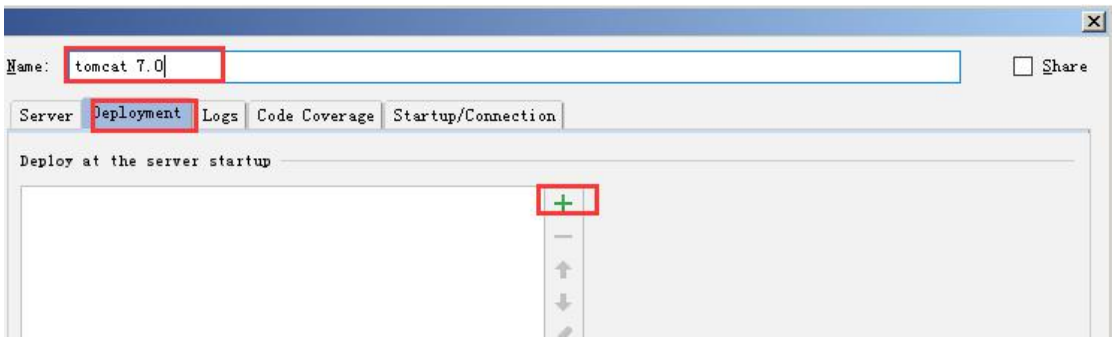
第二步：添加 tomcat 服务器。

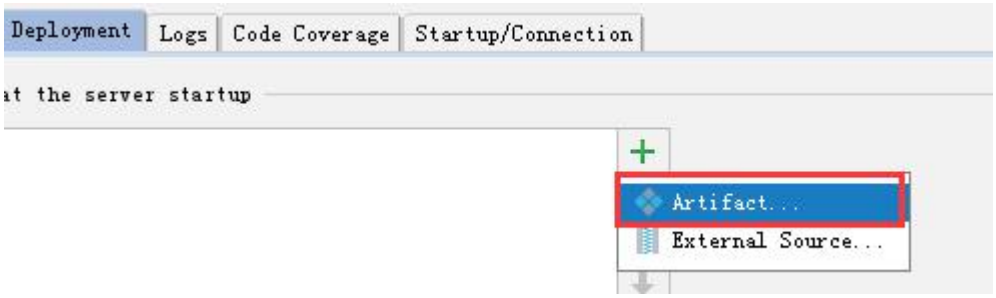


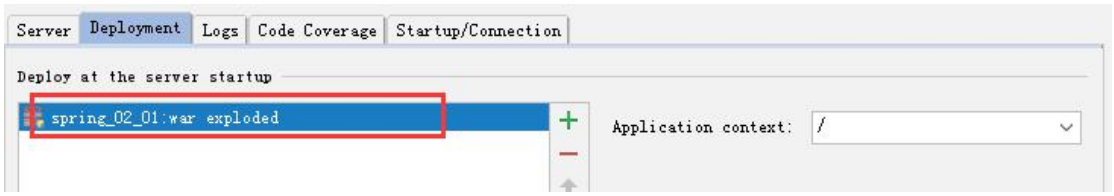

Run/Debug Configurations
Add New Configuration
Press the + button to create a new
Gulp.js
JAR Application
JavaScript Debug
Jest
JUnit
Kotlin
Kotlin (JavaScript)
Kotlin script
Maven
npm
NW.js
Protractor
React Native
Remote
Spy-js
Spy-js for Node.js
TestNG
Tomcat Server
XSLT
33 items more (irrelevant)...
FlexUnit
Add New 'Tomcat Server' Configuration
Local
Remote

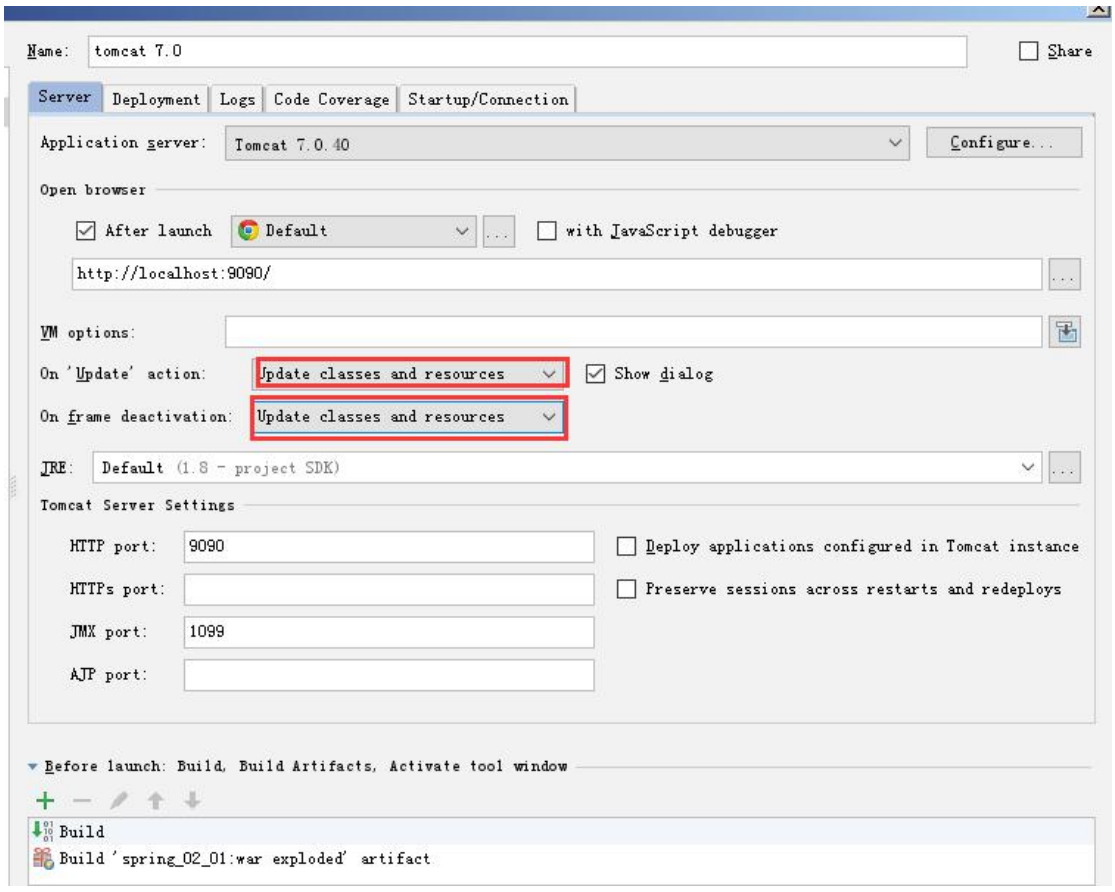

Name: Unnamed
Share
Server
Deployment
Logs
Code Coverage
Startup/Connection
Application server: Tomcat 7.0.40
Configure...
Open browser
After launch
Default
with JavaScript debugger
http://localhost:9090/
VM options:
On 'Update' action: Restart server
Show dialog
JRE: Default (1.8 - project SDK)
Tomcat Server Settings
这里是设置端口号
HTTP port: 9090
HTTPs port:
JMX port: 1099
AJP port:
Deploy applications configured in Tomcat instance
Preserve sessions across restarts and redeploys
Before launch: Build, Activate tool window
Build
Show this page
Activate tool window
Warning: No artifacts marked for deployment
Fix
OK
Cancel
Apply


Name: Unnamed
Share
Server
Deployment
Logs
Code Coverage
Startup/Connection
Application server: Tomcat 7.0.40
Configure...
Open browser
Application Servers
Tomcat 7.0.40
Name: Tomcat 7.0.40
Tomcat Home: E:\apache-tomcat-7.0.40
Tomcat Version: 7.0.40
Tomcat base directory: E:\apache-tomcat-7.0.40
Libraries
Classes
E:\apache-tomcat-7.0.40\lib\jsp-api.jar
E:\apache-tomcat-7.0.40\lib\servlet-api.jar


Name: tomcat 7.0
Share
Server
Deployment
Logs
Code Coverage
Startup/Connection
Deploy at the server startup

