# 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

1. **IOC和DI的详解：**
2. **IOC：**

IOC（Inversion of control）控制反转，不是一种技术，而是一种设计思想，将设计好的对象转交给IOC容器控制（不需要自己使用new等方式创建对象）。

**1）所谓的“控制”：**即IOC容器控制了对象，控制外部资源的获取（对象、文件等等）；

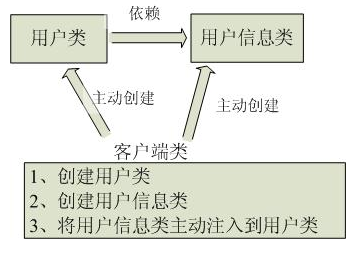
**2）所谓的“反转”：**即使用容器来查找、注入依赖对象，对象只是被动的接受依赖对象（也就是依赖对象的获取方式被“反转”了，正常的new操作属于“正转”）；

**总结：**

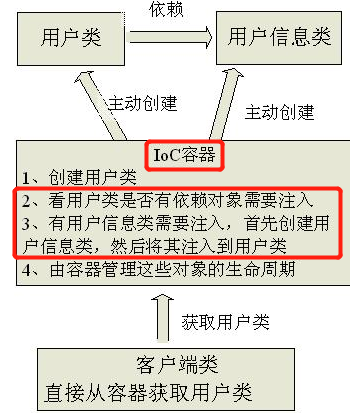
简而言之，IOC在Spring中的体现在于，使用Spring来控制对象的生命周期、对象间的关系。所有的类在Spring有登记，告诉Spring该类是什么、需要什么资源，在Spring运行期间，适当的时候Spring会动态的把该类需要的东西注入到该类中，包括该类的创建也是由Spring控制的。

**eg：**

传统的设计程序思路：



IOC/DI容器设计思路：



1. **DI：**

DI（Dependency Injection），依赖注入，组件之间依赖关系由容器在**运行期间**确定，容器动态的将某个依赖关系租入到组件之中，提升组件的重用率。

**DI的关键：**谁依赖谁，为什么需要依赖，谁注入谁，注入什么？

1. 谁依赖谁：应用程序依赖于IOC容器；
2. 为什么需要依赖：应用程序需要IOC容器提供对象需要的外部资源；
3. 谁注入谁：IOC容器注入应用程序的某个对象（应用程序依赖的对象）；
4. 注入什么：注入某个对象需要的外部资源（对象、资源、常量数据）；

**总结：**

IOC需要在系统运行时，动态的向某个类提供该类需要的其他对象，DI就是用来实现该功能的。

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**="hello01" **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) {  
  **//1、创建Spring的IOC容器对象**  
 **ApplicationContext** applicationContext = new **ClassPathXmlApplicationContext**("applicationContext.xml");  
  **//2、根据id：从IOC容器中获取Bean实例，需要id唯一**  
 HelloWorld helloWorld = (HelloWorld)applicationContext.getBean("hello01");  
 UserInfo userInfo = **(UserInfo)applicationContext.getBean("user01");**  
 **//3、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);**  
  
  **//4、调用方法**  
 helloWorld.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">  
 **<!-- 错误提示：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.properties -->**  
 <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：持久层组件
* @Service：服务层组件
* @Controller：控制层组件
* 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，还需要在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. SpringAop和AspectJ的关系：

Spring AOP提供两种编程风格：

1. AspectJ support：使用AspectJ的注解实现AOP功能；

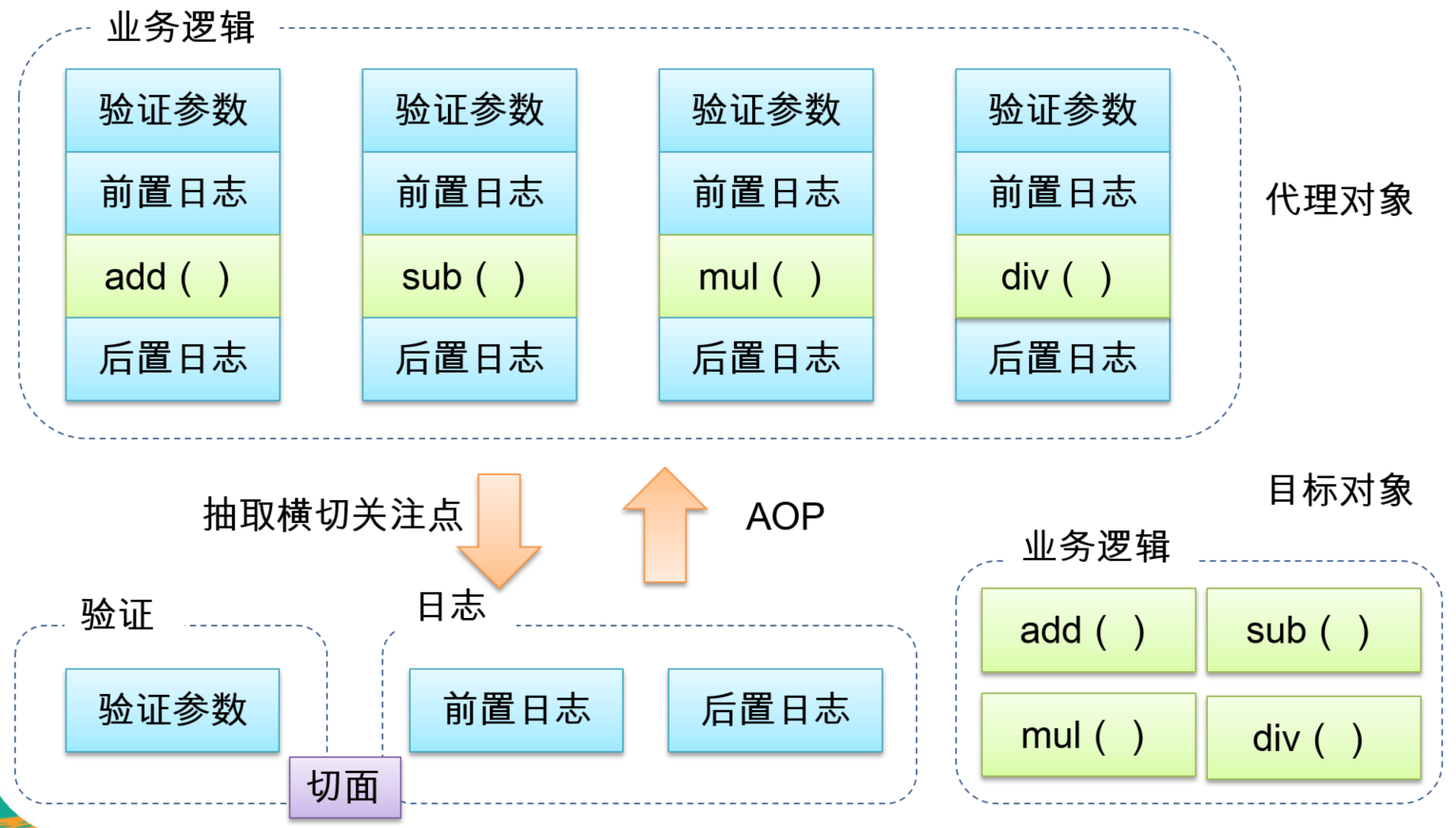
2）Schema-based AOP support：使用XML配置文件实现AOP功能；

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。将代码分成不同的层次，eg：事务、日志等，主要涉及到：织入、切面、切点、通知等。其中：

