# 回顾：

<resultMap type=”com.itheima.domain.User”>

<collection javaType=”java.util.List” ofType=”com.itheima.domain.Account”>

</collection>

one of my best friends

</resultMap>

<resultMap type=”com.itheima.domain.Account”>

<association javaType=”com.itheima.domain.User” >

</association>

</resultMap>

领域模型命名规约  
1） 数据对象： xxxDO， xxx 即为数据表名。 UserDO  
2） 数据传输对象： xxxDTO， xxx 为业务领域相关的名称。 ProductDTO  
3） 展示对象： xxxVO， xxx 一般为网页名称。 LoginVO  
4） POJO 是 DO/DTO/BO/VO 的统称，禁止命名成 xxxPOJO。

DO（Data Object） ：与数据库表结构一一对应，通过 DAO 层向上传输数据源对象。  
DTO（Data Transfer Object） ：数据传输对象， Service 或 Manager 向外传输的对象。  
BO（Business Object） ：业务对象。 由 Service 层输出的封装业务逻辑的对象。  
AO（Application Object）： 应用对象。 在 Web 层与 Service 层之间抽象的复用对象模型，  
极为贴近展示层，复用度不高。  
VO（View Object） ： 显示层对象，通常是 Web 向模板渲染引擎层传输的对象。  
Query：数据查询对象，各层接收上层的查询请求。 注意超过 2 个参数的查询封装，禁止  
使用 Map 类来传输。 UserQuery private Date startDate; private Date endDate;

# 今日重点：

1.了解Spring

2.理解程序的耦合与解耦：工厂模式+反射+配置文件（自定义）

3.理解Spring三大经典思想之IOC

4.掌握使用 Spring 的 IOC 解决程序耦合的案例

# 了解Spring

## Spring是什么？

官网：https://spring.io/

Spring是JavaSE/JavaEE企业开发的一站式（全栈）的轻量级开源框架。

请求

-> 表现层（web层接收请求并且响应） SpringMVC/Struts2...

-> 业务层（Service层） Spring不仅仅在业务层

-> 持久层（Dao层 Mapper） Mybatis/hibernate...

-> 数据库（mysql、oracle）

## 1.2 Spring的发展历程

1997 年 IBM 提出了 EJB 的思想  
1998 年， SUN 制定开发标准规范 EJB1.0  
1999 年， EJB1.1 发布  
2001 年， EJB2.0 发布  
2003 年， EJB2.1 发布  
2006 年， EJB3.0 发布  
Rod Johnson（Spring 之父，音乐、金融、计算机全才）  
Expert One-to-One J2EE Design and Development(2002)  
阐述了 J2EE 使用 EJB 开发设计的优点及解决方案  
Expert One-to-One J2EE Development without EJB(2004)  
阐述了 J2EE 开发不使用 EJB 的解决方式（Spring 雏形）  
2017 年 9 月份发布了 spring 的最新版本 spring 5.0 通用版（GA）

