Spring第二天

**第二天知识点：**

第二天: IoC控制反转的注解配置管理、Spring Web集成、Spring Junit集成, Spring AOP面向切面编程底层原理

**第二天的主要内容（IoC相关）：**

Spring IoC容器装配Bean的配置（注解方式）

Spring Web集成

Spring Junit集成

AOP面向切面编程的相关概念（思想、原理、相关术语）

AOP编程底层实现机制（动态代理机制：JDK代理、Cglib代理）

**学习目标：**

掌握：什么是IoC，什么是Di，怎么通过spring装配Bean

了解：web集成和junit集成。

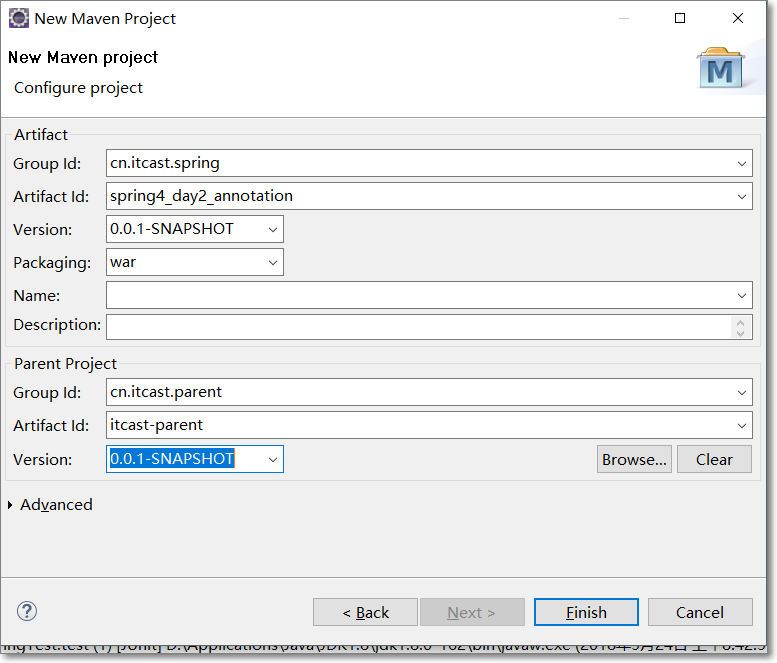
掌握: AOP的概念、思想、应用

动态代理（JDK代理、CGLIB代理）-了解-知道-会写--会用

# IoC容器装配Bean\_基于注解配置方式

## Bean的定义（注册） -- 扫描机制

新建web项目：spring4\_day02\_annotation



第一步：

引入依赖

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

导入log4j.properties,

导入applicationContext.xml

第二步： 编写Service和DAO

xml做法 ： <bean id=”customerService” class=”…” />，用<bean>的方式创建对象

注解做法 ： spring2.5引入 @Component 注解 如果放置到类的上面，相当于在spring容器中定义<bean id=”” class=””>

创建包：cn.itcast.spring.a\_ioc

创建类：CustomerService.java类

/\*\*

\* **@Component注解放置到类上**

\* 相当于spring容器中定义：<bean id="customerService" class="cn.itcast.spring.a\_ioc.CustomerService">

\* 其中id属性默认bean的名字是类名的小写

\* ——————————————————————————————————————————————————————

\* **@Component**(value="customerService")//自定义bean的名字

\* 相当于spring容器中定义：<bean id="customer" class="cn.itcast.spring.a\_ioc.CustomerService">

\* ——————————————————————————————————————————————————————

\*/

@Component(value="customerService")

**public** **class** CustomerService {

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

}

}

第三步： 配置注解开启和注解Bean的扫描。配置的示例如下：配置applicationContext.xml

参考：spring-framework-4.2.4.RELEASE/docs/spring-framework-reference/html/xsd-configuration.html，搜索context关键字即可

<?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"*>

<!-- 开启spring的注解功能 ：让注解有效了，识别注解-->

<context:annotation-config/>

<!-- 配置注解扫描

context:component-scan:专门扫描含有@Component注解的类，自动将其作为bean

base-package：要扫描包的路径,包含子包,cn.itcast.spring表示子包下的所有类定义注解都有效

注解扫描配置的时候，会自动开启注解功能

-->

<context:component-scan base-package=*"cn.itcast.spring"*/>

</beans>

第四步：测试：

**public** **class** SpringTest {

@Test

**public** **void** test(){

//spring容器

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

//获取bean

CustomerService customerService=(CustomerService) applicationContext.getBean("customerService");

customerService.save();

