Spring第一天

# Spring概述

## spring概述[了解]

### spring 是什么

### Spring 的发展历程

Spring是分层的、JavaSE/EE一站式(full-stack)、轻量级开源框架。以IoC（Inverse Of Control：反转控制）和AOP（Aspect Oriented Programming：面向切面编程）为内核，提供了表现层Spring MVC和持久层Spring JDBC以及业务层事务管理等众多的企业级应用技术，还能整合开源世界众多著名的第三方框架和类库，逐渐成为使用最多的Java EE企业应用开源框架。

Spring的发展历史：

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）**

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）

### spring 的优势

Spring 出现是为了解决JavaEE 实际问题：

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. **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(面向对象)实现的功能可以通过AOP轻松应付。

1. **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. **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. **方便集成各种优秀框架**

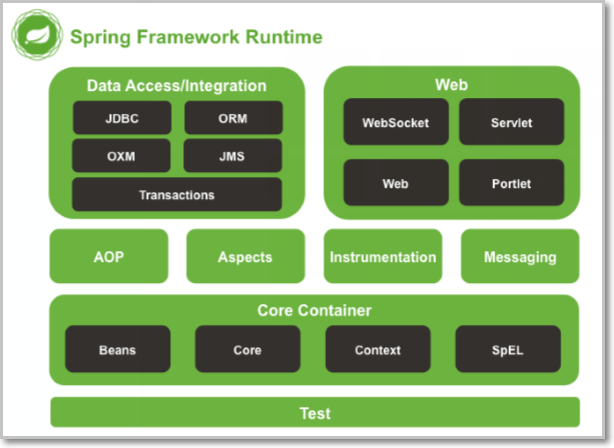
Spring可以降低各种框架的使用难度，提供了对各种优秀框架Struts、Hibernate、Hessian(远程通讯，类似于webService)、Quartz(定时任务)等的直接支持。

1. **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。

关于框架的特性，我们也会俗称Spring为开发架构的粘合剂。

### spring 的体系结构



**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和依赖注入特性。这里的基础概念是BeanFactory，它提供对Factory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**

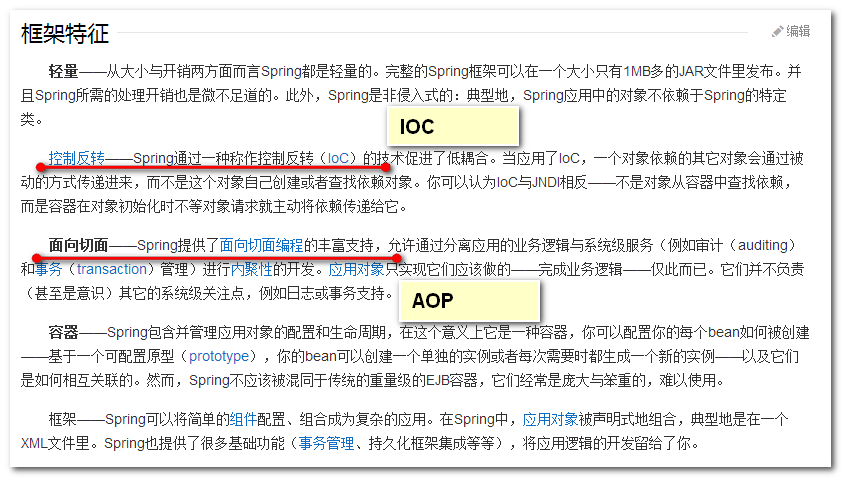
1：Test模块，提供对使用JUnit和TestNG来测试Spring组件的支持，它提供一致的ApplicationContexts并缓存这些上下文，它还能提供一些mock对象，使得你可以独立的测试代码。