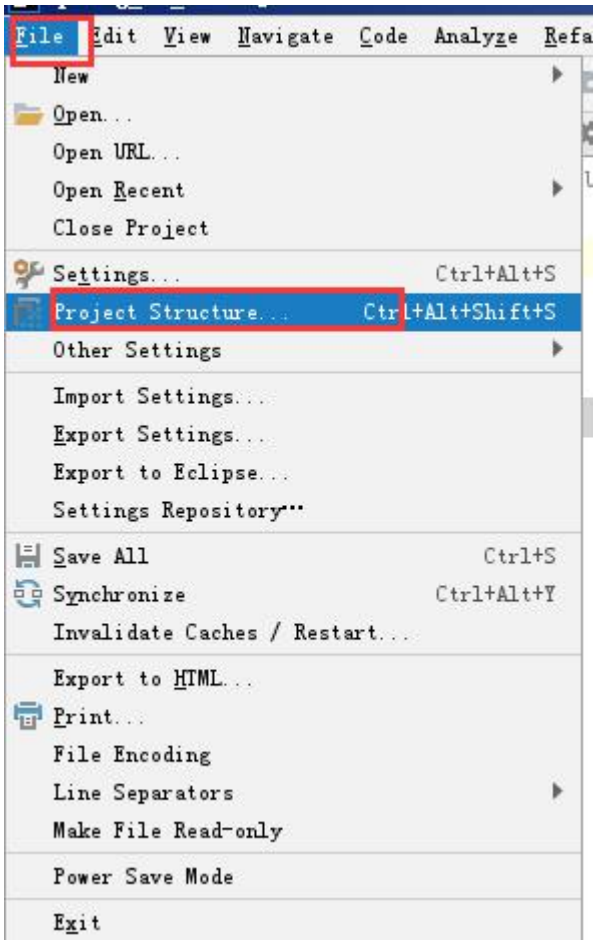

Deployment
Logs
Code Coverage
Startup/Connection
t the server startup
Artifact...
External Source...

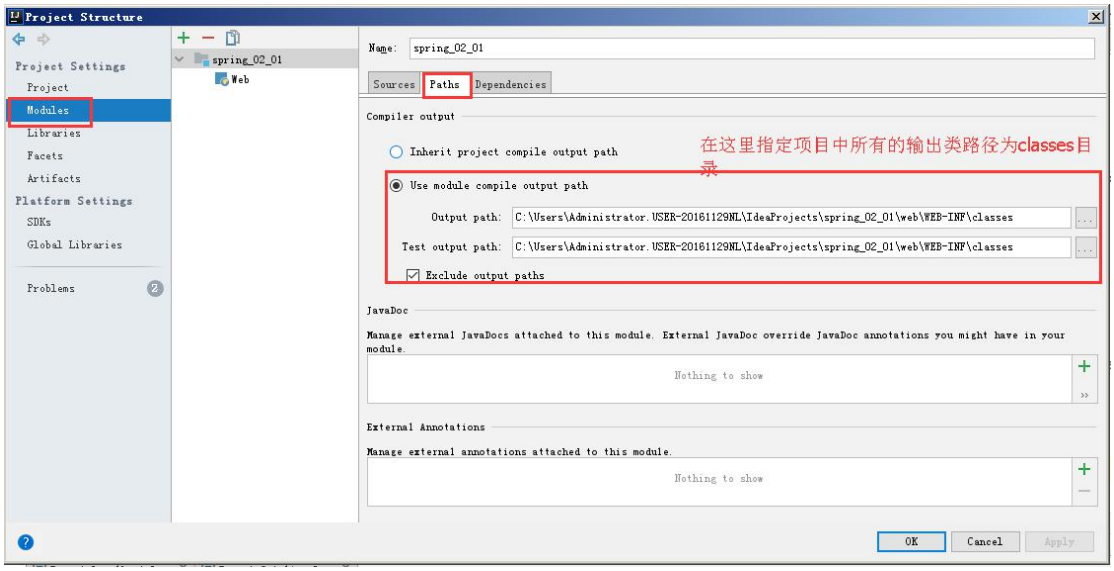

Server
Deployment
Logs
Code Coverage
Startup/Connection
Deploy at the server startup
spring_02_01:war exploded
Application context: /

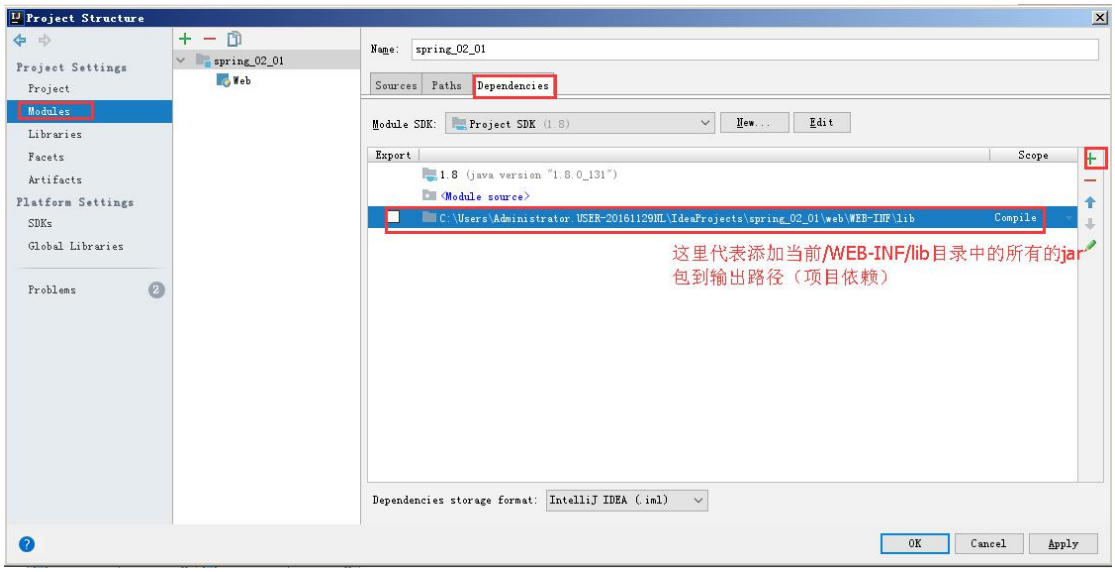

Name: tomcat 7.0
Share
Server
Deployment
Logs
Code Coverage
Startup/Connection
Application server: Tomcat 7.0.40
Configure...
Open browser
After launch
Default
with JavaScript debugger
http://localhost:9090/
VM options:
On 'Update' action: Update classes and resources
Show dialog
On frame deactivation: Update classes and resources
JRE: Default (1.8 - project SDK)
Tomcat Server Settings
HTTP port: 9090
HTTPs port:
JMX port: 1099
AJP port:
Deploy applications configured in Tomcat instance
Preserve sessions across restarts and redeploys
Before launch: Build, Build Artifacts, Activate tool window
Build
Build 'spring_02_01:war exploded' artifact