}

}

扩展优化：

1．注解扫描配置

在配置包扫描的时候，spring会自动开启注解功能，所以，注解开启功能可以不配置。

即去掉：<context:annotation-config/>

因为<context:componet-scan> 具有 <context:annotation-config> 作用 ！

2．衍生注解的问题

实际开发中，使用的是@Component三个衍生注解（“子注解”）

子注解的作用：有分层的意义（分层注解）。

Spring3.0为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。

除了@Component外，Spring提供了3个功能基本和@Component等效的注解

功能介绍

@Service用于标注业务层组件、（如Service层）

@Controller用于标注控制层组件（如struts中的action层,springMVC中的controller）

@Repository用于标注数据访问组件，（如DAO层组件）。

而@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。

第一步：

修改CutomerService.java

//@Component(value="customerService")//注释掉

@Service(value="customerService")

**public** **class** CustomerService {

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

}

}

创建CustomerDao.java

//持久层

@Repository("customerDao")

**public** **class** CustomerDao {

**public** **void** save(){

System.*out*.println("CustomerDao层被调用了");

}

}

问题：如果将Dao注入到Service呢？

回顾：如果使用xml的配置，那么可以使用setter方法进行注入

<bean id=”” class=””>

<property name=”” ref=””></property>

</bean>

## Bean属性的依赖注入

### 简单数据类型依赖注入（了解）

Spring3.0后，提供 @Value注解，可以完成简单数据的注入

//@Component(value="customerService")

@Service(value="customerService")

**public** **class** CustomerService {

//简单类型的成员变量

**@Value("Rose")**//参数的值简单类型

**private** String name="Jack";

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

System.*out*.println("name:"+name);

}

}

### 复杂类型数据依赖注入

下面完成，将Dao类的对象注入到Service类进行使用。

注解实现属性依赖注入，将注解加在setXxx方法上 或者 属性定义上 ！（任选其一，省代码了）

第一种： 使用@Value 结合SpEL ---- spring3.0 后用

//@Component(value="customer")

@Service(value="customer")

**public** **class** CustomerService {

//简单类型的成员变量

@Value("Rose")//参数的值简单类型

**private** String name="Jack";

//在属性声明上面注入，底层自动还是生成setCustomerDao()

//第一种： 使用@Value 结合SpEL ---- spring3.0 后用

//其中customerDao表示<bean>节点id的属性值

@Value("#{customerDao}")

**private** CustomerDao customerDao;

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

System.*out*.println("name:"+name);

customerDao.save();

}

}

第二种：使用@Autowired 结合 @Qualifier

单独使用@Autowired ，表示按照类型注入，会到spring容器中查找CustomerDao的类型，对应<bean class=””>，class的属性值，如果找到，可以匹配。

//第二种：使用spring的@Autowired

@Autowired//默认按照类型注入

**private** CustomerDao customerDao;

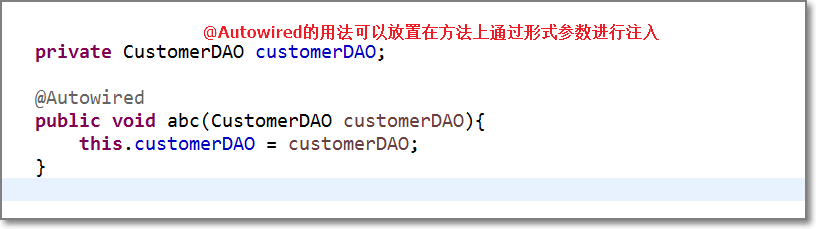
使用@Autowired + @ Qualifier 表示按照名称注入，回到spring容器中查找customerDao的名称，对应<bean id=””>，id的属性值，如果找到，可以匹配。

//第二种：使用spring的@Autowired 结合 @Qualifier

@Autowired//默认按照类型注入的

@Qualifier("customerDao")//必须配合@Autowired注解使用，根据名字注入

**private** CustomerDao customerDao;



第三种： JSR-250标准（基于jdk） 提供注解@Resource

单独使用@Resource注解，表示先按照名称注入，会到spring容器中查找customerDao的名称，对应<bean id=””>，id的属性值，如果找到，可以匹配。

如果没有找到，则会按照类型注入，会到spring容器中查找CustomerDao的类型，对应<bean class=””>，class的属性值，如果找到，可以匹配，如果没有找到会抛出异常。

//第三种： JSR-250标准（jdk） 提供@Resource

@Resource//默认先按照名称进行匹配，再按照类型进行匹配

**private** CustomerDao customerDao;

如果@Resource注解上添加name名称

使用@Resource注解，则按照名称注入，会到spring容器中查找customerDao的名称，对应<bean id=””>，id的属性值，如果找到，可以匹配。

如果没有找到，抛出异常。

//第三种： JSR-250标准（jdk） 提供@Resource

@Resource(name="customerDao")//只能按照customerDao名称进行匹配

**private** CustomerDao customerDao;

在实际开发过程中,第二种方式用的最多(推荐!)