## Spring的核心

IoC（Inverse of Control 控制反转）： 将对象创建权利交给Spring工厂进行管理。

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

百度百科：Spring 是基于IOC和AOP的一套编程框架。



今天主要学习IoC，而IOC其实就是一个解耦的过程。

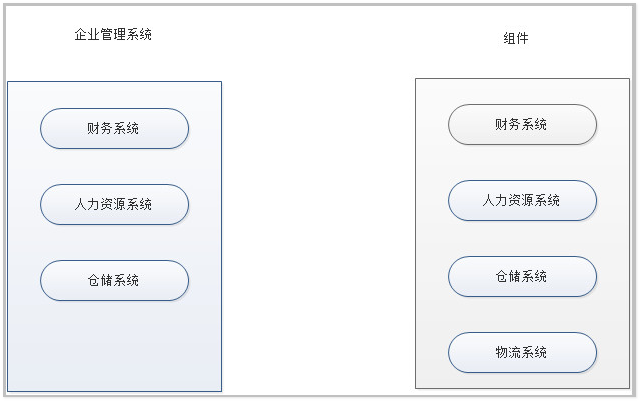
# IoC 的概念和作用

## 程序的耦合和解耦[理解]

### 什么是程序的耦合

我们在开发中，会写很多的类，而有些类之间不可避免的产生依赖关系，这种依赖关系称之为耦合。

代码中的耦合关系不利于组件式开发：



#### 概念

耦合性(Coupling)，也叫耦合度，是对模块间关联程度的度量。

耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采用以下原则： 高内聚，低耦合。

内聚与耦合：

内聚标志一个模块内各个元素彼此结合的紧密程度，它是信息隐蔽和局部化概念的自然扩展。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。

耦合是软件结构中各模块之间相互连接的一种度量，耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。 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。

内聚和耦合是密切相关的，同其他模块存在高耦合的模块意味着低内聚，而高内聚的模块意味着该模块同其他模块之间是低耦合。在进行软件设计时，应力争做到高内聚，低耦合。

#### 代码中的体现

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？

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。

/\*\*

\* 程序的耦合

\* 耦合：程序间的依赖关系

\* 包括：

\* 类之间的依赖

\* 方法间的依赖

\* 解耦： 降低程序间的依赖关系

\* 实际开发中：

\* 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

\*/

**public** **class** JdbcDemo {

**public** **static** **void** main(String[] args){

Connection con = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

**try** {

// 1.加载数据库驱动

Class.*forName*("com.mysql.jdbc.Driver");

// 2.建立数据库连接

// jdbc:mysql:连接mysql数据库的固定写法，localhost：ip地址，3306：端口号，jdbcDemo：数据库名称

String url = "jdbc:mysql://localhost:3306/heima25";

// 第一个参数：连接mysql数据库,第二和第三个参数分别是数据库的用户名和密码

con = DriverManager.*getConnection*(url, "root", "root");

// 3.获取statement对象

String sql = "select \* from cst\_customer";

ps = con.prepareStatement(sql);

// 4.执行sql语句，并且获取结果

rs = ps.executeQuery();

// 5.处理结果

**while** (rs.next()) {

System.***out***.println(rs.getString("cust\_name"));

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally**{

// 6.释放资源

**try** {

**if** (rs!=**null**) {

rs.close();

}

} **catch** (Exception e) {

e.printStackTrace();

}

**try** {

**if** (ps!=**null**) {

ps.close();

}

} **catch** (Exception e) {

e.printStackTrace();

}

**try** {

**if** (con!=**null**) {

con.close();

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

}

### 解决程序耦合的思路

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。

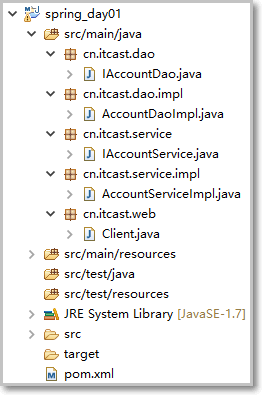
### 工厂模式解耦

#### 思路

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。

#### 代码准备



dao接口和实现类：

dao接口:

**public** **interface** IAccountDao {

/\*\*

\* 模拟保存账户

\*/

**void** saveAccount();

}

dao实现类：

**public** **class** AccountDaoImpl **implements** IAccountDao {

**public** **void** saveAccount() {

System.***out***.println("保存了账户");

}

}

service接口和实现类：

service接口：

**public** **interface** IAccountService {

/\*\*

\* 模拟保存账户

\*/

**void** saveAccount();

}

service实现类：

**public** **class** AccountServiceImpl **implements** IAccountService {

//手动初始化

**private** IAccountDao accountDao = **new** AccountDaoImpl();

**public** **void** saveAccount() {

accountDao.saveAccount();

}

}

Web层：模拟一个表现层

**public** **class** Client {

**public** **static** **void** main(String[] args) {

IAccountService as = **new** AccountServiceImpl();

as.saveAccount();

}

}

#### ※代码解耦

在上述的代码中，代码耦合体现在两个地方：

1. service中手动初始化了dao
2. web层手动初始化了service

以上两个地方，缺少任意一个实现类，编译就会报错。

而解耦最明显的特征就是：哪怕缺少一个对象，起码在编译时不会报错，只有在运行时才会报错。

既然在jdbc中我们可以使用反射来进行解耦，那么我们能否在三层中使用这个方案呢？

第一步：创建工厂

在resources目录下创建资源文件，bean.properties:

accountDao=cn.itcast.dao.impl.AccountDaoImpl

accountService=cn.itcast.service.impl.AccountServiceImpl

编写一个工厂:用来反射生成需要的对象

/\*\*

\* 一个创建 Bean 对象的工厂

\* 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 资源文件加载对象

**private** **static** ResourceBundle *rb*;

// 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** {

// 加载外部资源文件

*rb* = ResourceBundle.*getBundle*("bean");

