# Spring

内容：

Part I：Spring入门、IoC、Spring Web集成、Spring Junit集成

Part II： AOP面向切面编程、AspectJ集成配置、

Part III：JdbcTemplate、Spring声明式事务管理、

Part IV：ssh整合、综合练习SSH、jQuery Ajax、分页

**PartⅠ**

PartⅠ知识点：

Spring的概述、IoC和DI概念

Spring IoC快速入门

Spring IoC容器装配Bean的配置（XML方式、注解方式）

Spring Web集成

Spring Junit集成

# Spring概述

官方：https://spring.io/

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

* 分层

Spring遵循了JavaEE规范的三层结构，并对三层都提供了解决方案。经典三层MVC结构：

* 表现层（页面数据显示、页面跳转调度），例如jsp/servlet/struts1/struts2/springMVC
* 业务层（业务处理和逻辑实现、事务控制），例如JavaBean、EJB、spring
* 持久层（数据crud、和数据库打交道），例如jdbc/hibernate/mybatis
* 一站式

Spring对JavaEE三层都提供了解决方案，并且支持三层中的第三方框架整合：

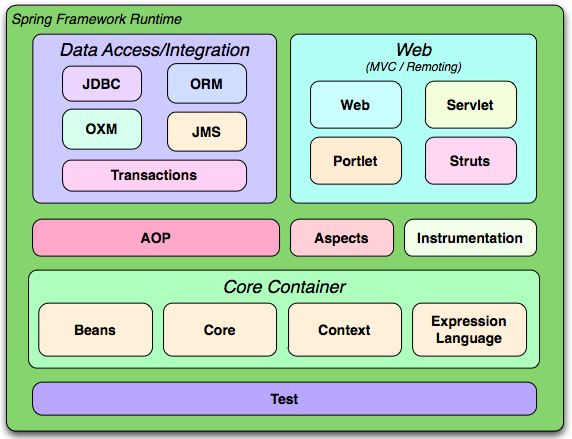
* 表现层：struts1、struts2、Spring MVC
* 业务层：Ioc、AOP、事务控制
* 持久层：JdbcTemplate、HibernateTemplate、Spring Data JPA、ORM框架（对象关系映射）、Mybatis
* 轻量级

Spring的出现取代了EJB的臃肿、低效、繁琐复杂。且使用spring编程是非侵入式的。

## **Spring体系结构**

Spring框架是个分层架构，包含一系列的功能要素，被分为约20个模块（7大块）

分别为Core Container、Data Access/Integration、Web、AOP（Aspect Oriented Programming）、Aspects、Instrumentation和test部分



* **核心容器(Core Container)**

包括Core、Beans、Context、EL模块

1：Core和Beans模块提供了Spring最基础的功能，提供IoC和依赖注入DI特性。这里的基础概念是BeanFactory，它提供对工厂模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

2：Context模块基于Core和Beans来构建，它提供了用一种框架风格的方式来访问对象，有些像JNDI注册表。Context封装包继承了beans包的功能，还增加了国际化（I18N）,事件传播，资源装载，以及透明创建上下文，例如通过servlet容器，以及对大量JavaEE特性的支持，如EJB、JMX。核心接口是ApplicationContext。

3：Expression Language，表达式语言模块，提供了在运行期间查询和操作对象图的强大能力。支持访问和修改属性值，方法调用，支持访问及修改数组、容器和索引器，命名变量，支持算数和逻辑运算，支持从Spring 容器获取Bean，它也支持列表投影、选择和一般的列表聚合等。

* **数据访问/集成部分(Data Access/Integration)**

1：JDBC模块，提供对JDBC的抽象，它可消除冗长的JDBC编码和解析数据库厂商特有的错误代码。

2：ORM模块，提供了常用的"对象/关系"映射APIs的集成层。 其中包括JPA、JDO、Hibernate 和 iBatis 。利用ORM封装包，可以混合使用所有Spring提供的特性进行"对象/关系"映射，如简单声明性 事务管理 。

3：OXM模块，提供一个支持Object和XML进行映射的抽象层，其中包括JAXB、Castor、XMLBeans、JiBX和XStream。

4：JMS模块，提供一套"消息生产者、消费者"模板用于更加简单的使用JMS，JMS用于用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

5：Transaction模块，支持程序通过简单声明性 事务管理，只要是Spring管理对象都能得到Spring管理事务的好处，即使是POJO，也可以为他们提供事务。

* **Web**

1：Web模块，提供了基础的web功能。例如多文件上传、集成IoC容器、远程过程访问、以及Web Service支持，并提供一个RestTemplate类来提供方便的Restful services访问

2：Web-Servlet模块，提供了Web应用的Model-View-Controller（MVC）实现。Spring MVC框架提供了基于注解的请求资源注入、更简单的数据绑定、数据验证等及一套非常易用的JSP标签，完全无缝与Spring其他技术协作。

3：Web-Struts模块， 提供了对Struts集成的支持，这个功能在Spring3.0里面已经不推荐了，建议你迁移应用到使用Struts2.0或Spring的MVC。

4：Web-Portlet模块，提供了在Portlet环境下的MVC实现

* **AOP**

1：AOP模块，提供了符合AOP 联盟规范的面向方面的编程实现，让你可以定义如方法拦截器和切入点，从逻辑上讲，可以减弱代码的功能耦合，清晰的被分离开。而且，利用源码级的元数据功能，还可以将各种行为信息合并到你的代码中 。

2：Aspects模块，提供了对AspectJ的集成。

3：Instrumentation模块， 提供一些类级的工具支持和ClassLoader级的实现，可以在一些特定的应用服务器中使用。

* **Test**

1：Test模块，提供对使用JUnit和TestNG来测试Spring组件的支持，它提供一致的ApplicationContexts并缓存这些上下文，它还能提供一些mock对象，使得你可以独立的测试代码。

## **Spring核心**

**IOC**（Inverse of Control控制反转）：

将对象创建权利交给Spring工厂管理。由Book book = new Book()变为Book book = Spring工厂.getBook()。工厂模式 + 反射 + xml中class路径配置

**AOP**（Aspect Oriented Programming面向切面编程）：基于动态代理的功能增强方式

## **Spring的优点**

Spring 的出现是为了解决JavaEE的实际问题。俗称Spring为开发架构的粘合剂

* （1）方便解耦，简化开发

所有对象的创建和依赖关系维护，都可以交给Spring管理

* （2）AOP编程

Spring提供面向切面编程，方便实现对程序的权限拦截、运行监控等功能

* （3）声明式事务

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

* （4）方便程序测试

Spring对Junit4支持，通过注解可以方便地测试Spring程序

* （5）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。

* （6）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

# IOC快速入门

* 下载并导入jar包
* 编写代码（基础代码和调用代码）
* 编写XML配置文件

## **Spring的jar包**

开发包下载：http://repo.spring.io/libs-release-local/org/springframework/spring/

以5.0.5GA版本为例：[spring-framework-5.0.5.RELEASE-dist.zip](http://repo.spring.io/libs-release-local/org/springframework/spring/5.0.5.RELEASE/spring-framework-5.0.5.RELEASE-dist.zip) 目录结构：

docs：API文档、开发指南docs\spring-framework-reference\pdf

libs：jar包

schema：schema文件-xml配置文件的xsd约束

对开发环境的Minimum requirements：

JDK 8+ for Spring Framework 5.x

JDK 6+ for Spring Framework 4.x

JDK 5+ for Spring Framework 3.x

其中使用spring 3.x和Java8不兼容

这里采用：4.3.2版本

开发过程中还需要其他开源技术框架依赖Jar包集

## **搭建项目**

* 第一步：新建Web工程。

以Eclipse为例，新建Dynamic Web Project（修改项目build文件的放置位置，从软件默认的build/classes，修改为传统的WebContent/WEB-INF/classes）

* 第二步：导入jar包

1. Spring项目的核心容器的最基本Jar包（4个）：

spring-framework-4.32.RELEASE-dist\spring-framework-4.3.2.RELEASE\libs下：beans、core、context、EL

1. Spring框架所需的日志包（2个）：

默认采用apache commons-logging(JCL)日志框架+log4j的日志系统实现，还需要添加log4j的配置文件

commons-logging-1.2.jar、log4j-1.2.17.jar

1. 添加log4j.properties到src下

### direct log messages to stdout ###

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.err

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} [method]%l%n%m%n

### direct messages to file mylog.log ###

log4j.appender.file=org.apache.log4j.FileAppender

log4j.appender.file.File=c\:mylog.log

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} [method]%l%n%m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=info, stdout

