# Spring学习大纲

第一部分： spring框架的概述以及spring中基于XML的IOC（反转控制）配置

第二部分：Spring中基于注解的IOC和IOC案例

第三部分：spring中的AOP和基于XML以及注解的AOP（面向切面编程）配置

第四部分：spring中的JDBC Temlate以及Spring事物控制

### Spring学习第一天——Spring的IOC-控制反转

在这一部分学习内容中，主要的内容是学习Spring的控制反转。实现控制反转主要有两种方式——基于xml配置文件实现，以及xml文件的各种标签的使用。在这一部分学习中，主要学习了什么是控制反转，以及xml配置实现控制方法，以及xml的细节内容。

**1、Spring概述**

**1.1 Spring是什么**

Spring 框架是一个分层架构，由 7 个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式。Spring**以IoC（Inverse Of Control：反转控制）和AOP（Aspect Oriented Programming：面向切面编程）为内核**。

**Spring的优势：**

**1.2 程序的耦合与解耦**

**1.2.1程序的耦合**

耦合指的就是就是**对象之间的依赖性**。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是**高内聚低耦合**。内聚是从功能角度来度量模块内的联系，一个好的内聚模块应当恰好做一件事。它描述的是模块内的功能联系。

解耦：降低程序间的依赖关系。

在实际开发中应该做到：**编译期不依赖，运行时才依赖**。

耦合实例1：

JDBC操作中注册驱动时,使用DriverManager的register方法的流程如下：

//1.注册驱动

//DriverManager.registerDriver(new com.mysql.jdbc.Driver());

//2.获取连接

//3.获取预处理 sql 语句对象

//4.获取结果集

//5.遍历结果集

如果用这样的方式，DriverManager类依赖了数据库的具体驱动类（MySQL），如果这时候更换了数据库类型，需要修改源码来重新数据库驱动。这显然是高耦合的。

因此实际上使用jdbc 时，是通过反射来注册驱动的，代码如下： Class.forName("com.mysql.jdbc.Driver");

此时的好处是，我们的类中不再依赖具体的驱动类，**此时就算删除 mysql 的驱动 jar 包，依然可以编译。**即使驱动类不存在,在编译时也不会报错,解决了**编译器依赖**。实际上，开发中此类名会从properties文件中读取。

耦合实例2：

在Web项目中,UI层,Service层,Dao层之间有着前后调用的关系。

public class AccountServiceImpl implements IAccountService

{

private IAccountDao accountDao = new AccountDaoImpl();

}

**业务层依赖持久层的接口和实现类,若编译时不存在没有持久层实现类,则编译将不能通过,这构成了编译期依赖。**

**1.2.2程序的解耦**

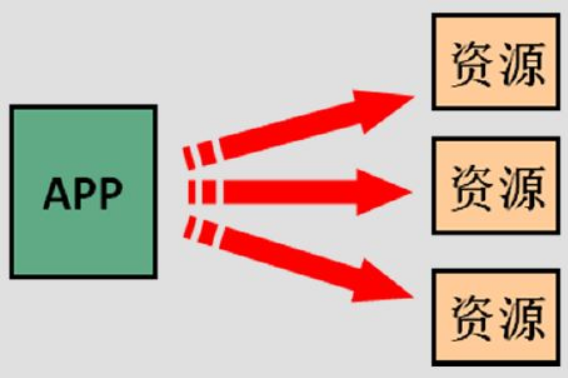
解耦的思路: 使用反射来创建对象,而避免使用new关键字,并通过读取配置文件来获取要创建的对象全限定类名。

在实际开发中可以把三层的对象的全类名都使用配置文件保存起来,当启动服务器应用加载的时候,创建这些对象的实例并保存在容器中. 在获取对象时,不使用new的方式,而是直接从容器中获取,这就是工厂设计模式。