第三步：在当前 web 项目的/WEB-INF 下添加 classes 目录及 lib 目录。

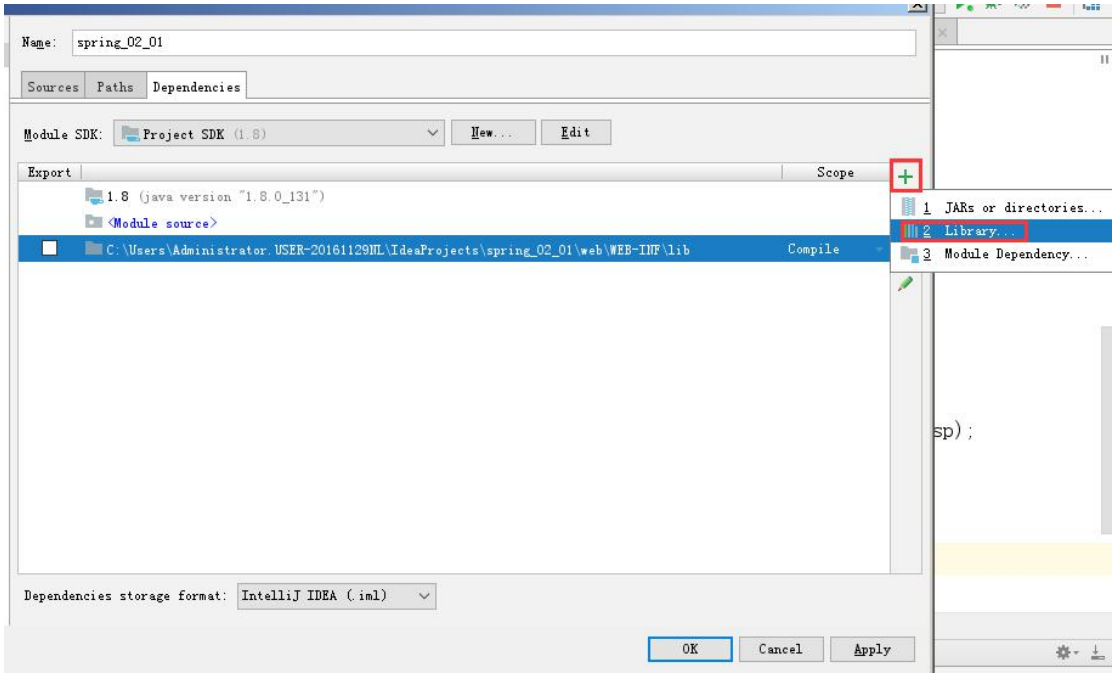
第四步：设置 classes 目录为输出目录,并同时指定 lib 目录为输出 jar 包目录（我们自己放入的 jar 包）

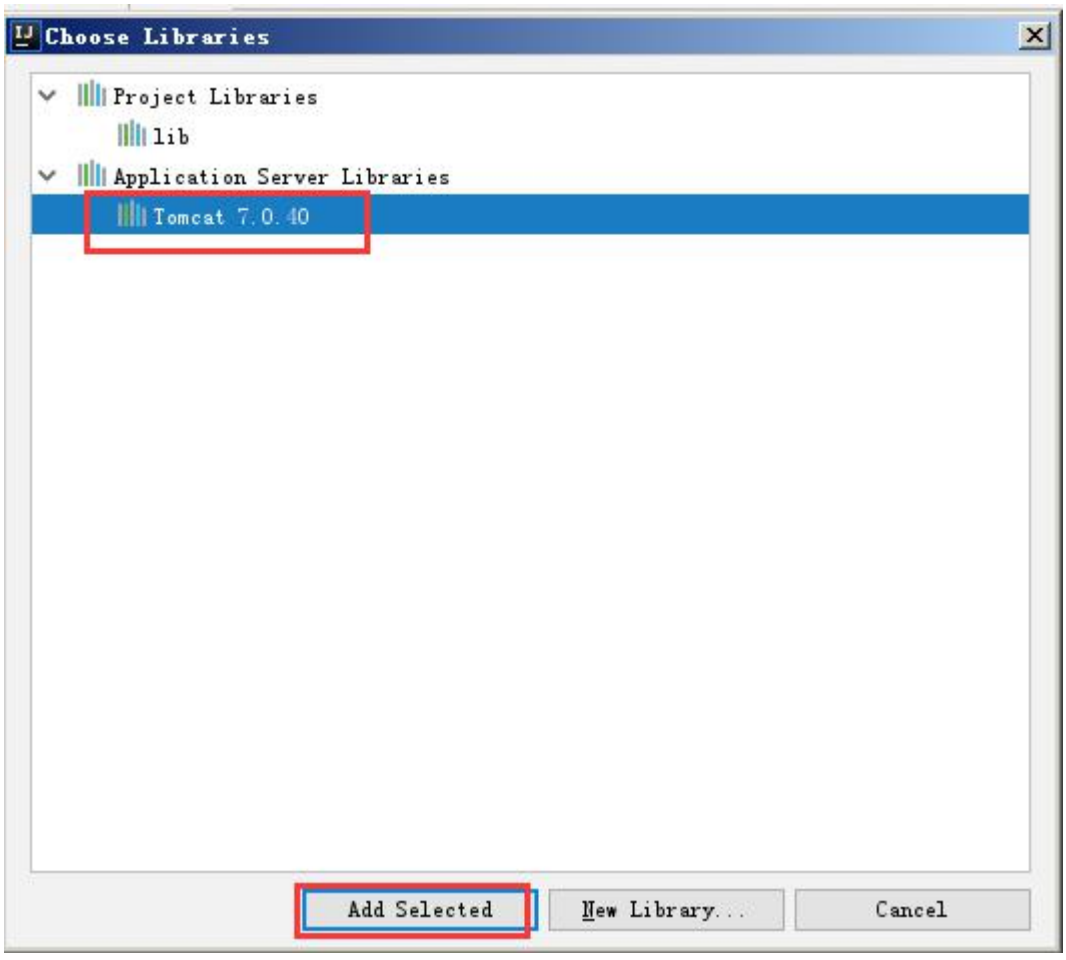


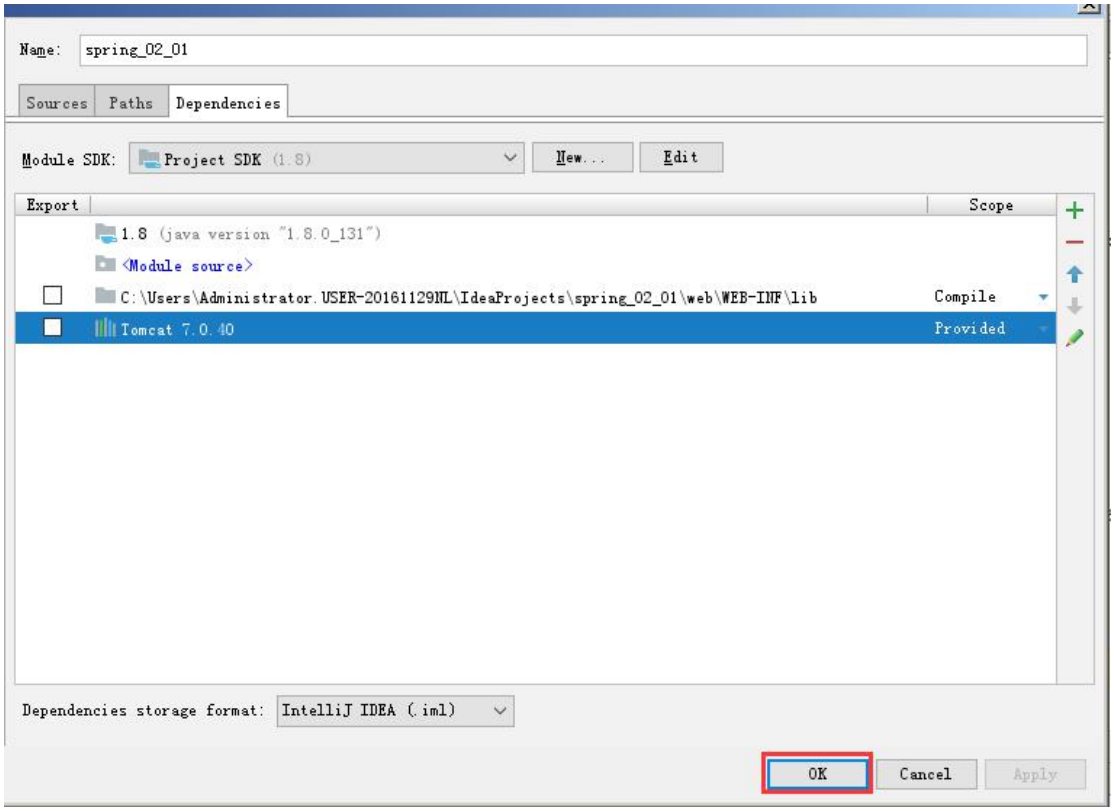

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
spring_02_01
Web
Name: spring_02_01
Sources
Paths
Dependencies
Compiler output
Inherit project compile output path
Use module compile output path
在这里指定项目中所有的输出类路径为classes目录
Output path: C:\Users\Administrator.USER-20161129NL\IdeaProjects\spring_02_01\web\WEB-INF\classes
Test output path: C:\Users\Administrator.USER-20161129NL\IdeaProjects\spring_02_01\web\WEB-INF\classes
Exclude output paths
JavaDoc
Manage external JavaDocs attached to this module. External JavaDoc override JavaDoc annotations you might have in your module.
Nothing to show
External Annotations
Manage external annotations attached to this module.
Nothing to show
OK
Cancel
Apply