## 1.3 Spring的优势

### 1.3.1方便解耦，简化开发

通过 Spring 提供的 IoC 容器，可以将对象间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

### 1.3.2 AOP编程的支持

通过 Spring 的 AOP 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付。

### 1.3.3声明式事务的支持

可以将我们从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活的进行事务的管理，提高开发效率和质量。

### 1.3.4方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。

**1.3.5方便集成各种优秀框架**

Spring 可以降低各种框架的使用难度，提供了对各种优秀框架（Struts、Hibernate、Hessian、 Quartz等）的直接支持。

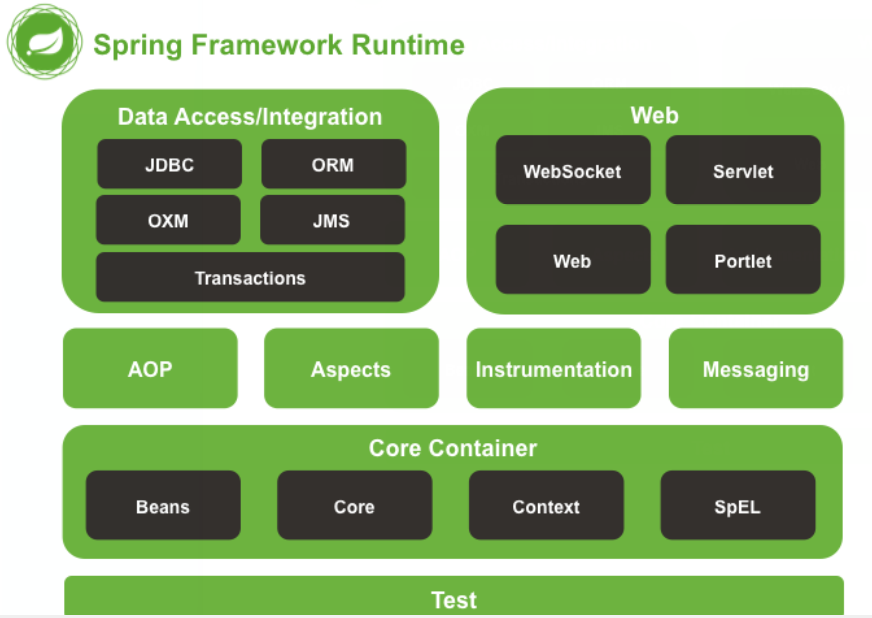
### 1.3.6降低 JavaEE API 的使用难度

Spring 对 JavaEE API（如 JDBC、 JavaMail、远程调用等）进行了薄薄的封装层，使这些 API 的使用难度大为降低。

### 1.3.7 Spring的源码是经典学习范例

Spring 的源代码设计精妙、结构清晰、匠心独用，处处体现着大师对 Java 设计模式灵活运用以及对 Java 技术的高深造诣。它的源代码无意是 Java 技术的最佳实践的范例。

## 1.4 Spring的体系架构



参考

spring-framework-5.0.2.RELEASE-dist\spring-framework-5.0.2.RELEASE\docs\spring-framework-reference\images下的spring-overview.png这张图

# 2.理解程序的耦合

## 2.1什么是程序的耦合？

耦合是指程序间的依赖关系，包括：类之间的依赖和方法间的依赖。

注意：程序讲究的是低耦合，高内聚。就是同一个模块内的各个元素之间要高度紧密，但是各个模块之间的相互依存度却要不那么紧密。

举生活中的例子：搭建舞台的架子，由很多部分组成，每个部分之间是互相独立的，这叫低耦合，各个部分可以紧密结合在一起，这叫高内聚。

举编程上的例子：登录功能与注册功能，我们在开发的时候，登录功能与注册功能是互相独立的，这叫低耦合；同时登录功能与注册功能要结合起来形成完整流程，这叫高内聚。

## 2.2解耦的思路？

为什么要解耦？就是为了提高我们程序的灵活性和可扩展性。

解耦是指降低程序间的依赖关系，在实际开发中：我们应该做到：编译期不依赖，运行时才依赖。

解耦的思路：

第一步：使用反射来创建对象，而避免使用new关键字。

第二步：通过读取配置文件来获取要创建的对象全限定类名。

具体做法：用到工厂模式+配置文件（可以是xml，也可以是properties属性文件）

比如：  
早期我们的 JDBC 操作，注册驱动时，我们为什么不使用 DriverManager 的 register 方法，而是采用 Class.forName 的方式？  
public class JdbcDemo1 {  
/\*\*  
\* @author 黑马程序员  
\* @Company http://www.ithiema.com  
\* @Version 1.0  
\* @param args  
\* @throws Exception  
\*/  
public static void main(String[] args) throws Exception {  
 //1.注册驱动  
 //DriverManager.registerDriver(new com.mysql.jdbc.Driver());  
 Class.forName("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  
 }  
}  
原因就是：  
我们的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品牌（比如 Oracle） ，需要修改源码来重新数据库驱动。这显然不是我们想要的。