## Bean的初始化和销毁

使用注解定义Bean的初始化和销毁

Spring初始化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法。

回顾配置文件的写法：<bean id=“foo” class=“...Foo” init-method=“setup”destory-method=“teardown”/>

注解的写法：

（1）当bean被载入到容器的时候调用setup ，

注解方式如下:

@PostConstruct

初始化

（2）当bean从容器中删除的时候调用teardown(scope= singleton有效)

注解方式如下:

@PreDestroy

销毁

使用 @PostConstruct 注解， 标明初始化方法 ---相当于 init-method 指定初始化方法

使用 @PreDestroy 注解， 标明销毁方法 ----相当于 destroy-method 指定对象销毁方法

第一步：创建类：LifeCycleBean.java，定义构造方法、初始化的方法、销毁的方法。

//测试生命周期过程中的初始化和销毁bean

@Component("lifeCycleBean")

**public** **class** LifeCycleBean {

**public** LifeCycleBean() {

System.*out*.println("LifeCycleBean构造器调用了");

}

//初始化后自动调用方法：方法名随意，但也不能太随便，一会要配置

@PostConstruct//初始化的方法

**public** **void** init(){

System.*out*.println("LifeCycleBean-init初始化时调用");

}

//bean销毁时调用的方法

@PreDestroy

**public** **void** destroy(){

System.*out*.println("LifeCycleBean-destroy销毁时调用");

}

}

第二步：配置文件，配置spring容器applicationContext.xml

<?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"*>

<!-- 配置注解扫描

context:component-scan:专门扫描含有@Component注解的类，自动将其作为bean

base-package：要扫描包的路径,包含子包,cn.itcast.spring表示子包下的所有类定义注解都有效

注解扫描配置的时候，会自动开启注解功能

-->

<context:component-scan base-package=*"cn.itcast.spring"*/>

</beans>

第三步：使用SpringTest.java完成测试

@Test

**public** **void** testLifeCycle() **throws** Exception{

//spring容器

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

//单例；此时初始化的方法已经被调用

LifeCycleBean lifeCycleBean = (LifeCycleBean)applicationContext.getBean("lifeCycleBean");

**applicationContext.close();**

}

注意：如果要执行对象的销毁方法

条件一： 单例Bean （在容器close时，单例Bean才会执行销毁方法 ）

条件二： 必须调用容器 close 方法

## Bean的作用域

通过@Scope注解，指定Bean的作用域（默认是 singleton 单例）

//测试生命周期过程中的初始化和销毁bean

@Component("lifeCycleBean")

//@Scope(value=ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

@Scope("prototype")//默认是单例(singleton),更改为多例(prototype)

**public** **class** LifeCycleBean {

}

## XML和注解混合配置 （重点）

一个项目中XML和注解都有（时代变迁,夹缝中的产物）

* Spring2.0 就有@Autowired注解
* Spring2.5 之后才有@Component注解

使用

XML 完成Bean定义

注解 完成Bean属性注入

创建包：cn.itcast.spring.b\_mixed

第一步：

（1）创建ProductDao类

//产品的数据层

**public** **class** ProductDao {

**public** **void** save(){

System.*out*.println("查询保存到数据口--数据层调用了");

}

}

（2）创建ProductService类

//产品的业务层

**public** **class** ProductService {

//注入dao

//强调：注入必须是bean注入bean

@Autowired

**private** ProductDao productDao;

//产品的保存

**public** **void** save(){

System.*out*.println("产品保存了，--业务层");

//调用dao层

productDao.save();

}

}

第二步：使用XML的方式完成Bean的定义

创建applicationContext-mixed.xml文件，定义：

<?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"*>

<!-- xml方式定义bean -->

<bean id=*"productDao"* class=*"cn.itcast.spring.b\_mixed.ProductDao"*/>

<bean id=*"productService"* class=*"cn.itcast.spring.b\_mixed.ProductService"*/>

<!-- 需要单独开启注解功能 -->

<context:annotation-config/>

</beans>

备注：这里配置 <context:annotation-config>

才能使用 @PostConstruct @PreDestroy @Autowired @Resource

<!-- 需要在spring容器中单独开启注解功能 -->

<context:annotation-config/>

提示：因为采用注解开发时， <context:component-scan> 具有<context:annotation-config>的功能 。

如果没有配置注解扫描，需要单独配置 <context:annotation-config>， 才能使用注解注入！

# Spring的web集成 (Spring监听器)

第一步： 新建web项目 spring4\_day02\_web ，导入jar包，导入applicationContext.xml和log4j.properties文件

第二步： 创建cn.itcast.spring.service包，

编写HelloService.java 业务类

//业务层

**public** **class** HelloService {

**public** **void** sayHello(){

System.*out*.println("嘿，传智播客。。。。");

}

}

将对象的创建，交给Spring容器管理，applicationContext.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"*>

<!-- HelloService的bean -->

<bean id=*"helloService"* class=*"cn.itcast.spring.service.HelloService"*></bean>

</beans>

第三步： 创建cn.itcast.spring.servlet包，编写HelloServlet.java ，调用HelloService类

使用ApplicationContext applicationContext = new ClassPathXmlApplicationContext()的方式加载spring容器;