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
spring_02_01
Web
Name: spring_02_01
Sources
Paths
Dependencies
Module SDK: Project SDK (1.8)
New...
Edit
Export
Scope
1.8 (java version "1.8.0_131")
<Module source>
C:\Users\Administrator.USER-20161129NL\IdeaProjects\spring_02_01\web\WEB-INF\lib
Compile
这里代表添加当前/WEB-INF/lib目录中的所有的jar包到输出路径（项目依赖）
Dependencies storage format: IntelliJ IDEA (.iml)
OK
Cancel
Apply

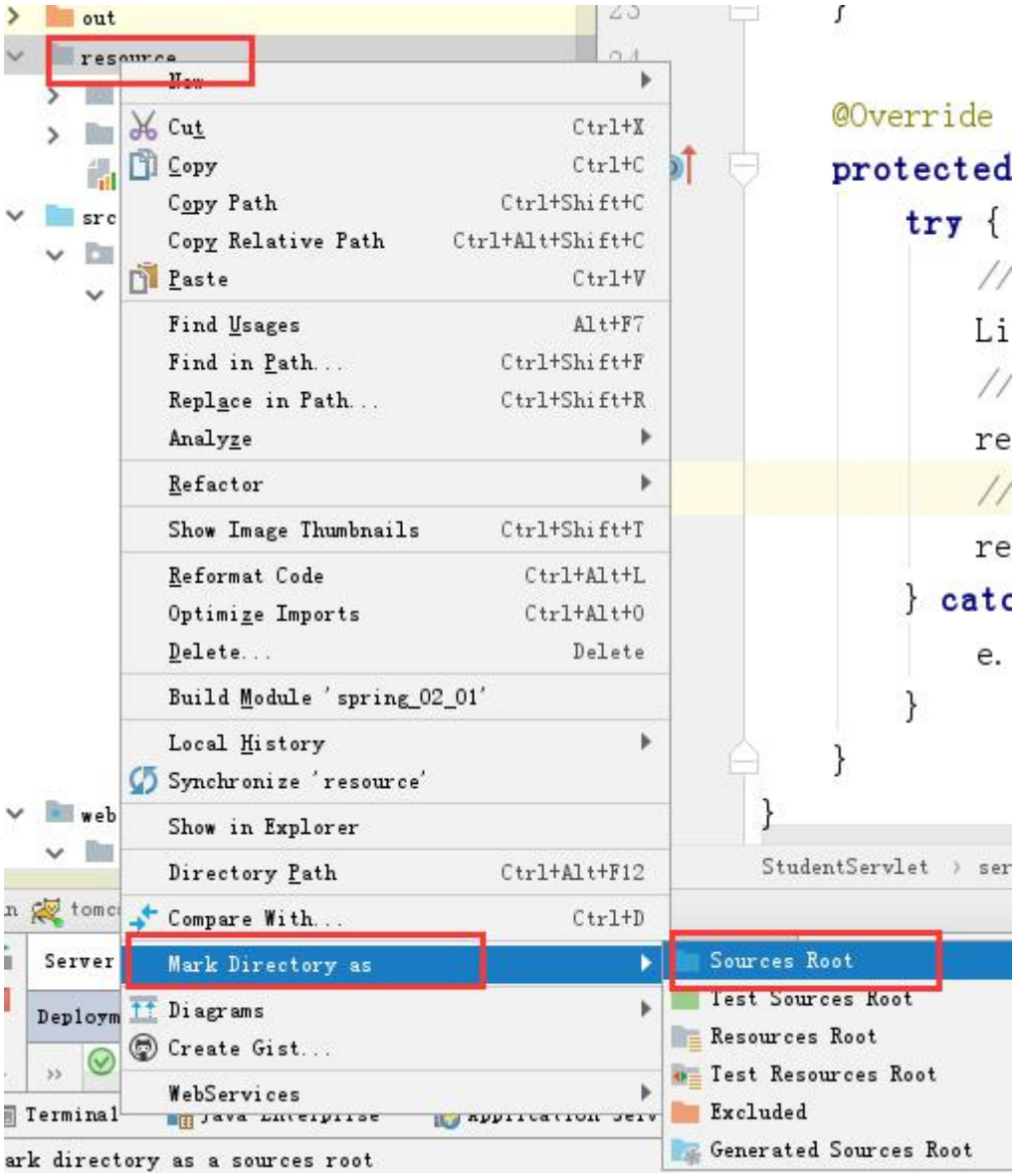
第五步：添加 tomcat 到当前项目的外部库






Name: spring_02_01
Sources
Paths
Dependencies
Module SDK: Project SDK (1.8)
New...
Edit
Export
Scope
1.8 (java version "1.8.0_131")
<Module source>
C:\Users\Administrator.USER-20161129NL\IdeaProjects\spring_02_01\web\WEB-INF\lib
Compile
Tomcat 7.0.40
Provided
Dependencies storage format: IntelliJ IDEA (.iml)
OK
Cancel
Apply

第六步：定义我们自己的资源文件夹为系统的 sourceFolder



2、 Spring 与 Junit 整合测试

2.1）第一步：添加如下 jar 包：

2.2）第二步：添加使用 dbutils+c3p0 需要的 jar 包：

2.3）定义 DAO 及 Service 层：

【 StudentDaoImpl.java 】

```
public class StudentDaoImpl implements StudentDao {
    private QueryRunner qr;
```

```
    public StudentDaoImpl() {
        this.qr = new QueryRunner(JdbcUtils.getDataSource());
    }
    @Override
    public List<Student> findAll() throws Exception {
        String sql = "select * from student";
        return qr.query(sql, new BeanListHandler<>(Student.class));
    }

}
```

【StudentServiceImpl.java】

```
public class StudentServiceImpl implements StudentService {

    private StudentDao studentDao;
    public void setStudentDao(StudentDao studentDao) {
        this.studentDao = studentDao;
    }

    @Override
    public List<Student> findAll() throws Exception {
        return studentDao.findAll();
    }
}
```

2.4）进行单元测试：

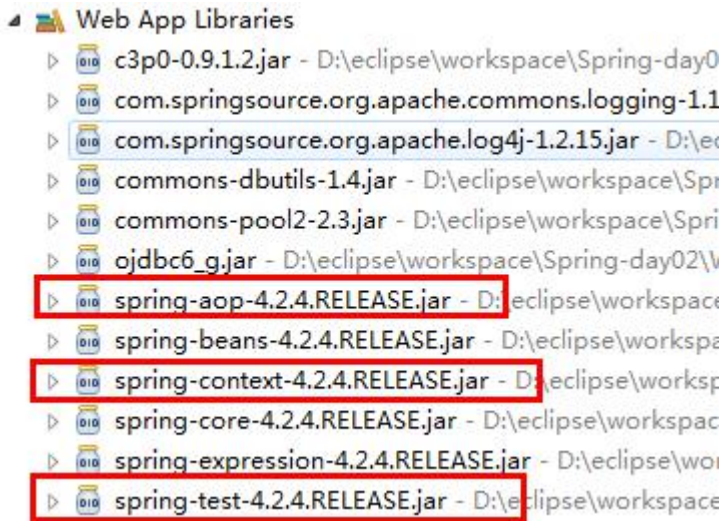
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class TestStudentServiceImpl {