//获取资源文件中所有的key

Enumeration<String> keys = *rb*.getKeys();

// 实例化容器

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

// 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 = *rb*.getString(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);

}

}

第二步:代码解耦

Web层：

/\*\*

\* 模拟一个表现层

\* **@author** zhang

\*

\*/

**public** **class** Client {

**public** **static** **void** main(String[] args) {

//IAccountService as = new AccountServiceImpl();

//反射生成service

IAccountService as = (IAccountService) BeanFactory.*getBean*("accountService");

as.saveAccount();

}

}

Service层：

**public** **class** AccountServiceImpl **implements** IAccountService {

//手动初始化

//private IAccountDao accountDao = new AccountDaoImpl();

//反射生成dao

**private** IAccountDao accountDao = (IAccountDao) BeanFactory.*getBean*("accountDao");

**public** **void** saveAccount() {

accountDao.saveAccount();

}

}

### 控制反转-Inversion Of Control

通过上面的代码我们可以看出： 绿色被注释上的部分就是我们程序中的耦合体现。而下面的红色代码通过 BeanFactory 中 getBean方法获取对象就解决了我们代码中对具体实现类的依赖。

我们通过使用工厂模式，实现了表现层——业务层以及业务层——持久层的解耦。

它的核心思想就是：

1、通过读取配置文件反射创建对象。

2、把创建出来的对象都存起来，当我们下次使用时可以直接从存储的位置获取。

这里面要解释两个问题：

第一个：存哪去？

分析：由于我们是很多对象，肯定要找个集合来存。这时候有 Map 和 List 供选择。

到底选 Map 还是 List 就看我们有没有查找需求。有查找需求，选 Map。

所以我们的答案就是在应用加载时，创建一个 Map，用于存放三层对象。我们把这个 map 称之为容器。

第二个：什么是工厂？

工厂就是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。

第三个：什么是控制反转（Inversion Of Control）

原来：

我们在获取对象时，都是采用 new 的方式。是主动的。

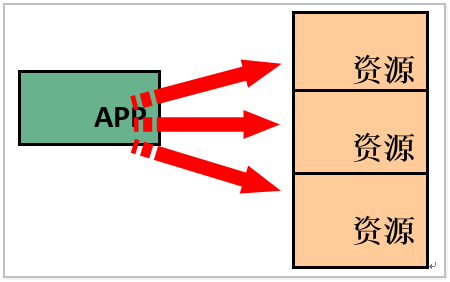
现在由工厂提供，控制权发生了改变。

由主动初始化到被动由工厂创建的这个过程，叫做控制反转（即IOC）：工厂+反射+配置文件



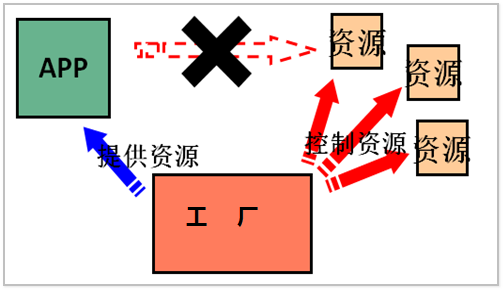
原来：

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，同时跟工厂要，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获取对象的思想就是控制反转，它是spring框架的核心之一。

它的作用只有一个：削减计算机程序的耦合。

那spring具体是怎么操作的呢？

同样有工厂类，同样有配置文件，同样有控制反转的过程。

# 使用 spring的 IOC解决程序耦合

## 案例的前期准备[会用]

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。

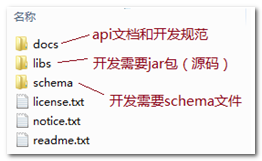
在开始 spring 的配置之前，我们要先准备一下环境。由于我们是使用 spring 解决依赖关系，并不是真正的要做增删改查操作，所以此时我们没必要写实体类。并且我们在此处使用的是 java 工程，不是 java web 工程。

官网：http://spring.io/

下载地址： <http://repo.springsource.org/libs-release-local/org/springframework/spring>