1. **织入：**将切面中的通知添加到目标对象方法的过程，分为：静态织入（编译时织入，eg：AspectJ）、动态织入（运行时织入，eg：SpringAOP。涉及到：JDK代理、CGLIB代理）
2. **切面：**是切点 + 通知的集合；
3. **切点：**是连接点的集合，定义了在什么地方执行通知中的方法；涉及切点表达式：@PointCut(“execution(\* com.\*.\*(..))”)
4. **通知：**定义了需要执行的具体方法，以及在什么时候执行方法；

**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**：由通知、目标、代理、连接点、切点等组成。
* **Advice通知**（或“增强”）：**定义被执行的自定义方法、何时使用**。有5种通知类型：Before、After、After-returning、After-throwing、Arround。
* **JoinPoint连接点**：**Advice方法执行的位置**。

连接点是一个应用执行过程中能够插入一个切面的点。连接点可以是调用方法时、抛出异常时、甚至修改字段时。切面代码可以利用这些点插入到应用的正规流程中。

* **Pointcut切点**：**连接点的集合**。每个类有多个切点，AOP通过切点定位到特定的连接点。

1. **通知：**用于定义“内容”、执行时间；
2. **切点：**用于定义执行地点，指定通知需要织入哪些连接点（即：**通知需要在哪些地方执行**）；

* **织入：**

**将切面应用到目标对象来创建代理对象的过程**。织入的时间：

1. 编译期：
2. 类加载期：
3. JVM：
4. 运行期：切面在程序运行的某个时间点被织入，AOP容器会对目标对象动态的创建代理对象。Spring AOP就是用此方法织入切面。

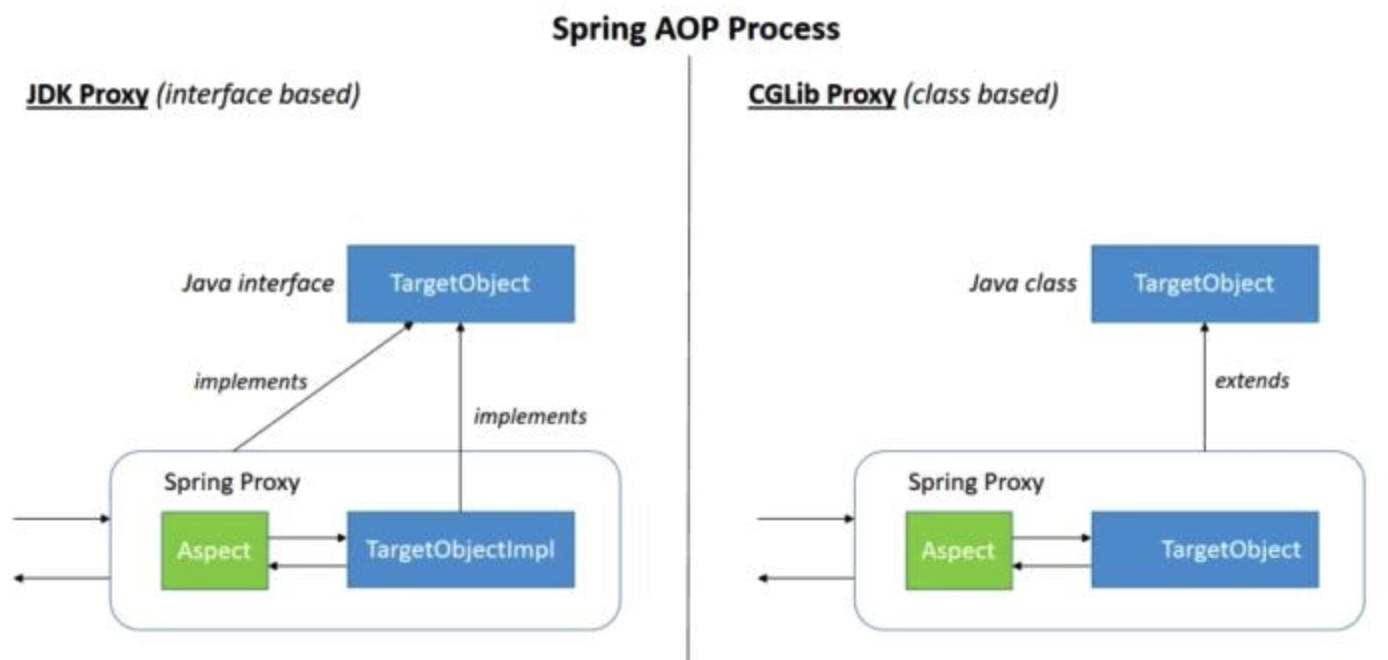
* **Target**目标：被通知的对象
* **Proxy**代理：向目标对象应用通知之后创建的对象
* **AOP执行的过程：**



advice通过PointCut切点找到target目标对象的join point连接点，再将advice放到join point连接点中运行。

* **AOP：织入-》切面-》切点-》通知的代码实现方式**

*/\*\*  
 \** ***织入Waving：****将切面和需要配置AOP对象连接起来的过程，织入可以在编译时完成，或者程序运行时完成。  
 \* （1）编译时完成织入（编译时增强）：称"静态代理"，AspctJ框架  
 \* （2）运行时完成织入（运行时增强）：称"动态代理"，Spring AOP框架（仿照AspectJ），  
 \* 分为：JDK动态代理、CGLIB动态代理  
 \* 1）JDK动态代理：基于反射实现，只要代理对象实现了一个接口，核心：InvocationHandler接口、Proxy类；  
 \* 2）CGLIB动态代理：基于AMS实现，可以使用CGLIB生成这个类的一个子类，核心：MethodInterceptor接口、Enhancer类  
 \*/  
  
/\*\*  
 \** ***切面aspect：****是****切点 + 通知的集合****，可以把@Aspect注解的类当成一个切面  
 \* 一定要给spring 管理，因为AOP还是基于IOC进行实例化  
 \*/*@Component  
@Aspect  
public class MyAspectJ {  
  
 */\*\*  
 \** ***切点pointcut：****是****连接点****的集合（就是方法的集合）  
 \* 为什么切点要声明在一个方法上?方便统一配置配置通知  
 \*/* @Pointcut("execution(\* com.ving.dao.\*.\*(..))")  
 public void pointCut(){  
  
 }  
  
 */\*\*  
 \** ***通知advice：****配置切点，  
 \* 即：在执行目标方法的哪个时间段执行这个通知代码（前置、后置、环绕、抛出异常）  
 \*/* @After("com.ving.config.VingAspectJ.pointCut()")  
 public void after(){  
 System.*out*.println("after");  
 }  
}



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\*：返回类型，**\*表示匹配任意修饰符、任意返回值；
* **第二个\*：对象名，**\*表示任意类的对象；
* **第三个\*：方法名，**\*表示任意方法；
* **（..）：方法的参数，**..表示任意参数**；**

**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 afterMethod(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.xml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 ：pointcut-ref，指定需要使用的切入点（用其id引入） -->**  
 <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");  
 }  
}