此时：

Lib：6个jar

src：1个日志配置文件log4j.properties

关于java日志的补充阅读：

日志框架：commons-logging（JCL）、slf4j

日志系统实现：log4j、JUL、simplelog、logback、log4j2

|  |
| --- |
| Java的常见的日志系统有log4J，jdk-jul，logback等，这些日志系统各自独立，编程方式也不一致。如果你一个系统要用到多个框架库，而这些框架库又可能要用不同的日志系统，那么你一个系统中就得使用多套日志系统，那多套日志系统如何融合在一起是个很大的问题。  那么如何解决呢？进行抽象，抽象出一个接口层，对每个日志实现都适配或者转接，这样这些提供给别人的库都直接使用抽象层即可，不需要关注具体的日志实现。常见的日志抽象出来的框架有Apache commons-logging和slf4j。  最先是开源社区提供了commons-logging抽象，被称为JCL日志框架，出色地完成了兼容主流的日志实现（log4j、JUL、simplelog），基本一统江湖，就连spring也是依赖了JCL。  接下来另一个优秀的日志框架slf4j的加入导致了更加混乱的场面。比较巧的是slf4j的作者(Ceki Gülcü)就是log4j的作者，他觉得JCL不够优秀，所以他要自己搞一套更优雅的出来，于是slf4j日志体系诞生了，并为slf4j实现了一个亲子——logback，确实更加优雅。  但由于之前很多代码库已经使用JCL，虽然出现slf4j和JCL之间的桥接转换，但是集成的时候问题依然多多，对很多新手来说确实会很懊恼，因为比单独的log4j时代“复杂”多了。到此本来应该完了，但是Ceki Gülcü觉得得回头拯救下自己的“大阿哥”——log4j，于是log4j2诞生了，同样log4j2也参与到了slf4j日志体系中。日志体系的江湖又将面临血雨腥风的混乱。 |

## **传统开发方式**

采用的示例业务-模拟用户登录，进行业务代码编写（只包含业务层、数据持久层）

创建包com.test.traditional

* 第一步：Dao层

UserDao.java

**package** com.test.traditional;

**public** **interface** UserDao {

**void** selectOne()

}

UserDaoImpl.java

**package** com.test.traditional;

**public** **class** UserDaoImpl **implements** UserDao {

@Override

**public** **void** selectOne() {

System.***out***.println("dao层被调用了");

}

}

* 第二步：Service层

UserService.java

**package** com.test.traditional;

**public** **interface** UserService {

**void** login();

}

UserServiceImpl.java

**package** com.test.traditional;

**public** **class** UserServiceImpl **implements** UserService {

@Override

**public** **void** login() {

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

//传统方式-自己实例化对象，再调用方法

UserDao userDao = **new** UserDaoImpl();

userDao.selectOne();

}

}

第三步：测试

使用Junit4进行测试，为UserServiceImpl创建Junit Test Case（注意：测试类名如果是Test，@Test注解就需要加包名。这里避免使用Test）

**package** com.test.traditional;

**import** org.junit.Test;

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

@Test

**public** **void** testLogin() {

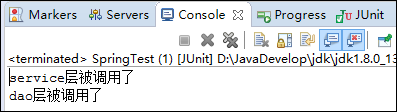
UserService userServiceImpl = **new** UserServiceImpl();

userServiceImpl.login();

}

}

运行测试方法，控制台输出：



【思考分析】

传统开发方式存在的问题：

代码耦合度高，上层代码过度依赖下层代码的实现：如在service层：UserDao userDao = new UserDaoImpl(); 如果要更换实现类，或实现类更换名字，则必须要修改业务代码！

## **IOC原理**

使用IOC（Inverse of Control，控制反转）思想解决代码耦合问题

简单的说，就是引入spring工厂（第三者），将原来在程序中手动创建管理的UserDaoImpl对象，交给工厂来创建管理。

**IoC原理：工厂设计模式 + 反射 + 配置文件（xml）**

创建包com.test.ioc.principle

将UserDao、UserDaoImpl、UserService、UserServiceImpl、SpringTest类拷贝到新包

* 步骤一：提供userDAO实例对象的工厂

UserDAOFactory.java

**package** com.test.ioc.principle;

**public** **class** UserDaoFactory {

/\*//方式1-耦合性仍然强: UserDaoFactory和UserDaoImpl耦合

public UserDao getUserDao() {

UserDao userDao = new UserDaoImpl();

return userDao;

}\*/

//方式2-反射创建实例

**public** Object getBean() {

Object bean = **null**;

**try** {

bean = Class.*forName*("com.test.ioc.principle.UserDaoImpl").newInstance();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** bean;

}

}

类路径字符串是固定的，后面使用xml配置文件动态传入class路径

* 步骤二: 修改UserServiceImpl中获得对象的方式

UserServiceImpl.java

**package** com.test.ioc.principle;

**public** **class** UserServiceImpl **implements** UserService {

@Override

**public** **void** login() {

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

//1.传统方式：自己实例化对象，再调用方法

//UserDao userDao = new UserDao();

//2.IoC方式：工厂+反射+配置文件

UserDaoFactory userDaoFactory = **new** UserDaoFactory();

//UserDao userDao = userDaoFactory.getUserDao();

UserDao userDao = (UserDao) userDaoFactory.getBean();

userDao.selectOne();

}

}

* 测试：在Service层，通过自定义的工厂，也能获取到Dao层实例对象

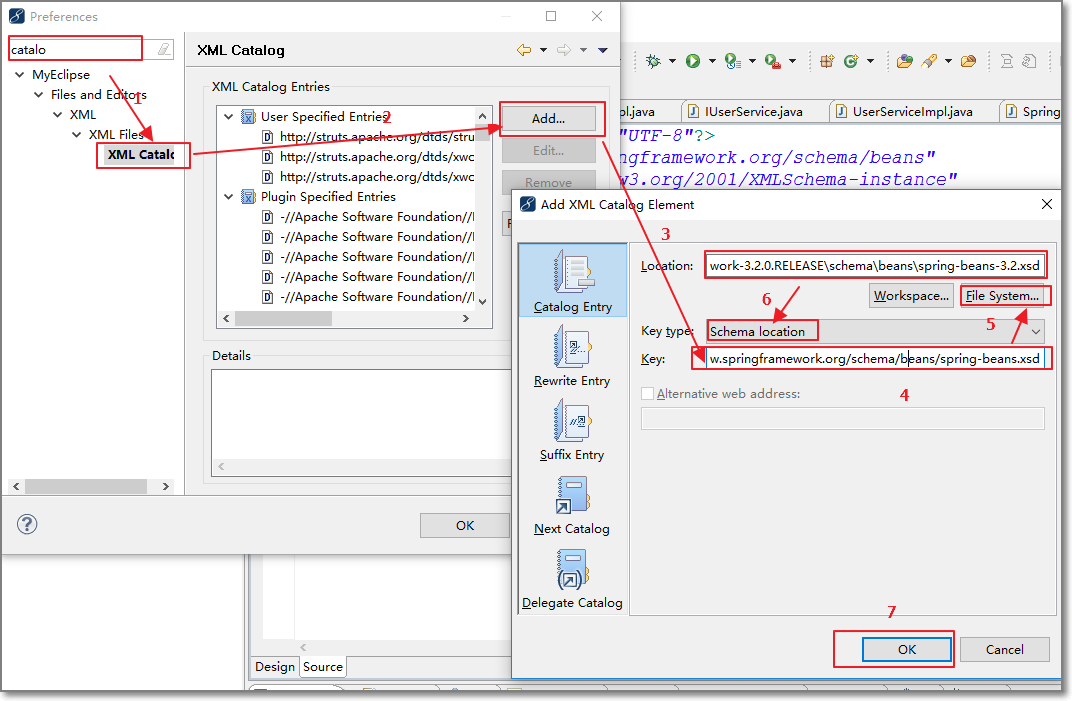
**IoC底层实现：工厂（设计模式）+反射（机制） + 配置文件（xml）**

### **Spring配置文件**

上面使用工厂模式+反射创建实例，下面介绍spring的核心配置文件applicationContext.xml

在该配置文件中进行spring的相关配置，其中就包括了bean配置（类路径字符串的动态传入）

* 步骤一：在src下创建applicationContext.xml（applicationContext.xml可放置到任何目录，但习惯上放在src目录或者 WEB-INF目录）
* 步骤二：参考spring-framework-reference/pdf规范文档配置xml的头信息：bean schema
* 步骤三（选做）：如果没有提示，配置本地提示：(联网时eclipse会自动下载关联相关约束文件)



* 步骤四：配置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"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<!--

