## 1、<context:annotation-config> 和 <context:component-scan>的区别

<context:annotation-config> **是用于激活那些已经在spring容器里注册过的bean，也就是说在spring中添加好了bean，需要被resource**（无论是通过xml的方式还是通过package sanning的方式）上面的注解。

<context:component-scan>除了具有<context:annotation-config>的功能之外，<context:component-scan>还可以在指定的package下扫描以及注册javabean 。

## 2、事务管理器

|  |
| --- |
| <bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean> |

## 3、 <tx:annotation-driven 用于启动事物注解（加入事务管理器）

<tx:annotation-driven transaction-manager=*"transactionManager"* proxy-target-class=*"false"*/>

# 1、spring 开发环境选择

## 1、web.xml中配置选择哪种方式

|  |
| --- |
| <!-- 环境切换配置：production/develop/test -->  <context-param>  <param-name>spring.profiles.active</param-name>  <param-value>develop</param-value>  </context-param> |

## 2、spring 配置文件中添加两种数据源配置信息（这里是通过将数据源配置信息写到了properties文件进行，也可以在spring配置文件中写上多找了国内数据源）

<http://blog.csdn.net/hejingyuan6/article/details/51243985>

数据源定义为

**[html]** [view plain](http://blog.csdn.net/hejingyuan6/article/details/51243985) [copy](http://blog.csdn.net/hejingyuan6/article/details/51243985)

[print?](http://blog.csdn.net/hejingyuan6/article/details/51243985)

1. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
2. **<property** name="user" value="${jdbc.user}" **/>**
3. **<property** name="password" value="${jdbc.password}" **/>**
4. **<property** name="jdbcUrl" value="${jdbc.jdbcUrl}" **/>**
5. **<property** name="driverClass" value="${jdbc.driverClass}" **/>**
6. **<property** name="initialPoolSize" value="${c3p0.initialPoolSize}"**/>**
7. **<property** name="acquireIncrement" value="${c3p0.acquireIncrement}"**/>**
8. **<property** name="minPoolSize" value="${c3p0.minPoolSize}"**/>**
9. **<property** name="maxIdleTime" value="${c3p0.maxIdleTime}"**/>**
10. **<property** name="idleConnectionTestPeriod" value="${c3p0.idleConnectionTestPeriod}" **/>**
11. **<property** name="preferredTestQuery" value="${c3p0.preferredTestQuery}"**/>**
12. **</bean>**

classpath下外部资源文件有两个 settings-development.properties 和 settings-production.properties，分别是开发环境和生产环境的数据源配置参数，内容如下

settings-development.properties

**[html]** [view plain](http://blog.csdn.net/hejingyuan6/article/details/51243985) [copy](http://blog.csdn.net/hejingyuan6/article/details/51243985)

[print?](http://blog.csdn.net/hejingyuan6/article/details/51243985)

1. jdbc.user=root
2. jdbc.password=111111
3. jdbc.driverClass=com.mysql.jdbc.Driver
4. jdbc.jdbcUrl=jdbc:mysql://localhost:3306/xxx
5. c3p0.minPoolSize=5
6. c3p0.initialPoolSize=5
7. c3p0.acquireIncrement=5
8. c3p0.maxIdleTime=3600
9. c3p0.idleConnectionTestPeriod=3600
10. c3p0.preferredTestQuery=select 1

settings-production.properties

**[html]** [view plain](http://blog.csdn.net/hejingyuan6/article/details/51243985) [copy](http://blog.csdn.net/hejingyuan6/article/details/51243985)

[print?](http://blog.csdn.net/hejingyuan6/article/details/51243985)

1. jdbc.user=xxx
2. jdbc.password=xxxx
3. jdbc.driverClass=com.mysql.jdbc.Driver
4. jdbc.jdbcUrl=jdbc:mysql:///xxx
5. c3p0.minPoolSize=20
6. c3p0.initialPoolSize=20
7. c3p0.acquireIncrement=10
8. c3p0.maxIdleTime=3600
9. c3p0.idleConnectionTestPeriod=3600
10. c3p0.preferredTestQuery=select 1

#### 1. 定义 profile

现在就可以通过定义 profile 来将开发和生产环境的数据源配置分开，这里我们定义两个 profile，一个名称是 development，另一个名称是 production（配置到Spring的配置文件中即可）

**[html]** [view plain](http://blog.csdn.net/hejingyuan6/article/details/51243985) [copy](http://blog.csdn.net/hejingyuan6/article/details/51243985)

[print?](http://blog.csdn.net/hejingyuan6/article/details/51243985)

1. <!-- 开发环境配置文件 -->
2. **<beans** profile="development"**>**
3. **<context:property-placeholder** location="classpath:settings-development.properties"**/>**
4. **</beans>**
6. <!-- 生产环境配置文件 -->
7. **<beans** profile="production"**>**
8. **<context:property-placeholder** location="classpath:settings-production.properties"**/>**
9. **</beans>**

#### 2. 定义默认 profile

默认 profile 是指在没有任何 profile 被激活的情况下，默认 profile 内定义的内容将被使用，通常可以在 web.xml 中定义全局 servlet 上下文参数 spring.profiles.default 实现，代码如下

**[html]** [view plain](http://blog.csdn.net/hejingyuan6/article/details/51243985) [copy](http://blog.csdn.net/hejingyuan6/article/details/51243985)

[print?](http://blog.csdn.net/hejingyuan6/article/details/51243985)

1. <!-- 配置spring的默认profile -->
2. **<context-param>**
3. **<param-name>**spring.profiles.default**</param-name>**
4. **<param-value>**development**</param-value>**
5. **</context-param>**

当然此种方式也是使用默认profile的方式，即如果不指定激活哪个环境则使用默认方式加载文件

**[html]** [view plain](http://blog.csdn.net/hejingyuan6/article/details/51243985) [copy](http://blog.csdn.net/hejingyuan6/article/details/51243985)

[print?](http://blog.csdn.net/hejingyuan6/article/details/51243985)

1. **<beans** profile="default"**>**
2. **<context:property-placeholder** ignore-resource-not-found="true" location="classpath:application.properties"**/>**
3. **</beans>**

## 3、配置文件中写死 数据源信息

<http://blog.csdn.net/figo0423/article/details/54925993>

1. **<span** style="font-size:18px;"**>**  **<context-param>**
2. **<param-name>**spring.profiles.active**</param-name>**
3. **<param-value>**test**</param-value>**
4. **</context-param>**  **</span>**

             在spring xml配置文件这么配置

**[html]** [view plain](http://blog.csdn.net/figo0423/article/details/54925993) [copy](http://blog.csdn.net/figo0423/article/details/54925993)

