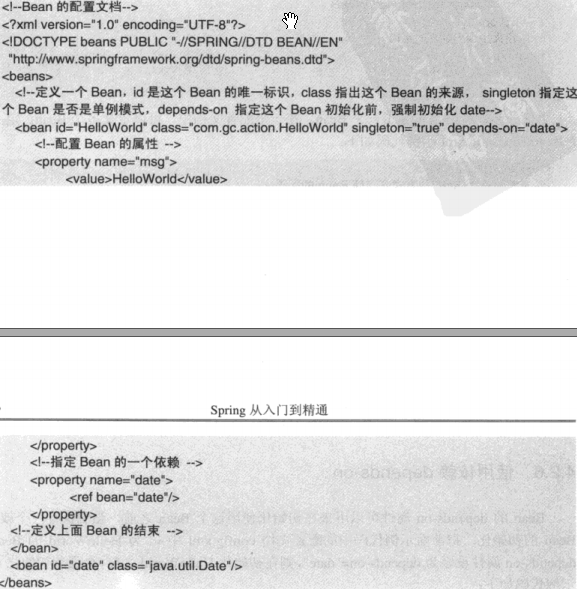
**Spring核心组件**

1. bean的基本属性

Spring依赖注入的三种方式：接口注入，set注入，构造注入



depends-on：初始化HelloWorld时会强制初始化date

singleton：表示项目中只有一个bean实例

reg 有三个属性：local bean parent

1. bean的生命周期：

定义->初始化->使用->销毁

1. bean自动装配：

byNme模式

byType模式

constructor模式

autodetect模式

no模式

1. **管理bean**

BeanWrapper（很少用）

BeanFactory

ApplicationContext

**Spring的AOP**

1. 概念

面向切面编程，把散落在程序中的公共部分提取出来，做成切面类，这样的好处在于，代码的可重用，一旦涉及到该功能的需求发生变化，只要修改该代码就行，否则需要到处修改

传统OOP（面向对象编程）通过抽象，封装，继承，多态等方法把事物进行抽象和封装，建立一种以对象为基础的层次结构。但是具体到事物的内部，OOP可能会导致大量重复的代码，比如日志，使用OOP会导致代码散布在所有对象内部。

AOP则解决了这种问题，它利用一种叫横切的技术刨解开对象内部，将那些与核心业务代码无关，并且到处都用到的代码封装成单独的模块。

1. point cut（切入点）

<aop:config>

<aop:pointcut expression=”execution(……)”id=””/>

<aop:aspect id =”” ref=””>

<aop:beforemethod=””/>

<aop:aftermethod=””/>

</aop:aspect>

</aop:config>

**Spring常见用法：**

1. **配置文件：**

web.xml:

<!-- Spring配置 -->  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>  
  
<!-- 指定Spring Bean的配置文件所在目录。默认配置在WEB-INF目录下 -->  
<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:applicationContext.xml</param-value>  
</context-param>

applicationContext.xml:

<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:mvc**="http://www.springframework.org/schema/mvc"** xsi:schemaLocation**="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>  
 <!--在xml配置了这个标签后，spring可以自动去扫描base-pack下面或者子包下面的Java文件，如果扫描到有@Component@Controller@Service等这些注解的类，则把这些类注册为bean-->  
 <context:component-scan base-package**="mars"**>  
 <!-- 扫描时跳过 @Controller 注解的JAVA类（控制器） -->  
 <context:exclude-filter type**="annotation"** expression**="org.springframework.stereotype.Controller"**/>  
 </context:component-scan>  
 <mvc:annotation-driven />

<!—配置Hibernate的相关配置 -->  
 <context:property-placeholder location**="classpath:/config.properties"** />  
 <bean id**="dataSource"** class**="com.mchange.v2.c3p0.ComboPooledDataSource"** destroy-method**="close"**>  
 <property name**="driverClass"** value**="${jdbc.driver}"** /> <!--数据库连接驱动-->  
 <property name**="jdbcUrl"** value**="${jdbc.url}"** /> <!--数据库地址-->  
 <property name**="user"** value**="${jdbc.username}"** /> <!--用户名-->  
 <property name**="password"** value**="${jdbc.password}"** /> <!--密码-->  
 <property name**="maxPoolSize"** value**="40"** /> <!--最大连接数-->  
 <property name**="minPoolSize"** value**="1"** /> <!--最小连接数-->  
 <property name**="initialPoolSize"** value**="10"** /> <!--初始化连接池内的数据库连接-->  
 <property name**="maxIdleTime"** value**="20"** /> <!--最大空闲时间-->  
 </bean>  
 <!--配置session工厂-->  
 <bean id**="sessionFactory"** class**="org.springframework.orm.hibernate4.LocalSessionFactoryBean"**>  
 <property name**="dataSource"** ref**="dataSource"** />  
 <!--自动扫描目录下的entity类-->  
 <property name**="packagesToScan"** value**="mars.entity"** />  
 <property name**="hibernateProperties"**>  
 <props>  
 <prop key**="hibernate.hbm2ddl.auto"**>${hibernate.hbm2ddl.auto}</prop> <!--hibernate根据实体自动生成数据库表-->  
 <prop key**="hibernate.dialect"**>${hibernate.dialect}</prop> <!--指定数据库方言-->  
 <prop key**="hibernate.show\_sql"**>${hibernate.show\_sql}</prop> <!--在控制台显示执行的数据库操作语句-->  
 <prop key**="hibernate.format\_sql"**>${hibernate.format\_sql}</prop> <!--在控制台显示执行的数据哭操作语句（格式）-->  
 </props>  
 </property>  
 </bean>  
  