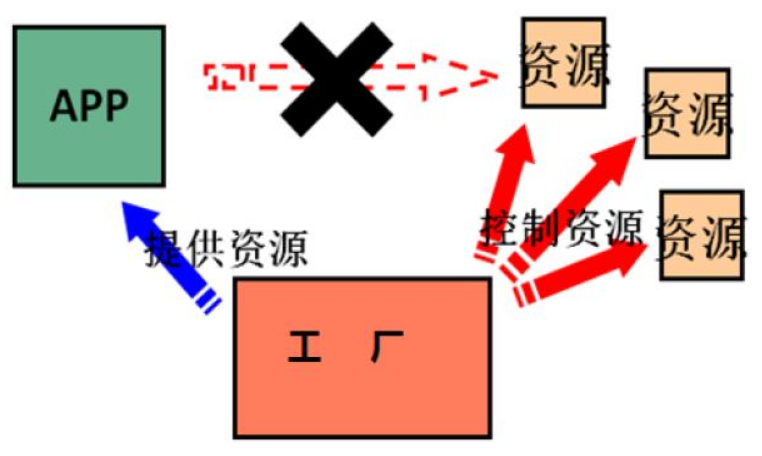
工厂是什么？**这个读取配置文件，创建和获取三层对象的类就是工厂。**

工厂负责给我们从容器中获取指定对象的类。这时候我们获取对象的方式发生了改变。

我们在获取对象时，都是采用 new 的方式。是主动的。



现在我们获取对象时，同时跟工厂（容器）获取，有工厂为我们查找或者创建对象，**是被动的**。



这种**被动接收**的方式（**由主动变被动**）获取对象的思想就是控制反转IOC，它是spring框架的核心之一。

**1.3 Spring的IOC**

IoC对于spring框架来说，就是由**spring来负责控制对象的生命周期和对象间的关系**,而不是由程序直接控制。

控制 ：指的是对象创建（实例化、管理）的权力

反转 ：控制权交给外部环境（Spring 框架、IoC 容器）

A对象要使用B对象，就主动创建一个B对象，那么A就对B产生了依赖，也就是A和B之间存在一种耦合关系，而在spring中，创建对象B的工作是由Spring来做的，Spring创建好B对象，然后存储到一个容器里面，当A对象需要使用B对象时，Spring就从存放对象的那个容器里面取出A要使用的那个B对象，然后交给A对象使用，至于Spring是如何创建那个对象，以及什么时候创建好对象的，A对象不需要关心这些细节问题

所以控制反转IoC(Inversion of Control)是说创建对象的控制权进行转移，传统创建对象的主动权是由自己把控的，而现在这种权力转移到第三方，即spring，spring控制所有对象，程序只需要被动的接受就行了。

**2、Spring IOC的xml配置实现**

**2.1 准备工作**  
创建maven项目,配置其pom.xml如下:

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

<project xmlns="http://maven.apache.org/POM/4.0.0"

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

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itheima</groupId>

<artifactId>learnspring</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- 引入-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.0.2.RELEASE</version>

</dependency>

</dependencies>

</project>

2.2三层接口类和实现类

业务层：

public interface IAccountService

{

void saveAccount();

}

public class AccountServiceImpl implements IAccountService

{

private IAccountDao accountDao = new AccountDaoImpl();//此处的依赖关系有待解决 @Override

public void saveAccount()

{

accountDao.saveAccount();

}

}

可以看到，此处的业务层类与持久层类IAccountDao存在耦合关系

持久层:

public interface IAccountDao

{

void saveAccount();

}

public class AccountDaoImpl implements IAccountDao

{

@Override

public void saveAccount()

{

System.out.println("保存了账户");

}

}

可以看到，此时需要我们解决业务层与逻辑层类的耦合关系，添加xml配置文件来进行解耦。

**2.3 xml配置文件**

首先，创建bean.xml文件：



给xml文件导入约束：

<?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">

把对象的创建交给spring来管理：

<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"></bean>

<bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl"></bean>

</beans>

在这里先后定义了accountService和accountDao两个类的容器，其中

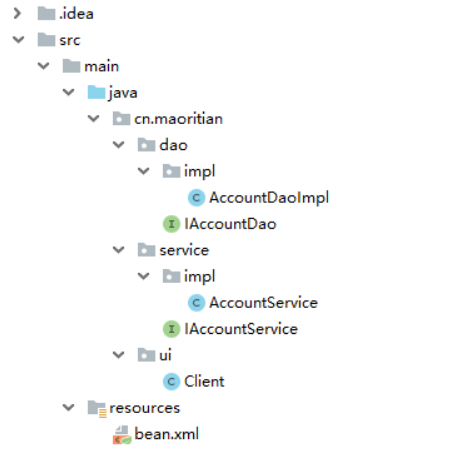
bean 标签：用于配置让 spring 创建对象，并且存入 ioc 容器之中

id 属性：对象的唯一标识。

class 属性：指定要创建对象的全限定类名

这样一来，就完成了Spring IOC的xml文件的配置。

整个项目结构如下：



**2.4通过核心容器创建对象**

在代码中创建核心容器，通过核心容器的getBean()方法获取具体对象

// 获取核心容器对象

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

// 根据id获取bean对象

IAccountService as = (IAccountService) ac.getBean("accountService");