1. **<span** style="font-size:18px;"**><?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:jee="http://www.springframework.org/schema/jee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:jpa="http://www.springframework.org/schema/data/jpa" xmlns:tx="http://www.springframework.org/schema/tx"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
7. http://www.springframework.org/schema/data/jpa
8. http://www.springframework.org/schema/data/jpa/spring-jpa-1.1.xsd
9. http://www.springframework.org/schema/tx
10. http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
11. http://www.springframework.org/schema/jee
12. http://www.springframework.org/schema/jee/spring-jee-3.1.xsd"**>**


16. **<bean** id="propertyConfigurer"
17. class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>**
18. **<property** name="location" value="classpath:database.properties" **/>**
19. **</bean>**

22. **<beans** profile="prod"**>**
23. **<jee:jndi-lookup** id="dataSource" jndi-name="${jpreTest.datasource}"
24. resource-ref="true" **/>**
25. **<jee:jndi-lookup** id="dataSource2" jndi-name="${jpreTest.datasource2}"
26. resource-ref="true" **/>**
27. **</beans>**
29. **<beans** profile="test"**>**
31. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
32. **<property** name="driverClass" value="${jpreTest.connection.driver\_class}" **/>**
33. **<property** name="jdbcUrl" value="${jpreTest.connection.url}" **/>**
34. **<property** name="user" value="${jpreTest.connection.username}" **/>**
35. **<property** name="password" value="${jpreTest.connection.password}" **/>**
36. <!-- 初始线程数 -->
37. **<property** name="initialPoolSize" value="5" **/>**
38. <!-- 最小线程数 -->
39. **<property** name="minPoolSize" value="${jpreTest.minPoolSize}" **/>**
40. <!-- 最大线程数 -->
41. **<property** name="maxPoolSize" value="${jpreTest.maxPoolSize}" **/>**
42. <!-- 空闲时间，单位秒 -->
43. **<property** name="maxIdleTime" value="1200" **/>**
44. <!-- 获取连接失败后该数据源将申明已断开并永久关闭 -->
45. **<property** name="breakAfterAcquireFailure" value="true" **/>**
46. <!--每60秒检查所有连接池中的空闲连接。Default: 0 -->
47. **<property** name="idleConnectionTestPeriod" value="60"**/>**
48. <!--如果设为true那么在取得连接的同时将校验连接的有效性。Default: false -->
49. **<property** name="testConnectionOnCheckin" value="true" **/>**
50. <!-- 链接池耗尽时一次获取的连接数 -->
51. **<property** name="acquireIncrement" value="5" **/>**
52. **</bean>**
54. **<bean** id="dataSource2" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
55. **<property** name="driverClass" value="${jpreTest.connection.driver\_class}" **/>**
56. **<property** name="jdbcUrl" value="${jpreTest.connection.url2}" **/>**
57. **<property** name="user" value="${jpreTest.connection.username2}" **/>**
58. **<property** name="password" value="${jpreTest.connection.password2}" **/>**
59. <!-- 初始线程数 -->
60. **<property** name="initialPoolSize" value="5" **/>**
61. <!-- 最小线程数 -->
62. **<property** name="minPoolSize" value="${jpreTest.minPoolSize}" **/>**
63. <!-- 最大线程数 -->
64. **<property** name="maxPoolSize" value="${jpreTest.maxPoolSize}" **/>**
65. <!-- 空闲时间，单位秒 -->
66. **<property** name="maxIdleTime" value="1200" **/>**
67. <!-- 获取连接失败后该数据源将申明已断开并永久关闭 -->
68. **<property** name="breakAfterAcquireFailure" value="true" **/>**
69. <!--每60秒检查所有连接池中的空闲连接。Default: 0 -->
70. **<property** name="idleConnectionTestPeriod" value="60"**/>**
71. <!--如果设为true那么在取得连接的同时将校验连接的有效性。Default: false -->
72. **<property** name="testConnectionOnCheckin" value="true" **/>**
73. <!-- 链接池耗尽时一次获取的连接数 -->
74. **<property** name="acquireIncrement" value="5" **/>**
75. **</bean>**
76. **</beans>**
77. **</beans>**
78. **</span>**

## 4、中科软配置这种方式

|  |
| --- |
| <beans profile=*"production"*>  <bean id=*"propertyConfigurer"* class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>  <property name=*"location"* value=*"classpath:jdbc.properties"* />  </bean>  </beans>  <beans profile=*"develop"*>  <bean id=*"propertyConfigurer"* class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>  <property name=*"location"* value=*"classpath:jdbc-develop.properties"* />  </bean>  <import resource=*"classpath:/module/applicationContext-simulate.xml"*/>  </beans>  <beans profile=*"test"*>  <bean id=*"propertyConfigurer"* class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>  <property name=*"location"* value=*"classpath:jdbc-test.properties"* />  </bean>  <import resource=*"classpath:/module/applicationContext-simulate.xml"*/>  </beans> |

## 5、同时导入多个properties

|  |
| --- |
| <!-- 引入properties配置文件 -->  <bean id=*"propertyConfigurer"*  class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>  <property name=*"locations"*>  <list>  <value>classpath:properties/\*.properties</value>  <!--要是有多个配置文件，只需在这里继续添加即可 -->  </list>  </property>  </bean> |

# 3、spring资源文件读取（一般都是class路径下的配置文件，也就是src或者和resource下的配置文件都是在编译后的WEB-INF/class路径下了）

## 1、src下有文件夹（包）module

|  |
| --- |
| ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("module/client-beans.xml"); |

//ApplicationContext ac2=new FileSystemXmlApplicationContext("src\\com\\hlj\\ioc\\beans.xml");

## 1.2从bean工厂容器中获取bean，但我们实例化这个容器的时候不会去实例化里面的bean，当我们使用容器中的bean时候，才会被实例化

### 1.测试

BeanFactory factory = **new** XmlBeanFactory(**new** ClassPathResource("com/hlj/ioc/beans.xml"));