一个bean标签对应spring工厂创建的一个对象(反射机制)

id/name:对象引用, 一般为类名或接口名称的首字母小写

class:要创建的对象类型的类字符串,类名全路径

-->

<bean id=*"userDao"* class=*"com.test.ioc.principle.UserDaoImpl"* />

</beans>

### **通过Spring工厂获取Bean**

前面是自定义工厂来实现，spring提供了工厂

在程序中创建spring工厂对象，通过工厂对象加载spring的xml配置文件，由spring工厂创建配置文件中的bean对象，在代码中直接获取即可

* 修改UserServiceImpl.java：

**package** com.test.ioc.principle;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** UserServiceImpl **implements** UserService {

@Override

**public** **void** login() {

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

//1.传统方式：自己实例化对象，再调用方法

//UserDao userDao = new UserDao();

//2.自定义工厂IOC方式：自定义工厂+反射

//UserDaoFactory userDaoFactory = new UserDaoFactory();

//UserDao userDao = userDaoFactory.getUserDao();

//UserDao userDao = (UserDao) userDaoFactory.getBean();

//3.spring工厂IOC方式：spring工厂+反射+xml配置

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

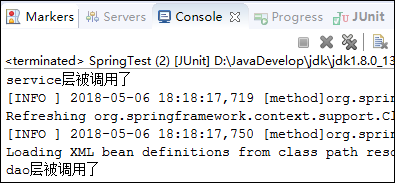
UserDao userDao = (UserDao) ac.getBean("userDao");

userDao.selectOne();

}

}

* 运行测试:



【思考分析】

该方式虽然解决了类与类之间的耦合，但却需要在每次获取对象的时候创建spring工厂ApplicationContext，有没有更方便获取对象的方法呢，或解决userService依赖userDao的方法?

## **DI依赖注入**

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（简单的说，是将一个bean对象动态的注入到另一个bean中）

回顾之前的代码:

要在Service层使用Dao对象实例，就要创建一次spring工厂，同getBean来获取Dao层的实例；Controller层需要Service层对象，还要创建spring工厂再getBean

DI方式：由Spring容器来创建Service、Dao对象，并且在applicationContext.xm中将Dao和Servcie的依赖关系配置好（Service依赖Dao），那么Service对象就包含了Dao对象的引用。

只需要将userDAO定义成userService的一个成员变量，使用setter方法完成属性注入：

* 步骤一：将service对象也交给spring容器管理

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"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<!-- bean: spring工厂创建的一个对象(反射机制)

id/name:对象的名字,可以用来引用或者获取对象, 一般为类名或接口名称的首字母小写

class:要创建的对象类型的类字符串,类名全路径 -->

<bean id=*"userDao"* class=*"com.test.ioc.principle.UserDaoImpl"*/>

<bean id=*"userService"* class=*"com.test.ioc.principle.UserServiceImpl"*>

<!-- property 根据类中的setter方法进行属性注入 -->

<!-- name：setter方法的后缀小写,比如setXxx 对应的name为xxx(写好setter()后，可自动补全) -->

<!-- ref：要引入bean的id/name -->

<property name=*"userDao"* ref=*"userDao"* />

</bean>

</beans>

* 步骤二：在被注入类UserService中定义成员变量（注入对象UserDAO的引用），并提供setter方法：

修改UserServiceImpl.java

**package** com.test.ioc.principle;

**public** **class** UserServiceImpl **implements** UserService {

//4.IoC+DI

**private** UserDao userDao;

**public** **void** setUserDao(UserDao userDao) {

**this**.userDao = userDao;

}

@Override

**public** **void** login() {

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

//1.传统方式：自己实例化对象，再调用方法

//UserDao userDao = new UserDao();

//2.自定义工厂IOC方式：自定义工厂+反射

/\*UserDaoFactory userDaoFactory = new UserDaoFactory();

UserDao userDao = userDaoFactory.getUserDao();

UserDao userDao = (UserDao) userDaoFactory.getBean();\*/

//3.spring工厂IOC方式：spring工厂+反射+xml配置

//ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");

//UserDao userDao = (UserDao) ac.getBean("userDao");

**this**.userDao.selectOne();

}

}

* 步骤三：测试运行

注意测试类中获取对象必须从spring容器获取，不能自己new(从spring容器获取的才有依赖注入，自己创建的对象没有注入依赖关系)

修改SpringTest.java

**package** com.test.ioc.principle;

**import** org.junit.Test;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

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

@Test

**public** **void** testLogin() {

//UserService userServiceImpl = new UserServiceImpl();

//userServiceImpl.login();

//1.根据id/name查找bean(这里不能new，要用spring工厂配置的userService，才有userDao的依赖注入)

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

UserService userService = (UserService) ac.getBean("userService");

userService.login();

}

}

运行结果:

[INFO ] 2018-05-07 10:51:59,939 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 10:51:59 CST 2018]; root of context hierarchy

[INFO ] 2018-05-07 10:51:59,960 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext.xml]

service层被调用了

dao层被调用了

【小结】

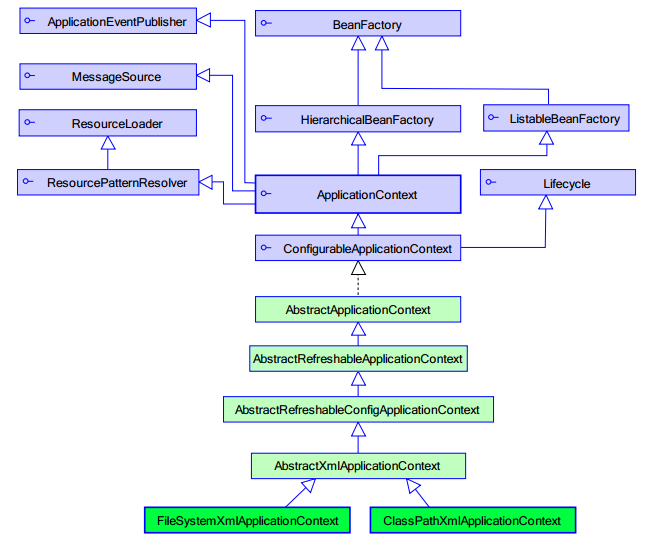
IOC：控制反转，将对象创建管理的权利交给spring容器，获取对象通过spring工厂

DI：在spring容器中创建管理对象，通过 property标签配置对象间依赖关系，将一个bean实例注入到需要依赖的bean实例中

## **Spring工厂**

ApplicationContext直译为应用上下文，又称spring容器，用来加载Spring配置文件。

ApplicationContext只是BeanFactory（Bean工厂）（spring提供） 一个子接口，ClassPathXmlApplicationContext是一个实现类。



为什么不直接使用顶层接口BeanFactory来操作呢？

* BeanFactory 采取延迟加载，即第一次getBean时才会初始化Bean，用法：

BeanFactory ac = new XmlBeanFactory(new FileSystemResource("D:\\applicationContext.xml"));

* ApplicationContext是对BeanFactory的扩展

对于单例类，在创建ac工厂时就创建并初始化Bean，并且提供了更多功能：国际化处理、事件传递、Bean自动装配、各种不同应用层的Context实现。即ApplicationContext 更加强大

【扩展】

获取Bean的两种方式（getBean的重载方法）：

* 第一种：根据id/name查找（唯一，推荐使用）

**package** com.test.ioc.principle;

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

@Test

**public** **void** testLogin() {

//UserService userServiceImpl = new UserServiceImpl();

//userServiceImpl.login();

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

//1.根据id/name查找bean(这里不能new，要用spring工厂配置的userService，才有userDao的依赖注入)

UserService userService = (UserService) ac.getBean("userService");

userService.login();

}

}

* 第二种：根据类名/接口名查找

根据bean类型或bean接口类型获取，一般用接口类型（会自动去找实现类）

**package** com.test.ioc.principle;

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

@Test

**public** **void** testLogin() {

//UserService userServiceImpl = new UserServiceImpl();

//userServiceImpl.login();

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

//1.根据id/name查找bean(这里不能new，要用spring工厂配置的userService，才有userDao的依赖注入)

//UserService userService = (UserService) ac.getBean("userService");

//2.根据接口类型查找bean(要求不能有class属性相同的bean)

UserService userService = (UserService) ac.getBean(UserService.**class**);

userService.login();

}

}

这种方式查找的bean可能有多个。即多个class属性一样但对象名id/name不同的<bean/>，此时会抛异常：NoSuchBeanDefinitionException:No unique bean of type...

