Spring

1. IOC

控制反转：Spring容器来实现相互依赖对象的创建、协调工作。对象只需要关系业务逻辑本身就可以了。从这方面来说，对象如何得到他的协作对象的责任被反转了

IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。比如对象A需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的。那么DI是如何实现的呢？ Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring就是通过反射来实现注入的

2. AOP

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。

SpringAOP的实现机制：

Spring AOP采用动态代理机制和字节码生成技术实现。与第一代AspectJ采用编译器将横切逻辑织入目标对象不同，动态代理机制和字节码生成都是在运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

默认情况下，如果Spring AOP发现目标对象实现了相应Interface，则采用动态代理机制为其生成代理对象实例。而如果目标对象没有实现任何Interface，Spring AOP会尝试使用CGLIB(Code Generation Library)这种动态字节码生成类库，为目标对象生成动态的代理对象实例。

通过MethodInterceptor的invoke方法的MethodInvocation参数，我们可以控制对相应的Joinpoint的拦截行为。通过调用MethodInvocation的proceed（）方法，可以让程序执行继续沿着调用链传播。如果在哪一个MethodInterceptor中没有调用proceed（），那么程序的执行将会在当前MethodInterceptor处“短路”，Joinpoint上的调用链将被中断，同一Joinpoint上的其他MethodInterceptor的逻辑以及Joinpoint出的方法逻辑将不会被执行。

3. 反射

有一个叫Class的类，它是反射的源头，当我们编写一个类，编译完成后，在生成的.class文件中，就会产生一个Class对象，用于表示这个类的类型信息。Class类不是我们创建的，是由java虚拟机，在我们生成.class文件的时候创建的，Class 类的实例表示正在运行的 Java 应用程序中的类和接口

我们需要使用反射，就要获得Class这个类，Class.forName("cn.classes.OneClass");来获取，还有其他方式比如getClass。然后再根据Class类实例化。

bean.xml  
<bean id="id" class="com.xy.Student" />

Spring将采用的代码创建代码Java实例  
Class c = Class.forName("com.xy.Student");  
Object bean = c.newInstance();

1. Bean作用域

作用域 描述

|  |  |
| --- | --- |
| singleton | 在每个Spring IoC容器中一个bean定义对应一个对象实例。 |
| prototype | 一个bean定义对应多个对象实例。 |
| request | 在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例， 它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext 情形下有效。 |
| session | 在一个HTTP Session 中，一个bean定义对应一个实例。该作用域仅在基于web的SpringApplicationContext 情形下有效。 |
| global session | 在一个全局的HTTP Session 中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext 情形下有效。 |

1. Spring三种注入方式

接口注入（不推荐）；getter，setter方式注入（比较常用）；构造器注入（死的应用）

构造器注入：

<!--配置bean,配置后该类由spring管理-->

<bean name="springAction" class="com.bless.springdemo.action.SpringAction">

<!--(2)创建构造器注入,如果主类有带参的构造方法则需添加此配置-->

<constructor-arg ref="springDao"></constructor-arg>

<constructor-arg ref="user"></constructor-arg>

</bean>

<bean name="springDao" class=

"com.bless.springdemo.dao.impl.SpringDaoImpl"></bean>

<bean name="user" class="com.bless.springdemo.vo.User"></bean>

（Spring  auto-wire的 五种方式：

1：no   默认的方式是不进行自动装配，通过手工设置ref 属性来进行装配bean

2：byName   通过参数名 自动装配，如果一个bean的name 和另外一个bean的 property 相同，就自动装配。

3：byType   通过参数的数据类型自动自动装配，如果一个bean的数据类型和另外一个bean的property属性的数据类型兼容，就自动装配

4：construct   构造方法中的参数通过byType的形式，自动装配。

5：autodetect   如果有默认的构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。用于spring2.5 ，spring3.0测试不通过，估计是废弃了）：