IAccountDao adao = ac.getBean("accountDao",IAccountDao.class);

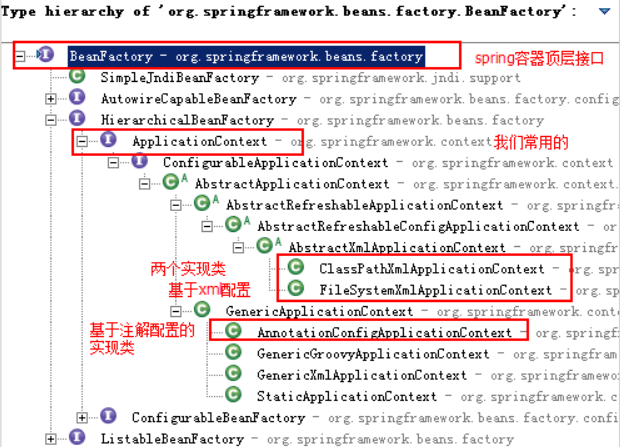
在这里，我们的核心容器实际上相当于一个map。

常用的容器有3中：ClassPathXmlApplicationContext,FileSystemXmlApplicationContext,AnnotationConfigApplicationContext.

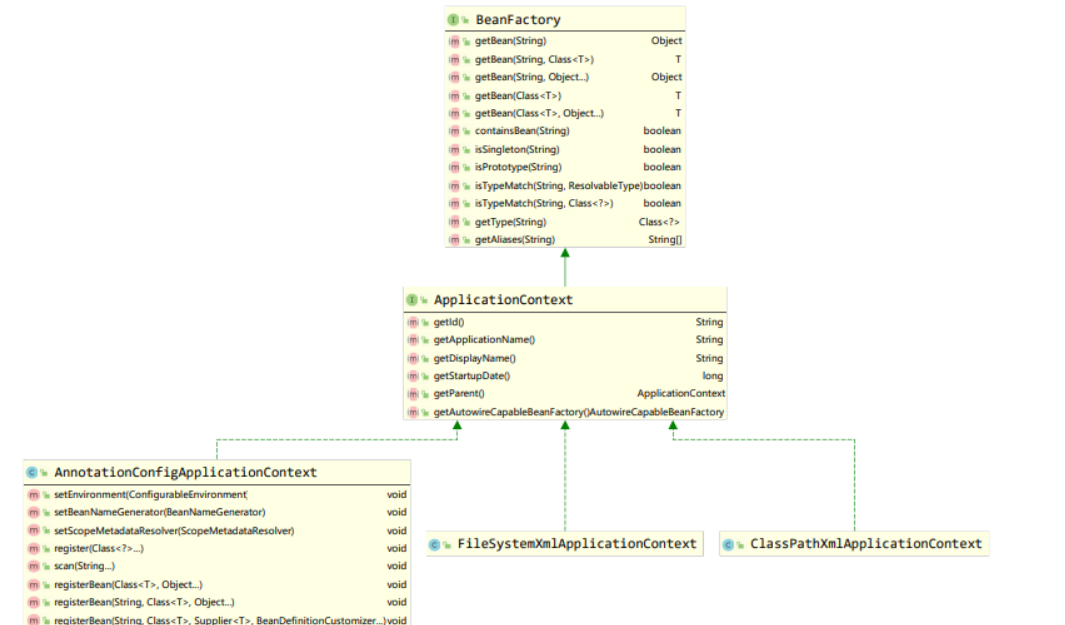
1. ClassPathXmlApplicationContext: 它是从类的根路径下加载配置文件（推荐）

2、FileSystemXmlApplicationContext: 它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。

3、AnnotationConfigApplicationContext: 读取注解创建容器，当我们使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。



BeanFactory和ApplicationContext 的区别：



ApplicationContext：它在创建核心容器时，创建对象采取的策略是采用**立即加载**的方式，也就是说，只要一读取完配置文件就马上创建配置文件中配置的对象

单例对象适用

开发中常采用此接口

BeanFactory:它在构建核心容器时，创建对象的策略是采用**延迟加载**的方式，什么时候获取id对象了，什么时候就创建对象。

多例对象适用

2.5 小结

在这一部分中，学习了用xml实现springIOC的方法与步骤，从创建bean.xml、导入约束，定义类的容器以及利用容器创建类、容器的关系等等，主要学习的是xml的基本使用方法。

**3、xml配置bean标签细节内容**

**3.1bean 标签**

作用： 用于配置对象让 spring 来创建的。

**默认情况下它调用的是类中的无参构造函数。如果没有无参构造函数则不能创建成功。**

属性：

id：给对象在容器中提供一个唯一标识。用于获取对象。

class：指定类的全限定类名。用于反射创建对象。默认情况下调用无参构造函数。 scope：指定对象的作用范围。

\* singleton :默认值，单例的.