例如在applicationContext.xml中配置如下

<bean id=*"userDao1"* class=*"com.test.ioc.principle.UserDaoImpl"*/>

<bean id=*"userDao2"* class=*"com.test.ioc.principle.UserDaoImpl"*/>

# IOC配置bean---XML方式

spring工厂底层创建Bean类有四种方式，对应了我们可以有4中方式在xml中配置bean。可对应bean标签中的各个属性来学习

## **bean配置的4种方式**

这四种方式开发中可能一种都用不到（是在底层由spring工厂创建），用的话第一种最常用

创建包cn.itcast.spring.b\_xmlnewbean

* 第一种方式：无参构造 （最常用）

spring在创建bean的时候自动调用无参构造来实例化，相当于new Bean1()。

该种方式要求：必须要有空参构造

* 第一步：创建Bean1.java

**package** com.test.ioc.bean.xml;

**public** **class** Bean1 {

//不加任何构造，默认系统给空参构造

}

* 第二步：applicationContext.xml中配置

<!-- 1.默认使用无参构造实例化对象 -->

<bean id=*"bean1"* class=*"com.test.ioc.bean.xml.Bean1"*/>

* 第三步:创建测试类获取bean对象

SpringTest.java:

**package** com.test.ioc.bean.xml;

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

@Test

**public** **void** testBean() {

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

Bean1 bean1 = (Bean1) ac.getBean("bean1");

System.***out***.println(bean1);

}

}

运行结果：

[INFO ] 2018-05-07 11:55:18,874 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 11:55:18 CST 2018]; root of context hierarchy

[INFO ] 2018-05-07 11:55:18,896 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext.xml]

com.test.ioc.bean.xml.Bean1@4fb64261

【错误演示】：

**package** com.test.ioc.bean.xml;

**public** **class** Bean1 {

**private** String name;

//给出有参构造，系统就不会再给空参构造，即只有有参构造，没有空参构造

**public** Bean1(String name) {

**super**();

**this**.name = name;

}

}

运行报错：（没有提供无参构造）

org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'bean1' defined in class path resource [applicationContext.xml]: Instantiation of bean failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instantiate [com.test.ioc.bean.xml.Bean1]: No default constructor found; nested exception is java.lang.NoSuchMethodException: com.test.ioc.bean.xml.Bean1.<init>()

* 第二种方式：静态工厂方式

该方式不用创建工厂实例，区别于实例工厂方式

直接【类名.方法名】调用即可。即不需要在applicationContext.xml中配置工厂的bean。

* 第一步：创建Bean2.java

**package** com.test.ioc.bean.xml;

**public** **class** Bean2 {

}

* 第二步：创建Bean2Factory.java

**package** com.test.ioc.bean.xml;

**public** **class** Bean2Factory {

//静态方法，方便调用

**public** **static** Bean2 getBean2() {

//do other things,like Connection to Driver...

**return** **new** Bean2();

}

}

* 第三步：applicationContext.xml

<!-- class:直接指定到静态工厂类, factory-method: 指定到生产实例的方法, spring容器在实例化工厂类的时候会自动调用该方法并返回实例对象 -->

<bean id=*"bean2"* class=*"com.test.ioc.bean.xml.Bean2Factory"* factory-method=*"getBean2"*/>

* 第四步：测试类

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

@Test

**public** **void** testBean() {

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

Bean2 bean2 = (Bean2) ac.getBean("bean2");

System.***out***.println(bean2);

}

}

运行结果：

[INFO ] 2018-05-07 14:13:53,594 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 14:13:53 CST 2018]; root of context hierarchy

[INFO ] 2018-05-07 14:13:53,618 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext.xml]

com.test.ioc.bean.xml.Bean2@43a0cee9

* 第三种方式：实例工厂方法

该方式需先创建工厂实例，区别于静态工厂方式

即需要额外在applicationContext.xml中配置工厂的bean。

* 第一步：创建Bean3.java

**package** com.test.ioc.bean.xml;

**public** **class** Bean3 {

}

* 第二步：创建实例工厂Bean3Factory

**package** com.test.ioc.bean.xml;

**public** **class** Bean3Factory {

//非静态方法，需要先创建工厂实例，才能调用

**public** Bean3 getBean3() {

//do other things,like Connection to Driver

**return** **new** Bean3();

}

}

* 第三步：applicationContext.xml（需要配两个bean标签）

<bean id=*"bean3Factory"* class=*"com.test.ioc.bean.xml.Bean3Factory"*/>

<bean id=*"bean3"* class=*"com.test.ioc.bean.xml.Bean3"* factory-bean=*"bean3Factory"* factory-method=*"getBean3"*/>

* 第四步：测试类

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

@Test

**public** **void** testBean() {

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

Bean3 bean3 = (Bean3) ac.getBean("bean3");

System.***out***.println(bean3);

}

}

运行结果：

[INFO ] 2018-05-07 14:38:02,009 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 14:38:02 CST 2018]; root of context hierarchy

[INFO ] 2018-05-07 14:38:02,031 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext.xml]

com.test.ioc.bean.xml.Bean3@2eda0940

* 第四种方式：实现FactoryBean接口方式

该方式在spring底层源码用的多

spring在实例化bean时，会判断是否实现了FactoryBean接口。如果实现了就调用getObject方法返回目标类型实例

* 第一步：创建Bean4.java

**package** com.test.ioc.bean.xml;

**public** **class** Bean4 {

}

* 第二步：自定义FactoryBean接口的实现，重写getObject()方法

**package** com.test.ioc.bean.xml;

**import** org.springframework.beans.factory.FactoryBean;

**public** **class** Bean4FactoryBean **implements** FactoryBean<Bean4> {

@Override

**public** Bean4 getObject() **throws** Exception {

//do other things,like Connection to Driver

**return** **new** Bean4();

}

@Override

**public** Class<?> getObjectType() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**public** **boolean** isSingleton() {

// **TODO** Auto-generated method stub

**return** **false**;

}

}

* 第三步：applicationContext.xml

<!-- class属性指向FactoryBean接口的实现类 -->

<bean id=*"bean4"* class=*"com.test.ioc.bean.xml.Bean4FactoryBean"*/>

运行结果：

[INFO ] 2018-05-07 14:53:18,115 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 14:53:18 CST 2018]; root of context hierarchy

[INFO ] 2018-05-07 14:53:18,139 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext.xml]

com.test.ioc.bean.xml.Bean4@6eceb130

【总结四种方式】

第一种：最常用

第三种：一些框架初始化的时候用的多

第四种：spring底层用的多

【面试题】 BeanFactory和FactoryBean的区别

BeanFactory：是spring工厂的超级接口，（ApplicationContext是其子接口，但功能比它强大），用来管理和获取Bean对象

FactoryBean：是一个Bean生成工具，是用来获取一种类型对象的Bean，它是构造Bean实例的一种方式。

## **bean的作用域**

由spring创建的bean对象在什么情况下有效



项目开发中常用：singleton 单例、 prototype多例

Singleton：在一个spring容器中，只有一个该类实例（默认设置，声明可省略）

Prototype：在一个spring容器中，允许存在多个实例，每次getBean 时，才创建初始化实例，然后返回（在配置bean时，要声明是多例）

创建包com.test.ioc.bean.lifescope

第一步：创建类SingletonBean.java和PrototypeBean.java

SingletonBean.java

//单例bean

**package** com.test.ioc.bean.lifescope;

**public** **class** SingletonBean {

**public** SingletonBean() {

System.***out***.println("SingletonBean..空参构造..初始化了");

}

}

PrototypeBean.java

//多例bean

**package** com.test.ioc.bean.lifescope;

**public** **class** PrototypeBean {

**public** PrototypeBean() {

System.***out***.println("PrototypeBean..空参构造..初始化了");

}

}

第二步：配置applicationContext.xml

<!-- scope:配置作用范围的，默认值就是singleton单例 -->

<bean id=*"singletonBean"* class=*"com.test.ioc.bean.lifescope.SingletonBean"*/>

<bean id=*"prototypeBean"* class=*"com.test.ioc.bean.lifescope.PrototypeBean"* scope=*"prototype"*/>

第三步：测试代码，创建SpringTest.java

package com.test.ioc.bean.lifescope;

public class SpringTest {

@Test

public void testScope() {

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

System.out.println("------准备get单例1------");

SingletonBean sb1=(SingletonBean)ac.getBean("singletonBean");

System.out.println("------准备get单例2------");

SingletonBean sb2=(SingletonBean)ac.getBean("singletonBean");

System.out.println(sb1);

System.out.println(sb2);

System.out.println("------准备get多例1------");

PrototypeBean pb1=(PrototypeBean)ac.getBean("prototypeBean");

System.out.println("------准备get多例1------");

PrototypeBean pb2=(PrototypeBean)ac.getBean("prototypeBean");

System.out.println(pb1);

System.out.println(pb2);

