Spring是分层的JavaSE/EE full-stack(一站式) 轻量级开源框架

Spring框架中包括JavaEE 三层的 每一层的解决方案 （一站式）

\* web层 SpringMVC

\* 业务层 Spring Bean管理、AOP、事务管理

\* 持久层 Spring JDBCTemplate、 ORM模块（整合其他持久层框架）

Spring 框架优点

\* 方便解耦，简化开发

Spring就是一个大工厂，可以将所有对象创建和依赖关系维护，交给Spring管理

\* AOP编程的支持

Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能

\* 声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无需手动编程

\* 方便程序的测试

Spring对Junit4支持，可以通过注解方便的测试Spring程序

\* 方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的直接支持

\* 降低JavaEE API的使用难度

Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低

Spring 包括哪些东西？

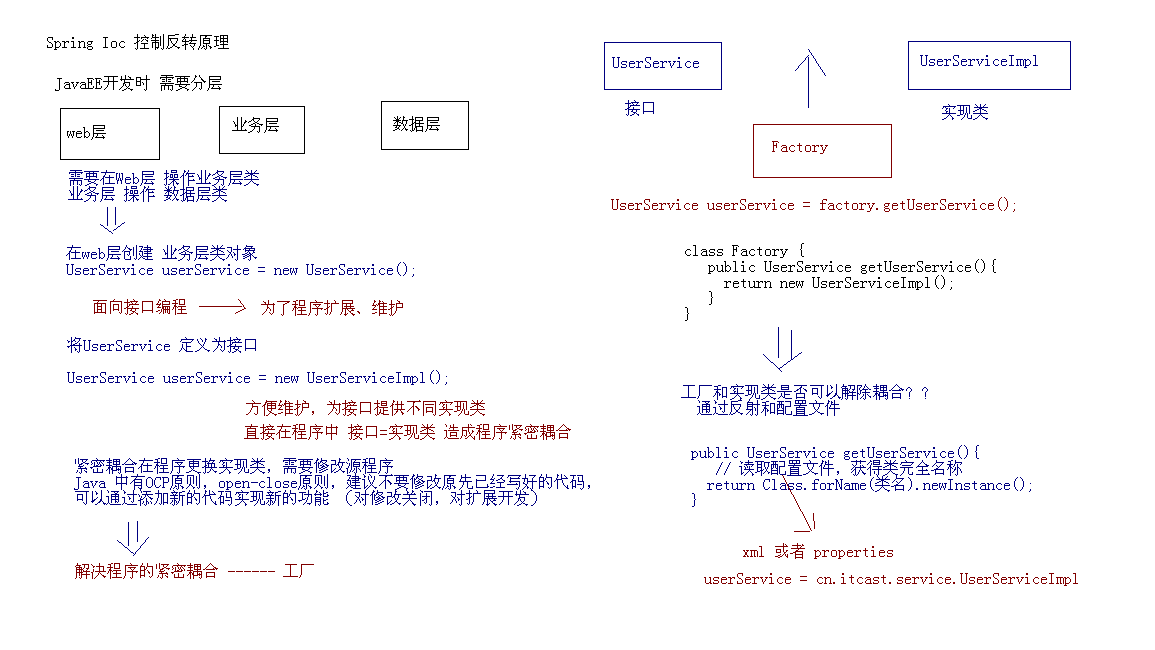
\* 核心技术 IoC 和 AOP

\* 数据访问 （持久层解决方案）

\* Web层解决方案 SpringMVC

\* 集成 （整合其他开源框架）





理解IoC控制反转和DI依赖注入

IoC Inverse of Control 反转控制的概念，就是将原本在程序中手动创建HelloService对象的控制权，交由Spring框架管理，简单说，就是创建HelloService对象控制权被反转到了Spring框架

DI：Dependency Injection 依赖注入，在Spring框架负责创建Bean对象时，动态的将依赖对象注入到Bean组件

面试题： IoC 和 DI的区别？

IoC 控制反转，指将对象的创建权，反转到Spring容器 ， DI 依赖注入，指Spring创建对象的过程中，将对象依赖属性通过配置进行注入

在程序中读取Spring配置文件，通过Spring框架获得Bean，完成相应操作

加载classpath （src）：

new ClassPathXmlApplicationContext("applicationContext.xml");

加载磁盘路径：

new FileSystemXmlApplicationContext("applicationContext.xml");