\* prototype :多例的.

\* request :WEB 项目中,Spring 创建一个 Bean 的对象,将对象存入到 request 域中.

\* session :WEB 项目中,Spring 创建一个 Bean 的对象,将对象存入到 session 域中.

\* global session :WEB 项目中,应用在 Portlet 环境.如果没有 Portlet 环境那么 globalSession 相当于 session.

init-method：指定类中的初始化方法名称。

destroy-method：指定类中销毁方法名称。

### bean的作用范围和生命周期

1、单例对象: scope="singleton"

作用范围: 每个应用只有一个该对象的实例,它的作用范围就是整个应用

生命周期: 单例对象的创建与销毁 和 容器的创建与销毁时机一致

对象出生: 当应用加载,创建容器时,对象就被创建

对象活着: 只要容器存在,对象一直活着

对象死亡: 当应用卸载,销毁容器时,对象就被销毁

1. 多例对象: scope="prototype"

作用范围: 每次访问对象时,都会重新创建对象实例.

生命周期: 多例对象的创建与销毁时机不受容器控制

对象出生: 当使用对象时,创建新的对象实例

对象活着: 只要对象在使用中,就一直活着

对象死亡: 当对象长时间不用时,被 java 的垃圾回收器回收了

### 3.3实例化 Bean 的三种方式

1.使用默认无参构造函数创建对象: **默认情况下会根据默认无参构造函数来创建类对象,若Bean类中没有默认无参构造函数**,将会创建失败.

<bean id="accountService"

class="cn.itheima.service.impl.AccountServiceImpl"></bean>

2、使用静态工厂的方法创建对象:

创建静态工厂如下:

// 静态工厂,其静态方法用于创建对象

public class StaticFactory {

public static IAccountService createAccountService(){

return new AccountServiceImpl();

}

}

使用StaticFactory类中的静态方法createAccountService创建对象,涉及到<bean>标签的属性:

1. id属性: 指定对象在容器中的标识,用于从容器中获取对象
2. class属性: 指定静态工厂的全类名

3、factory-method属性: 指定生产对象的静态方法

<bean id="accountService"

class="cn.itheima.factory.StaticFactory"

factory-method="createAccountService"></bean>

实际上，类的构造函数也是静态方法,因此默认无参构造函数也可以看作一种静态工厂方法

3、使用实例工厂的方法创建对象

创建实例工厂如下:

public class InstanceFactory {

public IAccountService createAccountService(){

return new AccountServiceImpl();

}

}

先创建实例工厂对象instanceFactory,通过调用其createAccountService()方法创建对象,涉及到<bean>标签的属性:

1. factory-bean属性: 指定实例工厂的id

2、factory-method属性: 指定实例工厂中生产对象的方法

<bean id="instancFactory" class="cn.itheima.factory.InstanceFactory"></bean>

<bean id="accountService"

factory-bean="instancFactory"

factory-method="createAccountService"></bean>

## 3.4依赖注入

**在3.3中学习了spring的实例创建方法，此外，Spring还通过依赖注入来创建存在耦合的类和属性。**

**3.4.1** Spring中的依赖注入

依赖注入(Dependency Injection)是spring框架核心ioc的具体实现.

通过控制反转,我们把创建对象托管给了spring,但是代码中不可能消除所有依赖,例如:业务层仍然会调用持久层的方法,因此业务层类中应包含持久化层的实现类对象。  
我们等待框架通过配置的方式将持久层对象传入业务层,而不是直接在代码中new某个具体的持久化层实现类,这种方式称为依赖注入.

#### 3.4.2使用构造函数注入

通过类默认的构造函数来给创建类的字段赋值,相当于调用类的构造方法.