    //方法一：使用ApplicationContext获取spring容器中的对象
    /*@Test
    public void testFindAll() throws Exception{
        ApplicationContext ac =
                new ClassPathXmlApplicationContext("applicationContext.xml");
        StudentService studentService =   (StudentService) ac.getBean("studentService");
        List<Student> students = studentService.findAll();
        for(Student student : students){
            System.out.println(student);
        }
    }*/
```

将Spring与junit4整合，需要使用的注解。

传统方法获取Spring容器中的某个javabean对象。

3、 Spring 的注解使用（重要）

3.1）准备工作：添加相关的 aop 与 test 及 context 三个 jar 包：



注解：

第一部分：spring 中将 javabean 放到容器中的注解有四种：(最重要)

① Repository ② Service ③ Controller ④ Component

这四种注解的作用，在目前现在有的版本是一样的，将 javabean 放到 spring 容器中，使用时要注意，一般如果是 dao 层使用@Repository 注解，如果是 service 层使用@Service 注解，如果后面学习到了 springmvc 的控制器层，使用 @Controller ，如果不能确定是哪种组件可以使用@Component 注解

第二部分：取出 spring 中的 javabean 的相关注解：(最重要)

① @Autowired ② @Qualifier ③ @Resource

第一种@Autowired 注解 ,代表根据类型从 spring 容器中取出 javabean 注入到当

前的工程中（常用）第二种@Qualifier 注解，代表如果 javabean 的类型一样，则按照指定的名字取 javabean 第三种@Resource 注解，代表首先按照名字取 javabean，如果名字一样，则按照类型取。

第三部分： javabean 作用范围的注解

@Scope 注解可取值: singleton（单例）、prototype（多例）、request、session

第四部分：javabean 生命周期的注解

@PostConstruct 和@Predestroy 分别代表 javabean 初始化时的注解和 javabean 销毁时的注解。

3.2）在单元测试环境下使用注解：

3.2.1）第一部分：从容器中取 javabean 的注解：（非常重要）

```
//如下两组注解作用：从spring容器中取得某个javabean对象
//第一组：@Autowired + @Qualifier("studentService1")
//@Autowired      //默认情况下：此注解按照类型在spring容器中进行bean的查找，如果有多个bean的类型一样，哪么这个注解不能满足要求
//@Qualifier("studentService1")   //在类型一致的情况下，指定同类型的某个bean的名称，这样就可以得到某个具体的bean了.
//一般情况下，使用@Autowired与@Qualifier进行联合使用

//第二组：@Resource(name="studentService1")
//小结：第一组的使用与第二组的使用一样，但第二组简单，所以常用第二组
@Resource(name="studentService1")
private StudentService studentService;
@Test
public void testFindAll() throws Exception {
    List<Student> students = studentService.findAll();
    for (Student student : students) {
        System.out.println(student);
    }

}
```

Spring注解第一部分：获取JavaBean的两组注解。

3.2.1.1）运行效果如下：

```
log4j:WARN No appenders could be found for logger (org.springframework
log4j:WARN Please initialize the log4j system properly.
Student [sid=1005, sname=王五, sex=男, age=20, addr=深圳, cid=2]
Student [sid=1001, sname=张三, sex=男, age=21, addr=上海, cid=1]
Student [sid=1002, sname=李四, sex=男, age=20, addr=杭州, cid=2]
Student [sid=1003, sname=赵六, sex=女, age=19, addr=广州, cid=3]
```

3.2.2）第二部分：向容器中放入 javabean 的注解：（非常重要）

```
//第二部分注解：主要有如下四个，向容器中放javabean
//向容器中存放名为"studentSerice"的javabean
@Service("studentService")
//@Component("studentService")
//@Controller("studentService")
//@Repository("studentService")

//spring发明了以上四种将javabean放到容器中的注解，功能目前一样，未来会扩展功能。
//一般情况下，@Service用在serice层，@Component具有宽泛型，可以通用，但一般
//不用。@Controller一般用在控制器层中，如:springMvc中。@Repository一般
//用在dao层
public class StudentServiceImpl implements StudentService {
```

3.2.2.1）第二部分注解应用：

```
//向容器中存放名为"studentDao"的javabean
@Repository("studentDao")
public class StudentDaoImpl implements StudentDao {
    private QueryRunner qr;
    public StudentDaoImpl() {
        this.qr = new QueryRunner(JdbcUtils.getDataSource());
    }
```

```
@Service("studentService")
public class StudentServiceImpl implements StudentService {

    //使用@Resource注解从容器中取出名为"studentDao"的哪个javabean
    @Resource(name="studentDao")
    private StudentDao studentDao;
    public void setStudentDao(StudentDao studentDao) {
        this.studentDao = studentDao;
    }
```

3.2.2.2）测试：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext2.xml")
public class TestAnnoServiceImpl {

    @Resource(name="studentService")
    private StudentService studentService;
    @Test
    public void test() throws Exception{
        List<Student> students = studentService.findAll();
        for(Student student :students){
            System.out.println(student);
        }
    }
}
```

3.2.2.3）运行效果：

```
log4j:WARN No appenders could be found for logger (org.springframework
log4j:WARN Please initialize the log4j system properly.
Student [sid=1005, sname=王五, sex=男, age=20, addr=深圳, cid=2]
Student [sid=1001, sname=张三, sex=男, age=21, addr=上海, cid=1]
Student [sid=1002, sname=李四, sex=男, age=20, addr=杭州, cid=2]
Student [sid=1003, sname=赵六, sex=女, age=19, addr=广州, cid=3]
```

3.2.3）第三部分：设置 javabean 作用范围的注解：

3.2.3.1）StudentServ iceImpl.java 关键代码：

```
//设置当前bean的作用范围，四个值：
//singleton(单例,默认值)，prototype(多例),
//requst(与web中request有同样的作用域，不常用)
//session(与web中session有同样的作用域，不常用)
@Scope(scopeName="prototype")
public class StudentServiceImpl implements StudentService {
```

3.2.3.2）测试代码：

```
@Resource(name="studentService")
private StudentService studentService;
@Resource(name="studentService")
private StudentService studentService1;
@Test
public void test() throws Exception{
    List<Student> students = studentService.findAll();
    for(Student student :students){
        System.out.println(student);
    }
    System.out.println(studentService == studentService1);
}
```

3.2.3.3）运行结果：

```
log4j:WARN Please initialize the log4j system properly.
Student [sid=1005, sname=王五, sex=男, age=20, addr=深圳, cid=2]
Student [sid=1001, sname=张三, sex=男, age=21, addr=上海, cid=1]
Student [sid=1002, sname=李四, sex=男, age=20, addr=杭州, cid=2]
Student [sid=1003, sname=赵六, sex=女, age=19, addr=广州, cid=3]
false
```

代表两个service不是同一个对象。

3.2.4）第四部分：初始化/销毁的注解：

3.2.4.1）Student.java 关键代码：

```
@PostConstruct      //初始化注解
public void init(){
    System.out.println("Student→init().");
}
@PreDestroy         //销毁注解
public void destroy(){
    System.out.println("Student→destroy().");
}
```

初始化及销毁注解。

3.2.4.2）单元测试的关键代码：

```
@Resource(name="student")
private Student student;
@Test
public void test1() throws Exception{
    System.out.println(student);
}
```

3.2.4.3）运行结果：

```
log4j:WARN Please initialize the log4j system properly.
Student→init().
Student [sid=0, sname=null, sex=null, age=0, addr=null, cid=0]
Student→destroy().
```

❀ 实战案例：Sping 注解+BS+Ajax 完成列表分页

1）在 web.xml 文件中配置 spring 监听器及指定配置文件位置：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">
  <display-name>spring-web-02</display-name>
<!--指定配置文件的位置，如果不指定默认加载/WEB-INF/applicationContext.xml-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring/applicationContext.xml</param-value>
  </context-param>
<!--配置 spring 监听器，加载 spring 容器-->
  <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</
listener-class>
  </listener>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <description></description>
    <display-name>StudentServlet</display-name>
    <servlet-name>StudentServlet</servlet-name>
    <servlet-class>com.zelin.web.servlet.StudentServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>StudentServlet</servlet-name>
```