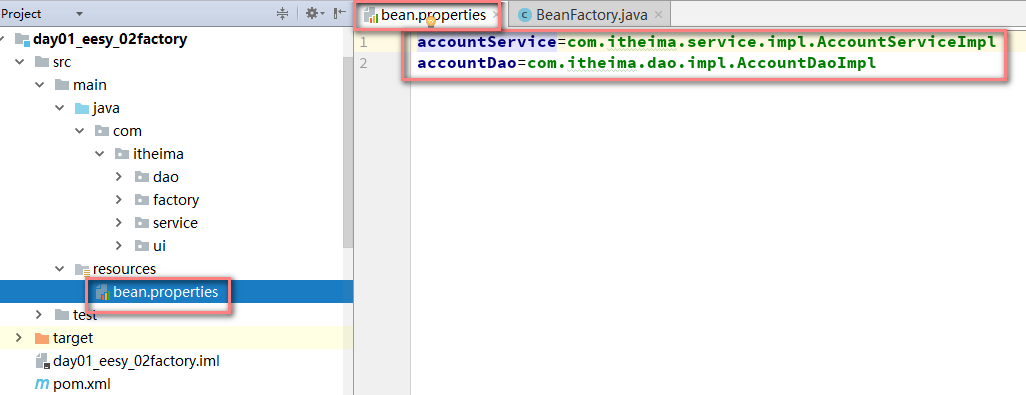
当是我们讲解 jdbc 时，是通过反射来注册驱动的，代码如下：  
Class.forName("com.mysql.jdbc.Driver");//此处只是一个字符串

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（运行就不要想了，没有驱动不可能运行成功的） 。同时，也产生了一个新的问题， mysql 驱动的全限定类名字符串是在 java 类中写死的，一旦要改还是要修改源码。解决这个问题也很简单，使用配置文件配置。

## 2.3工厂模式结合配置文件解耦

在实际开发中我们可以把三层的对象都使用配置文件配置起来，当启动服务器应用加载的时候， 让一个类中的方法通过读取配置文件，把这些对象创建出来并存起来。在接下来的使用的时候，直接拿过来用就好了。那么，这个读取配置文件， 创建和获取三层对象的类就是工厂。

### 2.3.1 在day01\_eesy\_02factory工程里配置文件



### 2.3.2编写工厂类

package com.itheima.factory;

import java.io.InputStream;

import java.util.Enumeration;

import java.util.HashMap;

import java.util.Map;

import java.util.Properties;

/\*\*

\* 一个创建Bean对象的工厂

\*

\* Bean：在计算机英语中，有可重用组件的含义。

\* JavaBean：用java语言编写的可重用组件。

\* javabean > 实体类

\*

\* 它就是创建我们的service和dao对象的。

\*

\* 第一个：需要一个配置文件来配置我们的service和dao

\* 配置的内容：唯一标识=全限定类名（key=value)

\* 第二个：通过读取配置文件中配置的内容，反射创建对象

\*

\* 我的配置文件可以是xml也可以是properties

\*/

public class BeanFactory {

//定义一个Properties对象

private static Properties props;

//使用静态代码块为Properties对象赋值

static {

try {

//实例化对象

props = new Properties();

//获取properties文件的流对象

InputStream in =

BeanFactory.class.getClassLoader().getResourceAsStream("bean.properties");

props.load(in);

}catch(Exception e){

throw new ExceptionInInitializerError("初始化properties失败！");

}

}

/\*\*

\* 根据Bean的名称获取bean对象

\* @param beanName

\* @return

\*/

public static Object getBean(String beanName){

Object bean = null;

try {

String beanPath = props.getProperty(beanName);

//System.out.println(beanPath);

//每次都会调用默认构造函数创建对象

bean = Class.forName(beanPath).newInstance();

}catch (Exception e){

e.printStackTrace();

}

return bean;

}

}

### 2.3.3改进工厂类

package com.itheima.factory;

import java.io.InputStream;

import java.util.Enumeration;

import java.util.HashMap;

import java.util.Map;

import java.util.Properties;

/\*\*

\* 一个创建Bean对象的工厂

\*

\* Bean：在计算机英语中，有可重用组件的含义。

\* JavaBean：用java语言编写的可重用组件。

\* javabean > 实体类

\*

\* 它就是创建我们的service和dao对象的。

\*

\* 第一个：需要一个配置文件来配置我们的service和dao

\* 配置的内容：唯一标识=全限定类名（key=value)

\* 第二个：通过读取配置文件中配置的内容，反射创建对象

\*

\* 我的配置文件可以是xml也可以是properties

\*/

public class BeanFactory {

//定义一个Properties对象

private static Properties props;