创建Spring 核心工厂对象

通过工厂的getBean 获得配置实例对象applicationContext.getBean("helloService");

BeanFactory和applicationContext的区别

BeanFactory采用延时加载，第一次getBean时才会初始化Bean

applicationContext在加载配置文件时加载Bean，并进行了扩展

国际化、事件传递、bean自动装配、各种不同应用层的Context

web项目使用webapplicationContext

IOC容器装配bean（xml）

spring提供3种配置bean的实例化方式

1. 类构造器 id class
2. 静态工厂 calss是工厂的完整类名，factory-method是工厂提供的方法
3. 实例工厂先创建工厂的实例在创建bean，在配置bean时用factory-bean和factory-method

bean的作用域

scope如果没有指定默认为singleton（单例）可以使用prototype（多例），request将bean放在request相当于request.setAttribute(),在同一个request获得同一个bean，还有session，globalSession一般用于Porlet（分布式系统）全局session

bean的生命周期

在配置 <bean> 元素，通过 init-method 指定Bean的初始化方法，通过 destroy-method 指定Bean销毁方法

\* destroy-method 只对 scope="singleton" 有效

\* 销毁方法，必须关闭ApplicationContext对象，才会被调用

ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");

applicationContext.close();

Bean的完整生命周期 （十一步骤）

1、instantiate bean对象实例化

2、populate properties 封装属性

3、如果Bean实现BeanNameAware 执行 setBeanName

4、如果Bean实现BeanFactoryAware 或者 ApplicationContextAware 设置工厂 setBeanFactory 或者上下文对象 setApplicationContext