 <!-- 事物管理器配置 -->  
 <bean id**="transactionManager"** class**="org.springframework.orm.hibernate4.HibernateTransactionManager"**>  
 <property name**="sessionFactory"** ref**="sessionFactory"** />  
 </bean>  
</beans>

注意点：springMVC和sping都有一个<context:component-scan base-package=””>标签

但是略有不同， spring的注解一般都是自动注入，标识仓储，管理事务的作用，如controller，service，repository等。springmvc更注重控制层接受数据等，如@requestMapping，@ResponseBody，@RequestBody等。

1. **Spring的相关注解**

@Service用于标注业务层组件

@Controller用于标注控制层组件（如struts中的action）

@Repository用于标注数据访问组件，即DAO组件

@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Required注解检查 但他只检查属性是否已经设置而不会[测试](http://lib.csdn.net/base/softwaretest)属性是否非空

@Autowired可以对成员变量、方法和构造函数进行标注，来完成自动装配的工作

扩展：

@PostConstruct(JSR250)

@PreDestroy(JSR250)

@Resource(JSR250)

@WebServiceRef(JAX-WS)

@EJB(EJB)

@PersistenceContext(JPA)

@PersistenceUnit(JPA)

1. **IOC**

原理解读：<http://blog.csdn.net/it_man/article/details/4402245>

依赖注入的方法：

a、接口注入；  
b、setter方法注入；  
c、构造方法注入；  
**3.1接口注入**

1. **public** **class** ClassA {
2. **private** InterfaceB clzB;
3. **public** **void** doSomething() {
4. Ojbect obj = Class.forName(Config.BImplementation).newInstance();
5. clzB = (InterfaceB)obj;
6. clzB.doIt();
7. }
8. }

ClassA依赖于InterfaceB的实现，获得InterfaceB的实现实例的传统的方法是在代码中创建 InterfaceB实现类的实例，并将赋予clzB.这样一来，ClassA在编译期即依赖于InterfaceB的实现。为了将调用者与实现者在编译期分离，于是有了上面的代码。我们根据预先在配置文件中设定的实现类的类名(Config.BImplementation),动态加载实现类，并通过InterfaceB强制转型后为ClassA所用，这就是接口注入的一个最原始的雏形。

**3.2 setter方法注入**

setter注入模式在实际开发中有非常广泛的应用，setter方法更加直观，我们来看一下spring的配置文件：

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xmlns:tx="http://www.springframework.org/schema/tx"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
7. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.1.xsd
8. http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">
9. <!-- 使用spring管理对象的创建，还有对象的依赖关系 -->
10. <bean id="userDao4Mysql" **class**="com.tgb.spring.dao.UserDao4MysqlImpl"/>
11. <bean id="userDao4Oracle" **class**="com.tgb.spring.dao.UserDao4OracleImpl"/>
12. <bean id="userManager" **class**="com.tgb.spring.manager.UserManagerImpl">
13. <!-- (1)userManager使用了userDao，Ioc是自动创建相应的UserDao实现，都是由容器管理-->
14. <!-- (2)在UserManager中提供构造函数，让spring将UserDao实现注入（DI）过来 -->
15. <!-- (3)让spring管理我们对象的创建和依赖关系，必须将依赖关系配置到spring的核心配置文件中 -->
16. <property name="userDao" ref="userDao4Oracle"></property>
17. </bean>
18. </beans>

接着我们来看一下，setter表示依赖关系的写法

1. **import** com.tgb.spring.dao.UserDao;
2. **public** **class** UserManagerImpl **implements** UserManager{
3. **private** UserDao userDao;
4. //使用设值方式赋值
5. **public** **void** setUserDao(UserDao userDao) {
6. **this**.userDao = userDao;
7. }
8. @Override
9. **public** **void** addUser(String userName, String password) {
10. userDao.addUser(userName, password);
11. }
12. }

**3.3 构造器注入**

构造器注入，即通过构造函数完成依赖关系的设定。我们看一下spring的配置文件：

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xmlns:tx="http://www.springframework.org/schema/tx"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
7. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.1.xsd
8. http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">
10. <!-- 使用spring管理对象的创建，还有对象的依赖关系 -->
11. <bean id="userDao4Mysql" **class**="com.tgb.spring.dao.UserDao4MysqlImpl"/>
12. <bean id="userDao4Oracle" **class**="com.tgb.spring.dao.UserDao4OracleImpl"/>
13. <bean id="userManager" **class**="com.tgb.spring.manager.UserManagerImpl">
14. <!-- (1)userManager使用了userDao，Ioc是自动创建相应的UserDao实现，都是由容器管理-->
15. <!-- (2)在UserManager中提供构造函数，让spring将UserDao实现注入（DI）过来 -->
16. <!-- (3)让spring管理我们对象的创建和依赖关系，必须将依赖关系配置到spring的核心配置文件中 -->
17. <constructor-arg ref="userDao4Oracle"/>
18. </bean>
19. </beans>

构造器表示依赖关系的写法，代码如下所示：

1. **import** com.tgb.spring.dao.UserDao;
2. **public** **class** UserManagerImpl **implements** UserManager{
3. **private** UserDao userDao;
4. //使用构造方式赋值
5. **public** UserManagerImpl(UserDao userDao) {
6. **this**.userDao = userDao;
7. }
8. @Override
9. **public** **void** addUser(String userName, String password) {
10. userDao.addUser(userName, password);
11. }
12. }