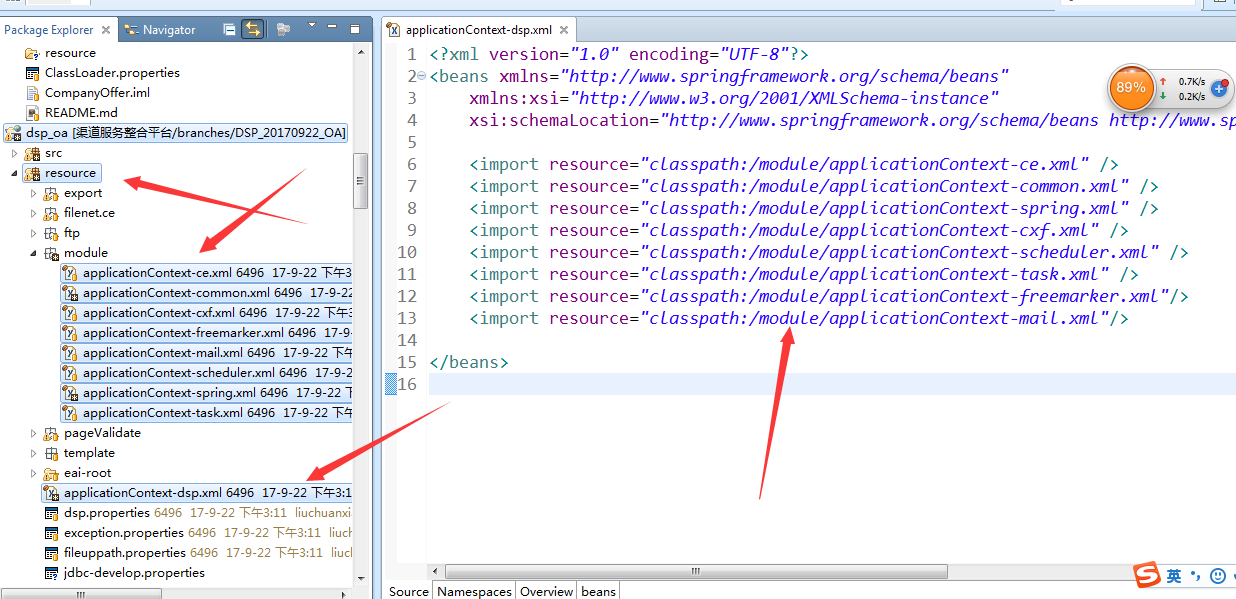
### 1.1输出结果 ：什么都没有

### 2.1如果加上使用这个容器

BeanFactory factory = **new** XmlBeanFactory(**new** ClassPathResource("com/hlj/ioc/beans.xml"));

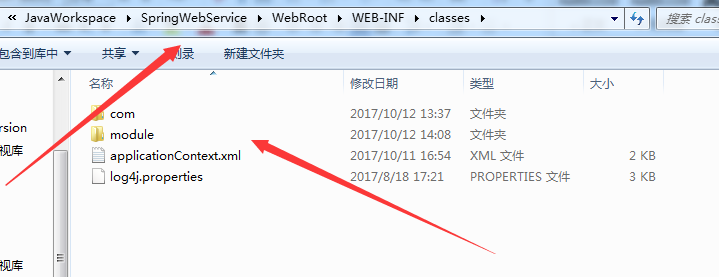
factory.getBean("student");

## 2、Resourse(中科软)

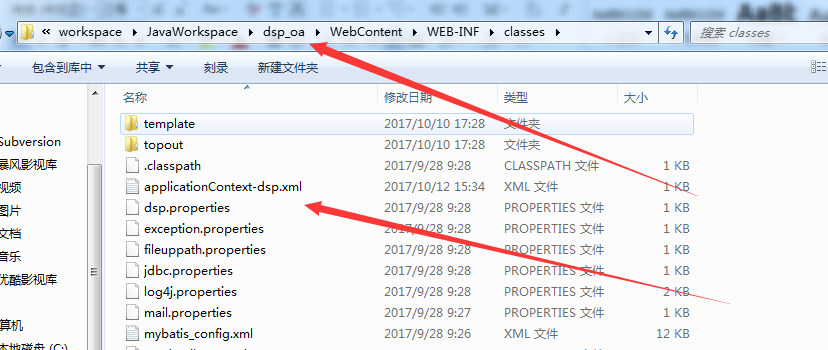


## 3、class路径下的文件

### 1、src class



### 2、resource 在class中



# 5、spring独立事物

## 1、接口文件

|  |
| --- |
| **public** **interface** ITransactionDao {    **public** **void** pessimisticLock(String lockInfo, Runnable runnable);  **public** <T> T pessimisticLock(String lockInfo, RunnableWithReturn<T> runnable);    /\*\*  \* 业务锁次数  \*/  **public** <T> T pessimisticLockForTime(**int** time, String lockInfo, RunnableWithReturn<T> runnable);    **public** <T> T independenceTrasaction(TransactionCallback<T> callback);    **public** **void** independenceTrasaction(TransactionCallbackWithoutResult callback);    **public** <T> T trasactionStrategy(**int** propagation, TransactionCallback<T> callback);    **public** **void** trasactionStrategy(**int** propagation, TransactionCallbackWithoutResult callback);    } |

## 2、实现类 *PROPAGATION\_REQUIRES\_NEW*,新建事务，如果当前存在事务，把当前事务挂起。

|  |
| --- |
| @Component  **public** **class** TransactionDaoImp **implements** ITransactionDao {    **private** **static** **int** *times* = 0;    @Autowired  **private** PlatformTransactionManager transactionManager;    @Autowired  **private** JdbcTemplate jdbcTemplate;    @Override  **public** **void** pessimisticLock(String lockInfo, Runnable runnable) {  Assert.*hasText*(lockInfo);    **boolean** acquired = **false**;  **try** {  acquired = acquireLock(lockInfo);  **if** (!acquired) {  **throw** **new** BusinessException("业务锁，请稍后执行!");  }  runnable.run();  } **finally** {  **if** (acquired) {  freeLock(lockInfo);  }  }  }  @Override  **public** <T> T pessimisticLock(String lockInfo, RunnableWithReturn<T> runnable) {  Assert.*hasText*(lockInfo);    **boolean** acquired = **false**;  **try** {  acquired = acquireLock(lockInfo);  **if** (!acquired) {  **throw** **new** BusinessException("业务锁，请稍后执行!");  }  **return** runnable.run();  } **finally** {  **if** (acquired) {  freeLock(lockInfo);  }  }  }    @Override  **public** <T> T pessimisticLockForTime(**int** time, String lockInfo, RunnableWithReturn<T> runnable) {  Assert.*hasText*(lockInfo);    **boolean** acquired = **false**;  **try** {  acquired = acquireLock(lockInfo);  **if** (!acquired) {  **throw** **new** BusinessException("业务锁，请稍后执行!");  }  **return** runnable.run();  } **catch** (Exception e) {  **try** {  Thread.*sleep*(3000);  } **catch** (Exception e1) {  **throw** **new** BusinessException("线程执行错误!");  }  **if**(*times* > time){  **throw** **new** BusinessException("业务锁，请稍后执行!");  }  *times* ++;  **return** pessimisticLockForTime(time, lockInfo, runnable);  } **finally** {  **if** (acquired) {  freeLock(lockInfo);  }  }  }    **private** **boolean** acquireLock(**final** String lockInfo) {  Boolean acquired = independenceTrasaction(**new** TransactionCallback<Boolean>(){  **public** Boolean doInTransaction(TransactionStatus status) {  **int** count = jdbcTemplate.queryForInt("select count(\*) from csip\_lock where pk\_lock = ?", lockInfo);  **if** (count > 0) {  **return** Boolean.*valueOf*(**false**);  }  jdbcTemplate.update("insert into csip\_lock (pk\_lock, ts) values(?, ?)", lockInfo, **new** Date());  **return** Boolean.*valueOf*(**true**);  }  });  **return** acquired.booleanValue();  }    **private** **void** freeLock(**final** String lockInfo) {  independenceTrasaction(**new** TransactionCallbackWithoutResult() {  @Override  **protected** **void** doInTransactionWithoutResult(TransactionStatus status) {  jdbcTemplate.update("delete from csip\_lock where pk\_lock = ?", lockInfo);  }  });  }  /\* 有返回值的独立事物 ，在受托主动发送信息的的时候用到 WsSender 类中\*/  @Override  **public** <T> T independenceTrasaction(TransactionCallback<T> callback) {  **return** trasactionStrategy(TransactionTemplate.*PROPAGATION\_REQUIRES\_NEW*, callback);  }  /\* 没有返回值的独立事物 \*/  @Override  **public** **void** independenceTrasaction(TransactionCallbackWithoutResult callback) {  trasactionStrategy(TransactionTemplate.*PROPAGATION\_REQUIRES\_NEW*, callback);  }    @Override  **public** <T> T trasactionStrategy(**int** propagation, TransactionCallback<T> callback) {  TransactionTemplate transactionTemplate = **new** TransactionTemplate(transactionManager);  // 设置事务传播属性  transactionTemplate.setPropagationBehavior(propagation);  **return** transactionTemplate.execute(callback);  }    @Override  **public** **void** trasactionStrategy(**int** propagation, TransactionCallbackWithoutResult callback) {  TransactionTemplate transactionTemplate = **new** TransactionTemplate(transactionManager);  // 设置事务传播属性  transactionTemplate.setPropagationBehavior(propagation);  transactionTemplate.execute(callback);  }    **public** JdbcTemplate getJdbcTemplate() {  **return** jdbcTemplate;  }  **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {  **this**.jdbcTemplate = jdbcTemplate;  }    } |