涉及的标签: <constructor-arg>用来定义构造函数的参数,其属性可大致分为两类:

1. 寻找要赋值给的字段

1. index: 指定参数在构造函数参数列表的索引位置

2. type: 指定参数在构造函数中的数据类型

3. name: 指定参数在构造函数中的变量名,**最常用的属性**

2. 指定赋给字段的值

1. value: 给基本数据类型和String类型赋值

2. ref: 给其它Bean类型的字段赋值,**ref属性的值应为配置文件中配置的Bean的id**

实体类代码:

public AccountServiceImpl(String name, Integer age, Date birthday) {

this.name = name;

this.age = age;

this.birthday = birthday;

xml配置:

<!-- 使用Date类的无参构造函数创建Date对象 -->

<bean id="now" class="java.util.Date" scope="prototype"></bean>

<bean id="accountService" class=" com.itheima.service.impl.AccountServiceImpl">

<constructor-arg name="name" value="myname"></constructor-arg>

<constructor-arg name="age" value="18"></constructor-arg>

<!-- birthday字段为已经注册的bean对象,其id为now -->

<constructor-arg name="birthday" ref="now"></constructor-arg>

</bean>

**3.4.3 set 方法注入**

在类中提供需要注入成员属性的set方法,创建对象只调用要赋值属性的set方法.

涉及的标签: <property>,用来定义要调用set方法的成员. 其主要属性可大致分为两类:

1、指定要调用set方法赋值的成员字段

name：要调用set方法赋值的成员字段

2、指定赋给字段的值

value: 给基本数据类型和String类型赋值

ref: 给其它Bean类型的字段赋值,ref属性的值应为配置文件中配置的Bean的id

实体类代码：

public void setName(String name) {

this.name = name;

}

public void setAge(Integer age) {

this.age = age;

}

public void setBirthday(Date birthday) {

this.birthday = birthday;

}

xml配置：

<!-- 使用Date类的无参构造函数创建Date对象 -->

<bean id="now" class="java.util.Date" scope="prototype"></bean>

<bean id="accountService" class=" com.itheima.service.impl.AccountServiceImpl ">

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

<property name="age" value="21"></property>

<!-- birthday字段为已经注册的bean对象,其id为now -->

<property name="birthday" ref="now"></property>

</bean>

**3.4.4注入集合字段**

集合字段及其对应的标签按照集合的结构分为两类: 相同结构的集合标签之间可以互相替换，本次学习了数组，List,Set,Map,Properties。

**只有键的结构:**

数组字段: <array>标签表示集合,<value>标签表示集合内的成员.

List字段: <list>标签表示集合,<value>标签表示集合内的成员.

Set字段: <set>标签表示集合,<value>标签表示集合内的成员.

其中<array>,<list>,<set>标签之间可以互相替换使用.

**键值对的结构:**

Map字段: <map>标签表示集合,<entry>标签表示集合内的键值对,其key属性表示键,value属性表示值.

Properties字段: <props>标签表示集合,<prop>标签表示键值对,其key属性表示键,标签内的内容表示值.

其中<map>,<props>标签之间,<entry>,<prop>标签之间可以互相替换使用.

实体类的字段：

private String[] myArray;

private List<String> myList;

private Set<String> mySet;

private Map<String,String> myMap;

private Properties myProps;

xml配置：

<bean id="accountService" class=" com.itheima.service.impl.AccountServiceImpl ">

<!--给数组注入数据-->

<property name="myStrs">

<array>

<value>value1</value>

<value>value2</value>

<value>value3</value>

</array>

</property>

<!--注入 list 集合数据-->

<property name="myList">

<list>

<value>value1</value>

<value>value2</value>

<value>value3</value>

</list>

</property>

<!--注入 set 集合数据 --->

<property name="mySet">

<set>

<value>value1</value>

<value>value2</value>

<value>value3</value>

</set>

</property>

<!--注入 Map 数据-->

<property name="myMap">

<map>

<entry key="key1" value="value1"></entry>

<entry key="key2">

<value>value2</value>

</entry>

</map>

</property>

<!--注入 properties 数据-->

<property name="myProps">

<props>

<prop key="key1">value1</prop>

<prop key="key2">value2</prop>

</props>

</property>

</bean>

**注意：在注入集合数据时，只要结构相同，标签可以互换。**

其中，List 结构的： array,list,set

Map 结构的 map,entry,props,prop

**3.5 小结**

在这一部分中，学习了使用bean标签具体实例化一个类的各种细节。首先，学习了bean标签内部的一些基本标签，声明该实例的注入属性、作用范围和生命周期等等。接着学习了实例化一个bean时的三种方法，以及实体类中所需要的函数：构造函数等等。最后，通过依赖注入我们可以注入实体类中的字段/属性的三种方法，同时实体类中也需要像一个的构造函数、set方法。

**4、总结**

在这一部分学习内容中，首先，我们了解了什么是Spring的IOC，主要的内容是学习Spring的控制反转。实现控制反转的一种方式——基于xml配置文件实现，以及xml文件的各种标签的使用。在这一部分中还学习了使用bean标签进行实例化的三种方法，以及对实体类中的各种基本属性与集合属性进行依赖注入的方法。