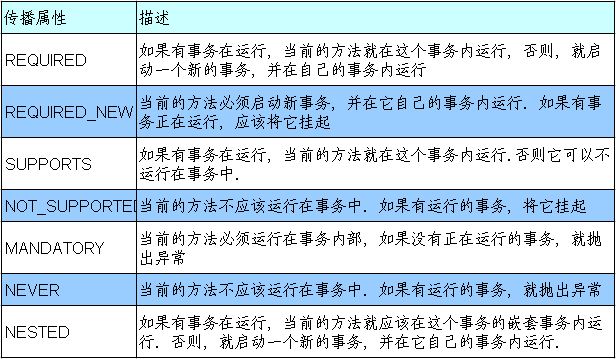
# 事务

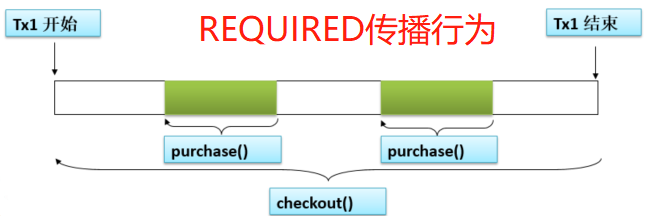
1. **事务：**

* 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。

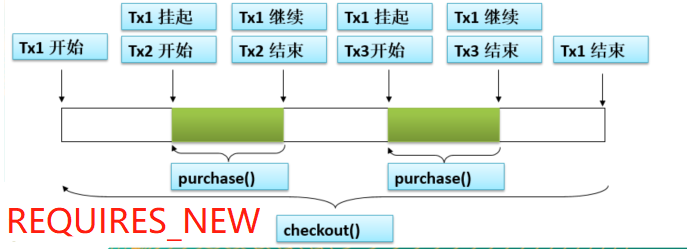
事务管理器在Spring的IOC容器中以普通Bean形式来声明。

* **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：使用注解配置事务：@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

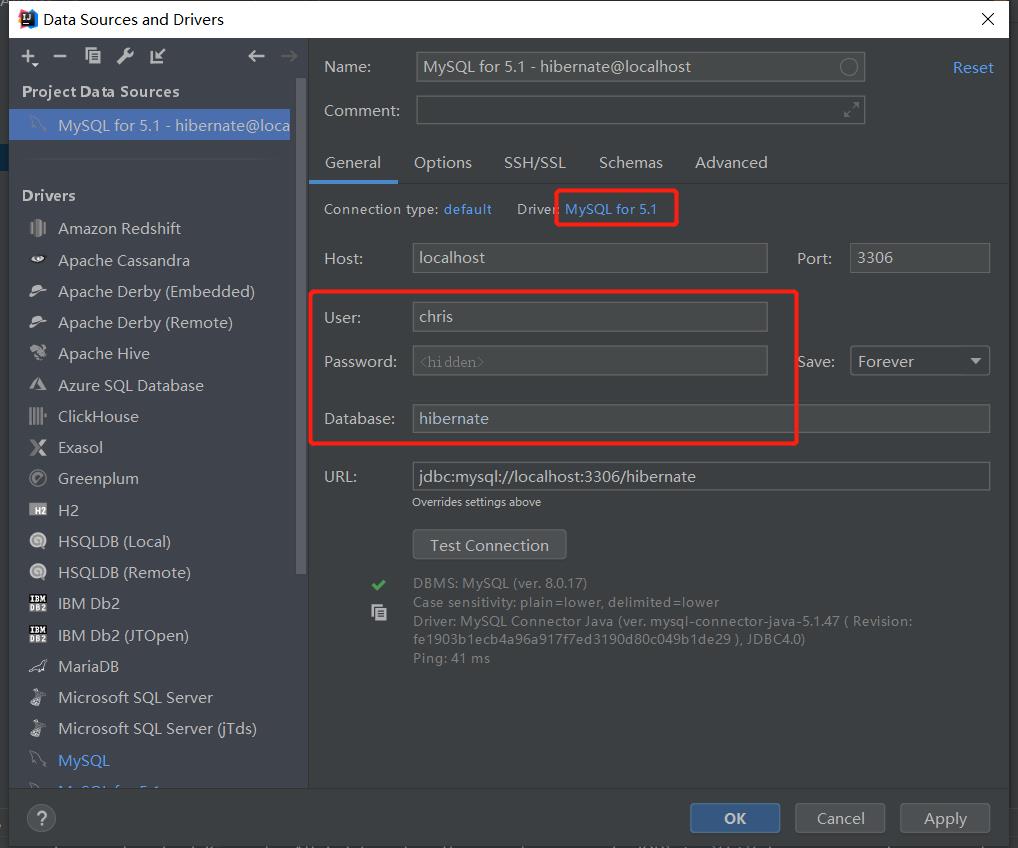
1. Hibernate是Java的一种ORM（对象关系映射）：面向对象和关系型数据库的映射，提供一个持久化的框架，提供数据检索机制，减少手动处理SQL、JDBC的工作。