5、如果存在类实现 BeanPostProcessor（后处理Bean） ，执行postProcessBeforeInitialization

6、如果Bean实现InitializingBean 执行 afterPropertiesSet

7、调用<bean init-method="init"> 指定初始化方法 init

8、如果存在类实现 BeanPostProcessor（处理Bean） ，执行postProcessAfterInitialization

9、执行业务处理

10、如果Bean实现 DisposableBean 执行 destroy

11、调用<bean destroy-method="customerDestroy"> 指定销毁方法 customerDestroy

第三步和第四步，使Bean 了解Spring容器

第五步和第八步，使用BeanPostProcessor 就是钩子函数，作用用来对Bean对象进行扩展，和动态代理结合

问题： 在userDAO对象所有方法上 添加运行时间监控

解决: 使用 BeanPostProcessor 完成

spring依赖注入

构造器、setter方法、接口（spring只支持前两种）

<constructor-arg index="" type="“ >

<property name="name" value="宝马"></property>

名称空间p 为了简化属性注入

p:<属性名>=”xxx“引入常量值

p:<属性名>-ref=”xxx”引入其他Bean对象

1. 引入p

xmlns:p=<http://www.springframework.org/schema/p>

1. 改写< property>为p名称空间注入

<bean id=" " class="" p:name="宝马" p:price="1000000"></bean>

3.0后引入了spel表达式

1、 完成对象之间注入

<property name="car2" ref="car2"></property>

改写为

<property name="car2" value="#{car2}"></property>

2、 使用另一个Bean属性完成注入

<bean id="carInfo" class="cn.itcast.spring.e\_di.CarInfo"></bean>

<bean id="car2\_2" class="cn.itcast.spring.e\_di.Car2">

<property name="name" value="#{carInfo.name}"></property>

</bean>

3、 使用另一个Bean方法完成注入

<bean id="carInfo" class="cn.itcast.spring.e\_di.CarInfo"></bean>

<bean id="car2\_2" class="cn.itcast.spring.e\_di.Car2">

<property name="name" value="#{carInfo.name}"></property>

<property name="price" value="#{carInfo.caculatePrice()}"></property>

</bean>

集合注入

spring提供专门标签完成 List、Set、Map、Properties 等集合元素属性注入

1） 注入List （数组）

<property name="hobbies">

<list>

<!-- <value>注入简单类型，<ref />注入复杂类型 -->

<value>体育</value>

</list>

</property>

2) 注入Set

<property name="numbers">

<set>

<value>10</value>

<value>6</value>

<value>15</value>

</set>

</property>

3） 注入Map

<property name="map">

<map>

<!--复杂类型 <entry key-ref="" value-ref=""></entry> -->

<entry key="name" value="itcast"></entry>

<entry key="address" value="北京"></entry>

</map>

</property>

4) 注入Properties

\* java.utils.Properties 类 继承 java.utils.HashTable

Properties key和value都是String类型

例如：

<property name="properties">

<props>

<prop key="company">传智播客</prop>

<prop key="pnum">100</prop>

</props>

</property>

使用注解开发spring

1、编写class，在声明上添加注解

@component（）//<bean id="" class=" "></bean>

1. 编写applicationContext.xml

引入context

<context:component-scan base-package=" "></context:component-scan>

@Repository用于对Dao实现类注解

@Service用于对Service实现

@Controller对Controller（表现层）

1. 属性注入

简单属性注入@value（）

复杂属性使用@Autowred默认按照类型注入

@Autowred结合@Qualifier（“”）通过名字注入

@Resource（name=””）注解完成复杂对象的装配

1. bean其它属性设置

指定bean的初始化方法和销毁方法（注解）

@PostConstruct=init-method

@PreDestroy=destroy-method

bean的作用范围

<bean scope="" />

@Scope 注解 ，默认作用域 singleton 单例

5、3.0注册bean的注解

@Configuration 指定POJO类为Spring提供Bean定义信息

@Bean 提供一个Bean定义信息

@Configuration

public class BeanConfig {

// 提供两个方法 获得Car和Product对象

@Bean(name = "car")

public Car initCar() {

Car car = new Car();

car.setName("大众");

car.setPrice(10000);

return car;

}

@Bean(name = "product")

public Product showProduct() {

Product product = new Product();

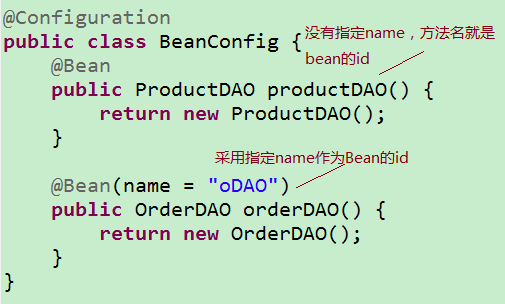
product.setPname("空调");

product.setPnum(100);

return product;

}

}



使用配置Bean 被Spring 扫描到 就可以了

混合使用

Bean 注册 通过XML完成

注入使用 @Autowired 注解完成

<context:annotation-config/> 启用四个注解 使@Resource、@ PostConstruct、@ PreDestroy、@Autowired注解生效

web使用spring

导入spring-web.jar

保存 ContextLoaderListener 完成在Servlet初始化阶段，加载Spring配置文件，将工厂对象放入 ServletContext

配置web.xml

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

默认读取 WEB-INF/applicationContext.xml

配置 全局参数 contextConfigLocation 指定 配置文件位置

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

WebApplicationContext applicationContext = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

aop 底层原理：代理

相关术语

Joinpoint(连接点):所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.

Pointcut(切入点):所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

Advice(通知/增强):所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

Introduction(引介):引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.