}

}

测试结果：

[INFO ] 2018-05-07 15:59:25,761 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 15:59:25 CST 2018]; root of context hierarchy

[INFO ] 2018-05-07 15:59:25,788 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext.xml]

SingletonBean..空参构造..初始化了

------准备get单例1------

------准备get单例2------

com.test.ioc.bean.lifescope.SingletonBean@67b467e9

com.test.ioc.bean.lifescope.SingletonBean@67b467e9

------准备get多例1------

PrototypeBean..空参构造..初始化了

------准备get多例2------

PrototypeBean..空参构造..初始化了

com.test.ioc.bean.lifescope.PrototypeBean@47db50c5

com.test.ioc.bean.lifescope.PrototypeBean@5c072e3fs

【结论】

单例：

是默认配置，可省略声明。每次从spring容器中获取的对象，是同一个对象

初始化：在spring容器ac初始化的时候，就初始化了

多例：

多例需要声明。每次从spring容器中获取的对象，不是同一个对象

初始化：在getBean的时候才创建和初始化，相当于每次getbean就是new Bean()

## **bean的生命周期**

通过配置，可以控制spring工厂中的bean生命周期

### **bean初始化和销毁方法**

在applicationContext.xml中：通过init-method属性，指定初始化后的调用方法；通过destroy-method属性，指定销毁对象前的方法

创建包com.test.ioc.bean.lifecycle

* 第一步：创建LifeCycleBean，自定义初始化方法和销毁方法

**package** com.test.ioc.bean.lifecycle;

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

//空参构造

**public** LifeCycleBean() {

System.***out***.println("LifeCycleBean空参构造调用了");

}

//初始化后自动调用方法，方法名随意，要配置到xml中

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

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

}

//bean销毁时调用的方法，方法名随意，要配置到xml中

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

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

}

}

* 第二步：Spring的核心容器，applicationContext.xml的配置

<!-- 生命周期调用的两个方法 init-method：初始化时（后）调用的，bean中的public方法即可；destroy-method：销毁时（前）被调用的 -->

<bean id=*"lifeCycleBean"* class=*"com.test.ioc.bean.lifecycle.LifeCycleBean"* init-method=*"init"* destroy-method=*"destroy"*/>

* 第三步：SpringTest.java测试

package com.test.ioc.bean.lifecycle;

public class SpringTest {

@Test

public void testLifeCycle() {

//对于单例此时也会创建和初始化

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

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

System.out.println(lifeCycleBean);

//此时没有调用销毁方法

//原因是：使用debug模式jvm直接就关了，spring容器/工厂还没有来得及销毁对象。

//解决：手动关闭spring容器，此时自动销毁单例的对象

((ClassPathXmlApplicationContext)applicationContext).close(); //要有外括号；ApplicationContext接口中没有close方法，子类中有，所以要强转

}

}

测试时查看控制台打印，发现销毁方法没有执行

【原因】：销毁方法的执行必须满足两个条件：

1. 单例bean才可以手动销毁
2. 必须手动关闭容器（调用close()）时，才会执行手动销毁的方法

多例只有ac.getBean()时才会创建和初始化实例，并且没法销毁

LifeCycleBean空参构造调用了

LifeCycleBean-init初始化时调用

com.test.ioc.bean.lifecycle.LifeCycleBean@47db50c5

[INFO ] 2018-05-07 16:36:02,261 [method]org.springframework.context.support.AbstractApplicationContext.doClose(AbstractApplicationContext.java:982)

Closing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 16:36:02 CST 2018]; root of context hierarchy

LifeCycleBean-destroy销毁时调用

【扩展】

数据（属性）的初始化方式：

1. 直接赋值

耦合性最强。一般用于给默认值，或给静态常量赋值

**private** String name="Tom";

**private** **final** **static** Double **PI** = 3.1415;

2. 在构造器中初始化

即有参构造的动态初始化，一般创建对象时使用。有些在空参/有参构造中，初始化其他属性，这种耦合性也高

**public** LifeCycleBean(String name) {

**this**.name = name;

**this**.age = 18;

}

3. 使用单独的初始化的方法（setter方法）（推荐）

完全解耦。初始化方法专门用来初始化，构造器只用来构造class对象。即：setter+空参构造

### **BeanPostProcessor接口**

BeanPostProcessor接口，也称Bean后处理器。

作用是：在Bean初始化前后，对Bean对象进行增强。它既可以增强一个指定的Bean，也可以增强所有的Bean，底层很多功能（如AOP等）的实现都是基于它的，Spring可在容器中直接识别调用。

【示例】对所有的bean初始化时进行增强

* 第一步：自定义BeanPostProcessor接口的实现类，实现2个方法

**package** com.test.ioc.bean.lifecycle;

//自定义bean后处理器：用来对bean进行功能增强，可以实现，对所有或某个bean的初始化进行增强

**public** **class** MyBeanPostProcessor **implements** BeanPostProcessor {

//初始化前调用

//参数1：要增强的bean对象（xml配置文件中的，即spring管理的）；参数2：bean的名字，id/name

**public** Object postProcessBeforeInitialization(Object bean, String beanName)

**throws** BeansException {

//System.out.println(beanName+"在初始化前开始增强了");//增强所有bean

//只增强一个bean

**if**(beanName.equals("lifeCycleBean")){

System.***out***.println(beanName + "在初始化前开始增强了");

}

**return** bean;//放行

}

//初始化后调用

**public** Object postProcessAfterInitialization(Object bean, String beanName)

**throws** BeansException {

//System.out.println(beanName+"在初始化后开始增强了");//增强所有bean

**if**(beanName.equals("lifeCycleBean")){

System.***out***.println(beanName + "在初始化后开始增强了");

}

**return** bean;

}

}

* 第二步：applicationContext.xml中配置注册自定义的bean后处理器

<!-- bean后处理器：spring在初始化MyBeanPostProcessor类时，判断该类是否实现了BeanPostProcessor。如果实现了，就采用动态代理的方式，对所有的bean对象增强 -->

<bean id=*"myBeanPostProcessor"* class=*"com.test.ioc.bean.lifecycle.MyBeanPostProcessor"*/>

测试：在生命周期bean上进行测试

SingletonBean..空参构造..初始化了

LifeCycleBean空参构造调用了

lifeCycleBean在初始化前开始增强了

LifeCycleBean-init初始化时调用

lifeCycleBean在初始化后开始增强了

com.test.ioc.bean.lifecycle.LifeCycleBean@954b04f

[INFO ] 2018-05-07 17:44:31,709 [method]org.springframework.context.support.AbstractApplicationContext.doClose(AbstractApplicationContext.java:982)

Closing org.springframework.context.support.ClassPathXmlApplicationContext@50675690: startup date [Mon May 07 17:44:31 CST 2018]; root of context hierarchy

LifeCycleBean-destroy销毁时调用

【总结】

BeanPostProcessor接口，实现了以无代码侵入的方式，对bean进行增强（初始化前与后）

## **bean属性的依赖注入**

什么是Bean属性的依赖注入：就是给一个对象的属性赋值。该属性引用一般指向的，也是spring工厂管理的bean实例

在上面IOC快速入门中，使用了DI，向userService的userDao属性，进行了依赖注入，注入的正是spring管理的userDao实例

给属性赋值：一般使用有参构造，或无参构造+setter方法，所以DI依赖注入的方式也是这两种思路

依赖注入的3种方式

* 第一种：有参构造注入 constructor-arg标签
* 第二种：无参构造 + setter方法注入
* 第三种：~~接口注入~~

Spring框架规范中，通过xml配置文件的方式，只支持构造器参数注入和setter方法属性注入，不支持接口注入。

创建包com.test.ioc.bean.xml.di

### **方式1-有参构造注入**

【示例】

* 第一步：创建Car.java，定义有参构造

**package** com.test.ioc.bean.xml.di;

**public** **class** Car {

**private** Integer id;

**private** String name;

**private** Double price;

//有参构造

**public** Car(Integer id, String name, Double price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

@Override

**public** String toString() {

**return** "Car [id=" + id + ", name=" + name + ", price=" + price + "]";

}

}

* 第二步：applicationContext.xml中配置

<!-- 第1种：有参构造注入属性的值 -->

<bean id=*"car"* class=*"com.test.ioc.bean.xml.di.Car"*>

<!-- constructor-arg：告诉spring容器，要调用有参构造，不再调用默认的空参构造

new Car(1,"宝马",99999d)