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。

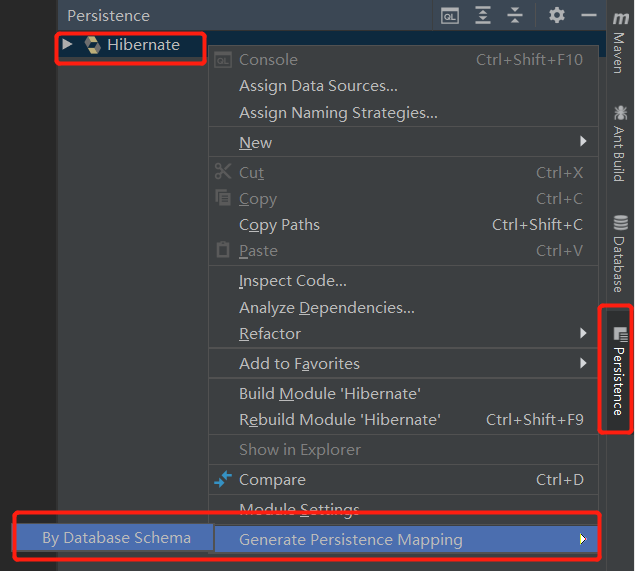
Hibernate：简化了其中SQL的编写。

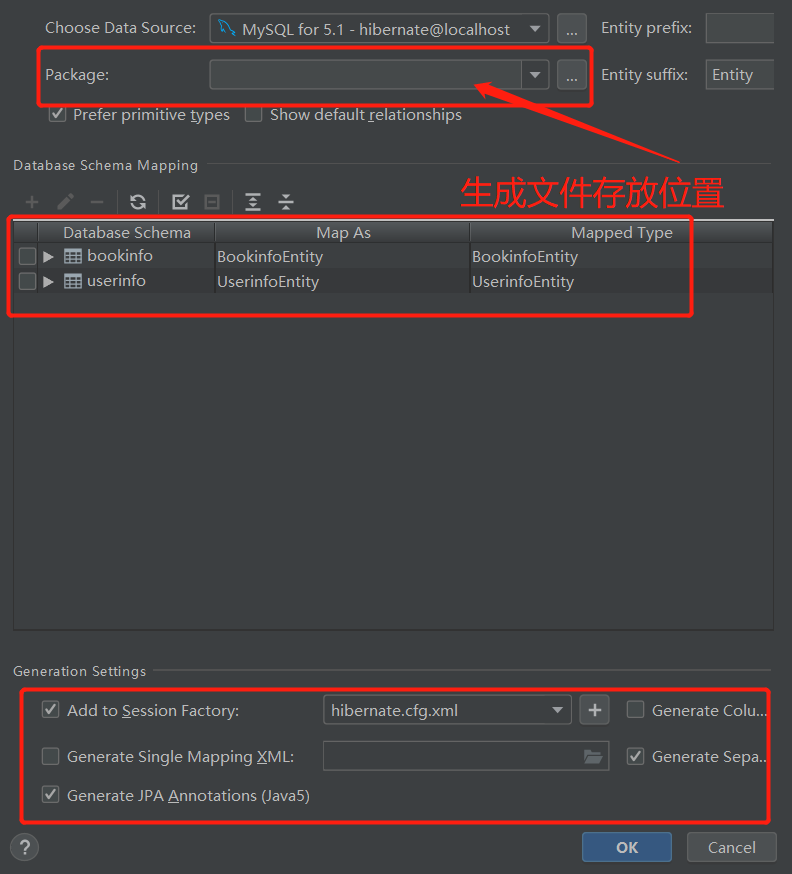
1. **IntelliJ IDEA的MySQL关联：**

* IntelliJ IDEA和MySql关联的设置：

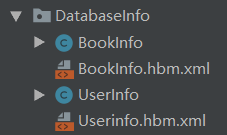


* IntelliJ IDEA关于从Mysql中导入表格数据🡪生成类、Hibernate的.hbm.xml映射文件的步骤：

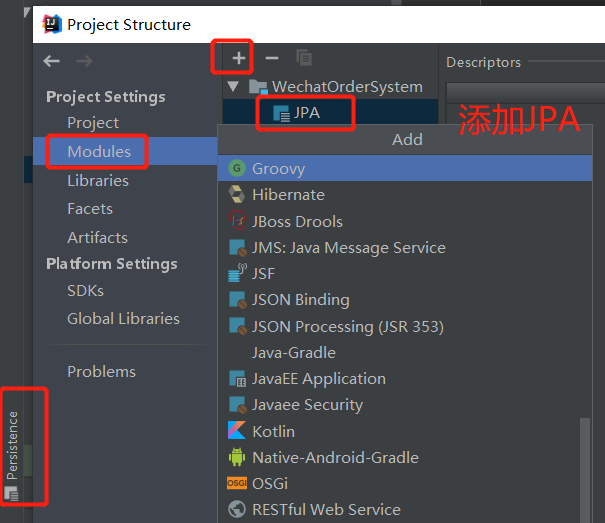




生成的文件：数据库表格属性对应的类、hibrenate表格的配置文件。



* **注意：**当IntelliJ IDEA没有出现persistence窗口时，需要添加JPA的Modules。



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…

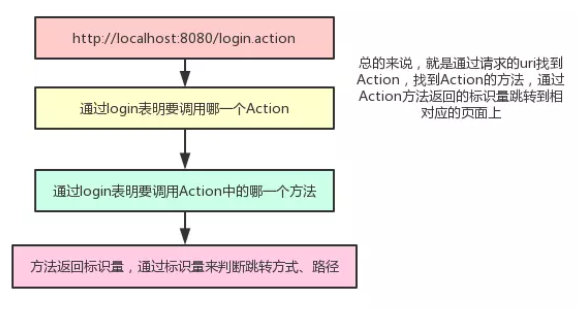
也可以配置连接数据库的参数（ 在IOC容器中配置）

* .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
* **使用注解自动配置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：**表示该类为持久层组件
* **@Service：**表示该类为服务层组件
* **@Controller：**表示该类为控制层组件
* **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)



补充知识

1. **IOC容器设计理念（IOC是设计思想，Spring是基于IOC思想设计的）**
2. IOC容器的目的：

规范对象的构建方式，自动维护对象的依赖关系，降低系统的实现成本，需要在xml中对目标对象声明。

1. 实体Bean的构建
2. 基于ClassName构建：

底层基于class属性，采用反射实现；

eg：

<bean class=”com.tulling.HelloWorld”></bean>

1. 构造方法构建：

若需要基于参数构建对象，就需要使用构造方法构建对象；

1. 静态工厂方法构建
2. FactoryBean构建

指定一个Bean工厂创建对象，对象构建初始化交给工厂来实现。

eg：

<bean class=”com.tuling.DriverFactoryBean”>

<property name=”jdbuurl” value=”jdbc:mysql://localhost:3306”>

</bean>

注意：这边的DriverFactoryBean需要继承对应的工厂接口

1. Bean的特性
2. 作用范围

使用scope指定范围

1. 生命周期

bean对象的创建🡺初始化🡺销毁

使用init-method、destory-method属性指定构建方法、处事方法、销毁方法；

1. 装载机制

使用lazy-init属性指定；

true：懒加载；

false（默认）：非懒加载；

1. **依赖注入的方式：**
2. **set方法注入：**

<bean class=”com.hello”>

<property name=”fine” ref=”fineSpring”/>

</bean>

1. **构造方法注入：**

<bean class=”com.hello”>

<contructor-arg name=”fine”>

<bean class=”com.hello.FineSpring”/>

</contructor-arg>

</bean>

1. **自动注入： 使用注解@Autowired**
2. byName：（默认）

利用反射机制，获取装配类中的set方法名，将set字段去除，然后令首字母小写，再去查找是否有对应的beanId，如果这个bean的类型和set方法的一样，就使用set方法进行注入。

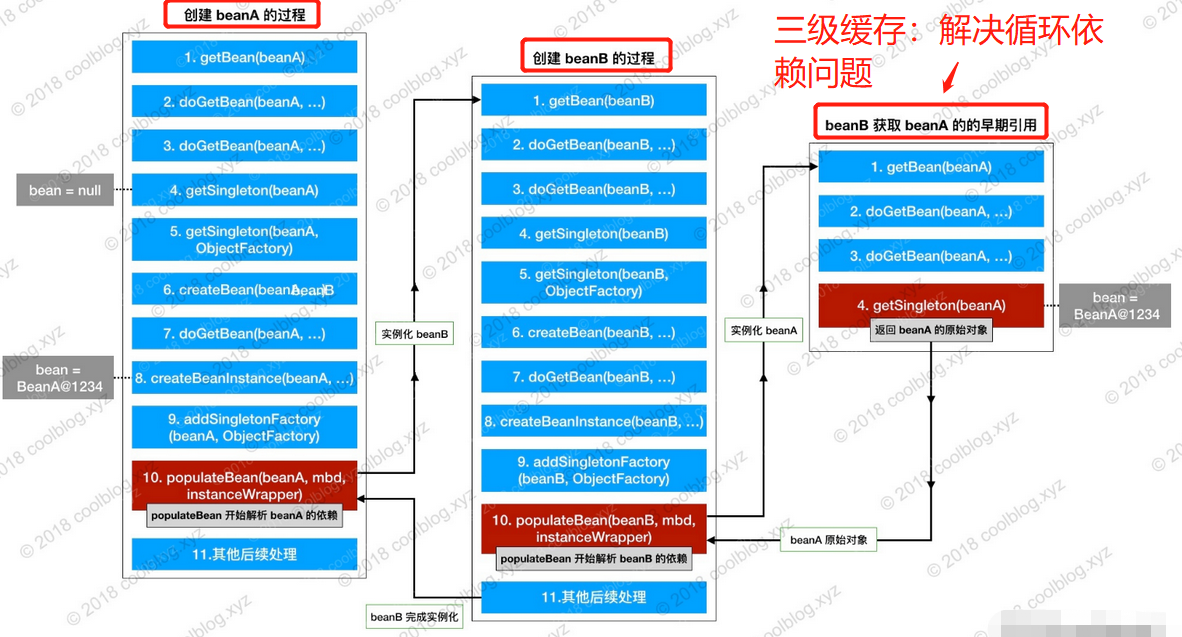
1. byType：

当byName失效时，会使用byType，也是使用反射机制获取set方法中的参数类型，去除set后小写首字母，对比beanId中的参数类型，如果一致，就是用这个set方法进行注入。

