# Spring4入门

1. **Spring简介 ：核心IOC、AOP**

* **ID**（dependency injection）：依赖注入，另一种IOC表达方式，本质上是IOC。只是组件预先定义好方式接收来自容器的资源注入（eg：setter）。
* **IOC**（inversion of control）：反转资源获取方向，容器主动将资源推送给组件，组件只需要接收资源即可。因此，组件不需要直到其他资源的具体方法，减小耦合程度。
* **AOP**（aspectoriented programming）：面向切面编程
* 整合IOC、AIP，提供SpringMVC、Spring JDBC
* 使用XML、Java注解组合对象

# IOC（DI）

1. **使用xml中的bean配置**

* **用bean配置属性：id、class属性(使用反射机制，实例化对象)**
* 使用setter方法注入属性值

**<!—使用setter注入属性值 -->**  
<bean **id**="helloWorld" **class**="com.HelloWorld">  
  **<!—调用类中的name（方法名）=setName01方法，参数值：chris -->**  
 <property name="name01" value="chris"></property>  
</bean>

* 使用构造器注入属性值

**<!-- 使用构造器注入属性值：  
 当有多个构造器时：使用valus（参数值）、index（参数位置）、type（参数类型）来定位使用哪个构造器**

**name（方法名，去除set关键字）**

**value：若存在特殊字符，使用CDATA，<value><![CDATA[<chris@]]></value>  
 -->**  
<bean **id** = "**user01**" **class**="HelloWorld.UserInfo">  
 **<!-- 调用构造器：初始化属性 -->**  
 <constructor-arg **value**="1" **index**="0"></constructor-arg>  
 <constructor-arg **value**="Chris" **index**="1"></constructor-arg>  
 <constructor-arg **value**="25" **type**="**int**"></constructor-arg>  
  **<!-- 调用setter方法：setId、setAge、setName -->**  
 <property **name**="id" **value**="1"></property>  
 <property **name**="age" **value**="20"></property>  
 <property **name**="name" **value**="Chris"></property>  
</bean>

* **调用：**

public static void main(String[] args) {  
  **//2.1、创建Spring的IOC容器对象**  
 **ApplicationContext** applicationContext = new **ClassPathXmlApplicationContext**("applicationContext.xml");  
  **//2.2、根据id：从IOC容器中获取Bean实例，需要id唯一**  
 HelloWorld helloWorld1 = (HelloWorld)applicationContext.getBean("hello01");  
 UserInfo userInfo = **(UserInfo)applicationContext.getBean("user01");**  
 **//2.2、根据类型：从IOC容器中获取Bean实例，只有IOC容器中，每个类仅存在唯一一个class类型，才不会出错**  
 **//HelloWorld helloWorld1 = applicationContext.getBean(HelloWorld.class);**  
  
  **//3、调用方法**  
 helloWorld1.getName();  
 System.*out*.println(userInfo);  
 }

* **Bean中引用其他Bean**
* 使用ref：指定当前属性，使用哪个bean进行赋值
* 示例代码：

<bean id = "**car01**" class="BeanRef.Car">  
 <constructor-arg value="VW" index="0"></constructor-arg>  
</bean>  
<bean id = "**car02**" class="BeanRef.Car">  
 <constructor-arg value="AG" index="0"></constructor-arg>  
</bean>  
<bean id="**user02**" class="BeanRef.UserInfo">  
 <property name="name" value="Alice"></property>  
 <property name="cars">  
  **<!-- 使用List配置集合属性cars（cars是一个List） -->**  
  **<list>**  
  **<!-- 使用ref引用其他bean，装配到cars中 -->**  
 <**ref** bean="**car01**"></ref>  
 <**ref** bean="**car02**"></ref>  
  **</list>**  
 </property>  
</bean>

* **使用p标签配置属性：**简化步骤

<bean id="user02" class="AutoBean.User"  
 p:name="Chris" p:age="20"  
></bean>

* **自动装配：不常使用，局限性太大**
* 根据方法的名字匹配：**byName**，方法名和被引用的bean的id必须一致，否则完成自动匹配。
* 根据方法的类型匹配：**byType**，当同一个.xml中存在多个相同的class（类）时，会出现错误。

**如： <bean id = "car01" class="BeanRef.Car">，仅有一个class="BeanRef.Car"**

* **示例代码：**

**<!-- 根据名字匹配:byName -->**  
<bean id="user01" class="AutoBean.User"  
 p:name="Chris" p:age="20" autowire="**byName**"  
></bean>  
  
**<!-- 根据类型匹配:byType -->**  
<bean id="user02" class="AutoBean.User"  
 p:name="Chris" p:age="20" autowire="**byType**"  
></bean>

* **bean的继承：**可用于制作bean模板
* 使用parent继承：子类回继承父bean的配置参数值
* bean模板：抽象bean，没有class属性，使用abstract属性
* 示例代码：

**<!-- 使用P标签  
 对元素进行赋值:cars-ref表示引用car的bean集合  
 -->**  
<bean id="**user05**" class="BeanRef.UserInfo"  
  **p:cars-ref**="cars" p:name="Tian" p:age="20"  
></bean>  
  
**<!-- bean继承  
使用p + parent  
-->**  
<bean id="user06" **parent="user05"**  
 p:name="FYJ"  
></bean>

* **bean作用域**
* 默认是单例：创建多个id相同的bean对象，都是同一个对象
* 修改：使用scope修改
* 示例代码：

**<!-- 修改作用范围:单例,多例模式  
使用scope修改:默认是单例  
 -->**

<bean id="user07" class="BeanRef.UserInfo"  
  **scope="prototype"**  
 p:cars-ref="cars" p:name="Tian" p:age="20"  
></bean>