//定义一个Map,用于存放我们要创建的对象。我们把它称之为容器

private static Map<String,Object> beans;

//使用静态代码块为Properties对象赋值

static {

try {

//实例化对象

props = new Properties();

//获取properties文件的流对象

InputStream in =

BeanFactory.class.getClassLoader().getResourceAsStream("bean.properties");

props.load(in);

//实例化容器

beans = new HashMap<String,Object>();

//取出配置文件中所有的Key

Enumeration keys = props.keys();

//遍历枚举

while (keys.hasMoreElements()){

//取出每个Key

String key = keys.nextElement().toString();

//根据key获取value

String beanPath = props.getProperty(key);

//反射创建对象

Object value = Class.forName(beanPath).newInstance();

//把key和value存入容器中

beans.put(key,value);

}

}catch(Exception e){

throw new ExceptionInInitializerError("初始化properties失败！");

}

}

/\*\*

\* 根据bean的名称获取对象

\* @param beanName

\* @return

\*/

public static Object getBean(String beanName){

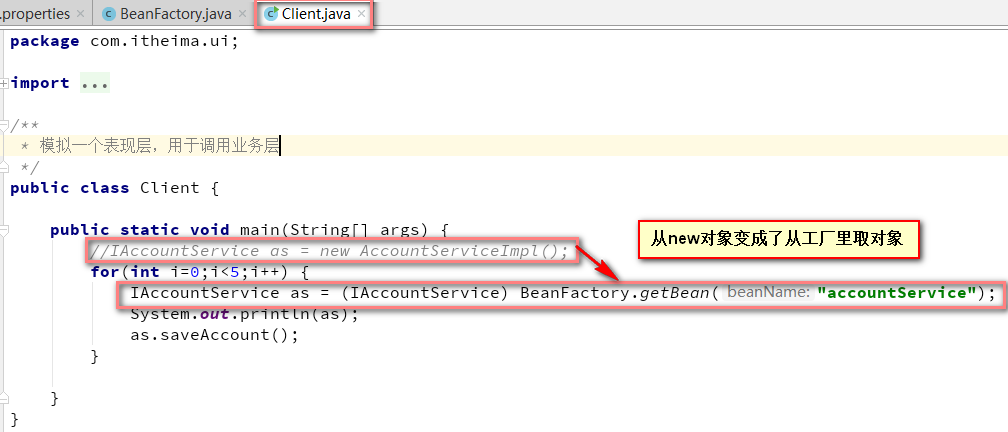
return beans.get(beanName);

}

}

2.3.4 调用业务层的写法改变

对象由原来的new改成了从工厂里获取对象



HashTable用法与HashMap类似

HashTable线程安全的，HashMap是线程不安全的。

Enumeration是个接口，实现类Enumerator，这个类是HashTable的内部类。

# 3.理解Spring三大经典思想（IOC、DI、AOP 面试笔试必考）

## 3.1 IOC

IOC：Inversion Of Control 翻译成“控制反转”

IOC的本质：对象交给Spring来管，而不是自己new。

IOC的作用：  
 削减计算机程序的耦合(解除我们代码中的依赖关系)。

IOC实现原理：工厂模式+反射+xml配置文件

## 3.2 DI

DI：Dependency Injection，简称DI。 翻译成“依赖注入”，它是 spring 框架核心IOC 的具体实现。  
DI的本质：给某个对象的变量赋值。

DI的作用：对象之间有依赖关系，再也不需要我们手动去设置值了，Spring框架会帮我们解决。

DI的原理：通过反射来给对象的变量赋值

## 3.3 AOP

AOP后面会专门讲到。

# 4.掌握使用 Spring 的 IOC 解决程序耦合的案例

本章我们使用的案例是， 账户的业务层和持久层的依赖关系解决。在开始 spring 的配置之前，我们要先准备一下环境。由于我们是使用 Spring 解决依赖关系，并不是真正的要做增删改查操作，所以此时我们没必要写实体类。并且我们在此处使用的是 java 工程，不是 java web 工程。

## 4.1 Spring 的 IOC案例的前期准备

### 4.1.1准备 spring 的开发包

官网： http://spring.io/  
下载地址：  
http://repo.springsource.org/libs-release-local/org/springframework/spring  
解压:(Spring 目录结构:)  
\* docs :API 和开发规范