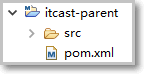
我们上课使用的版本是spring4.3.13

解压:(Spring 目录结构:) ：



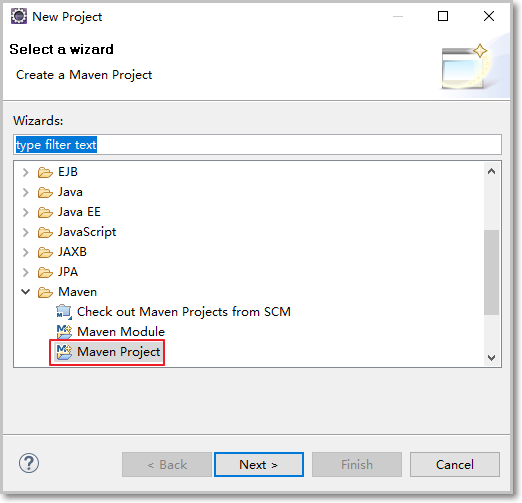
### 步骤一:导入父工程

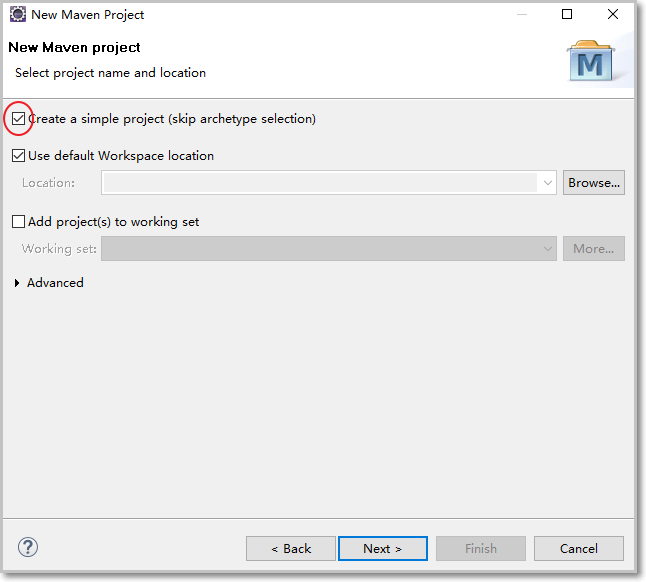
为了引入依赖并且管理依赖版本方便，可以选择使用父工程，父工程在资料中有，直接导入即可。

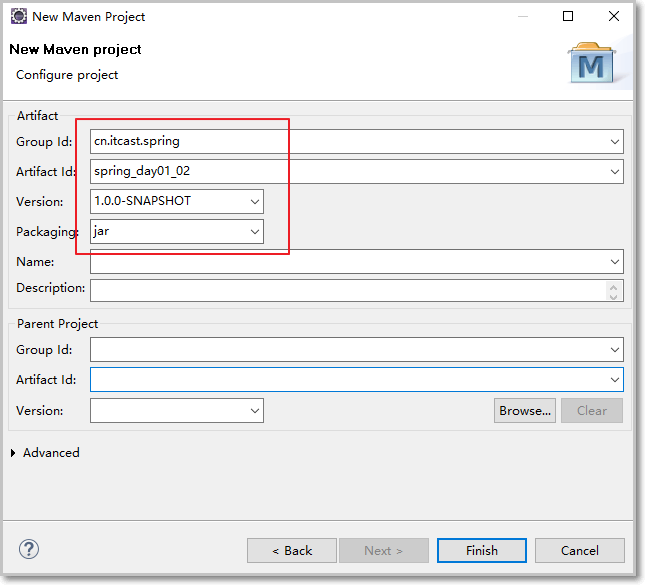


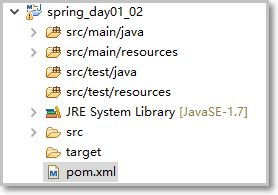
### 步骤二：创建工程导入依赖

使用eclipse创建工程：









pom.xml如下:依赖父工程并添加spring-context的依赖（该依赖受父工程管理）

<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.itcast.spring</groupId>

<artifactId>spring\_day01\_02</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<!-- spring核心依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

</dependencies>

</project>

### 步骤三：创建业务层接口和实现类

接口:

**public** **interface** IAccountService {

/\*\*

\* 模拟保存账户

\*/

**void** saveAccount();

}

实现类：

**public** **class** AccountServiceImpl **implements** IAccountService {

//手动初始化

**private** IAccountDao accountDao = **new** AccountDaoImpl();

**public** **void** saveAccount() {

accountDao.saveAccount();

}

}

### 步骤四：创建持久层接口和实现类

接口:

**public** **interface** IAccountDao {

/\*\*

\* 模拟保存账户

\*/

**void** saveAccount();

}

实现类:

**public** **class** AccountDaoImpl **implements** IAccountDao {

**public** **void** saveAccount() {

System.***out***.println("保存了账户");

}

}

### 步骤五：创建表现层

**public** **class** Client {

**public** **static** **void** main(String[] args) {

IAccountService as = **new** AccountServiceImpl();

as.saveAccount();