* **使用外部文件，配置、连接Mysql**
* **需要的库文件：c3p0（spring中使用）、mysql-connector-java8.0**
* 需要的文件：db.properties（存储mysql参数）

jdbc-bean.xml（配置bean，实例化c3p0对象）

* **示例代码;**

**jdbc-bean.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.xsd">  
 **<!-- 错误;通配符的匹配很全面, 但无法找到元素。  
 因为xsi:schemaLocation = 后面没加http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd  
 需要添加：xmlns:context="http://www.springframework.org/schema/context"  
 -->**  
  
 **<!-- 导入jdbc的配置文件 -->**  
 <context:property-placeholder location="classpath:db.properties"/>  
  **<!-- 使用db.properties文件,配置JDBC数据库 -->**  
 <bean id="mysqlConf" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
  **<!-- 使用db.properties文件的变量 -->**  
 <property name="user" value=**"${mysqlUser}"**></property>  
 <property name="password" value=**"${mysqlPasswd}"**></property>  
 <property name="driverClass" value=**"${mysqlDriver}"**></property>  
 <property name="jdbcUrl" value=**"${mysqlURL}"**></property>  
 </bean>  
  
</beans>

**db.properties文件：**存储参数，方便后期修改

**mysqlDriver**:com.mysql.cj.jdbc.Driver  
**mysqlURL**:jdbc:mysql://localhost:3306/jdbctest?serverTimezone=UTC  
**mysqlUser**:root  
**mysqlPasswd**:1230re0321re

**main文件：**使用方法

public static void main(String[] args) throws SQLException {

**//创建IOC容器**  
 ApplicationContext context = new ClassPathXmlApplicationContext("jdbc-bean.xml");

**//从IOC容器中，实例化bean对象**  
 DataSource dataSource = (DataSource) context.getBean("mysqlConf");  
  **//调用方法**  
 System.*out*.println(dataSource.getConnection());  
}

1. **Spring表达语句：**SpEL

* 用于引用参数，可以写字面量：#{5}、#{chris}、#{true}、…
* 引用其他对象：#{sequenceGeneratorr2.toString()}、…
* 算术运算：#{a+b}
* 字符串连接：#{name + “ ”+ age}
* eg：

<property name=”equal” valus=”**#{counter.total == 100}**”/>

1. **IOC生命周期：**

* **bean完整的生命周期：**
* 构造器、工厂方法创建Bean实例
* 对Bean属性赋值、引用
* 将Bean实例传给Bean后置处理器的

postprocessBeforeInitialization方法

* 调用自定义的Bean初始化方法：init（）
* 将Bean实例传给Bean后置处理器的

postprocessAfterInitialization方法

* 自此，完成Bean初始化，可以使用了
* 关闭容器时，调用自定义的Bean销毁方法：destroy（）
* **示例代码：**

<bean id="car" class="BeanPostProcessorTest.Car"  
  **init-method**="init"  
  **destroy-method**="destroy">  
 <property name="band" value="Audio"></property>  
 <property name="company" value="VW"></property>  
</bean>  
  
**<!-- 配置bean的后置处理器 -->**  
<bean class="**BeanPostProcessorTest.BeanPostProcessorTest**"></bean>

1. **用工厂方法配置Bean**

* 静态工厂方法配置bean

将创建对象的过程，封装到静态方法中

在没创建对象的时候，通过某个静态方法得到对应的实例对象。

**<!-- 使用静态工厂方法配置bean  
 calss：静态工厂方法的全类名  
 factory-method：静态工厂方法的名字  
 constructor-arg：用于传入factory-method静态工厂方法的参数  
 -->**  
<bean id="cars01" class="FactoryBean.StaticCarFactory"  
 **factory-method**="getCar">  
 <**constructor-arg** **value**="Audio"></constructor-arg>  
</bean>

* 实例工厂方法配置bean

将创建对象的过程，封装到某个实例对象的方法中

**<!-- 使用实例工厂方法配置bean  
 factory-bean: 指向实例工厂方法的bean  
 -->**  
<bean id="carsInstanceFactory" class="FactoryBean.InstanceCarfactory"></bean>  
<bean id="cars02" **factory-bean**="carsInstanceFactory"  
  **factory-method**="getCar">  
 <**constructor-arg value**="BMW"></constructor-arg>  
</bean>

1. **使用FactoryBean配置Bean**

IOC容器中，很多方法都是以通过FactoryBean配置，因此使用FactoryBean来配置较为合适。这个FactoryBean和工厂方法配置的Bean不同，FactoryBean返回的对象是该工厂的Bean的getObject方法返回的对象。

相应的类需要继承FactoryBean，并重写对应的方法

（上面所有的Bean配置方式：都是使用xml配置）

**配置Bean有两种方式：（1）xml方式配置；（2）注解方式配置**

1. **使用注解的方式配置Bean**

* **组件扫描：**component scanning

Spring从classpath下自动扫描特定注解的组件；

* **特定注解：**
* @Component：基本注解，表示该组件受Spring管理
* @Respository：表示持久层组件
* @Service：表示服务层组件
* @Controller：表现层组件
* 命名示例：UserServicelmpl userService （原名 + 现名）
* 示例代码：

**//持久层：指定名字为—user,原名(默认名)：userRepositoryImpl**  
**@Repository**(**value** = "**user**")  
public class UserRepositoryImpl implements UserRepository {  
 @Override  
 public void save() {  
 System.*out*.println("UserRepository Save ...");  
 }  
}