参数第一组：用于定位属性

\* index：根据索引定位属性，以0开始

\* name：根据属性参数名称定位属性

\* type：根据属性数据类型定位属性

参数第二组：用于赋值

\* value:简单的值，字符串

\* ref:复杂的（由spring容器创建的bean对象）

-->

<!-- <constructor-arg index="0" value="1"/> -->

<constructor-arg index=*"0"* name=*"id"* value=*"1"*/>

<!-- <constructor-arg name="name" value="宝马1代"/> -->

<constructor-arg name=*"name"* >

<value>宝马2代</value>

</constructor-arg>

<constructor-arg type=*"java.lang.Double"* value=*"99999d"*/>

</bean>

* 第三步：使用SpringTest.java测试

package com.test.ioc.bean.xml.di;

public class SpringTest {

@Test

public void testDi() {

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

Car car = (Car) ac.getBean("car");

System.out.println(car);

}

}

测试结果：

控制台输出：Car [id=1, name=宝马2代, price=99999.0]

【补充】

1．属性定位：定位属性的标签index、name、type，可以混用

<constructor-arg index=*"0"* name=*"id"* value=*"1"*/>

2．属性赋值：自标签的属性赋值，也可以使用子标签value/ref，效果和value/ref属性一样

<constructor-arg name="name" value="宝马1代"/>

等同于

<constructor-arg name=*"name"* >

<value>宝马1代</value>

</constructor-arg>

### **方式2-无参构造+ setter方法**

使用空参构造(new Bean()) + setter方法，配合配置文件中的property属性，完成属性依赖注入。是最常用的方式。

【示例】

* 第一步：创建Person.java

**package** com.test.ioc.bean.xml.di;

//测试setter方法依赖注入

**public** **class** Person {

**private** Integer id;

**private** String name;

**private** Car car;

//默认给出空参构造

//必须提供setter方法

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** setCar(Car car) {

**this**.car = car;

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", name=" + name + ", car=" + car + "]";

}

}

* 第二步：配置applicationContext.xml

<!-- 第2种：无参构造+setter方法注入属性值 -->

<bean id=*"person"* class=*"com.test.ioc.bean.xml.di.Person"*>

<!--

property：专门进行setter属性注入用的标签

\* name：setter方法的属性名,例如setXxx-那么name的属性值为xxx。不是成员变量名（跟setter方法走）

\* value：简单的值

\* ref：bean的名字，对象的引用

-->

<property name=*"id"* value=*"1"*/>

<property name=*"name"* value=*"Tony"*/>

<property name=*"car"* ref=*"car"*/>

<!-- <property name="name">

<value>Tony</value>

</property> -->

<!-- <property name="car">

<ref bean="car"/>

</property> -->

</bean>

* 第三步：使用SpringTest.java测试：

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

@Test

**public** **void** testDi() {

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

Person person = (Person) ac.getBean("person");

System.***out***.println(person);

}

}

测试输出：

Person [id=1, name=Tony, car=Car [id=1, name=宝马2代, price=99999.0]]

【扩展】

ref/value标签的用法：

ref：用于对象引用（引用数据类型，常指向配置的其他bean）、value：用于简单值（基本数据类型）

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

等同

<property name=*"car"*>

<ref bean=*"car"*/>

</property>

## **p名称空间**

什么是名称空间？

作用：Schema区分同名元素。（有点类似于java的包）

<beans xmlns=*"http://www.springframework.org/schema/beans"*

回顾：Xmlns没有前缀是默认的名称空间

为简化XML文件的配置，Spring2.5版本开始引入了一个新的p名称空间。简单的说，它的作用是为了简化setter方法属性依赖注入配置的，它不是真正的名称空间。

使用方法：

p:<属性名>="xxx" 引入常量值

p:<属性名>\_ref="xxx" 引用其它Bean对象

操作步骤：

* 第一步：先引入p名称空间，增加一行

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

* 第二步：将<property>标签简化为bean标签的属性

<!-- 使用p名称空间简化setter方法属性注入 -->

<!--

p:name：简单数据类型的属性注入

P:car-ref：复杂数据类型（bean）的属性注入

-->

<bean id=*"person1"* class=*"com.test.ioc.bean.xml.di.Person"* p:id=*"2"* p:name=*"Lucy"* p:car-ref=*"car"*/>

* 第三步：测试

@Test

**public** **void** testDi() {

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

Person person = (Person) ac.getBean("person1");

System.***out***.println(person);

}

测试结果：

Person [id=2, name=Lucy, car=Car [id=1, name=宝马2代, price=99999.0]]

【结论】

配置时不需要<property > 子元素，简化了配置 。其实简化不多，所以一般不用p名称空间

## **spEL表达式**

spEL（Spring Expression Language）是一种表达式语言，它是spring3.x版本的新特性。

作用：支持在运行时操作和查询对象，其语法类似统一的EL语言，但是SpEL提供了额外的功能，功能更强大。

什么是EL、OGNL、spEL？

EL：操作servlet相关的一些对象和相关的值

OGNL：主要操作struts2值栈

spEL：操作bean相关的

语法： #{…} , 引用另一个Bean、属性、方法、运算（取其他类的属性值时，依赖getXxx方法）

SpEL表达式的使用功能比较多，其中与bean操作相关的通常有：

#{beanid} 引用Bean(具体对象)

#{beanId.属性} 引用Bean的属性

#{beanId.方法(参数)} 调用Bean的方法

【案例一】

配置applicationContext.xml

<bean id=*"person2"* class=*"com.test.ioc.bean.xml.di.Person"*>

<property name=*"id"* value=*"#{person1.id}"*/>

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

<property name=*"car"* ref=*"car"*/>

</bean>

<bean id=*"person3"* class=*"com.test.ioc.bean.xml.di.Person"* p:id=*"#{person1.id + 10}"* p:name=*"car.name"* p:car-ref=*"car"*/>

<bean id=*"person4"* class=*"com.test.ioc.bean.xml.di.Person"* p:id=*"#{1 + 100}"* p:name=*"#{'alivn'.toUpperCase()}"* p:car-ref=*"car"*/>

测试抛出异常：

org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'person2' defined in class path resource [applicationContext.xml]: Initialization of bean failed; nested exception is org.springframework.beans.factory.BeanExpressionException: Expression parsing failed; nested exception is org.springframework.expression.spel.SpelEvaluationException: EL1008E:(pos 8): Property or field 'id' cannot be found on object of type 'com.test.ioc.bean.xml.di.Person' - maybe not public?

【原因】

spEL需要访问对象的属性时，要求对象的属性能访问到，所以：

1. 要么属性为public（不能是default或private等）
2. 要么私有属性时，必须提供getter方法（常用）

在对应类中添加getter方法后/公开属性即可

**public** **class** Person {

//1.公开属性，一般不用这种方式

**public** Integer id;

...

**public** **class** Car {

**private** String name;

//2.使用getter，配合spEL表达式

**public** String getName() {

**return** name;

}

...

1. 测试结果：

Person [id=2, name=宝马2代, car=Car [id=1, name=宝马2代, price=99999.0]]

Person [id=12, name=car.name, car=Car [id=1, name=宝马2代, price=99999.0]]

Person [id=101, name=ALIVN, car=Car [id=1, name=宝马2代, price=99999.0]]

关于springEL表达式的更多用法，参考官方资料

# IOC配置bean---注解方式

和xml的4种方式一样，用注解声明bean，也是将类实例交给spring工厂管理

使用**@Component**注解类，相当于<bean id=”” class=””/>

**@Component**有3个衍生注解，它们功能和**@Component**一样，但有分层的含义，开发中常用折3个子注解

@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）

@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

Spring3.0为我们引入了组件自动扫描机制，它可以在指定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。并且有丰富的过滤条件，不用在applicationContext.xml中配置大量的<bean>，使配置文件清晰简洁。

## **bean配置--注解方式**

创建包：com.test.ioc.bean.anno

* 第一步：创建类：CustomerService.java和CustomerDao.java，并同时完成bean的声明：

xml做法：<bean id=”customerService” class=”…” />

注解做法：@Component，spring2.5引入该注解，放置到类上，相当于在spring容器中定义<bean id=”” class=””/>

CustomerService.java

**package** com.test.ioc.bean.anno;

/\*\*

\* **@Component**

\* 相当于spring容器中定义：<bean id="customerService" class="com.test.ioc.bean.anno">

\* 其中不指定bean的id/name时，id属性默认bean的名字是类名的小写

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

\* **@Component**(value="xxx")//自定义bean名称

\* 相当于：<bean id="xxx" class="com.test.ioc.bean.anno">

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

\*/

//@Component(value = "customerService")

@Service

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

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

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

}

}

CustomerDao.java

**package** com.test.ioc.bean.anno;

//@Component

@Repository("customerDao")