}

}

## Spring基于xml的IOC解耦方案

### 步骤一：创建配置文件引入约束

之前我们模拟的控制反转的实现过程：工厂+反射+配置文件

IoC底层实现：工厂（设计模式）+反射（机制） + 配置文件

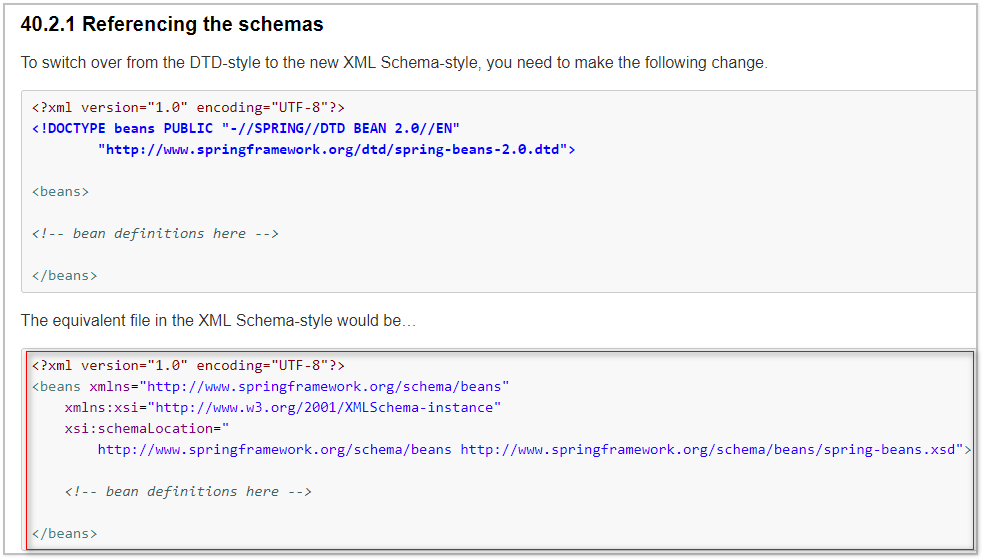
在spring中，工厂和反射机制由框架完成，我们只需要编写配置文件即可。

在src下创建：applicationContext.xml



配置约束,位置：资料\spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html

建议将这个页面添加到收藏，方便以后查看。



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=*"*

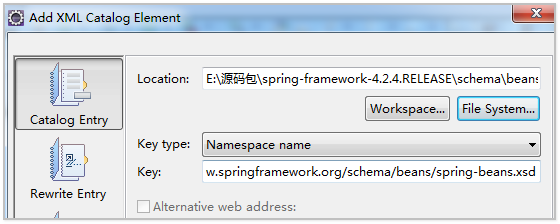
*http://www.springframework.org/schema/beans*

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

</beans>

问题：发现引入规范约束后，没有关联对应的配置，原因是需要配置约束：配置本地约束或者联网即可。

配置本地提示：点击Myeclipse属性——>选择XML Catalog



### 步骤二：导入依赖

<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.itcast.spring</groupId>

<artifactId>spring\_day02</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

</dependencies>

</project>

### 步骤三：将dao和service的实现类装配到容器中

<?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"*>

<!-- spring 将自己管理的资源统称为bean

id:资源标识名称，相当于properties中的key，通过该名称可以获取对象

class:资源路径,一般是实现类的全路径

-->

<!-- 将service实现类装配到容器中 -->

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*></bean>

<!-- 将dao实现类装配到容器中 -->

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*></bean>

</beans>

### 步骤四：从容器中获取dao和service实现类

将web层代码修改如下：

**public** **class** Client {

**public** **static** **void** main(String[] args) {

//加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

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

//从容器中获取service

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

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

//从容器中获取dao

IAccountDao accountDao = (IAccountDao) ac.getBean("accountDao");

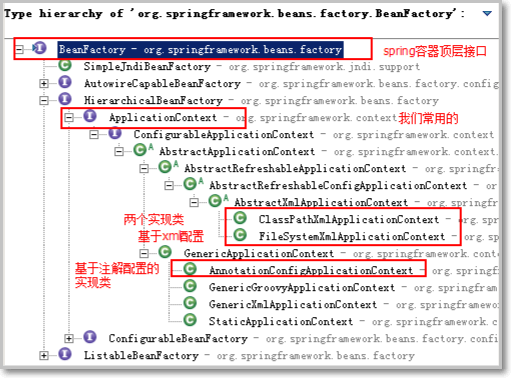
System.***out***.println(accountDao);