* **组件使用特定注解之后，还需要再xml配置文件中声明：自动扫描**

**<context：component-scan**

**base-package**=”全类名”

**resource-pattern**=”过滤类”>

<context:include-filter>：要包含的类

<context:exclude-filter>：要排除的类

**示例代码：**

**<context:exclude-filter** **type="annotation" expression="AnnotationBean.Repository"/>**

* **命名：**Spring有默认的命名策略：第一个字母小写，或者在注解中通过value属性进行赋值
* **组件自动装配：和xml中的自动装配不同**

<context:component-scan>自动注册后置处理器：

AutowiredAnnotationbeanPoetprocessor

* @Autowired（常用）：自动装配单个具有相同类型的Bean属性，所有有参数的方法、构造器、普通字段都可以使用
* @Qualifiter：提供Bean名称，可用于注入指定名称
* @Resource：和Autowired相似
* @Inject：和Autowired相似
* **示例代码：**

**//服务层：默认名字就是“userService “**  
**@Service(value = “userService”)**  
public class **UserService** {  
 **//自动装配（原类名 + 现类名）：UserRepository，命名为userRepository**

**//方法一：自动实例化UserRepository**  
  **@Autowired UserRepository** **userRepository;**

**//方法二：这两句等同于上面一句**

**//@Autowired**

**//private UserRepository userRepository;**   
 public void add(){  
 System.*out*.println("UserService add...");  
  **//调用自动装配的组件：实现解耦，调用其他类的方法**  
 **userRepository**.save();  
 }  
}

public static void main(String[] args) {  
 ApplicationContext context = new ClassPathXmlApplicationContext("annotation-bean.xml");  
 **//根据userService名字，自动装配UserService类  
 //该类使用了自动装配：UserRepository**  
 UserService userService01 = (UserService) context.getBean("**userService**");  
 System.*out*.println(userService01);  
 userService01.add();  
}

1. **泛型依赖注入**

使用泛型类注入属性

1. **整合多个配置文件：**

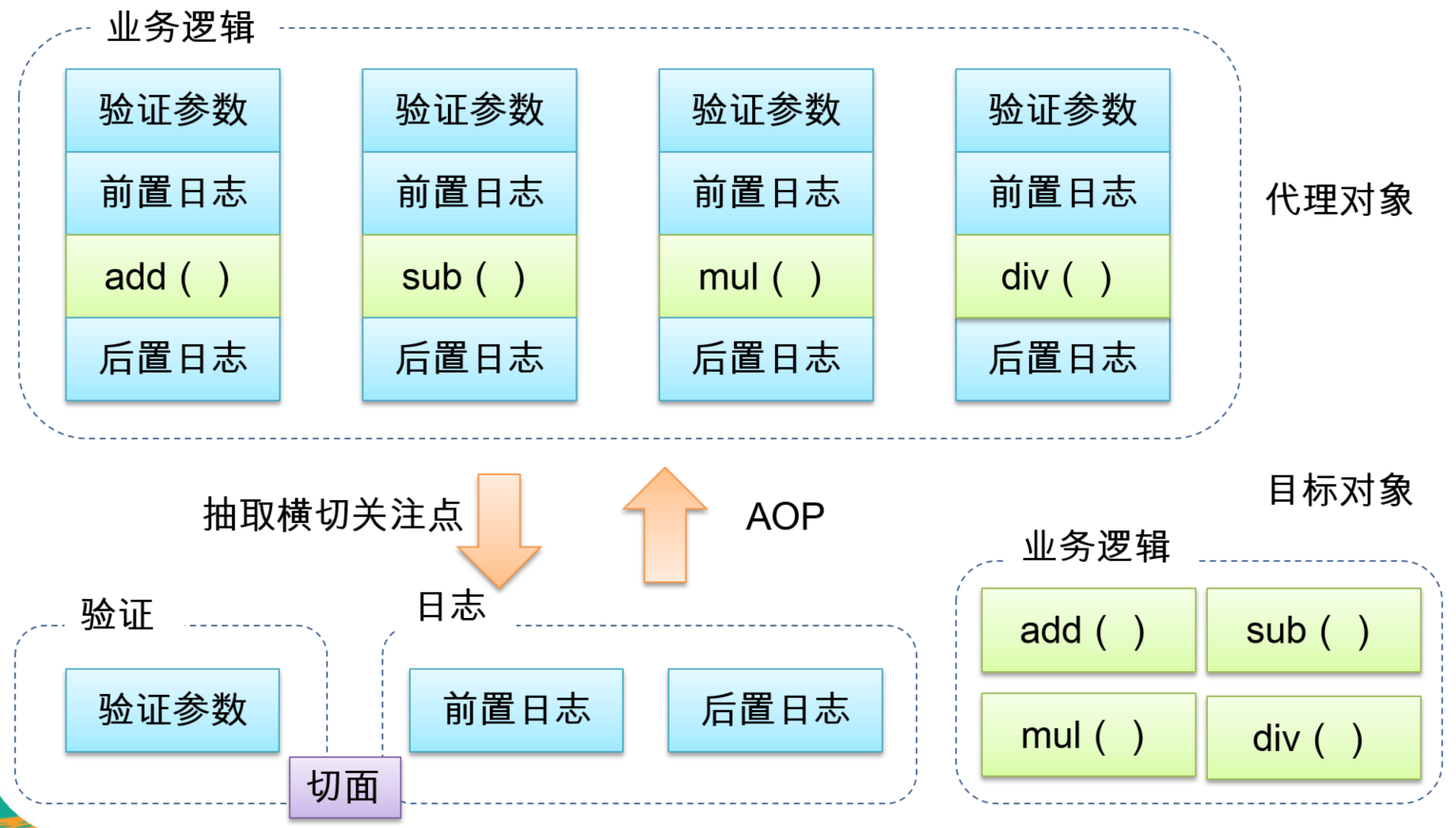
使用<import>，导入多个配置文件，import的元素为：resource，获取配置文件的路径