**public** **class** HelloServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

//传统方式：

//new service

//HelloService helloService = new HelloService();

//spring方式：只要看到new，你就想到spring容器中的<bean>

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

HelloService helloService=(HelloService)applicationContext.getBean("helloService");

helloService.sayHello();

}

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response)

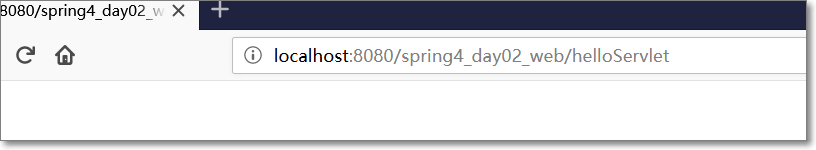
**throws** ServletException, IOException {

**this**.doGet(request, response);

}

}

第四步：将程序部署到tomcat测试：



【思考、阅读】直接new ClassPathXmlApplicationContext()有什么缺点？

缺点：在创建Spring容器同时，需要对容器中对象初始化。而每次初始化容器的时候，都创建了新的容器对象，消耗了资源，降低了性能。

解决思路：保证容器对象只有一个。

解决方案：将Spring容器绑定到Web Servlet容器上，让Web容器来管理Spring容器的创建和销毁。

分析：ServletContext在Web服务运行过程中是唯一的， 其初始化的时候，会自动执行ServletContextListener 监听器 （用来监听上下文的创建和销毁），具体步骤为：

编写一个ServletContextListener监听器，在监听ServletContext到创建的时候，创建Spring容器，并将其放到ServletContext的属性中保存（setAttribute(Spring容器名字，Spring容器对象) ）。

我们无需手动创建该监听器，因为Spring提供了一个叫ContextLoaderListener的监听器，它位于spring-web-4.3.13.RELEASE.jar中。

开发步骤：

第一步：引入spring-web的依赖

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

</dependency>

第二步：在web.xml 配置Spring的核心监听器

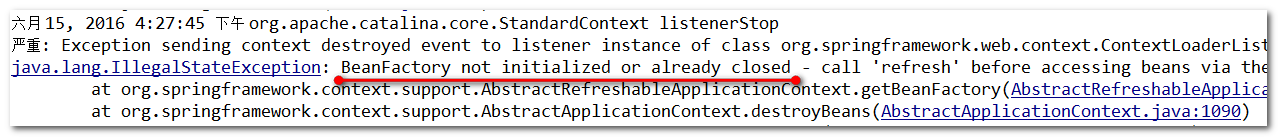
<!-- spring的核心监听器 -->

<listener>

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

</listener>

第三步：启动tomcat服务器，结果发现异常，因为默认会加载

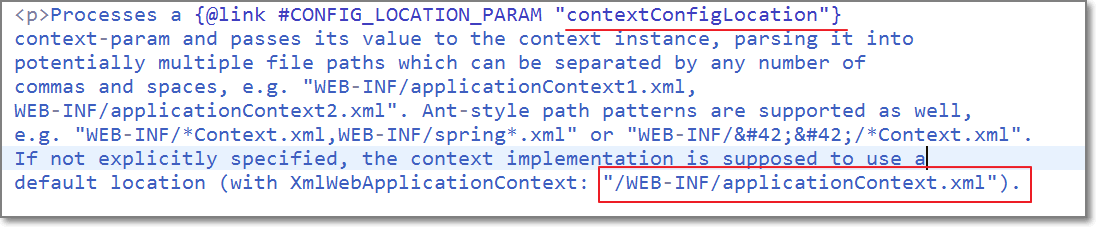


根据异常提示：发现spring的BeanFactory没有初始化，说明没有找到spring容器，即applicationContext.xml文件

第四步：在web容器中配置spring文件路径

为什么没有找到applicationContext.xml文件呢？因为此时加载的是WEB-INF/applicationContext.xml，而不是src下的applicationContext.xml文件

原因：找到ContextLoaderListener.class，再找到ContextLoader.class，发现默认加载的WEB-INF/applicationContext.xml



解决方案：需要在web.xml中配置，加载spring容器applicationContext.xml文件的路径

<!-- spring的核心监听器 -->

<listener>

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

</listener>

<!-- 全局参数变量 -->

<context-param>

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

<!-- applicationContext.xml文件的位置，使用classpath定义 -->

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

</context-param>

重新启动tomcat服务器，没有异常，问题解决。

第五步：修改Servlet代码。在Servlet 中通过ServletContext 获取Spring容器对象

第一种方式： 使用getAttribute

//每次获取的都是一个spring容器

ApplicationContext applicationContext =

(ApplicationContext)**this**.getServletContext().getAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*);