## 2.1、spring配置文件

|  |
| --- |
| <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*/>  </bean> |

## 3、有返回值的独立事物（中科软模拟发送数据，调用别人的接口，并取得返回值，。期间有日志的操作等。而这本身就是一个切面的事物。为了能够及时插入到数据库，这里需要一个需要独立事物在这个切面的事物内部。）

|  |
| --- |
| **private** WsResponse sendIndependenceTrasaction(**final** SendReasonVO reason, **final** JAXBElement<?> param) {  // 发送信息启用独立事物  **return** transactionDao.independenceTrasaction(**new** TransactionCallback<WsResponse>() {  @Override  **public** WsResponse doInTransaction(TransactionStatus status) {  **return** execute(reason, param, getWsDestInfo());  }  });  } |

## 4、没有返回值的独立事物（ 中科软，本例子使用是因为，这个insert外面还有一层事物，这里只有开启独立事物，才能直接导入到数据库，无需等待后续的数据库事物代码是否成功）

|  |
| --- |
| transactionDao.independenceTrasaction(**new** TransactionCallbackWithoutResult() {  @Override  **protected** **void** doInTransactionWithoutResult(  TransactionStatus arg0) {  tmpPlanOAVOMapper.insert(tmpPlanOAVO);  }  }); |

# 4、事物切面（也就是上面的中科软中上面3中独立事物之外的事物）

<http://www.cnblogs.com/rushoooooo/archive/2011/08/28/2155960.html>

## 1、spring配置（关于在*org.dicp.oa.\*.service.imp*包（包括包中包）下所有的服务类的方法加上事物（有些只只读的事物下方有配置），以第一次进入事物imp方法为事物的开启，不管中间调用imp其他方法多少次，都要以这个的结束，而事物终止。（这段期间，程序代码可以操作数据库，导入数据，但是我们直接利用第三方数据库查询工具是查询不到数据的。因为事物还没有提交）

|  |
| --- |
| <!-- 启用Spring对AspectJ的支持 -->  <aop:aspectj-autoproxy />  <!-- 事务切面 -->  - <aop:config>  <aop:pointcut id=*"servicePointcut"* expression=*"execution(public \* org.dicp.oa.\*.service.imp.\*.\*(..))"* />  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"servicePointcut"* />  </aop:config>  <!-- 事务通知 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <tx:method name=*"query\*"* read-only=*"true"* />  <tx:method name=*"get\*"* read-only=*"true"* />  <tx:method name=*"load\*"* read-only=*"true"* />  <tx:method name=*"count\*"* read-only=*"true"* />  <tx:method name=*"\*"* propagation=*"REQUIRED"* />  </tx:attributes>  </tx:advice> |

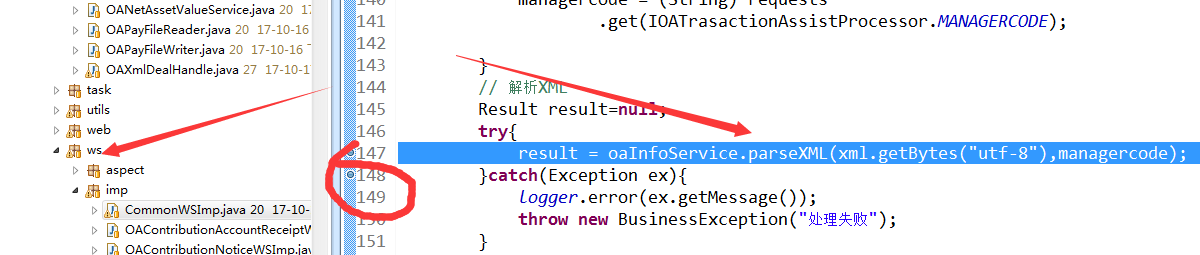
## 2、补充：而且上面的imp所在类下可以添加类似于@acpect的前置通知，后置通知等等。<http://blog.csdn.net/u011983531/article/details/70504281>

## 3、以中科软的模拟对方发送为例，当imp外的方法调用本

### 1、imp里面的方法



### 2、imp外面的方法，当这个外面的方法调用到149行的时候结束paserxml的事物



## 4、总结下

* < aop:aspect>：定义AOP通知器
* < aop:advisor>：定义切面

### **< aop:advisor>大多用于事务管理。** < aop:aspect>大多用于日志，缓存、

# 5、@Async 异步线程

## 1、spring 添加支持

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:task=*"http://www.springframework.org/schema/task"* xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-3.0.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-3.0.xsd*  *http://www.springframework.org/schema/task*  *http://www.springframework.org/schema/task/spring-task-3.0.xsd"*>  <!-- com.hlj.Tow.Async -->  <context:component-scan base-package=*"com.hlj.Tow"* />  <!--Spring 的配置文件中一定要配置这一项 ,它是用来激活 @Asysnc 还记得那个 激活事物注解吧，类似哦-->  <task:annotation-driven/>  </beans> |