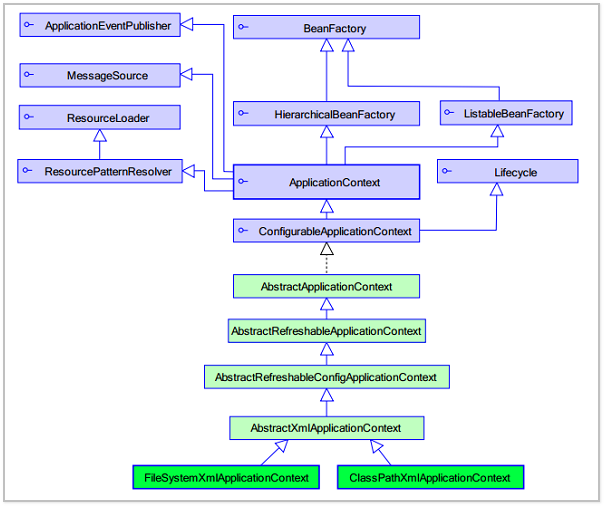
}

}

## Spring基于 XML 的 IOC 细节[掌握]

### spring 中工厂的类结构图





#### BeanFactory和 ApplicationContext 的区别

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：只要一读取配置文件，默认情况下就会创建对象。

BeanFactory：什么使用什么时候创建对象。

Beanfactory的用法：

BeanFactory ac = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));

#### 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： -了解

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。

### IOC 中 bean 标签和管理对象细节

#### 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 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(生命周期相关)。
* destroy-method：指定类中销毁方法名称(生命周期相关)。

#### bean的作用域

scope：指定对象的作用范围。

\* singleton :默认值，单例的.

\* prototype :多例的.

##### 步骤一：创建ScopeBean



内容如下;

**public** **class** ScopeBean {

}

##### 步骤二：装配ScopeBean

<!--装配scopeBean-->

<bean id=*"scopeBean"* class=*"cn.itcast.bean.ScopeBean"* scope=*"prototype"*></bean>

##### 步骤三：创建ClientScope

**public** **class** ClientScope {

**public** **static** **void** main(String[] args) {

//加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

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

//从容器中获取service

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

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

IAccountService accountService2 = (IAccountService) ac.getBean("accountService");

System.***out***.println(accountService2);

}

}

##### 步骤四：设置多例

将bean标签中的scope属性设置为prototype:

<!-- 将service实现类装配到容器中 -->

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"* scope=*"prototype"*></bean>

#### bean 的生命周期

单例对象：scope="singleton"

一个应用只有一个对象的实例。它的作用范围就是整个应用。

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（getBean）。

对象活着：只要对象在使用中，就一直活着。

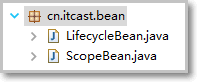
对象死亡：当对象长时间不用时，被java的垃圾回收器回收了。

生命周期方法：

init-method：指定类中的初始化方法名称(生命周期相关)。

destroy-method：指定类中销毁方法名称(生命周期相关)。

##### 步骤一：创建LifecycleBean



内容如下:

**public** **class** LifecycleBean {

**public** LifecycleBean() {

System.***out***.println("lifecycleBean被初始化了");

}

**public** **void** start(){

System.***out***.println("初始化方法执行了");

}

**public** **void** end(){

System.***out***.println("销毁方法执行了");

}

}

##### 步骤二：装配LifecycleBean

<!--装配lifecycleBean-->

<bean id=*"lifecycleBean"* class=*"cn.itcast.bean.LifecycleBean"* init-method=*"start"* destroy-method=*"end"*></bean>

##### 步骤三：创建ClientLifecycle



**public** **class** ClientLifeCycle {

**public** **static** **void** main(String[] args) {

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

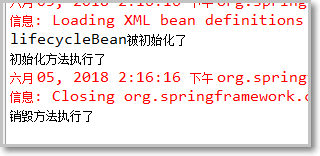
LifecycleBean lifecycleBean = (LifecycleBean) ac.getBean("lifecycleBean");

ac.close();

}

}

生命周期方法执行了：

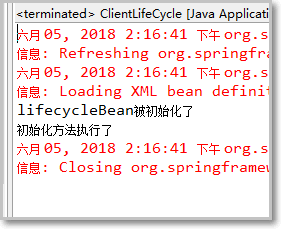


##### ※步骤四：修改配置文件

<!--装配lifecycleBean-->

<bean id=*"lifecycleBean"* class=*"cn.itcast.bean.LifecycleBean"* init-method=*"start"* destroy-method=*"end"* scope=*"prototype"*></bean>

再次执行main方法发现,LifecycleBean被延迟加载了，并且只执行了初始化方法-start,没有执行销毁方法end.