第二种方式：使用WebApplicationContextUtils （推荐）

//工具类

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

# Spring的junit测试集成

Spring提供spring-test-4.3.13.RELEASE 可以整合junit。

优势：可以简化测试代码（不需要手动创建上下文）

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.13.RELEASE</version>

</dependency>

使用spring和junit集成

第一步： 通过@RunWith注解，使用junit整合spring

通过@ContextConfiguration注解，指定spring容器的位置

//目标：测试一下spring的bean的某些功能

@RunWith(SpringJUnit4ClassRunner.**class**)//junit整合spring的测试//立马开启了spring的注解

@ContextConfiguration(locations="classpath:applicationContext.xml")//加载核心配置文件，自动构建spring容器

**public** **class** SpringTest {

//使用注解注入要测试的bean

@Autowired

**private** HelloService helloService;

@Test

**public** **void** testSayHello(){

helloService.sayHello();

}

}

上述代码表示：在测试类运行前的初始化的时候，会自动创建ApplicationContext对象

第二步： 通过@Autowired注解，注入需要测试的对象

在这里注意2点：

1. 将测试对象注入到测试用例中
2. 测试用例不需要配置<context:annotion-config/>，因为使用测试类运行的时候，会自动启动注解的支持。

//使用注解注入要测试的bean

@Autowired

**private** HelloService helloService;

第三步：调用测试方法完成测试

@Test

**public** **void** testSayHello(){

helloService.sayHello();

}

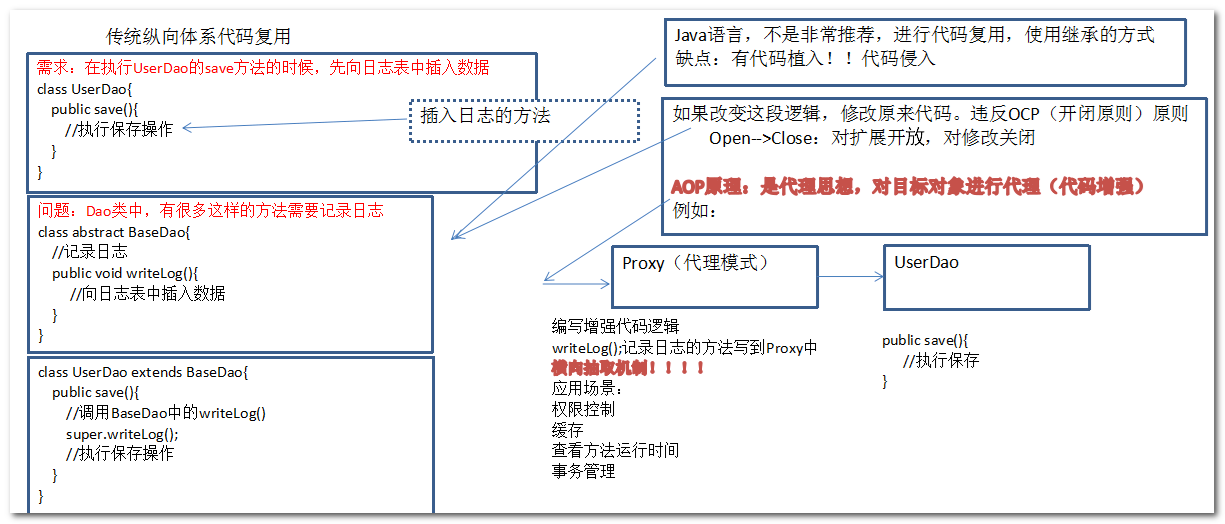
# AOP面向切面编程的相关概念

## 什么是AOP ？

AOP (Aspect Oriented Programing) 称为：面向切面编程，它是一种编程思想。

AOP采取**横向抽取机制**，取代了**传统纵向继承**体系重复性代码的编写方式（例如性能监视、事务管理、安全检查、缓存、日志记录等）。

【扩展了解】AOP 是 OOP（面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）是一种计算机编程架构），思想延续 ！



AOP 思想： 基于代理思想，对原来目标对象，创建代理对象，在不修改原对象代码情况下，通过代理对象，调用增强功能的代码，从而对原有业务方法进行增强 ！

切面：需要代理一些方法和增强代码 。

## AOP的应用场景

场景一： 记录日志

场景二： 监控方法运行时间 （监控性能）

场景三： 权限控制

场景四： 缓存优化 （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

场景五： 事务管理 （调用方法前开启事务， 调用方法后提交或者回滚、关闭事务 ）

## Spring AOP编程两种方式

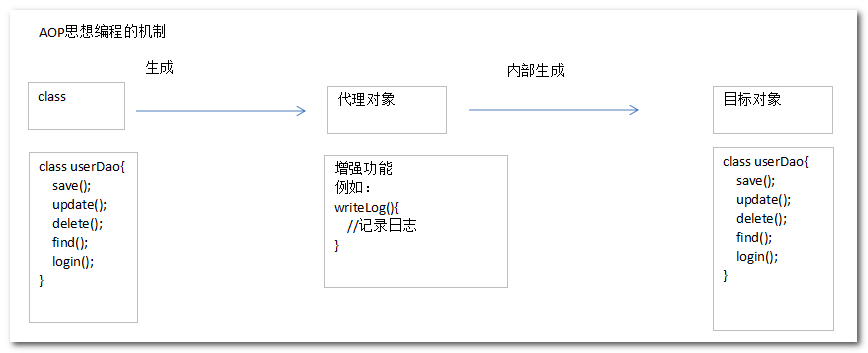
* Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类植入增强代码。
* AsPectJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持。