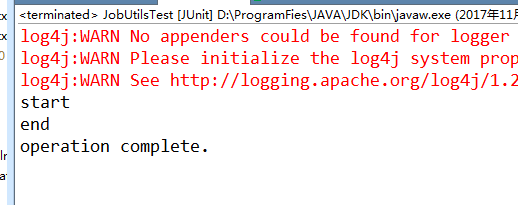
## 2、service中添加异步

|  |
| --- |
| @Service  **public** **class** DaoService {  @Async  **public** **void** update() {  **try** {  Thread.*sleep*(2000);  // do something  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("operation complete.");  }  } |

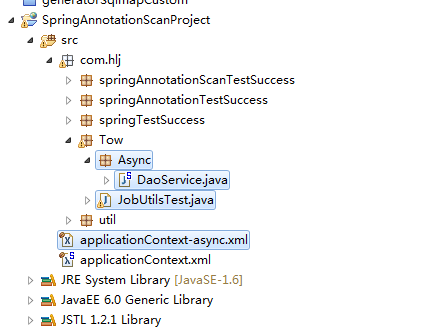
## 3、测试实例，观察运行顺序

|  |
| --- |
| **public** **class** JobUtilsTest {    @Test  **public** **void** testAsync() **throws** Exception {    ApplicationContext applicationContext = ApplicaionContextUtil.*getApplicationContext*();    DaoService service = (DaoService) applicationContext.getBean("daoService");    System.*out*.println("start" );  service.update(); // ★ 假设这个方法会比较耗时，需要异步执行  System.*out*.println("end");    //主线程testAsync 休眠  Thread.*sleep*(3000); // 因为junit结束会结束jvm，所以让它等会异步线程  }    } |

## 4、测试结果



## 5、代码



# 6、scheduled ，spring定时器， xml方式和注解方式

## 1、spring配置

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:task=*"http://www.springframework.org/schema/task"* xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-3.0.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-3.0.xsd*  *http://www.springframework.org/schema/task*  *http://www.springframework.org/schema/task/spring-task-3.0.xsd"*>  <!-- com.hlj.Tow.Async -->  <context:component-scan base-package=*"com.hlj.three.Schelule "*/>    <!-- 任务执行器 ,配置任务线性池  task:executor/@pool-size：可以指定执行线程池的初始大小、最大大小  task:executor/@queue-capacity：等待执行的任务队列的容量  task:executor/@rejection-policy：当等待队已满时的策略，分为丢弃、由任务执行器直接运行等方式  -->  <task:executor id=*"executor"*  keep-alive=*"3600"*  pool-size=*"100-200"*  queue-capacity=*"500"*  rejection-policy=*"CALLER\_RUNS"* />    <!--任务调度器 , -->  <task:scheduler id=*"scheduler"* pool-size=*"3"* />      <!-- 定时器开关 启用annotation方式 ,开启注解调度支持 @Async @schedule(不写参数也可以的)-->  <!-- <task:annotation-driven /> -->  <task:annotation-driven  scheduler=*"scheduler"*  executor=*"executor"*  proxy-target-class=*"true"* />    <bean id=*"myTaskXml"* class=*"com.hlj.three.Schelule.MyTaskXml"*></bean>  <task:scheduled-tasks scheduler=*"scheduler"*>  <!-- xml方式激活，这里表示的是每隔五秒执行一次 -->  <task:scheduled ref=*"myTaskXml"* method=*"show"* cron=*"\*/5 \* \* \* \* ?"* />  <task:scheduled ref=*"myTaskXml"* method=*"print"* cron=*"\*/10 \* \* \* \* ?"*/>  </task:scheduled-tasks>      </beans> |

## 2、xml激活定时器

|  |
| --- |
| **public** **class** MyTaskXml {  // 这里spring xml 中配置过，表示的是每隔五秒执行一次  **public** **void** show(){  System.*out*.println("XMl:is show run");  }  // 这里spring xml 中配置过这里表示的是每隔10秒执行一次  **public** **void** print(){  System.*out*.println("XMl:print run");  }  } |

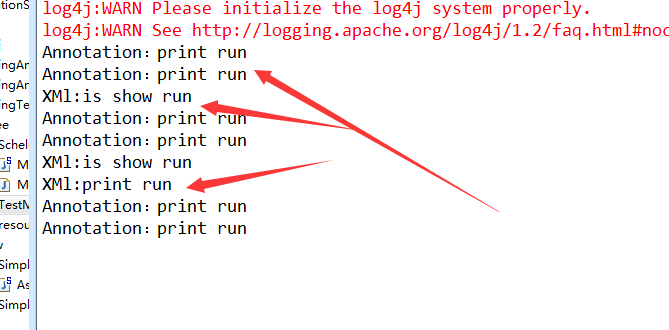
## 3、注解激活定时器

|  |
| --- |
| @Component  **public** **class** MyTaskAnnotation {    /\*\*  \* 定时计算。每天凌晨 01:00 执行一次  \*/  @Scheduled(cron = "0 0 1 \* \* \*")  **public** **void** show(){  System.*out*.println("Annotation：is show run");  }    /\*\*  \* 心跳更新。启动时执行一次，之后每隔2秒执行一次  \*/  @Scheduled(fixedRate = 1000\*2)  **public** **void** print(){  System.*out*.println("Annotation：print run");  }  } |

## 4、测试main，其实就是启动下spring 配置文件而已

|  |
| --- |
| **public** **class** TestMain {  **public** **static** **void** main(String[] args) {    ApplicationContext applicationContext = ApplicaionContextUtil.*getApplicationContext*();    }  } |

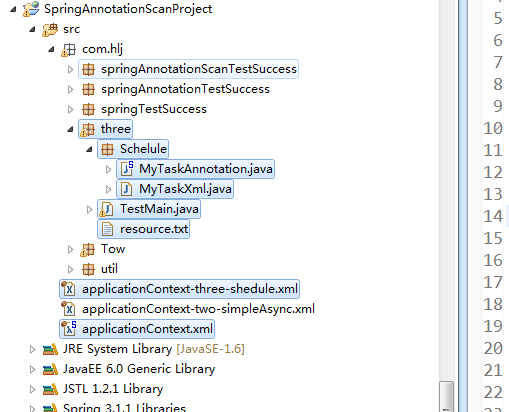
## 5、测试成功，有一个定时在，凌晨1点的没有执行



## 7、定时器时间配置

|  |
| --- |
| "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 |

## 8、代码位置



# 7、[@SuppressWarnings忽略警告](http://www.cnblogs.com/jingzhenhua/p/5986689.html)

## 解释：可以标注在类、字段、方法、参数、构造方法，以及局部变量上。

## 作用：告诉编译器忽略指定的警告，不用在编译完成后出现警告信息。

## ****使用：****

### @SuppressWarnings(“”)

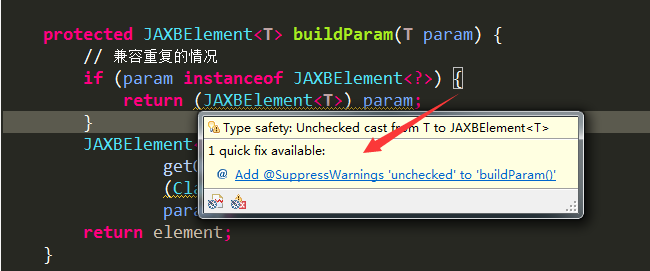
### @SuppressWarnings({})