# AOP

1. 使用Java社区里面的**AspectJ**作为Spring的AOP框架，可以使用Aspect注解和基于xml配置AOP
2. **AOP概述：**

* 对OOP（object oriented programming）面向对象编程的补充，关注切面
* 横切关注点，将其模块化到对象中，不需要修改核心代码。即：将非核心功能归类。

**例如：**在代码中实现打印日志的功能——调用某个方法，需要打印对应的日志（方法执行前后都需要打印对应日志等功能）；**管理事务；**



* **AOP的术语：**
* **Aspect**切面：横切关注点（跨越多个模块的功能）
* **Advice**通知：切面必须完成的工作
* **Target**目标：被通知的对象
* **Proxy**代理：向目标对象应用通知之后创建的对象
* **JoinPoint连接点**：程序执行的某个特定位置
* **Pointcut切点**：每个类有多个切点，AOP通过切点定位到特定的连接点（好比：数据库中，切点为查询条件，连接点为记录的数据）

1. **使用注解方式配置AOP：(AOP通知的使用)**

* **需要额外导入的包：AspectJ、aopalliance**

\* com.springsource.net.sf.cglib-2.2.0.jar  
\* com.springsource.org.**aopalliance**-1.0.0.jar  
\* com.springsource.org.**aspectj**.weaver-1.6.8.RELEASE.jar   
\* spring-aspects-4.0.0.RELEASE.jar

* **xml配置文件：**

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:**aop**="http://www.springframework.org/schema/aop"  
 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-4.0.xsd  
 http://www.springframework.org/schema/aop**

**http://www.springframework.org/schema/aop/spring-aop-4.0.xsd**">  
  **<!-- 自动扫描包：ImplementAop -->**  
 <context:component-scan base-package="ImplementAop"></context:component-scan>  
  **<!-- 使AspectJ的注解生效 -->**  
 <aop:aspectj-autoproxy></aop:aspectj-autoproxy>  
</beans>

* **编写切面类：**如：打印日志的功能

就是一个Java类，编写想实现的方法即可

* **配置切面： 切面必须使IOC中的bean**
* 配置切面的优先级：**@Order(n)**，n越小，优先级越高
* 声明给类是一个切面：**@Aspect**
* 使用注册来配置Bean：**@component(value = “该类的别名”)**
* 声明通知：**@关键字** **+ 切入点表达式**（该方法在哪些包中执行）
* **@Before：前置通知，方法执行前运行**
* **@After：后置通知，方法执行后运行**
* **@AfterRunning：返回通知，方法返回结果后运行**
* **@AfterThrowing：异常通知，抛出异常后运行**
* **@Arround：环绕通知（很少用）**

**示例代码：**该方法在ImplementAop包中任何类型的方法之前执行。

**@Before**("execution( \* ImplementAop.Calculator.\*(int, int))")

* 在通知中访问连接细节：使用**JoinPoint**参数，可以获取当前执行包中方法的签名、参数
* **切入点表达式的使用：**

**execution(\* 包名.\*.\*(..))：表示该包内的任意类型的任意方法**

* **第一个\*：**匹配任意修饰符、任意返回值；
* **第二个\*：**任意类的对象；
* **第三个\*：**任意方法；

**注意：千万不能写成这样：execution(\* 包名.\*(..))**

* **完整的切面配置代码如下：（AOP通知）**

**@Order(1) //配置通知的优先级：数字越小，优先级越高**  
**@Aspect //通过Aspec注解，声明一个切面**  
**@Component //使用注解来配置Bean**  
public class LoggingAspcet {  
  **/\*\* 合并切入点表达式**

**\* 切入点定义：引用该定义即可获取代理类的信息（只需使用该方法名即可引用）  
 \* 用来确定哪些类需要代理,该表达式含义：Calculator类的所有方法都会被代理  
 \*/**  
 @**Pointcut**("execution(\* ImplementAop.Calculator.\*(\*, \*))")  
 private void **joinPointExpression()**{}  
 **/\*\*  
 \* 前置通知  
 \* '\*'：表示任意匹配符  
 \* 使用Before注解，声明该方法在ImplementAop包内的：任何返回值类型的任何方法之前执行  
 \*/**  
 @**Before**("**joinPointExpression()**")  
 public void beforeMethod(JoinPoint joinPoint){  
 String methodName = joinPoint.getSignature().getName();  
 Object[] args = joinPoint.getArgs();  
  
 System.*out*.println("The method " + methodName + " begin with " + Arrays.*asList*(args));  
 }  
  **/\*\*  
 \* 后置通知  
 \* 使用After  
 \*/**  
 @**After**("**joinPointExpression()**")  
 public void afterMethno(JoinPoint joinPoint){  
 String methodName = joinPoint.getSignature().getName();  
 Object[] args = joinPoint.getArgs();  
 System.*out*.println("The method " + methodName + " end with " + Arrays.*asList*(args));  
 }  
 **/\*\*  
 \* 返回通知  
 \* 在切点表达式中添加：returning = "result"，可以访问连接点的返回值  
 \* 但必须使用原始的切点表达式，不能使用重用的表达式  
 \* @param joinPoint  
 \*/**  
**@AfterReturning**(pointcut = "execution(\* \*.\*(..))", **returning** = "result")  
public void afterReturning(JoinPoint joinPoint, **Object result**){  
 String methodName = joinPoint.getSignature().getName();  
 Object[] args = joinPoint.getArgs();  
 System.*out*.println("AfterReturning " + methodName + " end " + result);  
}