\* libs :jar 包和源码  
\* schema :约束

### 4.1.2创建业务层接口和实现类

/\*\*  
\* 账户的业务层接口  
\* @author 黑马程序员  
\* @Company http://www.ithiema.com  
\* @Version 1.0  
\*/  
public interface IAccountService {  
 /\*\*  
 \* 保存账户（此处只是模拟，并不是真的要保存）  
 \*/  
 void saveAccount();  
}  
/\*\*  
\* 账户的业务层实现类  
\* @author 黑马程序员  
\* @Company http://www.ithiema.com  
\* @Version 1.0  
\*/  
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();  
 }  
}

### 4.1.3创建持久层接口和实现类

/\*\*  
\* 账户的持久层接口  
\* @author 黑马程序员  
\* @Company http://www.ithiema.com  
\* @Version 1.0  
\*/  
public interface IAccountDao {  
 /\*\*  
 \* 保存账户  
 \*/  
 void saveAccount();  
}  
/\*\*  
\* 账户的持久层实现类  
\* @author 黑马程序员  
\* @Company http://www.ithiema.com  
\* @Version 1.0  
\*/  
public class AccountDaoImpl implements IAccountDao {  
 @Override  
 public void saveAccount() {  
 System.out.println("保存了账户");  
 }  
}

## 4.2基于xml的配置

### 第一步：建工程，导坐标

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

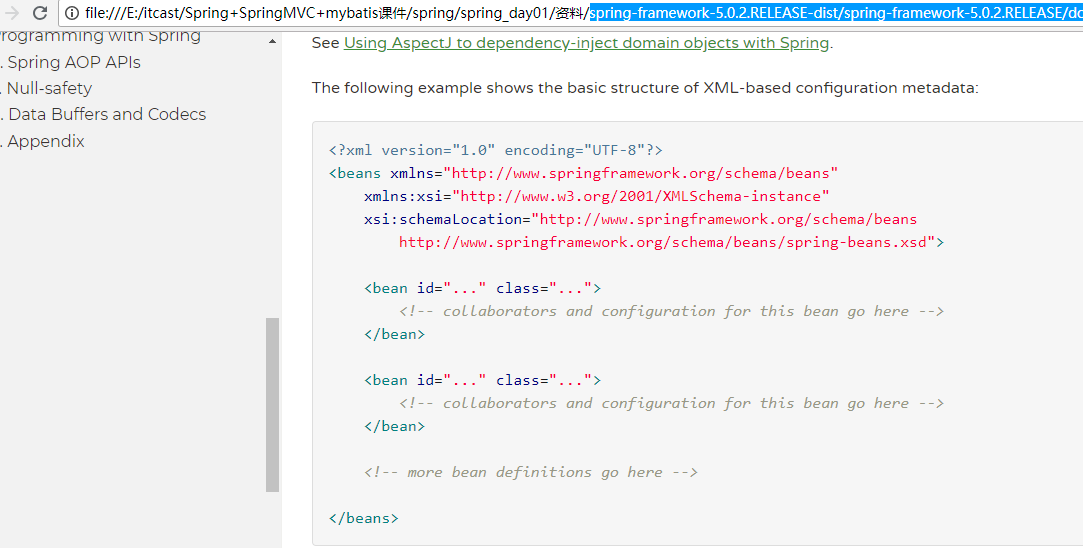
<version>5.0.2.RELEASE</version>

</dependency>

### 第二步：在resources下创建一个任意名称的 xml 文件（不能是中文）

参考：

spring-framework-5.0.2.RELEASE-dist/spring-framework-5.0.2.RELEASE/docs/spring-framework-reference/core.html



### 第三步： 让 Spring 管理对象，在配置文件中配置 service 和 dao

<!-- 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  
-->  
<!-- 配置 service -->  
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"></bean>  
<!-- 配置 dao -->  
<bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl"></bean>