```
    <url-pattern>/student</url-pattern>
  </servlet-mapping>
</web-app>
```

2）配置类路径下的 spring/applicationContext.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.2.xsd">
    <!-- 0、配置 spring 的扫描包及其子包 -->
    <context:component-scan base-package="com.zelin"/>
    <!-- 1、读取配置文件 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 2、配置数据源 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${db.driver}"/>
        <property name="jdbcUrl" value="${db.url}"/>
        <property name="user" value="${db.user}"/>
        <property name="password" value="${db.password}"/>
    </bean>
    <!-- 3、配置 queryRunner 对象 -->
    <bean id="qr" class="org.apache.commons.dbutils.QueryRunner">
        <!-- 为 queryRunner 对象注入数据源 -->
        <constructor-arg name="ds" ref="dataSource"/>
    </bean>
</beans>
```

3）定义 StudentDaoImpl 类：

```
/**
 *  Spring 注解第一部分：【将 javaBean 放入到 spring 容器中】<非常重要>
    @Repository("studentDao")
    @Component("studentDao")
    @Service("studentDao")
    @Controller("studentDao")
    小结：
    ① 上面的四个注解在目前 spring3.x,4.x,5.x 所有的版本中作用是一致的，都是将当前的类实例化
后放入到 spring 容器中
    ② 当前，不排除以后 spring 后对其赋予不同的意义。
    ③ 一般使用时，约定如下：
        如果是 dao 层选用@Repository
        如果是 service 层选用@Service 注解，
```

```
        如果是控制器层选用@Controller
*/
@Repository("studentDao")                    //注意:这里如果不为其指定 bean 的名称，默认就是类名首字母小写
public class StudentDaoImpl implements StudentDao {
    /**
     *  Spring 注解第二部分：【如何从 spring 容器中取出一个 javaBean】  <非常重要>
     *  ① @Autowired  //此注解代表从 spring 容器中取出与指定属性同类型的 javaBean(byType)，所以，当 spring 容器中
     *     如果有多个同类型的 javaBean 时，就没办法取出了，这时，就有使用@Qualifier("javaBean 名称")注解来配合@Autowired 注解
     *  ② Resource("qr") //此注解代表从 spring 容器中取出与指定属性同名的 javaBean(byName)，此注解还有一个功能，就
     *     是如果按名称取不出 javaBean，就按照类型来取
     */
    //@Autowired @Qualifier("qr")
    //@Resource(name="qr")

    @Autowired
    private QueryRunner qr;
    @Override
    public List<StudentCustom> findAll() throws Exception {
        return qr.query("select st.*,cname from student st,classes c where c.cid=st.cid",
                new BeanListHandler<>(StudentCustom.class));
    }
    @Override
    public Long findCount() throws SQLException {
        return (Long) qr.query("select count(*) from student st,classes c where c.cid=st.cid",
                new ScalarHandler());
    }
    @Override
    public List<StudentCustom> findStudentsPerPage(int page, int pagesize) throws SQLException {
        String sql = "select st.*,cname from student st,classes c where c.cid=st.cid ";
        sql += "limit ?,?";
        return qr.query(sql, new BeanListHandler<>(StudentCustom.class),
                (page-1)*pagesize,pagesize);
    }
}
```

4）定义 StudentServiceImpl 类：

```
@Service
public class StudentServiceImpl implements StudentService {
    @Autowired
    private StudentDao studentDao;
    @Value("${pagesize}")
    private int pagesize;
    @Override
    public List<StudentCustom> findAll() throws Exception {
        return studentDao.findAll();
    }
    @Override
    public PageResult<StudentCustom> findPage(int page) throws Exception {
        //1、计算总页数
        int totalPage =
(int)Math.ceil(studentDao.findCount()*1.0/pagesize);
        //2.获取每页记录的集合
        List<StudentCustom> students =
studentDao.findStudentsPerPage(page,pagesize);
        return new PageResult<>(totalPage, page, students, pagesize);
    }
}
```

5）定义 db.properties 文件：

```
db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql:///java1301?useUnicode=true&amp;characterEncoding=utf-8
db.user=root
db.password=123
pagesize=6
```

6）定义 StudentServlet 类：

```
public class StudentServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private StudentService studentService;
    @Override
    public void init() throws ServletException {
        //1.取得 springweb 容器
        WebApplicationContext wac =
WebApplicationContextUtils.getWebApplicationContext(getServletContext())
;
        //2.得到容器中的 studentService 对象
        studentService = (StudentService)
wac.getBean("studentServiceImpl");
    }
```

```
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        //1.指定响应的内容类型
        resp.setContentType("text/html;charset=utf-8");
        //2.得到请求参数
        String method = req.getParameter("method");
        //3.根据请求参数作处理
        if("list".equals(method)){                //列表所有学生
            list(req,resp);
        }else if("listpage".equals(method)){ //分页列表所有学生
            listpage(req,resp);
        }
    }
    private void list(HttpServletRequest req, HttpServletResponse resp) {
        try {
            //1.查询所有的学生
            List<StudentCustom> students = studentService.findAll();
            //2.转换集合为 json 串
            String jsonString = JSON.toJSONString(students);
            //3.输出
            resp.getWriter().print(jsonString);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //分页查询所有学生
    private void listpage(HttpServletRequest req, HttpServletResponse resp)
{
        try {
            //1.得到当前页
            String pageStr = req.getParameter("page");
            int page =  new Integer(pageStr) ;
            //1.获取分页的 PageResult 对象
            PageResult<StudentCustom> students =
studentService.findPage(page);
            //2.转换集合为 json 串
            String jsonString = JSON.toJSONString(students);
            //3.输出
            resp.getWriter().print(jsonString);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
}
```

7）定义 index.html 页面：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>列表学生信息</title>
  <link rel="stylesheet" href="bootstrap-3.3.7/css/bootstrap.min.css">
  <script src="bootstrap-3.3.7/js/jquery.min.js"></script>
  <script src="bootstrap-3.3.7/js/bootstrap.min.js"></script>
  <script src="bootstrap-3.3.7/js/docs.min.js"></script>
  <style>
    .clearfix:after{
        display:block;
        clear:both;
        height:0;
        visibility:hidden;
        content:'';
    }
    .table{
        text-align:center;
    }
    .panel-footer{
        padding:0px;
        padding-right:10px;
    }
  </style>
</head>
<body>
    <div class="container">
        <div class="page-header">
            <h3>学生管理系统</h3>
        </div>
        <div class="panel panel-primary">
            <div class="panel-heading">
                <h4 class="glyphicon glyphicon-th-list"> 学生列表</h4>
            </div>
            <table class="table table-hover ">
                <thead>
                    <tr>
                        <td>学号</td>
                        <td>姓名</td>
                        <td>性别</td>
```

```
                    <td>年龄</td>
                    <td>住址</td>
                    <td>所在班级</td>
                    <td>操作</td>
                </tr>
            </thead>
            <tbody></tbody>

        </table>
        <div class="panel-footer clearfix">
            <!-- 这里定义分页导航 -->
            <nav aria-label="Page navigation ">
                <ul class="pagination pull-right "></ul>
            </nav>
        </div>
    </div>
</div>
<script>
    function reloadList(page){
        $.post(
                "student?method=listpage&page="+page,
                function(data){
                    var info = "";
                    var navInfo = "<li><a
href='#' ><span>&laquo;</span></a></li>";
                    $.each(data.students,function(i,v){
                        //1.遍历出数据表格
                        info += "<tr>";
                        info += "<td>" + v.sid + "</td>"
                        info += "<td>" + v.sname + "</td>"
                        info += "<td>" + v.sex + "</td>"
                        info += "<td>" + v.age + "</td>"
                        info += "<td>" + v.addr + "</td>"
                        info += "<td>" + v.cname + "</td>"
                        info += "<td><a class='btn btn-success btn-xs'
href=''><span class='glyphicon glyphicon-pencil'>修改</span></a> " ;
                        info +="<a class='btn btn-danger btn-xs'
href=''><span class='glyphicon glyphicon-remove'>删除</span></a></td>";
                        info += "</tr>";
                    })
                    //遍历总页数，动态生成导航条
                    for(var i = 1;i <= data.totalPage;i++){
                        navInfo += "<li><a
href='javascript:reloadList("+i+")'>" + i + "</a></li>"
```