1. **<beans>**
2. **<bean** id="orderItem" class="org.jia.OrderItem"**>**
3. **<property** name="orderdec" value="item00001"**></property>**
4. **</bean>**
5. **<bean** id="order" class="org.jia.Order" **>**
6. <!-----注入变量 名字必须与类中的名字一样------->
7. **<property** name="orderNum" value="order000007"**></property>**
8. **<**！--注入对象 名字为orderitem，所属的类的应用id为orderItem--**>**
9. **<property** name="orderitem" ref="orderItem"**></property>**
11. --**></bean>**
12. **</beans>**
14. **<beans>**
15. <!--此时的id就必须与Order.java中所定义的OrderItem的对象名称一样了，不然就会找不到-->
16. **<bean** id="orderitem" class="org.jia.OrderItem"**>**
17. **<property** name="orderdec" value="item00001"**></property>**
18. **</bean>**
19. **<bean** id="order" class="org.jia.Order"**<span** style="color:#ff0000;"**>** autowire="byName"**</span>>**
20. **<property** name="orderNum" value="order000007"**></property>**
21. **</bean>**
22. **</beans>**
23. <!—byName就是第二个bean不需要再**<property** name="orderitem" ref="orderItem"**></property>**
24. 而是根据 orderitem这个属性自动找name为orderitem的bean，需要Order类中的属性名字和要找的bean id相同才能找到，比如都是orderitem
25. -->

方式第三种注入：byType（如果容器中存在一个与指定属性类型相同的bean，那么将与  
                                  该属性自动装配；如果存在多个该类型bean，那么抛出异常，  
                                  并指出不能使用byType方式进行自动装配；如果没有找  
                                  到相匹配的bean，则什么事都不发生）

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd"**>**
3. **<beans>**
4. <!--按照byType注入则就与id没有关系，可以随便定义id ！！！但是不能出现多个此类的id-->
5. **<bean** id="orderitdfadafaem" class="org.jia.OrderItem"**>**
6. **<property** name="orderdec" value="item00001"**></property>**
7. **</bean>**
8. **<bean** id="order" class="org.jia.Order" **<span** style="color:#ff0000;"**>**autowire="byType"**</span>>**
9. **<property** name="orderNum" value="order000007"**></property>**
10. **</bean>**
11. **</beans>**
12. Bean继承依赖引用

<bean id="address"

p:city="Beijing" p:street="WuDaoKou" abstract="true"></bean> 抽象bean，不能实例化

<bean id="address2" class="com.coslay.beans.autowire.Address"

parent="address"></bean>

<bean id="person" class="com.coslay.beans.autowire.Person"

p:name="Tom" p:address-ref="address2" depends-on="car"></bean>

1. Bean创建过程

读取xml文件；获取文档下的bean节点；获取id属性值，获取class属性值；通过反射

Class.forName(beanDefinition.getClassName()).newInstance()完成bean实例化；通过getBean返回bean实例

1. AOP实现

1.通知(Advice):  
通知定义了切面是什么以及何时使用。描述了切面要完成的工作和何时需要执行这个工作。  
2.连接点(Joinpoint):  
程序能够应用通知的一个“时机”，这些“时机”就是连接点，例如方法被调用时、异常被抛出时等等。  
3.切入点(Pointcut)  
通知定义了切面要发生的“故事”和时间，那么切入点就定义了“故事”发生的地点，例如某个类或方法的名称，Spring中允许我们方便的用正则表达式来指定  
4.切面(Aspect)  
通知和切入点共同组成了切面：时间、地点和要发生的“故事”  
5.引入(Introduction)  
引入允许我们向现有的类添加新的方法和属性(Spring提供了一个方法注入的功能）  
6.目标(Target)  
即被通知的对象，如果没有AOP,那么它的逻辑将要交叉别的事务逻辑，有了AOP之后它可以只关注自己要做的事（AOP让他做爱做的事）  
7.代理(proxy)  
应用通知的对象，详细内容参见设计模式里面的代理模式  
8.织入(Weaving)  
把切面应用到目标对象来创建新的代理对象的过程，织入一般发生在如下几个时机:  
(1)编译时：当一个类文件被编译时进行织入，这需要特殊的编译器才可以做的到，例如AspectJ的织入编译器  
(2)类加载时：使用特殊的ClassLoader在目标类被加载到程序之前增强类的字节代码  
(3)运行时：切面在运行的某个时刻被织入,SpringAOP就是以这种方式织入切面的，原理应该是使用了JDK的动态代理技术