1. **使用配置文件方式配置AOP**

* 使用xml来手动配置Bean

**<!-- 配置实例化对象的Bean -->**  
<bean id="calculatorImplement" class="XMLAop.CalculatorImplement"></bean>  
  
**<!-- 配置切面的Bean -->**  
<bean id="loggingAspect" class="XMLAop.LoggingAspect"></bean>  
  
**<!-- 配置AOP -->**  
<aop:config>  
 **<!-- 配置切点表达式 ：注意这个切点表达式\*的含义 -->**  
 <aop:pointcut id="pointcut" expression=**"execution(\* XMLAop.Calculator.\*.\*(int, int))"**/>  
 **<!-- 配置切面和通知 -->**  
 <aop:aspect ref="loggingAspect" order="1">  
 <aop:before method="beforeMethod" pointcut-ref="pointcut"></aop:before>  
 </aop:aspect>  
 <aop:aspect ref="loggingAspect" order="1">  
 <aop:after method="afterMethno" pointcut-ref="pointcut"></aop:after>  
 </aop:aspect>  
 <aop:aspect ref="loggingAspect" order="1">  
 <aop:after-returning method="afterReturning" pointcut-ref="pointcut" returning="result"></aop:after-returning>  
 </aop:aspect>  
</aop:config>

# JDBCTemplate

1. **JdbcTemplate查询库的使用**

* **需要额外导入的库：** c3p0：数据源，用于登陆数据库等操作

mysql-connector-java：mssql的驱动

JdbcTemplate：Spring的JDBC库（这个自带）

* **配置文件：**db.properties：存放mysql用户名、密码、连接等等

xml ：配置bean

* **xml配置：**

<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.xsd  
 http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-4.0.xsd  
 http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd">

**<!-- 导入Mysql配置资源文件：db.propetties -->**  
<context:property-placeholder location="classpath:db.properties"></context:property-placeholder>  
 **<!-- 配置c3p0数据源 ：使用db.properties资源文件 -->**  
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="user" value="${mysqlUser}"></property>  
 <property name="password" value="${mysqlPasswd}"></property>  
 <property name="jdbcUrl" value="${mysqlURL}"></property>  
 <property name="driverClass" value="${mysqlDriver}"></property>  
 <property name="initialPoolSize" value="5"></property>  
 <property name="maxPoolSize" value="10"></property>  
</bean>  
  
**<!-- 配置Spring的JDBCTemplate：给spring库函数JDBCTemplate，配置参数 dataSource**

**使用jdbcTemplate实例化对象之后，就可以使用Mysql数据库**

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

* **调用实例：**

public class Main {  
 private ApplicationContext applicationContext = null;  
 private JdbcTemplate jdbcTemplate;  
 private EmployeeInfo employeeInfo;  
  
 **//初始化代码块**  
 {  
 applicationContext = new ClassPathXmlApplicationContext("jdbc-template-bean.xml");  
 **jdbcTemplate = (JdbcTemplate)applicationContext.getBean("jdbcTemplate");**  
 namedParameterJdbcTemplate = applicationContext.getBean(NamedParameterJdbcTemplate.class);  
 }  
  
 **/\*\*  
 \* 测试连接MySQL  
 \* 这里使用的DataSource：属于java.sql库  
 \*/**  
 public void connecttion() throws SQLException {  
 DataSource dataSources = applicationContext.getBean(DataSource.class);  
 System.*out*.println(**dataSources.getConnection()**);  
 }  
  
 **/\*\*  
 \* 修改某个数据  
 \* @param id  
 \* @param attribute：属性名字  
 \* @param data：修改的值  
 \*/**  
 public void upate(int id ,String attribute,Object data){  
 String sql = "update employeeInfo set " + attribute + "=? where id=?";  
 **jdbcTemplate.update**(sql, data, id);  
 }

public static void main(String[] args) throws Exception{  
 Main main = new Main();  
  **//1、连接数据库**  
 main.connecttion();  
  **//2、修改某个元素**  
 main.upate(1,"name","vky");  
 }  
}

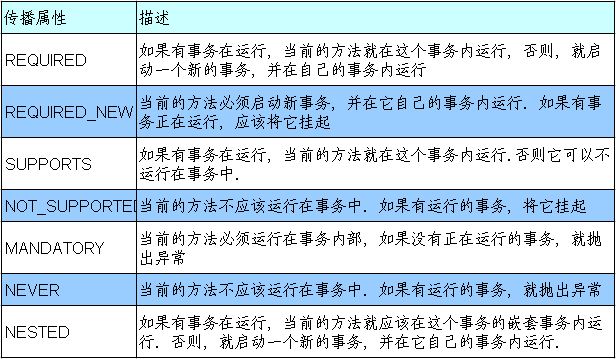
# 事务

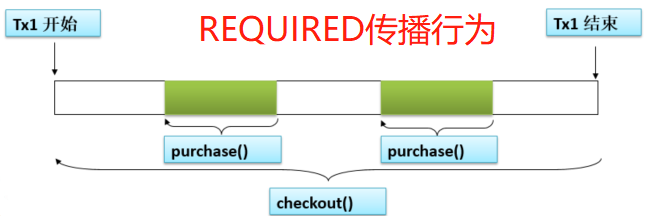
1. **事务：**

* 事务就是一系列的动作作为一个操作单元，这一些列的动作只有完成和未完成两个状态（即：起作用和不起作用）。
* 用于保证数据的完整性、一致性；
* **事务的管理方式：将操作的方法嵌入到事务中，当这个单元的所有方法执行成功后，才提交该操作，否则回滚到该事务的起点位置——保证了数据的完整性、一致性。**
* Spring使用AOP方法来模块化事务：将事务管理代码从业务中分离出来。事务管理作为横切关注点。