```
                    };
                    navInfo += "<li><a
href='#' ><span>&raquo;</span></a></li>";
                    $('table tbody').html(info);
                    //将导航条的内容赋值给 class="pagination"的 ul 即可
                    $(".pagination").html(navInfo);
                },'json');
        }
        $(function(){
            //加载数据列表
            reloadList(1);
        })

    </script>
</body>
</html>
```

8）运行效果如下：

4、 Spring 中的 Aop 深入详解

AOP: Aspect Oriented Programming

4.1）静态代理：

4.1.1) UserDao 接口：

```
public interface UserDao {
    public void add();
```

```
    public void update();
    public void delete();
    public void query();
}
```

4.1.2) UserDaoImpl 实现类：

```
public class UserDaoImpl implements UserDao {
    @Override
    public void add() {
        System.out.println("添加用户。");
    }
    @Override
    public void update() {
        System.out.println("修改用户。");
    }
    @Override
    public void delete() {
        System.out.println("删除用户。");
    }
    @Override
    public void query() {
        System.out.println("查询用户。");
    }
}
```

4.1.3) UserDaoImplStaticProxy 代理类：

```
public class UserDaoImplStaticProxy implements UserDao{
    private UserDao userDao;
    public UserDaoImplStaticProxy(UserDao userDao) {
        this.userDao = userDao;
    }
    @Override
    public void add() {
        checkSecurity();
        userDao.add();
    }
    @Override
    public void update() {
        checkSecurity();
        userDao.update();
    }
    @Override
    public void delete() {
        userDao.delete();
    }
```

```
    @Override
    public void query() {
        userDao.query();
    }
    private void checkSecurity(){
        System.out.println("进行安全性检查。。。");
    }
}
```

4.1.3) 测试代码：

```
public class TestStaticProxy {
    @Test
    public void test01(){
        UserDao userDao = new UserDaoImpl();
        UserDaoImplStaticProxy userDaoImplStaticProxy =
        new UserDaoImplStaticProxy(userDao);
        userDaoImplStaticProxy.add();
    }
}
```

4.1.4) 运行效果：

```
进行安全性查询。。。
添加用户。
```

小结：静态代理需要代理类与目标类实际同样的接口，即如果想实现代理，则会多出一个与实现类（目标类）相似的类，这样，如果，程序中多处需要使用代理的话，就会多出许多这种多余的类，导致程序中类过多。

4.2）动态代理：（了解+掌握）

4.2.1）动态代理的代码实现：

```
//利用 JDK 的动态代理技术( Proxy.newProxyInstance)实现
public class UserDaoImplDynamicProxy implements InvocationHandler{
    //引入目标对象
    private UserDao userDao;
    public UserDaoImplDynamicProxy(UserDao userDao) {
        this.userDao = userDao;
    }
    //获取代理对象
    public UserDao getProxyObject(){
        //参数 1：获取当前目标类的类加载器对象
        //参数 2：获取目标对象所实现接口的 class 对象
        //参数 3：代表实现了 InvocationHandler 接口的对象，在此代表当前对象
        UserDao proxyObject = (UserDao) Proxy.newProxyInstance(
```

```
                                userDao.getClass().getClassLoader(),
                                userDao.getClass().getInterfaces(),
                                this);
        return proxyObject;
    }
    //调用目标对象的方法
    //参数 1：代理对象
    //参数 2：被调用的方法
    //参数 3：被调用方法的参数
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //在调用某个方法时检查安全性,如果调用的是 query（），就直接调用，不进行安全
性检查
        if(method.getName().equals("query")){
            //参数 1：代表目标对象
            //参数 2：代表传递给目标对象方法的参数
            return method.invoke(userDao, args);
        }
        checkSecurity();        //如果调用的不是 query 方法，而是诸如：
add(),update(),delete()这几个方法时
        return method.invoke(userDao, args);
    }
    //检查安全性
    private void checkSecurity(){
        System.out.println("检查安全性.");
    }
}
```

4.2.2）动态代理的测试：

```
//测试动态代理
    @Test
    public void test02(){
        UserDao userDao = new UserDaoImpl();
        UserDaoImplDynamicProxy udp = new
UserDaoImplDynamicProxy(userDao);
        UserDao proxyObject = udp.getProxyObject();

        proxyObject.add();
        System.out.println("--------------------------------");
        proxyObject.update();
        System.out.println("--------------------------------");
        proxyObject.delete();
        System.out.println("--------------------------------");
```

```
        proxyObject.query();
        System.out.println("--------------------------------");

        //检查代理对象与目标对象的关系：(没有关系，兄弟关系)
        System.out.println(proxyObject instanceof UserDaoImpl);
    }
```

4.2.3）测试结果：

```
检查安全性.
添加用户。
----------------------------------
检查安全性.
修改用户。
----------------------------------
检查安全性.
删除用户。
----------------------------------
查询用户。
----------------------------------
false
```

4.3）cglib 代理：(观光代码，了解)

4.3.1）cglib 代码实现：

```
//利用 CGLIB 的动态代理技术( 第三方代理，已经被 Spring 集成)继承实现
public class UserDaoImplCGLIBProxy implements MethodInterceptor{
    //引入目标对象

    //1.产生一个代理对象
    public UserDaoImpl getProxyObject(){
        //1.1) 得到 Enhancer 对象
        Enhancer enhancer = new Enhancer();
        //1.2) 设置 Enhancer 对象的父对象
        enhancer.setSuperclass(UserDaoImpl.class);
        //1.3) 设置 enhancer 对象要做的事情
        enhancer.setCallback(this);
        //1.4) 创建代理对象
        UserDaoImpl userDao = (UserDaoImpl) enhancer.create();
        //1.5) 返回代理对象
        return userDao;
    }
    @Override
```

```
    public Object intercept(Object proxy, Method method, Object[] args,
MethodProxy methodProxy)
            throws Throwable {
        //1.取得方法名
        if(method.getName().equals("query")){
            return  methodProxy.invokeSuper(proxy, args);
        }
         checkSecurity();      //检查安全性
        return methodProxy.invokeSuper(proxy, args);
    }
    //检查安全性
    private void checkSecurity(){
        System.out.println("检查安全性.");
    }
}
```

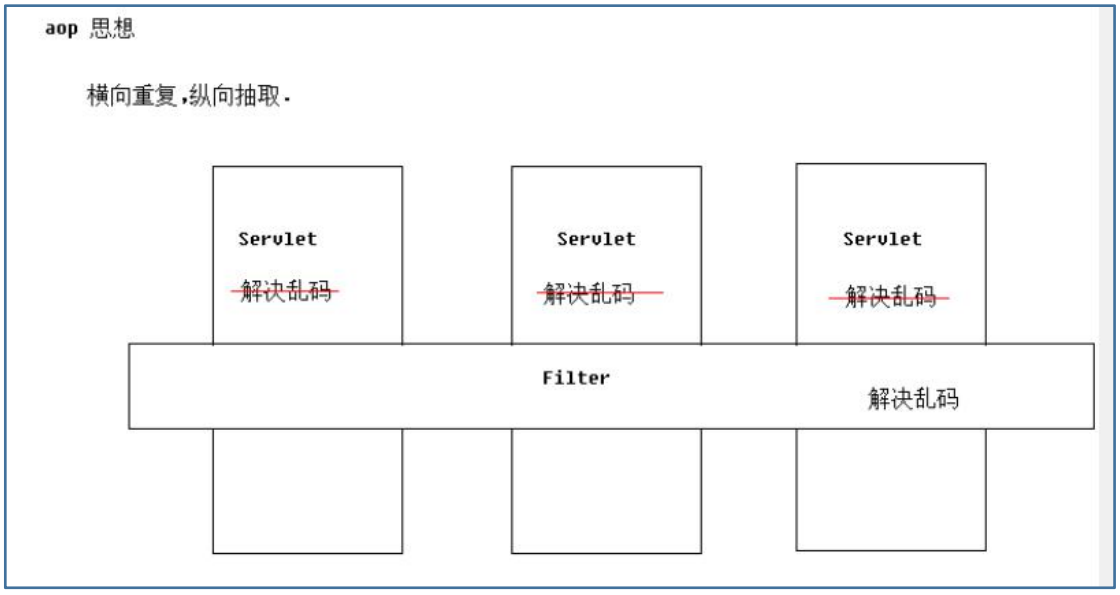
4.3.2）测试代码：