1. **方法注入：**

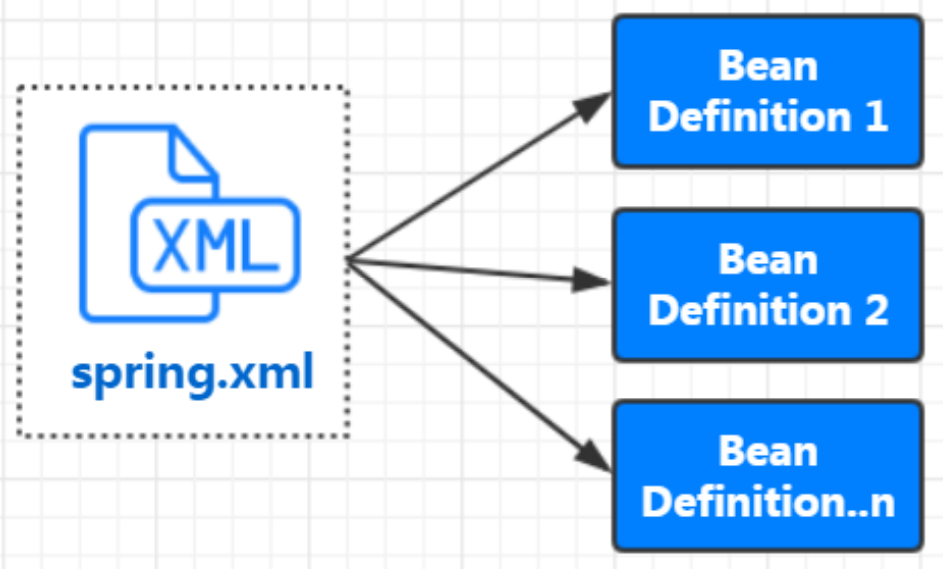
lookup-method，基于动态代理。

1. **spring解决循环注入问题：使用三级缓存：**

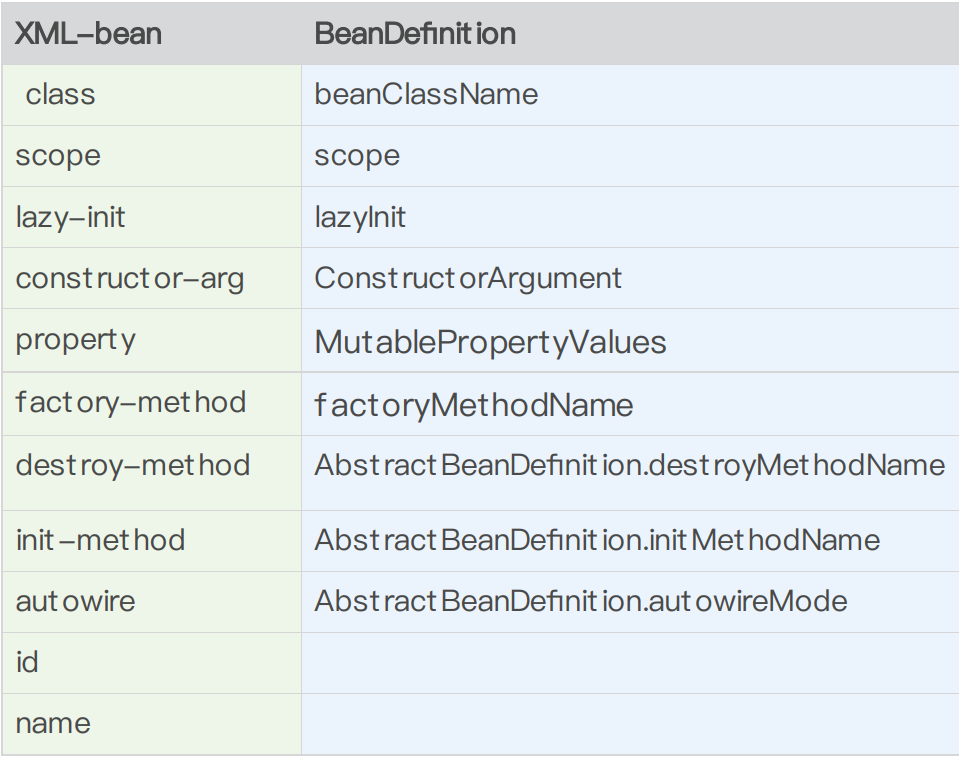


1. 使用context.getBean(A.class)，旨在获取容器内的单例A(若A不存在，就会走A这个Bean的创建流程)，显然初次获取A是不存在的，因此走**A的创建之路~**
2. 实例化A（注意此处仅仅是实例化），并将它放进缓存（此时A已经实例化完成，已经可以被引用了）
3. 初始化A：@Autowired依赖注入B（此时需要去容器内获取B）
4. 为了完成依赖注入B，会通过getBean(B)去容器内找B。但此时B在容器内不存在，就走向**B的创建之路~**
5. 实例化B，并将其放入缓存。（此时B也能够被引用了）
6. 初始化B，@Autowired依赖注入A（此时需要去容器内获取A）
7. 此处重要：初始化B时会调用getBean(A)去容器内找到A，上面我们已经说过了此时候因为A已经实例化完成了并且放进了缓存里，所以这个时候去看缓存里是已经存在A的引用了的，所以getBean(A)能够正常返回
8. **B初始化成功**（此时已经注入A成功了，已成功持有A的引用了），return（注意此处return相当于是返回最上面的getBean(B)这句代码，回到了初始化A的流程中~）。
9. 因为B实例已经成功返回了，因此最终**A也初始化成功**
10. **到此，B持有的已经是初始化完成的A，A持有的也是初始化完成的B。**
11. **IOC设计原理**
12. **Bean的构建过程：**

application.xml文件中保存了对Bean的配置信息，BeanDefinition类保存.xml中的Bean配置信息（**BeanDefinition与.xml中的bean一一对应**），BeanFactory会读取这些配置生成对应的Bean；采用CurrentHashMap（线程安全）存放singletonObjects。



IOC 容器只存放单例bean。IOC容器初始化的时候，会将所有bean初始化在singletonObjects这个CurrentHashMap中， bean是单例的。在获取bean的时候，首先会从singletonObjects去取值，通过debug，发现如果scope是单例，则可以获取到bean，如果scope是多例，则获取不到bean，需要 从一个叫mergedBeanDefinitions的CurrentHashMap中去获取bean的定义，然后再根据bean的scope去决定如何创建bean，如果scope=prototype，则每次都会创建一个新的实例。