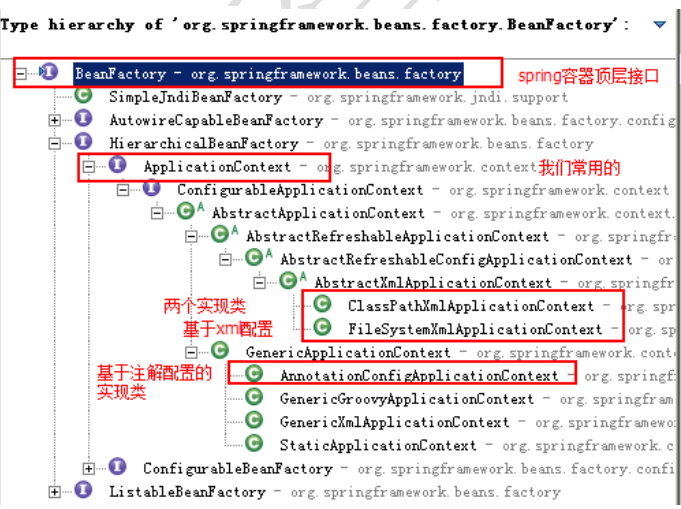
### 第四步：测试配置是否成功

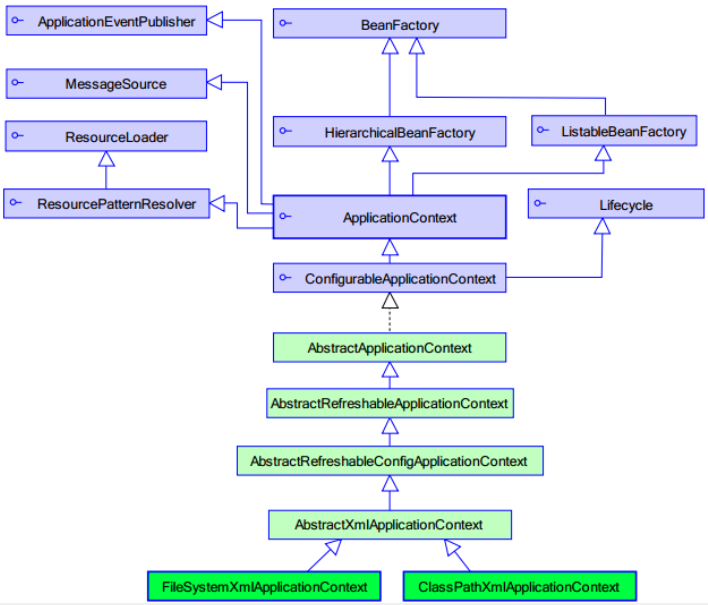
/\*\*  
\* 模拟一个表现层  
\* @author 黑马程序员  
\* @Company http://www.ithiema.com  
\* @Version 1.0  
\*/  
public class Client {  
 /\*\*  
 \* 使用 main 方法获取容器测试执行

\*/  
 public static void main(String[] args) {  
 //1.使用 ApplicationContext 接口，就是在获取 spring 容器  
 ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");  
 //2.根据 bean 的 id 获取对象  
 IAccountService aService = (IAccountService) ac.getBean("accountService");  
 System.out.println(aService);  
 IAccountDao aDao = (IAccountDao) ac.getBean("accountDao");  
 System.out.println(aDao);  
 }  
}

## 4.3掌握Spring基于xml的IOC细节

### 4.3.1 Spring中工厂的类结构图





#### 4.3.1.1 BeanFactory 和 ApplicationContext 的区别

在BeanFactory这棵继承树上的都叫Bean工厂，也叫Spring容器。

BeanFactory 才是 Spring 容器中的顶层接口。ApplicationContext 是它的子接口。  
BeanFactory 和 ApplicationContext 的区别：  
创建对象的时间点不一样：  
ApplicationContext：推荐使用，只要一读取配置文件，默认情况下就会创建对象。

它在构建核心容器时，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马上就创建配置文件中配置的对象。  
BeanFactory：什么时候使用什么时候创建对象。   
它在构建核心容器时，创建对象采取的策略是采用延迟加载的方式。也就是说，什么时候根据id获取对象了，什么时候才真正的创建对象。

#### 4.3.1.2 ApplicationContext 接口的实现类

ClassPathXmlApplicationContext：  
它是从类的根路径下加载配置文件，推荐使用这种  
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。  
AnnotationConfigApplicationContext:  
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。

### 4.3.2 IOC 中 bean标签和管理对象细节

#### 4.3.2.1 bean 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,应用在集群环境.如果没有集群环境那么  
 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。