简单的说，Spring内部支持两套AOP编程的方案：

* Spring 1.2 开始支持AOP编程 （传统SpringAOP 编程），编程非常复杂 ---- 更好学习Spring 内置传统AOP代码
* Spring 2.0 之后支持第三方 AOP框架（AspectJ ），实现另一种 AOP编程 -- 推荐

## AOP编程相关术语

**AOP思想编程的机制**



**AOP的相关术语**

**Aspect(切面):** 是通知和切入点的结合,通知和切入点共同定义了关于切面的全部内容---它的功能、在何时和何地完成其功能

**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.

通知定义了切面的”什么”和”何时”，切入点就定义了”何地”.

**Advice(通知):**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)

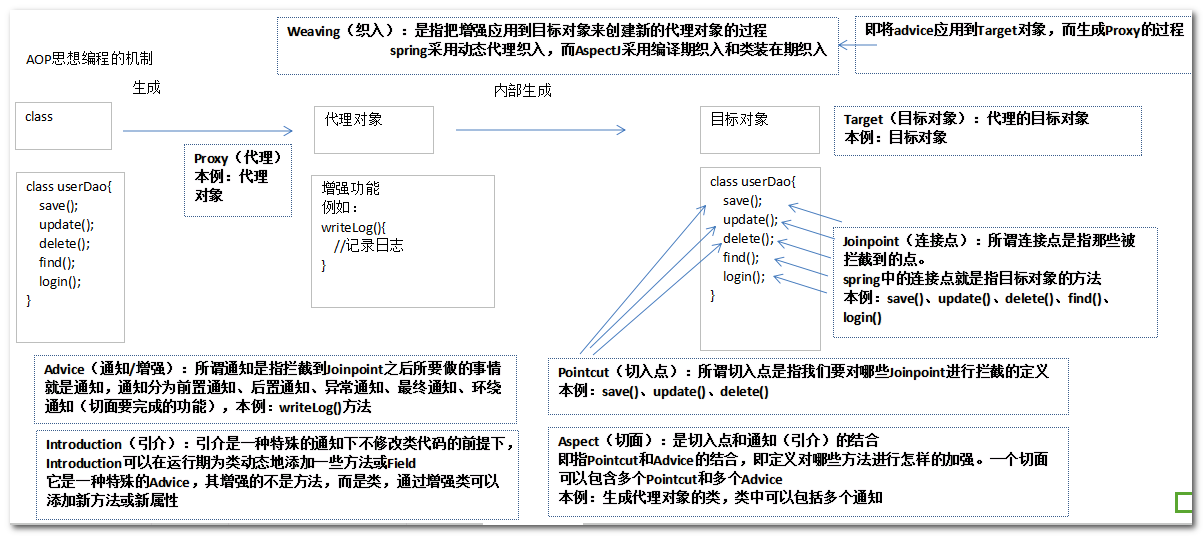
**Target(目标对象):**代理的目标对象

**(织入):**是指把切面应用到目标对象来创建新的代理对象的过程.切面在指定的连接点织入到目标对象

**Introduction(引入):**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.

**通过案例解释AOP相关概念**

需求：UserDao中有5个方法，分别是save()、update()、delete()、find()、login()；在访问UserDao的save()、update()、delete()方法之前，进行记录日志的操作。



Aspect切面（类）：增强代码 Advice（writeLog方法）和 切入点 Pointcut（save，update，delete） 的结合。换句话说：对哪些方法进行怎样的代码增强。

# AOP编程底层实现机制（了解）

AOP 就是要对目标进行代理对象的创建， Spring AOP是基于动态代理的，基于两种动态代理机制： JDK动态代理和CGLIB动态代理

## JDK动态代理

JDK动态代理，针对目标对象的接口进行代理 ，动态生成接口的实现类 ！（必须有接口）

【过程要点】：

1、 必须对接口生成代理

2、 采用Proxy类，通过newProxyInstance方法为目标创建代理对象。

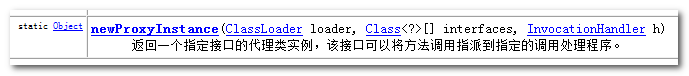
该方法接收三个参数 ：

（1）目标对象类加载器

（2）目标对象实现的接口

（3）代理后的处理程序InvocationHandler

使用 Proxy类提供 newProxyInstance 方法对目标对象接口进行代理



参数说明：

loader：定义代理类的类加载器

interfaces：代理类要实现的接口列表

h:指派方法调用的调用处理程序

3、 实现InvocationHandler 接口中 invoke方法，在目标对象每个方法调用时，都会执行invoke

【示例】：需求：对目标对象中存在保存和查询的方法，在执行保存的方法的时候，记录日志

第一步：新建web工程：spring4\_day02\_proxy。创建包cn.itcast.spring.a\_proxy包

第二步：编写业务接口，接口中定义save()和find()的方法。

//接口（表示代理的目标接口）

**public** **interface** ICustomerService {

//保存

**public** **void** save();

//查询

**public** **int** find();

}

第三步：编写业务类，类要实现接口

//实现类

**public** **class** CustomerServiceImpl **implements** ICustomerService{

**public** **void** save() {

System.*out*.println("客户保存了。。。。。");