#### 实例化 Bean 的三种方式

第一种方式：使用默认无参构造函数创建对象

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

第二种方式：使用静态工厂的方法创建对象

使用工厂中的静态方法创建对象，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

第三种方式：使用实例工厂的方法创建对象

先把工厂的创建交给 spring 来管理。

然后再使用工厂的 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。

##### 默认无参构造创建对象

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*></bean>

##### 静态工厂方法创建对象

第一步：创建静态工厂

/\*\*

\* 假设service的实现类是由工厂中的静态方法创建的

\* **@author** zhang

\*

\*/

**public** **class** StaticFactory {

**public** **static** IAccountService createAccountService(){

**return** **new** AccountServiceImpl();

}

}

第二步：编写配置文件

将之前的service的bean注释掉重新装配一个即可

<!-- 通过静态工厂方法创建对象并装配到容器中 -->

<bean id=*"accountService"* class=*"cn.itcast.utils.StaticFactory"* factory-method=*"createAccountService"*></bean>

第三步：测试

**public** **class** ClientStatic {

**public** **static** **void** main(String[] args) {

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

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

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

}

}

##### 实例工厂方法创建对象

第一步：创建实例工厂

/\*\*

\* 假设service的实现类是由一个工厂的非静态方法创建的

\* **@author** zhang

\*

\*/

**public** **class** InstanceFactory {

**public** IAccountService createAccountService(){

**return** **new** AccountServiceImpl();

}

}

第二步：编写配置文件

将之前的service的bean注释掉重新装配一个即可

<!-- 将实例工厂装配到容器中 -->

<bean id=*"instanceFactory"* class=*"cn.itcast.utils.InstanceFactory"*></bean>

<!-- 调用实例工厂中的非静态方法创建对象 -->

<bean id=*"accountService"* factory-bean=*"instanceFactory"* factory-method=*"createAccountService"*></bean>

第三步：测试

**public** **class** ClientInstance {

**public** **static** **void** main(String[] args) {

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

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

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

}

}

### Spring的依赖注入

依赖注入的概念

依赖注入：Dependency Injection。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。

#### 构造方法注入属性

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硬编码的，而是通过配置的方式，让spring框架来为我们注入。

要求：

1. bean对象需要创建有参数的构造方法
2. 在配置文件中通过constructor-arg标签注入属性

##### 步骤一：创建AccountServiceImpl2

创建一个AccountServiceImpl2并添加有参构造函数

**public** **class** AccountServiceImpl2 **implements** IAccountService {

**private** String name;

**private** Integer age;

**private** IAccountDao accountDao;

//使用有参构造来注入属性值

**public** AccountServiceImpl2(String name, Integer age, IAccountDao accountDao) {

**super**();

**this**.name = name;

**this**.age = age;

**this**.accountDao = accountDao;

}

**public** **void** saveAccount() {

System.***out***.println(name+"..."+age+"..."+accountDao);

}

}

##### 步骤二：编写配置文件

<!--

使用构造函数注入属性值：

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

=======上面三个属性都是指给谁赋值，下面两个属性指的是赋什么值的==============

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=*"accountService2"* class=*"cn.itcast.service.impl.AccountServiceImpl2"*>

<constructor-arg name=*"name"* value=*"张三"*></constructor-arg>

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

<constructor-arg name=*"accountDao"* ref=*"accountDao"*></constructor-arg>

</bean>

##### ※步骤三：创建Client2测试

创建一个Client2：

**public** **class** Client2 {

**public** **static** **void** main(String[] args) {

//加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

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

//从容器中获取service

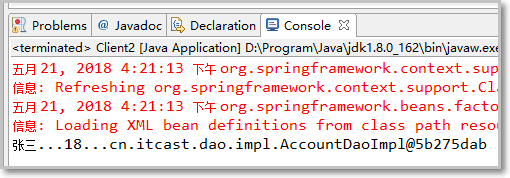
IAccountService accountService = (IAccountService) ac.getBean("accountService2");

accountService.saveAccount();

}

}

打印效果:发现属性值已经注入到service实现类中。



#### Set方法注入

顾名思义，就是使用类中的set方法，给成员变量赋值。注意，赋值的操作不是我们硬编码的，而是通过配置的方式，让spring框架来为我们注入。

要求：

1. 在配置文件中通过proprety标签注入属性

##### 步骤一：创建AccountServiceImpl3

创建AccountServiceImpl3并提供属性的set方法

**public** **class** AccountServiceImpl3 **implements** IAccountService {

**private** String name;

**private** Integer age;

**private** IAccountDao accountDao;

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

**this**.name = name;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

**public** **void** setAccountDao(IAccountDao accountDao) {

**this**.accountDao = accountDao;

}

**public** **void** saveAccount() {

System.***out***.println(name+"..."+age+"..."+accountDao);

}

}

##### 步骤二：编写配置文件

<!--

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相关属性：

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=*"accountService3"* class=*"cn.itcast.service.impl.AccountServiceImpl3"*>