#### 4.3.2.2 bean 的作用范围和生命周期

单例对象： scope="singleton" Spring默认创建Bean  
 一个应用只有一个对象的实例。它的作用范围就是整个引用。  
 生命周期：  
 对象出生：当应用加载，创建容器时，对象就被创建了。  
 对象活着：只要容器在，对象一直活着。  
 对象死亡：当应用卸载，销毁容器时，对象就被销毁了。  
多例对象： scope="prototype"  
 每次访问对象时，都会重新创建对象实例。  
 生命周期：  
 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。

#### 4.3.2.3实例化 Bean 的三种方式

第一种方式：使用默认无参构造函数  
<!--在默认情况下：  
它会根据默认无参构造函数来创建类对象。如果 bean 中没有默认无参构造函数，将会创建失败。 -->  
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"/>

第二种方式： Spring 管理静态工厂-使用静态工厂的方法创建对象  
/\*\*  
\* 模拟一个静态工厂，创建业务层实现类  
\*/  
public class StaticFactory {  
 public static IAccountService createAccountService(){  
 return new AccountServiceImpl();  
 }  
}  
<!-- 此种方式是:  
使用 StaticFactory 类中的静态方法 createAccountService 创建对象，并存入 spring 容器  
id 属性：指定 bean 的 id，用于从容器中获取  
class 属性：指定静态工厂的全限定类名  
factory-method 属性：指定生产对象的静态方法  
-->  
<bean id="accountService"  
class="com.itheima.factory.StaticFactory"  
factory-method="createAccountService"></bean>

第三种方式： spring 管理实例工厂-使用实例工厂的方法创建对象  
/\*\*  
\* 模拟一个实例工厂，创建业务层实现类  
\* 此工厂创建对象，必须现有工厂实例对象，再调用方法  
\*/  
public class InstanceFactory {  
 public IAccountService createAccountService(){  
 return new AccountServiceImpl();  
 }  
}  
<!-- 此种方式是：  
先把工厂的创建交给 spring 来管理。  
然后在使用工厂的 bean 来调用里面的方法  
factory-bean 属性：用于指定实例工厂 bean 的 id。  
factory-method 属性：用于指定实例工厂中创建对象的方法。  
-->  
<bean id="instancFactory" class="com.itheima.factory.InstanceFactory"></bean>  
<bean id="accountService"  
factory-bean="instancFactory"  
factory-method="createAccountService"></bean>

### 4.3.3 Spring 的依赖注入

#### 4.3.3.1依赖注入的概念

依赖注入： Dependency Injection，简称DI。 它是 spring 框架核心IOC 的具体实现。  
我们的程序在编写时， 通过控制反转， 把对象的创建交给了 spring，但是代码中不可能出现没有依赖的情况。  
IOC解耦只是降低他们的依赖关系，但不会消除。 例如：我们的业务层仍会调用持久层的方法。  
那这种业务层和持久层的依赖关系， 在使用 Spring 之后， 就让 Spring 来维护了。  
简单的说，就是坐等框架把持久层对象传入业务层，而不用我们自己去获取。

#### 4.3.3.2构造函数注入

顾名思义，就是使用类中的构造函数，给成员变量赋值。注意，赋值的操作不是我们自己做的，而是通过配置的方式，让 spring 框架来为我们注入。具体代码如下：  
/\*\*  
\*/  
public class AccountServiceImpl implements IAccountService {  
 private String name;  
 private Integer age;  
 private Date birthday;  
 public AccountServiceImpl(String name, Integer age, Date birthday) {  
 this.name = name;  
 this.age = age;  
 this.birthday = birthday;  
 }  
 @Override  
 public void saveAccount() {  
 System.out.println(name+","+age+","+birthday);  
 }  
}  
<!-- 使用构造函数的方式，给 service 中的属性传值  
要求：  
类中需要提供一个对应参数列表的构造函数。  
涉及的标签：  
 constructor-arg  
 属性：  
 index:指定参数在构造函数参数列表的索引位置  
 type:指定参数在构造函数中的数据类型

name:指定参数在构造函数中的名称 用这个找给谁赋值  
=======上面三个都是找给谁赋值，下面两个指的是赋什么值的==============  
 value:它能赋的值是基本数据类型和 String 类型  
 ref:它能赋的值是其他 bean 类型，也就是说，必须得是在配置文件中配置过的 bean  
-->  
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">  
 <constructor-arg name="name" value="张三"></constructor-arg>  
 <constructor-arg name="age" value="18"></constructor-arg>  
 <constructor-arg name="birthday" ref="now"></constructor-arg>  
</bean>  
<bean id="now" class="java.util.Date"></bean>