1. **Bean创建的相关类：**

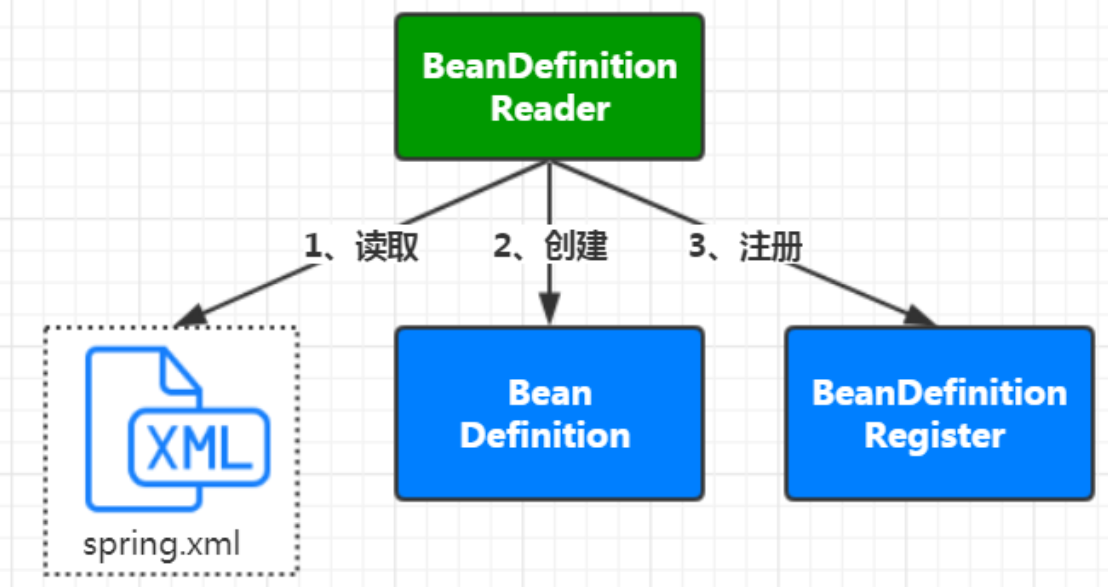
* **BeanDefinition：**继承自下面的两个类

BeanDifinitionRegistry：保存当前bean的id

AttributeAccessor：保存当前bean的name

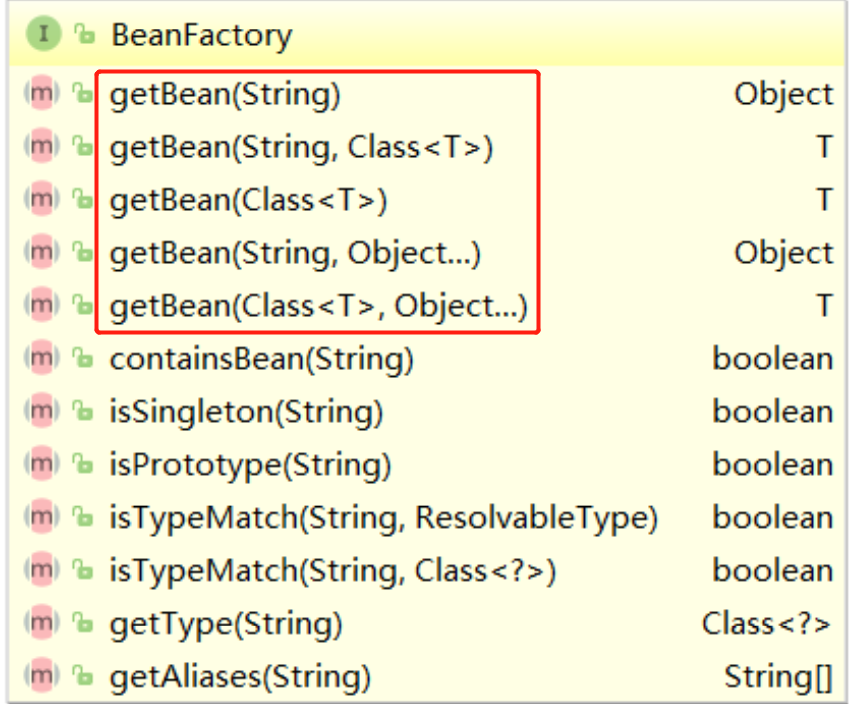
* **BeanDefinitionReader：**

作用：将bean的定义从xml文件中读取出来，并构建BeanDefinition，然后基于别名注册到BeanDefinitionRegister中。

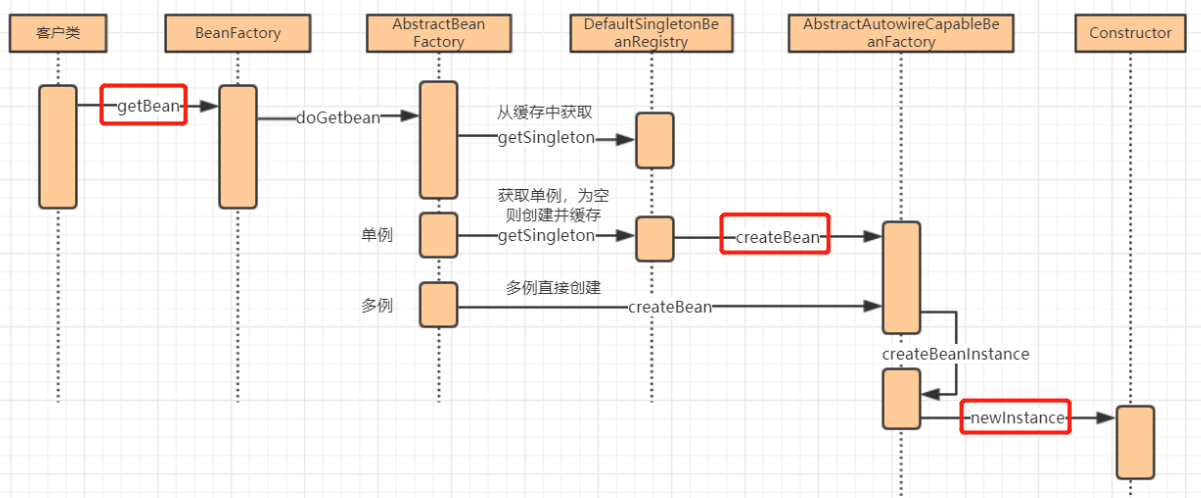


* **BeanFactory**（bean工厂）

IOC中，Beanfactory负责构建Bean对象，使用getBean()；



**Bean创建的时序图：**

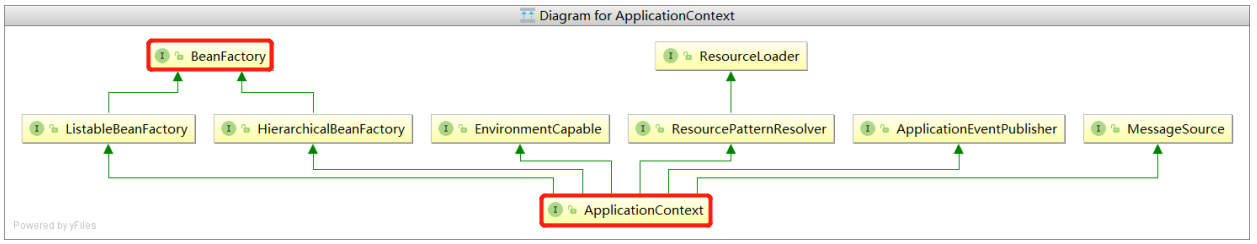


**总结：**创建Bean的过程中

1. 调用BeanFactory.getBean()会触发Bean的实例化；
2. DafaultSingletonBeanRegistry中缓存了单例Bean
3. Bean创建、初始化由AbstractAutowireCapableBeanFactory完成的；

* BeanFactory和ApplicationContext的区别：

ApplicationContext继承了BeanFactory，还继承了其他接口：MessageSource等，拥有更多的功能。






2. **AOP事务**
3. 事务的4种隔离级别：

|  |  |  |  |
| --- | --- | --- | --- |
| **隔离级别** | **脏读**  **（Dirty Read）** | **不可重复读**  **（NonRepeatable Read）** | **幻读**  **（Phantom Read）** |
| 未提交读  （Read uncommitted） | 可能 | 可能 | 可能 |
| 已提交读  （Read committed） | 不可能 | 可能 | 可能 |
| 可重复读  （Repeatable read） | 不可能 | 不可能 | 可能 |
| 可串行化  （SERIALIZABLE） | 不可能 | 不可能 | 不可能 |

1. 脏读：

事务1读取到事物2种未提交的更新数据，当事务2回滚时，就会出现事务1读取的数据不存在MySQL种的现象。

1. 不可重复读：

在同一个事物中，多次读取同一个数据返回的结果不同。例如：事务1在第一次读取数据1后。事务2对数据1进行更新操作。事务2再次读取数据1，这两次读取操作的结果不同。

（可重复读：与之相反，多次去读统一数据，结果相同）

1. 幻读：

多线程下，向数据库中插入数据，并发时可能会插入多条相同的数据，在查找时就会出现多条相同数据的现象。

**注意：**

数据库中的事物默认级别：

1、Oracle（Read committed）

2、MySQL（Repeatable read），MySQL中执行一条查询语句，默认是一个独立事务，效果和Read committed一样。

1. **Spring事物**

Spring事务在数据库事务的基础上进行封装、扩展，主要有如下特性：

1. 支持原有的数据事务的隔离级别；
2. 加入事务传播概念，提供多个事务合并、隔离的功能；
3. 提供**声明式事务**，让业务代码和事务分离；

1. **Spring提供了3个接口使用事务**
2. TransactionDefinition：定义事务
3. PlatformTransactionManager：事务管理
4. TransactonStatus：事务运行的状态