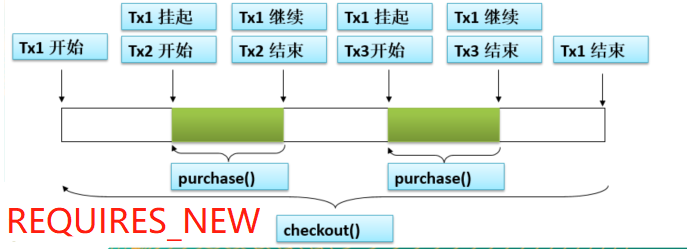
事务管理器在Spring的IOC容器中以普通Bean形式来声明。

* **事务的传播属性，如下表：**





将几个函数合并在一起执行，要么一起成功、或者一起失败



将几个函数分开执行，要么允许一个函数成功，另一个函数失败（失败的自动回滚到未改变前的状态）

* **配置事务有两种方法：**xml、注解配置Bean (@Transactional)
* **方式一：注解配置：@Transactional**
* **xml中：**

**<!—1、使用注解配置Bean ：自动扫描 -->**  
<**context:component-scan** base-package="AffairAOPAutowiredTX"></context:component-scan>  
 **<!—2.1、导入Mysql配置资源文件：db.propetties -->**  
<context:property-placeholder location="classpath:**db.properties**"></context:property-placeholder>

**<!—2.2、配置c3p0数据源 ：使用db.properties资源文件 -->**  
<bean id="**dataSource**" class="**com.mchange.v2.c3p0.ComboPooledDataSource**">  
 <property name="user" value="${mysqlUser}"></property>  
 <property name="password" value="${mysqlPasswd}"></property>  
 <property name="jdbcUrl" value="${mysqlURL}"></property>  
 <property name="driverClass" value="${mysqlDriver}"></property>  
</bean>

**<!-- 3.1、配置事务Bean，管理dataSource -->**  
<bean id="transactionManager" class="org.springframework.jdbc.**datasource.DataSourceTransactionManager**">  
 <property name="**dataSource**" ref="**dataSource**"></property>  
</bean>  
**<!-- 3.2、启用事务注解：使用自动注解方式配置Bean的情况下才能使用 -->**  
<**tx:annotation-driven** transaction-manager="transactionManager"></tx:annotation-driven>

* **需要事务管理的类文件中：**

**@Transactional**(**propagation** = Propagation.*REQUIRED*,  
 **isolation** = Isolation.*READ\_COMMITTED*,  
 **readOnly** = false,  
 **timeout** = 3)  
@Override  
public void purchaseBook(String username, String isbn) {  
  **//1、获取图书单价**  
 int price = bookstoreDAOImpl.findBookISBN(isbn);  
  
 **//2、更新图书库存数据**  
 bookstoreDAOImpl.updateBookStock(isbn);  
  
  **//3、更新用户余额**  
 bookstoreDAOImpl.updateUserInfo(username,price);  
}

* **方式二：使用xml配置事务**

**<!-- 1、导入Mysql配置资源文件：db.propetties -->**  
<context:property-placeholder location="classpath:db.properties"></context:property-placeholder>  
  
**<!-- 2、配置c3p0数据源 ：使用db.properties资源文件 -->**  
<bean id="**dataSource**" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="user" value="${mysqlUser}"></property>  
 <property name="password" value="${mysqlPasswd}"></property>  
 <property name="jdbcUrl" value="${mysqlURL}"></property>  
 <property name="driverClass" value="${mysqlDriver}"></property>  
 <property name="initialPoolSize" value="5"></property>  
 <property name="maxPoolSize" value="10"></property>  
</bean>

**<!-- ======================= 3、配置事务的 Bean ======================= -->**  
**<!-- 3.1、配置事务Bean：用于管理dataSource—即：管理数据库 -->**  
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="**dataSource**" ref="**dataSource**"></property>  
</bean>  
  
**<!-- 3.2、配置事务属性 -->**  
<tx:advice id="**txAdvice**" transaction-manager="transactionManager">  
 <tx:attributes>  
 **<!-- 根据方法名指定事务的属性 -->**  
 <tx:method name="purchaseBook" propagation="REQUIRES\_NEW"/>  
 <tx:method name="get\*" read-only="true"></tx:method>  
 <tx:method name="find\*" read-only="true"></tx:method>  
 <tx:method name="\*"></tx:method>  
 </tx:attributes>  
</tx:advice>  
  