#### 4.3.3.3 set 方法注入 ，也叫属性注入（实际开发中，此种方式用的较多）

顾名思义，就是在类中提供需要注入成员的 set 方法。具体代码如下：  
/\*\* \*/  
public class AccountServiceImpl implements IAccountService {  
 private String name;  
 private Integer age;  
 private Date birthday;  
 public void setName(String name) {  
 this.name = name;  
 }  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
 public void setBirthday(Date birthday) {  
 this.birthday = birthday;  
 }  
 @Override  
 public void saveAccount() {  
 System.out.println(name+","+age+","+birthday);  
 }  
}  
<!-- 通过配置文件给 bean 中的属性传值：使用 set 方法的方式  
涉及的标签：  
 property

属性：  
 name：找的是类中 set 方法后面的部分  
 ref：给属性赋值是其他 bean 类型的  
 value：给属性赋值是基本数据类型和 string 类型的  
 实际开发中，此种方式用的较多。  
-->  
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">  
 <property name="name" value="test"></property>  
 <property name="age" value="21"></property>  
 <property name="birthday" ref="now"></property>  
</bean>  
<bean id="now" class="java.util.Date"></bean>

#### 4.3.3.4注入集合属性

顾名思义，就是给类中的集合成员传值，它用的也是 set方法注入的方式，只不过变量的数据类型都是集合。  
我们这里介绍注入数组， List,Set,Map,Properties。具体代码如下：  
/\*\*\*/  
public class AccountServiceImpl implements IAccountService {  
 private String[] myStrs;  
 private List<String> myList;  
 private Set<String> mySet;  
 private Map<String,String> myMap;  
 private Properties myProps;  
 public void setMyStrs(String[] myStrs) {  
 this.myStrs = myStrs;  
 }  
 public void setMyList(List<String> myList) {  
 this.myList = myList;  
 }  
 public void setMySet(Set<String> mySet) {  
 this.mySet = mySet;  
 }  
 public void setMyMap(Map<String, String> myMap) {  
 this.myMap = myMap;  
 }  
 public void setMyProps(Properties myProps) {  
 this.myProps = myProps;  
 }  
 @Override  
 public void saveAccount() {  
 System.out.println(Arrays.toString(myStrs));  
 System.out.println(myList);  
 System.out.println(mySet);  
 System.out.println(myMap);

System.out.println(myProps);  
 }  
}  
<!-- 注入集合数据  
List 结构的：  
 array,list,set  
Map 结构的  
 map,entry,props,prop  
-->  
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">  
 <!-- 在注入集合数据时，只要结构相同，标签可以互换 -->  
 <!-- 给数组注入数据 -->  
 <property name="myStrs">  
 <set>  
 <value>AAA</value>  
 <value>BBB</value>  
 <value>CCC</value>  
 </set>  
 </property>  
 <!-- 注入 list 集合数据 -->  
 <property name="myList">  
 <array>  
 <value>AAA</value>  
 <value>BBB</value>  
 <value>CCC</value>  
 </array>  
 </property>  
 <!-- 注入 set 集合数据 -->  
 <property name="mySet">  
 <list>  
 <value>AAA</value>  
 <value>BBB</value>  
 <value>CCC</value>  
 </list>  
 </property>  
 <!-- 注入 Map 数据 -->  
 <property name="myMap">  
 <props>  
 <prop key="testA">aaa</prop>  
 <prop key="testB">bbb</prop>  
 </props>  
 </property>  
 <!-- 注入 properties 数据 -->

<property name="myProps">  
 <map>  
 <entry key="testA" value="aaa"></entry>  
 <entry key="testB">  
 <value>bbb</value>  
 </entry>  
 </map>  
 </property>  
</bean>

# 作业：

调试IOC的案例