### @SuppressWarnings(value={})

## 1、举例

### 1、@SuppressWarnings("unchecked")

#### 使用场景



## 2、抑制单类型警告

|  |
| --- |
| @SuppressWarnings("unchecked")  public void addItems(String item){  @SuppressWarnings("rawtypes")  List items = new ArrayList();  items.add(item);  } |

## 2、抑制多类型警告

|  |
| --- |
| @SuppressWarnings(value={"unchecked", "rawtypes"})  public void addItems(String item){  List items = new ArrayList();  items.add(item);  } |

## 3、抑制全部警告

|  |
| --- |
| @SuppressWarnings("all")  public void addItems(String item){  List items = new ArrayList();  items.add(item);  } |

## 抑制警告的关键字

|  |  |
| --- | --- |
| **关键字** | **用途** |
| all | to suppress all warnings |
| boxing | to suppress warnings relative to boxing/unboxing operations |
| cast | to suppress warnings relative to cast operations |
| dep-ann | to suppress warnings relative to deprecated annotation |
| deprecation | to suppress warnings relative to deprecation |
| fallthrough | to suppress warnings relative to missing breaks in switch statements |
| finally | to suppress warnings relative to finally block that don’t return |
| hiding | to suppress warnings relative to locals that hide variable |
| incomplete-switch | to suppress warnings relative to missing entries in a switch statement (enum case) |
| nls | to suppress warnings relative to non-nls string literals |
| null | to suppress warnings relative to null analysis |
| rawtypes | to suppress warnings relative to un-specific types when using generics on class params |
| restriction | to suppress warnings relative to usage of discouraged or forbidden references |
| serial | to suppress warnings relative to missing serialVersionUID field for a serializable class |
| static-access | o suppress warnings relative to incorrect static access |
| synthetic-access | to suppress warnings relative to unoptimized access from inner classes |
| unchecked | to suppress warnings relative to unchecked operations |
| unqualified-field-access | to suppress warnings relative to field access unqualified |
| unused | to suppress warnings relative to unused code |

# 8、spring生命周期

## 1、编辑service，让它实现下面的接口

**BeanNameAware,**

**BeanFactoryAware,**

**ApplicationContextAware,**

**InitializingBean,**

**DisposableBean**

|  |
| --- |
| **public** **class** PersonService **implements** BeanNameAware,BeanFactoryAware,ApplicationContextAware,InitializingBean,DisposableBean {    **private** String name;  **private** Integer age;  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  **public** String getName() {  **return** name;  }  **public** PersonService(){  System.*out*.println("1、无参构造函数 PersonService ");  }  **public** **void** setName(String name) {  System.*out*.println("2、这是一个set函数 setName(String name) ");  **this**.name = name;  }  //表示正在被实例化的id是多少 也就是bean的名字 本类名字 PersonService  **public** **void** setBeanName(String arg0) {  // **TODO** Auto-generated method stub  System.*out*.println("3、调用BeanNameAware中的setBeanName String正在被实例化的id是 "+arg0);  //  }  @Override  //这个bean的工厂是，不止是这个bean ，凡是被实例化的都会显示  **public** **void** setBeanFactory(BeanFactory arg0) **throws** BeansException {  // **TODO** Auto-generated method stub  System.*out*.println("4、调用 BeanFactoryAware 中的setBeanFactory----这个bean工厂包含 "+arg0);  //4、这个bean需要的工厂是  //org.springframework.beans.factory.xml.XmlBeanFactory@fcfa52:  //defining beans [personService,personService2,myBeanPostProcessor];  //root of factory hierarchy  }  //返回的上下文  @Override  **public** **void** setApplicationContext(ApplicationContext arg0)  **throws** BeansException {  System.*out*.println("5、调用ApplicationContextAware 的setApplicationContext ---这个项目的上下文 "+arg0);  //这个项目的上下文  // org.springframework.context.support.ClassPathXmlApplicationContext@2be9cb75:  // display name [org.springframework.context.support.ClassPathXmlApplicationContext@2be9cb75];  // startup date [Mon Nov 20 18:14:39 CST 2017]; root of context hierarchy  }  @Override  //在设置属性之后，执行，相当于是后置处理器MyBeanPostProcessor处理之后，没有立刻执行处理器的after方法，而是执行下面这个方法  **public** **void** afterPropertiesSet() **throws** Exception {  // **TODO** Auto-generated method stub  System.*out*.println("7、调用 InitializingBean中的afterPropertiesSet的方法");  }  //自己定义的中间的方法，执行完成这个，才回去执行后置处理器的after方法  **public** **void** zhangInit(){  System.*out*.println("8、方法 init-method='zhangInit'");  }    **public** **void** sayHi(){  System.*out*.println("10、hi "+ name);  }  /\*\*  \* 下面的应该在项目死掉的时候依次执行 destroy mydestory  \*/  @Override  **public** **void** destroy() **throws** Exception {  // **TODO** Auto-generated method stub  System.*out*.println("这个是接口定义的 销毁方法也许看不到");  }  //自己定义的  **public** **void** mydestory(){  System.*out*.println("final 这个是我自己的销毁方法 释放各种资源，也许看不到");  }  } |

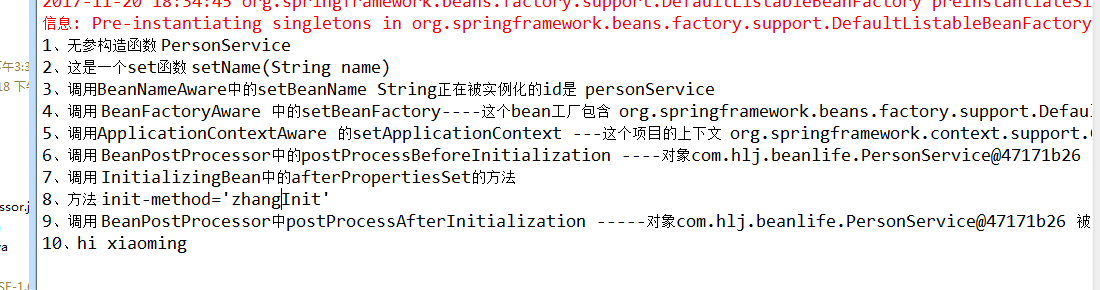
## 2、后置处理器如下

|  |
| --- |
| /\*\*  \* ,setApplicationContenxt完成之后开始调用  配置自己的后置处理器让，让其它bean都是单例singoton, \*  \*/  **public** **class** MyBeanPostProcessor **implements** BeanPostProcessor {    //这个先运行  **public** Object postProcessBeforeInitialization(Object arg0, String arg1)  **throws** BeansException {  // **TODO** Auto-generated method stub  System.*out*.printf("6、调用 BeanPostProcessor中的postProcessBeforeInitialization ----");  System.*out*.println("对象"+arg0+" 被创建的时间是 "+**new** java.util.Date());  **return** arg0;  }    **public** Object postProcessAfterInitialization(Object arg0, String arg1)  **throws** BeansException {  System.*out*.printf("9、调用 BeanPostProcessor中postProcessAfterInitialization -----");  System.*out*.println("对象"+arg0+" 被创建的时间是 "+**new** java.util.Date());  **return** arg0;  } |