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

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

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

}

}

* 第二步： applicationContext.xml中开启注解功能和包扫描功能

这里要使用context标签，需要先引入context名称空间（如果eclipse没有提示，可配置xsd约束）

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

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,有component-scan时可省略 -->

<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,包含子包下的所有类

注解扫描配置的时候，会自动开启注解功能

-->

<context:component-scan base-package=*"com.test.ioc.bean.anno"*/>

</beans>

* 第三步：测试

package com.test.ioc.bean.anno;

public class SpringTest {

@Test

public void testAnno(){

//spring容器

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

//获取bean

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

customerService.save();

CustomerDao customerDao=(CustomerDao) ac.getBean("customerDao");

customerDao.save();

}

}

测试可能会报错：缺少aop的jar包，导入spring-aop-4.3.2.RELEASE.jar即可

测试结果：

CustomerService业务层被调用了

CustomerDao层被调用了

【扩展优化】

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。所以此时，注解开启功能可以省略，即去掉：<context:annotation-config/>

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

【问题】

这里只是将如果将CustomerService 和CustomerDao交给了spring工厂，但如何将Dao依赖注入到Service呢？前面xml中的DI有2种方式，注解该如何注入呢？

回顾：如果使用xml的配置，那么可以使用空参+setter方法进行注入（有参构造不常用）

<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**

在Spring3.0后，提供了 @Value注解，可以完成简单数据的注入

不过该方式还不如直接赋值（private String name = “Rose”）

@Service

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

@Value(value = "Rose")

**private** String name;

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

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

System.***out***.println(name);

}

}

### **复杂类型数据依赖注入**

下面将Dao类的对象注入到Service类进行使用。

Dao和Service的bean此时已在spring容器中，要想依赖注入，无非按id/name或class属性这两种标记来注入。下面的4种方式都是按这两种思路

注解位置：可以将注解加在setXxx方法，也可以注解在成员变量上（任选其一，这里都以注解到成员变量为例）

#### **方式1：@Value结合SpEL**

按名称id注入

在成员变量上注入，底层还是会自动生成setCustomerDao()

@Service

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

//其中customerDao表示<bean>节点的id

@Value("#{customerDao}")

**private** CustomerDao customerDao;

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

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

customerDao.save();

}

}

或注解到setter方法上

**private** CustomerDao customerDao;

@Value("#{customerDao}")

**public** **void** setCustomerDao(CustomerDao customerDao) {

**this**.customerDao = customerDao;

}

#### **方式2：@Autowired[结合@Qualifier]**

* @Autowired

单独使用@Autowired，该方式最常用。

表示按照类型class注入，会到spring容器中查找class为CustomerDao类型的bean，对应<bean class=””>。如果找到，就匹配（类似xml获取属性按接口查找）

@Autowired

**private** CustomerDao customerDao;

* @Autowired + @Qualifier

使用@Autowired + @Qualifier表示按照名称注入，会到spring容器中查找id=customerDao的bean，对应<bean id=””>。如果找到，就匹配（类似xml获取属性按id/name查找）

//Autowired默认按class注入；想按id注入，必须配合@Qualifier

@Autowired

@Qualifier("customerDao")

**private** CustomerDao customerDao;

#### **方式3：@Resource**

JSR-250标准（基于jdk）提供的@Resource。

默认先按照名称id进行匹配，再按照类型class进行匹配；

如果加上name属性，则只会按名称id匹配

* 单独使用@Resource

表示先按照名称id注入，会到spring容器中查找customerDao的名称，对应<bean id=””>。如果找到，可以匹配；如果没有找到，则会再按照类型class注入，会到spring容器中查找CustomerDao的类型，对应<bean class=””>。如果找到，可以匹配，如果没有，抛出异常。

//默认先按照id注入，找不到，再按class注入，找不到抛异常

@Resource()

**private** CustomerDao customerDao;

* @Resource注解添加name属性

在@Resource注解上添加name属性限定，则只会按照名称id注入（name不能省略简写），会到spring容器中查找id=customerDao的bean，对应<bean id=””>。如果找到，可以匹配。没有找到，抛出异常。

//只按id注入，找不到抛异常

@Resource(name = "customerDao")

**private** CustomerDao customerDao;

#### **方式4：@Inject**

JSR-330标准（jdk）提供的@Inject。需要先导入javax.inject-1.jar

* @Inject

默认按类型class注入

//默认按类型注入

@Inject

**private** CustomerDao customerDao;

* @inject + @Named

按照名称id注入

//默认按类型注入;若按名字注入，必须配合@Named使用

@Inject

@Named("customerDao")

**private** CustomerDao customerDao;

【说明】【拓展】

以上4种方式（7种具体形式），也可以注解到setter方法上（直接注解到成员变量，底层也是会创建setter方法，因为是private的）

实际上，方法名不一定是setCustomerDao，可以任意，是按形参注入的

**private** CustomerDao customerDao;

//@Value("#{customerDao}")

//@Autowired

/\*@Autowired

@Qualifier("customerDao")\*/

//@Resource()

//@Resource(name = "customerDao")

//@Inject

@Inject

@Named("customerDao")

**public** **void** setCustomerDao(CustomerDao customerDao) {

**this**.customerDao = customerDao;

}

任意的方法名：

**private** CustomerDao customerDao;

//@Autowired

/\*@Autowired

@Qualifier("customerDao")\*/

//@Resource()

//@Resource(name = "customerDao")

//@Inject

@Inject

@Named("customerDao")

**public** **void** abc(CustomerDao customerDao) {

**this**.customerDao = customerDao;

}

## **bean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，有时需做一些处理工作，因此spring可以在创建和销毁bean时，调用bean的两个生命周期方法。

回顾xml配置的写法：<bean id=“foo” class=“...Foo” init-method=“setup” destory-method=“teardown”/>

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、初始化方法、销毁方法

**package** com.test.ioc.bean.anno;

//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");

}

//初始化后调用方法：方法名随意，但也不能太随便，一会要配置

@PostConstruct

**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：applicationContext.xml中开启注解扫描

<context:component-scan base-package=*"com.test.ioc.bean.anno"*/>

第三步：使用SpringTest.java完成测试

@Test

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

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

//单例；此时，构造和初始化后调用的方法，都已经被调用（创建applicationContext的同时）

AnnoLifeCycleBean lifeCycleBean = (AnnoLifeCycleBean)applicationContext.getBean("annoLifeCycleBean");

//关闭spring容器

//方案一：强转到子类

//((ClassPathXmlApplicationContext)applicationContext).close();

//方案二：反射机制调用close方法

//获取引用applicationContext的运行时类（即子类ClassPathXmlApplicationContext），再获取该子类的close方法

Method method = applicationContext.getClass().getMethod("close");

//方法调用。invoke参数1：拥有该方法的对象的名字，参数2：方法里面的参数的值

method.invoke(applicationContext);

}

测试结果：

AnnoLifeCycleBean构造器调用了

AnnoLifeCycleBean-init初始化时调用

[INFO ] 2018-05-09 13:53:05,995 [method]org.springframework.context.support.AbstractApplicationContext.doClose(AbstractApplicationContext.java:982)

Closing org.springframework.context.support.ClassPathXmlApplicationContext@4c51cf28: startup date [Wed May 09 13:52:19 CST 2018]; root of context hierarchy

AnnoLifeCycleBean-destroy销毁时调用

【扩展】

手动关闭spring容器applicationContext的两种方式：

//方案一：强转到子类

//((ClassPathXmlApplicationContext)applicationContext).close();

//方案二：反射机制调用close方法

//获取引用applicationContext的运行时类（即子类ClassPathXmlApplicationContext），再获取该子类的close方法

Method method = applicationContext.getClass().getMethod("close");

//方法调用。invoke参数1：拥有该方法的对象的名字，参数2：方法里面的参数的值

method.invoke(applicationContext);

【注意】

如果要执行对象的销毁方法，需满足两个条件：

1：单例Bean（在容器close时，单例Bean才会执行销毁方法）

2：必须调用容器（applicationContext）close()

## **bean的作用域**

通过@Scope注解，指定bean的作用域（默认是singleton，所以单例不用配置）

package com.test.ioc.bean.anno;

@Component

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

@Scope("prototype")

public class AnnoScopeBean {

public AnnoScopeBean() {

super();

System.out.println("AnnoScopeBean的构造执行了");

}

@PostConstruct

public void init(){

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

}

@PreDestroy

public void destroy(){

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

}

}

测试类

@Test

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

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

//多例；此时，才执行构造和初始化后调用的方法

AnnoScopeBean annoScopeBean = (AnnoScopeBean)applicationContext.getBean("annoScopeBean");

((ClassPathXmlApplicationContext)applicationContext).close();