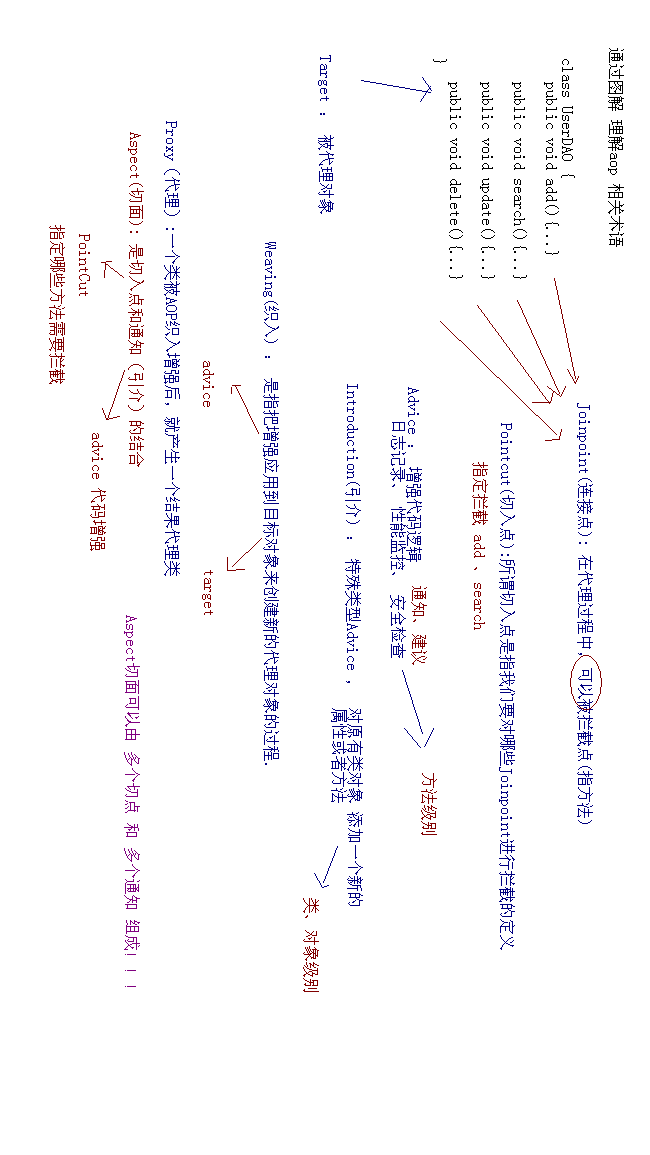
Target(目标对象):代理的目标对象

Weaving(织入):是指把增强应用到目标对象来创建新的代理对象的过程.

spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入

Proxy（代理）:一个类被AOP织入增强后，就产生一个结果代理类

Aspect(切面): 是切入点和通知（引介）的结合



aop的底层实现

CGLIB(Code Generation Library)是一个开源项目！

是一个强大的,高性能,高质量的Code生成类库,它可以在运行期扩展Java类与实现Java接口。Hibernate支持CGlib 来实现PO字节码的动态生成。Hibernate 默认PO 字节码生成技术 javassist

public class MyCglibProxy implements MethodInterceptor {

// 目标对象

private ProductDAO productDAO;

// 通过构造器 传入被代理对象

public MyCglibProxy(ProductDAO productDAO) {

this.productDAO = productDAO;

}

// 创建代理

public ProductDAO createCglibProxy() {

// 创建代理核心对象

Enhancer enhancer = new Enhancer();

// 设置被代理类 （为类创建子类）

enhancer.setSuperclass(productDAO.getClass());

// 设置回调函数

enhancer.setCallback(this);

// 返回代理 (返回代理子类对象)

return (ProductDAO) enhancer.create();

}

@Override

// 被代理对象所有方法执行 ，都会调用 intercept 方法

public Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) throws Throwable {

// 为 addProduct 计算运算时间

if (method.getName().equals("addProduct")) {// 当前执行方法

long start = System.currentTimeMillis();

Object result = methodProxy.invokeSuper(proxy, args);

long end = System.currentTimeMillis();

System.out.println("addProduct方法运行时间 : " + (end - start));

return result;

} else {

// 不进行增强

return methodProxy.invokeSuper(proxy, args);

}

}

}

传统Spring AOP 提供 五类 Advice

前置通知(代码增强) org.springframework.aop.MethodBeforeAdvice

\* 在目标方法执行前实施增强

后置通知 org.springframework.aop.AfterReturningAdvice

\* 在目标方法执行后实施增强

环绕通知 org.aopalliance.intercept.MethodInterceptor

\* 在目标方法执行前后实施增强

异常抛出通知 org.springframework.aop.ThrowsAdvice

\* 在方法抛出异常后实施增强

引介通知 org.springframework.aop.IntroductionInterceptor （课程不讲 了解）

\* 在目标类中添加一些新的方法和属性

Spring 的切面 Advisor

Advisor 就是对PointCut 应用 Advice （通常所说Advisor 指只有一个Point 和 一个 Advice ）

类型：

Advisor : 代表一般切面，Advice本身就是一个切面，对目标类所有方法进行拦截 （没有切点）

PointcutAdvisor : 代表具有切点的切面，可以指定拦截目标类哪些方法

IntroductionAdvisor : 代表引介切面，针对引介通知而使用切面（不要求掌握）

案例一（不带切点的切面） ： 使用普通Advisor， 使用Advice作为一个切面 ，不定义切点，拦截目标类 所有方法

1） 导入jar包

导入 aop联盟的规范 ： com.springsource.org.aopalliance-1.0.0.jar

导入 spring aop实现 ： spring-aop-3.2.0.RELEASE.jar

2） 编写被代理 接口和实现类

CustomerDAO

CustomerDAOImpl

3） 编写前置通知 （在目标方法前执行...）

MyMethodBeforeAdvice

4） 为目标对象创建 , 配置applicationContext.xml

使用ProxyFactoryBean 为目标对象创建代理

<!-- 被代理对象 -->

<bean id="customerDAO" class="cn.itcast.aop.c\_advisor.CustomerDAOImpl"></bean>

<!-- 增强 -->

<bean id="mybeforeadvice" class="cn.itcast.aop.c\_advisor.MyMethodBeforeAdvice"></bean>