## 3、spring配置文件

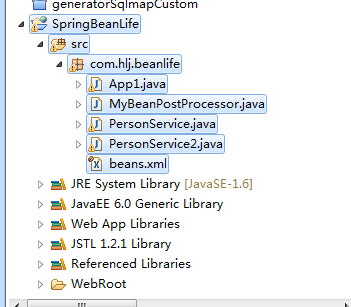
|  |
| --- |
| <?xml version=*"1.0"* encoding=*"utf-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd*  *http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd"*>  <bean id=*"personService"* init-method=*"zhangInit"* destroy-method=*"mydestory"* class=*"com.hlj.beanlife.PersonService"*>  <!-- 这里注入我们属性，前提就是有setName才能ok destroy-method="mydestory" -->  <property name=*"name"*>  <value>xiaoming</value>  </property>  </bean>  <bean id=*"personService2"* scope=*"prototype"* class=*"com.hlj.beanlife.PersonService2"*>  </bean>    <!-- 配置我们的自己后置处理器(有点类似我们的filter) -->  <bean id=*"myBeanPostProcessor"* class=*"com.hlj.beanlife.MyBeanPostProcessor"* />  </beans> |

## 4、测试开始





## 5、代码



# 9、Spring原理

## 1、spring 的3大组件，骨架

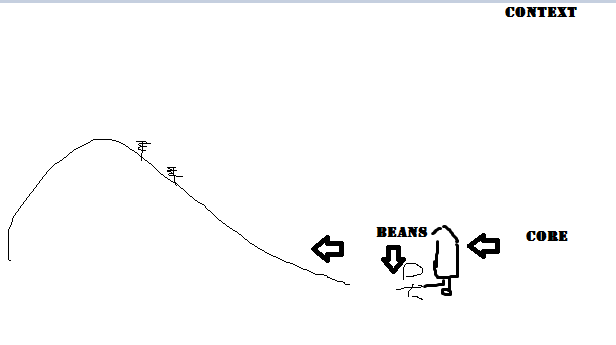
spring 的骨架，也是spring 的核心包。主要包含三个内容

1.context：spring 的上线文-------**导演**

2.core：spring的核心包，主要包括spring所以用到的工具-------**道具**

3.beans：spring的bean实例 -------**演员**

core包侧重于帮助类，操作工具，beans包更侧重于bean实例的描述。context更侧重全局控制，功能衍生。

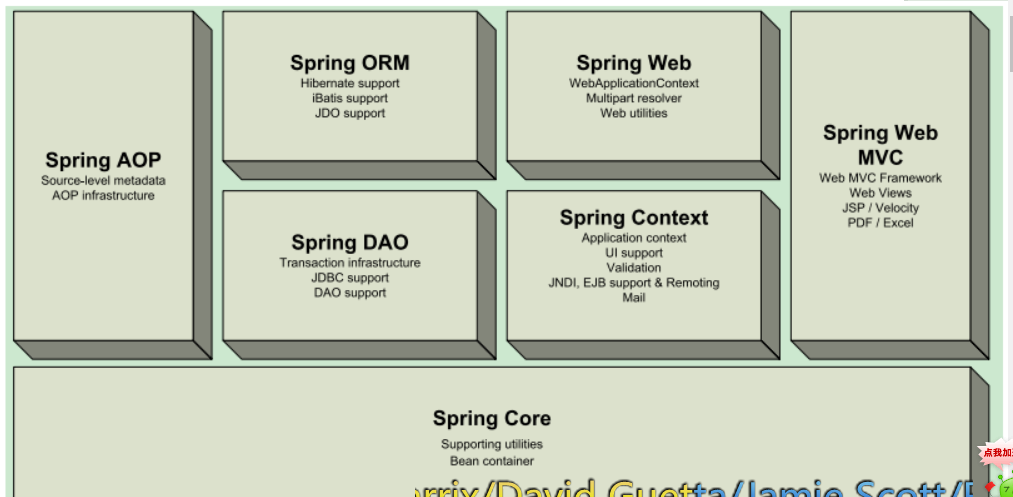


导演负责安排演出，演员负责按照导演的指示来演出，演出过程中需要使用道具。

我想大家看完这些图片之后就明白大致的包关系了。



## 2、spring7大模块



**Core包**是框架的最基础部分，并提供依赖注入（Dependency Injection）管理Bean容器功能。

**Context包**，构建于Core包上，提供了一种框架式访问对象的方式，有些像JNDI注册。Context封装包继承了beans包的功能，还增加了国际化（I18N）,事件传播，资源装载，以及透明创建上下文，例如通过servlet容器。

**DAO包**提供了JDBC的抽象层，它可消除冗长的JDBC编码和解析数据库厂商特有的错误代码。并且，JDBC封装包还提供了一种比编程性更好的声明性事务管理方法，不仅仅是实现了特定接口，而且对所有的POJOs都适用。

**ORM包**为流行的“关系/对象”映射APIs提供了集成层，包括JDO，Hibernate和iBatis。通过ORM包，可以混合使用所有Spring提供的特性进行“对象/关系”映射，如前边提到的简单声明性事务管理。

**AOP包**提供了符合AOP Alliance规范的面向方面的编程实现，例如方法拦截器

（method-interceptors）和切点（pointcuts），从逻辑上讲，从而减弱代码的功能耦合，清晰的被分离开。

**Web包**提供了基本的面向Web的综合特性，例如多方文件上传，利用Servlet listeners进行IoC容器初始化和针对Web的applicationcontext。当与WebWork或Struts一起使用Spring时，这个包使Spring可与其他框架结合。

**Web MVC**提供了面向Web应用的Model-View-Controller实现。Spring的MVC框架并不是仅仅提供一种传统的实现，它提供了一种清晰的分离模型，在领域模型代码和web form之间。并且，还可以借助Spring框架的其他特性。