<property name=*"name"* value=*"李四"*></property>

<property name=*"age"* value=*"20"*></property>

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

</bean>

##### 步骤三：创建Client3测试

**public** **class** Client3 {

**public** **static** **void** main(String[] args) {

//加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

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

//从容器中获取service

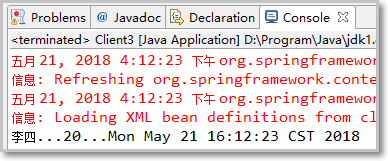
IAccountService accountService = (IAccountService) ac.getBean("accountService3");

accountService.saveAccount();

}

}

打印：



#### P名称空间注入

此种方式是通过在 xml中导入 p名称空间，使用 p:propertyName属性 来注入数据，它的本质仍然是调用类中的 set 方法实现注入功能。

##### 步骤一：创建AccountServiceImpl4

**public** **class** AccountServiceImpl4 **implements** IAccountService {

**private** String name;

**private** Integer age;

**private** IAccountDao accountDao;

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

**this**.name = name;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

**public** **void** setAccountDao(IAccountDao accountDao) {

**this**.accountDao = accountDao;

}

**public** **void** saveAccount() {

System.***out***.println(name+"..."+age+"..."+accountDao);

}

}

##### 步骤二：编写配置文件

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

配置文件：

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

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

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

<!-- 将dao实现类装配到容器中 -->

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*></bean>

<!--

本质上还是通过set方法注入属性值

涉及的属性：

p:属性名称:直接注入属性值

p:属性名称-ref：引用其他bean类型的属性进行注入

-->

<bean id=*"accountService4"* class=*"cn.itcast.service.impl.AccountServiceImpl4"* p:name=*"王五"* p:age=*"22"* p:accountDao-ref=*"accountDao"*></bean>

</beans>

##### 步骤三：创建Client4测试

**public** **class** Client4 {

**public** **static** **void** main(String[] args) {

//加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

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

//从容器中获取service

IAccountService accountService = (IAccountService) ac.getBean("accountService4");

accountService.saveAccount();

}

}

#### 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。

##### 步骤一：创建AccountServiceImpl5

**public** **class** AccountServiceImpl5 **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;

}

**public** **void** saveAccount() {

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

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

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

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

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

}

}

##### 步骤二：编写配置文件

<!--

注入集合属性：

使用set方法注入集合属性：

array:一般用来设置数组

list:一般用来设置list集合

map:一般用来设置map集合

props:一般用来设置properties

-->

<bean id=*"accountService5"* class=*"cn.itcast.service.impl.AccountServiceImpl5"*>

<property name=*"myStrs"*>

<array>

<value>AAA</value>

<value>BBB</value>

<value>CCC</value>

</array>

</property>

<property name=*"myList"*>

<list>

<value>AAA</value>

<value>BBB</value>

<value>CCC</value>

</list>

</property>

<property name=*"mySet"*>

<set>

<value>AAA</value>

<value>BBB</value>

<value>CCC</value>

</set>

</property>

<property name=*"myMap"*>

<map>

<entry key=*"name1"* value=*"AAA"*></entry>

<entry key=*"name2"* value=*"BBB"*></entry>

<entry key=*"name3"* value=*"CCC"*></entry>

</map>

</property>

<property name=*"myProps"*>

<props>

<prop key=*"name1"*>AAA</prop>

<prop key=*"name2"*>BBB</prop>

<prop key=*"name3"*>CCC</prop>

</props>

</property>

</bean>

##### 步骤三：创建Client5测试

**public** **class** Client5 {

**public** **static** **void** main(String[] args) {

//加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

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

//从容器中获取service

IAccountService accountService = (IAccountService) ac.getBean("accountService5");

accountService.saveAccount();

}

}

# 作业

## 需求

实现账户表的增删改查操作

数据库表：

CREATE TABLE account (

id INT PRIMARY KEY auto\_increment,

name VARCHAR (40),

money FLOAT

) CHARACTER

SET utf8 COLLATE utf8\_general\_ci;

INSERT INTO account (NAME, money) VALUES ('aaa', 1000);

INSERT INTO account (NAME, money) VALUES ('bbb', 1000);

INSERT INTO account (NAME, money) VALUES ('ccc', 1000);

## 要求

1.独立编写实体类，业务层和持久层代码。（此处不涉及表现层，创建 maven 的 jar 工程即可）

2.持久层技术采用 dbutils 实现。（不要用 Mybatis）

3.连接池（数据源）dbcp 或者 c3p0 均可。

4.要求使用 junit 测试代码。

5.运用今天所学的 spring基于 xml的 ioc实现业务层和持久层解耦。