Spring提供了4种实现AOP的方式：  
1.经典的基于代理的AOP（用Spring提供的一个接口作为代理，把ref目标，不常用）  
2.@AspectJ注解驱动的切面  
3.纯POJO切面(常用)  
4.注入式AspectJ切面

注解：

package test.mine.spring.bean;  
  
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
@Aspect  
public class SleepHelper {  
  
    public SleepHelper(){  
          
    }  
      
    @Pointcut("execution(\* \*.sleep())")  
    public void sleeppoint(){}  
      
    @Before("sleeppoint()")  
    public void beforeSleep(){  
        System.out.println("睡觉前要脱衣服!");  
    }  
      
    @AfterReturning("sleeppoint()")  
    public void afterSleep(){  
        System.out.println("睡醒了要穿衣服！");  
    }  
      
}  
  
用@Aspect的注解来标识切面,注意不要把它漏了，否则Spring创建代理的时候会找不到它,@Pointcut注解指定了切点，@Before和@AfterReturning指定了运行时的通知，注  
  
意的是要在注解中传入切点的名称  
  
然后我们在Spring配置文件上下点功夫,首先是增加AOP的XML命名空间和声明相关schema  
命名空间:  
xmlns:aop="http://www.springframework.org/schema/aop"  
schema声明:  
http://www.springframework.org/schema/aop  
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd  
  
然后加上这个标签:  
<aop:aspectj-autoproxy/> 有了这个Spring就能够自动扫描被@Aspect标注的切面了

最后是运行，很简单方便了：  
public class Test {  
  
    public static void main(String[] args){  
        ApplicationContext appCtx = new ClassPathXmlApplicationContext("applicationContext.xml");  
        Sleepable human = (Sleepable)appCtx.getBean("human");  
        human.sleep();  
    }  
}

Spring来定义纯粹的POJO切面

前面我们用到了<aop:aspectj-autoproxy/>标签,Spring在aop的命名空间里面还提供了其他的配置元素:  
<aop:advisor> 定义一个AOP通知者  
<aop:after> 后通知  
<aop:after-returning> 返回后通知  
<aop:after-throwing> 抛出后通知  
<aop:around> 周围通知  
<aop:aspect>定义一个切面  
<aop:before>前通知  
<aop:config>顶级配置元素，类似于<beans>这种东西  
<aop:pointcut>定义一个切点  
  
我们用AOP标签来实现睡觉这个过程:  
代码不变，只是修改配置文件,加入AOP配置即可:  
<aop:config>  
    <aop:aspect ref="sleepHelper">  
    <aop:before method="beforeSleep" pointcut="execution(\* \*.sleep(..))"/>  
    <aop:after method="afterSleep" pointcut="execution(\* \*.sleep(..))"/>  
    </aop:aspect>  
</aop:config>

1. 事务

数据库系统提供了四种事务隔离级别供用户选择：  
  
A.Serializable（串行化）：一个事务在执行过程中完全看不到其他事务对数据库所做的更新（事务执行的时候不允许别的事务并发执行。事务串行化执行，事务只能一个接着一个地执行，而不能并发执行。）。  
  
B.Repeatable Read（可重复读）：一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，但是不能看到其他其他事务对已有记录的更新。  
  
C.Read Commited（读已提交数据）：一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，而且能看到其他事务已经提交的对已有记录的更新。  
  
D.Read Uncommitted（读未提交数据）：一个事务在执行过程中可以看到其他事务没有提交的新插入的记录，而且能看到其他事务没有提交的对已有记录的更新。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 丢失更新 | 脏读 | 非重复读 | 覆盖更新 | 幻像读 |
| 未提交读 | Y | Y | Y | Y | Y |
| 已提交读 | N | N | Y | Y | Y |
| 可重复读 | N | N | N | N | Y |
| 串行化 | N | N | N | N | N |

1. iBatis