**<!-- 3.3、配置事务切入点，将事务切入点和事务属性关联起来 -->**  
<aop:config>  
 **<!-- 关联所有属性、所有方法 -->  
 <!-- 后面两个\*,表示service包下的所有类下的所有方法  
 execution(\* AffairAOPXMLTX.\*(..)) ：会报错误  
 -->**  
 <aop:pointcut id="**txPointCut**" expression=**"execution(\* AffairAOPXMLTX.\*.\*(..))"/>  
 <!-- 将切入点和事务关联 -->**  
 <aop:advisor **advice-ref**="**txAdvice**" **pointcut**-**ref**="**txPointCut**"></aop:advisor>  
</aop:config>

# Hibernate

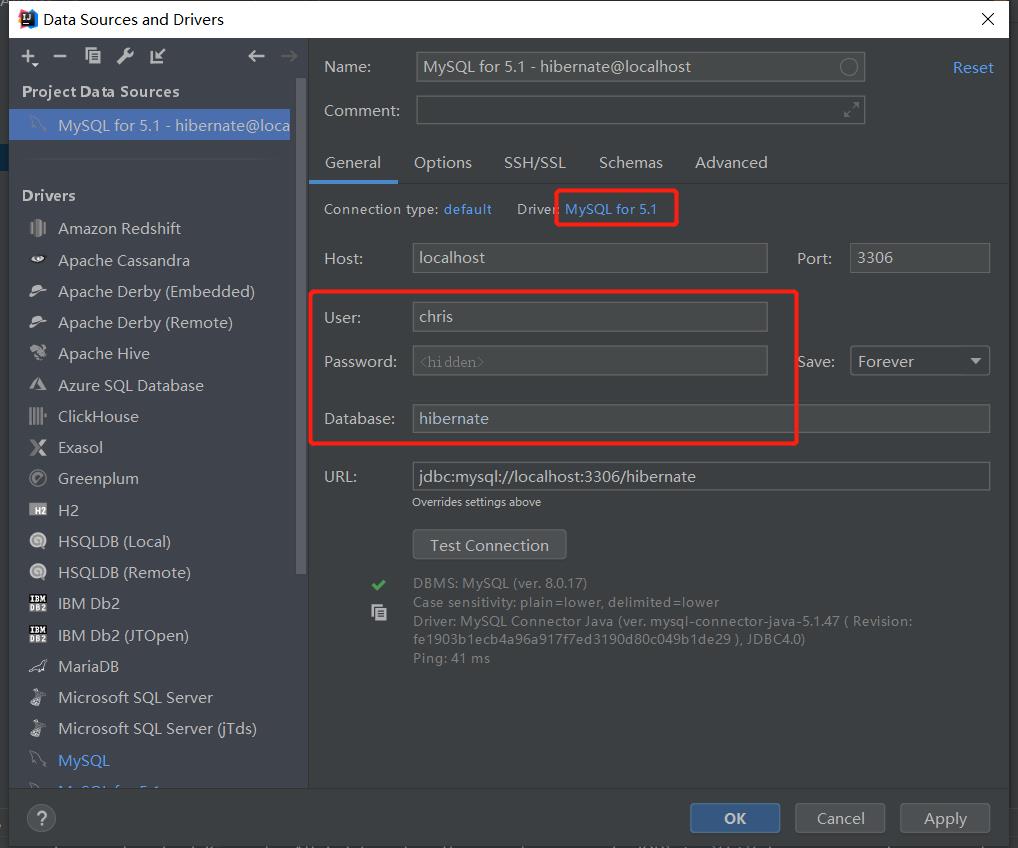
1. Hibernate是Java的一种ORM（对象关系映射）：面向对象和关系型数据库的映射，提供一个持久化的框架，提供数据检索机制，减少手动处理SQL、JDBC的工作。