```
//测试 CGLIB 代理
    @Test
    public void test03(){
        UserDaoImplCGLIBProxy ucp = new UserDaoImplCGLIBProxy();
        //获取代理对象
        UserDao proxyObject = ucp.getProxyObject();
        //调用目标对象的方法
        proxyObject.add();
        System.out.println("--------------------------------");
        proxyObject.update();
        System.out.println("--------------------------------");
        proxyObject.delete();
        System.out.println("--------------------------------");
        proxyObject.query();
        System.out.println("--------------------------------");
        proxyObject.add();
    }
System.out.println(proxyObject instanceof UserDaoImpl);
```

4.3.3）运行结果：

```
检查安全性.
添加用户。
----------------------------------
检查安全性.
修改用户。
----------------------------------
检查安全性.
删除用户。
----------------------------------
查询用户。
----------------------------------
检查安全性.
添加用户。
true
```

4.4）Aop 思想介绍：



主要思想：横向重复，纵向抽取(形成一个新的组件如 Filter 或 Interceptor)。

4.5）Aop 名词学习：

Joinpoint(连接点)：目标对象中，所有可以增强的方法。

Pointcut(切入点)：目标对象，已经增强的方法。

Advice(通知/增强)：增强的代码

Target(目标对象)：被代理对象

Weaving(织入)：将通知应用到切入点的过程

Proxy（代理）：将通知织入到目标对象之后，形成代理对象

aspect(切面)：切入点+通知

4.6）实战 Aop：（重要）

第一种方式：使用 xml 方式完成 AOP 功能：（重要）

4.6.1）除了 4+2 包之外，另外，再导入 4 个包，如下：

4.6.2）准备目标对象：

```
//目标对象
public class UserDaoImpl implements UserDao {
    @Override
    public void add() {
        System.out.println("添加用户。");
    }
    @Override
    public void update() {
        System.out.println("修改用户。");
    }
    @Override
    public void delete() {
        System.out.println("删除用户。");
    }
    @Override
    public void query() {
        System.out.println("查询用户。");
    }
}
```

4.6.3）准备通知对象：

```
//定义通知
public class MyAdvice {
    /**
     * 前置通知(Before)：是调用方法之前调用
     * 后置通过(AfterReturning)：在调用方法之后调用（出现异常不调用 ）
     * 环绕通知(Around):在调用方法的前后，都会执行
     * 异常通知(After-Throwing):在方法调用出现异常时执行
     * 后置通知(After):无论是否出现异常都会调用
     */

    public void before(){
        System.out.println("前置通知.");
    }
    public void afterReturning(){
        System.out.println("后置通知，出现异常不调用.");
    }
    public Object around(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("环绕通知-前面代码");
        Object proceed = pjp.proceed();
        System.out.println("环绕通知-后面代码");
        return proceed;
```

```
    }
    public void afterThrowing(){
        System.out.println("不得了了，出了异常了！");
    }
    public void after(){
        System.out.println("无论是否出现异常，都会调用！");
    }
}
```

4.6.4）在 applicationContext3.xml 中进行通知、目标对象、切面的配置：

切记：引入 aop 命名空间。

```
<!-- 1、配置目标对象 -->
    <bean name="userDao" class="com.zelin.dao.impl.UserDaoImpl"/>
    <!-- 2.配置通知 -->
    <bean name="myadvice" class="com.zelin.proxy.MyAdvice"/>
    <aop:config>
        <!-- 3.配置切入点 -->
        <aop:pointcut expression="execution(*
com.zelin.dao.impl.*DaoImpl.*(..))" id="pointCut1"/>
        <!-- 4.配置切面 -->
        <aop:aspect ref="myadvice">
            <!-- 4.1) 配置前置通知 -->
            <aop:before method="before" pointcut-ref="pointCut1"/>
            <!-- 4.2) 配置后置通知 -->
            <aop:after-returning method="afterReturning"
pointcut-ref="pointCut1"/>
            <!-- 4.3) 配置环绕通知 -->
            <aop:around method="around" pointcut-ref="pointCut1"/>
            <!-- 4.4) 配置异常通知 -->
            <aop:after-throwing method="afterThrowing"
pointcut-ref="pointCut1"/>
            <!-- 4.5) 配置后置通知（出现异常还会执行） -->
            <aop:after method="after" pointcut-ref="pointCut1"/>
        </aop:aspect>
    </aop:config>
```

4.6.5）测试 Aop:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext3.xml")
public class TestAop {

    @Resource(name="userDao")
    private UserDao userDao;
    @Test
    public void test01(){
        userDao.add();

    }
}
```

4.6.6）运行效果如下：

```
log4j:WARN Please initialize the log4j system properly.
前置通知.
环绕通知-前面代码
添加用户。
无论是否出现异常，都会调用!
不得了了，出了异常了!
```

第二种方式：使用注解方式完成 AOP 功能：

4.6.1）在 applicationContext4.xml 文件中配置自动扫描及自动生成代理：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/contex
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/b
        http://www.springframework.org/schema/context http://ww
        http://www.springframework.org/schema/aop http://www.sp
    <!-- 1.配置自动扫描包 -->
    <context:component-scan base-package="com.zelin"/>
    <!-- 2.配置切面自动生成代理对象 -->
    <aop:aspectj-autoproxy/>
</beans>
```

4.6.2）利用注解的方式定义目标对象和通知对象：

定义目标对象：

```java
//目标对象
@Repository("userDao")
public class UserDaoImpl implements UserDao {
    @Override
    public void add() {
        System.out.println("添加用户。");
        int c = 10 / 0;
    }
```

定义通知对象：

```java
//定义切面
@Aspect
@Component
public class MyAdvice2 {
    /**
     * 前置通知(Before)：是调用方法之前，调用
     * 后置通过(AfterReturning)：在调用方法之后，调用（出现异常不调用 ）
     * 环绕通知(Around):在调用方法的前后，都会执行
     * 异常通知(After-Throwing):在方法调用出现异常时，执行
     * 后置通知(After):无论是否出现异常都会 调用
     */
    //下面利用一个空方法，将切入点抽取出来，后面调用时，直接使用
Myadvice2.pc()即可调用切入点表达式
    @Pointcut("execution(* com.zelin.dao.impl.*DaoImpl.*(..))")
    public void pc(){

    }
    @Before("MyAdvice2.pc()")
    public void before(){
        System.out.println("前置通知.");
    }
    @AfterReturning("MyAdvice2.pc()")
    public void afterReturning(){
        System.out.println("后置通知，出现异常不调用.");
    }
    @Around("MyAdvice2.pc()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("环绕通知-前面代码");
        Object proceed = pjp.proceed();
        System.out.println("环绕通知-后面代码");
        return proceed;
    }
    @AfterThrowing("MyAdvice2.pc()")
```

```
    public void afterThrowing(){
        System.out.println("不得了了，出了异常了！");
    }
    @After("MyAdvice2.pc()")
    public void after(){
        System.out.println("无论是否出现异常，都会调用！");
    }
}
```

4.6.3）测试注解方式 AOP 实现：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext4.xml")
public class TestAop2 {

    @Resource(name="userDao")
    private UserDao userDao;
    @Test
    //通过注解实现
    public void test01(){
        userDao.add();

    }

}
```

4.6.4）运行结果：

```
log4j:WARN Please initialize the log4j system properly.
环绕通知-前面代码
前置通知.
添加用户。
无论是否出现异常，都会调用!
不得了了，出了异常了!
```