}

DEBUG测试结果：

1.多例getBean时才创建和初始化实例

2.多例并没有执行销毁方法（没有销毁）

AnnoScopeBean的构造执行了

AnnoScopeBean-init初始化时调用

[INFO ] 2018-05-09 14:14:17,986 [method]org.springframework.context.support.AbstractApplicationContext.doClose(AbstractApplicationContext.java:982)

Closing org.springframework.context.support.ClassPathXmlApplicationContext@3930015a: startup date [Wed May 09 14:14:17 CST 2018]; root of context hierarchy

## XML和注解混合配置

目前，一般项目中XML和注解都有。以后的趋势：纯注解开发

背景：Spring2.0就有@Autowired，而Spring2.5之后才有@Component等，目前是两者转换的过度期

一般使用：

XM：完成其他

注解：完成Bean定义、属性注入（IOC + DI）

创建包：com.test.ioc.bean.mixed

第一步：

创建ProductDao.java、ProductService.java

**package** com.test.ioc.bean.mixed;

@Repository

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

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

System.***out***.println("ProductDao-save()执行了");

}

}

**package** com.test.ioc.bean.mixed;

@Service

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

@Autowired

**private** ProductDao productDao;

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

System.***out***.println("ProductService-save执行了");

productDao.save();

}

}

第二步：

创建applicationContext-mixed.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

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"*>

<!-- 开启注解功能 -->

<!-- <context:annotation-config/> -->

<!-- 包扫描 -->

<context:component-scan base-package=*"com.test.ioc.bean.mixed"*/>

</beans>

第三步：测试类

**package** com.test.ioc.bean.mixed;

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

@Test

**public** **void** testAnno() {

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

ProductService productService=(ProductService) ac.getBean("productService");

productService.save();

}

}

测试结果：

ProductService-save执行了

ProductDao-save()执行了

# Spring的web集成

该部分重点：将Spring容器ApplicationContext绑定成ServletContext属性，保证Web应用中只有一个spring容器

可以再建一个web项目，这里就只创建一个新包：com.test.ioc.bean.web

创建HelloService.java

**package** com.test.ioc.bean.web;

@Service

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

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

System.***out***.println("HelloService-sayHello调用了");

}

}

创建applicationContext-web.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

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"*>

<!-- 开启注解功能 -->

<!-- <context:annotation-config/> -->

<!-- 包扫描 -->

<context:component-scan base-package=*"com.test.ioc.bean.web"*/>

</beans>

创建HelloServlet.java

@WebServlet是Servlet3.0支持的注解方式。如果不使用该注解，就需要在web.xml中配置

<servlet>

<servlet-name>HelloServlet</servlet-name>

<servlet-class>com.test.ioc.bean.web.HelloServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

package com.test.ioc.bean.web;

//@WebServlet(name = "HelloServlet", urlPatterns="/", loadOnStartup = 1)

public class HelloServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

//传统方式：

//HelloService helloService = new HelloService();

ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext-web.xml");

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

helloService.sayHello();

response.setContentType("text/html;charset=utf-8");//中文乱码

response.getWriter().print("来自HelloServlet响应");

}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

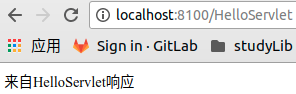
this.doGet(request, response);

}

}

将程序部署到tomcat/Jetty：

访问如下



【思考】

直接new ClassPathXmlApplicationContext()有什么问题？

缺点：在创建Spring容器时，需要对容器中的对象实例创建和初始化，这样会消耗资源，降低性能

思路：保证spring容器对象唯一，即只有一个ApplicationContext实例

方案：

将Spring容器绑定到Web Servlet容器上。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，可以自定义一个ServletContextListener监听器，在监听到ServletContext创建时，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) 。 而我们也无需手动自定义该监听器，因为Spring提供了一个叫ContextLoaderListener的监听器，位于spring-web-4.3.2.RELEASE.jar中，直接拿来用

绑定步骤：

第一步：导入spring web的spring-web-4.3.2.RELEASE.jar

第二步：在web.xml 配置Spring提供的核心监听器

<!-- spring web监听器，监听到ServletContext创建时，跟着创建ApplicationContext，并作为属性绑定到ServletContext上 -->

<listener>

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

</listener>

第三步：启动tomcat/jetty，结果发现异常，没找到applicationContext.xml文件

Caused by:

java.io.FileNotFoundException: Could not open ServletContext resource [/WEB-INF/applicationContext.xml]

【原因】

在ContextLoader.class中有如下说明

需要配置名为contextConfigLocation的context-param参数，来指定spring配置文件的位置；如果没有配置，默认到/WEB-INF/applicationContext.xml下去找

\* <p>Processes a {@link #CONFIG\_LOCATION\_PARAM "contextConfigLocation"} context-param

\* and passes its value to the context instance, parsing it into potentially multiple

\* file paths which can be separated by any number of commas and spaces, e.g.

\* "WEB-INF/applicationContext1.xml, WEB-INF/applicationContext2.xml".

\* Ant-style path patterns are supported as well, e.g.

\* "WEB-INF/\*Context.xml,WEB-INF/spring\*.xml" or "WEB-INF/&#42;&#42;/\*Context.xml".

\* If not explicitly specified, the context implementation is supposed to use a

\* default location (with XmlWebApplicationContext: "/WEB-INF/applicationContext.xml").

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

<!-- spring web监听器，监听到ServletContext创建时，跟着创建ApplicationContext，并作为属性绑定到ServletContext上 -->

<listener>

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

</listener>

<!-- 指定spring配置文件路径 -->

<context-param>

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

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

</context-param>

重新启动服务器，可以找到配置文件。

第四步：修改Servlet代码

在Servlet中，通过ServletContext来获取Spring容器对象

如果还用new对象，多个请求时会执行多次doGet方法（service方法），也就会创建多个Spring容器实例。而通过ServlectContext得到的ApplicationContext是同一个实例（从ServlectContext属性中获取的，getAttribute）

在doGet中使用new来获取ApplicationContext时，多次请求的日志如下，会多次加载xml和创建新的spring容器实例

[INFO ] 2018-05-09 16:49:41,835 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1159c435: startup date [Wed May 09 16:49:41 CST 2018]; root of context hierarchy

[INFO ] 2018-05-09 16:49:41,836 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext-web.xml]

HelloService-sayHello调用了

[INFO ] 2018-05-09 17:02:50,201 [method]org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:581)

Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@749feaea: startup date [Wed May 09 17:02:50 CST 2018]; root of context hierarchy

[INFO ] 2018-05-09 17:02:50,203 [method]org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)

Loading XML bean definitions from class path resource [applicationContext-web.xml]

HelloService-sayHello调用了

从ServletContext中获取ApplicationContext的方式有2种

* 第一种方式：getAttribute
* 第二种方式：使用WebApplicationContextUtils （推荐）

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

//传统方式：

//HelloService helloService = new HelloService();

//使用new，多个请求还是会多次加载xml，多次创建ac实例

//ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext-web.xml");

//方式1

//ApplicationContext ac = (ApplicationContext) request.getServletContext().getAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE);

//ApplicationContext ac = (ApplicationContext) this.getServletContext().getAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE);

//方式2

//WebApplicationContext是ApplicationContext的子接口

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

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

helloService.sayHello();

response.setContentType("text/html;charset=utf-8");

response.getWriter().print("来自HelloServlet响应");

}

启动测试：多次请求，不再多次创建ApplicationContext

HelloService-sayHello调用了

HelloService-sayHello调用了

# Spring的junit集成

Spring提供的spring-test-4.3.2.RELEASE.jar，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）

第一步：项目导入junit 开发包 （使用@Test，导Junit包，或在项目的Java Build Path中添加Junit的Library）

第二步：导入spring-test-4.3.2.RELEASE.jar

第三步：编写测试类

通过@RunWith，使用junit整合spring，开启该类下的注解功能（不是全局）

通过@ContextConfiguration注解，指定spring容器的配置文件位置

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

@ContextConfiguration(locations="classpath:applicationContext-web.xml")//加载核心配置文件，（也是在该类中）自动构建spring容器

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

//使用Autowired注解注入要测试的bean（不使用ac.getBean了）

@Autowired

**private** HelloService helloService;

@Test

**public** **void** testHello(){

//1.获取spring容器

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

//2.从spring容器中获取bean对象

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

helloService.sayHello();

}

}

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

集成的作用就是用@RunWith、@ContextConfiguration和@Autowired省略上述1和2步骤

【注意】

1. @Autowired会将测试对象helloService注入到测试用例SpringTest中

2.测试用例不需要配置<context:annotion-config/>，使用测试类运行的时候，（@RunWith）会自动启动注解支持