## 3、spring 提供的jar包

自3.0以后包结构就有所变化，将每个模块独立出一个包来，分工明确，用户可以根据不同的需求自行组装Jar，而不是像以前一样将整个Spring.jar引入项目中，下面以3.2M1为例，简单介绍一下Spring的包结构：

**spring-core.jar**：包含Spring框架基本的核心工具类，Spring其它组件要都要使用到这个包里的类，是其它组件的基本核心；

**spring-aop.jar**：面向切面编程，提供AOP（面向切面编程）实现；

**spring-aspects.jar**：spring整合了aspectjrt.jar，也可以使用aspectjrt.jar来代替；

**spring-beans.jar**：springIoC（依赖注入）的基础实现，所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行Inversion of Control / DependencyInjection（IoC/DI）操作相关的所有类；

**spring-context.jar**：spring提供在基础 IoC 功能上的扩展服务，此外还提供许多企业级服务的支持，如 邮件服务、任务调度、JNDI定位、EJB集成、远程访问、 缓存以及各种视图层框架的封装等；

**spring-context-support.jar**：spring-context的扩展支持，包含支持缓存Cache（ehcache）、JCA、JMX、邮件服务（JavaMail、COS Mail）、任务计划Scheduling（Timer、Quartz）方面的类。

**spring-asm.jar**：ASM是小巧轻便的 Java 字节码操控框架 ，它能方便地生成和改造 Java 代码。 Hibernate和 Spring在底层都用到了 ASM(CgLib底层用的 asm)；

**spring-expression.jar**：spring表达式语言；

**spring-instrument.jar**：spring3.1对服务器的代理接口

**spring-instrument-tomcat.jar**：spring3.1对 Tomcat 的连接池的集成；

**spring-jdbc.jar**：包含对Spring对JDBC数据访问进行封装的所有类；

**spring-jms.jar**：提供了对JMS1.0.2/1.1的支持类；

**spring-orm.jar**：包含Spring对DAO特性集进行了扩展，使其支持 iBATIS、JDO、OJB、TopLink，因为Hibernate已经独立成包了，现在不包含在这个包里了。这个jar文件里大部分的类都要依赖spring-dao.jar里的类，用这个包时你需要同时包含spring-dao.jar包；

**spring-oxm.jar**：spring对Object/XMI的映射的支持，可以让JAVA与XML之间来回切换；

**spring-struts.jar**：整合Struts的支持；

**spring-test.jar**：spring对Junit等测试框架的简单封装；

**Spring-tx.jar**：提供对事务的支持；

**spring-web.jar**：包含Web应用开发时，用到Spring框架时所需的核心类，包括自动载入WebApplicationContext特性的类、Struts与JSF集成类、文件上传的支持类、Filter类和大量工具辅助类；

**Spring-webmvc.jar**：这个jar文件包含Spring MVC框架相关的所有类。包含国际化、标签、Theme、视图展现的FreeMarker、JasperReports、Tiles、Velocity、 XSLT相关类。当然，如果你的应用使用了独立的MVC框架，则无需这个JAR文件里的任何类；

**spring-webmvc-portlet.jar**：springMVC的增强，支持portlet标准（JSR168/JSR286）；

## 4、spring aop原理

<http://blog.csdn.net/moreevan/article/details/11977115/>

### 1、透彻理解aop，马勒巴子，写的非常好

AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。

OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。

例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。

这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。

所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。

业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。

横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。

Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如Avanade公司的高级方案构架师Adam Magee所说，AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

### 2、实现aop的技术

实现AOP的技术，主要分为两大类：

一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

### 3、Spring AOP代理对象的生成

Spring提供了两种方式来生成代理对象: JDKProxy和Cglib，

具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。

下面我们来研究一下Spring如何使用JDK来生成代理对象，具体的生成代码放在JdkDynamicAopProxy这个类中，直接上相关代码：

|  |
| --- |
| 1. /\*\* 2. \* <ol> 3. \* <li>获取代理类要实现的接口,除了Advised对象中配置的,还会加上SpringProxy, Advised(opaque=false) 4. \* <li>检查上面得到的接口中有没有定义 equals或者hashcode的接口 5. \* <li>调用Proxy.newProxyInstance创建代理对象 6. \* </ol> 7. \*/ 8. **public** Object getProxy(ClassLoader classLoader) { 9. **if** (logger.isDebugEnabled()) { 10. logger.debug("Creating JDK dynamic proxy: target source is " +**this**.advised.getTargetSource()); 11. } 12. Class[] proxiedInterfaces =AopProxyUtils.completeProxiedInterfaces(**this**.advised); 13. findDefinedEqualsAndHashCodeMethods(proxiedInterfaces); 14. **return** Proxy.newProxyInstance(classLoader, proxiedInterfaces, **this**); 15. } |

# 10、Spring加载resource时classpath\*:与classpath:的区别

Spring可以通过指定classpath\*:与classpath:前缀加路径的方式从classpath加载文件,如bean的定义文件.

### 1、classpath\*:的出现是为了从多个jar文件中加载相同的文件.

### 2、lasspath:只能加载找到的第一个文件.

## 3、举例说明

resource1.jar中的package 'com.test.rs' 有一个 'jarAppcontext.xml' 文件,内容如下:

<bean name="ProcessorImplA" class="com.test.spring.di.ProcessorImplA" />

resource2.jar中的package 'com.test.rs' 也有一个 'jarAppcontext.xml' 文件,内容如下:

<bean id="ProcessorImplB" class="com.test.spring.di.ProcessorImplB" />

通过使用下面的代码则可以将两个jar包中的文件都加载进来

ApplicationContext ctx = new ClassPathXmlApplicationContext( "classpath\*:com/test/rs/jarAppcontext.xml");

而如果写成下面的代码,就只能找到其中的一个xml文件(顺序取决于jar包的加载顺序)

ApplicationContext ctx = new ClassPathXmlApplicationContext( "classpath:com/test/rs/jarAppcontext.xml");

## 4、原理

classpath\*:的使用是为了多个component(最终发布成不同的jar包)并行开发,各自的bean定义文件按照一定的规则:package+filename,而使用这些component的调用者可以把这些文件都加载进来.

classpath\*:的加载使用了classloader的 getResources() 方法,如果是在不同的J2EE服务器上运行,由于应用服务器提供自己的classloader实现，它们在处理jar文件时的行为也许会有所不同。 要测试classpath\*: 是否有效，可以用classloader从classpath中的jar文件里加载文件来进行测试：getClass().getClassLoader().getResources("<someFileInsideTheJar>")。(上面的例子是在sun的jre中运行的状态)

## 5、项目实战，多点

|  |
| --- |
| @ImportResource(value = **"classpath:applicationContext.xml"**) |

|  |
| --- |
| applicationContext.getResources(  **"classpath\*:com/duodian/admore/dao/mybatis/\*\*/mysql/\*.xml"**)); |