1. **Spring实现事务的方式：**
2. 方法一：编程式事务，使用API去配置事务：（繁琐）

需要创建TranscationTmplate对象，重写doInTranscation方法。

@Autowired  
TransactionTemplate transactionTemplate;  
boolean result = transactionTemplate.execute(new TransactionCallback<Boolean>() {  
 @Override  
 public Boolean doInTransaction(TransactionStatus status) {  
 try {  
 **// 任务**  
 } catch (Exception e) {  
  **//手动开启事务回滚**  
 status.setRollbackOnly();  
 return false;  
 }  
 return true;  
 }  
});

1. 方法二：声明式事务：

基于AOP实现，本质是对这个方法进行前后拦截，使用xml文件、注解配置**@Transactional**（上面有例程）

1. **事务的传播机制：**

|  |  |  |
| --- | --- | --- |
| **类别** | **事物传播类型** | **说明** |
| 支持当前事物 | PROPAGATION\_REQUIRED  （必须的） | 如果当前没有事物，就新建一个事物，如果已经存在一个事物中，加入到这个事物中。这是最常见的选择。 |
| PROPAGATION\_SUPPORTS  （支持） | 支持当前事物，如果当前没有事物，就以非事物方式执行。 |
| PROPAGATION\_MANDATORY  （强制） | 使用当前的事物，如果当前没有事物，就抛出异常。 |
| 不支持当前事物 | PROPAGATION\_REQUIRES\_NEW  (隔离) | 新建事物，如果当前存在事物，把当前事物挂起。 |
| PROPAGATION\_NOT\_SUPPORTED  (不支持) | 以非事物方式执行操作，如果当前存在事物，就把当前事物挂起。 |
| PROPAGATION\_NEVER  (强制非事物) | 以非事物方式执行，如果当前存在事物，则抛出异常。 |
| 嵌套事物 | PROPAGATION\_NESTED  （嵌套事物） | 如果当前存在事物，则在嵌套事物内执行。如果当前没有事物，则执行与PROPAGATION\_REQUIRED类似的操作。 |

**常用的事务传播机制：**

1. PROPAGATION\_**REQUIRED**（默认机制）：

如若外层有事务，就把当前的事务加入到外层事务，一起提交、一起回滚；没有的话，就新建一个事务；

1. PROPAFGATION\_**SUPPORTS**：

如果调用这个方法的bean有声明事务，就使用这个事务。否则不使用事务；

1. PROPAGATION\_**REQUIRED\_NEW**：

直接创建新事物，把外层的事务挂起，等待当前事务执行完，才会执行外层事务；

1. PROPAGATION\_**NOT\_SUPPORTED**：

不启用事务操作，即使操作出现错误时，也不会回滚；

1. **事务的回滚规则：**
2. runtime exception：运行时异常
3. **AOP事务底层实现原理**

Spring声明式事务，是基于动态代理实现的，当调用同一个类中的方法时，是不会走代理逻辑，因此也就无法在调用两个同类方法时产生两个不同的事务。

**示例代码：（同一类中）**

@Override

**@Transactional**

public void createUser(String name) {

jdbcTemplate.**update**("INSERT INTO `user` (name) VALUES(?)", name);

**addAccount**(name, 10000); //**同类中**，该方法配置的事务无法启用

TestClass.**addAccount**(name, 10000); //**不同类中**有相同的方法，则可以使用事务

// 人为报错

int i = 1 / 0;

}

**@Transactional**(propagation = **Propagation.REQUIRES\_NEW**)

public void **addAccount**(String name, int initMoney) {

String accountid = new SimpleDateFormat("yyyyMMddhhmmss").format(new Date());

jdbcTemplate.update("insert INTO account (accountName,user,money) VALUES (?,?,?)", accountid, name, initMoney);