}

**public** **int** find() {

System.*out*.println("客户查询数量了。。。。。");

**return** 100;

}

}

第四步：使用JDK代理完成

代理工厂：

有三种方案完成JDK动态代理：

方案一：在内部实现new InvocationHandler()，指定匿名类

//专门用来生成jdk的动态代理对象的-通用

**public** **class** JdkProxyFactory{

//成员变量

**private** Object target;

//注入target目标对象

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyObject(){

//参数1：目标对象的类加载器

//参数2：目标对象实现的接口

//参数3：回调方法对象

/\*\*方案一：在内部实现new InvocationHandler()，指定匿名类\*/

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(), **new** InvocationHandler(){

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象方法的返回值

**return** object;

}

});

}

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

}

方案二：传递内部类的对象，指定内部类

//专门用来生成jdk的动态代理对象的-通用

**public** **class** JdkProxyFactory{

//成员变量

**private** Object target;

//注入target

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyObject(){

//参数1：目标对象的类加载器

//参数2：目标对象实现的接口

//参数3：回调方法对象

/\*\*方案二：传递内部类的对象，指定内部类\*/

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(),**new** **MyInvocationHandler**());

}

//自己制定内部类:类的内部可以多次使用类型

**private** **class** **MyInvocationHandler** **implements** InvocationHandler{

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象方法的返回值

**return** object;

}

}

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

}

方案三：直接使用调用类作为接口实现类，实现InvocationHandler接口

//专门用来生成jdk的动态代理对象的-通用

**public** **class** JdkProxyFactory **implements InvocationHandler**{

//成员变量

**private** Object target;

//注入target

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyObject(){

//参数1：目标对象的类加载器

//参数2：目标对象实现的接口

//参数3：回调方法对象

/\*\*方案三：直接使用调用类作为接口实现类，实现InvocationHandler接口\*/

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(),**this**);

}

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

//参数1：代理对象

//参数2：目标的方法对象

//参数3：目标的方法的参数

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象

**return** object;

}

}

第五步：使用SpringTest.java进行测试

//目标：使用动态代理，对原来的方法进行功能增强,而无需更改原来的代码。

//JDK动态代理：基于接口的（对象的类型，必须实现接口！）

@Test

**public** **void** testJdkProxy(){

//target（目标对象）

ICustomerService target = **new** CustomerServiceImpl();

//实例化注入目标对象

JdkProxyFactory jdkProxyFactory = **new** JdkProxyFactory(target);

//获取Proxy Object代理对象:基于目标对象类型的接口的类型的子类型的对象

ICustomerService **proxy** = (ICustomerService)jdkProxyFactory.getProxyObject();

//调用目标对象的方法

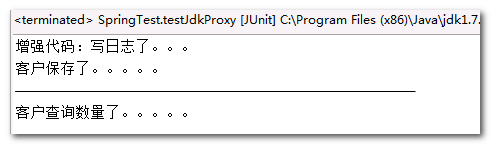
proxy.save();

System.*out*.println("————————————————————————————————————————");

proxy.find();

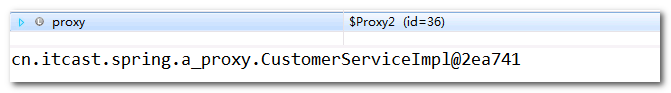
}

第六步：在控制台查看输出结果



从结果上看出：在保存方法的前面，输入了日志增强。

最后，使用断点查看JDK代理，生成的代理对象



JDK原理分析:

Interface ICustomerService{

//目标接口

}

Class CustomerServiceImpl implements ICustomerService{

//目标类实现接口

}

JDK代理对接口代理

Class $Proxy2 implements ICustomerService{

//JDK代理类是目标接口的实现类

**ICustomerService** **customerService = new CustomerServiceImpl();**

public void save() {

**writeLog()**

**customerService.save();**

}

public int find() {

**int returnValue = customerService.find();**

**return returnValue;**

}

//记录日志

private static void writeLog(){

System.out.println("增强代码：写日志了。。。");