<!-- 创建代理 -->

<bean id="customerDAOProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<!-- 目标 -->

<property name="target" ref="customerDAO"></property>

<!-- 针对接口代理 -->

<property name="proxyInterfaces" value="cn.itcast.aop.c\_advisor.CustomerDAO"></property>

<!-- 增强

interceptorNames 表示可以运用多个 Advice, 必须写value

-->

<property name="interceptorNames" value="mybeforeadvice"></property>

</bean>

注意事项： 在编程时，应该使用 ProxyFactoryBean 创建后代理对象(CustomerDAOProxy )， 不要引入原来Bean （CustomerDAO）

案例二 （带有切点的切面）： 定义单独切点切面，指定被代理对象 哪些方法 会被增强

\* JdkRegexpMethodPointcut 构造正则表达式切点

\* 使用正则表达式 切点切面 org.springframework.aop.support.RegexpMethodPointcutAdvisor

1) 创建被代理对象 （没有接口 的类 ）

OrderDAO

2) 增强 （编写环绕通知）

MyMethodInterceptor

3） 通过配置 ProxyFactory 为目标对象创建代理

<!-- 定义切点切面 -->

<bean id="myadvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<!-- 正则表达式规则 -->

<property name="pattern" value=".\*"></property>

<property name="advice" ref="mymethodinterceptor"></property>

</bean>

<!-- 创建代理 -->

<bean id="orderDAOProxy" class="org.springframework.aop.framework.ProxyFactoryBean" >

<!-- 目标 -->

<property name="target" ref="orderDAO"></property>

<!-- 针对类代理 -->

<property name="proxyTargetClass" value="true"></property>

<!-- 增强 -->

<property name="interceptorNames" value="myadvisor"></property>

</bean>

正则表达式案例

cn\.itcast\.aop\.d\_pointcutadvisor\.OrderDAO\.save.\* ---- 执行OrderDAO 的save方法

.\*save.\* ----- 包含save方法

<property name="patterns" value=".\*save.\*,.\*delete.\*"></property> ---- 同时增强save和delete方法

自动代理

使用ProxyFactoryBean 创建代理，需要为每个Bean 都配置一次 ，非常麻烦

自动代理原理： 根据xml中配置advisor的规则，得知切面对哪个类的哪个方法进行代理 （切面中本身就包含 被代理对象信息） ，就不需要ProxyFactoryBean ，使用BeanPostProcessor 完成自动代理

BeanNameAutoProxyCreator 根据Bean名称创建代理

DefaultAdvisorAutoProxyCreator 根据Advisor本身包含信息创建代理

\* AnnotationAwareAspectJAutoProxyCreator 基于Bean中的AspectJ 注解进行自动代理

1) BeanNameAutoProxyCreator

<!-- 第一种 BeanName自动代理 -->

<!-- 后处理Bean 不需要配置 id -->

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<!-- 对所有DAO结尾Bean 进行代理 -->

<property name="beanNames" value="\*DAO"></property>

<!-- 增强 -->

<property name="interceptorNames" value="mymethodinterceptor"></property>

</bean>

\*\*\* 自动代理和ProxyFactoryBean 本质区别 ：

ProxyFactoryBean， 先有被代理对象， 传递ProxyFactoryBean，创建代理

自动代理 ， 在Bean构造过程中， 使用后处理Bean 创建代理，返回构造完成对象就是代理对象

2） DefaultAdvisorAutoProxyCreator

基于切面信息进行代理

<!-- 切面 -->

<bean id="myadvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<!-- 切点拦截信息 -->

<property name="patterns" value="cn\.itcast\.aop\.d\_pointcutadvisor\.OrderDAO\.save.\*"></property>

<!-- 增强 -->

<property name="advice" ref="mybeforeadvice"></property>

</bean>

<!-- 第二种 基于切面信息自动代理 -->

<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></bean>

使用Aspectj实现aop

基于注解的@AspectJ