使用DAO（数据访问对象）：插入数据时，需要拆分JavaBean对象，拼装SQL语句；查询数据时，需要组合SQL，拼装成JavaBean对象。

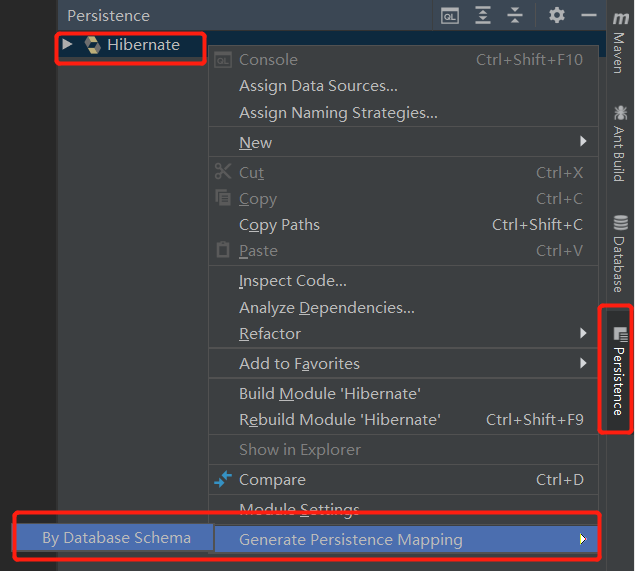
Hibernate：简化了其中SQL的编写。

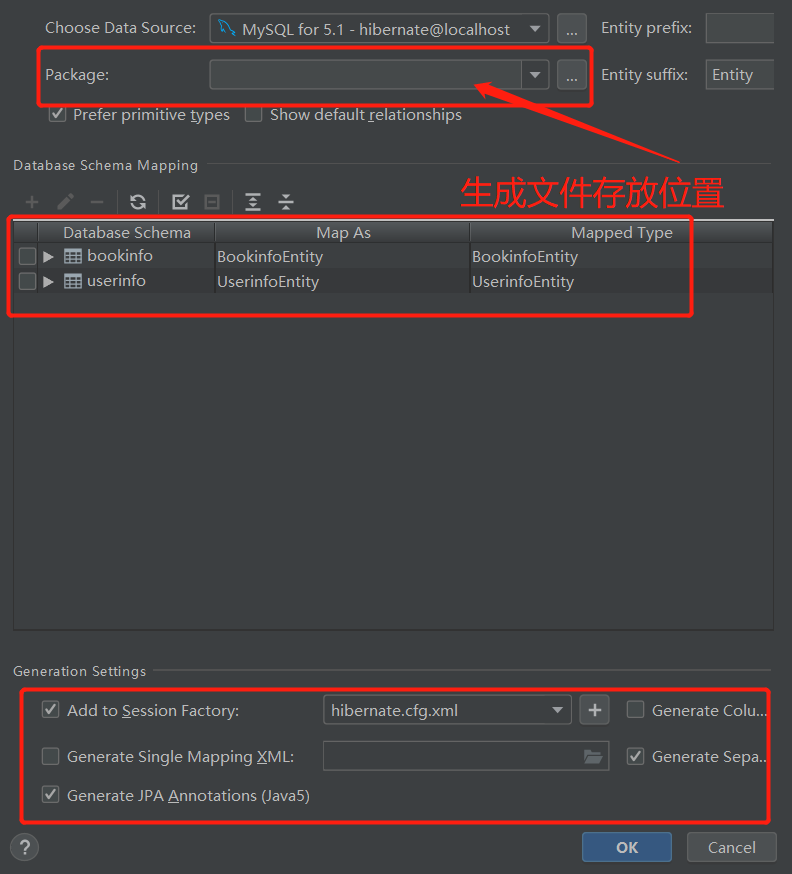
1. **几个使用的心得：**

* IntelliJ IDEA和MySql关联的设置：

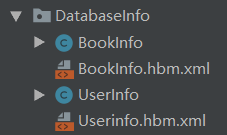


* IntelliJ IDEA关于从Mysql中导入表格数据🡪生成类、Hibernate的.hbm.xml映射文件的步骤：





生成的文件：数据库表格属性对应的类、hibrenate表格的配置文件。





1. Hibernate在Spring中的使用：

* Hibernate的配置文件共两种:
* .cfg.xml（配置文件）：Hibernate基本属性设置，可以配置Hibernate的基本属性：方言、SQL显示、生成数据表策略…

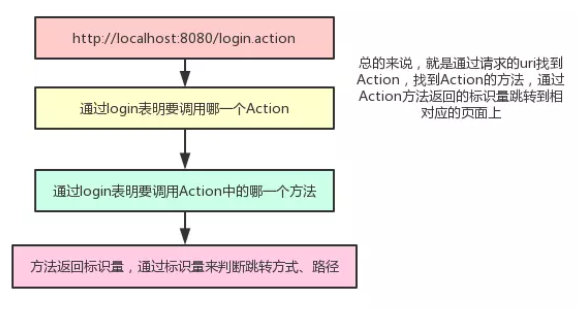
也可以配置连接数据库的参数（ 在IOC容器中配置）

* .hbm.xml（映射文件）：数据库中的表格配置
* 在IOC容器中，声明Hibernate的.cfg.xml
* 配置Spring事务，用事务去管理Hibernate



# Struts2

1. **网页响应的流程：**



1. **Struts2：**网页应用框架，延申Servlet API

# 总结

1. **Spring中实现属性注入、配置Bean的方式：**

有两种，**一个工程中，尽量使用其中一种方式进行配置。**

* XML文件手动配置Bean
* 注解自动配置Bean：**@Autowired**自动装配如下特定注解的类
* **@Component：**基本注解，表示该组件受Spring管理
* **@Respository：**表示持久层组件
* **@Service：**表示服务层组件
* **@Controller：**表现层组件
* **命名示例：**UserServicelmpl userService （原名 + 现名）

**上面4个特定注解，会默认给被注解的类命名为对应的类名（首字母小写）**

1. **IOC、AOP的关系：**

* **IOC：**就是使用ApplicationContext类，实现xml中配置的Bean对象实例化。

ApplicationContext继承自ClassPathXmlApplicationContext。在使用时，可以直接使用ClassPathXmlApplicationContext来创建对象，这样在调用方法时，可以多出一个close()方法，来关闭对象。

* **AOP：**在Spring中，使用的是开源社区提供的AspetcJ包（需要手动导入两个包：aopalliance、aspectj），AOP的实现是基于IOC的，只有AOP是无法使用的。
* **IOC、AOP都可以使用XML文件、注解配置**

1. **事务：**

* 事务是一个非常重要的概念，在hibernate、Mybatis…中有广泛的应用，需要重点掌握
* 需要手动导入的包：aopalliance、aspextj
* 事务的配置方式有两种：XML文件、注解配置
* **XML文件配置流程：**

1. **配置事务的Bean：需要被事务管理的类**
2. **配置事务的属性：**指定事务的传播方式、需要管理的方法名…
3. **配置事务的切入点（配置AOP）：**将2中配置的**事务属性**和**事务切入点**关联起来。**特别注意：切入点表达式中，符号：\*的含义**

**eg：execution(\* 包名.\*.\*(..))**

* **注解配置流程：**

1. xml中开启自动扫描
2. xml中配置需要被事务管理的类
3. xml中开启事务注解
4. 在想被事务管理的方法之上，使用注解：@Transactional，可填写相应的属性，eg：

@Transactional(propagation = propagation.REQUIRED

isolation = isolation.READ\_COMMITTED)