}

}

注意：

JDK动态代理的缺点： 只能面向接口代理，不能直接对目标类进行代理 ，如果没有接口，则不能使用JDK代理。

## Cglib动态代理

Cglib的引入为了解决类的直接代理问题(生成代理子类)，不需要接口也可以代理 ！

什么是cglib ？

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。



该代理方式需要相应的jar包，但不需要导入。因为Spring core包已经包含cglib ，而且同时包含了cglib 依赖的asm的包（动态字节码的操作类库）

【示例】：需求：对目标对象中存在保存和查询的方法，在执行保存的方法的时候，记录日志

第一步： 引入依赖spring-context

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

spring-core中已经包含了cglib

第二步：编写业务类，创建类ProductService.java，类不需要实现接口

//没有接口的类

**public** **class** ProductService {

**public** **void** save() {

System.*out*.println("商品保存了。。。。。");

}

**public** **int** find() {

System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

第三步：使用cglib代理，创建类CglibProxyFactory.java

//cglib动态代理工厂:用来生成cglib代理对象

**public** **class** CglibProxyFactory **implements** **MethodInterceptor**{

//声明一个代理对象引用

**private** Object target;

//注入代理对象

**public** CglibProxyFactory(Object target) {

**this**.target = target;

}

//获取代理对象

**public** Object getProxyObject(){

//1.代理对象生成器(工厂思想)

Enhancer enhancer = **new** Enhancer();

//2.在增强器上设置两个属性

//设置要生成代理对象的目标对象：生成的目标对象类型的子类型

enhancer.setSuperclass(target.getClass());

//设置回调方法

enhancer.setCallback(**this**);

// Callback

//3.创建获取对象

**return** enhancer.create();

}

//回调方法（代理对象的方法）

//参数1：代理对象

//参数2：目标对象的方法对象

//参数3：目标对象的方法的参数的值

//参数4：代理对象的方法对象

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

MethodProxy methodProxy) **throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象方法的执行结果

**return** object;

}

//写日志的增强功能

//Advice通知、增强

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

}

第四步：测试代码，使用SpringTest.java进行测试

//cglib动态代理：可以基于类（无需实现接口）生成代理对象

@Test

**public** **void** testCglibProxy(){

//target目标：

ProductService target = **new** ProductService();

//weave织入，生成proxy代理对象

//代理工厂对象，注入目标

CglibProxyFactory cglibProxyFactory = **new** CglibProxyFactory(target);

//获取proxy:思考：对象的类型

//代理对象，其实是目标对象类型的子类型

ProductService proxy=(ProductService) cglibProxyFactory.getProxyObject();

//调用代理对象的方法

proxy.save();

System.*out*.println("————————————————————————————————————————");

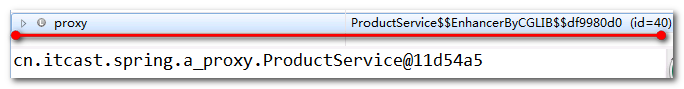
proxy.find();

}

第五步：控制台输出结果



最后，使用断点查看cglib代理，生成的代理对象



Cglib代理原理分析:

Class ProductService{

//目标类

}

Cglib对类代理

Class ProductService$$EnhancerByCGLIB$$df9980d0 extends ProductService{

//CGLIB代理类是目标类的子类

**ProductService** **productService= new ProductService();**

public void save() {

**writeLog()**

**productService.save();**

}

public int find() {

**int returnValue = productService.find();**

**return returnValue;**

}

//记录日志

private static void writeLog(){

System.out.println("增强代码：写日志了。。。");

}

}

## 代理知识小结

区别：

* Jdk代理：基于接口的代理，一定是基于接口，会生成目标对象的接口类型的子对象。
* Cglib代理：基于类的代理，不需要基于接口，会生成目标对象类型的子对象。

代理知识总结：

* spring在运行期，生成动态代理对象，不需要特殊的编译器.
* spring有两种代理方式：

1.若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。

2.若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。

* 使用该方式时需要注意:

1.对接口创建代理优于对类创建代理，因为会产生更加松耦合的系统，所以spring默认是使用JDK代理。

对类代理是让遗留系统或无法实现接口的第三方类库同样可以得到通知，这种方式应该是备用方案。

2.标记为final的方法不能够被通知。spring是为目标类产生子类。任何需要被通知的方法都被复写，将通知织入。final方法是不允许重写的。

3.spring只支持方法连接点：不提供属性接入点，spring的观点是属性拦截破坏了封装。

面向对象的概念是对象自己处理工作，其他对象只能通过方法调用的得到的结果。

提示：

* Spring AOP 优先对接口进行代理 （使用Jdk动态代理）
* 如果目标对象没有实现任何接口，才会对类进行代理 （使用cglib动态代理）

知识点梳理：

1. 注解方式装配bean对象
2. 混合配置
3. web集成，配置Spring监听器 (ContextLoaderListener)，webappp…util.get(context)
4. 测试集成 (@runwith @ContextConfiguration(核心配置))---熟悉
5. aop的原理（动态代理：jdk，cglib）
6. jdk代理和cglib代理 (理解代理对象的工作原理)