}

1. **AOP底层技术：**

**Spring AOP是在运行时织入切面**。

1. **JDK动态代理：**

**基于接口实现**，因为JDK动态代理类已经继承了Proxy类，因此只能够继承其他的接口（Java只能单继承）。

使用接口：InvocationHandler

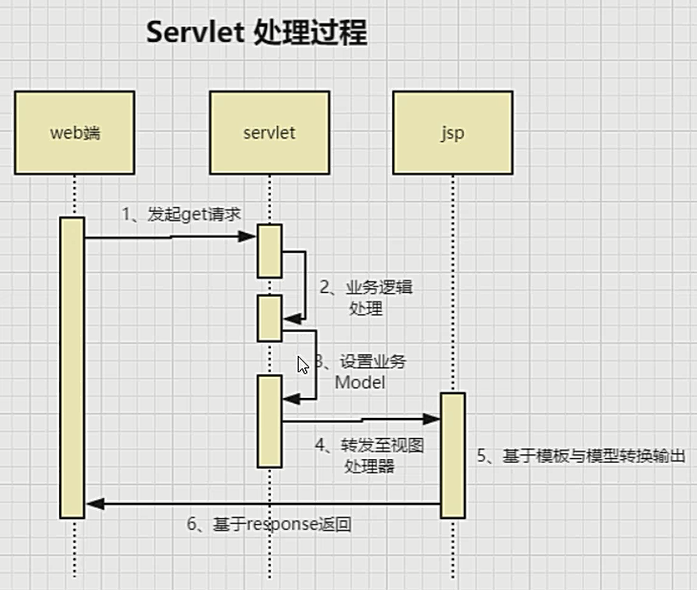
使用的类：Proxy

1. **CGLIB代理：**

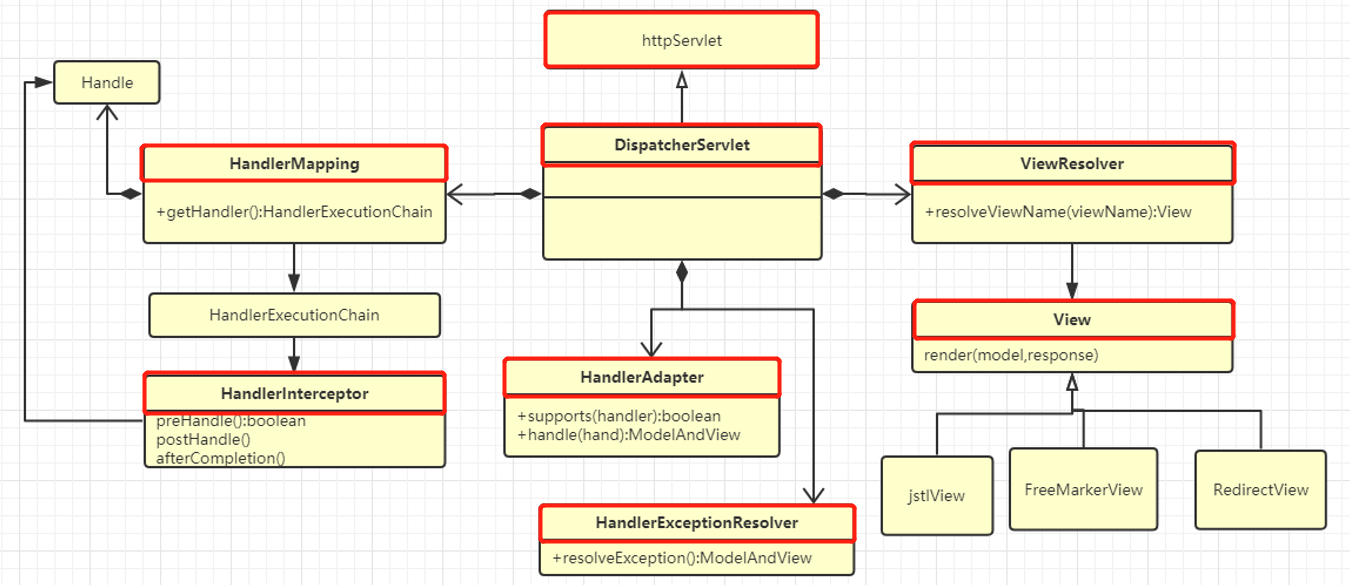
**基于继承类实现**。

使用的接口：MethodInteceptor

1. **SpringMVC原理**
2. **Servlet执行流程（标准的MVC模式）**



1. **SpringMVC结构：**
2. **整体架构**



DispatcherServlet：

1. **HandlerMapping**

url、控制器的映射，用于寻找control；有3种主流mapping：

* **BeanNameUrlHandlerMapping**：

基于IOC的name中”/”开头Bea注册的映射；

**eg：**

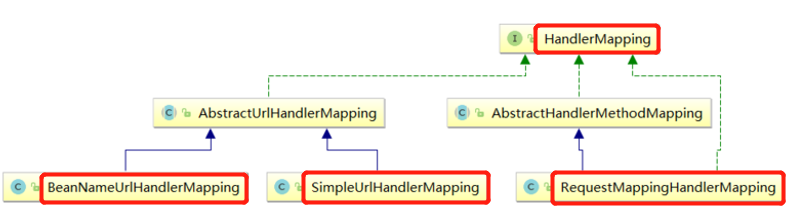
<bean id="/user.do" class="com.tuling.mvc.control.BeanNameControl"/>

* **SimpleHandlerMapping**：

基于手动配置url和control映射；

* **RequestMappingHandlerMapping**：

基于@RequestMapping注解配置的映射；



1. **HandlerAdapter**

控制器执行适配器，SpringMVC采用适配器模式调用指定的Handler（即：control控制器），不同类型的Handler对应不同的HandlerAdapter：

|  |  |  |
| --- | --- | --- |
| Handler类别 | 对应适配器 | 描述 |
| Controller | SimpleControllerHandlerAdapter | 标准控制器，返回ModelAndView |
| HttpRequestHandler | HttpRequestHandlerAdapter | 业务自行处理 请求，不需要通过modelAndView 转到视图 |
| Servlet | SimpleServletHandlerAdapter | 基于标准的servlet 处理 |
| HandlerMethod | RequestMappingHandlerAdapter | 基于@requestMapping对应方法处理 |

**eg**：简单的SimpleServletHandlerAdapter使用示例

<!-- 配置控制器 -->

<bean id="/hello.do" class="com.tuling.mvc.control.HelloServlet"/>

<!-- 配置适配器 -->

<bean class="org.springframework.web.servlet.handler.SimpleServletHandlerAdapter"/>

// 标准Servlet

public class HelloServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

resp.getWriter().println("hello luban ");

}

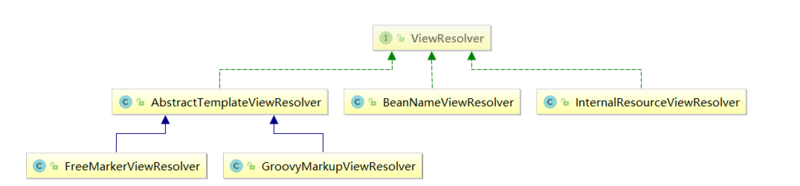
}

上述例子中当IOC 中实例化这些类之后 DispatcherServlet 就会通过

org.springframework.web.servlet.DispatcherServlet#getHandlerAdapter() 方法查找对应handler的适配器 ，如果找不到就会报 No adapter for handler 。

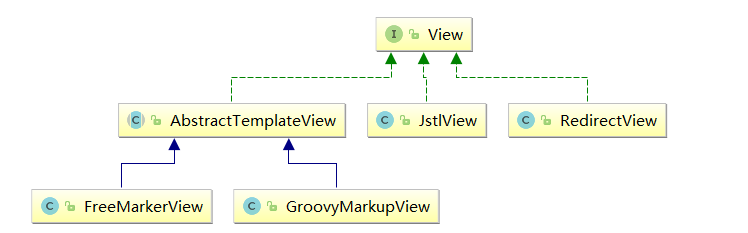
1. **ViewResolver**

视图仓库。当找到Adapter之后，会基于是配置调用业务处理，业务（control）处理完之后会返回一个ModelAndView。然后去查找对应的视图进行处理。



1. **View**

具体视图解析，根据ViewResolver.resolveViewname()获取对应的View，解析之后生成html并返回；



1. **HandlerExceptionResolver**

异常捕捉器，SpringMVC遇到异常时会

1. **HandlerInterceptor**

拦截器，类似于filter过滤器。其中SimpleHandlerInterceptor的实现机制：基于HandlerExecutionChain.doDispatch()中执行3种方法：

* prehandle：业务处理前执行
* postHandle：业务处理后执行（出现异常时，不执行）
* afterCompletion：视图处理后执行

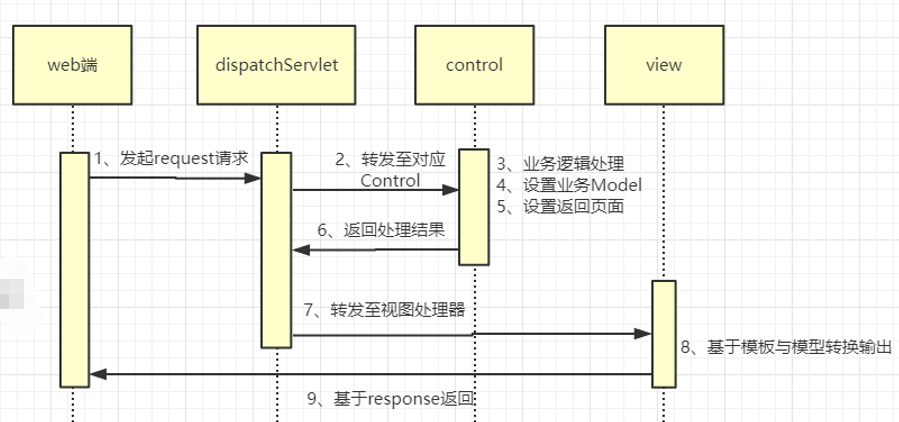
1. **创建SpringMVC的流程：**

* 创建Controller类
* 配置DispatchServlet
* 创建spring-mvc.xml文件
* 配置SimpleYrlHandlerMapping
* 配置InternalResourceViewResolver

1. **SpringMVC组件（2-7）的执行流程：**



1. **SpringMVC执行流程：**
2. SpringMVC本质上还是执行Servlet流程，只不过是在其基础上进行封装、简化了开发流程；（**增加了Control层**）





1. **常见问题：**
2. 为什么基于 <mvc:annotation-driven/> 配置就能实现mvc 的整个配置了，之前所提到的 handlerMapping 、与handlerAdapter 组件都不适用了？

只要查看以类的源就可以知晓其中原因：

1. 认识 NamespaceHandler 接口
2. 查看 MvcNamespaceHandler
3. 查看AnnotationDrivenBeanDefinitionParser

**结论**：

在<mvc:annotation-driven/>对应的解析器，自动向ioc里面注册了两个BeanDefinition。分别是：RequestMappingHandlerMapping与BeanNameUrlHandlerMapping